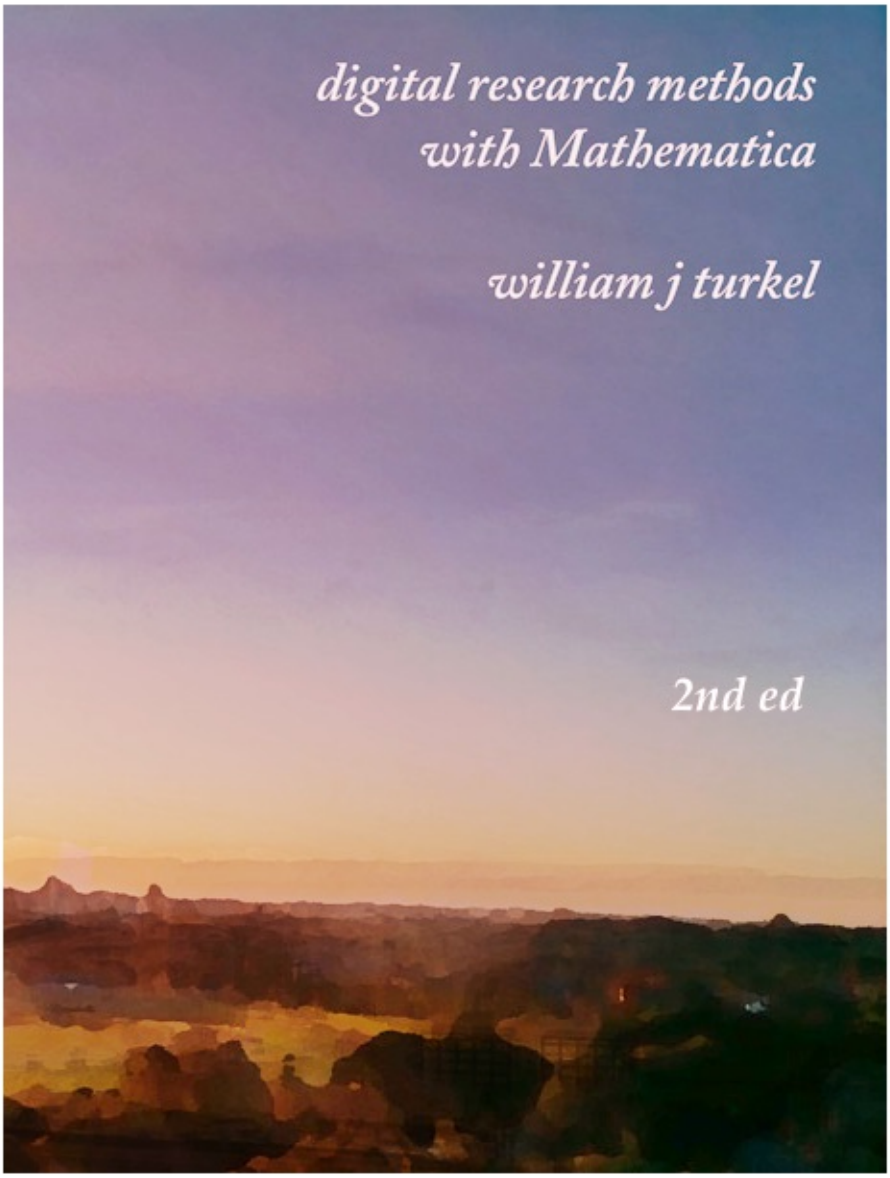


Digital Research Methods with *Mathematica*, 2nd Ed

William J. Turkel
Professor of History

University of Western Ontario
william.j.turkel@gmail.com

v2.0
Summer 2019

This is a *Mathematica* notebook

If you have never used one before, you can expand and collapse sections clicking on the cell group opener above. When a section is closed, it looks like this » and when a section is open the two arrows point downwards.

Each cell has its own bracket, which you can see on the right. Brackets are nested into larger groupings.

The cell beneath this one contains code. You can *evaluate* it by selecting its cell bracket and pressing SHIFT↵

```
2 + 2
```

Table of Contents

You can click on a heading to jump directly to that chapter.

- Lesson 12. **Linked Open Data**. Resource description framework (RDF), SPARQL queries and endpoints, JSON-LD.
- Lesson 13. **Markup Languages**. Scraping and parsing, XML, really simple syndication (RSS) and text encoding initiative (TEI).
- Lesson 14. **Studying Societies**. Computational social science, search data, social media and social networks.
- Lesson 15. **Extracting Keywords**. Information retrieval, term frequency-inverse document frequency (TF-IDF) and rapid automatic keyword extraction (RAKE).
- Lesson 16. **Word and Document Vectors**. Feature extraction, dimension reduction, word embeddings and global vectors.
- Lesson 17. **Citations**. References, web services, bibliographic linked open data and citation networks.
- Lesson 18. **Natural Language**. Multilingual analysis, computational linguistics and sentiment analysis.
- Lesson 19. **Web Services**. Entity networks, publication search, dashboards, manipulating JSON.
- Lesson 20. **Databases**. Parts, selections and transformations, computations and querying, relations.
- Lesson 21. **Measuring Images**. Photogrammetry, georectification, handwriting and facial 3D reconstruction.
- Lesson 22. **Machine Learning**. Unsupervised clustering, classify, predict and transfer learning.
- Project Ideas

about

About this Book

Version

This is version 2.0

This book is compatible with *Mathematica* desktop version 12.0 and later. Many of the examples should be compatible with other Wolfram Language products, but I haven't tested them so I use the term *Mathematica* to refer to the language throughout.

The latest version of this book is freely available at
http://williamjturkel.net/digital-research-methods-with-mathematica/

Code repository

There is a GitHub repository
https://github.com/williamjturkel/Digital-Research-Methods-2nd-Ed

Licenses

Cover image

The cover image is based on a photographs of Sydney taken by the Climate Change Research Centre in August 2015 and December 2016 with CC-BY licenses.

https : // www.flickr.com / photos / ccrc_weather / 20 372 731 668 /

https : // www.flickr.com / photos / ccrc_weather / 30 994 898 194 /

Feedback

Feedback is always welcome. You can email me at *william.j.turkel@gmail.com*

preface

Preface

This is the second edition of an open source, open content and open access textbook that I made available on the web in the summer of 2015. Although I had been using *Mathematica* off and on in my own work since version 3 (1996), the language had always been too expensive to consider assigning for undergraduate courses in the humanities and social sciences, especially since my university did not have a site license. The fortuitous combination of inexpensive student licenses for the software, an 'infinite archive' of born digital and digitized textual sources on the web, the growth of digital humanities, and a rapid expansion of the *Mathematica* language to include higher level building blocks for text mining, web crawling and unsupervised machine learning made the first edition of the text both possible and desirable.

Using this material, I have taught graduate and undergraduate courses in digital research methods, gradually expanding and refining the content of the first edition with a series of slides that I revised each year. With the release of *Mathematica* v12 in April 2019, however, it became clear to me that it was time for a new edition of the original text.

Entire lessons in the first edition were devoted to outcomes that can now be accomplished with a command or two. In keeping with the general development of *Mathematica* (now also known as the Wolfram Language) there is a steady trend toward ***higher level*** commands, and the second edition of this text reflects that. One of the key advantages of using *Mathematica* is that a few lines of code can accomplish something that would take hundreds or thousands of pages of code to implement in another language. This puts even relatively ambitious projects within the reach of beginners, and it

makes it possible for an experienced programmer to develop tailored and sophisticated software support for a research career by her or himself.

A related change in the second edition is that there is now much more emphasis on working with associations, datasets and other high-level ***data structures***. This is particularly important in the context of linked open data and computable knowledge, and many of the lessons use not only Wolfram entities and connections to the Wolfram Data Framework, but take advantage of the new support for SPARQL in version 12.

There are many new commands to support working with text and natural language and a greatly increased number of neural net examples. Coverage of image processing and computer vision, geospatial computation, and network analysis have all been expanded. I have also greatly extended the range of examples, to make better connections with recent literature in the digital humanities, information retrieval, and other fields.

The most significant change in this edition is support for a new approach to learning. In order to provide motivation and scaffolding for beginners, and to support learners at different levels, I emphasize the skills of ***code reading***. Learning to code is often a matter of reading and understanding the code of someone else, copying it, and then modifying it. This is especially true in *Mathematica*, where there are over 5000 commands accompanied by an extensive and truly superb system of documentation. In many ways, *Mathematica* can only be learned as a guided tour through the help files: I link to them and build on them wherever possible. Although I have put an MIT license on my own code it should be understood that in the forty years I have been programming, I have been learning from and copying other programmers every day. So the code is not really mine, and I hope you feel free to take what you find and make new things with it.

Introduction

This is a book about using computation in your research process. It shows you how you might use computational tools to better find, manage, excerpt, cluster, explore and analyze digital sources. Ideally, you should be able to put some of these techniques to work regardless of your own research area. *Mathematica* (also known as the Wolfram Language) has specific tools that can be used by humanists and scientists, journalists and doctors, lawyers and engineers... but the focus in this book is mostly on tools and processes that can be used in a wide variety of settings. These are the building blocks of scholarship. We all work with texts, images, sound and video. We all need to be able to consult the work of others and cite it in our own work.

For more than two decades, 'doing research' has mostly meant doing research online, and yet there are few textbooks that bridge the gap between the basic stuff—using search engines, evaluating the quality of sources, verifying information, keeping track of citations with bibliographic software—and the kind of computational techniques that researchers need to thrive in a networked environment with superabundant source materials. For the key point about digital sources is that they can be read and processed by machines, in vast numbers and very quickly, often autonomously. The most scarce

resource now is human care and attention. You want to be able to focus on reading, thinking and writing, and let computers do what they do best.

There are many cases where computational research methods can be far more efficient than more traditional approaches. Here are three examples from the humanities.

1. You gain temporary access to a large collection of non-digital sources that are important for your research. These might be in an archive or private collection; they might be pamphlets or books or boxes of file folders. Given the ubiquity of smartphones and digital cameras, most people would now choose to photograph the pages if possible, and researchers who do extensive archival work often return home with thousands or tens of thousands of digital photographs of various documents. You could spend the next few years going through the pictures one at a time and writing notes by hand or typing them into a word processor. Or you could write a small script to convert each printed or typescript page image into readable text and drop the whole batch into a custom search engine. In less than an hour you could be searching for words and phrases anywhere in your primary sources. The tools for processing handwritten pages are less well developed but improving rapidly enough that they might be of use in this case, too.
2. You discover a collection of hundreds or thousands of online texts that are directly related to your research. You could look through the list of titles in your web browser and click on the links one at a time, scanning each to see if it is relevant. Even if you cut-and-paste notes from the sources to a word processor, it will still take you at least a few months to go through the collection. Or you could write a small script to download all of the sources to your own machine and run a clustering program on them. This sorts the texts into batches of closely related documents, then subdivides those by topic. In less than an hour, you would be able to visualize the contents of the whole collection and focus in on the topics that are of immediate interest to you.
3. You've been working with the written corpus of a historically significant figure. You have the books and essays that he or she wrote, their diary entries and their correspondence with a large number of other individuals. How do you make sense of a lifetime of writing? Can you chart important changes in someone's conceptual world? Spot the emergence of new ideas in the discourse of a community? Map the ever-changing social relations between a network of correspondents? If you do research with contemporary sources, you may have access to millions of emails, tweets or forum entries, and wish to do similar kinds of analysis.

These examples are just the tip of the iceberg, however. *Digital Research Methods* introduces a wide variety of other powerful techniques: automatically extracting all of the images that appear in series of page images (say the run of a newspaper or journal) and classifying them into photos, drawings, charts, and so on; automatically identifying the people, places, institutions, dates, and other entities mentioned in texts; mapping and visualizing huge data sets; linking records to computable data; and many others. Computation won't magically do your research for you, but it can make you *much* more efficient. You can focus on close reading, interpretation and writing, and use machines to help you find, summarize, organize, analyze and visualize sources.

Mathematica has a number of features which make it particularly useful for doing digital research. One of the main advantages is that the notebook model allows you to mix prose, data, executable code, visualizations, simulations, interface elements, hyperlinks and other elements. The computer scientist

Donald Knuth called this 'literate programming': "Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do" (Knuth 1984). This is crucial if you are collaborating with other people, which is something that most digital research projects require. Even if you're working by yourself, however, you'll find that your own code is much easier to understand a few months later if it is surrounded with some text that explains what the heck you were thinking when you wrote it. Notebooks are enhanced by the possibility of embedding code with interfaces that can be manipulated interactively or dynamically updated.

Mathematica has a vast number of very powerful commands, which makes it possible for an individual programmer to write very short programs to perform very sophisticated tasks. The problem is usually finding the command(s) that you need for a given problem. Whenever possible, I have provided links to the documentation, and you should get in the habit of having documentation windows open on your screen and reading them continuously while you work. You can also try entering commands in plain English, as shown in the early lessons. Since every technical domain is described with a body of mathematics, another advantage to using *Mathematica* is that it already knows how to operate with the objects of these domains (e.g., matrices and graphs). And you have access to an enormous amount of computable data via the Wolfram Data Framework and Knowledgebase, *Wikipedia*, and other online sources.

Digital Research Methods is suitable for self-study or for a one-term undergraduate or graduate course. Since it is intended for as wide an audience as possible, I've tried to keep mathematical prerequisites to a minimum. Not to say that there isn't some math in the book, just that it is included in sections that are designed to encourage further or deeper exploration, and can be skimmed on a first reading. Likewise, I don't assume that you already know how to program, although I do hope that you will learn some new techniques, whatever your level of previous programming experience.

I use this book to teach both undergraduate and graduate courses in the humanities and social sciences. For both courses, I schedule two blocks per week of two hours each. In class I work through basic examples carefully, explaining how they work, answering any questions the students have, and asking them questions that I think will deepen their interest or understanding. Common programming idioms are introduced then repeatedly reused in later lessons. Rather than requiring the students to memorize anything, I focus on developing skills of *reading code*. I have been teaching programming for almost forty years now, and I think that one of the best ways to learn how to program is to do it collaboratively with other people who have more experience programming. This is true not only in the classroom, but whenever a programmer studies code written by someone else.

In addition to one core competency (reading code), I have one desired outcome for the course. I hope that students will come away with a greatly expanded sense of the possibilities and limitations of computation and computable knowledge. Often the most important lesson is simply to know that something is possible, and to know that you can figure out how to do it, with some effort and some assistance. This familiarity with algorithmic thinking also has a social benefit. At a time when so many aspects of our lives are governed by algorithm, it is crucial for every individual—every citizen—to have a sense of how computation affects their life choices, the information they are exposed to, their health and wellbeing, and so on.

For self-study or for students in more technical disciplines, this book can be combined with a guidebook to using *Mathematica* for doing math and/or a book on *Mathematica* programming (such as the excellent text by Wellin, *Essentials of Programming in Mathematica*, 2016). Some of the project ideas require more substantial background in math or computer science. In general, you will know you are ready to tackle a problem or technique when you are able to read the code. To help you get to that point, I have compiled some resources in the section below titled "Jumping off points".

A few notes for instructors

Here are a few challenges that I have run into when I have taught courses based on this book.

If you are teaching somewhere without a site license, you have to assume that some students are not going to have *Mathematica* installed for at least the first two lessons. If they do have it, you want them to be able to follow along and try things, but if they don't then you don't want them to get frustrated. So the first two lessons emphasize group discussion of programming at a leisurely pace. The lessons are also structured so that the first six cover fundamental tasks that are reused throughout the course (using notebooks and the documentation system, basic text analysis, geospatial mapping, data structures, etc.) That way students are able to go back to the lessons from the first quarter of the course to refresh their memory about how to do something.

Some students will be working with computers that are taxed by the resource intensive nature of *Mathematica*. The individual lessons are written so that each can be copied to a separate notebook and run by itself if necessary. That is to say that there are no dependencies between lessons.

Another challenge is that I often have students coming from very different backgrounds, so I need to provide a way for more advanced students not to feel bored while making the less experienced ones feel comfortable. In class I am OK with advanced students exploring the documentation system and trying things on their own, while the rest of us are more focused on the lesson at hand. I try to ask the students questions about the code at multiple levels, ranging from basic comprehension, through ways the code might be modified or extended, to more subtle questions of efficiency or style. I also try to indicate points where deeper knowledge provides a more rewarding view of the material. For example, if you don't know anything about set theory, you learn what operations like Union and Complement do to lists. If you do know something about set theory, then you know why they do the things they do.

Since most of my students are operating very far outside of their comfort zones, I deliberately make my assignments very easy. The project ideas included here, on the other hand, range greatly in difficulty and are intentionally open ended. Depending on your students, you may wish to emphasize one end of the spectrum or the other.

Jumping off points

Here are some resources to help you get started, whatever your level of previous experience.

How Tos

These provide step-by-step instructions for common tasks.

https://reference.wolfram.com/language/guide/HowToTopics.html

Wolfram U courses

There are a variety of video courses at every level.

https://www.wolfram.com/wolfram-u/

Basic programming.

https://www.wolfram.com/wolfram-u/basic-programming/

A fast introduction for experienced programmers.

https://www.wolfram.com/wolfram-u/fast-introduction-for-programmers/

An advanced programming course.

https://www.wolfram.com/wolfram-u/advanced-programming/

Tutorials

An overview of expressions.

https://reference.wolfram.com/language/tutorial/ExpressionsOverview.html

Principles of expression evaluation.

https://reference.wolfram.com/language/tutorial/PrinciplesOfEvaluation.html

Challenges

A fun way to practice and a good source of assignments for instructors.

https://challenges.wolfram.com

Stack Exchange

A good place to ask questions and a great source of code examples to practice reading.

https://mathematica.stackexchange.com

References

Knuth, Donald E. "Literate Programming," 1984.
http://www.literateprogramming.com/knuthweb.pdf

Wellin, Paul. 2016. *Essentials of Programming in Mathematica*. Cambridge: Cambridge University Press.
https://www.programmingmathematica.com/essentials-of-programming-in-mathematica.html

ch01

Lesson 01. Reading Code

What we will be doing in this course

In this course you will see many examples of using the *Mathematica* programming language to accomplish research tasks with digital or digitized sources. I present a basic way of doing something, unpack it a bit so you can see how the various pieces work, then encourage you to build on, refine and expand the approach. The focus throughout is on *reading* code. If you are a complete beginner, one of the best ways to learn to program is to copy something that sort of does what you want, and then keep modifying it until it does exactly what you want. If you already know how to program in another language, then reading and experimenting with the examples presented here will enable you to transfer your knowledge to *Mathematica*. Even if you are already an experienced *Mathematica* programmer there is still benefit to studying examples that solve problems you may not have given much thought to.

Don't try to memorize commands. You will see the most useful ones over and over and learn them gradually. Instead, have a documentation window open at all times (press F1 or use the Help → Wolfram Documentation menu option). Whenever you need to remind yourself how to do something or want to learn how to do something, search the documentation.

One of the main things that I hope that you get from the course is a greatly expanded appreciation for some of the things that one can do with computation and digital / digitized sources. In the long run, the knowledge that something is possible, and a general sense for how you might go about it, is much more important than the specific details of commands and syntax (which change over time).

Word frequency

A large part of learning how to do technical work (including programming) is learning what to pay attention to. In this first lesson, we go through a number of examples of code, and I point out things for you to notice. As you get better at reading other people's code, you will realize that your ability to write code of your own is growing. This is a much better method than sitting down and trying to memorize what commands do.

Evaluating some code

Here is a blob of code. I don't expect you to understand it (although it is OK if you do).

In[•]:=
```
TakeLargest[
 WordCounts[StringTake[ExampleData[{"Text", "AliceInWonderland"}], 1003]], 15]
```

When we evaluate the input cell containing the code, it creates an output.

Notice that the input and output cells are labelled in the left margin after you evaluate the code.

What happens if you evaluate the input cell again?

Looking at the code

What do you notice first?

What do you notice about capitalization?

How many different kinds of punctuation marks are there? What are they?

Do the punctuation marks follow any kind of pattern?

How does the output differ from the input?

What is in boldface and what is not?

Copy and modify

One way to begin to figure out what this code is doing is to copy the input cell, paste it into a new cell and make some changes before we evaluate it.

Notice the green highlighting that appears when you select part of the code. What does it tell you? Try clicking the cursor after each left square bracket, beginning with the one after TakeLargest.

Now select all of the code, copy it, and paste it into a new input cell but don't evaluate it yet.

What happens if we replace the command TakeLargest with TakeSmallest then evaluate?

What happens if we replace the number 1003 with a much smaller number then evaluate?

What happens if we replace "AliceInWonderland" with "TheBourneIdentity" then evaluate?

Notice that when we tried to replace "AliceInWonderland" we started typing "The" and the system made a suggestion. What was it?

Try replacing "AliceInWonderland" with the suggestion then evaluating the code.

The Head command

The Head command tells us what something is

In[ ]:=
```
Head[1003]
```

(An integer is a whole number that can be positive, negative or zero).

What other parts of our code could we try using this command on? Copy part of the code, paste it into a new input cell, then put the Head command around it, like this

In[ ]:=
```
Head["Text"]
```

(A string is a sequence of zero or more characters, including letters, digits, punctuation and whitespace, enclosed in a pair of quotation marks).

How many different types of things are there in the code?

What are they?

There are four different types of things in the output. What are they?

Break the code into pieces

Another good strategy for understanding code is to make sure that you understand what individual commands do.

Start with the innermost command, copy it to an input cell then evaluate it. What is the output? Spend a moment looking at it. You can delete the output by selecting its cell bracket to the right and hitting the Delete key. Do that, then try wrapping this command in Head so you can figure out what the output of the innermost command is.

Now try the same process with the following code. Note that we have moved outwards to include the command that is wrapped around ExampleData.

In[•]:=
```
StringTake[ExampleData[{"Text", "AliceInWonderland"}], 1003]
```

Now try it again with the following

In[•]:=
```
WordCounts[StringTake[ExampleData[{"Text", "AliceInWonderland"}], 1003]]
```

The outermost command is TakeLargest. What does it do?

We started this process with the innermost command (ExampleData). What would happen if we started even further in?

In[•]:=
```
{"Text", "AliceInWonderland"}
```

What type of thing is this? Use Head to test it if you are not sure.

What do you notice about the way that strings are presented in output cells?

A note on terminology. A command, like ExampleData, is also known as a function. The function expects to receive some information, known as the argument(s).

Look at the ExampleData command. What argument(s) does it receive?

Look at the StringTake command. What argument(s) does it receive?

Sorted word list

Here is another blob of code. Let's call these blobs "expressions" from now on.

In[•]:=
```
AlphabeticSort[Select[
  Keys[WordCounts[StringTake[ExampleData[{"Text", "AliceInWonderland"}], 1003]]],
  LowerCaseQ]]
```

How does the output of this expression differ from the first example?

What is being computed here?

Why do you think the word "Alice" does not appear in the output?

Notice that this expression and the first one have some code in common. What is it?

The Set and Clear commands

In *Mathematica* we can use the Set command to assign an expression to a symbol. Let's build this up in stages...

Notice the color of this input cell. What happens when you evaluate it?

In[•]:=
```
text
```

Here is an example of the Set command. Notice what happens to the color of the symbol name (i.e., *text*) when you evaluate it.

In[•]:=
```
Set[text, "This is a test"]
```

Notice the color when you type the name of the symbol in an input cell. Try evaluating it.

In[•]:=
```
text
```

If we want to get rid of our assignment, we use Clear. Note what happens to the color of the symbol name when we evaluate the following

In[•]:=
```
Clear[text]
```

Since *Mathematica* commands always begin with an uppercase letter, it is very important that we always begin our symbol names with a lowercase letter!

Extracting part of the expression into a symbol

The shorthand way to write the Set command is to use an equals sign. Here we use the same symbol but assign it a different value

In[•]:=
```
text = StringTake[ExampleData[{"Text", "AliceInWonderland"}], 1003]
```

Again notice what happens to the color of every copy of the symbol name in the notebook when you evaluate the code.

Now that we have defined the symbol *text*, we can use it to replace code in our two expressions. Note that these next two expressions give exactly the same outputs as the earlier versions.

In[•]:=
```
TakeLargest[WordCounts[text], 15]
```

In[•]:=
```
AlphabeticSort[Select[Keys[WordCounts[text]], LowerCaseQ]]
```

We can build up complex expressions by finding elements that we want to reuse and assigning them to symbols. This makes our code easier to read and clarifies the relationships between expressions.

Note that it is now obvious that both the first and second code examples are processing the same thing (i.e., the first 1003 characters of *Alice in Wonderland*) in different ways.

Make a hypothesis then test it

One habit that will serve you very well as a programmer is the following: before you execute code, make a prediction to yourself about what it is going to do. If you are right, it helps to consolidate your knowledge. If you are wrong, take some time to figure out why it did not work the way you expected it to. This habit of comparing your expectations to what is going on when you interact with the computer is a fundamental skill.

Before you execute the following command, make a prediction about what it is going to do.

In[•]:=
```
WordCounts[text]
```

What about this one?

In[•]:=
```
Keys[WordCounts[text]]
```

Compare the output of the previous two expressions. Were you right about the Keys command?

There is also a command called Values. Can you predict what the next expression will evaluate to?

In[•]:=
```
Values[WordCounts[text]]
```

One strategy for learning about commands is trying them with simple data. Look at the following expression and predict what it will do.

In[•]:=
```
Select[{"DOWN", "Alice", "beginning"}, LowerCaseQ]
```

What do you think the next expression will evaluate to?

In[•]:=
```
Select[{"DOWN", "Alice", "beginning"}, UpperCaseQ]
```

If you had to describe the action of this Select command in words, how would you describe it?

There is more than one way to do something

In most programming languages there are different ways to accomplish the same goal, and this is particularly true in *Mathematica*.

Suppose we are interested in figuring out the frequency of a particular word (say “Alice”) in the portion of *Alice in Wonderland* that we have assigned to the symbol *text.*

We know how to calculate the number of times each word appears (with WordCounts) but it would be nice to pull out the exact information we are looking for. The following expression does this

In[•]:=
```
WordCounts[text]["Alice"]
```

Make sure you understand. How many times does “pink” appear? What about “key”?

The frequency of a word is simply the number of times the word appears, divided by the total number of words in the text.

How do we know the total number of words in the text? One way to calculate that would be to extract the Values from our WordCounts association, then add them all together, like this

In[•]:=
```
Total[Values[WordCounts[text]]]
```

A different way to calculate the same value is to break the text into a list of words, then count the number of items in the list, like this

In[•]:=
```
Length[TextWords[text]]
```

Either way, the frequency of “Alice” in our text is

In[•]:=
```
4 / 192
```

Mathematica keeps fractional expressions in fractional form. If we want to see this as a decimal num-

ber, we use the N command.

In[•]:=
```
N[4 / 192]
```

In fact, *Mathematica* has a command specifically for calculating word frequencies. So here is another way we could calculate the same result

In[•]:=
```
WordFrequency[text, "Alice"]
```

Even when you already know how to program, reading code is fun because you continually discover new ways to do things.

Word clouds

The WordCloud is a very popular way of visualizing the frequencies of words in a text.

In[•]:=
```
WordCloud[text]
```

Spend some time comparing this image with the output of our first expression. What is missing?

In[•]:=
```
TakeLargest[WordCounts[text], 15]
```

When *Mathematica* creates the word cloud, it automatically removes stopwords. These are words that are so frequent in natural language that they are often removed for the purposes of indexing or visualizing a text.

We can modify the WordCloud command to include these words. In order to do this, we have to use TextWords to split the string into a list first.

In[•]:=
```
WordCloud[TextWords[text]]
```

Note that this image corresponds more closely to the output of our first expression.

In[•]:=
```
TakeLargest[WordCounts[text], 15]
```

Removing stopwords

Going in the other direction, we can also modify our WordCounts command so that it ignores stopwords.

Analysis and synthesis

So far our process of reading code has been to break down complicated expressions into smaller parts and focus on understanding those. Working in this direction (sometimes referred to as “top-down”) we are performing an analysis of the code.

It is also possible to work “bottom-up”, to build up larger expressions from simpler parts by a process of synthesis.

The TextWords command takes a string and returns a list of words in the string.

In[•]:=
```
TextWords[text]
```

The DeleteStopwords command takes a list of words and removes any stopwords. We wrap it around

the previous expression.

In[•]:=
```
DeleteStopwords[TextWords[text]]
```

We now have a list of words, but remember that the WordCounts command expects a string. So we need to convert our list of words into a string. We do this with the StringRiffle command, which joins all of the strings together, with spaces inserted between them.

In[•]:=
```
StringRiffle[DeleteStopwords[TextWords[text]]]
```

Now we can use WordCounts. Note that the stopwords do not occur in this Association.

In[•]:=
```
WordCounts[StringRiffle[DeleteStopwords[TextWords[text]]]]
```

And finally we can compare that output with the default output of WordCloud.

In[•]:=
```
WordCloud[text]
```

Most research projects involve reading a lot of text. If your texts are in machine readable form, you can use a computer to help analyze them. The most basic kind of analysis is to study the frequency with which particular words appear in the text(s).

Further Examples

Mathematica has access to vast amounts of knowledge about the real world in computable form (this will be the subject of the next lesson). Here are two ways to access historic data about words, using a relatively infrequent word that occurs in *Alice in Wonderland*.

First recorded use of a word

In[•]:=
```
WolframAlpha["hookah", {{"FirstRecordYear:WordData", 1}, "Plaintext"}]
```

Historic word frequency data

In[•]:=
```
DateListPlot[
 WordFrequencyData["hookah", "TimeSeries", {1700, Now}], PlotRange → Full]
```

Learning more

Video: Hands-on Start to Mathematica

This video walks you through the process of creating a new notebook, entering cells, performing calculations, making graphics, and so on. No prior experience with Mathematica is necessary.

https://www.wolfram.com/wolfram-u/catalog/gen005/

If you already have some programming experience...

... start with the following links.

Fast Introduction for Programmers
Wolfram Language Principles and Concepts

Notes for Programming Language Experts
Mathematica Stack Exchange

ch02

Lesson 02. Computable Knowledge

What we did in the last lesson

In the last lesson, we looked at a series of expressions and the values that were output when we evaluated those expressions. We approached the problem of understanding code as a process of close reading. This involved picking apart expressions into simpler components (top-down analysis), substituting one argument or command for another, and combining smaller expressions into more complex ones (bottom-up synthesis).

We saw examples of a wide range of different types of things. These included some of the atomic elements from which the *Mathematica* language is built (such as Integer, Symbol and String) and more complex data structures (such as List and Association). To continue the chemical metaphor, if things like integers and strings are the atoms of the language, we might think of the more complex data structures as the molecules.

The examples that we considered involved one of the most basic processes of text analysis: determining and visualizing word frequencies. We will return to and expand upon this in future lessons.

Entity lists

Clearing all defined symbols

Before we get started, we are going to Clear all symbols that have been defined previously. This will prevent something we defined in another session or another notebook from interfering with our current work. From now on, this command will appear near the top of each new lesson, before we evaluate any code.

In[•]:=
```
Clear["Global`*"]
```

An Entity list

Here is an expression that has an Entity as one of its arguments. Entities appear in your notebook in a tan-colored box with rounded edges. Evaluate it and look at the output.

In[•]:=
```
hc = EntityList[
  EntityClass["HistoricalCountry", "CurrentCountries" → Canada COUNTRY ☑]]
```

What does it appear to do?

What is the output? Confirm with the Head command.

We can extract the first element of a list with the First command and the remainder of the list with Rest, as shown in the example below.

```
In[•]:= CharacterRange["X", "Z"]
In[•]:= First[CharacterRange["X", "Z"]]
In[•]:= Rest[CharacterRange["X", "Z"]]
```

How would you use First and Head to determine what type of things are in the list assigned to the symbol *hc*? Could you use First, Rest and Head to figure out what every element of the list is?

Getting properties

You can copy and paste entities or expressions containing entities. If you want to know what kind of information is associated with a particular entity, use the following notation

```
In[•]:= New France HISTORICAL COUNTRY ["Properties"]
```

Note that the output of this expression is another list. What type of elements does it contain? Use Head and First to find out.

If we want to request a specific property for an entity, we need to know the canonical name of that property.

```
In[•]:= New France HISTORICAL COUNTRY ["PropertyCanonicalNames"]
```

What are the elements of this output list? Use Head and First to confirm.

So, to request the current countries where New France was once located, we write

```
In[•]:= New France HISTORICAL COUNTRY ["CurrentCountries"]
```

Note that we get yet another list, of entities this time.

Remember that there is usually more than one way to do something. If we can copy and paste a property entity, we can also write

```
In[•]:= New France HISTORICAL COUNTRY [ current countries ]
```

Arguments and outputs

Let me draw your attention to something that we have seen a number of times so far. A command *takes* certain types of arguments, and it *returns* certain types of outputs. It is good to get in the habit of thinking explicitly about these types. When you see

```
In[•]:= CharacterRange["X", "Z"]
```

Say to yourself “the CharacterRange command takes a pair of strings and returns a list of strings”.

Then, when you are trying to understand how some code works, you can start with the innermost command and work outwards. Looking at the next expression, you say “the CharacterRange command

takes a pair of strings and returns a list of strings".

```
In[•]:= First[CharacterRange["X", "Z"]]
```

But that means you can substitute the output of CharacterRange in place of the command itself, like this

```
In[•]:= First[{"X", "Y", "Z"}]
```

Now you can say to yourself "the First command takes a list and returns the first element, which is a string in this case". Note that lists can contain many different types of things, so we don't necessarily know what that element is going to be.

This expression is equivalent to the one above it, in terms of the output it creates. (In fact, this is a very simplified way of thinking about how *Mathematica* itself interprets the expressions you give it. Code is a kind of representation that is understandable by both humans and machines.)

Look at the following expressions and describe each in terms of the arguments it takes and the output it returns.

```
In[•]:= WordCounts["the quick brown fox jumps over the lazy dog"]
```

```
In[•]:= AlphabeticSort[{"quick", "brown", "fox"}]
```

```
In[•]:= Plus[2, 2]
```

```
In[•]:= Range[5]
```

```
In[•]:= Total[{1, 2, 3, 4, 5}]
```

Note that many commands can take different numbers or types of arguments under different conditions.

```
In[•]:= Range[5]
```

```
In[•]:= Range[7, 11]
```

```
In[•]:= Range[7, 21, 3]
```

What does the third argument do in the last example of Range?

DateObjects

How do we request the start date for New France using the canonical property name?

```
In[•]:= New France HISTORICAL COUNTRY ["StartDate"]
```

What is the output? The formatting suggests that it is special.

Try requesting the end date.

Discovering entities

What if we wanted to find all of the historical country entities that once occupied the territory where Hungary is now? We can copy the expression we started with above, but we can't cut and paste an

entity for Hungary because we don't have one in the notebook yet.

We can discover new entities by starting an input cell with a single equals sign. You will see a white equals sign in an orange box. Just type what you are looking for in English and evaluate it.

In[•]:=
```
=
```

hungary

Hungary COUNTRY

Now we can copy our expression and replace the Canada entity with the one for Hungary.

In[•]:=
```
EntityList[
 EntityClass["HistoricalCountry", "CurrentCountries" → Hungary COUNTRY ✓]]
```

We can also copy an expression, delete an entity, and type CTRL= in its place, this will allow us to type something in English at that location. Try this.

Tables

A common way to present information is in the form of a table. *Mathematica* has a command for creating tables. We will build this up gradually.

Iterators

The simplest use of the Table command makes a list. In the expression below, the symbol *i* is called an iterator. It takes on a series of values when the table is being created. Evaluate the expression. What values does *i* take on?

In[•]:=
```
Table[i, {i, 3}]
```

Suppose we wanted our Table to run from 4 to 7. How might we change the expression?

Sometimes we can figure out what the arguments to a command are likely to be, especially as we gain experience with the language. An easy alternative is to use the Information command.

In[•]:=
```
Information[Table]
```

Note that the second entry in the blue panel shows the expression we just used to generate a list of values from 1 to the maximum value of *i* (3 in our case). The third entry shows us how to generate a Table that runs from 4 to 7.

The first entry in the blue panel suggests an interesting use for Table. Try it.

Making a table from elements in a list

Another form of the Table command allows us to process each element in a list, one at a time. Here is the output of our first expression, which we assigned to the symbol *hc*.

In[•]:=
```
hc
```

To step through each element of this list with a Table, we use the following expression (I've chosen the symbol *c* to be the iterator here, but it doesn't matter what its name is, as long as it begins with a lowercase letter). Evaluate it.

In[•]:=
```
Table[c, {c, hc}]
```

That doesn't seem to have bought us anything!

But suppose we modify our Table expression so it looks like the following. Spend a moment looking at it before you evaluate it and try to predict what it is going to do.

In[•]:=
```
Table[{c, c["StartDate"]}, {c, hc}]
```

How would we modify the expression so it includes end dates also?

TableForm

These outputs are easier to read if we wrap our expression in the TableForm command

In[•]:=
```
TableForm[Table[{c, c["StartDate"], c["EndDate"]}, {c, hc}]]
```

Timelines

Missing data

In[•]:=
```
Canada HISTORICAL COUNTRY ["StartDate"]
```

In[•]:=
```
Canada HISTORICAL COUNTRY ["EndDate"]
```

Neither the contemporary country of Canada nor the historical country of the Iroquois has an end date available. Instead *Mathematica* has returned a special type of object that reflects missing data.

In[•]:=
```
Head[Missing["NotApplicable"]]
```

If we try to visualize or manipulate a set of data, the fact that we have missing elements may cause problems. For visualizations, *Mathematica* sometimes silently drops items with missing data. In most research projects, we would probably want to handle missing elements more gracefully than simply ignoring them. Here we are going to explicitly remove the items with missing dates.

Union, Intersection, Complement

Mathematica has three commands that are very useful for working with Lists. Here are some lists we will use to demonstrate

In[•]:=
```
af = CharacterRange["A", "F"]
```

In[•]:=
```
bd = CharacterRange["B", "D"]
```

In[•]:=
```
ch = CharacterRange["C", "H"]
```

In[•]:=
```
xz = CharacterRange["X", "Z"]
```

How would you describe the output of the next expression? Does it matter if we rearrange the arguments (i.e., put *ch* before *xz*)?

In[•]:=
```
Union[xz, ch]
```

Describe the output here.

In[•]:=
```
Union[ch, af]
```

Finally note this behaviour.

In[•]:=
```
Union[af, bd]
```

Note that Union is not simply joining the two lists together. Try describing the behavior of Union in words.

Here is Intersection in action.

In[•]:=
```
Intersection[af, xz]
```

In[•]:=
```
Intersection[af, bd]
```

In[•]:=
```
Intersection[af, ch]
```

Try rearranging the order of the arguments. Does it matter?

Describe the behavior of Intersection in English.

Finally, here is Complement. Note that the order of arguments does matter here!

In[•]:=
```
Complement[af, bd]
```

In[•]:=
```
Complement[bd, af]
```

In[•]:=
```
Complement[af, ch]
```

In[•]:=
```
Complement[ch, af]
```

If you are familiar with set theory, you will recognize that these commands are treating lists like sets, and Venn diagrams may help you to visualize what is going on. You don't need to know anything about set theory to use the commands to manipulate lists, however.

Removing the entities with missing data

Here is the Table of data we made, showing which historical country entities have missing end date data.

In[•]:=
```
TableForm[Table[{c, c["StartDate"], c["EndDate"]}, {c, hc}]]
```

We can use the Complement command to remove these entities from our list of historical countries

In[•]:=
```
Complement[hc, {Canada HISTORICAL COUNTRY, Iroquois HISTORICAL COUNTRY}]
```

TimelinePlot

One way to visualize relationships between dates is to place them on a timeline. If we use the TimelinePlot command with a list of historical countries that have fully specified date information, *Mathematica* will use the start and end dates to place them.

```
In[•]:= TimelinePlot[Complement[hc, {Canada HISTORICAL COUNTRY, Iroquois HISTORICAL COUNTRY}]]
```

TimelinePlot with missing data replaced

A different strategy for dealing with the missing dates would be to replace both of them with the current year (since the historical country of Canada has persisted to become the contemporary country, and the Iroquois First Nation / Native American people continue to live in Canada and the United States). If we do this, we have to dig inside the TimelinePlot command a bit to deal with formatting issues using the Labeled and Interval commands, we have to use the DateObject command to replace all dates with the corresponding year, and we have to use ReplaceAll to substitute the current year for missing data. I won't unpack the code here, but rather leave it as a slightly more complicated example for you to practice your code reading skills on.

```
In[•]:= TimelinePlot[
 ReplaceAll[Table[{Labeled[Interval[{DateObject[c["StartDate"], "Year"],
       DateObject[c["EndDate"], "Year"]}], c["Name"]]},
   {c, hc}], _Missing → CurrentDate["Year"]]]
```

The advantage of this approach is that once we have figured it out, we can use it to visualize the timelines of other historical countries. Here is a similar figure for the United States.

```
In[•]:= hcus = EntityList[
   EntityClass["HistoricalCountry", "CurrentCountries" → United States COUNTRY ☑]]
```

```
In[•]:= TimelinePlot[
 ReplaceAll[Table[{Labeled[Interval[{DateObject[c["StartDate"], "Year"],
       DateObject[c["EndDate"], "Year"]}], c["Name"]]},
   {c, hcus}], _Missing → CurrentDate["Year"]]]
```

Maps

Datasets

We can request that Mathematica return a Dataset of all of the properties of a particular entity. The dataset is a complex data structure that we will use throughout the course. Again, the formatting suggests that this output is special in some way.

```
In[•]:= New France HISTORICAL COUNTRY ["Dataset"]
```

There are only a few properties associated with a historical country.

The entity for a contemporary country like Canada is associated with much more information than the

entities for historical countries.

```
In[•]:= Canada COUNTRY ✓ ["Dataset"]
```

Mapping

Since the entity for the contemporary country of Canada has information about the extent of its borders, we can use the Polygon and GeoGraphics command to create a map.

```
In[•]:= GeoGraphics[Polygon[ Canada COUNTRY ✓ ]]
```

Maps in *Mathematica* are highly customizable. One strategy for discovering some of these options is to go to the help page for a command and simply look at some of the examples. Here is a link to the help page for GeoGraphics.

https://reference.wolfram.com/language/ref/GeoGraphics.html

Try copying and modifying the above code to change the EdgeForm and FaceForm that is used to display the Polygon.

Try changing the GeoBackground to a relief map.

Further examples

Connection to Wolfram Alpha

If you start an Input cell by typing a pair of equals signs, Mathematica will respond with an icon of a white equals sign inside an orange 'spikey'. Anything that you type at this prompt will be submitted to Wolfram Alpha, and the content returned in a formatted output.

```
In[•]:= = new france
```

Notice the little plus signs on the right hand side of the output that Wolfram Alpha returns. These enable you to paste the data, formatted graphics and/or input code intro your notebook in a form that you can modify.

For example, if I wanted to get the formatted Wikipedia page hits history, I would click the plus sign and choose "Formatted pod". The following input cell would then be pasted in my notebook.

```
In[•]:= WolframAlpha["new france",
  IncludePods → "PopularityPod:WikipediaStatsData", AppearanceElements → {"Pods"},
  TimeConstraint → {20, Automatic, Automatic, Automatic}]
```

Now that I know how to use WolframAlpha to create that formatted pod, I could copy and modify the code. Here is a similar pod for "British North America".

```
In[•]:= WolframAlpha["british north america",
  IncludePods → "PopularityPod:WikipediaStatsData", AppearanceElements → {"Pods"},
  TimeConstraint → {20, Automatic, Automatic, Automatic}]
```

Historical country borders

Using the Dated command, it is possible to plot the boundaries of historical countries at a particular point in time. Here is a map of North America circa 1800.

In[•]:=
```
GeoGraphics[Entity["GeographicRegion", "NorthAmerica"],
 GeoBackground → Dated["CountryBorders", 1800]]
```

Learning more

Video: Using Wolfram Documentation

This video provides a quick tour of the Wolfram documentation system.

https://www.wolfram.com/wolfram-u/catalog/wl002/

How To: Make a Table

The How To pages provide step-by-step instructions for performing common tasks in *Mathematica*.

https://reference.wolfram.com/language/howto/MakeATable.html

Wolfram Alpha Integration

Select and copy results in several formats

http://www.wolfram.com/mathematica/new-in-8/combine-knowledge-and-computation/select-and-copy-results-in-several-formats.html

Insert data into *Mathematica* expressions

http://www.wolfram.com/mathematica/new-in-8/combine-knowledge-and-computation/insert-data-into-mathematica-expressions.html

Programming with Wolfram Alpha data

http://www.wolfram.com/mathematica/new-in-8/combine-knowledge-and-computation/programming-with-wolframalpha-data.html

HistoricalCountry

The HistoricalCountry entities and their properties are discussed on this page. We have only scratched the surface in this lesson.

https://reference.wolfram.com/language/ref/entity/HistoricalCountry.html

ch03

Lesson 03. Text Content

Building on the first few lessons

In the first two lessons I emphasized some of the skills that you need to develop to be able to read code fluently, introducing analysis, synthesis and substitution. Our first lesson focused on text analysis at the level of strings; the second focused on computable knowledge in the form of entities. Along the way we began to see some of the wide range of options for visualizing information as word clouds, tables, timelines and maps.

Watching someone else work is one way to learn, but it is much more effective to try to do the work yourself. As you do, you will find that things that looked easy while you were watching seem much more difficult when you try to do them yourself. Learning to code is much the same as learning a natural language, in the sense that your ability to read or understand will always outstrip your ability to produce. So be patient with yourself when you find it difficult to express yourself in code. Try not to get too frustrated, and take every opportunity to practice your new skills: copy, modify, then try to create from scratch.

Before you go too far, you want to make sure you know how to do basic stuff like execute commands, collapse and expand notebook sections, and so on. There is a lot to learn, but don't worry about getting it all at once. Practice a little bit each day.

Working with *Mathematica* notebooks

Clearing all defined symbols

In[•]:=
```
Clear["Global`*"]
```

Cells and cell brackets

Mathematica notebooks are hierarchically structured documents.

They are structured in the sense that everything in a notebook is contained in one or more cells. If you look to the right of this text, for example, you will see a little bracket. That represents the extent of the cell containing this text.

They are hierarchical in the sense that cells can contain one another. Text cells, for example, can be contained in Subsection cells. Note that there is a larger bracket just to the right of the bracket for this cell. That is the Subsection cell bracket. The Subsection cell is the one above in red that contains the text "Cells and cell brackets".

There are also still larger brackets to the right of the Subsection cell bracket. These higher level brackets contain more and more of the notebook's contents.

Cursors

Try moving the mouse cursor slowly around the notebook and watch its shape. When it is a cell it is vertical and looks a little bit like a capital letter I. When it is between parts of the notebook that have text or code, it flips on to its side to become horizontal.

When the (vertical) cursor is over text or code, you can click to insert. The cursor changes to a simple blinking vertical line. Notice that this marks the insertion point for new text. Even if you move the mouse cursor away, text will be inserted at this insertion point if you start typing. The insertion point doesn't change unless you click somewhere else in a cell, or use the arrow keys to move around the current cell.

When the mouse cursor is horizontal, you are between cells. If you click at that point, you will get a long horizontal hairline that crosses the entire notebook. This allows you to insert new cells. Note that there is a little plus sign on the left side of this hairline. Click on that to choose the kind of cell to insert. The default is an input cell (i.e., a code cell) but you can also choose plain text from this drop down menu.

Selecting cells

Note that when you move your mouse cursor over a cell bracket it changes to a bar and leftward arrow. Note that the cell bracket it is over changes from grey to blue. If you click, you can select the whole cell. Once you do, the cell will look as if it has blue highlighting.

Select multiple cells by shift clicking or command clicking (OS X) / control clicking (Windows).

You can copy, paste and delete whole cells once you select them.

You can also change the formatting for a cell by selecting its cell bracket, then using the Format → Style menu option. This is one way to create Subsection and Section headings.

Shortcut keys

If you look at the menu items for Format → Style you will see a number of shortcut keys listed for the various cell types. Pressing one of these shortcut keys while you are editing a cell is a fast way to change the cell formatting.

In general, try to learn and use as many shortcut keys as you can for software that you use frequently, because it will improve your quality of life.

Merging and dividing cells

Two separate cells of the same kind (e.g., two text cells) can be selected and merged with the Cell → Merge Cells menu option.

A single cell can be divided into two cells by placing the insertion cursor where you want the split then choosing the Cell → Divide Cells menu option.

Expanding and collapsing cell groups

If you double click on a cell bracket that contains other cells, it will collapse. Note that the cell bracket itself changes. The downward pointing arrow on the bracket lets you know that you can double click on it to show its contents.

Collapsed cells also have inline cell group openers

http://www.wolfram.com/language/12/notebook-interface/inline-cell-group-openers.html?product=-mathematica

A Section or Subsection won't have these openers until you collapse it at least once.

Hyperlinks

You can add hyperlinks to your notebook by highlighting some text and choosing *Insert→Hyperlink* from the menu. When you click on a hyperlink it opens in your default web browser.

Evaluating expressions

Input and output cells

Input cells use a different font than text cells and have a different kind of cell bracket. You evaluate an input cell by clicking on the cell and hitting SHIFT↵

Try evaluating this input cell

In[•]:=
```
RandomWord[]
```

Evaluating an input cell results in output cell. Notice that once you evaluate the input cell and Mathematica returns the output cell, both are labeled in the left hand margin with a pair of little blue numbers.

You can refer to the previous output cell with the Out command

In[•]:=
```
Out[]
```

This can also be written with the shorthand notation %

In[•]:=
```
%
```

Finally, you can give Out a number as an argument to refer to a particular output cell, like Out[61] or %212

It is very important to understand this notation because it is used frequently in the *Mathematica* documentation.

It is not a good idea to use Out in your own work, however. In and out cell numbering depends on the order in which you evaluate input cells in your notebook. Go back to the RandomWord[] input cell and evaluate it again, noticing what happens to the input and output numbering. Look also at the output of the two Out expressions below it. If you jump around in your notebook, evaluating things until you get something working, none of the expressions that include Out are guaranteed to work the same way when you restart *Mathematica* and reopen the notebook. It is possible to have many notebooks open at the same time, and this increases the likelihood of creating some sequence of steps you can't recre-

ate later.

The Suggestions bar

When you select an output cell, you will see either see a rightward pointing arrow in a circle next to the cell bracket, or you will see a shaded menu bar running across the notebook beneath the output cell. This is the Suggestions bar. (When it is collapsed you get the arrow in a circle, but you can click on that to expand it.) The Suggestions bar is very useful for figuring out how to build up a sequence of commands to create a more complex expression.

For example, if we evaluate the following input cell, *Mathematica* outputs a list of three random words.

In[•]:=
```
Table[RandomWord[], {3}]
```

Try using the Suggestions bar to Sort and Reverse the list.

Note again that built-in commands always start with a capital letter.

The *Mathematica* documentation is amazing

Recall that you can get Information for a command (the shortcut for this is a single question mark before the command name).

In[•]:=
```
Information[Reverse]
```

In[•]:=
```
? RandomWord
```

The italic lowercase i in a blue circle is a link to the corresponding help file.

How would you produce a random noun or verb?

Remember, you can open the help browser at any time by pressing the F1 key or using the Help → Wolfram Documentation menu option. It is a good idea to always have one or more documentation windows open on your desktop.

You can also copy an expression from the documentation, paste it into your notebook, and evaluate it. Try that.

The documentation is also available online at

Wolfram Language & System Documentation Center

There is also a very useful list of Common “How Tos” at

How To Topics

Defining and clearing symbols

If you type a word beginning with a lowercase letter into an input cell, *Mathematica* will color it blue.

In[•]:=
```
foobar
```

This is called syntax coloring. It shows you that *Mathematica* has interpreted that word as an undefined symbol.

You use an equals sign to assign a value to a symbol. Note that when you evaluate the expression, the color of the symbol name changes. It has now been defined.

```
In[•]:= sampleNumber = RandomInteger[{1, 10}]
```

Every time you refer to this symbol, it will return the same value. This is the value that you have assigned to the symbol.

```
In[•]:= sampleNumber
```

```
In[•]:= sampleNumber
```

Note that the command that you used to create a value for your symbol has the possibility of returning a different output each time.

```
In[•]:= RandomInteger[{1, 10}]
```

```
In[•]:= RandomInteger[{1, 10}]
```

When you are finished with a symbol, you can use Clear to get rid of it. If you try to evaluate an undefined symbol, it just returns itself.

```
In[•]:= Clear[sampleNumber]
```

```
In[•]:= sampleNumber
```

Suppressing output

If you don't want to see the output of an expression when you evaluate it, you put a semicolon at the end of the line.

```
In[•]:= sampleNumber = RandomInteger[{13, 21}];
```

```
In[•]:= sampleNumber
```

```
In[•]:= Clear[sampleNumber]
```

Commenting out code

Sometimes you don't want an input cell to evaluate. You can add comment markers to code or remove them by selecting the cell and using CMD/ (OS X) or right clicking (Windows).

```
In[•]:= (*Print["By default this is commented out"]*)
```

There are a number of places in this book where I have commented out code so that it won't evaluate automatically. If you want to evaluate one of these expressions, simply remove the comment markers.

When you are developing code of your own, sometimes you temporarily include statements that output intermediate values for debugging purposes. Some programmers like to leave these statements in their code in commented out form rather than deleting them.

How to input unusual characters

Mathematica includes a complete system for typesetting mathematical expressions and other techni-

cal prose. If you need to input an unusual character, the first thing you should try is using one of the Palettes. These are available as a menu option. Try opening the Special Characters palette.

When you hover over an entry in the palette, a tooltip will show you the characters you have to type to enter the same thing from the keyboard. Try entering some symbols with the palette and then with keystroke combinations. Ones that you will need particularly often are ⟦, ⟧ and →

If you need a particular character often, it is usually worth learning the keystrokes required to create it.

Try to have one input cell per expression

People who are new to *Mathematica* often don't pay enough attention to cell bracketing, and this can make output difficult to read or cause errors that could be avoided. Try predicting the output of the following input cell before you evaluate it.

In[•]:=
```
sampleNumber = RandomInteger[{3, 4}];
Table[RandomWord[], {sampleNumber}]
sampleNumber = 8;
```

When you have a number of input rows in the same cell, use the Cell → Divide Cells menu option to give each its own. That will make it much easier to debug your code.

Evaluating units larger than a single cell

You can select a cell bracket to evaluate all of the cells within it. Try evaluating this whole subsection by selecting its cell bracket and then pressing SHIFT↵

In[•]:=
```
RandomInteger[{7, 11}]
```

In[•]:=
```
RandomWord[]
```

You can also evaluate all of the cells in a notebook with the Evaluation → Evaluate Notebook menu option. Depending what is in the notebook, this could take a long time. Personally I prefer to look at a cell before evaluating it, especially if I didn't write the code myself.

When you are working through a notebook like this one in a linear fashion, make sure to evaluate each input cell as you come to it. If you are returning to a notebook that you already worked through, then evaluating the whole notebook usually works best.

Sometimes an evaluation gets stuck. If this happens, you can usually stop it with the Evaluation → Abort Evaluation menu option.

Manipulating strings

Getting texts

Recall that when we needed a text to analyze we were able to use the ExampleData command to get one. In fact there are a number of texts in different natural languages that are available this way:

In[•]:=
```
ExampleData["Text"]
```

The Short command lets you see a small amount of a larger expression (a string in this case). The argument says to show about six lines of material. Three lines are from the beginning of the text and three are from the end, with the ellipsis (...) showing where the intervening material was not displayed.

In[•]:=
```
Short[ExampleData[{"Text", "FriendsRomansCountrymen"}], 6]
```

Typically we want to be able to use our methods on any text and throughout the course we will learn a number of different ways to get texts into *Mathematica* for analysis.

Wikipedia

Mathematica has a number of commands that allow Wikipedia to be accessed from within the notebook. This expression shows up to 10 items with “Prague” in the title.

In[•]:=
```
WikipediaSearch["Title" → "Prague", MaxItems → 10]
```

If we want to request the text of a Wikipedia article, we can do it with a command like the following.

In[•]:=
```
clockText = WikipediaData["Prague astronomical clock"];
```

For long texts it is usually a good idea to suppress output with a semicolon and assign the text to a symbol. Then you can use other commands to get a sense of what the text contains.

Working with strings

One way to analyze and manipulate text is at the level of the string, a delimited, ordered sequence of characters. *Mathematica* has hundreds of commands for string processing.

How long is our string? This command shows us the number of characters. Remember that this includes whitespace and punctuation.

In[•]:=
```
StringLength[clockText]
```

The StringTake command allows us to see a specified number of characters from the beginning. We will assign the output to a symbol for further processing.

In[•]:=
```
clockBeginString = StringTake[clockText, 308]
```

We can also use StringTake to see the last characters of the string with a negative argument. Again we will assign the output to a symbol for further processing.

In[•]:=
```
clockEndString = StringTake[clockText, -310]
```

Now suppose we want to turn a string into a list of words. We can use StringSplit, which by default uses whitespace as the separators.

In[•]:=
```
clockSplitList = StringSplit[clockBeginString]
```

What is the first element of this list?

How would you sort the list alphabetically?

A command like StringSplit is good for finding most words, but it treats punctuation oddly. To see this, let’s look at some of the elements in the list that StringSplit outputs.

The Part command

A different way to get the first element of a list is with the Part command.

In[•]:=
```
Part[clockSplitList, 1]
```

This same command can also be written like this, using double brackets.

In[•]:=
```
clockSplitList[[1]]
```

Or like this, with a special character that is equivalent to a pair of square brackets.

In[•]:=
```
clockSplitList⟦1⟧
```

To create the double square brackets special characters use shortcut keys ESC[[ESC and ESC]]ESC

If you use a pair of semicolons as shown below, you can extract a range of elements with Part.

In[•]:=
```
clockSplitList⟦15 ;; 17⟧
```

Here is the eighth element of the list

In[•]:=
```
clockSplitList⟦8⟧
```

Note that the punctuation marks have been included in the 'word', since StringSplit used whitespace to figure out where to split the string.

We can fix this by giving StringSplit a list of delimiters to use to identify where to split. In this case we are using string patterns to describe where we want the splits to take place. In English, the argument says "split the string whenever you find one or more whitespace characters or one or more punctuation characters."

In[•]:=
```
clockSplitListFixed =
 StringSplit[clockBeginString, {WhitespaceCharacter .., PunctuationCharacter ..}]
```

The matching element of this list is now correct. (Note that we had to adjust the argument we gave to Part here because different ways of splitting the string result in different numbers of elements in the output list.)

In[•]:=
```
clockSplitListFixed⟦10⟧
```

The StringCases command lets us search through a list of strings for a pattern. It returns a list containing other lists. (This is known as a nested list). Each of these is either empty (if there is no match) or it contains the matching string.

In[•]:=
```
StringCases[clockSplitListFixed, "Prague"]
```

This is a little awkward to look at, so we will use the Flatten command, which flattens out nested lists. There are three copies of the string "Prague" in the list assigned to the symbol *clockSplitListFixed*.

In[•]:=
```
Flatten[StringCases[clockSplitListFixed, "Prague"]]
```

We can use the Total command to count elements of various kinds in the list.

In[•]:=
```
Total[Flatten[StringCases[clockSplitListFixed, "Prague"]]]
```

The StringExpression command lets us search for strings matching a pattern. In this case, the pattern is

"a string beginning with the characters Pr followed by one or more characters."

In[•]:=
```
Flatten[StringCases[clockSplitListFixed, StringExpression["Pr", __]]]
```

This command is usually written in a condensed form that uses two tildes. Compare the next expression with the previous one.

In[•]:=
```
Flatten[StringCases[clockSplitListFixed, "Pr" ~~ __]]
```

Once again we can use Total to count elements.

In[•]:=
```
Total[Flatten[StringCases[clockSplitListFixed, "Pr" ~~ __]]]
```

Using the example immediately above, try to find all of the elements in the list that begin with a lowercase o, then count them.

Did that work the way you expected? String matching is often tricky precisely because the computer is completely literal. It does exactly what you tell it to do, rather than what you want it to do.

In this case we need to modify the StringExpression so that it has the WordBoundary pattern on the left edge.

In[•]:=
```
Total[Flatten[StringCases[clockSplitListFixed, WordBoundary ~~ "o" ~~ __]]]
```

We will return to string processing at various points throughout the course.

Natural language processing

TextWords and TextSentences

Part of the problem of analyzing texts by working with strings is that they are very low level. As we saw when we tried to split a string into a list of words, things like whitespace and punctuation have to be handled thoughtfully. Fortunately, *Mathematica* has higher level commands that are more sensitive to the underlying structure of natural language.

We can turn a text string into a list of word strings with the TextWords command.

In[•]:=
```
clockWords = TextWords[clockText];
```

Here is a version of the Part command that shows us the first 48 words.

In[•]:=
```
clockWords[[1 ;; 48]]
```

How would we get a list of the words numbered 49 to 66?

Here is how we get a list of the last 48 words. Note the order of arguments to Part.

In[•]:=
```
clockWords[[-48 ;; -1]]
```

Sentence boundaries are even more difficult to handle properly at the level of string processing than word boundaries. A period, for example, might mark the end of a sentence, or it might be part of an abbreviation like "Ms." or "Mr.", part of a URL or email address, and so on.

In *Mathematica*, the TextSentences command turns a text string into a list of sentence strings. Occasionally it makes mistakes, but it is pretty good.

```
In[•]:= clockSentences = TextSentences[clockText];

In[•]:= First[clockSentences]

In[•]:= clockSentences[[4]]
```

Answering questions with FindTextualAnswer

Mathematica can also use commands based on machine learning to try to answer factual questions given a text.

```
In[•]:= FindTextualAnswer[clockSentences, "When was the clock first installed?"]
```

Here is the line where *Mathematica* found this answer.

```
In[•]:= FindTextualAnswer[clockSentences,
         "When was the clock first installed?", 1, "HighlightedLine"]
```

Note that questions can be more general, like the following. Here the eight best answers have relevant portions highlighted.

```
In[•]:= FindTextualAnswer[clockSentences,
         "What renovations were made to the clock?", 8, "HighlightedSentence"]
```

TextContents

The TextContents command uses artificial intelligence to identify different kinds of entities in text. Here we ask for locations in the first sentence of the Wikipedia article. Note the probability measurements. It is most confident that Prague and the Czech Republic are locations, not so confident about some of the other results.

```
In[•]:= TextContents[First[clockSentences], "Location"]
```

One way to think about the performance of a system like this is in terms of four categories: *hit*, *miss*, *false positive* and *correct rejection*. A hit is when the system correctly identifies the phenomenon of interest, a miss is when the phenomenon is present but the system fails to identify it. A false positive is when the system thinks the phenomenon is present but it is not. A correct rejection is when the system does not think the phenomenon is present and it is not. Hits and correct rejections are good, misses and false positives are not.

The real question is whether we have more tolerance for misses or false positives. In the expression above, we correctly get locations like Prague and Czech Republic, but also get false positives like Orloj. By increasing the AcceptanceThreshold we can eliminate many of the false positives at the expense of potentially increasing the number of misses.

In the next expression we increase the AcceptanceThreshold and ask for *Mathematica* to interpret the locations.

```
In[•]:= orlojLocations = TextContents[First[clockSentences],
          {"Location", "Quantity"}, All, AcceptanceThreshold → .9]
```

Note that TextContents has returned a Dataset. We can extract and plot the locations.

In[•]:=
```
GeoListPlot[orlojLocations[[1 ;; 2, 5]], GeoRange → "Country"]
```

In fact, TextContents can extract many different types of information in addition to location.

In[•]:=
```
TextContents[clockSentences[[1]]]
```

More from Wikipedia

Wikipedia pages contain links to related pages. Here is the list for the page we have been using.

In[•]:=
```
WikipediaData["Prague astronomical clock", "SeeAlsoList"]
```

We can use Table to retrieve the Wikipedia summary page for each of these, extract the first sentence, then format in a nice display with TextGrid.

In[•]:=
```
TextGrid[Table[{First[TextSentences[WikipediaData[sa, "SummaryPlaintext"]]]},
  {sa, WikipediaData["Prague astronomical clock", "SeeAlsoList"]}], Frame → All]
```

Wolfram Language entities

Mathematica (also known more generally now as the Wolfram Language) has entities that represent its own commands, making it possible to use the same techniques that you are learning to investigate the structure and history of the programming language itself.

Here is an example of the WolframLanguageData command. If we find the TextWords command to be useful for a particular problem, these are some other commands we should investigate.

In[•]:=
```
WolframLanguageData["TextWords", "RelatedSymbols"]
```

In the next expression, some of the commands that we used in this lesson are plotted on a timeline. Notice that the list processing command (Flatten) was introduced more than a decade before the string command, that textual example data preceded a specific command for working with Wikipedia, and that lower level natural language processing commands (like TextWords and TextSentences) were followed by the introduction of higher level ones (like TextContents).

Spend a moment studying the code, to work on your code reading skills.

In[•]:=
```
TimelinePlot[WolframLanguageData[{Entity["WolframLanguageSymbol", "StringSplit"],
   Entity["WolframLanguageSymbol", "Flatten"],
   Entity["WolframLanguageSymbol", "TextWords"], Entity["WolframLanguageSymbol",
    "TextSentences"], Entity["WolframLanguageSymbol", "ExampleData"],
   Entity["WolframLanguageSymbol", "WikipediaSearch"],
   Entity["WolframLanguageSymbol", "TextContents"]}, "TimelineEvents"] ]
```

Further examples

Getting other properties from Wikipedia

The WikipediaData command can return much more information than simply the text in the article.

Here is a map of the location of the Prague astronomical clock.

```
In[•]:= GeoGraphics[WikipediaData["Prague astronomical clock", "GeoPosition"]]
```

The next expression retrieves all of the images on the page as a list of thumbnails, then makes a collage of them.

```
In[•]:= ImageCollage[WikipediaData["Prague astronomical clock", "ImageList"]]
```

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

This online text by Stephen Wolfram teaches the fundamentals of *Mathematica* through short lessons. Each has exercises and answers so you can check your work. If you work through the *Elementary Introduction* text in conjunction with this class (and do the exercises!) you will get much more out of the course.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 1-4.

DH Transformations

These are one page summaries of important techniques in the Digital Humanities.

Ben Schmidt, “Tokenization, Normalization, and Lemmatization” and “Named Entity Extraction”

Video: Using Notebooks Effectively

Notebooks are hierarchically structured documents that can contain text, graphics, code you can evaluate, custom user interfaces and much more. This video shows you how to use many of the advanced features of notebooks. No prior experience assumed.

https://www.wolfram.com/wolfram-u/catalog/wl001/

Create a presentation notebook

Notebooks can also be used to give slide show presentations. Here is an overview of some of the features.

http://www.wolfram.com/language/12/notebook-interface/present-notebooks.html?product=mathematica

How To: Use the Wolfram Language’s Syntax

The guides and tutorials linked here provide a complete overview of

https://reference.wolfram.com/language/howto/UseMathematicasSyntax.html

Text Content Types

The different types of content that can be extracted from natural language are discussed on this page. As with everything else we do in the course, we are really only scratching the surface of what is possible with *Mathematica*.

https://reference.wolfram.com/language/guide/TextContentTypes.html

Access Wikipedia Content using Wolfram Language Entities

This example is a variation of the code that we used to create a list of first sentences for Wikipedia articles related to the one we were considering. It also shows how Entities work with WikipediaData.

http://www.wolfram.com/language/11/knowledgebase-expansion/access-wikipedia-content-using-wolfram-language-en.html?product=mathematica

ch04

Lesson 04. Data Structures

Why we need data structures

Earlier I suggested that if we think of the most basic elements of *Mathematica* (like integers and strings) as the atoms of the language, then more complex data structures (like lists and associations) are the molecules. Taking the metaphor to the next level, we might imagine those molecules as the building blocks of a kind of artificial life. The directions of low and high are usually applied to these levels. The integer is a lower level element than a list. Commands that operate on integers are usually at a lower level than those that operate on lists. Note that there is no implication of the relative conceptual difficulty involved in solving problems at particular levels. There are easy problems in Chemistry and hard ones in Biology, and vice versa.

We need data structures to reflect relationships between our data. If we count the number of customers that went into a particular store one day, we might store that information in an integer. If we keep a list of daily counts, then the list both keeps related data together (i.e., each element of the list is a daily customer count) and keeps the order of the sequence (i.e., the first count is for a day that preceded the second count and so on.)

We also need data structures so we don't have to keep reinventing the wheel. If you need to keep a list of daily customer counts, you may also need to keep lists of other things. Other people will, too. So once someone implements code for manipulating lists, it makes sense to reuse that code if at all possible.

One of the main differences between *Mathematica* and many other programming languages, is that higher and higher level commands are continually added to the language. In most other programming languages, the language itself consists of a relatively small number of commands (on the order of tens or hundreds) and you have to add libraries if you want to reuse higher level code. In the lessons, I often

try to present lower level code first, and progressively higher level code after that. In the last lesson, we saw that it is possible to analyze text with strings, but that *Mathematica* also has commands that are sensitive to the properties of natural language, like TextSentences. If you need to dig into the details, you can always go down to a lower level but you don't have to.

Lists

Clearing all defined symbols

One of the reasons that we Clear all of the symbols at the beginning of a lesson is that symbols are defined across notebooks and persist until you Clear them or restart *Mathematica*. Try the following.

Note that the following symbol is not yet defined

In[•]:=
```
testSymbol
```

Create a new notebook and define a symbol in it named *testSymbol*. Note that the syntax coloring in the expression above changed. The symbol *testSymbol* is now defined in this notebook.

In[•]:=
```
testSymbol
```

Close the new notebook that you just created without saving it. The symbol is still defined

In[•]:=
```
testSymbol
```

The sequence of steps that you just took created a symbol definition that you might use for further computation in this session. When you restart Mathematica, however, your code will stop working unless you just happen to remember how you defined *testSymbol* in the first place. That is why we start with a clean slate each lesson.

In[•]:=
```
Clear["Global`*"]
```

Import

The Import command allows you to bring sources into your notebook. These can be files on your local computer or online. In general, if you can view something in a web browser or a file browser, you can import it into your *Mathematica* notebook for further analysis.

As an example, here is web page from a BBC.com site that lists events that happened on particular days of the year between 1950 and 2005. This page is an entry for February 22 that summarizes the story of the cloning of Dolly the sheep.

Rather than link to the page itself (which might be taken down or modified), I created an archival copy of it in the Wayback Machine of the Internet Archive. So we are actually looking at the page as it appeared on May 16, 2019. Open a copy in your web browser so you can compare the page on the web with the information we import and extract from it.

https://web.archive.org/web/20190516115902/http://news.bbc.co.uk/onthisday/hi/dates/stories/february/22/newsid_4245000/4245877.stm

If you are doing research with internet sources, it is a good idea to make archival copies of them. You

can do this by pasting a URL into the box labeled “Save Page Now” on this page

https://archive.org/web/

If we want to know what kinds of information we can retrieve from the page, we use the Elements property of Import, like this

```
In[•]:= dollyURL =
          "https://web.archive.org/web/20190516115902/http://news.bbc.co.uk/onthisday/hi/
            dates/stories/february/22/newsid_4245000/4245877.stm";

In[•]:= dollyElements = Import[dollyURL, "Elements"]
```

We are going to Import a number of these different web page elements and assign the output to symbols for further processing. Some of them may be large, so we suppress output. Note that I put each expression in its own cell.

```
In[•]:= dollyData = Import[dollyURL, "Data"];

In[•]:= dollyFullData = Import[dollyURL, "FullData"];

In[•]:= dollyHyperlinks = Import[dollyURL, "Hyperlinks"];

In[•]:= dollyPlaintext = Import[dollyURL, "Plaintext"];

In[•]:= dollySource = Import[dollyURL, "Source"];

In[•]:= dollyTitle = Import[dollyURL, "Title"];
```

Characterizing the imported data

Let’s start with the title information. We discover it is a short string.

```
In[•]:= Head[dollyTitle]

In[•]:= StringLength[dollyTitle]

In[•]:= dollyTitle
```

Let’s look at the plaintext information next.

```
In[•]:= Head[dollyPlaintext]

In[•]:= StringLength[dollyPlaintext]
```

It is a longer string, so we use StringTake to look at part of it.

```
In[•]:= StringTake[dollyPlaintext, 150]
```

Repeat this process for the source information. What does this web page element contain?

Length

The data, full data and hyperlinks web page elements are returned as lists.

```
In[•]:= Head[dollyData]

In[•]:= Head[dollyFullData]
```

In[•]:= **Head[dollyHyperlinks]**

The Length command tells us how many elements are at the top level of a list. Note that since lists can contain other lists (they can be nested), we don't actually know how many further elements might be lurking within.

In[•]:= **Length[dollyData]**

In[•]:= **Length[dollyFullData]**

In[•]:= **Length[dollyHyperlinks]**

Parts of lists

As we've seen, we can use Part to return an element from a list

In[•]:= **dollyHyperlinks[[34]]**

Here is the first element of the list containing web page data. What other command could we use to retrieve the same list element?

In[•]:= **dollyData[[1]]**

Note that this element is also a list. So *dollyData* is a nested list.

In[•]:= **Head[dollyData[[1]]]**

If we use the Short command, we can see a bit more of this list.

In[•]:= **Short[dollyData, 6]**

Let's have a look at the whole list. In general, you only want to evaluate an expression like this if you know it is not going to fill up your notebook with a huge output cell. (But if it does, you can always delete the output.)

In[•]:= **dollyData**

Suppose we want to pull out the sublist containing the "Watch/Listen" information. One way to do this is with a series of Part commands.

In[•]:= **dollyData[[6]]**

In[•]:= **dollyData[[6]][[2]]**

In[•]:= **dollyData[[6]][[2]][[1]]**

This can be written more compactly as

In[•]:= **dollyData[[6, 2, 1]]**

How much structure do you need for scraping?

One of the most common tasks in working with online data is known as *web scraping*. You find a website that contains information you need for your research, then write expressions to extract that information into your notebook for further analysis. We will see examples of scraping throughout the course.

When you are extracting information from a web page, the Import command gives you a number of different options for elements to retrieve. Each of these options is suitable for some research tasks and not for others.

If you are characterizing a whole web site (building a site directory, for example) having just the title of each page on the site may be sufficient.

If you are creating a network diagram showing which pages link to which other pages, the list of hyperlinks on each page will be crucial. We will see examples of this later in the course.

If you want to analyse the frequency of terms used on the page, the plaintext element is most useful.

In[•]:=
```
WordCloud[dollyPlaintext]
```

In[•]:=
```
TakeLargest[WordCounts[DeleteStopwords[dollyPlaintext]], 10]
```

Sometimes, however, you can only identify the exact information you want on the page by referring to the HTML source. We will see examples of this later.

Associations

Word frequencies

The WordCounts command returns an Association as its output.

In[•]:=
```
dollyWordCounts = WordCounts[dollyPlaintext];
```

In[•]:=
```
Head[dollyWordCounts]
```

An association is a data structure that contains pairs of elements. Here are the ten most frequently occurring words on the webpage we have been analyzing.

In[•]:=
```
TakeLargest[dollyWordCounts, 10]
```

The first element of each pair is known as the Key and the second as the Value. You can use a Key to look up the corresponding Value.

In[•]:=
```
dollyWordCounts["Dolly"]
```

In[•]:=
```
dollyWordCounts["announced"]
```

The next expression shows us how many key-value pairs are in the association.

In[•]:=
```
Length[dollyWordCounts]
```

The Keys command returns a list of all of the keys in our association. There are a lot of them, so we will use Short to see a couple of lines worth.

In[•]:=
```
Short[Keys[dollyWordCounts], 4]
```

If we try to retrieve a Key that doesn't exist we get a Missing error.

In[•]:=
```
dollyWordCounts["lamb"]
```

We can test to see if a key is present in an association with an expression like the following. Note that

commands whose names end in a capital letter Q return a True or False.

```
In[•]:= KeyMemberQ[dollyWordCounts, "Dolly"]
```

```
In[•]:= KeyMemberQ[dollyWordCounts, "lamb"]
```

The Values command returns a list of all of the Values.

```
In[•]:= Short[Values[dollyWordCounts], 4]
```

When we plot the number of times each of the different words appears, we see a so-called 'long-tail' distribution. This shows us that some of the words (like 'the' and 'Dolly') occur many times, but most of the words (like 'announced') only occur once. We will revisit this idea in a later lesson.

```
In[•]:= ListPlot[Values[dollyWordCounts], PlotRange → All, Joined → True]
```

Word case

Notice that WordCounts treats words in different cases as different words for the purposes of determining frequency.

```
In[•]:= dollyWordCounts["the"]
```

```
In[•]:= dollyWordCounts["The"]
```

Sometimes this is what you want. If not, you can use the IgnoreCase option with the WordCounts command. In the next expression, we compute the WordCounts for the plaintext, ignoring the case. This returns an association, which we use to look up the value associated with the key *the*.

```
In[•]:= WordCounts[dollyPlaintext, IgnoreCase → True]["the"]
```

Modal verbs

Words like *can*, *will* and *must* are called modal verbs. They express possibility, probability, obligation and permission, and can help to give us a sense of the tone of a text and the author's view of the future.

```
In[•]:= modalVerbs =
  {"can", "could", "may", "might", "must", "shall", "should", "will", "would"};
```

Here are the modal verbs used in the article about the cloning of Dolly the sheep.

```
In[•]:= Intersection[Keys[dollyWordCounts], modalVerbs]
```

You may not need to search for modal verbs specifically in your own research, but searching for a small set of terms in a much larger collection is a very common task.

Bigram and trigram frequencies

Although we haven't seen this yet, the WordCounts command can also be used to study longer sequences of words. A two-word sequence is called a bigram or 2-gram, and three word sequence is a trigram or 3-gram, and so on. In the next lesson we will look at the general case, referred to as n-grams.

```
In[•]:= dollyBigramCounts = WordCounts[dollyPlaintext, 2];
```

In[•]:=
```
TakeLargest[dollyBigramCounts, 10]
```

This is the syntax that you use to look up a value with a key.

In[•]:=
```
dollyBigramCounts[{"clone", "humans"}]
```

Notice that “Dr Ian” and “Ian Wilmut” each occur three times. Study the following expression and then describe what it does in words.

In[•]:=
```
WordCounts[dollyPlaintext, 3][{"Dr", "Ian", "Wilmut"}]
```

If you wanted to do more extensive work with trigrams than checking just this one thing, it would make sense to assign the output of WordCounts to a symbol.

Pattern matching in bigrams with Cases

The Keys of *dollyBigramCounts* consist of pairs of words that appear sequentially in the text. Here are the ten most frequent.

In[•]:=
```
Keys[TakeLargest[dollyBigramCounts, 10]]
```

We can use the Cases command to pull elements from a list that match a particular pattern. The pattern we are using below says to show us all Keys in the association of bigram frequencies that have “Dolly” as the first word or the second word, and any other word following or preceding it. Note that we are looking at all of the bigrams, not just the ten most frequent.

In[•]:=
```
Cases[Keys[dollyBigramCounts], {"Dolly", _}]
```

In[•]:=
```
Cases[Keys[dollyBigramCounts], {_, "Dolly"}]
```

In *Mathematica*, the underscore is a command called Blank. It is a pattern object that can stand for any expression. We will be using it frequently.

In[•]:=
```
? _
```

Datasets

Species data

Recall that one of the things we can request for an entity is a dataset of associated properties. The expression below shows the dataset of properties for the species specification of Sheep.

In[•]:=
```
sheep SPECIES SPECIFICATION ··· ✓ ["Dataset"]
```

Try clicking on the list of alternate scientific names. Note that the Dataset display collapses to show you the information in TableForm. You can return to the overall display by clicking the blue icon that represents the Dataset as a whole.

There are also forward and back arrows at the bottom of the Dataset display, and a scrollbar on the left.

We can request a Property of a SpeciesData Entity with a command like the following.

In[•]:= `SpeciesData[` sheep SPECIES SPECIFICATION `, "Genus"]`

The genus Ovis (the next level up in the taxonomic hierarchy) also has a dataset. Try clicking on the Value for *sibling taxa* and then returning to the main Dataset display.

In[•]:= Ovis SPECIES SPECIFICATION `["Dataset"]`

Sample datasets

Mathematica also has some built in example Datasets to explore.

In[•]:= `ExampleData["Dataset"]`

Note that in the example dataset for planets we find that Jupiter has 63 moons. What is the radius of Cyllene?

In[•]:= `ExampleData[{"Dataset", "Planets"}]`

We can also select a particular column or row to focus on. Try clicking the Radius column header for the planets, then returning. Then try clicking the Radius column header under Moons. Note that each gives you a different view of the Dataset.

Try clicking the row for Jupiter, then returning.

Person Entities

When an entity is recognized it appears in your notebook in a tan-colored box with rounded edges. Note the check box inside the right edge of the Entity box. If this is the correct interpretation, you can click it to confirm.

In[•]:= Ian Wilmut PERSON ☑

Each entity has a given type. *Mathematica* knows different things about different kinds of entities.

In[•]:= `EntityTypeName[` Ian Wilmut PERSON `]`

If we want to refer to an entity directly, it helps to know its canonical name. We can get this with the InputForm and CanonicalName commands.

In[•]:= `InputForm[` Ian Wilmut PERSON `]`

In[•]:= `CanonicalName[` Ian Wilmut PERSON `]`

When we have an entity type and canonical name, we can refer to a unique entity in our code.

In[•]:= `Entity["Person", "IanWilmut::53q4t"]`

For presentation purposes we sometimes want to use the person's common name instead of the entity.

In[•]:= `CommonName[` Ian Wilmut PERSON ☑ `]`

Names are not always unique identifiers. If there is more than one interpretation, the Entity box also contains a button with an ellipsis (...) If you are not sure you have found the correct entity, try clicking on that button to find other interpretations.

Robert H. Jackson PERSON

There are other properties associated with the entity which we can retrieve.

In[•]:= Ian Wilmut PERSON ["BirthDate"]

In[•]:= Ian Wilmut PERSON ["NationalityCountries"]

Here are all of the properties that are currently associated with the person entity.

In[•]:= EntityProperties["Person"]

If you need to type the property in, the CanonicalName is usually more convenient than the entity box.

In[•]:= CanonicalName[EntityProperties["Person"]]

We can ask for a complete list of properties associated with an entity to be returned as a dataset.

In[•]:= Ian Wilmut PERSON ["Dataset"]

Constructing queries with computable data

The vast network of entities and properties accessible in *Mathematica* allows us to construct complicated queries.

In[•]:= Richard M. Nixon PERSON

In[•]:= InputForm[Richard M. Nixon PERSON]

In[•]:= PersonData[Richard M. Nixon PERSON , "Children"]

What was the age difference of the daughters of Richard Nixon?

In[•]:= PersonData[PersonData[Entity["Person", "RichardMNixon::39q5n"], "Daughters"], {"Name", "BirthDate"}]

In[•]:= DateDifference[Day: Thu 21 Feb 1946 , Day: Mon 5 Jul 1948 , "Year"]

You are probably not specifically interested in sheep, planets, embryologists or presidential daughters, but the point is that there are datasets of computable entities for whatever real world phenomena you are interested in. If you can find and manipulate these entities, you can take advantage of *Mathematica*'s progressively higher level commands to sketch out solutions to research problems. In later lessons we will learn how to augment the built-in entity data of *Mathematica* with curated data that we retrieve from other sources or create on our own.

Building up a dataset

If you have a file of data, you can create a dataset by building up associations. In the next expression we import one of the files of sample data that comes with *Mathematica*. Note that this file has XML markup which we will ignore by using the Plaintext option for the Import command. The file consists of a collection of four-line *records*. Each record has information about the auction of a painting. The first line is the painting title, the second line is the artist, the third the year, and the fourth the price at auction. These lines are called the *fields* of the record.

```
In[•]:= auctionFile = Import["ExampleData/paintings.xml", "Plaintext"]
```

```
In[•]:= Head[auctionFile]
```

First we need to break the string into separate lines.

```
In[•]:= auctionList = StringSplit[auctionFile, "\n"]
```

Now we need to step through the list and label each element of each record. Recall that we can use Table for this kind of task. We need to set the iterator's step size to four so it steps through whole records. Spend a moment studying this code and make sure you understand what it is doing.

```
In[•]:= Table["title" → auctionList[[r]], {r, 1, 20, 4}]
```

Let's add the labels to the artist field. Make sure you understand why I am adding one to *r*.

```
In[•]:= Table[{"title" → auctionList[[r]], "artist" → auctionList[[r + 1]]}, {r, 1, 20, 4}]
```

Add the other two field labels. You should be able to predict what number we need to add to *r* in each case.

```
In[•]:= Table[{"title" → auctionList[[r]], "artist" → auctionList[[r + 1]],
    "year" → auctionList[[r + 2]], "price" → auctionList[[r + 3]]}, {r, 1, 20, 4}]
```

Right now we have a list of lists (of rules). We can convert each to an association by adding an Association command inside of the Table, so the output is a list of associations.

```
In[•]:= auctionData =
  Table[Association[{"title" → auctionList[[r]], "artist" → auctionList[[r + 1]],
     "year" → auctionList[[r + 2]], "price" → auctionList[[r + 3]]}], {r, 1, 20, 4}]
```

At this point, if we wrap the *auctionData* expression in the Dataset command, we have a dataset. Note that you can click on a column like artist or price.

```
In[•]:= auctionDataset = Dataset[auctionData]
```

Working with Lists and Associations

You can make a list of rules from a list of pairs of elements with the Rule and Apply commands. The Apply command can work at different levels in an expression, so we need the {1} argument to tell it to work only at the first level. Note that a, b and c in this expression are undefined symbols, so they evaluate to themselves.

In[•]:=
```
Apply[Rule, {{a, 12}, {b, 3}, {c, 7}}, {1}]
```

You can make an association from a list of rules with the Association command.

In[•]:=
```
Association[{a → 12, b → 3, c → 7}]
```

You can make an association into a list of rules with the Normal command.

In[•]:=
```
Normal[<|a → 12, b → 3, c → 7|>]
```

You can make a list of rules into a list of pairs of elements with the ReplaceAll command.

In[•]:=
```
ReplaceAll[{a → 12, b → 3, c → 7}, Rule → List]
```

You often face situations where you have information in one form and you need to convert it to another form so that you can perform some processing step.

The Query command

The Query command lets us retrieve specific information from a dataset. Here is how you request the first row. How would you request the second through fourth rows?

In[•]:=
```
Query[1][auctionDataset]
```

Request all artists. How would you request all prices?

In[•]:=
```
Query[All, "artist"][auctionDataset]
```

This expression pulls out specific rows, and specific columns for each row.

In[•]:=
```
Query[3 ;; 5, {"artist", "year"}][auctionDataset]
```

Compare this expression with the immediately previous one. How would you describe the difference in English?

In[•]:=
```
Query[{3, 5}, {"artist", "year"}][auctionDataset]
```

In the next expression, we order the rows of the Dataset by ascending year.

In[•]:=
```
Query[SortBy["year"]][auctionDataset]
```

What happens if you try to sort by artist? In order to get the correct behavior, how might we modify our dataset? How you choose to structure your data into records and fields will affect the ways you can interact with it.

Further examples

Here are some further examples of the Import command. Use these expressions to practice your code reading skills, consulting the documentation to learn more about commands you haven't seen so far. As always, the goal is not to memorize particular expressions, but rather to get a sense of what kind of things are possible and what steps you might take to achieve them.

Comics

Free public domain Golden Age comics.

https://www.digitalcomicmuseum.com

```
In[•]:= First@TakeLargestBy[
  Import["http://digitalcomicmuseum.com/preview/index.php?did=17932&page=2",
   "Images"], First[ImageDimensions[#]] &, 1]
```

UFO Reports

Canada's UFOs: The search for the unknown. Database of 9.5K digitized documents from government records at Library and Archives Canada.

https://www.collectionscanada.gc.ca/databases/ufo/index-e.html

```
In[•]:= ImageResize[
 Import["http://data2.collectionscanada.gc.ca/e/e110/e002744315.jpg"], 600]
```

Lyrics

The Music Lyrics Database. Lyrics for 236K songs from 23K albums.

http://www.mldb.org

```
In[•]:= StringJoin[StringTake[
  Import["http://www.mldb.org/song-205656-alphabet-aerobics.html", "Plaintext"],
  {692, 1255}], "\n..."]
```

Magazines

The Pulp Magazine Archive at the Internet Archive

https://archive.org/details/pulpmagazinearchive&tab=collection

```
In[•]:= iaPageSource = Import[
   "https://archive.org/details/pulpmagazinearchive?and[]=subject%3A%22science+
     fiction%22", "Source"];
```

```
In[•]:= Grid[Partition[ConformImages[Import["https://archive.org" <> #] & /@
    StringCases[iaPageSource, Shortest["<img class=\"item-img \"" ~~
         __ ~~ "source=\"" ~~ link__ ~~ "\""] → link][[1 ;; 21]], 7]]
```

TV Episodes

Doctor Who Time Travel Journeys from the *Guardian.*

https://docs.google.com/spreadsheets/d/1NubZNu9pQm5AvTVtNxaddW0PJ8kFN2-o2XNnTB9u5U0/edit#gid=26

```
In[•]:= drWhoTimeTravelJourneys = Import[
   "https://docs.google.com/spreadsheets/d/1NubZNu9pQm5AvTVtNxaddW0PJ8kFN2-
     o2XNnTB9u5U0/edit#gid=26", "Data"];
```

In[•]:=
```
Style[TableForm[drWhoTimeTravelJourneys[[1, 2, 1 ;; 24, 3 ;; 6]]], Medium]
```

See also the *Guardian*'s dataset of *Doctor Who* villains and monsters

https://docs.google.com/spreadsheet/ccc?key=0AonYZs4MzlZbdHJNSVh6clE4MVR4OHhqZWs4WGpSZlE #gid=0

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 5-8.

Workflow: Select Elements in a Dataset

https://reference.wolfram.com/language/workflow/SelectElementsInADataset.html

Workflow: Extract Columns in a Dataset

https://reference.wolfram.com/language/workflow/ExtractColumnsInADataset.html

References

On the difficulty of different problems at different levels

Anderson, P.W. “More is Different.” *Science* vol 177 no 4047 (4 Aug 1972): 393-396. DOI: 10.1126/-science.177.4047.393
https://science.sciencemag.org/content/177/4047/393

ch05

Lesson 05. Reusing Code

Black boxes

In this lesson we learn how to define functions, new commands that we create from the building blocks of built-in commands and expressions that have been written previously. In general, you don’t want to keep solving problems that you (or someone else) have already solved. Instead you package the solution in one or more functions, then use those when you need them. This code reuse allows programmers to build on one another’s work, and it serves an important conceptual purpose. By hiding the details of how something is computed, it allows you to focus on problems that haven’t been solved yet.

An engineering metaphor that is often used here is the 'black box'. As long as you understand the inputs it takes and the outputs it returns, you can safely ignore the details of how it works inside. If you need to look into the black box for some reason and make some adjustements, then *Mathematica* is designed so that you can do that, too.

Ways to input expressions

Clearing all definitions

```
In[•]:= Clear["Global`*"]
```

Standard Form

In *Mathematica*, every built-in command can be written with a name that begins with a capital letter and zero or more arguments in square brackets at the end. This is called the standard form and we have seen many examples of it.

```
In[•]:= RandomWord[]
```

```
In[•]:= Table[RandomWord[], {3}]
```

Some of the commands that we have seen did not appear in this standard form.

```
In[•]:= 2 + 2
```

Nevertheless, they have a standard form.

```
In[•]:= Plus[2, 2]
```

Infix form

The way that we usually write Plus is called infix form. Many commands have a shorthand infix notation, like Plus. StringJoin, for example, appears like this in standard form

```
In[•]:= StringJoin["#", "hashtag"]
```

It can also be written with infix notation as

```
In[•]:= "#" <> "hashtag"
```

In fact, any command can be written in infix form by putting tildes on either side of the command name, as shown in the following examples using List commands.

```
In[•]:= Join[{c, d, e}, {b, a, c}]
```

```
In[•]:= {c, d, e} ~Join~ {b, a, c}
```

```
In[•]:= Union[{c, d, e}, {b, a, c}]
```

```
In[•]:= {c, d, e} ~Union~ {b, a, c}
```

```
In[•]:= Intersection[{c, d, e}, {b, a, c}]
```

```
In[•]:= {c, d, e} ~Intersection~ {b, a, c}
```

Note the differences between the outputs created by Join and Union in the examples above. Try to describe what each command is doing.

Prefix and Postfix forms

When we write expressions using standard form notation, we *nest* them. The innermost command is evaluated, passing its output to the immediately enclosing command, which is then evaluated, passing its output to its immediately enclosing command, and so on. The next expression generates a random word, breaks the String into a List of individual Characters, and then Reverses that List.

In[•]:=
```
Reverse[Characters[RandomWord[]]]
```

In standard form notation, when we wrap an expression with another command, we put the command on the left and we have to add a closing square bracket somewhere to the right. Use your mouse cursor to select the square bracket after Reverse in the expression above, and note the green highlighting on its closing bracket. Try selecting the square brackets after Characters and then RandomWord.

In prefix form, the same expression would be written like this

In[•]:=
```
Reverse@Characters@RandomWord[]
```

This notation can be handy for building up a sequence of commands without worrying about placing the closing bracket. For example, if I write a Table command to create a list of RandomWords...

In[•]:=
```
Table[RandomWord[], {3}]
```

then decide I want them to appear one per line, I can add TableForm and the at sign to the beginning of my expression without having to place opening or closing brackets.

In[•]:=
```
TableForm@Table[RandomWord[], {3}]
```

Mathematica also has a postfix form which can be invoked with a pair of slashes. In this case, the command we are starting with is on the left, and its output flows into the command to its right, and so on.

In[•]:=
```
RandomWord[] // Characters // Reverse
```

This notation is especially good for building up an expression by adding one transformation at a time.

HoldForm

The main reason to understand these different kinds of notations is to be able to read and unpack expressions from the documentation. For example, here are two expressions that we used in the last lesson, written in the style in which they are most likely to appear in the documentation. They convert a List of List pairs into a List of Rule pairs, and back again.

In[•]:=
```
Rule @@@ {{a, 12}, {b, 3}, {c, 7}}
```

In[•]:=
```
{a → 12, b → 3, c → 7} /. Rule → List
```

If we come across an expression like this that we don't immediately understand, we can use two commands to turn it into a form we recognize.

The first is HoldForm. It prevents an expression from being evaluated.

In[•]:=
```
HoldForm[2 + 2]
```

Once we have wrapped an expression in HoldForm, it may become more clear that we have seen it before.

```
In[•]:= HoldForm[Rule @@@ {{a, 12}, {b, 3}, {c, 7}}]
```

This is not always the case, however.

```
In[•]:= HoldForm[{a → 12, b → 3, c → 7} /. Rule → List]
```

If the *held* expression still contains unfamiliar notation, try adding the FullForm command.

```
In[•]:= FullForm[HoldForm[{a → 12, b → 3, c → 7} /. Rule → List]]
```

This allows us to see that the slash dot notation is shorthand for the ReplaceAll command. It also underlines the fact that our expression contains Lists and Rules.

Being comfortable with these expressions also allows you to experiment with different ways of conceptualizing your code. Are you starting with a representation and applying a sequence of transformations to it? Did you come up with a complicated algorithm to do something then add visualization options as an afterthought? Do you want to emphasize something up front? When you are first learning to code, it is usually enough of a win just to get something working. As you become more experienced, you realize that some ways of doing things are more elegant, more efficient or easier to understand than others, and you become more concerned with matters of programming style. Not only is there more than one way to do something, but there are often many factors that can guide you in making a choice. But if you are just getting started with programming, the main thing is to get something working correctly.

Defining functions

A useful tool for viewing long sources

We have already seen that it is possible to assign a value to a symbol with the equals sign. The following expression assigns the poem 'The Raven' to the symbol *ravenText*. In this case, we have the option of getting a list of lines so we are using that. The semicolon at the end suppresses the output.

```
In[•]:= ravenText = ExampleData[{"Text", "TheRaven"}, "Lines"];
```

We know that *ravenText* now contains a list of Strings, and if we are familiar with the poem we may have a sense of roughly how long it is. But in general, we have been suppressing the output when we assign a text to a symbol because we don't want to fill our notebook up by displaying a long text. Instead we have been using the Short or StringTake commands to try to get a sense of what the text is like without displaying the whole thing.

In *Mathematica* we can build our own commands out of expressions. This is known as *defining a function*. As with our symbols, we want to be careful to begin our function names with a lower case letter.

The following expression defines a function called *viewData*. *viewData* takes one argument, which is the source that we want to view. Note that we describe the argument using a pattern involving Blank. The expression *x*_ means roughly "take any expression and refer to it as *x* for the purposes of this function." The *viewData* function displays whatever we give it in a graphic box known as a Pane, which

will be as wide as our notebook and 200 printer's points high. (A printer's point is the unit used for describing fonts. A 12 point font has characters which are 12 points high.) The Pane will have scrollbars and be enclosed in a box drawn with Framed.

```
In[•]:= viewData[x_] :=
         Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Now that we have defined this function, we can use it to explore our sample text. Note the curly braces at beginning and end, and the comma at the end of every line. This reminds us that this data is a list of strings.

```
In[•]:= viewData[ravenText]
```

Searching through a list of sentences

Using *viewData* we can read or skim through our source, but it would also be nice to be able to search for items of interest. The StringContainsQ command tells us whether one String contains another. (What does the fact that the command name ends in a capital letter Q tell you?)

```
In[•]:= StringContainsQ[ravenText, "tapping", IgnoreCase → True]
```

```
In[•]:= StringContainsQ[ravenText, "tipping", IgnoreCase → True]
```

If we take the first sentence from our list, we can test it using this command.

```
In[•]:= ravenText[[1]]
```

```
In[•]:= StringContainsQ[ravenText[[1]], "pondered", IgnoreCase → True]
```

```
In[•]:= StringContainsQ[ravenText[[1]], "tapping", IgnoreCase → True]
```

What we really want is a list of all of the sentences that contain the word of interest. To get this list, we are going to use the Select command. We have to tell Select what we are looking for, and to do this, we are going to use a pure function (also known as an anonymous or lambda function). The next section discusses pure functions in more detail.

```
In[•]:= Select[ravenText, StringContainsQ[#, "tapping", IgnoreCase → True] &]
```

Pure functions

We have learned that it is possible to define a named function using an expression like the one below.

```
In[•]:= stringContainsTapping[str_] :=
         StringContainsQ[str, "tapping", IgnoreCase → True]
```

```
In[•]:= stringContainsTapping["this is about TAPPING"]
```

```
In[•]:= stringContainsTapping["this is about TIPPING"]
```

This is a pretty specific function, since it only tests for the one word! Nevertheless, we can use it in our Select command.

```
In[•]:= Select[ravenText, stringContainsTapping]
```

The problem with this approach, of course, is that we don't want to have to define a new function for every word we might be interested in. The problem is compounded if we are searching for something complicated like "the word *tapping* within ten words of the word *Lenore* or the word *nevermore*".

In *Mathematica*, a pure function can be used in place of a named function. To create the pure function, we take the body of the defined function above, and replace the named variable *str* with a special argument called Slot (shorthand notation is #). We add the Function command (shorthand notation &) at the end of the expression.

```
In[•]:= StringContainsQ[#, "tapping", IgnoreCase → True] &
```

As we learned in this lesson, we can use the FullForm command to spell out each of the commands in our expression in standard form.

```
In[•]:= StringContainsQ[#, "tapping", IgnoreCase → True] & // FullForm
```

Now that we have our pure function, we can *apply* it to an argument like this

```
In[•]:= (StringContainsQ[#, "tapping", IgnoreCase → True] &)["this is about TAPPING"]
```

```
In[•]:= (StringContainsQ[#, "tapping", IgnoreCase → True] &)["this is about TIPPING"]
```

And we can use it in our Select command as we have already seen.

```
In[•]:= Select[ravenText, StringContainsQ[#, "tapping", IgnoreCase → True] &]
```

Now suppose we want to search for the word *rustling*. All we have to do is copy our Select expression and change the string.

```
In[•]:= Select[ravenText, StringContainsQ[#, "rustling", IgnoreCase → True] &]
```

Pure functions are used extensively in *Mathematica* so it is worth spending some time getting familiar with them.

Keyword in context (KWIC)

Word frequencies tell us a lot about a text, but one kind of information that they lose is the order of words. To overcome this problem, we can also extract small ordered sequences (called n-grams) and analyze their frequencies and distribution, too.

n-Grams and Partition

Recall that a sequence of two items is called a bigram, a sequence of three items is called a trigram, and so on. The items that we have considered so far have been words, but it is also possible to study sequences at other levels, like character bigrams and trigrams.

In the general case, we refer to *n-grams*. If we have a List of items, we can use the Partition command to create n-grams. The following expression creates a list of bigrams for the Integers in sequence from 1 to 10. The first argument is the input List, the second argument is the *n* of n-gram (2 in this case) and the third argument is the size of the offset (which is 1 in this case). Imagine that we have a window that is two characters wide, and we are sliding it across our input, moving it one character to the right each

time, until we reach the end.

```
In[•]:= Range[1, 10]
```

```
In[•]:= Partition[Range[10], 2, 1]
```

Here are the character trigrams of a RandomWord.

```
In[•]:= rW = RandomWord[]
```

```
In[•]:= Characters[rW]
```

```
In[•]:= Partition[Characters[rW], 3, 1]
```

Here are the word 4-grams of a sentence.

```
In[•]:= lincolnQuote = "that government of the people by
     the people for the people shall not perish from the earth";
```

```
In[•]:= Partition[TextWords[lincolnQuote], 4, 1]
```

n-Grams are useful because they preserve some of the information that is contained in sequences while still allowing comparisons to be made between different texts. Knowing that a text contains one copy of each of the words *bites*, *dog* and *man* is useful because it gives us at least some idea of what the text is about. This is word frequency. Knowing that one text contains the trigram {*dog*, *bites*, *man*} while another contains the trigram {*man*, *bites*, *dog*} allows us to extract a deeper level of meaning and to compare the two texts.

Importing a text

For most research projects that have a digital component, the majority of your sources will be either files on your computer or information on the web (which you can download or scrape to files on your computer). Here we use the Import command to retrieve a copy of the “long telegram” sent by George Kennan in February 1946 (see the “References” section for more information).

There are page images of the document on the website of the Harry S. Truman Presidential Library and Museum.

https://www.trumanlibrary.org/whistlestop/study_collections/coldwar/documents/index.php?documentdate=1946-02-22&documentid=6-6&pagenumber=1

A later lesson focuses on learning how to work with page images. For now we will extract a transcribed version of the text from another website. Click the link so you have a version open in a web browser

http://teachingamericanhistory.org/library/document/the-long-telegram/

Instead of working directly with the web sources we have identified, we are going to use an archived copy that I put in the Internet Archive’s Wayback Machine on 6 May 2018. (That way, this lesson will continue to work if the original links are broken in the future.)

We start by retrieving the text of the Long Telegram as a string. I have had to do a little bit of pattern matching to pull the text itself out of the surrounding text of the webpage. When you are reviewing this lesson, practice your code reading skills on this example.

```
In[•]:= kennanLongTelegramText = First[StringCases[Import[
          "https://web.archive.org/web/20180506192416/http://teachingamericanhistory.
            org/library/document/the-long-telegram/", "Plaintext"],
         Shortest["Related Documents" ~~ x__ ~~ "Source"] → x]];

In[•]:= viewData[kennanLongTelegramText]
```

Keyword in context (n-Gram version)

Here is some code to generate a Keyword in Context (KWIC) listing that is based on n-Grams. We will see what it does first, then spend some time reading the code to see how it does it.

```
In[•]:= stopwords = WordData[All, "Stopwords"];

In[•]:= nonStopwordQ[w_] :=
         Not[MemberQ[stopwords, w]]

In[•]:= kwicNGram[textstring_] :=
         Framed[Pane[Style[TableForm[
             SortBy[Select[Partition[ToLowerCase[TextWords[textstring]], 7, 1],
               nonStopwordQ[#[[4]]] &], {#[[4]], #[[5]], #[[6]], #[[7]]} &]], "Small"],
           {Automatic, 200}, Scrollbars → True]]

In[•]:= kwicNGram[kennanLongTelegramText]
```

The KWIC display shows us 7-grams sorted by the fourth (i.e., middle) term. If you look at the display, you will see the fourth column starts with some numbers then the words *abandon*, *abhor*, *able*, *abroad*, ... Using the KWIC display, we can see that the word *abhor* occurs once in the document (word frequency) and that the surrounding text is *be taught to abhor both anglo-saxon powers* (context).

Note that we can use the Cases command with a pattern to pull out this n-Gram.

```
In[•]:= kennan7Grams = Partition[ToLowerCase[TextWords[kennanLongTelegramText]], 7, 1];

In[•]:= Cases[kennan7Grams, {_, _, _, "abhor", _, _, _}]
```

If there are multiple matching n-Grams, Cases will retrieve them all.

```
In[•]:= Cases[kennan7Grams, {_, _, _, "abroad", _, _, _}]
```

Reading a more complicated code example

How should we approach reading a more complicated code example like this one? The first thing to note is that we have seen a lot of these commands before, in some form or other. The expression that computes *kennan7Grams* immediately above also appears in the *kwicNGram* code. The use of Framed and Pane is familiar from the *viewData* code. We've seen TableForm, SortBy and Select, and we just learned about pure functions. Even without knowing exactly how the code works yet, we can be confident that we will be able to figure it out because it is built from pieces that we can figure out.

What does the *stopwords* list contain?

```
In[•]:= Short@stopwords
```

How does *nonStopwordQ* work? If we give it a stopword it returns False, and otherwise it returns True.

```
In[•]:= nonStopwordQ["the"]
```

```
In[•]:= nonStopwordQ["moscow"]
```

At this point we could try building up the expression used in the body of *kwicNGram* from the inside out. For testing purposes, it is often a good idea to use more manageable data, so we will use the Lincoln quote instead of the Long Telegram.

We start with 7-grams.

```
In[•]:= Partition[ToLowerCase[TextWords[lincolnQuote]], 7, 1]
```

The next step is to use Select to get ones that do not have a stopword in the fourth position. This makes our KWIC display more useful, since we are rarely interested in all of the contexts in which we find the word *the*.

```
In[•]:= Select[Partition[ToLowerCase[TextWords[lincolnQuote]], 7, 1], nonStopwordQ[#[[4]]] &]
```

This expression sorts the list of 7-grams by the fourth column, and then the fifth column, and then the sixth column, and then the seventh. Note the use of a pure function as an argument for SortBy.

```
In[•]:= SortBy[Select[Partition[ToLowerCase[TextWords[lincolnQuote]], 7, 1],
  nonStopwordQ[#[[4]]] &], {#[[4]], #[[5]], #[[6]], #[[7]]} &]
```

It is easier to see the effect of the sorting when we display the List using TableForm.

```
In[•]:= TableForm[
 SortBy[Select[Partition[ToLowerCase[TextWords[lincolnQuote]], 7, 1],
   nonStopwordQ[#[[4]]] &], {#[[4]], #[[5]], #[[6]], #[[7]]} &],
 TableSpacing → {0, 1}]
```

From this point, it is all a matter of formatting.

Use a smaller font.

```
In[•]:= Style[TableForm[
  SortBy[Select[Partition[ToLowerCase[TextWords[lincolnQuote]], 7, 1],
    nonStopwordQ[#[[4]]] &], {#[[4]], #[[5]], #[[6]], #[[7]]} &]], "Small"]
```

Put it in a Pane with scrollbars (they are not necessary in this case, so they don't appear).

```
In[•]:= Pane[Style[TableForm[
    SortBy[Select[Partition[ToLowerCase[TextWords[lincolnQuote]], 7, 1],
      nonStopwordQ[#[[4]]] &], {#[[4]], #[[5]], #[[6]], #[[7]]} &]], "Small"],
 {Automatic, 200}, Scrollbars → True]
```

Draw a box around it and you are done.

```
In[•]:= Framed[Pane[Style[TableForm[
     SortBy[Select[Partition[ToLowerCase[TextWords[lincolnQuote]], 7, 1],
       nonStopwordQ[#[[4]]] &], {#[[4]], #[[5]], #[[6]], #[[7]]} &]], "Small"],
   {Automatic, 200}, Scrollbars → True]]
```

Concordances

In traditional humanities scholarship, the keyword in context display is more commonly known as a concordance. Before computing, scholars could spend whole careers creating concordances because they had to read slowly through a text or group of texts, keeping track of each word and the contexts in which it appeared. Then came the laborious processes of sorting, editing, typesetting, etc. Here is a link to page 163 of Horace Howard Furness, *A Concordance to Shakespeare's Poems* (1875) which shows the entries for *knife* to *laugh'd*. Given the incredible amount of labor in producing such a reference work by hand, they were typically reserved for the most important texts (most of them are for sacred works), and they would not have been created if they were not considered to be extremely valuable for humanistic scholarship.

https://archive.org/details/in.ernet.dli.2015.24171/page/n169

In 1946, a Jesuit priest named Roberto Busa began compiling a 56-volume index of the writings of Thomas Aquinas. At the outset, he imagined that he would need about 13 million cards to finish compiling the work. He got in touch with IBM and other institutions in search of "such mechanical devices as would serve to achieve the greatest possible accuracy, with a maximum economy of human labor." Working with the computers of the time, he subsequently invented the machine-generated concordance, the first of which was published in 1951. Most scholars date the origins of humanities computing as a discipline to this episode. (See the paper by Thomas N. Winter in the References for more detail.)

Developing a function

Here is a step-by-step example of how you might come up with a function of your own. Recall that we created an association of bigram frequencies with the WordCounts command.

```
In[•]:= lincolnQuote = "that government of the people by
     the people for the people shall not perish from the earth";
```

```
In[•]:= WordCounts[lincolnQuote, 2]
```

We can write a function of our own that does the same thing. We start by converting a string to a list of words.

```
In[•]:= lincolnQuote // TextWords
```

We use Partition to generate the bigrams.

```
In[•]:= Partition[lincolnQuote // TextWords, 2, 1]
```

The Tally command counts things.

```
In[•]:= Tally@Partition[lincolnQuote // TextWords, 2, 1]
```

We want to sort by the counts (the last item in each list) in descending order. The ReverseSortBy command does this.

```
In[•]:= ReverseSortBy[Tally@Partition[lincolnQuote // TextWords, 2, 1], Last]
```

Each of these nested lists should be a rule.

```
In[•]:= Rule @@@ ReverseSortBy[Tally@Partition[lincolnQuote // TextWords, 2, 1], Last]
```

And the output should be an association.

```
In[•]:= Association[
 Rule @@@ ReverseSortBy[Tally@Partition[lincolnQuote // TextWords, 2, 1], Last]]
```

Now let's define the function. If we copy the expression and paste it into a function definition, the only thing we have to do is replace the specific symbol we used (*lincolnQuote*) with the name we are using for our function's argument (*x*).

```
In[•]:= ourBigramCounter[x_] :=
 Association[Rule @@@ ReverseSortBy[Tally@Partition[x // TextWords, 2, 1], Last]]
```

```
In[•]:= ourBigramCounter[lincolnQuote]
```

This gives us the same bigram counts as the WordCounts command does, although bigrams with the same frequency are ordered differently.

Further examples

Finding defined symbols and functions

Symbol and function definitions will persist until we quit *Mathematica* or use the Clear command. If we want to see all of the symbols and functions that have been defined in a session, we use the Names command.

```
In[•]:= Names["Global`*"]
```

We can also use Information to learn more about user-defined symbols. Click on *viewData* in the display below.

```
In[•]:= Information["Global`*"]
```

Since we have gotten in the habit of using Clear to remove all definitions at the beginning of each lesson, we have to remember to copy any symbol and function definitions that we made previously if we want to use them in the current lesson. You will see examples of this later.

It is also possible to use Clear to remove a single definition.

```
In[•]:= Names["Global`*"]
```

After we use Clear, *ravenText* is no longer defined.

```
In[•]:= Clear[ravenText]
```

In[•]:=
```
ravenText
```

The Remove command gets rid of it all together.

In[•]:=
```
Remove[ravenText]
```

In[•]:=
```
Names["Global`*"]
```

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 9-12.

Tutorial: Everything Is an Expression

https://reference.wolfram.com/language/tutorial/EverythingIsAnExpression.html

Tutorial: Special Ways to Input Expressions

https://reference.wolfram.com/language/tutorial/SpecialWaysToInputExpressions.html

Tutorial: Defining Functions

https://reference.wolfram.com/language/tutorial/DefiningFunctions.html

How To: Work with Variables and Functions

https://reference.wolfram.com/language/howto/WorkWithVariablesAndFunctions.html

How To: Work with Lists

https://reference.wolfram.com/language/howto/WorkWithLists.html

References

For the historical setting of Kennan's "long telegram" and the origins of the Cold War in the aftermath of WWII

https://www.trumanlibrary.org/whistlestop/study_collections/coldwar/index.php?action=bg

For Roberto Busa and the origins of humanities computing

Winter, Thomas Nelson. "Roberto Busa, S.J., and the Invention of the Machine-Generated Concordance." *Faculty Publications, Classics and Religious Studies Department*, no. 70 (1999). https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1069&context=classicsfacpub

ch06

Lesson 06. Networks

Using metadata to find Paul Revere

In this lesson we work through an example of network analysis, based on a blog post by sociologist Kieran Healy. Here is a link to an archived copy of the post. You should read it before you begin this lesson.

https://web.archive.org/web/20190520204820/https://kieranhealy.org/blog/archives/2013/06/09/using-metadata-to-find-paul-revere/

In this lesson we make use of some mathematical techniques and terminology that may be unfamiliar to you. Don't panic! People who do technical work often find themselves confronting a mass of material that they need to understand at some level, but may not be familiar with. A programmer who starts to work on a large team project needs to get up to speed on the existing code base or learn a new programming language. A researcher begins an interdisciplinary collaboration and is confronted with the terminology and mathematics of a unfamiliar discipline. Even reading articles on new methods often involves a period of wading through stuff you haven't seen before.

In *The Emperor's New Mind* (1989), Oxford mathematician Sir Roger Penrose included a note to the reader on reading mathematical equations: "At a number of places in this book I have resorted to the use of mathematical formulae, unabashed and unheeding of warnings that are frequently given: that each such formula will cut down the general readership by half. If you are a reader who finds any formula intimidating (and most people do), then I recommend a procedure that I normally adopt myself when such an offending line presents itself. The procedure is, more or less, to ignore that line completely and to skip over to the next actual line of text! Well, not exactly this; one should spare the poor formula a perusing, rather than a comprehending glance, and then press onwards. After a little, if armed with new confidence, one may return to that neglected formula and try to pick out some salient features. The text itself may be helpful in letting one know what is important and what can be safely ignored about it. If not, then do not be afraid to leave a formula behind altogether."

The dataset

Clearing all definitions

In[•]:=
```
Clear["Global`*"]
```

Wolfram Data Repository

Healy's dataset is derived from a book by David Hackett Fischer, *Paul Revere's Ride* and was made available on GitHub. We will be using a copy of the dataset from the Wolfram Data Repository.

https://datarepository.wolframcloud.com/resources/a06cf759-9f16-48dc-b61c-2b9666cb6b25

If we wanted to see if the data was in the repository, we could search for it with the ResourceSearch command.

```
In[•]:= ResourceSearch[{"Network", "Colonial Boston"}]
```

It is there, so we retrieve the resource using ResourceObject.

```
In[•]:= revereResource = ResourceObject["Paul Revere's Social Network in Colonial Boston"]
```

Now we can ask for Properties of the dataset.

```
In[•]:= revereResource["Properties"]
```

```
In[•]:= revereResource["Description"]
```

```
In[•]:= revereResource["Keywords"]
```

```
In[•]:= revereResource["Details"]
```

Once we have decided that this is the resource we want, we retrieve a copy with ResourceData.

```
In[•]:= revereData = ResourceData["Paul Revere's Social Network in Colonial Boston"]
```

Selecting elements

Using the Select command to discover which organizations Samuel Adams does and does not belong to. Note that we use a form of the Slot command which includes the name of the field (column) in the dataset.

```
In[•]:= Select[revereData, #Name == "Samuel Adams" &]
```

If we just want a list of the organizations that Samuel Adams belongs to, we can use a Select command to retrieve the row, convert it to an association with Normal, then use Keys and another Select command to pull out the fields with a value of True.

```
In[•]:= Keys[Select[Normal[Select[revereData, #Name == "Samuel Adams" &]][[1]], # == True &]]
```

We can also construct more complicated queries with Select. The next expression shows us people who are members of both the Loyal Nine and the Tea Party, but not the Long Room Club.

```
In[•]:= Select[revereData, (#LoyalNine && Not[#LongRoomClub] && #TeaParty) &]
```

Here are the people who are members of North Caucus.

```
In[•]:= Normal[Select[revereData, #NorthCaucus &][[All, 1]]]
```

Later in the lesson it will be useful to know that we can write our pure function using the standard form of the Function and Slot commands.

```
In[•]:= FullForm@HoldForm[#NorthCaucus &]
```

So this is equivalent to the previous expression.

```
In[•]:= Normal[Select[revereData, Function[Slot["NorthCaucus"]]][[All, 1]]]
```

Matrices

Names of people and groups

To label our data we will frequently need to refer to people and groups. Here are lists of their respective names.

```
In[•]:= peopleNames = Normal[Values[revereData]][[All, 1]];
```

```
In[•]:= Short[peopleNames]
```

```
In[•]:= groupNames = Normal[Keys[revereData]][[1, 2 ;; 8]]
```

The people by groups matrix

In the post, Healy shows how to create a person by person matrix by manipulating matrices where membership is denoted by 1 and lack of membership by 0. We could load his original data, but instead I have used a *Mathematica* expression to transform our dataset into the required form. The Boole command converts *True* to 1 and *False* to 0.

```
In[•]:= peopleByGroups = Boole[Normal[Values[revereData][[All, 2 ;; 8]]]];
```

This matrix has 254 rows and seven columns.

```
In[•]:= Dimensions[peopleByGroups]
```

To compare with the figure in his blog post, here are the first 10 rows. We use the TableForm command with the TableHeadings option that allows us to label the rows with the names of the people and the columns with the names of the organizations.

```
In[•]:= TableForm[peopleByGroups[[1 ;; 10, All]],
  TableHeadings → {peopleNames[[1 ;; 10]], groupNames}]
```

Look at this figure and make sure you see how it is related to our previous view of *revereData*.

The groups by people matrix

The groups by people matrix is computed by taking the Transpose of the people by groups matrix.

```
In[•]:= groupsByPeople = Transpose[peopleByGroups];
```

Check that the Dimensions are correct.

```
In[•]:= Dimensions[groupsByPeople]
```

Look at the first five columns. Again we use the TableHeadings option to label our rows and columns.

```
In[•]:= TableForm[groupsByPeople[[All, 1 ;; 5]],
  TableHeadings → {groupNames, peopleNames[[1 ;; 5]]}]
```

Compare this figure with the previous one. Even you haven't worked with matrices before, it should be clear that transposing the matrix is simply, as Healy puts it, "flip[ping] it over on its side, so that the

rows are now the columns and *vice versa*."

The people by people matrix

If we use matrix multiplication to multiply these two matrices, we will end up with a 254 x 254 people by people matrix. Fortunately, *Mathematica* handles the details of the multiplication which is not a difficult process but can be error-prone if you do it by hand.

```
In[•]:= peopleByPeople = peopleByGroups.groupsByPeople;
```

```
In[•]:= Dimensions[peopleByPeople]
```

We can look at part of this matrix and compare it to the figure in Healy's post. I have replaced values on the diagonal for ease of comparison.

```
In[•]:= TableForm[ReplacePart[peopleByPeople[[1 ;; 6, 1 ;; 4]], {i_, i_} → "-"],
         TableHeadings → {peopleNames[[1 ;; 6]], peopleNames[[1 ;; 4]]}]
```

Now what this matrix shows is the number of groups that two people have in common. John Adams and Samuel Adams have two groups in common. What are they?

This was our expression to get a list of groups given a person. Let's turn it into a function.

```
In[•]:= Keys[Select[Normal[Select[revereData, #Name == "John Adams" &]][1]], # == True &]]
```

```
In[•]:= personToGroupList[p_] :=
         Keys[Select[Normal[Select[revereData, #Name == p &]][1]], # == True &]]
```

```
In[•]:= personToGroupList["John Adams"]
```

```
In[•]:= personToGroupList["Samuel Adams"]
```

The groups they have in common will be the Intersection of the two lists. Let's bundle this up into a function, too.

```
In[•]:= groupsInCommon[p1_, p2_] :=
         Intersection[personToGroupList[p1], personToGroupList[p2]]
```

```
In[•]:= groupsInCommon["John Adams", "Samuel Adams"]
```

Gilbert Ash has no groups in common with Dr Allen.

```
In[•]:= groupsInCommon["Gilbert Ash", "Dr Allen"]
```

The groups by groups matrix

Finally, we compute the groups by groups matrix by multiplying our two original matrices in the other order, as described in Healy's post.

```
In[•]:= groupsByGroups = groupsByPeople.peopleByGroups;
```

```
In[•]:= TableForm[ReplacePart[groupsByGroups, {i_, i_} → "-"],
         TableHeadings → {groupNames, groupNames}]
```

At this point, we can see that our matrix doesn't match the one in Healy's post. This is a common

problem when you are replicating someone else's work! The two most likely possibilities are (1) we made a mistake, or (2) our dataset is different. So let's start debugging.

This matrix is supposed to show how many people a pair of groups have in common. We will need a function that takes a group name and returns a list of the people who are members.

```
In[•]:= groupToPersonsList[g_] :=
         Normal[Select[revereData, Function[Slot[g]]][[All, 1]]]
```

```
In[•]:= groupToPersonsList["NorthCaucus"]
```

How many people are members in North Caucus?

```
In[•]:= Length@groupToPersonsList["NorthCaucus"]
```

Now we can define a *peopleInCommon* function for groups.

```
In[•]:= peopleInCommon[g1_, g2_] :=
         Intersection[groupToPersonsList[g1], groupToPersonsList[g2]]
```

Our *groupByGroups* matrix says the St Andrews Lodge and Loyal Nine have two people in common.

```
In[•]:= peopleInCommon["StAndrewsLodge", "LoyalNine"]
```

And that North Caucus and Tea Party have thirteen in common.

```
In[•]:= peopleInCommon["NorthCaucus", "TeaParty"]
```

```
In[•]:= peopleInCommon["NorthCaucus", "TeaParty"] // Length
```

The fact that our numbers seem to check out at this stage suggests we may be working with a slightly different data set than Healy. We investigate that next.

Healy's data

We can Import a copy of Healy's dataset from GitHub. In this case we want to get the initial dataset he committed to the software repository, rather than any subsequent revisions. Here is a link to that file.

https://raw.githubusercontent.com/kjhealy/revere/f6d01a8af633f05076f1d052bf1a6a6d79c79531/data/PaulRevereAppD.csv

We Import it and have a look.

```
In[•]:= healyData = Import["https://raw.githubusercontent.com/kjhealy/revere/
          f6d01a8af633f05076f1d052bf1a6a6d79c79531/data/PaulRevereAppD.csv"];
```

```
In[•]:= TableForm[healyData[[1 ;; 10, All]]]
```

We do the same series of calculations we just did with our dataset. We have to clip off the row and column headers.

```
In[•]:= healyPeopleByGroups = healyData[[2 ;; -1, 2 ;; 8]];
```

```
In[•]:= healyGroupsByPeople = Transpose[healyPeopleByGroups];
```

```
In[•]:= healyPeopleByPeople = healyPeopleByGroups.healyGroupsByPeople;
```

In[•]:=
```
healyGroupsByGroups = healyGroupsByPeople.healyPeopleByGroups;
```

Now we can look at the TableForm of his group by group matrix and compare it with his blog post.

In[•]:=
```
TableForm[ReplacePart[healyGroupsByGroups, {i_, i_} → "-"],
 TableHeadings → {healyData[[1, 2 ;; 8]], healyData[[1, 2 ;; 8]]}]
```

Since it matches exactly, we can be confident that we haven't made a mistake in our computation, we are just using a revised dataset.

Visualizing networks

A network of groups

We have already seen that given a matrix which contains information about which people belong to which group, we can use simple matrix operations to learn new information, like which group contains which people, which people might know each other through one or more groups, and which groups might have overlapping members. These are the kinds of questions you could answer on a case-by-case basis by studying the original table carefully, but it is very convenient to be able to do a simple computation which answers all such questions in one or two steps.

Network information can be visualized with diagrams, which greatly adds to the appeal of doing this kind of work.

Our matrix of groups by groups looks like this

In[•]:=
```
groupsByGroups // TableForm
```

When we visualize this network, we won't need to know how many members a group has in common with itself (these are the large numbers along the diagonal of the matrix). We use ReplaceAll to set these to zero.

In[•]:=
```
groupsByGroupsZeroDiagonal = ReplacePart[groupsByGroups, {i_, i_} → 0] ;
```

In[•]:=
```
groupsByGroupsZeroDiagonal // TableForm
```

Now we can visualize this as a graph. In order to label the vertices in our graph, we need a list of rules. The MapThread command does this, creating rules that map between vertex number (e.g., 1) and vertex name (e.g., "StAndrewsLodge").

In[•]:=
```
groupNameLabels = MapThread[Rule, {Range[7], groupNames}]
```

The AdjacencyGraph command plots a network diagram from a matrix containing information about which elements are linked to one another and the ImageSize option scales the picture to the width of the notebook.

In[•]:=
```
AdjacencyGraph[groupsByGroupsZeroDiagonal,
 VertexLabels → groupNameLabels, ImageSize → Full]
```

Compare this figure with the one in Healy's paper. Is the same information being represented? Do you have a preference for one visualization over the other?

A network of people

We can create an initial image of the network of individuals in the same way.

```
In[•]:= peopleByPeopleZeroDiagonal = ReplacePart[peopleByPeople, {i_, i_} → 0] ;
```

```
In[•]:= peopleByPeopleGraph = AdjacencyGraph[peopleByPeopleZeroDiagonal]
```

Mathematica's default is usually not to display a big hairball unless we indicate that that is what want (here by wrapping our command in SimpleGraph).

```
In[•]:= SimpleGraph[peopleByPeopleGraph, ImageSize → Full]
```

Again, spend a moment comparing this visualization to the one in Healy's paper.

Highlighting Paul Revere

Healy zooms in to his figure to show the location of Paul Revere in the network. Instead, we will use HighlightGraph to show Revere's position.

The Position command shows us where something is in a list. We use that to highlight the vertex which corresponds to Revere.

```
In[•]:= Position[peopleNames, "Paul Revere"]
```

```
In[•]:= HighlightGraph[SimpleGraph[peopleByPeopleGraph, ImageSize → Full], {200}]
```

Betweenness centrality

Revere is of interest in this context precisely because he lies on the shortest path between many pairs of individuals. As Healy writes, betweenness centrality "is a way of asking 'If I have to get from person a to person z, how likely is it that the quickest way is through person x?'".

The BetweennessCentrality command returns this measure for each person in the graph. Recall that Paul Revere's position is number 200 in this list.

```
In[•]:= bcList = BetweennessCentrality[peopleByPeopleGraph]
```

```
In[•]:= bcList[[200]]
```

In fact, we can make an association to link people's names to their betweenness centrality measure. We make a list of rules with MapThread and wrap in an Association command.

```
In[•]:= peopleBCAssoc = Association[MapThread[Rule, {peopleNames, bcList}]];
```

```
In[•]:= peopleBCAssoc["Paul Revere"]
```

```
In[•]:= peopleBCAssoc["Dr Allen"]
```

Now it is easy to grab the top ten people in terms of this measure.

```
In[•]:= TakeLargest[peopleBCAssoc, 10]
```

(Note that our numbers don't match Healy's exactly because we are working with a slightly revised version of the dataset. The top five remain in the same order, however.)

More centrality measures

As Healy notes, Revere tops the list of colonists in terms of a number of centrality measures. Here we calculate eigenvector centrality.

```
ecList = EigenvectorCentrality[peopleByPeopleGraph];
```

In[•]:=
```
peopleECAssoc = Association[MapThread[Rule, {peopleNames, ecList}]];
```

In[•]:=
```
TakeLargest[peopleECAssoc, 10]
```

Subgraphs

We can pull out a part of the graph to study in more detail. Here we select the ten members with the highest betweenness centrality measures and plot connections between them.

In[•]:=
```
bcTop10Names = Keys[TakeLargest[peopleBCAssoc, 10]]
```

(Practice your code reading skills on this next command when you are reviewing this lesson. We will learn more about the Map command soon.)

In[•]:=
```
bcTop10Vertices =
 Flatten[Position[peopleNames, #] & /@ Keys[TakeLargest[peopleBCAssoc, 10]]]
```

In[•]:=
```
MapThread[Rule, {bcTop10Vertices, bcTop10Names}]
```

We can see that each of these people has a number of connections with each of the others.

In[•]:=
```
Subgraph[peopleByPeopleGraph, bcTop10Vertices,
 VertexLabels → MapThread[Rule, {bcTop10Vertices, bcTop10Names}]]
```

What are the potential connections between Henry Bass and Thomas Chase?

In[•]:=
```
groupsInCommon["Henry Bass", "Thomas Chase"]
```

Let's repeat this analysis with the bottom 20. (How did I come up with this command? Compare it to the previous example.)

In[•]:=
```
Subgraph[peopleByPeopleGraph,
 Flatten[Position[peopleNames, #] & /@ Keys[TakeSmallest[peopleBCAssoc, 20]]],
 VertexLabels → MapThread[Rule,
   {Flatten[Position[peopleNames, #] & /@ Keys[TakeSmallest[peopleBCAssoc, 20]]],
    Keys[TakeSmallest[peopleBCAssoc, 20]]}]]
```

Note that if we only consider these 20 people they have broken into four separate networks. According to our metadata, Dr Allen knows John Boit, but he has no acquaintances in this subset of people who could introduce him to Samuel Austin. These are people in the network who are not well connected.

So could Paul Revere introduce Dr Allen to Samuel Austin?

In[•]:=
```
groupsInCommon["Paul Revere", "Dr Allen"]
```

In[•]:=
```
groupsInCommon["Paul Revere", "Samuel Austin"]
```

Yes, Revere knows Allen through North Caucus, and knows Austin through London Enemies, and thus could potentially introduce the men if they don't know one another already.

Paul Revere doesn't necessarily need to be the one to broker this introduction

All the people who know Dr Allen directly.

```
In[•]:= personToGroupList["Dr Allen"]
```

```
In[•]:= groupToPersonsList["NorthCaucus"]
```

All the people who know Samuel Austin directly.

```
In[•]:= personToGroupList["Samuel Austin"]
```

```
In[•]:= groupToPersonsList["LondonEnemies"]
```

The Intersection is the set of all of the people in both groups.

```
In[•]:= Intersection[groupToPersonsList["NorthCaucus"],
         groupToPersonsList["LondonEnemies"]]
```

```
In[•]:= Intersection[groupToPersonsList["NorthCaucus"],
          groupToPersonsList["LondonEnemies"]] // Length
```

There are 16 people who might make the introductions.

Related entities

The main dataset that we have been using is a good example of the kind of data that you won't find in *Mathematica*'s system of Entities or in most general knowledgebases, as it is too fine grained and specific to particular research questions. We have seen, however, that it is just as easy to work with imported datasets as built-in ones. And it is also the case that Mathematica does have general knowledge that corresponds to some of the specifics of this dataset.

It knows Paul Revere in the sense of having a Person entity, which we can use with TimelinePlot to visualize Revere's lifespan.

```
In[•]:= Paul Revere PERSON
```

```
In[•]:= TimelinePlot[{Interval[Paul Revere PERSON [{"BirthDate", "DeathDate"}]]}]
```

Mathematica also has an entity for the historical event, the American Revolutionary War. We can add this to our timeline, to show when in Revere's life the war began and ended.

```
In[•]:= Revolutionary War HISTORICAL EVENT ["Dataset"]
```

```
In[•]:= TimelinePlot[{{Interval[Paul Revere PERSON [{"BirthDate", "DeathDate"}]]},
          {Revolutionary War HISTORICAL EVENT}}]
```

We can search for other historical events that Revere was directly involved in, and add those to the plot.

```
In[•]:= EntityClass["HistoricalEvent",
          "PeopleInvolved" → MemberQ[Paul Revere PERSON]] // EntityList

In[•]:= TimelinePlot[{{Interval[Paul Revere PERSON [{"BirthDate", "DeathDate"}]]},
          {Revolutionary War HISTORICAL EVENT}, {Battle of Lexington and Concord HISTORICAL EVENT,
           Boston Massacre HISTORICAL EVENT, Paul Revere's midnight ride HISTORICAL EVENT}}]
```

Finally note that Mathematica may have different entities representing different aspects of the same event. The Revolutionary War has corresponding entities both as a historical event and as a military conflict. The latter has much more finely grained detail of a military historical sort.

```
In[•]:= Revolutionary War HISTORICAL EVENT

        American Revolutionary War MILITARY CONFLICT

In[•]:= battleList = American Revolutionary War MILITARY CONFLICT ["Battles"];

In[•]:= Length[battleList]
```

Here are 10 of the 242 battles of the American Revolutionary War that *Mathematica* knows about.

```
In[•]:= TimelinePlot[battleList[[1 ;; 10]]]
```

As a final example, we pull out the Historical Site entities that commemorate events of the American Revolutionary War and plot them on a map centered on Boston.

```
In[•]:= GeoListPlot[EntityClass["HistoricalSite",
          "CommemoratedSubjects" → MemberQ@American Revolutionary War MILITARY CONFLICT] //
        EntityList, GeoLabels → True, GeoRange → Quantity[800, "Miles"],
       GeoCenter → Boston CITY, ImageSize → Full]
```

Further examples

Adjacency matrices and edge lists

The documentation for the dataset includes two other examples of graphing social networks.

https://datarepository.wolframcloud.com/resources/SwedeWhite_Paul-Reveres-Social-Network-in-Colonial-Boston

In both cases, they first transform the membership matrix into list of edges.

Here is a very simple matrix of adjacencies (this is the kind of representation we have been using).

In[•]:=
```
simpleAdjacencyMatrix = {{0, 1, 1, 1}, {1, 0, 1, 0}, {1, 1, 0, 1}, {1, 0, 1, 0}};
```

Looking at the matrix, we can see that vertex 1 is connected to the other three, but that vertex 2 is only connected to vertices 1 and 3.

In[•]:=
```
TableForm@simpleAdjacencyMatrix
```

Easier to visualize like this:

In[•]:=
```
AdjacencyGraph[simpleAdjacencyMatrix, VertexLabels → "Index", ImageSize → Small]
```

In *Mathematica*, a network can also be represented by specifying a list of edges. You make the ⟷ symbol by typing ESC ue ESC This is the shorthand notation for UndirectedEdge.

In[•]:=
```
simpleEdgeList = {1 ⟷ 2, 2 ⟷ 3, 3 ⟷ 1, 1 ⟷ 4, 3 ⟷ 4}
```

In this form, we can plot a network diagram with the Graph command.

In[•]:=
```
Graph[simpleEdgeList, VertexLabels → "Index", ImageSize → Small]
```

Keeping that in mind, spend some time reading the code that creates these next two figures. Remember, when you are reading code, start by focusing on the parts of the expression that you do understand or at least have seen before, then look up commands that you are less familiar with in the documentation. Even after studying an expression for a while, you may not completely understand it, but as it becomes more familiar you will start to have a sense of how you might copy and modify the code, or substitute different elements into it.

Graph of group memberships

Here the groups are highlighted.

In[•]:=
```
HighlightGraph[Flatten[Normal[
   With[{row = #, name = #Name}, If[row[#], name ↔ #, Nothing] & /@ groupNames] & /@
    revereData]], groupNames, ImageSize → Full]
```

Neighborhood graph of Paul Revere's group memberships

In[•]:=
```
With[
 {g = Flatten[Normal[With[{row = #, name = #Name}, If[row[#], name ↔ #, Nothing] & /@
         groupNames] & /@ revereData]]}, HighlightGraph[
  g, NeighborhoodGraph[g, "Paul Revere", 1, VertexLabels → Automatic],
  ImageSize → Full]]
```

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 13-16.

Video: Graphs and Networks: Concepts and Applications

Covers fundamental properties of graphs and introduces network analysis. For beginners.

https://www.wolfram.com/wolfram-u/catalog/dat021/

Example: Centrality and Prestige of Florentine Families

Shows how the Medici family gained a central position in the network of Florentine families through marriages, “crucial for communication, brokering deals, etc.”

http://www.wolfram.com/mathematica/new-in-9/social-network-analysis/centrality-and-prestige-of-florentine-families.html

References

Fischer, David Hackett. *Paul Revere’s Ride*, rev. ed. Oxford University Press (1995, original 1994).

Healy, Kieran. “Using Metadata to Find Paul Revere.” KieranHealy.org (9 June 2013). https://kieranhealy.org/blog/archives/2013/06/09/using-metadata-to-find-paul-revere/

Healy, Kieran. “Following Up on Paul Revere.” KieranHealy.org (11 June 2013). https://kieranhealy.org/blog/archives/2013/06/11/following-up-on-paul-revere/

Penrose, Roger. *The Emperor’s New Mind: Concerning Computers, Minds, and the Laws of Physics*, rev ed. Oxford University Press (2016, original 1989). Quote from vi.

ch07

Lesson 07. Indexing and Searching

Finding what you’re looking for in an age of abundance

In 2003, the late Roy Rosenzweig characterized the digital era as one of abundance. “[W]hat would it be like to write history,” he asked, “when faced by an essentially complete historical record?” At the same time, he noted that “As yet, no one has figured out how to ensure that the digital present will be available to the future’s historians.” They still haven’t. If anything, the problem is many orders of magnitude more acute now than it was at the time.

Digital sources are superabundant but attention is scarce. In order for anyone to be able to find anything at all in a reasonable amount of time, information must be constantly indexed. At the level of the web, this job is (partly) handled by giant search engine companies and a host of other institutions. In your own collection of files, the operating system and other search software does the task. For your own research you can and will rely on implicit and explicit indexing initiated by other agents (both human and machine), but you will also need to be able to create your own indexes, to find exactly what you need in a timely fashion.

Indexing

Clearing all definitions

In[•]:=
```
Clear["Global`*"]
```

Position

When we create an index for something, we indicate where to find it. Here is a small text to work with, the famous soliloquy from Shakespeare's *Hamlet*.

In[•]:=
```
sampleText = ExampleData[{"Text", "ToBeOrNotToBe"}]
```

We can turn this string into a list of words with the TextWords command, and we can determine whether a word is present in the list with the StringCases command.

In[•]:=
```
sampleTextWords = TextWords@sampleText;
```

In[•]:=
```
Flatten[StringCases[sampleTextWords, "life", IgnoreCase → True]]
```

In[•]:=
```
Flatten[StringCases[sampleTextWords, "loaf", IgnoreCase → True]]
```

The Position command tells us where it is in the list. Note that Position is case sensitive.

In[•]:=
```
Position[sampleTextWords, "life"]
```

In[•]:=
```
Position[sampleTextWords, "Nymph"]
```

In[•]:=
```
Position[sampleTextWords, "nymph"]
```

We can use the output of Position with Part, to see the word itself. Since Position returns a nested list, we also have to use Part to reach inside this list and pull out the index itself.

In[•]:=
```
sampleTextWords[[Position[sampleTextWords, "Nymph"][[1, 1]]]]
```

If we subtract and add a small number to the Position index we can see the n-gram context of a word. Here we subtract three and add three, to get a 7-gram centered on the index. Note that we didn't use Partition at all in this process. We know where our word is in the list (its index), so we can see the words to either side of it by using indexes to either side.

In[•]:=
```
sampleTextWords[[Position[sampleTextWords, "Nymph"][[1, 1]] - 3
  ;; Position[sampleTextWords, "Nymph"][[1, 1]] + 3]]
```

There is one more wrinkle. Suppose we are indexing the first or last word in the sample text. We have to be careful not to subtract past the beginning or add past the end of the list, or we will get an error message. Here the Max and Min commands prevent us from asking for positions that would be outside the list. It works for the first words in the list.

In[•]:=
```
sampleTextWords[[Max[1, Position[sampleTextWords, "To"][[1, 1]] - 3] ;;
  Min[Length[sampleTextWords], Position[sampleTextWords, "To"][[1, 1]] + 3]]]
```

Make sure it also works for the end of the list.

```
In[•]:= sampleTextWords[[Max[1, Position[sampleTextWords, "remember'd"][[1, 1]] - 3] ;;
          Min[Length[sampleTextWords], Position[sampleTextWords, "remember'd"][[1, 1]] + 3]]]
```

With these refinements, we can write a small function to grab a single n-gram context for a word in the text.

```
In[•]:= wordPositionToNGram[wordlist_, pos_, r_] :=
         wordlist[[Max[1, pos - r] ;; Min[Length[wordlist], pos + r]]]
```

The word *Nymph* is at position 271. Here we show a 9-gram around it.

```
In[•]:= wordPositionToNGram[sampleTextWords, 271, 4]
```

Next we want to be able to show the n-gram context for a word which occurs repeatedly, but before we do that, we pause for a quick introduction to the Map command.

Map and its relatives

We have seen the powerful Map command or one of its relatives a number of times. If you have a list, Map allows you to do something to each member of the list. Here are the first ten words of Hamlet's soliloquy in the order in which it appears in the text. Each has been reversed.

```
In[•]:= Map[StringReverse, sampleTextWords[[1 ;; 10]]]
```

We can also write this in infix form as

```
In[•]:= StringReverse /@ sampleTextWords[[1 ;; 10]]
```

The MapThread command takes a pair of lists of the same length, and combines them with some function. Here we use it to create a list of rules from two lists.

```
In[•]:= MapThread[Rule, {Range[10], CharacterRange["A", "J"]}]
```

As we have seen, this kind of expression can be very useful for creating associations.

Map, MapThread, Apply, Function and Slot are examples of commands that are used for a style of programming known as functional programming. It is a very natural and powerful way to do things in *Mathematica* (and in languages like LISP, Scheme, Haskell, Scala and their relatives.) If you don't already have much programming experience, functional style isn't hard to pick up. If you have programmed before (but mostly in languages like C, Java or Python) it may take a little longer to wrap your head around. It is definitely worth it, as part of the fun of learning a new language is learning new ways to express yourself.

Word context

To return to our ongoing example, we would like to see all of the n-gram contexts for a word that appears one or more times in our sample text. We create a pure function version of the *wordPositionToNGram* function then Map it across the list of positions we get from the Position command.

```
In[•]:= wordPositionToNGram[sampleTextWords, #, 4] & /@ Position[sampleTextWords, "Nymph"]
```

```
In[•]:= wordPositionToNGram[sampleTextWords, #, 4] & /@ Position[sampleTextWords, "life"]
```

If we format with TableForm this is starting to look a lot like our KWIC display.

```
In[•]:= wordPositionToNGram[sampleTextWords, #, 4] & /@
          Position[sampleTextWords, "sleep"] // TableForm
```

Of course we already knew how to create a list of n-grams for a particular term. In that case, however, we first had to pass the Partition command across the list of words to create a list of n-grams. In this case, we don't have to precompute all of the n-grams. We just see the ones that surround our term of interest.

In this particular text, one stopword plays an outsize role (*to*). Recall that we deleted stopwords for our KWIC display. Here we still have the stopwords, so we can study their distribution. The vertical bar (|) is shorthand for the Alternatives command. It lets us look for more than one thing at a time.

```
In[•]:= wordPositionToNGram[sampleTextWords, #, 4] & /@
          Position[sampleTextWords, "to" | "To"] // TableForm
```

StringPosition

The StringPosition command can be used to index a string. It returns a range of character positions.

```
In[•]:= StringPosition[sampleText, "life"]
```

```
In[•]:= StringPosition[sampleText, "Nymph"]
```

```
In[•]:= StringPosition[sampleText, "nymph"]
```

(How else could we find out if the *sampleText* string contains the string "nymph"?)

Let's create a string version of our KWIC display to give ourselves more practice with indexing. Here is a list of unique words.

```
In[•]:= sampleTextUniqueWords = Union@sampleTextWords
```

We will use Table to step through this list, creating an Association that maps the word to a list of character positions where it appears.

```
In[•]:= sampleTextWordIndex = Association[
          Table[w → StringPosition[sampleText, WordBoundary ~~ w ~~ WordBoundary],
           {w, sampleTextUniqueWords}]];
```

Looking at this association, we can see that the word *a* appears in five places, the word *action* only once.

```
In[•]:= Short[sampleTextWordIndex, 4]
```

Since it is an Association, we can look up a word easily enough.

```
In[•]:= sampleTextWordIndex["Nymph"]
```

```
In[•]:= sampleTextWordIndex["nymph"]
```

Now, given a string position, we would like to see its right and left context. As before, we have to be careful about requesting characters before the beginning or after the end of our string, which would

result in an error message.

```
In[•]:= stringPositionToContext[textstr_, {start_, end_}, r_] :=
         StringTake[textstr, {Max[1, start - r], Min[StringLength[textstr], end + r]}]
```

We can test with the position for *Nymph*, showing 40 characters on either side.

```
In[•]:= stringPositionToContext[sampleText, {1468, 1472}, 40]
```

As before, we can now Map this across the string positions associated with a particular word.

```
In[•]:= stringPositionToContext[sampleText, #, 40] & /@ StringPosition[sampleText, "Nymph"]
```

```
In[•]:= stringPositionToContext[sampleText, #, 40] & /@ StringPosition[sampleText, "life"]
```

```
In[•]:= stringPositionToContext[sampleText, #, 40] & /@
          StringPosition[sampleText, "sleep"] // TableForm
```

Indexing and searching

If we want to be able to search through our sources, we typically need an index. Either we build one ourselves, or some other program does it and hides the details from us.

For example, you can search your *Mathematica* notebook with CMD-f (OS X) or CTRL-f (Windows). This is actually very useful if you know that you have seen an example of a command but can't remember what lesson it was in. The fact that you can do this tells you that, at some level, *Mathematica* must have indexed the notebook. The same goes for search and replace in your word processor or your ability to search through your sent e-mail folder.

Pattern matching

We know how to find particular words in lists and strings, but sometimes we are not exactly sure what we are looking for. In this section we explore a variety of strategies for broadening our searches. Patterns allow us to describe collections of words or other linguistic elements. We can find, for example, all of the words in a text that begin with *str-*, all of the bigrams of the form *she X-ed*, or all of the sentences that contain *Saturday* and/or *Sunday* but not *September*.

Related words

If we have a term of interest, we can use dictionary information to generate other terms that may be of interest. The WordStem command removes plurals, inflections, etc.

```
In[•]:= WordStem["officer"]
```

Given a word stem, we can use the DictionaryLookup command to find all English words that contain "offic". The argument that we are giving to the DictionaryLookup command here is called a string pattern. This pattern says "match any string that contains "offic" with zero or more characters preceding it, and zero or more characters following it."

```
In[•]:= DictionaryLookup[___ ~~ "offic" ~~ ___]
```

The Intersection command can be used to find all the elements that two lists have in common. Here we discover whether any words containing “offic” also appear in Hamlet’s soliloquy.

```
In[•]:= Intersection[sampleTextWords, DictionaryLookup[___ ~~ "offic" ~~ ___]]
```

The WordData command can be used to find synonyms. Sometimes these provide additional ideas for searching.

```
In[•]:= WordData["calamity", "Synonyms", "List"]
```

```
In[•]:= WordData["troubles", "Synonyms", "List"]
```

Finding two terms in the same sentence

Here is Hamlet’s soliloquy broken down into a list of sentence strings.

```
In[•]:= sampleTextSentences = TextSentences[sampleText];
```

And the first ‘sentence.’ Since the TextSentences command was not specifically designed to parse Shakespearean plays, these sentence boundaries may not be exactly where you would expect them. If your research depends on a particular analysis, you can always manually divide a text into a list of linguistic units, of course.

```
In[•]:= First@sampleTextSentences
```

The StringCases command returns a list of matching elements in a string.

```
In[•]:= StringCases[First@sampleTextSentences, "sleep", IgnoreCase → True]
```

We have to be careful with word boundaries, however. The next output makes it look like the word *and* appears five times.

```
In[•]:= StringCases[First@sampleTextSentences, "and", IgnoreCase → True]
```

But when we are explicit about word boundaries, we see there are only four instances of the word *and*. Why does the previous output list five? What command(s) would you use to confirm your hypothesis?

```
In[•]:= StringCases[First@sampleTextSentences,
 WordBoundary ~~ "and" ~~ WordBoundary, IgnoreCase → True]
```

In order to find two words in the same sentence, we define a function that uses the StringCases command to pull out all sentences that begin with zero or more characters, followed by a word boundary, the first keyword, another word boundary, and finally zero or more characters. Once we have this list, we use a second StringCases command to pull out those sentences that also contain the second keyword (using a similar pattern). The Flatten commands get rid of nested lists. The TabView command lets us tab through a set of results.

```
In[•]:= searchNear[sentencelist_, kw1_, kw2_] :=
 TabView[Flatten[StringCases[
    Flatten[
     StringCases[sentencelist, ___ ~~ WordBoundary ~~ kw1 ~~ WordBoundary ~~ ___]],
    ___ ~~ WordBoundary ~~ kw2 ~~ WordBoundary ~~ ___]]]
```

Here is how we call our function. It will return all sentences that have either *die* or *death* and *sleep*. Remember that the vertical bar (|) is the infix form of the Alternatives command.

```
In[•]:= searchNear[sampleTextSentences, "die" | "death", "sleep"]
```

String pattern examples

In string matching, a single underscore matches one character. Here are all the letters that come before “m” in the soliloquy. The double tilde is shorthand for the StringExpression command. The Flatten command creates a flat list; the DeleteDuplicates command eliminates duplicates. Note that we are searching through the list of words that appear in the text.

```
In[•]:= DeleteDuplicates[Flatten[StringCases[sampleTextWords, _ ~~ "m"]]]
```

A double underscore matches one or more characters. All the words that end in “ly”:

```
In[•]:= DeleteDuplicates[Flatten[StringCases[sampleTextWords, __ ~~ "ly"]]]
```

The words that begin with “qu”.

```
In[•]:= DeleteDuplicates[Flatten[StringCases[sampleTextWords, "qu" ~~ __]]]
```

In the previous examples we were matching in our list of words. Here we try matching in our list of sentences.

All cases where there is a single character followed by “d”. Note that this matches both punctuation (the apostrophe in a hyphenated word) and the blank space before the word by itself.

```
In[•]:= DeleteDuplicates[Flatten[StringCases[sampleTextSentences, _ ~~ "d"]]]
```

The WordCharacter command matches any letter or digit, but not punctuation or whitespace. So neither a blank space nor an apostrophe before “d” show up with this search.

```
In[•]:= DeleteDuplicates[Flatten[StringCases[sampleTextSentences, WordCharacter ~~ "d"]]]
```

The Except command lets us pull out all the examples except the ones involving letters and digits.

```
In[•]:= DeleteDuplicates[
 Flatten[StringCases[sampleTextSentences, Except[WordCharacter] ~~ "d"]]]
```

You can search for repeated elements using Repeated (shorthand two dots) or RepeatedNull (shorthand three dots). Repeated matches one or more copies of something. One or more word characters joined to “and”.

```
In[•]:= DeleteDuplicates[
 Flatten[StringCases[sampleTextSentences, WordCharacter .. ~~ "and"]]]
```

Zero or more word characters joined to “and”.

```
In[•]:= DeleteDuplicates[
 Flatten[StringCases[sampleTextSentences, WordCharacter ... ~~ "and"]]]
```

The StartOfString and EndOfString commands can be used to pick out the first and last words in sentences.

```
In[•]:= DeleteDuplicates[
         Flatten[StringCases[sampleTextSentences, StartOfString ~~ WordCharacter ..]]]
```

Words at the end of sentences are preceded by one or more punctuation characters.

```
In[•]:= DeleteDuplicates[Flatten[StringCases[sampleTextSentences,
           WordCharacter .. ~~ Except[WordCharacter] .. ~~ EndOfString]]]
```

There is usually a string pattern for exactly what you are looking for, but they can sometimes be tricky to come up with.

Named patterns

If we just want to return part of an expression matched by a pattern, we give part of our pattern a name and use the Rule command.

All of the words following the word *to*. Note the use of WhitespaceCharacter here. Also note that DeleteDuplicates treats “To sleep” and “to sleep” as different.

```
In[•]:= DeleteDuplicates[Flatten[StringCases[sampleTextSentences,
           "to" ~~ WhitespaceCharacter ~~ WordCharacter .., IgnoreCase → True]]]
```

If we just want the words themselves, and we don’t want *to*, we can use a named pattern like this...

```
In[•]:= DeleteDuplicates[
         Flatten[StringCases[sampleTextSentences, "to" ~~ WhitespaceCharacter ~~
             following : WordCharacter .. → following, IgnoreCase → True]]]
```

Higher level indexing

TextCases and TextPosition

Earlier we briefly saw the powerful TextContents command in action. It uses neural nets to extract a wide variety of different kinds of grammatical, structural and semantic information from natural language. TextContents has two relatives, TextCases and TextPosition, which can be used to do higher level indexing tasks.

For example, consider the problem of indexing Twitter handles or email addresses in text. You can probably imagine creating a string pattern that would look for the characteristic at sign (@) and try to get the appropriate amount of material to either side. Indexing currencies would be more difficult because you have to take many different kinds of units into account. To index cities or countries you would probably start from a list or gazetteer. What about topics? How would you distinguish a conversation on food and drink from one on fitness?

This is where indexing based on artificial intelligence can be very valuable. Here are some examples of the TextCases and TextPosition commands at work.

```
In[•]:= TextCases[
 "I heard that someone sent 1500 ¥ to wjt@alum.mit.edu via PayPal for his
   autograph.", {"EmailAddress", "CurrencyAmount", "Company"}]
```

```
In[•]:= TextPosition[
 "I heard that someone sent 1500 ¥ to wjt@alum.mit.edu via PayPal for his
   autograph.", {"EmailAddress", "CurrencyAmount", "Company"}]
```

Indexing

We can modify some of the code we created in earlier sections of this lesson to take advantage of these commands. For example, here are the text positions of all of the verbs in Hamlet's soliloquy, starting with *outrageous*.

```
In[•]:= sampleTextAdjectiveIndex = TextPosition[sampleText, "Adjective"]
```

If we Map *stringPositionToContext* across this list, we get a KWIC display of the adjectives in context.

```
In[•]:= stringPositionToContext[sampleText, #, 35] & /@ sampleTextAdjectiveIndex // TableForm
```

You can use exactly the same kind of code to create KWIC displays for any of dozens of different text content types, including emoticons, foreign words, profanity, colors, quantities, dates, URLs, and a huge variety of entities like beaches, oil fields, library branches, tropical storms, board games, food brand names, cat breeds, proteins, etc.

Content topics

You can also identify various kinds of content topics, as the following example shows.

First we import and clean up an archived copy of breaking news headlines and short descriptions from the BBC (from 24 May 2019). This news feed is in XML, which we will learn to manipulate in a future lesson.

```
In[•]:= newsfeed = Import[
   "https://web.archive.org/web/20190524193057/http://feeds.bbci.co.uk/news/rss.
     xml", "SymbolicXML"];
```

Next we create a single text by using StringRiffle to join the short descriptions together, one per line.

```
In[•]:= newsfeedDescriptionText = StringRiffle[Flatten@Cases[newsfeed[[2, 3, 1, 3]],
     XMLElement["item", _, {___, XMLElement["description", _, x_], ___}] → x], "\n"];
```

Now we use the TextContents command to classify the stories, setting the AcceptanceThreshold high to eliminate false positives.

```
In[•]:= TextContents[newsfeedDescriptionText, "SportsTopic", AcceptanceThreshold → .8]
```

```
In[•]:= TextContents[newsfeedDescriptionText, "HealthTopic", AcceptanceThreshold → .8]
```

```
In[•]:= TextContents[newsfeedDescriptionText, "PoliticsTopic", AcceptanceThreshold → .8]
```

Plotting term distribution

Sometimes we would like to visualize how terms are distributed across a text. In this example we are going to use the novel *Alice in Wonderland*.

```
In[•]:= aliceText = ExampleData[{"Text", "AliceInWonderland"}];
```

We can use the Partition command to break the text into non-overlapping, 100-word blocks as follows.

```
In[•]:= alice100WordBlocks = Partition[TextWords[aliceText], 100, 100, 1, {}];
```

Here is the first such block.

```
In[•]:= First@alice100WordBlocks
```

We can use a command called SequenceCases to find a sequence of elements (i.e., an n-gram) in a block if it exists.

```
In[•]:= SequenceCases[First@alice100WordBlocks, {"making", "a", "daisy-chain"}]
```

If we just give it a list of one element to look for, it behaves a lot like Cases. (But note the differences in the arguments and the outputs).

```
In[•]:= SequenceCases[First@alice100WordBlocks, {"daisy-chain"}]
```

```
In[•]:= Cases[First@alice100WordBlocks, "daisy-chain"]
```

If we take the Length of the output of SequenceCases, we find that the value is 0 if the sequence does not appear, and 1 or more if it does.

```
In[•]:= SequenceCases[First@alice100WordBlocks, {"daisy"}] // Length
```

```
In[•]:= SequenceCases[First@alice100WordBlocks, {"daisy-chain"}] // Length
```

```
In[•]:= SequenceCases[First@alice100WordBlocks, {"sister"}] // Length
```

We can wrap this in a function that uses Map to count the number of times a sequence appears in each text block in the book.

```
In[•]:= getOccurrences[txtblks_, seqlist_] :=
 Map[Length[SequenceCases[#, seqlist]] &, txtblks]
```

Here we are looking for Alice.

```
In[•]:= getOccurrences[alice100WordBlocks, {"Alice"}]
```

This is easier to visualize with a command called MatrixPlot. The darker the color, the more copies of the search term appear in that text block.

```
In[•]:= MatrixPlot[{getOccurrences[alice100WordBlocks, {"Alice"}]}, ImageSize → Full]
```

Here are the distributions of some other terms in the book.

```
In[•]:= MatrixPlot[{getOccurrences[alice100WordBlocks, {"Rabbit"}]}, ImageSize → Full]
```

```
In[•]:= MatrixPlot[
 {getOccurrences[alice100WordBlocks, {"March", "Hare"}]}, ImageSize → Full]
```

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 17-20.

Example: Harvest entities in text

Another example of the use of TextContents to analyze WikipediaData, extracting and visualizing notable persons and their occupations.

http://www.wolfram.com/language/12/natural-language-processing/harvest-entities-in-text.html?product=mathematica

Example: Find dates in text

Using TextCases to extract dates and corresponding sentences from WikipediaData, to plot on a timeline.

http://www.wolfram.com/language/12/natural-language-processing/find-dates-in-text.html?product=mathematica

Guide: Text Manipulation

Acquisition, normalization and structural manipulation; searching and pattern matching; text analysis; natural language processing and understanding; text generation.

https://reference.wolfram.com/language/guide/ProcessingTextualData.html

References

Rosenzweig, Roy. “Scarcity or Abundance? Preserving the Past in a Digital Era.” *American Historical Review* 108, 3 (June 2003): 735-762.
https://rrchnm.org/essay/scarcity-or-abundance-preserving-the-past-in-a-digital-era/

ch08

Lesson 08. Geospatial Analysis

Raster, vector and attribute data

Information that can be located in space typically comes in one of three broad forms: raster, vector and attribute data. Raster data consists of rows and columns of cells, comparable to digital images. A raster dataset can be scaled, rotated, stretched and deformed in various ways. We will learn some applications of this kind of manipulation in a future lesson. Examples of common raster datasets are

things like elevation or satellite imagery.

Vector data comes in three types: point, line and polygon. Data in point form has a single address. It may be specified with latitude and longitude (the peak of Mt Everest), with a street address (your house or The White House) or the name of a city, state or other geographic region. Geospatial data specified as a line is usually represented as a series of points connected by straight lines. In order to represent the centerlines of curving streets and meandering rivers, you need more or less of these points, and as you zoom in, you eventually find the jagged edges. Polygons are used to represent closed forms like county and state boundaries. Since they are two dimensional, polygons have measurable perimeter and area. As with lines, polygons are represented as a series of straight segments.

Often you also have attribute data stored in the form of tables or other data structures. The number of people living in a city or county or the number of registered voters living at a particular address are examples of attribute data. As you work through the lesson, try to identify the different kinds of geographic data involved in the various examples.

Maps and texts

Clearing all definitions

In[•]:=
```
Clear["Global`*"]
```

Dealing with ambiguous place names

One of the challenges of working with natural language is that it is often ambiguous. We know that *The Simpsons* takes place in the United States in a town called Springfield, for example, but it is a running joke that we never learn which state it is in. There are many Springfields in the US.

The SemanticInterpretation command tries to assign the most probable meaning to a string, returning an Entity.

In[•]:=
```
SemanticInterpretation[{"big apple", "city of light", "city of brotherly love"}]
```

If we ask it to make its best guess for the input “Springfield”, it returns Springfield, Massachusetts.

In[•]:=
```
defaultSpringfield = SemanticInterpretation["Springfield"];
```

In[•]:=
```
defaultSpringfield["AdministrativeDivision"]
```

In this case, we would like to know all the alternatives. For this we use the AmbiguityFunction.

In[•]:=
```
springfieldList = SemanticInterpretation["Springfield", AmbiguityFunction → All]
```

Here are the Springfields that *Mathematica* knows about. All are in the US except for one in Manitoba, some are neighborhoods rather than cities, and three have hyphenated names.

In[•]:=
```
TableForm[InputForm /@ First[springfieldList]]
```

We can use Cases to pull out the Springfields that are cities, are in the US, and do not have hyphenated names.

```
In[•]:= springfields =
          Cases[First[springfieldList], Entity["City", {"Springfield", _, "UnitedStates"}]];

In[•]:= InputForm /@ springfields
```

Now we can plot them with GeoListPlot.

```
In[•]:= GeoListPlot[springfields]
```

Plotting locations in texts

We have seen that the TextCases and TextContents commands allow us to discover the names of geographic entities in texts. Here we retrieve the Wikipedia article on Silicon Valley and extract the names of cities. The Select command allows us to pull out the cities that have matching entities.

```
In[•]:= siliconValleyText = WikipediaData["Silicon Valley"];

In[•]:= cityList =
         Select[TextCases[siliconValleyText, "City" → "Interpretation"], Head[#] == Entity &]
```

We check the interpretations of the Entities to make sure that the correct city has been identified. Here is one way to find the city entities that are not in California.

```
In[•]:= cityList ~ Complement ~
          Cases[cityList, Entity["City", {_, "California", "UnitedStates"}]] // InputForm
```

Some of these are incorrectly identified and should be replaced by cities in California. In order to change these, we are going to make use of the ReplaceAll command, which lets us use a list of rules to transform an expression. First we need to make a list of the transformations we want.

```
In[•]:= transformList = {
          Entity["City", {"Marysville", "Michigan", "UnitedStates"}] →
           Entity["City", {"Marysville", "California", "UnitedStates"}],
          Entity["City", {"Menlo", "Georgia", "UnitedStates"}] →
           Entity["City", {"Menlo", "California", "UnitedStates"}],
          Entity["City", {"Moffett", "Oklahoma", "UnitedStates"}] →
           Entity["City", {"Moffett", "California", "UnitedStates"}],
          Entity["City", {"SantaClara", "VillaClara", "Cuba"}] →
           Entity["City", {"SantaClara", "California", "UnitedStates"}]};
```

Now we can use the ReplaceAll command, typically written in shorthand with a slash dot.

```
In[•]:= cleanedCityList = cityList /. transformList;
```

The counts represent the number of times each city is mentioned in the article.

```
In[•]:= Counts[cleanedCityList]
```

Now we can use GeoBubbleChart to plot the locations. Any entity in our list that is not recognized will not be displayed in a tan box with rounded edges. It will be ignored by the mapping commands.

```
In[•]:= GeoBubbleChart[Counts[cleanedCityList]]
```

We can select the cities in California first, if we wish to focus on those. I've used the GeoRangePadding option to provide a bit of geographical context.

```
In[•]:= GeoBubbleChart[
 Counts@Cases[cleanedCityList, Entity["City", {_, "California", "UnitedStates"}]],
 GeoRangePadding → Quantity[100, "Miles"]]
```

Religions and nations

Here is a nice example from the *Mathematica* documentation. It shows how to discover and visualize the density of religious adherents in a given region of the world. *Mathematica* has computable data about world religions.

```
In[•]:= EntityProperties["Religion"]
```

The "ChildReligions" property links to branches of major religions.

```
In[•]:= Islam RELIGION ["ChildReligions"]
```

You can request the number of adherents for each branch.

```
In[•]:= ReverseSort[EntityValue[Islam RELIGION ["ChildReligions"],
  "TotalPopulation", "EntityAssociation"]]
```

Given two or more religions and a region of the world, you can plot maps showing relative density of adherents.

```
In[•]:= Column[GeoRegionValuePlot[#, GeoRange → EntityClass["Country", "MiddleEast"],
     GeoRangePadding → {Quantity[1500, "Miles"], Full}, ImageSize → Scaled[0.6]] & /@
  EntityValue[{Entity["Religion", "Shiite"], Entity["Religion", "Sunni"]},
   "FractionOfCountryPopulations"]]
```

Religious texts

Here are the entities for major religions.

```
In[•]:= EntityList[religions RELIGIONS]
```

Look up religious texts with "MajorTexts".

```
In[•]:= Lutheran Church RELIGION ["MajorTexts"]
```

Each of these is a Book entity.

```
In[•]:= Ninety-Five Theses on the Power and Efficacy of Indulgences BOOK ["Dataset"]
```

Exploring place

The buildings of Tallinn

What are the notable buildings in the city of Tallinn, Estonia? The GeoNearest command allows us to find entities of various kinds near one another.

```
In[•]:= tallinnBuildings =
  GeoNearest["Building", Tallinn CITY, {All, Quantity[1, "Kilometers"]}]
```

Plot the buildings on a map with labels.

```
In[•]:= GeoListPlot[tallinnBuildings, GeoLabels → True]
```

We can zoom in on a particular building to see its immediate neighborhood.

```
In[•]:= GeoListPlot[Pikk Hermann BUILDING, GeoRange → Quantity[100, "Meters"]]
```

The GeoPosition command returns latitude and longitude.

```
In[•]:= GeoPosition[Pikk Hermann BUILDING]
```

A walking tour

Find the shortest path between the buildings. Note that the FindShortestTour command returns a list of items, so we use this syntax to assign each part of the output list to a different symbol.

```
In[•]:= {tourDistance, tourPath} = FindShortestTour[GeoPosition /@ tallinnBuildings]
```

Mathematica can even generate travel directions.

```
In[•]:= tallinnWalk = TravelDirections[tallinnBuildings[[tourPath]], TravelMethod → "Walking"]
```

This is a relatively complicated dataset.

```
In[•]:= tallinnWalk["Dataset"]
```

We can extract various parts of the dataset to visualize. Here is one of the paths.

```
In[•]:= Labeled[GeoListPlot[tallinnWalk["Dataset"][[13, 7]],
    GeoRangePadding → Quantity[30, "Meters"]], tallinnWalk["Dataset"][[13, 1]]]
```

What am I seeing?

We can look up Entities directly in Wikipedia.

```
In[•]:= WikipediaData[Pikk Hermann BUILDING, "SummaryPlaintext"]
```

```
In[•]:= Estonia COUNTRY [flag]
```

Use GeoNearbyDataset to find Wikipedia articles related to a particular place.

```
In[•]:= nearbyArticles = WikipediaData[ Pikk Hermann BUILDING , "GeoNearbyDataset"]
```

We can plot these articles on a map around the building. We use Tooltip so when we hover the mouse over an icon, a popup window displays the title of the article. Just north of Pikk Hermann (in the center of the map) is a marker for Toompea Castle. Hover over it to see the title. What is the marker to the east of Pikk Hermann?

```
In[•]:= GeoGraphics[
  {Tooltip[GeoMarker[#["Position"], "Color" → Blue, "Scale" → Scaled[.04]],
      #["Title"]]} & /@ (Normal[nearbyArticles])]
```

Geocoded photographs

Here is a photograph of Toompea Castle, taken by Sergei Gussev on April 17, 2017 and shared on Flickr.

```
In[•]:= toompeaCastleImage = Import[
    "https://web.archive.org/web/20190527123143/https://live.staticflickr.com/2831/
      33987372191_b75e885920_b_d.jpg", "ImageWithExif"]
```

I imported the file with Exif (Exchangeable Image File Format) metadata. Using the Options command, we can see that this image was taken with an iPhone 5s.

```
In[•]:= Options[toompeaCastleImage]
```

We can extract the GPS information from the photo and plot its location on the maps as follows. Note that when you hover over the map, you can see a thumbnail image of the building.

```
In[•]:= GeoPosition[toompeaCastleImage]
```

```
In[•]:= GeoGraphics[{GeoPosition[toompeaCastleImage], PointSize[Large],
    Tooltip[Point[GeoPosition[toompeaCastleImage]], Thumbnail[toompeaCastleImage]]},
  GeoRange → Quantity[500, "Meters"]]
```

We can also request high resolution satellite imagery. Here is the area surrounding the Tallinn TV Tower.

```
In[•]:= GeoGraphics[ Tallinn TV Tower BUILDING , GeoBackground → "Satellite",
  GeoServer → "DigitalGlobe", GeoRange → Quantity[300, "Meters"]]
```

Geographic information

Choropleth maps

We have already seen that *Mathematica* can display spatial data counts in the form of scaled bubbles, with the GeoBubbleChart command. Another common way to represent quantitative information is in the form of a choropleth map, which uses shading or coloring to represent the measurement of a statistical variable.

Here are the properties of the HistoricalSite entity.

```
In[•]:= EntityProperties["HistoricalSite"]
```

Here is how we get a list of historical sites in Canada.

```
In[•]:= canadianHistoricalSites =
          EntityList[EntityClass["HistoricalSite", {"Country" → Canada COUNTRY}]];
```

Here are five of them, selected at random. How can you tell how many there are in total?

```
In[•]:= RandomSample[canadianHistoricalSites, 5]
```

Next we compute a list of their locations.

```
In[•]:= canadianHistoricalSiteLocations = GeoPosition[canadianHistoricalSites];
```

The GeoHistogram command creates a grid of bins (250km square in this case) overlaying the country, and counts the number of sites that fall into each of the bins. The more sites in a particular bin, the more intense the color.

```
In[•]:= canadianBins = GeoHistogram[canadianHistoricalSiteLocations,
          {"Rectangle", Quantity[250, "Kilometers"]}, ImageSize → Scaled[0.6],
          GeoRange → Canada COUNTRY, PlotLegends → Automatic,
          ColorFunction → "Rainbow", GeoBackground → "CountryBorders"]
```

Here are the actual locations of each of the sites.

```
In[•]:= canadianPoints =
          GeoGraphics[{Blue, Opacity[0.5], Point[canadianHistoricalSiteLocations]},
          ImageSize → Scaled[0.6], GeoRange → Canada COUNTRY,
          GeoBackground → "CountryBorders"]
```

Mathematica allows us to combine multiple graphic images, including maps, with the Show command.

```
In[•]:= Show[canadianBins, canadianPoints, ImageSize → Scaled[0.6],
          GeoRange → Canada COUNTRY, GeoBackground → "CountryBorders"]
```

An elevation map

This example is adapted from the *Mathematica* documentation.

http://www.wolfram.com/language/gallery/make-an-elevation-map/

Whistler is a popular skiing destination in British Columbia. The GeoElevationData command returns a grid of elevations, which ReliefPlot draws as a raster image.

```
In[•]:= ReliefPlot[QuantityMagnitude[GeoElevationData[Whistler CITY]]]
```

If we would like to see topographic contours, we can use the ListContourPlot command.

```
In[•]:= ListContourPlot[
  QuantityMagnitude[GeoElevationData[Whistler CITY ✓]], Contours → 20]
```

A population map

This distribution of historical sites is not surprising, since most Canadians live close to the border with the United States. Here are the largest cities in Canada.

```
In[•]:= canadianLargest = CityData[{Large, "Canada"}]
```

And their associated populations.

```
In[•]:= canadianLargestPopulations =
  Rule @@@ EntityValue[canadianLargest, {"Entity", "Population"}]
```

This is in a form which we can plot directly with GeoBubbleChart.

```
In[•]:= GeoBubbleChart[canadianLargestPopulations]
```

Historical crime trends in the US

This example is a simplified version of one in the Mathematica documentation. It is an interactive display of crime rate trends in the United States.

Statistics for aggregate crime rates can be retrieved by using Select with EntityProperties. Note that we use a pure function to pull properties that have the string “CrimeRate” as part of their name.

```
In[•]:= usCrimeRates = Select[EntityProperties["AdministrativeDivision"],
    StringContainsQ[CanonicalName[#], "CrimeRate"] &]
```

These can be plotted for a particular region with DateListPlot. The use of the Dated command here says we want data for all available dates.

```
In[•]:= DateListPlot[Alaska, United States ADMINISTRATIVE DIVISION ✓[
    Dated[usCrimeRates, All], "PropertyAssociation"],
  PlotLegends → {"Total crime", "Property crime", "Violent crime"},
  ImageSize → Scaled[0.6]]
```

We can also retrieve rates for specific crimes. Here are recent thefts of motor vehicles in Atlanta, GA. The GeoRegionValuePlot creates a choropleth for geographic polygons, and the GeoDisk command says we want to focus on the region within a disk of 25 miles around the city.

```
In[•]:= GeoRegionValuePlot[Select[
  EntityValue[GeoEntities[GeoDisk[Atlanta CITY, Quantity[25, "Miles"]], "City"],
   "MotorVehicleTheftRate", "EntityAssociation"],
  QuantityQ], ImageSize → Scaled[0.6]]
```

We can also create a map of total crime rates for each of the states in the Continental US, again using GeoRegionValuePlot.

```
In[•]:= states = EntityList[Continental US states ADMINISTRATIVE DIVISIONS];
```

```
In[•]:= statesCrimeRates = Rule @@@ EntityValue[states, {"Entity", "CrimeRate"}];
```

```
In[•]:= GeoRegionValuePlot[statesCrimeRates]
```

These are contemporary data. *Mathematica* also has historical data that we can access with the Dated command. Here is how we retrieve both the current total crime rate in California, and the rate for 1970.

```
In[•]:= EntityValue[California, United States ADMINISTRATIVE DIVISION, "CrimeRate"]
```

```
In[•]:= EntityValue[California, United States ADMINISTRATIVE DIVISION, Dated["CrimeRate", 1970]]
```

Now for the interactive part. *Mathematica* has many commands that allow you to build interactive interfaces directly into your notebook. We have seen a little bit of this functionality in commands like TableForm, Pane and TabView, but those pale in comparison to the Manipulate command.

One way to get a sense of the power of Manipulate is to compare it with Table. In Table, we step through a series of values with an iterator and produce an output at each step. Here is a Table to create a list of years between 1990 and 2015.

```
In[•]:= Table[year, {year, 1990, 2015, 1}]
```

In Manipulate, we also have something like an iterator, but it is associated with an interface element like a slider that we can manipulate in real time, recomputing the output as we do so. Here is an interface where we adjust the slider to choose a year between 1990 and 2015.

```
In[•]:= Manipulate[year, {year, 1990, 2015, 1}]
```

Now suppose we had a map for each of those years... Be sure to click the little plus sign at the right end of the year slider, to access more controls.

```
In[•]:= Manipulate[
 GeoRegionValuePlot[
  Rule @@@ EntityValue[states, {"Entity", Dated["CrimeRate", year]}],
  PlotLabel → "Crime Rate in " <> ToString[year]],
 {year, 1990, 2015, 1}, SaveDefinitions → True]
```

The past is a foreign country

The Axial Age

Mathematica has an extensive collection of historical entities, some of which we have already seen. Here we use the concept of the Axial Age to explore historical geospatial information.

```
In[•]:= WikipediaData["Axial Age", "SummaryPlaintext"]
```

```
In[•]:= Karl Jaspers PERSON [
  {"Image", "BirthDate", "BirthPlace", "DeathDate", "DeathPlace"}]
```

Overlapping historical periods

We can find historical period entities which overlap with the Axial Age, which has an approximate date range of -800 to -200.

We use the FilteredEntityClass command in conjunction with EntityFunction to limit our search to historical periods that started before the end date of the Axial Age and ended after the start date.

```
In[•]:= overlapping = EntityList[FilteredEntityClass["HistoricalPeriod", EntityFunction[hp,
      hp["StartDate"] < DateObject[{-200}] && hp["EndDate"] > DateObject[{-800}]]]]
```

TimeLinePlot helps us to visualize some of the periods in question. Asia.

```
In[•]:= TimelinePlot[
  {{Interval[{DateObject[{-800}], DateObject[{-200}]}]}, {Qin Dynasty HISTORICAL PERIOD,
    Jōmon period HISTORICAL PERIOD, Yayoi period HISTORICAL PERIOD,
    Indian Vedic period HISTORICAL PERIOD, Asian Bronze Age HISTORICAL PERIOD,
    Spring and Autumn Period HISTORICAL PERIOD, Warring States Period HISTORICAL PERIOD,
    Western Han Dynasty HISTORICAL PERIOD, Western Zhou Dynasty HISTORICAL PERIOD}},
  Spacings → {0, 1}, PlotStyle → {Directive[Blue, AbsoluteThickness[5]], Red}]
```

Near East and Europe.

```
In[•]:= TimelinePlot[{{Interval[{DateObject[{-800}], DateObject[{-200}]}]},
    {Ancient Greek Period HISTORICAL PERIOD, Ancient Mesopotamian Period HISTORICAL PERIOD,
     Iron Age HISTORICAL PERIOD, Golden age of Perikles HISTORICAL PERIOD,
     Aegean Bronze Age HISTORICAL PERIOD, European Bronze Age HISTORICAL PERIOD,
     Hellenistic Greek Period HISTORICAL PERIOD, Classical Greek Period HISTORICAL PERIOD,
     Archaic Greek Period HISTORICAL PERIOD}}, Spacings → {0, 1},
  PlotStyle → {Directive[Blue, AbsoluteThickness[5]], Orange}]
```

Americas.

```
In[•]:= TimelinePlot[{{Interval[{DateObject[{-800}], DateObject[{-200}]}]},
    {Early Horizon period HISTORICAL PERIOD, Mesoamerican Pre-Classic period HISTORICAL PERIOD}},
  Spacings → {0, 1}, PlotStyle → {Directive[Blue, AbsoluteThickness[5]], Green}]
```

The largest historical countries in a year

Here are the 10 largest historical countries with polygons in the year 600 BCE.

```
In[•]:= largestCountries600BCE =
  TakeLargest[GeoArea /@ EntityValue[EntityClass["HistoricalCountry",
      {"ContemporaryCountriesForYearWithPolygon", -600}],
     "Polygon", "EntityAssociation"], 10]
```

All in one map.

```
In[•]:= GeoGraphics[{Red, EdgeForm[Black], DeleteMissing@
    EntityValue[Keys[largestCountries600BCE], Dated["Polygon", -600]]}]
```

As separate maps.

```
In[•]:= GeoGraphics[{Red, EdgeForm[Black],
   EntityValue[Gandhara grave culture HISTORICAL COUNTRY, Dated["Polygon", -600]],
   GeoBackground → "CountryBorders"}]
```

```
In[•]:= GeoGraphics[{Red, EdgeForm[Black],
   EntityValue[Neo-Babylonian Empire HISTORICAL COUNTRY, Dated["Polygon", -600]],
   GeoBackground → "CountryBorders"}]
```

Religious and philosophical thought

Here are some of the thinkers and schools of thought that Jaspers identified as being characteristic of the Axial Age.

In[•]:= Golden age of Perikles HISTORICAL PERIOD ["PeopleInvolved"]

In[•]:= Homer PERSON ["BirthDate"]

In[•]:= Heraclitus PERSON ["BirthDate"]

In[•]:= Socrates PERSON ["BirthDate"]

In[•]:= Plato PERSON ["BirthDate"]

In[•]:= Thucydides PERSON ["BirthDate"]

In[•]:= Archimedes PERSON ["BirthDate"]

In[•]:= Buddhism RELIGION [{"Founders", "FoundingDate"}]

In[•]:= Jainism RELIGION [{"Founders", "FoundingDate"}]

In[•]:= Taoism RELIGION [{"Founders", "FoundingDate"}]

In[•]:= Taoism RELIGION ["MajorPeople"]

In[•]:= Zhuangzi PERSON ["BirthDate"]

In[•]:= Confucianism RELIGION [{"Founders", "FoundingDate"}]

Further examples

The Fall of Rome

Since we have dated polygons for historical countries, and *Mathematica* is able to compute the area of those countries with GeoArea, we can also plot the changing area of a historical country over time. Here is one example from the Mathematica documentation. Note the precipitous loss in territory for the Roman Empire in the middle of the 200s CE.

```
In[•]:= DateListPlot@MapAt[GeoArea,
          Entity["HistoricalCountry", "RomanEmpire"][Dated["Polygon", All]], {All, 2}]
```

```
In[•]:= WikipediaData["Crisis of the Third Century", "SummaryPlaintext"]
```

The Roman Empire in the year 200 CE.

```
In[•]:= GeoGraphics[Entity["HistoricalCountry", "RomanEmpire"][Dated["Polygon", 200]]]
```

Two of the three competing states in 268 CE.

```
In[•]:= GeoGraphics[{{{Red, EdgeForm[Black], EntityValue[Roman Empire HISTORICAL COUNTRY, Dated["Polygon", 268]]}, {Blue, EdgeForm[Black],
    EntityValue[Gallic Empire HISTORICAL COUNTRY, Dated["Polygon", 268]]}
  }]
```

At the time of writing (May 2019) there wasn't an Entity for the Palmyrene Empire, but here is a map from the Wikimedia Commons.

```
In[•]:= Import[
 "https://upload.wikimedia.org/wikipedia/commons/thumb/6/67/Palmyrene_Empire.png/
   704px-Palmyrene_Empire.png"]
```

There are also nice animated and granular examples of the Roman Empire in the *Mathematica* documentation.

http://www.wolfram.com/language/11/knowledgebase-expansion/animate-the-rise-and-fall-of-historical-countries.html?product=mathematica

http://www.wolfram.com/language/12/units-dates-and-uncertainty/track-the-fall-of-rome-in-granular-steps.html?product=mathematica

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 21-24.

Workflow: Use the Manipulate Interface

https://reference.wolfram.com/language/workflow/UseTheManipulateInterface.html

Workflow: Build a Manipulate

Step by step instructions for building a manipulable interface, using font formatting as an example.

https://reference.wolfram.com/language/workflow/BuildAManipulate.html

Example: Geo Imagery at Any Scale

Satellite imagery available in *Mathematica* ranges from planetary scales to 1m / pixel.

http://www.wolfram.com/language/12/new-in-geography/geo-imagery-at-any-scale.html?product=-mathematica

Example: The Age of Local Bridges

These two examples show how to map the bridges of Pittsburgh with picture tooltips and then to visualize their ages, using opening dates associated with each entity.

http://www.wolfram.com/mathematica/new-in-10/geographic-visualization/map-the-bridges-of-pittsburgh.html

http://www.wolfram.com/language/12/geographic-entities/visualize-the-age-of-local-bridges.html?product=mathematica

Example: The Trees of Champaign, Illinois

This series of examples shows how to plot and visualize a dataset of trees in the city of Champaign, Illinois.

Import the dataset and visualize the most common trees.

http://www.wolfram.com/mathematica/new-in-10/semantic-data-import/identify-the-most-common-trees-in-a-city.html

Map the density of trees with GeoHistogram.

http://www.wolfram.com/language/11/enhanced-geo-visualization/measure-the-density-of-trees.html?product=mathematica

Map the density with GeoSmoothHistogram.

http://www.wolfram.com/language/12/geographic-visualization/opacityfunction-and-colorfunction.html?product=mathematica

Example: Crime Mapping

A series of examples of crime mapping that expand on what we did in this lesson.

http://www.wolfram.com/mathematica/new-in-10/entity-based-geocomputation/compare-ratios-of-crime-and-voting-in-the-united-s.html

http://www.wolfram.com/language/11/enhanced-geo-visualization/demographic-information.html?product=mathematica

http://www.wolfram.com/language/12/financial-and-socioeconomic-entities/explore-historical-crime-trends-in-the-united-states.html?product=mathematica

References

Gregory, Ian N. & Alistair Geddes, eds. *Toward Spatial Humanities: Historical GIS and Spatial History*. Indiana University Press, 2014.
http://www.iupress.indiana.edu/product_info.php?products_id=807086

ch09

Lesson 09. Images

Computer vision

Mathematica has a wide range of commands for dealing with images, from low-level operations that manipulate pixels and image regions to very high-level ones that can identify conceptual entities in images. Here we mostly focus on the higher-level end of the spectrum, drawing on a range of machine learning and computer vision techniques to identify faces and entities, extract hidden information, and search, cluster and classify images.

Faces and fakes

Clearing all definitions

In[•]:=
```
Clear["Global`*"]
```

Detecting faces

This 1945 photograph of Truman with Attlee and Stalin at Potsdam is from the Harry S Truman Presidential Library and Museum.

https://www.trumanlibrary.org/photographs/view.php?id=1777

As usual, we will work with an archived copy.

In[•]:=
```
potsdamPhoto = Import[
    "https://web.archive.org/web/20180521203337/https://www.trumanlibrary.org/
      photographs/58-455.jpg"];
```

The Show command allows us to display an image at different scales in our notebook.

In[•]:=
```
Show[potsdamPhoto, ImageSize → Large]
```

The image itself is not resized. We can find the dimensions of the image with ImageDimensions.

In[•]:=
```
ImageDimensions[potsdamPhoto]
```

The FindFaces command detects the faces in an image. If we give it the “Image” property, it returns a list of images.

In[•]:=
```
potsdamFaceList = FindFaces[potsdamPhoto, "Image"]
```

The default output of the FindFaces command is a list of Rectangles that locate each face in pixel coordinates.

In[•]:=
```
potsdamPhotoFaces = FindFaces[potsdamPhoto];
```

In[•]:=
```
First@potsdamPhotoFaces
```

With this information we can use the HighlightImage command to display the faces in context.

```
In[•]:= HighlightImage[potsdamPhoto, potsdamPhotoFaces]
```

Front row, left to right: Prime Minister Clement Attlee, President Harry S. Truman, General Joseph Stalin. Back row, left to right: Admiral William Leahy, Ernest Bevin, James F. Byrnes, and Vyacheslav Molotov.

Facial features

The FacialFeatures command uses computer vision to attempt to identify age, gender and emotional state.

```
In[•]:= facialFeatureDataset = Dataset@FacialFeatures[potsdamPhoto]
```

How well did it do? Obviously the gender is correct, and we can't be sure of the emotional state of the men. We can check the age estimates, however.

```
In[•]:= birthYears = {"Ernest Bevin" → 1881, "Joseph Stalin" → 1878,
    "Clement Attlee" → 1883, "Vyacheslav Molotov" → 1890,
    "William D Leahy" → 1875, "James F Byrnes" → 1882, "Harry S Truman" → 1884};
```

```
In[•]:= actualAges = Map[1945 - #[[2]] &, birthYears]
```

Here are the differences between actual and estimated ages. The estimates are about a decade too young.

```
In[•]:= actualAges - Normal[facialFeatureDataset[[All, 2]]]
```

```
In[•]:= Median[actualAges - Normal[facialFeatureDataset[[All, 2]]]]
```

Identifying notable people

The *Mathematica* documentation includes a little function to identify which notable person is in a picture, using the machine learning 'superfunction' Classify. (We will learn more about this command in future lessons.)

```
In[•]:= notableIdentify[image_] :=
 Row[{image, Column@Classify["NotablePerson", image, {"TopProbabilities", 3}]}]
```

Here is Stalin's image from the Potsdam photo.

```
In[•]:= potsdamFaceList[[2]]
```

Stalin is among its top three choices, but the system isn't very confident of its judgment here.

```
In[•]:= notableIdentify[potsdamFaceList[[2]]]
```

It is better at identifying people from the more recent past.

```
In[•]:= notableIdentify[ Margaret Thatcher PERSON [ image ] ]
```

```
In[•]:= notableIdentify[ Hillary Clinton PERSON [ image ] ✓ ]
```

The disappearing commissar

Commissar removed from picture of Stalin ca. 1930, from this page

https://web.archive.org/web/20180527051147/http://pth.izitru.com/1930_ 13_ 00.html

```
In[•]:= stalinCommissar = Import[
          "https://web.archive.org/web/20151114184054/http://static1.1.sqspcdn.com/static
            /f/905879/13024345/1309722188997/c1930-Stalin.jpg?token=1
            T9ldOxXGP6Vma91aMIglT6u%2FLU%3D"];
```

```
In[•]:= Show[stalinCommissar, ImageSize → Medium] // Framed
```

Note from the frame that both photographs have been placed side by side in a single digital image.

In order to compare these two photographs, we will use the ImageTrim command to crop out subimages. The coordinates are a rectangle with lower left corner {xmin, ymin} and upper right corner {xmax,ymax}. (There is a workflow in the “Learning More” section of this lesson that tells you how to find the pixel coordinates of interest in cases like this.) We visualize the regions we are clipping first, then do the cropping.

```
In[•]:= HighlightImage[stalinCommissar, Rectangle[{4, 2}, {350, 232}]]
```

```
In[•]:= HighlightImage[stalinCommissar, Rectangle[{373, 2}, {719, 232}]]
```

Now we can do the cropping. This results in two images, with a photo in each.

```
In[•]:= stalinBefore = ImageTrim[stalinCommissar, {{373, 2}, {719, 232}}]
```

```
In[•]:= stalinAfter = ImageTrim[stalinCommissar, {{4, 2}, {350, 232}}]
```

Here are the image dimensions after cropping. We want them to be the same.

```
In[•]:= {ImageDimensions[stalinBefore], ImageDimensions[stalinAfter]}
```

Now we can use the ImageDifference command to study differences in the two images.

```
In[•]:= ImageDifference[stalinAfter, stalinBefore]
```

This makes it easier to see that not only was the commissar removed, but that Stalin lost some weight in the process.

Image mining

Flickr Commons

The Flickr Commons is a large collection of images from the public photography archives of the world. Participating institutions include the Library of Congress, the British Library, the US National Archives, the Smithsonian Institution, NASA, the New York Public Library, the Getty Research Institute, and dozens of others, including archives from Canada, Brazil, Ireland, Scotland, the Netherlands, Finland,

Norway, Denmark, Sweden, Australia and New Zealand.

https://www.flickr.com/commons

ImageCases

Much like TextContents, the ImageCases command uses sophisticated artificial intelligence techniques to identify and label entities in photographs.

Here is a 1938 image of the town-crier announcing the news on the island of Terschelling, the Netherlands.

```
In[•]:= townCrier =
         Import["https://live.staticflickr.com/3206/3280639091_ac64768d41_o_d.jpg"]
```

ImageCases identifies two kinds of entities in the image and returns an Association of them.

```
In[•]:= ImageCases[townCrier]
```

Here is another example. Capel Street in Dublin, Ireland at 9:30am on June 28, 1960.

```
In[•]:= capelStreet =
         Import["https://live.staticflickr.com/8164/7459404340_22ac996ff0_b_d.jpg"]
```

```
In[•]:= ImageCases[capelStreet]
```

Here is an advertisement from the April 1948 issue of The Ladies' Home Journal.

```
In[•]:= advertisement =
         Import["https://live.staticflickr.com/2913/14766523142_e771f7fafa_b_d.jpg"]
```

```
In[•]:= ImageCases[advertisement]
```

Concepts and WordNet

The entities that ImageCases identifies are Concept entities. Let's get a random concept entity to explore.

```
In[•]:= rndEnt = RandomEntity["Concept"]
```

Every concept has a number of available properties, often including relationships to broader and narrower categories.

```
In[•]:= rndEnt["Dataset"]
```

Here are some other sample Concept entities and properties.

```
In[•]:= EntityValue["Concept", "SampleEntities"]
```

```
In[•]:= dachshund CONCEPT ["BroaderConcepts"]
```

```
In[•]:= hunting dog CONCEPT ["BroaderConcepts"]
```

```
In[•]:= cattle CONCEPT ["NarrowerConcepts"]
```

How many Concept entities does *Mathematica* know about?

```
In[•]:= EntityValue["Concept", "EntityCount"]
```

Querying images with concepts

If we are looking for something specific, we can use a concept entity to query an image. Here is a photo of a fruit and vegetable stand in Miami, Florida circa 1980.

```
In[•]:= fruitStand =
         Import["https://live.staticflickr.com/8434/7629968464_6e194eb78b_o_d.jpg"]
```

Find the bananas.

```
In[•]:= ImageCases[fruitStand, banana CONCEPT]
```

```
In[•]:= HighlightImage[fruitStand, ImageCases[fruitStand, banana CONCEPT → "BoundingBox"]]
```

Identifying images

The ImageIdentify command identifies common entities in photographs. Here are examples from the Internet Archive's Book Image collection in Flickr Commons.

```
In[•]:= strawberry =
          Import["https://live.staticflickr.com/8604/16463613907_beec556dc9_b_d.jpg"];
```

```
In[•]:= apple =
          Import["https://live.staticflickr.com/5593/14747851031_d2441f2032_b_d.jpg"];
```

```
In[•]:= pear = Import["https://live.staticflickr.com/3909/14782572692_a4b88a23d8_b_d.jpg"];
```

```
In[•]:= squirrel =
          Import["https://live.staticflickr.com/5560/14564396540_7f0004fbd2_b_d.jpg"];
```

```
In[•]:= cat = Import["https://live.staticflickr.com/2896/14596154550_a137c1dea5_b_d.jpg"];
```

We map the Thumbnail command across the list of images to see thumbnails of each.

```
In[•]:= Thumbnail /@ {strawberry, apple, pear, squirrel, cat}
```

When we map ImageIdentify across the list of images, we see that it does very well with these images.

```
In[•]:= ImageIdentify /@ {strawberry, apple, pear, squirrel, cat}
```

Latent information

Image processing sometimes also allows us to discover information that is hidden in imagery. Here are some examples.

Revealing history

This example comes from the Nebraska History Museum. While digitizing and studying the pho-

tographs of Solomon D. Butcher, scholars at the Nebraska State Historical Society used image processing techniques to reveal details that have not been seen since the negatives were exposed.

https://history.nebraska.gov/collections/revealing-history

Here is a photo of a grocery store in Overton, NB, 1904.

```
In[•]:= groceryStore = Import[
          "https://history.nebraska.gov/sites/history.nebraska.gov/files/collections-img/
            13142_0.jpg"]
```

Zooming in on the doorway and adjusting the contrast shows the stocked shelves inside the store.

```
In[•]:= doorway = 
```

```
In[•]:= ImageResize[ImageAdjust[doorway, {0, 0, .28}], 300]
```

Seances

This example comes from Devon Elliott. The Hamilton Family Fonds at the University of Manitoba contains a number of seance photographs taken in the early twentieth century, including some stereoscopic pairs.

```
telekinesis26f = Import[
    "https://devonelliottdotnet.files.wordpress.com/2013/09/um_pc012_a79-041_006_
      0001_026_0006-tif_jpg.jpg"];
```

```
In[•]:= Show[telekinesis26f, ImageSize → Medium]
```

If you create a small movie, alternating the left and right frames rapidly, you end up with a wobble image which gives a sense of depth.

```
In[•]:= teleLeft =
          ImageResize[ImageTrim[telekinesis26f, {{3979.243247391037`, 2148.387277470841`},
             {2076.2133210558623`, 164.9266421117254`}}], 240];
```

```
In[•]:= teleRight =
          ImageResize[ImageTrim[telekinesis26f, {{2065.998004910988`, 2155.0848680171885`},
             {142.83555862492324`, 164.52915899324762`}}], 240];
```

Here the ListAnimate command repeatedly plays the two images at 4 frames per second.

```
In[•]:= ListAnimate[{teleLeft, teleRight}, 4]
```

Palimpsests

This example comes from an article by Jo Marchant in *Smithsonian Magazine*. By photographing ancient manuscripts in high resolution under a variety of infrared, visible and ultraviolet lights (a

technique known as multispectral imaging), archaeologists are able to expose layers of writing underneath the top layer.

Here is what is visible to the naked eye.

```
In[•]:= Import[
 "https://web.archive.org/web/20190525090625im_/https://thumbs-prod.si-cdn.com/
   BwG_97S9n3lo7ti2guCVSa9MAHw=/fit-in/1072x0/https://public-media.si-cdn.com/
   filer/6a/f6/6af6864d-3749-47ed-9a7d-e3e4a757aa5e/0050_000001_psh_color-wr.jpg"
   ]
```

Here is the image photographed with an alternate light source.

```
In[•]:= palimpsestALS = Import[
  "https://web.archive.org/web/20190525090625im_/https://thumbs-prod.si-cdn.com/
    Pq2jC-QPj03UoNqSUZi-fgiUoQw=/fit-in/1072x0/https://public-media.si-cdn.com/
    filer/b0/fd/b0fd95ce-1d99-4161-9472-88072aa9067b/0050_000001-syriac_nf-1
    r_bands01-23_rffl_cal_bands_010217-23_undertext_only_ica_r_ratio2-1_g7_b7-
    wr.jpg"]
```

Using image processing techniques, we can remove the overtext to better see the undertext. The ColorSeparate command separates Red, Green and Blue layers into three channels. Here is the Red channel.

```
In[•]:= ColorSeparate[palimpsestALS][[1]]
```

Feature extraction

A number of commands in *Mathematica* use machine learning to extract features from data. These features can then be used to group the data into clusters.

FeatureSpacePlot

Here are images of a number of Pokemon.

```
In[•]:= pokemonImages = #["Image"] & /@ EntityValue["Pokemon", "SampleEntities"]
```

The FeatureSpacePlot command extracts features from the images, then visualizes their distribution in two-dimensional space. Images that are close to one another should be visually more similar than images that are far apart from one another.

```
In[•]:= FeatureSpacePlot[pokemonImages, LabelingFunction → Callout]
```

Here is another example, from the Mathematica documentation. Given a small collection of paintings (van Gogh, Picasso and 'Aboriginal-style'), FeatureSpacePlot clusters them into different groups.

In[•]:=
```
paintings = {
```

```
};
```

In[•]:=
```
FeatureSpacePlot[paintings, LabelingSize → 50, ImageSize → Scaled[0.6]]
```

These are examples of *unsupervised* machine learning. The system performs clustering based solely on statistical properties of the data.

Content based image retrieval

We can also use feature extraction to create an image search tool. Here is a collection of images of dogs from the following documentation page.

http://www.wolfram.com/language/11/improved-machine-learning/create-an-image-search-tool-using-feature-extracti.html?product=mathematica

```
dogs = {
```

```
};
```

We have already seen we can visualize a dataset like this with FeatureSpacePlot.

In[•]:=
```
FeatureSpacePlot[dogs, LabelingSize → 40, ImageSize → Scaled[.6]]
```

The FeatureNearest command learns a function that, given an image it hasn't seen will return the closest match.

In[•]:=
```
dogsNF = FeatureNearest[dogs]
```

Here are some dogs our system hasn't seen yet.

In[•]:= newDogs = {  ,  ,  };

And here are the ones in the feature space that it thinks are the closest match.

In[•]:= Table[{d → dogsNF[d]}, {d, newDogs}] // TableForm

Potsdam revisited

This example is adapted from the *Mathematica* documentation. The basic idea is that facial features can be used with FeatureSpacePlot to create a space of faces, then we can identify new face images by finding the closest face in the space.

http://www.wolfram.com/language/12/machine-learning-for-images/simple-face-recognition.html?product=mathematica

Here are the people from the first Potsdam Conference photograph.

In[•]:= potsdamFaceList

In[•]:= Keys@birthYears

We can look each up and see if there is a corresponding *Mathematica* entity, and if so, grab an image.

In[•]:= DeleteCases[Table[Interpreter["Person"][p], {p, Keys@birthYears}], _Failure]

In[•]:= #["Image"] & /@
 DeleteCases[Table[Interpreter["Person"][p], {p, Keys@birthYears}], _Failure]

In[•]:= FindFaces[#["Image"], "Image"] & /@
 DeleteCases[Table[Interpreter["Person"][p], {p, Keys@birthYears}], _Failure]

We can also get other photographs taken on the same day, and extract face images from those.

In[•]:= potsdamPhoto2 = Import[
  "https://upload.wikimedia.org/wikipedia/commons/2/2d/Bundesarchiv_Bild_183-R86965%2C_Potsdamer_Konferenz%2C_Gruppenbild.jpg"];

In[•]:= Show[potsdamPhoto2, ImageSize → Medium]

In[•]:= potsdamPhoto2FaceList = FindFaces[potsdamPhoto2, "Image"]

In[•]:= potsdamPhoto3 = Import[
  "https://upload.wikimedia.org/wikipedia/commons/b/b6/Bundesarchiv_Bild_183-14059-0016%2C_Potsdamer_Konferenz%2C_Molotow%2C_Byrnes%2C_Eden.jpg"];

In[•]:= Show[potsdamPhoto3, ImageSize → Small]

In[•]:= FindFaces[potsdamPhoto3, "Image"]

In[•]:= potsdamPhoto4 = Import[
 "https://www.trumanlibrary.org/whistlestop/archive/photos/images/58-574.jpg"]

In[•]:= FindFaces[potsdamPhoto4, "Image"]

```
In[•]:= potsdamPhoto5 = Import["https://media.iwm.org.uk/ciim5/37/915/large_000000.jpg"];

In[•]:= FindFaces[potsdamPhoto5, "Image"]

In[•]:= potsdamPhoto6 = Import[
          "https://upload.wikimedia.org/wikipedia/commons/thumb/7/7e/Bundesarchiv_Bild_
            183-29645-0001%2C_Potsdamer_Konferenz%2C_Stalin%2C_Truman%2C_Churchill.
            jpg/1280px-Bundesarchiv_Bild_183-29645-0001%2C_Potsdamer_Konferenz%2
            C_Stalin%2C_Truman%2C_Churchill.jpg"];

In[•]:= FindFaces[potsdamPhoto6, "Image"]

In[•]:= truman = FindFaces[ImageResize[Import[
            "https://www.whitehouse.gov/wp-content/uploads/2017/12/33_harry_s_truman.jpg"]
           , 200], "Image"]
```

Now that we have all of these face images, we create an association that maps each man to a list of face images.

In[•]:=
```
potsdamFaceAssoc = <|
  "Bevin" → {  ,  ,  },
  "Stalin" → {  ,  ,  ,  ,  ,  },
  "Attlee" → {  ,  ,  ,  },
  "Molotov" → {  ,  ,  ,  ,  ,  },
  "Leahy" → {  ,  ,  },
  "Byrnes" → {  ,  ,  ,  ,  },
  "Truman" → {  ,  ,  ,  ,  }
|>;
```

We compute facial descriptors for each of these images, and then use FeatureSpacePlot to plot the faces in a two dimensional space. For the most part, individuals are clustered in this space.

In[•]:=
```
descriptors = Map[
    FacialFeatures[#, "Descriptor", Method → "FaceBoxes" → Full] &, potsdamFaceAssoc];
```

In[•]:=
```
FeatureSpacePlot[MapThread[Callout[#1, ImageResize[#2, 20]] &,
  {Catenate[descriptors], Catenate[potsdamFaceAssoc]}],
 Sequence[LabelingFunction → None, ImageSize → Large]]
```

Next we use the Classify command to perform classification in this space of facial features.

In[•]:=
```
cl = Classify[descriptors];
```

Here is a photograph from the Potsdam conference that was not used to determine the feature space.

```
In[•]:= testimage = Import[
          "https://web.archive.org/web/20190601204331im_/http://albumwar2.com/wp-content/
            uploads/2017/10/02043-728x579.jpg"];
```

We find the bounding boxes for each face in this test image and label it with the system's best guess as to who the face belongs to.

```
In[•]:= res =
         MapAt[cl, FindFaces[testimage, {"BoundingBox", "Descriptor"}], {All, "Descriptor"}]
```

Finally we highlight and label the faces in the test image. Note that the system only made a single mistake, identifying President Truman as Byrnes.

```
In[•]:= HighlightImage[testimage,
         {#BoundingBox, Inset[Style[#Descriptor, Black, 10, Background → White],
             Last[#BoundingBox], {Right, Bottom}]} & /@ res, ImageSize → Large]
```

Further examples

Tone-mapping algorithms

In the "Latent Information" section of this lesson we looked at one way of displaying some of the information hidden in whose differences are too small to be seen with the naked eye. There are a number of other possibilities, explained on the this documentation page.

http://www.wolfram.com/language/12/new-in-image-processing/arithmetic-for-fast-algorithm-prototyping.html?product=mathematica

Here we apply them to the image of the store doorway. Simple multiplication.

```
In[•]:= 2 ^ Range[0, 4] doorway
```

Nonlinear mapping operator to rescale brightness.

```
In[•]:= doorway / (doorway + .1) // ImageAdjust
```

Transcendental log version.

```
In[•]:= Log[doorway / (doorway + .2)] // ImageAdjust
```

Luminance-only scaling.

```
In[•]:= luminance = ColorSeparate[doorway, "L"];
```

```
In[•]:= (doorway / luminance) ImageAdjust[Log[doorway + 1 / 10]] // ImageAdjust
```

Nonlinear mean of brightness.

```
In[•]:= Mean[Tanh[doorway 2 ^ Range[-3, 3, .4]]] // ImageAdjust
```

Each of these options reveals some details and hides others, so it is worth exploring a range of possibilities.

There's more than one way to do it

If you study the *Mathematica* documentation, you often find traces of earlier, lower-level workflows. For example, in the documentation for *Mathematica* version 8, there is a rudimentary face recognition example that uses image segmentation and color detection.

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/find-faces-in-an-image.html

The earliest FindFaces examples appear in *Mathematica* version 9, but image highlighting is still done at a lower level.

http://www.wolfram.com/mathematica/new-in-9/advanced-image-processing-with-large-image-support/face-detection.html

There is support for machine learning with face images in *Mathematica* version 10.

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/gender-classification.html

The face detection examples in *Mathematica* version 11 include support for entities.

http://www.wolfram.com/language/11/improved-machine-learning/identify-notable-persons.html?product=mathematica

In many cases, the newer, higher-level commands will do exactly what you want, or can be modified to do what you want. But knowing how problems were solved in earlier versions of the language puts more tools at your disposal.

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 25-28.

Workflow: Graphics and Images

Importing, sizing and positioning images and extracting data from them.

https://reference.wolfram.com/language/workflowguide/GraphicsAndImages.html

Workflow: Get Coordinates from an Image

Extract coordinates of points in an image using the coordinates tool in the image toolbar.

https://reference.wolfram.com/language/workflow/GetCoordinatesFromAnImage.html

Code Gallery: Steganography

Steganography is the practice of concealing one kind of message inside another, seemingly innocuous one. There is a nice code example of hiding messages in images in the *Mathematica* documentation.

http://www.wolfram.com/language/gallery/hide-secret-messages-in-images/

See also this blog post by Jon McLoone.

https://blog.wolfram.com/2010/07/08/doing-spy-stuff-with-mathematica/

Tutorial: Image Processing

Learn much more about how images are represented and manipulated in *Mathematica*.

https://reference.wolfram.com/language/tutorial/ImageProcessing.html

References

Elliott, Devon. “Sense of Depth.” DevonElliott.net (9 September 2013). https://devonelliott.net/2013/09/09/sense-of-depth/

Farid, Hany. “Image Forgery Detection: A Survey.” *IEEE Signal Processing Magazine* (March 2009): 16-25. https://www.cs.dartmouth.edu/farid/downloads/publications/spm09.pdf

Kee, Kevin & Timothy Compeau, eds. *Seeing the Past With Computers: Experiments with Augmented Reality and Computer Vision for History*. University of Michigan Press, 2019. https://www.press.umich.edu/9964903/seeing_the_past _with _computers

Marchant, Jo. “Archaeologists Are Only Just Beginning to Reveal the Secrets Hidden in These Ancient Manuscripts.” *Smithsonian Magazine* (11 December 2017). https://www.smithsonianmag.com/history/archaeologoists-only-just-beginning-reveal-secrets-hidden-ancient-manuscripts-180967455/

ch10

Lesson 10. Page Images

Pictures of documents

In the examples of text that we have seen so far, the text has been stored in a human-readable and machine-readable format. In addition to human-readable formats like raw text, HTML and XML, machine-readable text can also be stored in thousands of different file formats like Adobe’s Portable Document Format (PDF), Microsoft *Word* documents or *Excel* spreadsheets, and so on. *Mathematica* can import many of these formats natively.

It is also possible for text to be stored in the form of a digital picture, a *page image*, that was created by scanning or photographing a physical page. If the text on these page images was printed or typed, it can often be converted to machine-readable text by a process known as Optical Character Recognition (OCR). At present, handwritten text cannot be easily extracted from page images, although that may

change in the near future.

People who do archival research now typically return from the archives with (tens of) thousands of digital images of documents. Managing these effectively is an important part of digital research. Rather than scan paper documents, we will download some page images to work with, but the basic principles are the same.

Page images and optical character recognition (OCR)

Clearing all definitions and defining *viewData*

```
In[•]:= Clear["Global`*"]
```

Since we will be looking at digitized documents the *viewData* function will be useful.

```
In[•]:= viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Importing a PDF

The Wilson Center digital library has a copy of a 1948 CIA memorandum to President Truman on Soviet actions in Berlin stored as a PDF.

http://digitalarchive.wilsoncenter.org/document/116188

Rather than use the original, I archived a copy in the Wayback Machine on May 16, 2018. If we use that copy, we don't have to worry about the file changing on the original site.

https://web.archive.org/web/20180516195401/http://digitalarchive.wilsoncenter.org/document/116188.pdf?v=3379f0c03457ecbe104be8895b9a8ea8

PDFs can have multiple layers. What you see when you open the PDF in *Acrobat* or *Acrobat Reader* are the page images. In addition, the PDF can have a layer of searchable (that is, machine-readable) text associated with each page. The easiest way to see if a particular PDF has this layer is to open it in *Acrobat* and try to search for a word you can see on the page. If search doesn't work, the file is missing a text layer.

In *Mathematica*, we begin by using Import and requesting the elements associated with the file.

```
In[•]:= Import[
 "https://web.archive.org/web/20180516195401/http://digitalarchive.wilsoncenter.
   org/document/116188.pdf?v=3379f0c03457ecbe104be8895b9a8ea8", "Elements"]
```

If the file had a text layer, we could use Import to get it, but this fails, giving us an empty string.

```
In[•]:= viewData[Import[
  "https://web.archive.org/web/20180516195401/http://digitalarchive.wilsoncenter.
    org/document/116188.pdf?v=3379f0c03457ecbe104be8895b9a8ea8",
  "Plaintext"]]
```

We can, however, request the number of pages for the document. Note that this includes the Wilson

Center header page.

```
In[•]:= Import[
 "https://web.archive.org/web/20180516195401/http://digitalarchive.wilsoncenter.
   org/document/116188.pdf?v=3379f0c03457ecbe104be8895b9a8ea8", "PageCount"]
```

At present, *Mathematica*'s PDF importing capabilities are somewhat buggy. If you are having trouble with using PDFs, be sure to do a search on mathematica.stackexchange.com to see if there are other techniques that you can take advantage of.

Page images in a PDF

If you are working with traditional printed or typed sources, you can create page images by *digitizing* your sources with a scanner or photographing them with a digital camera. PDFs provide another source of page images.

We can also get all of the page images for our PDF document in a list.

```
In[•]:= ciaMemoPageImages = Import[
    "https://web.archive.org/web/20180516195401/http://digitalarchive.wilsoncenter.
      org/document/116188.pdf?v=3379f0c03457ecbe104be8895b9a8ea8",
    "Images"];
```

We can get a sense of the document by using the ConformImages command to make all of the pages the same height . We can see that Wilson Center graphics were extracted from the cover page, and that there are two typescript pages that will need to be rotated into upright position.

```
In[•]:= ConformImages[ciaMemoPageImages, Small]
```

Text recognition

The TextRecognize command performs OCR on a page image. Sometimes you have to adjust the image to get good results. Here I use the ImageResize, Scaled and Rotate commands to double the size of the page image for page 3 and then rotate it.

```
In[•]:= cleanedCIAMemoPage3Image =
 ImageRotate[ImageResize[ciaMemoPageImages[[3]], Scaled[2]], -90 Degree]
```

Now we can do OCR on the image.

```
In[•]:= viewData[TextRecognize[cleanedCIAMemoPage3Image]]
```

If you look through the OCR results, you will see that they are very rough. Depending on the quality of the page images, the typeface, lighting, and many other factors, OCR can range from excellent to unusable. Whether or not you can use OCR for your own research depends on the nature of your project.

If you use search engines to look for relevant sources in digitized newspapers, books, etc. you should be aware that the quality of the OCR is often no better than this (and sometimes much worse). If an online search doesn't turn up a keyword you can't be sure that the keyword doesn't exist in the sources

you are searching through.

OCRing a cleaner JPG

Not all OCR results are as bad as this, of course! Here is an excerpt of a memo on lunar exploration sent by James E. Webb and Robert McNamara to Vice President Lyndon B. Johnson on May 8, 1961.

```
In[•]:= lunarExplorationMemo = Import[
  "https://web.archive.org/web/20190602211448/https://airandspace.si.edu/sites/
    default/files/styles/slideshow_lg/public/images/5086h.jpg?itok=FyDbkxJU"]
```

```
In[•]:= viewData@TextRecognize[lunarExplorationMemo]
```

Recognition properties

The TextRecognize command can return more than just text. Here we use it on a photograph of a sign, requesting that it return the text, recognition confidence, bounding box, and image.

Photograph of the Vicksburg Cemetery sign in Delhi, MN (July 5, 2006) shared on Flickr by Matt Zaske.

https://www.flickr.com/photos/zaskem/1436348452/

```
In[•]:= cemeterySign =
 Import["https://live.staticflickr.com/1166/1436348452_71b62e3c5e_o.jpg"]
```

Here is the output, formatted as a table. Note that the confidence is higher for the two sections it recognizes properly.

```
In[•]:= TextRecognize[cemeterySign, "Block",
   {"Text", "Strength", "BoundingBox", "Image"}] // TableForm
```

We can get much better OCR if we clip out the portion of the sign containing the description and magnify it.

```
In[•]:= ImageResize[ImageTrim[cemeterySign, {{647, 279}, {414, 57}}], 600] // TextRecognize
```

We can also ask that TextRecognize analyze lines of text rather than blocks. This example is a photograph of a sign from a Dublin park (taken on September 18, 2011), shared on Flickr by Seán Ó Domhnaill.

https://www.flickr.com/photos/an_solas/11485225095/

```
In[•]:= litterSign =
 Import["https://live.staticflickr.com/7372/11485225095_fb66060732.jpg"]
```

```
In[•]:= TextRecognize[litterSign, "Line",
   {"Text", "Strength", "BoundingBox", "Image"}] // TableForm
```

Identifying text blocks

Here we download a single large page image from *Home Protection Exercises*, a 1953 (rev. 1957) publication of the United States Office of Civil Defense.

https://archive.org/details/home-protection-exercises

A folder of the full-size page images is available on the Internet Archive. These were created when the book was digitized by scanning. I've put one of the page images on GitHub so we can work with it easily.

```
In[•]:= homeProtectionExerciseP29 = Import[
          "https://github.com/williamjturkel/Digital-Research-Methods/raw/master/home-
            protection-exercises-p029.jpeg"];
```

Next we use the HighlightImage and TextRecognize commands to draw bounding boxes around all of the recognized text in the page image. Note that part of the bottom figure is incorrectly recognized as a text block.

```
In[•]:= HighlightImage[homeProtectionExerciseP29,
         {"Boundary", TextRecognize[homeProtectionExerciseP29, "Line", "BoundingBox"]}]
```

Extracting figures from page images

Using image processing techniques, we can automatically identify figures on page images and extract them for further processing. I have broken the process into a sequence of steps.

Part 1: Grayscale page

The page as it was scanned.

```
In[•]:= Show[homeProtectionExerciseP29, ImageSize → Medium] // Framed
```

Converted to grayscale.

```
In[•]:= Show[ColorConvert[homeProtectionExerciseP29, "Grayscale"],
          ImageSize → Medium] // Framed
```

Part 2: Binarizing and removing words

Binarized image (this is true black and white rather than grayscale).

```
homeProtectionExerciseP29BWImage =
  MorphologicalBinarize[ColorConvert[homeProtectionExerciseP29, "Grayscale"]];
```

```
Show[homeProtectionExerciseP29BWImage, ImageSize → Medium] // Framed
```

Next we use TextRecognize to put bounding boxes around all words.

```
wordBoundingBoxes =
  TextRecognize[homeProtectionExerciseP29BWImage, "Word", "BoundingBox"];
```

```
Show[HighlightImage[homeProtectionExerciseP29BWImage,
   Complement[wordBoundingBoxes, MaximalBy[wordBoundingBoxes, Area]]],
  ImageSize → Medium] // Framed
```

Now we fill all the word regions with solid white.

```
homeProtectionExerciseP29TextRemoved =
  HighlightImage[homeProtectionExerciseP29BWImage, {Opacity[1], White,
    Complement[wordBoundingBoxes, MaximalBy[wordBoundingBoxes, Area]]}];

Show[homeProtectionExerciseP29TextRemoved, ImageSize → Medium] // Framed
```

Part 3: Bounding boxes for morphological components

We get bounding boxes for morphological components and display the results. In image processing, morphology refers to operations that process the shapes of features that are larger than pixels.

```
In[•]:= homeProtectionExerciseP29BoundingBoxes =
  ComponentMeasurements[DeleteSmallComponents[Dilation[ColorNegate[
      homeProtectionExerciseP29TextRemoved], DiskMatrix[100]]], "BoundingBox"];
```

First figure.

```
In[•]:= Show[ImageTrim[homeProtectionExerciseP29TextRemoved,
    homeProtectionExerciseP29BoundingBoxes[[1, 2]]], ImageSize → Medium] // Framed
```

Second figure.

```
In[•]:= Show[ImageTrim[homeProtectionExerciseP29TextRemoved,
    homeProtectionExerciseP29BoundingBoxes[[2, 2]]], ImageSize → Medium] // Framed
```

Figure classification

A folder of page images

When we have extracted figures from page images, we can use machine learning to classify them into different types for further analysis. In this section, we are going to be using a zipped file of thumbnail images that turned up in a search of Library of Congress holdings that have been classified as *Date: 1950 to 1959* and *Subject: Soviet Union*.

https://www.loc.gov/search/?dates=1950/1959&fa=subject:soviet+union&sp=1

Before you save something to your own file system, you have to make sure to tell *Mathematica* where to put it.

```
In[•]:= Clear[localDownloadFilePath]
```

Make sure to change the following so that it is a valid file path on your own computer!

```
In[•]:= localDownloadFilePath = "/Users/wjt/Downloads/"
```

Now we use the URLSave command to save a local copy of the file.

```
In[•]:= URLSave[
 "https://github.com/williamjturkel/Digital-Research-Methods/blob/master/loc-
   soviet-union-1950s-thumbs.zip?raw=true",
 FileNameJoin[{localDownloadFilePath, "loc-soviet-union-1950s-thumbs.zip"}]]
```

Make sure that the zipped file has downloaded to your machine.

```
In[•]:= FileNames["*.zip", localDownloadFilePath]
```

```
In[•]:= localDownloadFilePath
```

Unzip the file with the ExtractArchive command and list the contents.

```
In[•]:= ExtractArchive[
  FileNameJoin[{localDownloadFilePath, "loc-soviet-union-1950s-thumbs.zip"}],
  localDownloadFilePath] // viewData
```

A collage of the images

Here is the set of images we are working with.

```
In[•]:= sovietUnion1950sThumbsList = Cases[
    Import[FileNameJoin[{localDownloadFilePath, "loc-soviet-union-1950s-thumbs"}],
     "*.jpg"], Except[$Failed]];
```

```
In[•]:= ImageCollage[sovietUnion1950sThumbsList]
```

Training

For *supervised* machine learning, we have to train the learning with a set of labeled examples of what we want it to learn. Then we test it on unlabeled examples.

We are going to use a subset of the images to train our machine learner, telling it which ones are photographs and which are drawings

```
In[•]:= sovietUnion1950sThumbsTrainingSet =
 {sovietUnion1950sThumbsList[[01]] → "Drawing", sovietUnion1950sThumbsList[[02]] →
   "Drawing", sovietUnion1950sThumbsList[[03]] → "Photo",
  sovietUnion1950sThumbsList[[04]] → "Photo", sovietUnion1950sThumbsList[[05]] →
   "Photo", sovietUnion1950sThumbsList[[06]] → "Photo",
  sovietUnion1950sThumbsList[[07]] → "Drawing",
  sovietUnion1950sThumbsList[[08]] → "Drawing", sovietUnion1950sThumbsList[[13]] →
   "Photo", sovietUnion1950sThumbsList[[14]] → "Drawing",
  sovietUnion1950sThumbsList[[15]] → "Drawing", sovietUnion1950sThumbsList[[16]] →
   "Drawing", sovietUnion1950sThumbsList[[17]] → "Drawing"}
```

Now to train a machine learning, all we have to do is call the Classify command.

```
In[•]:= sovietUnion1950sThumbsClassifier = Classify[sovietUnion1950sThumbsTrainingSet]
```

Testing

Now we test our image classifier on the images that we did not use to train it.

```
In[•]:= sovietUnion1950sThumbsTestingSet = Complement[sovietUnion1950sThumbsList, Flatten[
     {sovietUnion1950sThumbsList[[01 ;; 08]], sovietUnion1950sThumbsList[[13 ;; 17]]}]];
```

Get its classification for the first item in the testing set.

```
In[•]:= sovietUnion1950sThumbsTestingSet[[1]] →
         sovietUnion1950sThumbsClassifier[sovietUnion1950sThumbsTestingSet[[1]]]
```

Create a table showing the system's classifications for all items in the testing set.

```
In[•]:= Table[sovietUnion1950sThumbsTestingSet[[i]] →
           sovietUnion1950sThumbsClassifier[sovietUnion1950sThumbsTestingSet[[i]]],
         {i, Range[Length[sovietUnion1950sThumbsTestingSet]]}]
```

The system is working pretty well, despite the fact that we used very little labelled data to train it. In a typical research application, you would use much larger training and testing data sets.

Note that FeatureSpacePlot does a good job of separating drawings from photos, and could be used for unsupervised clustering of the same data.

```
In[•]:= FeatureSpacePlot[sovietUnion1950sThumbsList, LabelingSize → 40]
```

Figure mining

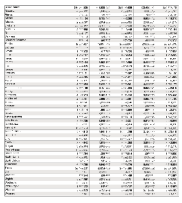
There is an enormous amount of quantitative data trapped in tables and graphs in page images. The *Mathematica* documentation includes some examples of workflows for extracting this information.

Convert an image of a table into CSV

In this example, an image of a table is converted into a human- and machine-readable CSV (comma separated values) file that can be imported into a spreadsheet. (It could also be analyzed in *Mathematica*, of course.)

http://www.wolfram.com/language/12/data-import-and-export/convert-an-image-of-a-table-into-csv.html?product=mathematica

```
In[•]:= tableImage =
```



```
;
```

Using image processing, the grid lines of the table are converted into row and column coordinates that can be handed off to TextRecognize.

```
In[•]:= getGridLines[img_, opts : OptionsPattern[]] :=
          Block[{vals}, vals = ImageData[Binarize[img, .7], opts];
           FindPeaks[1 - (Mean /@ #), 0, 0, .5][[All, 1]] & /@ {vals, Transpose[vals]}];

In[•]:= {tableRows, tableCols} = getGridLines[tableImage, DataReversed → True];

In[•]:= HighlightImage[tableImage,
         {PointSize[.008], Red, Outer[List, tableCols, tableRows]}]
```

```
In[•]:= segmentImage[img_] :=
          Block[{i = Binarize[img], rowPairs, colPairs},
           {rowPairs, colPairs} = Partition[#, 2, 1] & /@ getGridLines[img];
           Outer[ImageTake[i, ##] &, rowPairs, colPairs, 1]];

In[•]:= cleanUp[data_, pats_] := MapAt[StringReplace[pats], data, {All, 2 ;;}];

In[•]:= tableData =
          cleanUp[TextRecognize /@ segmentImage[tableImage], {" " → "", "‘I" -> "1"}];

In[•]:= TextGrid[tableData]

In[•]:= tableDataset = Import[
         Export["census.csv", tableData, "CSV",
          "TableHeadings" -> {"Geography", "2013", "2014", "2015", "2016",
           "2017"}], "Dataset", HeaderLines -> {1, 1}]
```

This dataset is now in a form that can be exported to CSV with the Export command.

Since the first row contains totals, it is possible to test the quality of the OCR by totalling the data and comparing it with the printed totals.

```
In[•]:= tableDataset[["United States"]]

In[•]:= ImageTake[tableImage, 80, {1250, -1}]

In[•]:= tableDataset[[2 ;;]][Total]
```

This example works because the table has been formatted with grid lines, but it is also possible to do similar kinds of processing using the fact that columns are usually aligned vertically (often right justified, as in this example).

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 29-32.

ch11

Lesson 11. Crawling

Interacting with the internet programmatically

Most people’s experience of the internet, and especially of the web, is shaped by the web browsers that they use. We are used to visiting sites, manually clicking on links, typing things into search boxes, and so on. In this lesson we will see some of the ways that we can interact with the internet directly, using *Mathematica* notebooks, and indirectly, using browsers controlled by *Mathematica*. We will also see

how we can use code to automate processes of online interaction, including finding, downloading, indexing and searching files.

In future lessons we will look at a different way of programmatically interacting with the internet, via web services and APIs (application program interfaces).

Browser automation

Clearing all definitions and defining *viewData*

```
In[•]:= Clear["Global`*"]
```

Since we will be looking at digitized documents the *viewData* function will be useful.

```
In[•]:= viewData[x_] :=
         Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

The uniform resource locator (URL)

Mathematica knows about domain names and can give you information about them. Use CTRL= for discovery

```
In[•]:=
```

archive.org

archive.org INTERNET DOMAIN

```
In[•]:= EntityTypeName[archive.org INTERNET DOMAIN]
```

```
In[•]:= EntityProperties["InternetDomain"]
```

```
In[•]:= Entity["InternetDomain", "http://www.archive.org"]["SiteAnnualVisitors"]
```

```
In[•]:= Entity["InternetDomain", "http://www.archive.org"]["SiteViewsPerSecond"] // N
```

You can plot a map showing the physical location of a web server

```
In[•]:= Entity["InternetDomain", "http://www.archive.org"]["Country"]
```

```
In[•]:= GeoGraphics[Entity["InternetDomain", "http://www.archive.org"]["HostLocation"]]
```

Retrieving information from the web

The Import command allows you to retrieve web pages or elements from them directly into your notebook. Start by requesting available Elements.

```
In[•]:= Import["http://www.archive.org", "Elements"]
```

```
In[•]:= Import["http://www.archive.org", "Title"]
```

Here is a list of all the images on the Internet Archive main page.

```
In[•]:= Import["http://www.archive.org", "Images"]
```

And a list of all the hyperlinks.

```
In[•]:= viewData[Import["http://www.archive.org", "Hyperlinks"]]
```

The Import command also allows us to retrieve the plain text of web pages.

```
In[•]:= internetArchivePlaintext = Import["http://www.archive.org", "Plaintext"];
```

We can then apply any of the techniques that we have already learned to analyze that text.

```
In[•]:= WordCloud[internetArchivePlaintext]
```

The WebImage command allows us to capture an image of a website without opening it.

```
In[•]:= Show[WebImage["https://www.archive.org"], ImageSize → Medium]
```

You can use the URLRead command to get lower-level details about the HTTP response.

```
In[•]:= URLRead["archive.org", {"StatusCode", "StatusCodeDescription", "Headers"}]
```

WebExecute

When you use Import, your *Mathematica* notebook is acting as the web client. Using the StartWebSession and WebExecute commands, you can programmatically interact with the Internet through a Chrome or Firefox browser instead.

Start a web session with the Chrome browser (if you don't have Chrome installed, you can change this to Firefox).

```
In[•]:= webSession = StartWebSession["Chrome"]
```

Open a webpage in the browser.

```
In[•]:= WebExecute[webSession, "OpenPage" → "https://archive.org/about/"]
```

Now we can extract HTML elements. Here we ask for the text inside of paragraph tags.

```
In[•]:= webParagraphs = WebExecute[webSession, "LocateElements" → "Tag" → "p"];

        WebExecute[webSession, "ElementText" → webParagraphs] // viewData
```

WebExecute also lets us capture website images and hide elements on the page. Here is the Internet Archive main page.

```
In[•]:= WebExecute[webSession, "OpenPage" → "https://archive.org"]

In[•]:= Show[WebExecute[webSession, "CapturePage"], ImageSize → Medium]
```

Now we hide the images and capture the page again.

```
In[•]:= WebExecute[webSession, "HideElement" → "Tag" → "img"];

In[•]:= Show[WebExecute[webSession, "CapturePage"], ImageSize → Medium]
```

We can also try clicking on a random page. Here are all of the anchor links.

```
In[•]:= webLinks = WebExecute[webSession, "LocateElements" → "Tag" → "a"];
```

Choose one at random and click on it. If it returns Failure when you evaluate the expression, the ran-

domly chosen anchor wasn't clickable. Otherwise it returns Success and the web browser is now on a different page.

In[•]:=
```
WebExecute[webSession, "ClickElement" → RandomChoice[webLinks]]
```

We can see the new page in the notebook by capturing it.

In[•]:=
```
Show[WebExecute[webSession, "CapturePage"], ImageSize → Medium]
```

When we are done with our web session we end it with DeleteObject.

In[•]:=
```
DeleteObject[webSession]
```

Batch downloading

Anything that you can do once with a computer, you can do over and over. In this section we learn to automate the downloading of arbitrary numbers of digital sources.

N.B. Always check the terms of use of a site before you use automated downloading. Many sites, especially those that provide journal articles or other scholarly resources, *expressly forbid* the use of computer programs to harvest or process their resources automatically. Be sure to ask a university librarian if you are in any doubt about what you can or cannot do on a particular site.

In this example, we do a search at the Internet Archive to find some documents, get the id numbers for each document, then go through the list of ids to download the actual documents. The Internet Archive explicitly allows batch downloading and has an incredible collection, so it is a great place to start doing this kind of work.

Figuring out where you are putting stuff

Before you download anything, you want to make sure you have a place on your local file system to put it.

In[•]:=
```
Clear[localDownloadFilePath]
```

Make sure to change the following so that it is a valid file path on your own computer!

In[•]:=
```
localDownloadFilePath = "/Users/wjt/Downloads/huac"
```

Doing a search and parsing the URL

On many websites, once you have entered a search, the system responds with a URL that encodes your search. For example, in your browser go to the search page of the Internet Archive

https://archive.org/search.php

and type the following into the Search box (make sure AND is in uppercase)

collection:(gutenberg) AND communist activities

This will search for any texts from the Project Gutenberg collection that have "communist activities" in the metadata. Now look at the URL in your browser. It should look something like this.

https://archive.org/search.php?query=collection %3 A %28 gutenberg %29 %20 AND %20 communist %20 activities

The *Mathematica* command URLParse can be used to break down URLs like this so they can be better understood.

```
In[•]:= parsedURL = URLParse[
  "https://archive.org/search.php?query=collection%3A%28gutenberg%29%20AND%20
    communist%20activities"]
```

Since this is an Association, we can retrieve parts of the URL.

```
In[•]:= parsedURL["Domain"]
```

```
In[•]:= parsedURL["Query"]
```

Understanding how a search URL is constructed allows us to generate new ones. The Internet Archive also has an advanced search page that allows us to return results as JSON. (We will learn more about JSON soon).

https://archive.org/advancedsearch.php

We can go to this page, enter the same query as above, chose “identifier” under “Fields to return” and “JSON format” then click Search. Now the URL looks like this:

https://archive.org/advancedsearch.php?q=collection %3 A %28 gutenberg %29+AND+communist+activities&fl %5 B %5 D=identifier&sort %5 B %5 D=&sort %5 B %5 D=&sort %5 B %5 D=&rows=50&page=1&output=json&callback=callback&save=yes

Once again we parse it to look at the fields.

```
In[•]:= parsedAdvancedURL = URLParse[
  "https://archive.org/advancedsearch.php?q=collection%3A%28gutenberg%29+AND+
    communist+activities&fl%5B%5D=identifier&sort%5B%5D=&sort%5B%5D=&sort%5B%5D
    =&rows=50&page=1&output=json&callback=callback&save=yes"]
```

Getting identifiers

Given this information, we can now generate a new search that returns the identifiers for the documents that our search turned up. Notice how I have specified search options in the list given to the URLExecute command. The output of this command contains the information returned by the Internet Archive in response to our search.

```
In[•]:= advancedSearchJSON = URLExecute["https://archive.org/advancedsearch.php",
  {"q" → "collection:(gutenberg) AND communist activities",
   "fl[]" → "identifier", "output" → "json"}, "JSON"]
```

Now we can get a list of identifiers quite easily. These identifiers will be used to download the actual documents in the next step.

```
In[•]:= huacIDs = Cases[advancedSearchJSON, {"identifier" → id : _} → id, Infinity]
```

JSON

JSON (short for JavaScript Object Notation) is a convenient human- and machine-readable format that is frequently used to exchange data between computers. Here is the information returned by the Internet Archive after we submitted our advanced search. It is in JSON format.

```
In[•]:= iaJSON = Import[
          "https://archive.org/advancedsearch.php?q=collection%3A%28gutenberg%29+AND+
            communist+activities&fl%5B%5D=identifier&output=json"]
```

Compare this with *advancedSearchJSON* immediately above. Note that curly braces are used to group lists of things together, and that there are *names* and *values* separated by a colon, e.g.,

"numFound":3

Downloading the files

Given an Internet Archive identifier, we can get the URL of the text file to download with the following function.

```
In[•]:= internetArchiveIDToURL[id_] :=
         "https://archive.org/download/" ~~ id ~~ "/" ~~
          Flatten[StringCases[Import["https://archive.org/download/" ~~ id ~~ "/",
              "Hyperlinks"], "/download/" ~~ id ~~ "/" ~~ f : {__ ~~ ".txt"} → f]][[1]]
```

Let's test it.

```
In[•]:= internetArchiveIDToURL["investigationofc56383gut"]
```

The next function downloads the text file for one ID number using the URLSave command. Note that we use the Pause command to wait for 2 seconds between each download so we don't overload their servers. It is polite to wait a bit between requests when you are batch downloading.

```
In[•]:= internetArchiveIDDownload[id_] :=
         {Pause[2];
          URLSave[internetArchiveIDToURL[id],
           FileNameJoin[{localDownloadFilePath, ToString[id] ~~ ".txt"}]]}
```

Now we can use the Map command to grab a batch of files, pausing before we request each one.

```
In[•]:= internetArchiveIDDownload /@ huacIDs
```

A larger batch of files

Batch downloading doesn't make much sense for a few files, but what we can do for a few files we can do for a hundred, or a million. Here is how we collect a larger list of Internet Archive ids. (In the interests of time and space we won't actually download the files but you know how to do that now).

As an example we will request a list of identifiers for texts on the subject of civil defense published between 1947 and 1967 appearing in the Internet Archive. Here is our search URL. Note that we specify

that we want fifty rows at a time and the first page of results.

```
In[•]:= civilDefenseJSON =
 URLExecute[{"https://archive.org/advancedsearch.php"}, {"q" → "subject:(civil
      defense) AND mediatype:(texts) AND date:[1947-01-01 TO 1967-12-31]",
   "fl[]" → "identifier", "rows" → 50, "page" → 1, "output" → "json"}, "JSON"]
```

The first thing to notice is that the *numFound* field shows more than 400 results.

```
In[•]:= Cases[civilDefenseJSON, {"numFound" → n : _, ___} → n, Infinity]
```

We can divide by 50 and round up to see how many pages we have to request if we want all of the results.

```
In[•]:= Ceiling[
 N[First[Cases[civilDefenseJSON, {"numFound" → n : _, ___} → n, Infinity]] / 50]]
```

Now we can use a Table to retrieve all of the IDs. Here I have used Range[3] to grab only the first three pages of results, but we could set that number to the maximum if we wanted them all.

```
In[•]:= Flatten[Table[Cases[URLExecute[{"https://archive.org/advancedsearch.php"},
      {"q" → "subject:(civil defense) AND mediatype:(texts)
          AND date:[1947-01-01 TO 1967-12-31]",
       "fl[]" → "identifier", "rows" → 50, "page" → pg, "output" → "json"}, "JSON"],
    {"identifier" → id : _} → id, Infinity], {pg, Range[3]}]] // viewData
```

When doing this kind of work, the UpTo command can be handy. We will use it later in the lesson.

Text Searching

Mathematica has a number of commands that allow you to search files on your disk. This can be handy because when you are batch downloading, you usually end up with too many files to load into a notebook all at once.

Create an index for a specific directory. Here we are indexing the files we downloaded above.

```
In[•]:= huacIndex = CreateSearchIndex[localDownloadFilePath]
```

Using the index we can search for files that contain a particular term.

```
In[•]:= TextSearch[huacIndex, "fifth amendment"][All]
```

How many documents contain this search term? None.

```
In[•]:= TextSearch[huacIndex, "anvil"]
```

The TextSearchReport command returns a dataset. Note that we still don't know where in a particular file the search term appears, but at least we have some idea of whether or not it is worth importing a file into the notebook for analysis.

```
In[•]:= huacReport = TextSearchReport[huacIndex, "fifth amendment"]
```

The Score field indicates how relevant a particular document is for our search.

```
In[•]:= huacReport[All, {"FileName", "Score"}]
```

If we want to see how a word is used in a particular file, we use the FindList command. Note that lines do not correspond to sentences in the text, and that we don't get any information about where in the file each line is to be found.

```
In[•]:= FindList[FileNameJoin[{localDownloadFilePath, "investigationofc56383gut.txt"}],
   "fifth", IgnoreCase → True] // TableForm // viewData
```

We can use this command to find other words that tend to co-occur with our search term, however. Note that these are words that are in the same file line, not the same sentence.

```
In[•]:= WordCloud[
 DeleteStopwords[Flatten[TextWords[FindList[FileNameJoin[{localDownloadFilePath,
       "investigationofc56383gut.txt"}], "fifth", IgnoreCase → True]]]]]
```

Crawling

A crawler is a program that visits a website, harvests some information including links to other sites, then visits each of those sites in turn. On a large scale, crawlers are used to index the web for searching (e.g., Google, Yahoo!), and to archive the web for future reference (e.g., the Internet Archive's Wayback Machine.) Crawling is possible because the internet and the world wide web are structured as networks. Sites have information about other sites that they are linked to. More generally, we can use the idea of crawling to explore any kind of network.

Wikipedia articles that link to a given article

To get an idea of how crawling works, we can explore a tiny region of the network that is Wikipedia. Any given page has a list of backlinks: other pages that link to it. For example, here are the pages that link to the Wikipedia page on President Harry S. Truman.

```
In[•]:= trumanBacklinks = WikipediaData["Harry S. Truman", "BacklinksList"];
```

```
In[•]:= viewData@trumanBacklinks
```

Let's write a little function that, given a Wikipedia page, returns a number of randomly chosen back links, formatting them as a list of directed edges. That way we will be able to graph them easily.

```
In[•]:= getBacklinks[wpage_, n_] :=
 Table[t ↔ wpage, {t, RandomChoice[WikipediaData[wpage, "BacklinksList"], n]}]
```

Here is how we ask for seven randomly chosen Wikipedia pages pointing to the Harry S Truman page.

```
In[•]:= startList = getBacklinks["Harry S. Truman", 7]
```

We can graph this with the Graph command. We use the GraphLayout option to force the vertexes apart so we can read the labels.

```
In[•]:= Graph[startList, VertexLabels → "Name", GraphLayout → "SpringEmbedding"]
```

Now remember that each of those pages that links to Truman's page also has other pages that link to it. Here is a list of each of the pages pointing to Truman's page.

```
In[•]:= startList[[All, 1]]
```

If we Map the *getBacklinks* function across that list, we can see a larger part of the network. Here I have asked for 3 random backlinks, to keep the graph manageable. This takes a little while to run.

```
In[•]:= oneHopList = getBacklinks[#, 3] & /@ startList[[All, 1]]
```

```
In[•]:= Graph[Join[startList, Flatten@oneHopList], VertexLabels → "Name",
  GraphLayout → "SpringEmbedding", ImageSize → Scaled[.8]]
```

Each time we rerun this code, we will get a different subnetwork, since we are randomly choosing only a few links to explore.

Genealogical connections

This example comes from the *Mathematica* documentation. Since person entities have familial links to one another, we can crawl through genealogical networks, too.

http://www.wolfram.com/language/12/cultural-and-historical-entities/explore-genealogical-connections.html

We start with King George VI.

```
In[•]:= King George VI PERSON ✓ [{"Parents", "Spouses", "Children"}]
```

Let's focus on his children, his children's children, and so on. The NestGraph command is a very powerful way to do exactly this, given a function that indicates what information to extract. We want three levels of descent from King George VI.

```
In[•]:= NestGraph[#["Children"] &, King George VI PERSON ✓, 3,
  VertexLabels → "Name", GraphLayout → "SpringEmbedding", ImageSize → Full]
```

Here are three levels of ancestors.

```
In[•]:= NestGraph[#["Parents"] &, King George VI PERSON ✓, 3,
  VertexLabels → "Name", GraphLayout → "SpringEmbedding", ImageSize → Full]
```

Web crawling

The NestGraph command can also be used to crawl websites. Here we start at the main website for Wolfram Research and take up to 20 hyperlinks for each page we visit. We iterate two times. We can see that there are a number of densely interconnected pages in the middle, and then other paths that lead outward from this cluster.

```
In[•]:= simpleWebCrawl =
  NestGraph[Import[#, "Hyperlinks"][[1 ;; UpTo[20]]] &, "https://www.wolfram.com", 2]
```

Here are the hyperlinks that were visited.

In[•]:=
```
viewData@VertexList@simpleWebCrawl
```

The function that we perform on each web site that we visit in the example above simply gets the list of hyperlinks and chooses a few to follow. Typically we would like to retrieve more information than this. We will look at other examples of crawlers in future lessons.

Web archives and WARC files

Many web crawlers create web archive (WARC) files, as does the OS X browser Safari when you save a page to your local drive. The Wayback Machine at the Internet Archive is built from massive web crawls stored in WARC files, allowing users to explore the history of hundreds of billions of web pages (Milligan 2019, Brügger 2018).

https://archive.org/web/

If you would like to archive a small portion of the web for future use, you can use the Webrecorder.io service. As you browse a site, a WARC file is created, which you can then download. Webrecorder also has a desktop player that allows you to browse through WARC files offline (i.e., when you are not connected to the internet.)

WARC files

On June 6, 2019 I used Webrecorder.io to create a very small WARC file by browsing a few pages of the Dictionary of Canadian Biography online.

http://www.biographi.ca/en/index.php

I stored a copy of the file on Github so that it would be available for later analysis. When we Import a WARC file, we receive a dataset.

In[•]:=
```
wDataset = Import[
  "https://github.com/williamjturkel/Digital-Research-Methods/raw/master/temp-
    20190605151547.warc"]
```

We can inspect each record in this dataset. In this case, the server responded by sending a JPEG that was linked on one of the pages.

In[•]:=
```
wDataset[[7]]
```

Here is how we extract the content of that record.

In[•]:=
```
wDataset[[7]]["Content"]
```

These are the elements.

In[•]:=
```
Import[wDataset[[7]]["Content"], "Elements"]
```

And this is the image itself.

In[•]:=
```
Import[wDataset[[7]]["Content"], "Image"]
```

Other records in this WARC file contain the textual content on the pages I visited. This one, for example, contains the biography of Jeanne Mance, who founded the Hôtel-Dieu (hospital) of Montreal in 1642.

```
In[•]:= wDataset[[230]]

In[•]:= Import[wDataset[[230]]["Content"], "Elements"]

In[•]:= Import[wDataset[[230]]["Content"], "Plaintext"] // viewData
```

Further examples

Searching for related commands

The NestGraph command can be used to find *Mathematica* commands in the neighborhood of one of interest. Here we look for commands related to StringTake. Looking at the Graph we can see that both FileNameTake and Take are analogous to StringTake.

```
In[•]:= NestGraph[WolframLanguageData[#,
    EntityProperty["WolframLanguageSymbol", "RelatedSymbols"]] &,
 Entity["WolframLanguageSymbol", "StringTake"], 2,
 VertexLabels → "Name", ImageSize → Scaled[0.7]]
```

An earlier web crawler

There is an earlier web crawler in the documentation for *Mathematica* 8, which is considerable more difficult to understand, although it basically does the same thing. This is a good example of the ever higher-level nature of commands in the language.

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-modeling/structure-of-the-web.html

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 33-36.

Workflow: Importing and analyzing data

https://reference.wolfram.com/language/workflowguide/ImportingAndAnalyzingData.html

Tutorial: Searching files

https://reference.wolfram.com/language/tutorial/SearchingFiles.html

WARC file format in *Mathematica*

https://reference.wolfram.com/language/ref/format/WARC.html

Official specification of the WARC format

https://iipc.github.io/warc-specifications/

References

Brügger, Niels. *The Archived Web: Doing History in the Digital Age*. Cambridge, MA: MIT Press, 2018.

Milligan, Ian. *History in the Age of Abundance? How the Web is Transforming Historical Research*. Montreal, QC and Kingston, ON: McGill-Queen's University Press, 2019.

ch12

Lesson 12. Linked Open Data

The internet as a global database

In this lesson we begin to work with Linked Open Data (LOD). Here is a tutorial by Jonathan Blaney on the principles of linked open data. You should read it before you begin this lesson.

https://programminghistorian.org/en/lessons/intro-to-linked-data

To understand linked open data, we start with linked data. The basic idea is that conceptual things have an address on the web, usually a name that begins with HTTP, so you can look them up with a browser or other computer program. When you do, you get standardized data with accompanying relationship information (Berners-Lee, 2009). Linked open data is linked data with an open license of some kind, so that it can be freely reused.

One motivating vision behind LOD is the idea of a *semantic web*. In addition to hyperlinked human- and machine-readable pages, the internet can be augmented by metadata about pages and conceptual entities and the relationships between them. Computer programs can then use this metadata to solve problems, treating the internet as a global database of meaning.

Linked open data and RDF

Clearing all definitions , defining *viewData* and loading a package

In[•]:=
```
Clear["Global`*"]
```

In[•]:=
```
viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

In *Mathematica*, packages are collections of commands that are not part of the language by default. You load a package with the Needs command. In this lesson we will be using a number of commands from the GraphStore package.

In[•]:=
```
Needs["GraphStore`"]
```

Keys and values

Throughout the course, we have seen examples of keys and values. These are often written as rules in the data structures of *Mathematica* or as properties for commands. Each rule has a key ("strawberry" in this case) and a value ("red" in this case). Keys are sometimes also known as *attributes*.

In[•]:=
```
Rule["strawberry", "red"]
```

In order to share data, or use data shared by other people, we have to refer to entities and concepts unambiguously. As speakers of English, we have an idea of what a "strawberry" is and what is meant by "red". But we also know that unripe strawberries are not red and that there are many shades of red that could never be used to describe a strawberry. To get around this problem, we need to be able to point to an authority file which specifies exactly what we mean by particular keys and values in a particular context. If we are talking about colors of fruit we might use one authority; if we are talking about lip gloss, liqueur, or lollipops we would use different authorities as appropriate. We might get information about fruit from one authority and information about colors from another.

Resource identifiers

Resource identifiers provide us with a reliably unique way of representing entities and concepts. In linked open data, the Uniform Resource Identifier (URI) is a name that uniquely identifies an entity or concept. Often URIs use HTTP so that they can be resolved (i.e., looked up) on the web. Suppose we have a domain called example.org where say exactly what we mean by "strawberry". Then we can rewrite the rule above as the following.

In[•]:=
```
"http://example.org/strawberry" → "red"
```

The domain example.org is actually a special domain which has been set aside to be used in examples like this one. It doesn't really say what we mean by our entities and concepts. So if we were creating a real application involving fruit or lip gloss or whatever, we would need to use different authorities. But it works fine for examples. We will see real URIs later in the lesson.

The URI is gradually being replaced by the Internationalized Resource Identifier (IRI) which is a generalization of URIs to include Unicode characters. *Mathematica* uses IRIs for its resource identifiers.

In *Mathematica*, we create a resource identifier (an IRI) as follows. Note that since we are using a command that is defined in a package, we can use the name of the package, followed by a backtick (`), followed by the name of the command to explicitly specify it.

In[•]:=
```
GraphStore`IRI["http://example.org/strawberry"]
```

Triples

We have been explicit about the entity "strawberry" being somehow linked to the concept "red", but we haven't explicitly stated the relationship between them. This is where the idea of triples comes in. In the figure below, our subject is "strawberry" and our object is "red". The predicate in this case is "has the color". The language of subjects, objects and predicates is traditional in philosophy and

linguistics.

Graph[...]

In[ ]:= Graph[...]

In order to specify a triple in *Mathematica*, we use the RDFTriple command from the GraphStore package. Here is how we say that our strawberry has the color red.

```
In[ ]:= GraphStore`RDFTriple[GraphStore`IRI["http://example.org/strawberry"],
         GraphStore`IRI["http://schema.org/color"], "red"]
```

A couple of things to notice. First, we have provided resource identifiers for strawberry and color, but not for red. Second, we have used a different domain as the authority for color. Unlike example.org, schema.org actually does resolve IRIs. You can see the webpage by clicking the double arrow opener (») in the IRI or by clicking the following link.

https://schema.org/color

We've said that our strawberry has the color red, but we haven't said what we mean by "strawberry". We could do this with another triple: strawberry has the type fruit. In this case we are using a third authority for the type predicate, the domain w3.org.

```
In[ ]:= GraphStore`RDFTriple[GraphStore`IRI["http://example.org/strawberry"],
         GraphStore`IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
         GraphStore`IRI["http://example.org/fruit"]]
```

(As an aside, I wanted to include the two network graphs at this point in the notebook, but I didn't want to focus attention on the code that I used to create them. So I used the Edit→Un/Iconize Selection menu option to hide the code by iconizing it. You can click on the plus sign after Graph[...] to uniconize the code if you wish to see how the figures were created. Since attention is always a scarce resource, this ability to hide unimportant details is useful. It is another example of creating black boxes to manage complexity.)

A simple graph

We build up a collection of linked data by specifying triples. Taken together, the two RDFTriple expressions above create the following network of subjects, predicates and objects.

In[ ]:= Graph[...]

We create a graph of triples with the RDFTriple command in the GraphStore package. Here is an expanded fruit example. Note that RDFStore takes a list of RDF triples.

```
In[•]:= fruitStore = GraphStore`RDFStore[
          {GraphStore`RDFTriple[GraphStore`IRI["http://example.org/banana"],
            GraphStore`IRI["http://schema.org/color"], "yellow"],
           GraphStore`RDFTriple[GraphStore`IRI["http://example.org/banana"],
            GraphStore`IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
            GraphStore`IRI["http://example.org/fruit"]],
           GraphStore`RDFTriple[GraphStore`IRI["http://example.org/cherry"],
            GraphStore`IRI["http://schema.org/color"], "red"],
           GraphStore`RDFTriple[GraphStore`IRI["http://example.org/cherry"],
            GraphStore`IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
            GraphStore`IRI["http://example.org/fruit"]],
           GraphStore`RDFTriple[GraphStore`IRI["http://example.org/strawberry"],
            GraphStore`IRI["http://schema.org/color"], "red"],
           GraphStore`RDFTriple[GraphStore`IRI["http://example.org/strawberry"],
            GraphStore`IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
            GraphStore`IRI["http://example.org/fruit"]]}, Association[]]
```

Serializations

As explained in the Blaney tutorial, RDF triples can be written in a number of different data formats. In *Mathematica* we can use the ExportString command to write our graph in different formats. Here is our graph in Turtle format.

```
In[•]:= ExportString[fruitStore, "Turtle"]
```

In SPARQL, *a* is a keyword that is shorthand for rdf:type.

If we want to use prefixes, we specify them in an association.

```
In[•]:= ExportString[fruitStore, "Turtle", "Prefixes" →
          <|"ex" → "http://example.org/",
           "schema" → "http://schema.org/"|>]
```

Here is how we export our graph in RDF/XML format, using prefixes.

```
In[•]:= ExportString[fruitStore, "RDFXML", "Prefixes" →
           <|"ex" → "http://example.org/",
            "schema" → "http://schema.org/"|>] // viewData
```

SPARQL queries and endpoints

Querying in-memory graphs

SPARQL is the query language for RDF graphs. Given an RDF graph, we can use SPARQL to retrieve and manipulate linked data. Here is how we query our RDF graph to get all of the entities of type fruit. The line that begins with select is the actual SPARQL query. Given the same graph, you could use this query in the same way in any system that can understand SPARQL queries. In English it says “in the graph

fruitStore find all of the triples that contain the predicate rdf:type (remember, that is what the lower-case *a* means) and the object fruit."

```
In[•]:= fruitStore // GraphStore`SPARQLQuery["
        select * where {?fruit a <http://example.org/fruit>.}
        "]
```

Here is a query to find the color of a banana. The * after select means "find all".

```
In[•]:= fruitStore // GraphStore`SPARQLQuery["
        select * where {<http://example.org/banana><http://schema.org/color>?color}
        "]
```

Here is how we find all red fruits. Note that we have to escape the quotation marks around "red". Note also that I didn't explicitly specify the package name here. I won't from now on, to make the examples easier to read.

```
In[•]:= fruitStore // SPARQLQuery["
          select * where {
            ?fruit a <http://example.org/fruit> .
            ?fruit <http://schema.org/color> \"red\" .
          }
          "]
```

This last query starts to suggest some of the power of SPARQL because we can see that the system isn't merely matching a pattern and returning a value. Instead it has to match two patterns (things that are fruit and things that are red), then fuse the information to provide the answer we are looking for. In a later lesson, when we study databases, we will see that we can use the same kind of querying for other data structures.

Symbolic SPARQL

If you are familiar with SPARQL (or plan to learn it), using SPARQLQuery with strings like we did in the example above works fine. If you would rather build up your queries using the symbolic expressions of *Mathematica*, that is possible, too.

We use the IRI, RDFTriple and SPARQLVariable commands to describe triple patterns, which we can pass to the SPARQLSelect command. This query shows us the colors of all fruit.

```
In[•]:= colorPattern =
          RDFTriple[SPARQLVariable["fruit"],
           IRI["http://schema.org/color"], SPARQLVariable["color"]];
```

```
In[•]:= fruitStore // SPARQLSelect[colorPattern]
```

Use this query to find the color of a banana.

```
In[•]:= bananaColorPattern =
          RDFTriple[IRI["http://example.org/banana"],
           IRI["http://schema.org/color"], SPARQLVariable["color"]];
```

```
In[•]:= fruitStore // SPARQLSelect[bananaColorPattern]
```

Find all red fruits. In this example, I didn't bother to save the search pattern separately. There usually isn't any point in assigning expressions to symbols unless you plan to reuse the symbol.

```
In[•]:= fruitStore // SPARQLSelect[
  {RDFTriple[SPARQLVariable["fruit"],
    IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
    IRI["http://example.org/fruit"]],
   RDFTriple[SPARQLVariable["fruit"], IRI["http://schema.org/color"], "red"]}]
```

Query Wikidata, a SPARQL endpoint

At this point you may be wondering why you would go to all the trouble of representing your own data in triples just so you could query it. You probably wouldn't unless you wanted to share it as linked open data. The real power of RDF and SPARQL comes when you query data from SPARQL endpoints, which are RDF datasets online.

Wikidata is a free knowledge base with 56M+ data items that anyone can edit. It can be queried with SPARQL.

https://www.wikidata.org/wiki/Wikidata:Main_Page

In this example, we will use a query string ask Wikidata for the highest mountain. Don't worry about the syntax of the query at first. We will unpack it soon.

Specify an endpoint.

```
In[•]:= endpoint = "https://query.wikidata.org/sparql";
```

Construct a query string.

```
In[•]:= mountainQuery = "
  select distinct ?name ?elevation where {
    ?mountain wdt:P31 wd:Q8502 .
    ?mountain p:P2044 / psn:P2044 / wikibase:quantityAmount ?elevation .
    ?mountain rdfs:label ?name . filter(lang(?name) = \"en\")
  }
  order by desc(?elevation)
  limit 1
  ";
```

Call SPARQLExecute. This takes a while, but it returns the elevation of Mt Everest in meters.

```
In[•]:= SPARQLExecute[endpoint, mountainQuery]
```

Check against *Mathematica*'s entities.

```
In[•]:= UnitConvert[Mount Everest MOUNTAIN[elevation], "Meters"]
```

Stop for a moment to think about this as a process. If you wanted to know the elevation of the highest mountain in the world before the advent of the world wide web, you would go to the reference section

of a library and use a book like the *Information Please Almanac*, which was published annually from 1947 into the 2000s and included lists of things like the heights of tallest buildings, winners of playoffs for various sports and so on. It might take a few hours to track down the answer including travel time to and from the library. In 1998 Information Please went online, and Wikipedia was launched in 2001. In the 21st century, it might only take a few minutes to track down answers, although verifying that they are correct can take (much) longer.

If you know how to write SPARQL queries, however, you can automate the process of finding answers to questions. If you need to know the elevation of Mt Kilimanjaro you can look it up in Wikipedia. But if you need to assemble an ordered table of the world's highest mountains, along with their latitudes and longitudes, you can write a program to do that relatively quickly. Let's make one very minor change to our query and rerun it.

```
In[•]:= mountainQuery2 = "
  select distinct ?name ?elevation where {
    ?mountain wdt:P31 wd:Q8502 .
    ?mountain p:P2044 / psn:P2044 / wikibase:quantityAmount ?elevation .
    ?mountain rdfs:label ?name . filter(lang(?name) = \"en\")
  }
  order by desc(?elevation)
  limit 10
  ";
```

Now we have a list of the 10 highest mountains with elevations in meters.

```
In[•]:= tenHighestMountains = SPARQLExecute[endpoint, mountainQuery2]
```

Let's format this information as a nice table. If we take the first element of the list, we can extract its name and elevation as follows.

```
In[•]:= tenHighestMountains[[1]]["name"]
```

```
In[•]:= tenHighestMountains[[1]]["elevation"]
```

In order to remove the RDFString formatting around the name, we can use the ReplaceAll command.

```
In[•]:= tenHighestMountains[[1]]["name"] /. RDFString[s_, _] → s
```

Combining both into a pure function, we map it across the list of the ten highest mountains.

```
In[•]:= {#["name"] /. RDFString[s_, _] → s, #["elevation"]} & /@
    tenHighestMountains // TableForm
```

Specifying Wikidata queries

Let's get into the query syntax that we just used a little bit. We know that a SPARQL select command looks like this

select variable where {triple(s)}

The *distinct* modifier eliminates duplicates. The *order by* part of the expression sorts the output, and

the *limit* part of the expression tells how many items to return. So now we have a query that looks like this

select distinct variable where {triple(s)}
order by something
limit n

The first triple in our query looks like this

?mountain wdt:P31 wd:Q8502 .

We know that *?mountain* is the variable that we are matching. The other two parts of the triple are Wikidata identifiers for items (*wd*) and properties (*wdt*). So what are they? We can look them up using Wikidata itself.

This is the resource identifier for *mountain* in Wikidata.

https://www.wikidata.org/wiki/Q8502

This is the resource identifier for the *instance of* property in Wikidata.

https://www.wikidata.org/wiki/Property:P31

So in English, that first triple says “something that is an instance of mountain.” The second and third triples in the query are slightly more complicated. The second uses what are called ‘property paths’ to succinctly describe a chain of triples to follow, and the third uses a filter to return mountain names in English. Using the SPARQL tutorial at Wikidata

https://www.wikidata.org/wiki/Wikidata:SPARQL_tutorial

you can teach yourself to read and write SPARQL queries like the one that we used. As with our other examples of code reading, don’t expect to understand everything at once. Just try copying and modifying queries until you understand how they work. Here is a huge list of sample queries to experiment with:

https://www.wikidata.org/wiki/Wikidata:SPARQL_query _service/queries/examples

There is more information in the “Further Examples” section of the lesson.

Aggregating

When you reduce a list of values to a single value, that is called aggregation.

Counting fruits

Here are all the fruits in our first example.

In[•]:=
```
fruitStore // SPARQLSelect[
  RDFTriple[SPARQLVariable["fruit"],
   IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
   IRI["http://example.org/fruit"]]]
```

We can count them with SPARQLAggregate.

```
In[•]:= fruitStore // SPARQLSelect[
          RDFTriple[SPARQLVariable["fruit"],
           IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
           IRI["http://example.org/fruit"]]] //
        SPARQLQuery[SPARQLAggregate["count" → SPARQLEvaluation["count"][]]]
```

If we want to count the fruits by color, we have to get color information when we retrieve them.

```
In[•]:= fruitStore // SPARQLSelect[
          {RDFTriple[SPARQLVariable["fruit"],
            IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
            IRI["http://example.org/fruit"]],
           RDFTriple[SPARQLVariable["fruit"], IRI["http://schema.org/color"],
            SPARQLVariable["color"]]
          }]
```

Now we can aggregate.

```
In[•]:= fruitStore // SPARQLSelect[
          {RDFTriple[SPARQLVariable["fruit"],
            IRI["http://www.w3.org/1999/02/22-rdf-syntax-ns#type"],
            IRI["http://example.org/fruit"]],
           RDFTriple[SPARQLVariable["fruit"], IRI["http://schema.org/color"],
            SPARQLVariable["color"]]
          }] // SPARQLQuery[SPARQLAggregate[{"color",
           "count" → SPARQLEvaluation["count"][]}, SPARQLVariable["color"]]]
```

Using SPARQL queries with *Mathematica*'s entities

Following an example in the documentation, we use SPARQL here to query and aggregate entity data.

Here is a query that returns all country-population pairs.

```
popQuery =
  SPARQLSelect[
   RDFTriple[SPARQLVariable["country"],
    EntityProperty["Country", "Population"], SPARQLVariable["population"]]
  ];
```

We execute the query like this.

```
In[•]:= Entity["Country"] // popQuery // viewData
```

Suppose we want to compute the total population. Then we would aggregate the population values and return the sum. We do this in SPARQL with the SPARQLAggregate command.

```
In[•]:= Entity["Country"] // popQuery // SPARQLQuery[SPARQLAggregate[
          "totalPopulation" → SPARQLEvaluation["sum"][SPARQLVariable["population"]]]]
```

What if we wanted the population broken down by continent? Then we have to request both the

population and the continent for each country, and then group the countries by continent for the purposes of aggregation.

```
popContinentQuery = SPARQLSelect[{
    RDFTriple[SPARQLVariable["country"], EntityProperty["Country", "Population"],
     SPARQLVariable["population"]], RDFTriple[SPARQLVariable["country"],
     EntityProperty["Country", "Continent"], SPARQLVariable["continent"]]
   }];
```

In[•]:=
```
populationByContinent =
 Entity["Country"] // popContinentQuery // SPARQLQuery[SPARQLAggregate[{"continent",
    "totalPopulation" → SPARQLEvaluation["sum"][SPARQLVariable["population"]]},
   SPARQLVariable["continent"]]]
```

This is in a form that can be easily turned into a Dataset.

In[•]:=
```
Dataset@populationByContinent
```

Requesting JSON-LD from a website

This final section is based on an example from the documentation.

http://www.wolfram.com/language/12/rdf-and-sparql/request-linked-data-from-a-website.html?product=mathematica

The MusicBrainz site is an open music encyclopedia that makes metadata available to the public. With a browser you can navigate to a page of information. Here is the page for the compilation album *Colección 78 RPM: Pura Milonga, 1932/1934* by the tango orchestra leader Francisco Canaro.

In[•]:=
```
albumURL = "https://musicbrainz.org/release/30b9c67f-812c-430e-8b4a-d6fda8a7b08b";
```

In[•]:=
```
Show[WebImage[albumURL], ImageSize → Medium]
```

Here is the page of information for Canaro himself.

In[•]:=
```
artistURL = "https://musicbrainz.org/artist/a7389752-9e5d-406d-9a02-9141600e4315";
```

In[•]:=
```
Show[WebImage[artistURL], ImageSize → Medium]
```

These pages are good for a human who is browsing the web, but if we are going to make use of the data programmatically, we usually want a machine-readable form like JSON. In fact, the MusicBrainz site will return JSON-LD, a JSON-based format for linked data. Here is how we request that.

In[•]:=
```
albumRequest =
  HTTPRequest[albumURL, <|"Headers" → {"accept" → "application/ld+json"}|>];
```

In[•]:=
```
albumData = URLExecute[albumRequest, "JSONLD"]
```

We now have the information about the album in the familiar RDFStore format. Recall that RDF triples are of the form *subject-predicate-object*. If we know what the predicates are, we can get a sense of the kind of data that is available. Here is a SPARQL query to get a list of predicates.

In[•]:=
```
albumData // SPARQLQuery["select distinct ?p where {[] ?p []}"] // Query[All, "p"]
```

(What does the Query command do in the expression above? We will learn more about this command and its relatives in a future lesson.)

Here is a query that will retrieve, for all tracks, the number and name. Note that we define the prefix function first.

```
Clear[schema];
schema[s_] := IRI["http://schema.org/" <> s];
```

```
In[•]:= trackData = albumData // SPARQLSelect[{
          RDFTriple[SPARQLVariable["track"],
           schema["trackNumber"], SPARQLVariable["number"]],
          RDFTriple[SPARQLVariable["track"], schema["name"], SPARQLVariable["name"]]
         } → {"name", "number"}];
```

If we look at the data, we see that the number is in the form *disk.track*, and all of these tracks are from disk 1.

```
In[•]:= trackData
```

We can transform the track numbers by splitting each string at the dot, keeping the second part only, and using the ToExpression command to convert each from a string to an integer.

```
In[•]:= ToExpression[StringSplit[Values[trackData][[All, 2]], "."][[All, 2]]
```

Here are the track names, in the same order.

```
In[•]:= Values[trackData][[All, 1]]
```

We put the track names back together with the track numbers, sort by the latter, and convert to a dataset.

```
In[•]:= SortBy[MapThread[List, {Values[trackData][[All, 1]], ToExpression[
          StringSplit[Values[trackData][[All, 2]], "."][[All, 2]]}], Last] // Dataset
```

The version in the documentation gets the same result through a slightly different method. Compare the code there with the code here when you are reviewing this lesson. Many of the SPARQL examples in the documentation (including this one) make use of the functional command RightComposition. Understanding how it works will make it easier to read those examples.

Further examples

Here are a number of further Wikidata query examples in *Mathematica*'s symbolic expression form, adapted from the Wikidata examples page.

Wikidata prefixes

If you write your SPARQL queries as strings, then you can use the Wikidata prefixes like *wd:* and *psn:* directly. If you want to use the symbolic Mathematica expressions, however, this list is very useful.

https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format #Prefixes _used

Wikidata definitions

We need to define prefixes when we are querying in this mode. The definitions are repeated and expanded here.

```
Clear[bd, endpoint, p, psn, rdfs, wd, wdt, wikibase]
```

In[•]:=
```
endpoint = "https://query.wikidata.org/sparql";
```

In[•]:=
```
bd[s_] := IRI["http://www.bigdata.com/rdf#" <> s];
p[s_] := IRI["http://www.wikidata.org/prop/" <> s];
psn[s_] := IRI["http://www.wikidata.org/prop/statement/value-normalized/" <> s];
rdfs[s_] := IRI["http://www.w3.org/2000/01/rdf-schema#" <> s];
wd[s_] := IRI["http://www.wikidata.org/entity/" <> s];
wdt[s_] := IRI["http://www.wikidata.org/prop/direct/" <> s];
wikibase[s_] := IRI["http://wikiba.se/ontology#" <> s];
```

Cats

A listing of all the cats in Wikidata.

In[•]:=
```
SPARQLExecute[endpoint,
  SPARQLSelect[
   {RDFTriple[SPARQLVariable["item"], wdt["P31"], wd["Q146"]],
     SPARQLService[wikibase["label"],
      {RDFTriple[bd["serviceParam"], wikibase["language"], "[AUTO_LANGUAGE],en"]}]
    } → {"item", "itemlabel"}
  ]
 ] // viewData
```

Cats with picture links

In[•]:=
```
SPARQLExecute[endpoint,
  SPARQLSelect[
   {RDFTriple[SPARQLVariable["item"], wdt["P31"], wd["Q146"]],
     RDFTriple[SPARQLVariable["item"], wdt["P18"], SPARQLVariable["pic"]],
     SPARQLService[wikibase["label"],
      {RDFTriple[bd["serviceParam"], wikibase["language"], "[AUTO_LANGUAGE],en"]}]
    } → {"item", "itemlabel", "pic"}
  ]
 ] // viewData
```

Horses with some info

In[ ]:=
```
SPARQLExecute[endpoint,
  SPARQLSelect[
   {RDFTriple[SPARQLVariable["horse"], wdt["P31"], wd["Q726"]],
     SPARQLOptional[RDFTriple[
       SPARQLVariable["horse"], wdt["P569"], SPARQLVariable["birthdate"]]],
     SPARQLService[wikibase["label"], {RDFTriple[bd["serviceParam"],
        wikibase["language"], "[AUTO_LANGUAGE],en"]}]
    } → {"horse", "horseLabel", "birthyear" →
      SPARQLEvaluation["year"][SPARQLVariable["birthdate"]]}
  ]
 ] // viewData
```

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 37-40.

Workflow: Load a Package

https://reference.wolfram.com/language/workflow/LoadAPackage.html

Convert Files between RDF Formats

Using the FileConvert command to get different serializations of an RDF graph.

http://www.wolfram.com/language/12/rdf-and-sparql/convert-files-between-rdf-formats.html?product=mathematica

Example: Find Public and Alive SPARQL Endpoints

This example uses a SPARQL query in Wikidata to find other SPARQL endpoints then tests each one to see if it is alive. It is a good example of using *Mathematica* for automated testing of online sites.

http://www.wolfram.com/language/12/rdf-and-sparql/find-public-and-alive-sparql-endpoints.html?product=mathematica

There is also a list of SPARQL endpoints at W3.org

https://www.w3.org/wiki/SparqlEndpoints

JSON-LD format

https://json-ld.org

References

Berners-Lee, Tim. “The next web.” TED2009.
https://www.ted.com/talks/tim_berners_lee _on _the _next _web

Blaney, Jonathan. “Introduction to the Principles of Linked Open Data.” *Programming Historian* (2017, rev 2019).
https://programminghistorian.org/en/lessons/intro-to-linked-data

ch13

Lesson 13. Markup Languages

Scraping and parsing marked up text

When information is presented online, the content provider needs some way of indicating how different elements are supposed to look, how they are related to one another, and, at least to some extent, what they mean. All of this *metadata* can be provided in the form of *tags* which are added to a text file. The process of adding such information is called ‘marking up’ a file, hence ‘markup languages’.

HTML, the HyperText Markup Language, is widely used on the web. In HTML there are a limited set of predefined tags to indicate how data should be rendered or displayed, usually in a web browser.

XML, the Extensible Markup Language, uses tags to *describe* data. There are many standards for representing various kinds of metadata with XML, but users are also free to define their own tags (hence ‘extensible’). This flexibility, and the fact that XML is both human- and machine-readable, makes it a very powerful way to represent textual and numeric information.

There is a spectrum of approaches to dealing with marked up text like HTML and XML. At one end is what is sometimes called *scraping*. By using low level character or string patterns, it is possible to find information based on surrounding context and extract it. At the other end is *parsing*. Higher-level commands are used to break a marked-up text into meaningful components for further processing.

As an analogy, think of the processes that we used to turn text strings into linguistic units like words and sentences. The lower level approach focused on matching things like punctuation marks and whitespace, in an effort to find useful boundaries.

In[•]:=
```
StringSplit[
 "The quick brown fox jumped over the lazy dog. The dog did not notice.",
 WhitespaceCharacter]
```

The higher level approach used commands that have a much more nuanced knowledge of natural language built into them.

In[•]:=
```
TextWords["The quick brown fox jumped over the lazy dog. The dog did not notice."]
```

Really Simple Syndication (RSS)

Clearing all definitions and defining *viewData*

```
In[•]:= Clear["Global`*"]
```

```
In[•]:= viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

RSS feeds

As an example of XML we will start with RSS feeds. RSS (Really Simple Syndication) is an XML format for storing information about frequently changed content on a website. A blog, for example, will typically have an RSS feed that you can subscribe to using a *feed reader*. When you run your feed reader, it contacts the site, checks to see if there is a blog post that you haven't read, and, if so, downloads some information about that post. Many news sites, too, use RSS feeds for new stories; libraries often use them to advertise books that have recently been purchased; retailers use them for new product information, and so on.

The *New York Times* has RSS feeds for a variety of topics that are covered frequently in the newspaper. On May 14, 2018, I made a copy of their RSS feed for the Cuban Missile Crisis (1962).

The Cuban Missile Crisis topic page is

https://www.nytimes.com/topic/subject/cuban-missile-crisis-1962

and the link to my copy of the RSS feed is

https://raw.githubusercontent.com/williamjturkel/Digital-Research-Methods/master/nytimes-cuban-missile-crisis-1962-rss.xml

We can use this link to explore some of the things we can do with RSS. The Import command, for example, shows us which elements we can request.

```
In[•]:= Import[
 "https://raw.githubusercontent.com/williamjturkel/Digital-Research-Methods/master
   /nytimes-cuban-missile-crisis-1962-rss.xml", "Elements"]
```

If we just want to read the feed (as we would with a feed reader), we can actually open it as a new *Mathematica* notebook with the CreateDocument command.

```
In[•]:= CreateDocument[Import[
   "https://raw.githubusercontent.com/williamjturkel/Digital-Research-Methods/
     master/nytimes-cuban-missile-crisis-1962-rss.xml"]]
```

Scraping the source

If we import the file as text, we end up with a string. We can use the StringTake command to see a part of it.

```
In[•]:= rssSource = Import[
          "https://raw.githubusercontent.com/williamjturkel/Digital-Research-Methods/
            master/nytimes-cuban-missile-crisis-1962-rss.xml", "Text"];

In[•]:= Head@rssSource

In[•]:= StringTake[rssSource, 400]
```

Suppose we wanted to know what elements in this file are marked up with the title tag. We can use string patterns to extract the information.

```
In[•]:= StringCases[rssSource, Shortest["<title>" ~~ t__ ~~ "</title>"] → t]
```

Note that this approach doesn't require that our commands know anything special about XML, or even that this text string is an example of XML as opposed to, say, a DNA sequence, an ISBN or a rhyming couplet.

So what kinds of things might a higher-level system know about XML? It would know that XML contains tags; that tags are formatted like

```
< tag attribute = "value" > data < / tag >
```

that tags need to be properly nested, that is to say that the following is OK

```
< outer > < inner > ... < / inner > < / outer >
```

but this is not OK

```
< outer > < inner > ... < / outer > < / inner >
```

and so on.

Extensible Markup Language (XML)

If we want a representation of XML that incorporates a higher-level understanding of the markup language, we import into a form called 'symbolic XML', which is native to the *Mathematica* language. In symbolic XML, the whole file is represented by an XMLObject and each tag by an XMLElement.

Symbolic XML

A regular XML expression like

```
< tag attribute = "value" > data < / tag >
```

Is converted to a symbolic XML expression of the form

```
XMLElement["tag", {"attribute" → "value"}, {"data"}]
```

Here is the New York Times RSS feed in Symbolic XML

```
In[•]:= rssSymbolic = Import[
          "https://raw.githubusercontent.com/williamjturkel/Digital-Research-Methods/
            master/nytimes-cuban-missile-crisis-1962-rss.xml",
          "SymbolicXML"];
```

```
In[•]:= viewData[rssSymbolic]
```

Parsing XML

Symbolic XML takes a bit of getting used to, but it is actually pretty easy to extract the information that you want. Here we use the Cases command to match XMLElements that have the tag “title”. The Infinity is a level specification which says to try to match the pattern at every level of nesting.

```
In[•]:= Cases[rssSymbolic, XMLElement["title", _, _], Infinity]
```

If we simply wanted to return the titles themselves, we could do it like this.

```
In[•]:= Cases[rssSymbolic, XMLElement["title", _, {t_}] → t, Infinity]
```

In this file, the item tag serves as a record, containing a number of fields. Here is how we return the first item. Within the item, we can see that there are title, link, description, author, and pubDate tags.

```
In[•]:= rssSampleRecord = First@Cases[rssSymbolic, XMLElement["item", _, _], Infinity]
```

The third part of the item XMLElement is the list of fields. Each is an XMLElement in its own right.

```
In[•]:= rssSampleRecord[[3]]
```

Using the ReplaceAll command, we can extract the tag and data field for each of these records.

```
In[•]:= rssSampleRecord[[3]] /. XMLElement[t_, a_, {d_}] → {t → d}
```

Now that we know how to extract the information in a single record, we can apply it to all of the records in the file, then create a Dataset from the results.

```
In[•]:= Dataset@
 Map[Association, Cases[rssSymbolic, XMLElement["item", _, _], Infinity][[All, 3]] /.
   XMLElement[t_, a_, {d_}] → {t → d}]
```

If we wrap these steps in a function, we can use it to parse the day’s RSS feed into a dataset automatically. Let’s do that then try it on the live feed.

```
In[•]:= nytFeedToDataset[url_] := Dataset[Map[Association,
    Cases[Import[url, "SymbolicXML"], XMLElement["item", _, _], Infinity][[All, 3]] /.
     XMLElement[t_, a_, {d_}] → {t → d}]]
```

```
In[•]:= todaysNews = nytFeedToDataset[
  "https://www.nytimes.com/svc/collections/v1/publish/http://www.nytimes.com/topic
    /subject/cuban-missile-crisis-1962/rss.xml"]
```

If you want to archive your feeds, just save copies of the XML files with URLDownload then work with your local copies by importing them from your disk.

Information trapping

Let’s stop for a moment to focus on what we’ve accomplished and how we might use this a part of a larger digital research strategy. Usually when you do a web or library search, you skim through the results and read, bookmark or take notes on whatever interests you. That is fine if you are looking for

something in particular, but what do you do if you want to keep up with the news on a topic, or create a steady stream of new results on a regular basis? Tara Calishain calls this process 'information trapping'. The basic idea is that whenever possible, new information should flow to you without you having to go and look for it.

We have seen that it is fairly straightforward to extract the content of RSS feeds into datasets. If you have a number of topics that you are monitoring, or if you want to study the ways that the news on a particular topic changes in realtime, or if you want to know when your library gets new books of interest, or when new books or papers on your subject are published, or when a new issue of a journal comes out, or if you are following the discussion on a forum or mailing list... you can have one or more notebooks that runs regularly, scrapes or parses out the content and analyzes and/or archives it.

Here are the locations mentioned in recent stories that the editors of the *New York Times* think are related to the Cuban Missile Crisis of 1962.

In[•]:=
```
todaysLocations =
 Union[Cases[Flatten[TextCases[StringJoin /@ Normal[todaysNews[All, "description"]],
     "Location" → (#Interpretation &)]], GeoPosition[_]]]
```

In[•]:=
```
GeoListPlot[todaysLocations]
```

Text Encoding Initiative (TEI)

Another application of XML that is often encountered in scholarly work, particularly in the humanities, is the Text Encoding Initiative (TEI). It is a standard for the representation of texts in digital form which provides guidelines for marking up things like the sections of documents (e.g., preface, acknowledgements, abstract, contents, chapters, glossaries); dates, times and locations; words or phrases that are highlighted (e.g., italics, block quotation); named entities (e.g., persons, organizations); and so on. The full TEI specification runs to a couple of fat volumes when printed. Most projects that present sources online using TEI work with a subset of the specification and tags that are specific to the particular application.

The Scissors and Paste Database

This section of the lesson is adapted from a tutorial by M. H. Beals that draws on the *Scissors and Paste Database*, "a collaborative and growing collection of articles from British and imperial newspapers in the 18th and 19th centuries."

http://scissorsandpaste.net

Files marked up with TEI are available at the project's GitHub site.

https://github.com/mhbeals/scissorsandpaste

Let's retrieve a simplified version of the database to explore. It is converted into Mathematica's symbolic XML when we import it.

```
In[•]:= simplifiedSAP = Import[
          "https://github.com/mhbeals/scissorsandpaste/raw/master/Outputs/XML/Simplified/
            SimplifiedSAP.xml"];

In[•]:= viewData@simplifiedSAP
```

Each entry in the database is enclosed in record tags. Here is the first one. Each record has a number of fields which capture the publication metadata (e.g., title, city, province, country, date), fields for the headline and text, and fields which indicate paragraph boundaries. There are also keywords.

```
In[•]:= First@Cases[simplifiedSAP, XMLElement["record", _, _], Infinity]
```

Extracting fields

We can extract various fields, just as we did with the RSS example. Here is how we request the ids, for example. Note the use of BlankNullSequence (___) which matches zero or more expressions.

```
In[•]:= Cases[simplifiedSAP,
          XMLElement["record", _, {___, XMLElement["id", {}, {id_}], ___}] → id,
          Infinity] // viewData
```

Here we extract a dataset of ids, titles and dates.

```
In[•]:= simplifiedSAPrecords =
         Map[Association, Cases[simplifiedSAP, XMLElement["record", _, {___,
                XMLElement["id", {}, {id_}], ___, XMLElement["title", {}, {title_}],
                ___, XMLElement["date", {"when" → date_}, _], ___}] →
              {"id" → id, "title" → title, "date" → date}, Infinity]] // Dataset
```

Now that we have a dataset, we can query it. How many distinct titles are there?

```
In[•]:= simplifiedSAPrecords[CountDistinct, "title"]
```

How many articles from each of the different titles?

```
In[•]:= Counts[simplifiedSAPrecords[[All, 2]]]
```

Plot publication dates on a timeline.

```
In[•]:= TimelinePlot[simplifiedSAPrecords[[All, 3]]]
```

Filtering results

Suppose we want to retrieve all of the keywords from 1789. We could parse all of the symbolic XML into a dataset, and then use dataset querying to retrieve the records of interest.

Here is a dataset of ids, dates and keywords.

```
In[•]:= simplifiedSAPKeywords =
          Dataset[Map[Association, Cases[simplifiedSAP, XMLElement["record",
                 _, {___, XMLElement["id", {}, {id_}], ___, XMLElement["date",
                    {"when" → date_}, _], ___, XMLElement["keywords", {}, keywords__], ___}] →
                {"id" → id, "date" → date, "keywords" → keywords}, Infinity] /.
              XMLElement["keyword", {}, {k_}] → k]];
```

We use the Select command to pull out ones from 1789.

```
In[•]:= simplifiedSAPTexts[Select[StringTake[#date, 4] == "1789" &]]
```

File differences

The Scissors and Paste Database was compiled as part of a study of reprinting and reuse of newspaper articles in the 19th century. Determining whether two files or texts are the same or different, and determining how they differ, is a fundamental scholarly problem in many fields ranging from traditional hermeneutics to molecular biology. In this section we look at relatively fine-grained approaches. In future lessons we will learn ways to measure the difference or similarity of whole files or texts.

'Fingerprinting' a text or file

Here are three sample texts. Are they the same or different?

```
In[•]:= textA =
          "Governor Philips describes Port Jackson, which is only about nine miles to
            the Northward of Botany Bay in South Wales, on the Coast
            of Holland, as one of the finest harbours in the world, in
            which 1000 sail of the line might ride in perfect security.";
```

```
In[•]:= textB =
          "Governor Philips describes Port Jackson, which is only about nine miles to
            the Northward of Botany Bay in South Wales, on the Coast
            of Holland, as one of the finest harbours in the world, in
            which l000 sail of the line might ride in perfect security.";
```

```
In[•]:= textC =
          "Governor Philips describes Port Jackson, which is only about nine miles to
            the Northward of Botany Bay in South Wales, on the Coast
            of Holland, as one of the finest harbours in the world, in
            which 1OOO sail of the line might ride in perfect security.";
```

One way to answer this question is to use a *hash function*, which computes a unique code from a file of any kind (in this case, we are using texts). Even the slightest change in the file results in a completely different hash.

```
In[•]:= Hash[textA, "MD5", "HexString"]
```

```
In[•]:= Hash[textB, "MD5", "HexString"]
```

```
In[•]:= Hash[textC, "MD5", "HexString"]
```

Small differences

Since the three sample texts have different hashes, we can conclude they are all different from one another. But how do they differ?

Once we have determined that there are differences between two texts , we can use another function, called SequenceAlignment to find the differences between each pair.

```
In[•]:= SequenceAlignment[textA, textB]
```

```
In[•]:= SequenceAlignment[textB, textC]
```

```
In[•]:= SequenceAlignment[textA, textC]
```

This allows us to see that *textA* has the numeral one where *textB* has a lowercase L; that *textB* has a lowercase L followed by three numeral zeros where *textC* has a numeral one followed by three capital Os; and that *textA* has three numeral zeros where *textC* has three capital Os.

The SequenceAlignment command also allows us to see insertions and deletions, specified in terms of characters rather than words (look at the line/lime part).

```
In[•]:= textD = "Governor Philips describes Port Jackson, which is only
    nine miles to the Northward of Botany Bay in New South Wales, on
    the Coast of Holland, as one of the finest harbours in the world,
    in which 1000 sail of the lime might ride in perfect security."
```

```
In[•]:= SequenceAlignment[textA, textD]
```

If we want to compare sequences at the level of words, we break each text into words with the TextWords command first.

```
In[•]:= SequenceAlignment[TextWords@textA, TextWords@textD]
```

Edit distance

The difference between texts that are close to one another but not exactly the same is often referred to as edit distance, the number of one-element deletions, insertions or substitutions required to transform the first into the second.

```
In[•]:= EditDistance[textA, textB]
```

```
In[•]:= EditDistance[textA, textC]
```

```
In[•]:= EditDistance[textA, textD]
```

This can be quite useful at the character level, for studying things like OCR errors or misspellings of proper names.

Distance matrix

The DistanceMatrix command computes the edit distance between every pair of texts. In the first row and column we have *textA*, in the second row and column *textB*, etc. Note that the diagonal of the

matrix is 0, since the edit distance of text from itself is 0. What is the edit distance between *textD* and *textC*? Is the edit distance the same between *textC* and *textD*? What property of the matrix reflects the answer to this question?

In[•]:=
```
DistanceMatrix[{textA, textB, textC, textD}] // MatrixForm
```

The MatrixPlot command helps us to visualize edit distances between each pair of texts. Darker colors represent higher values.

In[•]:=
```
MatrixPlot@DistanceMatrix[{textA, textB, textC, textD}]
```

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 41-44.

Teach Yourself TEI

A collection of readings, manuals and tutorials for TEI.

https://tei-c.org/support/learn/teach-yourself-tei/

Tutorial: XML in *Mathematica*

How XML is represented in *Mathematica* and how to import, transform and export it.

https://reference.wolfram.com/language/XML/tutorial/Overview.html

References

Beals, M. H. “Transforming Data for Reuse and Re-publication with XML and XSL.” *The Programming Historian* 5 (2016) https://programminghistorian.org/en/lessons/transforming-xml-with-xsl

Calishain, Tara. *Information Trapping: Real-Time Research on the Web*. New Riders (2006).

ch14

Lesson 14. Studying Societies

Computational social science

Social phenomena can be studied with a variety of computational methods. These include analysis of the so-called ‘big data’ that results from online activities (especially search and social media) and the modeling and simulation of behavioral, demographic and network data. We explore a number of examples of computational social science in this lesson, beginning with search.

The database of intentions

In 2003, John Battelle wrote, “The Database of Intentions is simply this: The aggregate results of every search ever entered, every result list ever tendered, and every path taken as a result. It lives in many places, but three or four places in particular hold a massive amount of this data (i.e., MSN, Google, and Yahoo). This information represents, in aggregate form, a place holder for the intentions of humankind – a massive database of desires, needs, wants, and likes that can be discovered, subpoenaed, archived, tracked, and exploited to all sorts of ends. Such a beast has never before existed in the history of culture, but is almost guaranteed to grow exponentially from this day forward. This artifact can tell us extraordinary things about who we are and what we want as a culture. And it has the potential to be abused in equally extraordinary fashion.”

Clearing all definitions and defining *viewData*

```
In[•]:= Clear["Global`*"]
```

```
In[•]:= viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Google Trends

The Google Trends site allows you to explore the use of search terms by region, date and topic. It has various kinds of visualization, including maps and graphs, shows trends and longer term patterns. You can put together a query (e.g., Pho vs. Ramen vs. Soba, Worldwide, 2004-Present) and download a CSV of the results.

https://trends.google.com/trends/?geo=US

The Google Trends team also curates CSV search datasets in a Datastore at GitHub and we will use some of these for our examples.

https://googletrends.github.io/data/

Searching for Chernobyl

This worldwide search dataset was collected in the week after the 30th anniversary of the 1986 nuclear disaster at Chernobyl (World, Apr 26, 2016).

```
In[•]:= chernobylSearch = Import[
   "https://raw.githubusercontent.com/googletrends/data/master/20160426_Chernobyl2
     .csv"];
```

As is often the case with CSV data files, the first line contains information about the file itself.

```
In[•]:= chernobylSearch[[1]]
```

The second line contains headers for each of the fields. Again, this is often the case.

```
In[•]:= chernobylSearch[[2]]
```

Here are the first ten records in table form, with header fields.

```
In[•]:= chernobylSearch[[2 ;; 12]] // TableForm
```

We could continue to use the data in this form, and in most programming languages that would be our only option. In *Mathematica*, however, we can make use of a command called SemanticImport which interprets data semantically, replacing lower-level representations with entities and returning a dataset.

```
In[•]:= chernobylSearchDataset = SemanticImport[
    "https://raw.githubusercontent.com/googletrends/data/master/20160426_Chernobyl2
      .csv", {"Country", "Integer", "Integer"},
    ExcludedLines → 1, HeaderLines → 1];
```

```
In[•]:= chernobylSearchDataset
```

```
In[•]:= chernobylSearchDataset[[1, 1]]
```

Now the country fields are entities, which makes it much easier to map the search values with GeoRegionValuePlot. Does this map more-or-less match your expectations? Are there any surprises?

```
In[•]:= GeoRegionValuePlot[
 Rule @@@ chernobylSearchDataset[[All, 1 ;; 2]], ImageSize → Scaled[0.8]]
```

Historical consciousness

Historical consciousness is the study of the ways that people understand and make use of the past in the present. People often search for information about a subject on anniversaries of events related to that subject, making search behavior one index of historical consciousness. If we look at Wikipedia search history for Chernobyl, we expect to see the same pattern. We request a time series of daily page hits using WikipediaData.

```
In[•]:= chernobylWikipediaPageHits = WikipediaData["chernobyl disaster", {"DailyPageHits"}]
```

Now we plot the time series with DateListLogPlot. The vertical axis is a log scale, which means that each mark on the scale is equal to the value of the previous mark multiplied by some number. If a linear scale is like a ruler with tick marks at 1, 2, 3, ... then a log scale is like a ruler with tick marks at 1, 10, 100, ... The advantage of the log plot is that it makes it easier to see differences between both large and small values. On this plot, I have also included grid lines at the April 26th anniversaries of the disaster. As expected, there are spikes of Wikipedia searching on the anniversaries of the disaster.

```
In[•]:= DateListLogPlot[chernobylWikipediaPageHits, ImageSize → Scaled[0.6], GridLines →
   {{"April 26, 2016", "April 26, 2017", "April 26, 2018", "April 26, 2019"}, None}]
```

The figure also shows that there are spikes of searching on other days, and these anomalies are of great interest in using searches to understand social behavior. Here are twenty days with the highest number of searches. Note that they all occurred in the spring of 2019.

```
In[•]:= MaximalBy[chernobylWikipediaPageHits["DatePath"], Last, 20]
```

We can extract a portion of the time series to focus on with TimeSeriesWindow.

```
In[•]:= cherobylWikipediaPageHitsSpring2019 =
         TimeSeriesWindow[chernobylWikipediaPageHits, {{2019, 4, 1}, {2019, 6, 30}}]
```

Here we plot the vertical axis on a linear scale. Note that there is no spike corresponding to the anniversary itself.

```
In[•]:= DateListPlot[cherobylWikipediaPageHitsSpring2019,
         ImageSize → Scaled[0.6], GridLines → {{"April 26, 2019"}, None}]
```

Let's extract the time series before this anomalous search behavior and look at that, too. A log plot is more useful in this case.

```
In[•]:= chernobylWikipediaPageHitsBeforeSpring2019 =
         TimeSeriesWindow[chernobylWikipediaPageHits, {{2015, 7, 1}, {2019, 3, 31}}]
```

```
In[•]:= DateListLogPlot[chernobylWikipediaPageHitsBeforeSpring2019,
         ImageSize → Scaled[0.6], GridLines →
          {{"April 26, 2016", "April 26, 2017", "April 26, 2018", "April 26, 2019"}, None}]
```

Here we plot the forty days before spring 2019 with the highest number of searches. Now we can see there is a cluster of searches around the 2016 anniversary (the 30th anniversary). We can also see that there are a few peaks that don't correspond to anniversaries.

```
In[•]:= DateListLogPlot[
         MaximalBy[chernobylWikipediaPageHitsBeforeSpring2019["DatePath"], Last, 40],
         ImageSize → Scaled[0.6], GridLines →
          {{"April 26, 2016", "April 26, 2017", "April 26, 2018", "April 26, 2019"}, None},
         Joined → False, Filling → Axis]
```

So to summarize, we see spikes of search interest corresponding to anniversaries, especially major ones. We also see days that have anomalously increased search activity which is not necessarily commemorative. Researchers at Leibniz Universität in Hannover, Germany showed that people also consult historical precedents in Wikipedia during high-impact events such as natural disasters (Kanhabua et al, 2014). During the 2011 Fukushima Daiichi nuclear disaster, for example, there were not only greatly increased Wikipedia page hits for Fukushima, but also for the Chernobyl disaster which occurred 25 years earlier. Wikipedia serves as a repository for global collective memory.

We still haven't explained the sudden surge of interest in the Chernobyl disaster in the spring of 2019, however. One way to do this is to return to Google Trends and see what other search terms people were using with "chernobyl". I did this on June 13, 2019 and archived the CSV file of results on GitHub. Importing the file, we see "HBO", "Television series", "Television show" and "Episode" as highly ranked terms, and "Stellan Skarsgård", "Jared Harris", "Emily Watson" as 'breakout' terms, which means that they were increasing at an anomalous rate in that period. Trending, in other words.

```
In[•]:= Import[
         "https://github.com/williamjturkel/Digital-Research-Methods/raw/master/chernobyl-
           search-related.csv"]
```

Look up the TV show in Wikipedia and get a summary.

```
In[•]:= WikipediaData["chernobyl (miniseries)", "SummaryPlaintext"]
```

If we wanted to explore further, we could look up each date of anomalous activity and try to match it to something. Here is how we find the first date in the spring of 2019 with more than 100K Wikipedia page hits.

```
In[•]:= SelectFirst[cherobylWikipediaPageHitsSpring2019["DatePath"], #[[2]] > 100 000 &]
```

Searching for water in Africa

Here is another example collected on World Water Day (March 22, 2018). This dataset shows worldwide Google search interest over the previous year in environmental issues impacting water availability. We start with SemanticImport, dropping one of the columns in the CSV file we won't need.

```
In[•]:= waterSearchDataset = SemanticImport[
  "https://raw.githubusercontent.com/googletrends/data/master/20180322
    _WorldWaterDay.csv", {"None", "Country", "Integer",
   "Integer", "Integer", "Integer"}, ExcludedLines → 1, HeaderLines → 1]
```

Suppose we want to focus on Africa. We can select African countries with an EntityList.

```
In[•]:= EntityList[ Africa COUNTRIES ]
```

```
In[•]:= africanCountryWaterSearchDataset = Select[waterSearchDataset,
  MemberQ[EntityList[ Africa COUNTRIES ], #["Country"]] &]
```

```
In[•]:= Length[africanCountryWaterSearchDataset]
```

We can represent the search interest in droughts vs the interest in floods with a pie chart. Here is how we do that for the second entry (Mozambique). If you mouse over the sections of the pie you will see the corresponding values.

```
In[•]:= PieChart[{africanCountryWaterSearchDataset[[2, 2]],
   africanCountryWaterSearchDataset[[2, 3]]}, ImageSize → Small]
```

For each country, we can calculate position and area. Here is how we do that for Mozambique.

```
In[•]:= africanCountryWaterSearchDataset[[2, 1]]
```

```
In[•]:= africanCountryWaterSearchDataset[[2, 1]]["Position"]
```

```
In[•]:= africanCountryWaterSearchDataset[[2, 1]]["Area"]
```

Now we plot the drought vs flood pie chart for each country using GeoBubbleChart. We use the area of each country to scale the pie chart.

```
In[•]:= waterPlotData =
  Map[#["Position"] → #["Area"] &, africanCountryWaterSearchDataset[[All, 1]]];
```

```
In[•]:= waterPlotChartElements = Table[PieChart[{africanCountryWaterSearchDataset[[c, 2]],
          africanCountryWaterSearchDataset[[c, 3]]}],
        {c, 1, Length[africanCountryWaterSearchDataset]}];

In[•]:= GeoBubbleChart[waterPlotData,
         ChartElements → waterPlotChartElements, ImageSize → Scaled[0.8]]
```

Social media

Mathematica includes methods for accessing your own social media accounts for a number of platforms. When you use these features a web browser will open so you can log into your account. Once connected, you can retrieve information about your friends, followers, comments, likes, topics etc. If you do use social media, try applying some of the techniques you learn in this lesson when you are reviewing it.

```
In[•]:= SocialMediaData[]
```

A Twitter dataset

Instead of assuming that you have an account on any particular platform, we are going to use a sample of public Twitter statuses from Europe, April 6-8 2016 that is stored in the Wolfram Data Repository.

https://datarepository.wolframcloud.com/resources/38c8ca48-d8de-4563-bc46-44eb1ff6c674

We retrieve the ResourceObject and view the data as follows.

```
In[•]:= ResourceObject["Geotagged Public Tweets (Europe, April 6-8 2016)"]

In[•]:= ResourceData["Geotagged Public Tweets (Europe, April 6-8 2016)"]
```

Here is a plot of the locations of tweets in the Estonian language. As you can see, a large number of them come from outside Estonia.

```
In[•]:= GeoGraphics[{EdgeForm[Black], FaceForm[Red],
          Polygon[Estonia COUNTRY ⋯ ✓], Blue, Opacity[0.3],
          Point@ResourceData["Geotagged Public Tweets (Europe, April 6-8 2016)"][
            Select[#lang == Entity["Language", "Estonian"] &]][All, "geo"]},
         GeoRange → EntityClass["Country", "Eurozone"]]
```

We can calculate the median friend count for accounts in each language if we use the GroupBy command on the language field. Nepali is an outlier here.

```
In[•]:= Sort[Median /@ GroupBy[DeleteDuplicates[
           ResourceData["Geotagged Public Tweets (Europe, April 6-8 2016)"][
            All, {"lang", "screen_name", "friends_count"}]],
          "lang"][All, All, "friends_count"], Greater]
```

Here are the text fields of English language tweets in this dataset.

```
In[•]:= englishTweetText =
          ResourceData["Geotagged Public Tweets (Europe, April 6-8 2016)"][
            Select[#lang == Entity["Language", "English"] &]][All, "text"];

In[•]:= First@englishTweetText
```

Here are the twenty most popular hash tags for English language tweets.

```
In[•]:= TakeLargestBy[Tally@
           Normal@Flatten[StringCases[#, "#" ~~ WordCharacter ..] & /@ englishTweetText],
          Last, 20] // TableForm
```

Here we select language and counts for a single user. Note that this user posts in two languages, that the statuses count increases with each tweet, and that s/he gains and/or loses friends and followers over time.

```
In[•]:= Select[ResourceData["Geotagged Public Tweets (Europe, April 6-8 2016)"],
          #["screen_name"] == "ARTEM_KLYUSHIN" &][All,
         {"screen_name", "lang", "statuses_count", "friends_count", "followers_count"}]
```

Social network analysis

17th-century networks of Friends

In this section we do some social network analysis, loosely following a tutorial in the *Programming Historian* by John Ladd, Jessica Otis, Christopher N. Warren and Scott Weingart.

https://programminghistorian.org/en/lessons/exploring-and-analyzing-network-data-with-python

Their data concerns Quakers (members of the Society of Friends) in the mid-to-late seventeenth century. It comes from the Oxford Dictionary of National Biography and from an ongoing project on the social networks of early modern Britain (1500-1700) called the Six Degrees of Francis Bacon. The dataset consists of two CSV files, a list of nodes and a list of edges. We import each as datasets.

```
In[•]:= quakerNodes = Import[
          "https://programminghistorian.org/assets/exploring-and-analyzing-network-data-
            with-python/quakers_nodelist.csv", "Dataset", HeaderLines → 1]

In[•]:= quakerEdges = Import[
          "https://programminghistorian.org/assets/exploring-and-analyzing-network-data-
            with-python/quakers_edgelist.csv", "Dataset", HeaderLines → 1]
```

As we saw in the Paul Revere example, we can use Graph to display some of these connections. Here are the people that George Keith was linked to. Since we are dealing with a network of acquaintance, the relationships are reciprocal. If George Keith knew William Penn, we assume that William Penn knew George Keith. So we use UndirectedEdge to represent these relationships (rather than Rule or DirectedEdge, which are both directional).

```
In[•]:= Graph[Normal[UndirectedEdge @@@ Select[quakerEdges, #Source == "George Keith" &]],
  VertexLabels → "Name", GraphLayout → "SpringEmbedding"]
```

Components

If we plot the whole collection of relationships, we can see that there is a large component of connected nodes, a smaller component consisting of three people and two relationships, and ten components that consist of a single relationship between a pair of people.

```
In[•]:= qe = Graph[Normal[UndirectedEdge @@@ quakerEdges]]
```

We can get a list of the different components with the ConnectedComponents command.

```
In[•]:= ConnectedComponents[qe]
```

Hubs and vertex degree

The number of connections that a given vertex has is known as its degree. If we scale the size of vertices to reflect the degree of each, it becomes more clear which vertices are most highly connected. These are known as hubs. *Mathematica* doesn't have a default option to do this scaling, so I adapted a function from the StackExchange forum.

```
In[•]:= qeScaled =
   SetProperty[qe, VertexSize → Thread[VertexList@qe → Rescale[VertexDegree@qe,
        {Min[VertexDegree@qe], Max[VertexDegree@qe]}, {.5, 3}]]];
```

```
In[•]:= Show[qeScaled, ImageSize → Scaled[0.8]]
```

Graph density

The density of a graph is the ratio of the number of actual edges divided by the total number of possible edges in a graph with the same number of vertices.

```
In[•]:= EdgeCount[qe]
```

As a fraction.

```
In[•]:= GraphDensity[qe]
```

As a floating point number.

```
In[•]:= N[GraphDensity[qe]]
```

Graph distance and diameter

The length of the shortest path between any two vertices is the graph distance. Here is how we calculate the distance between two vertices.

```
In[•]:= GraphDistance[qe, "Margaret Fell", "George Whitehead"]
```

Here is the shortest path between the two people.

In[•]:=
```
FindShortestPath[qe, "Margaret Fell", "George Whitehead"]
```

Here are the distances from Margaret Fell to all other vertices. Note that one node is zero (Margaret herself) and a number are Infinity (people who didn't belong to her network).

In[•]:=
```
GraphDistance[qe, "Margaret Fell"]
```

The diameter of the network is the longest of all the shortest paths. Since some of the members are not connected at all, the command returns Infinity.

In[•]:=
```
GraphDiameter[qe]
```

If we extract the largest component with ConnectedGraphComponents and then measure its diameter, we find that there are at most 8 steps between any two vertices.

In[•]:=
```
GraphDiameter[First@ConnectedGraphComponents[qe]]
```

Transitivity

If two people know a third person, it is likely that they know one another. Given that both Margaret Fell and George Whitehead knew William Penn, chances are that they knew one another, too. But when we look at the part of the graph containing the three people we are talking about, we can see that it is not closed to form a triangle.

In[•]:=
```
Subgraph[qe, {"Margaret Fell", "William Penn", "George Whitehead"}]
```

On the other hand relationships between these three people do form a triangle.

In[•]:=
```
Subgraph[qe, {"George Fox", "William Penn", "George Whitehead"}]
```

This property of a network can be measured with the GlobalClusteringCoefficient command. Given all of the paths of length 2, it measures the fraction that are closed to form a triangle.

In[•]:=
```
GlobalClusteringCoefficient[qe]
```

In[•]:=
```
N[177/1045]
```

Centrality

In the lesson on Paul Revere we looked at some measures of a vertex's centrality. In this network, we calculate the top ten vertexes by degree with the following expression.

In[•]:=
```
TakeLargestBy[MapThread[List, {VertexList[qe], DegreeCentrality[qe]}], Last, 10] //
 TableForm
```

As Ladd et al note, "Eigenvector centrality cares if you are a hub, but it also cares how many hubs you are connected to. It's calculated as a value from 0 to 1: the closer to one, the greater the centrality. Eigenvector centrality is useful for understanding which nodes can get information to many other nodes quickly. If you know a lot of well-connected people, you could spread a message very efficiently."

In[•]:=
```
TakeLargestBy[MapThread[List, {VertexList[qe], EigenvectorCentrality[qe]}],
  Last, 10] // TableForm
```

Finally, the betweenness centrality measure helps us to find brokers, people who connect otherwise separate parts of the network.

```
In[•]:= TakeLargestBy[MapThread[List, {VertexList[qe], BetweennessCentrality[qe]}],
  Last, 10] // TableForm
```

Suppose we want to compute a measure for some subset of the vertices. Here are the women in the dataset.

```
In[•]:= quakerWomen = Normal@Select[quakerNodes, #Gender == "female" &][[All, 1]]
```

Here are the five with the highest betweenness centrality.

```
TakeLargestBy[Select[MapThread[List, {VertexList[qe], BetweennessCentrality[qe]}],
   MemberQ[quakerWomen, #[[1]] &], Last, 5] // TableForm
```

Detecting communities

If a group of vertices is relatively densely connected to one another, but relatively loosely connected to other vertices, we say that it forms a community. The CommunityGraphPlot command can be used to identify these communities and visualize them. Here we apply it to the largest component in the social network.

```
In[•]:= CommunityGraphPlot[First@ConnectedGraphComponents[qe], ImageSize → Scaled[.6]]
```

We can get a list of the members of each with FindGraphCommunities.

```
In[•]:= FindGraphCommunities[First@ConnectedGraphComponents[qe]]
```

Management science

In this book I mostly focus on methods that can be used by all researchers using digital sources, regardless of their discipline or profession. That said, *Mathematica* can obviously be used to solve the problems that are specific to particular disciplines, not only in science and engineering, but in the humanities, social sciences and business as well. For example, there are a number of problems involving networks that arise in management science, the interdisciplinary study of optimal (or near optimal) solutions to complex problems. Here are two examples adapted from the *Mathematica* documentation.

Transportation networks

Suppose we have a railroad network serving six major Canadian cities (adapted from this example). Here are the cities.

```
In[•]:= cdnRailCities = {Vancouver CITY, Calgary CITY,
    Edmonton CITY, Regina CITY, Saskatoon CITY, Winnipeg CITY};
```

These are the connections between each pair of cities.

```
In[•]:= cdnRailEdges = GeoPath /@
 {{Saskatoon CITY, Calgary CITY},
  {Edmonton CITY, Saskatoon CITY}, {Edmonton CITY, Calgary CITY},
  {Vancouver CITY, Edmonton CITY}, {Saskatoon CITY, Winnipeg CITY},
  {Calgary CITY, Regina CITY}, {Regina CITY, Saskatoon CITY},
  {Regina CITY, Winnipeg CITY}, {Vancouver CITY, Calgary CITY}};
```

A certain number of cars travels in these directions each day. For example, there are 9 cars daily traveling west from Saskatoon to Calgary.

```
In[•]:= cdnRailCars = {"9W", "12E", "10S,4N", "16E", "20E", "14E", "7N", "4E", "13E"};
```

We can plot the vertices (cities) and edges (rail lines) of this network using GeoListPlot. Since we are using entities for our vertices, they are placed correctly on the map.

```
In[•]:= GeoListPlot[{cdnRailCities, MapThread[Labeled, {cdnRailEdges, cdnRailCars}]},
 GeoLabels → True, PlotStyle → {Automatic, None},
 GeoRangePadding → Quantity[4, "AngularDegrees"], ImageSize → Scaled[.8]]
```

In geospatial form, we can extract meaningful measures such as geodesic distances. For example,

```
In[•]:= GeoDistanceList[
  {Vancouver CITY, Calgary CITY, Saskatoon CITY, Winnipeg CITY}] // Normal
```

```
In[•]:= Total[{Quantity[676.786, "Kilometers"],
   Quantity[526.881, "Kilometers"], Quantity[711.576, "Kilometers"]}]
```

We can also represent our railroad network as a graph. In this case the topology of the network is the same, but the cities aren't laid out in their correct geospatial positions. That doesn't matter for the graph computations we will be doing.

```
In[•]:= cdnRailGraph =
 Graph[{"Vancouver", "Calgary", "Edmonton", "Regina", "Saskatoon", "Winnipeg"},
  {{{1, 2}, {1, 3}, {2, 3}, {3, 2}, {2, 4}, {3, 5}, {5, 2}, {4, 5}, {4, 6}, {5, 6}}, Null},
  EdgeCapacity → {13, 16, 4, 10, 14, 12, 9, 7, 4, 20}, VertexLabels → "Name",
  EdgeLabels → {DirectedEdge["Saskatoon", "Calgary"] → 9,
    DirectedEdge["Edmonton", "Saskatoon"] → 12,
    DirectedEdge["Edmonton", "Calgary"] → 10, DirectedEdge["Vancouver",
      "Edmonton"] → 16, DirectedEdge["Saskatoon", "Winnipeg"] → 20,
    DirectedEdge["Calgary", "Regina"] → 14, DirectedEdge["Calgary", "Edmonton"] → 4,
    DirectedEdge["Regina", "Saskatoon"] → 7, DirectedEdge["Regina", "Winnipeg"] → 4,
    DirectedEdge["Vancouver", "Calgary"] → 13}]
```

Suppose we want to find the maximum number of cars that can be carried from Vancouver to Winnipeg. For this, we use the FindMaximumFlow command.

```
In[•]:= FindMaximumFlow[cdnRailGraph, "Vancouver", "Winnipeg"]
```

Vancouver to Saskatoon.

```
In[•]:= FindMaximumFlow[cdnRailGraph, "Vancouver", "Saskatoon"]
```

Assignment problems

Another example of network optimization is assignment problems. For example, a research team has 6 RAs, each well suited to some tasks and not others.

```
raGraph = Graph[ ... ]
```

The maximum flow from all RAs to tasks gives the number of simultaneous tasks. (The assumption here is that only one person can be working on a task at a time).

```
In[•]:= raAssignment =
  FindMaximumFlow[raGraph, {"Janet", "Mark", "Erica", "Preet", "Jami", "Chris"},
   {"Library", "Archive", "Digitization", "Organization", "Communication"},
   "OptimumFlowData", VertexCapacity → ConstantArray[1, 11]];
```

```
In[•]:= raAssignment["FlowValue"]
```

Here is one possible arrangement of task assignments.

```
In[•]:= raAssignment["FlowGraph"]
```

Other examples of assignment problems in academia include mapping professors to possible courses given teaching preferences, mapping classes to classrooms, conference reviewers to abstracts, etc.

Further examples

Mythological figures

This example is adapted and condensed from one in the documentation.

http://www.wolfram.com/language/12/cultural-and-historical-entities/explore-and-compare-mythological-figures.html?product=mathematica

The FilteredEntityClass command allows you to select entities that match a particular EntityFunction. For example, here is how you would select mythological figures that shared any of the symbols of Athena.

```
In[•]:= Athena MYTHOLOGICAL FIGURE [ ... ] ["Dataset"]
```

```
In[•]:= Athena MYTHOLOGICAL FIGURE [ ... ] ["Patronages"]
```

```
In[•]:= athenaSymbols =
  FilteredEntityClass["Mythology", EntityFunction[s, ContainsAny[s["Symbols"],
      Athena MYTHOLOGICAL FIGURE ··· ✓ ["Symbols"]]]] // EntityList

In[•]:= GroupBy[athenaSymbols, #["Culture"] &]
```

Using FilteredEntityClass, we find all of the mythological figures associated with wisdom and generate a graph of their associated patronages.

```
In[•]:= Graph[Flatten[Function[e, e[[1]] ↔ # & /@ e[[2]]] /@
    FilteredEntityClass["Mythology", EntityFunction[s,
      ContainsAny[s["Patronages"], {"wisdom"}]]][{"Entity", "Patronages"}]],
  VertexLabels → "Name", GraphLayout → "RadialEmbedding",
  ImageSize → Scaled[.8]]
```

Learning more

An Elementary Introduction to the Wolfram Language, 2nd ed.

http://www.wolfram.com/language/elementary-introduction/2nd-ed/

Suggested: Chapters 45-47.

Google Trends Lessons

Google has a number of online lessons that teach journalists how to tell stories with (their) data.

https://newsinitiative.withgoogle.com/training/

https://newsinitiative.withgoogle.com/training/lessons?tool=Google %20 Trends&image=trends

Guide: Social Network Analysis

https://reference.wolfram.com/language/guide/SocialNetworks.html

Cheat Sheet: Social Network Analysis for Humanists

A one page overview of SNA terminology and concepts by Marten Düring (Apr 13, 2015).

https://cvcedhlab.hypotheses.org/106

Website: Visualizing Historical Networks

This site from the Center for History and Economics at Harvard University has a number of historical social network projects to explore.

http://histecon.fas.harvard.edu/visualizing/index.html

Code Gallery: Visualize Celebrity Gossip

A nice detailed example of studying a network of communities using data on visits to Wikipedia pages of celebrities.

http://www.wolfram.com/language/gallery/visualize-celebrity-gossip/

SNAP: Stanford Large Network Dataset Collection

A large collection of networks of various kinds to study, including social networks, communication networks, citation and collaboration networks, co-purchased products, roads, votes, reviews, news phrases, etc.

http://snap.stanford.edu/data/

Some of these have already been added to the Wolfram Data Repository, so it is usually most convenient to check there first.

https://datarepository.wolframcloud.com

Example: Maximum Flows and Minimum Cost Flows

This historical example is based on the 1940 Soviet railway network. It shows how to find and visualize the maximum amount of cargo that can be transported from sources in the Western Soviet Union to destinations in Eastern Europe.

http://www.wolfram.com/mathematica/new-in-9/enhanced-graphs-and-networks/maximum-flows-and-minimum-cost-flows.html

References

Battelle, John. “The Database of Intentions.” *John Battelle’s SearchBlog* (Nov 13, 2003). https://battellemedia.com/archives/2003/11/the_database_of _intentions

Kanhabua, Nattiya, Tu Ngoc Nguyen and Claudia Niederée. “What triggers human remembering of events? A large-scale analysis of catalysts for collective memory in Wikipedia.” *JCDL ‘14 Proceedings of the 14th ACM/IEEE-CS Joint Conference on Digital Libraries*, 341-350. London, United Kingdom (September 08-12, 2014).
https://dl.acm.org/citation.cfm?id=2740828

Ladd, John, Jessica Otis, Christopher N. Warren, and Scott Weingart. “Exploring and Analyzing Network Data with Python.” *The Programming Historian* 6 (2017).
https://programminghistorian.org/en/lessons/exploring-and-analyzing-network-data-with-python

Lemercier, Claire. “Formal network methods in history: why and how?” *Social Networks, Political Institutions, and Rural Societies*, Brepols, pp.281-310 (2015), 978-2-503-54804-3. 10.1484/M.RURHE-EB.4.00198
https://halshs.archives-ouvertes.fr/halshs-00521527/document

ch15

Lesson 15. Extracting Keywords

Information retrieval

Information retrieval (IR) is the process of querying a computer-based system to find relevant information in a content collection (e.g., text, audio, video, etc.) Prior to the growth of personal computing and the advent of the web, IR was a fairly specialized field that mostly concerned librarians. Now, of course, the search industry is key to the activities of many Fortune 500 companies.

One of the goals of an IR system is to return *relevant* information, and that means that the system must be capable of both determining what particular documents are about, and measuring their similarity. By hypothesis, the more similar two documents are to one another, the more likely they are to be relevant to the same query. In this lesson we focus on the problem of determining what documents are about by automatically identifying and extracting keywords. In the next lesson, we will focus on the measurement of similarity between words and documents.

Term frequency-inverse document frequency (TF-IDF)

Clearing all definitions and defining *viewData*

In[•]:=
```
Clear["Global`*"]
```

In[•]:=
```
viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Measuring the importance of a word

In order to figure out what a text is about, we need some way of measuring the importance of a given word in the text. In this section we will learn how to implement one widely used technique from the field of information retrieval, a measure called TF-IDF (term frequency-inverse document frequency). There are a lot of different ways to calculate TF-IDF, but they all capture the basic intuition that there are three categories of word.

1. A word that occurs on (practically) every page doesn't tell you anything special about a particular page. It is a stopword.
2. A word that occurs only once or is sprinkled a few times through the whole document or corpus of documents can probably be safely ignored.
3. A word that occurs a number of times on one page but is relatively rare in the document or overall corpus plays an important role in figuring out what that page is about.

We will introduce one simple method of calculating TF-IDF first and afterwards note some of the many varieties.

Sample text

For our example we will be using Darwin's *Origin of Species*, which is included in *Mathematica*'s sample data.

```
In[•]:= origin = ExampleData[{"Text", "OriginOfSpecies"}];
```

The Snippet command gives us a snippet of text from a document.

```
In[•]:= Snippet[origin, 5]
```

We will also be using a list of stopwords

```
In[•]:= stopwords = WordData[All, "Stopwords"];
```

Document frequencies

TF-IDF can be used at different levels. You might be comparing part of a document (e.g., abstract, paragraph, page, chapter) against the whole document. You might be comparing an abstract or a whole document against a corpus of documents. In any case, the term "document frequency" refers to the number of times that a term appears in the largest collection, whether that is a whole document, a corpus, or whatever.

Here are the document frequencies for the whole book.

```
In[•]:= originDocumentFrequency = WordCounts[ToLowerCase@origin];
```

The 50 most frequently occurring terms are mostly stopwords.

```
In[•]:= originDocumentFrequency[[1 ;; 50]]
```

Here are the 50 most frequently occurring words after we drop stopwords. We don't want to do this analysis on the whole book, so I use the Take command to grab the first 200 items in the association, then use KeySelect to pull the ones that are not stopwords.

```
In[•]:= KeySelect[Take[originDocumentFrequency, 200], Not[MemberQ[stopwords, #]] &][[1 ;; 50]]
```

Looking at this list we get a pretty good idea of what the whole book is about (natural selection; life; species, forms and varieties of animals and plants). Suppose, however, that we want to know what a portion of the book is about. Then we need to compare frequencies in that portion to the frequencies in the whole book.

Term frequencies for chapter 9 of *Origin*

To be concrete, let's say that we are interested in trying to figure out what Chapter 9 of *Origin* is about. We can use pattern matching to pull out Chapter 9 as a string.

```
In[•]:= originChapter9 =
  StringCases[origin, "CHAPTER 9. " ~~ Shortest[x__] ~~ "CHAPTER" → x][[1]];
```

In TF-IDF, "term frequency" simply refers to the number of times that a term appears in the smaller collection, whether it is an abstract, paragraph, page, chapter, or whatever. So the term frequencies

are simply the number of times each word appears in Chapter 9.

```
In[•]:= chapter9TermFrequency = WordCounts[ToLowerCase@originChapter9];
```

Here are the 50 most commonly occurring words in chapter 9 after we drop the stopwords.

```
In[•]:= KeySelect[Take[chapter9TermFrequency, 200], Not[MemberQ[stopwords, #]] &][[1 ;; 50]]
```

If you compare this list to the one generated above, you can get a sense of some of the similarities and differences. The word clouds below compare the frequencies for the book, on the left, with those for the chapter, on the right. In fact, term frequencies by themselves have sometimes been used for information retrieval tasks. But we will go on to use them to calculate the more power TF-IDF measure.

```
In[•]:= Row[{WordCloud[
    KeySelect[Take[originDocumentFrequency, 200], Not[MemberQ[stopwords, #]] &][[
     1 ;; 50]], ImageSize → Scaled[0.4]], WordCloud[
    KeySelect[Take[chapter9TermFrequency, 200], Not[MemberQ[stopwords, #]] &][[
     1 ;; 50]], ImageSize → Scaled[0.4]]}]
```

Calculating TF-IDF

Now that we have document frequencies for the whole book and term frequencies for the chapter, we have enough information to compute the TF-IDF for a given term. Here is a simple way to calculate the measure.

```
In[•]:= tfidf[termfreq_, docfreq_, numdocs_] :=
  Log[termfreq + 1.0] Log[numdocs / docfreq]
```

The number of documents in this case is the number of chapters in *Origin* (since we are comparing a chapter to the whole). If we were comparing a single page to the whole book, the number of documents would be equal to the total number of pages.

Recall that TF-IDF is designed to distinguish between three classes of word. First are words that occur on practically every page, i.e., stopwords. Let's look at the word 'the'. We need its term frequency for chapter 9, its document frequency and the number of documents (15).

```
In[•]:= {chapter9TermFrequency["the"], originDocumentFrequency["the"]}
```

```
In[•]:= tfidf[chapter9TermFrequency["the"], originDocumentFrequency["the"], 15]
```

Next are words that occur once. Here we will use a word that occurs only one time, in chapter 9. Note that its TF-IDF score is much higher than the score for a stopword. The word 'intermittence' is relatively more important for determining what chapter 9 is about than the word 'the'.

```
In[•]:= {chapter9TermFrequency["intermittence"], originDocumentFrequency["intermittence"]}
```

```
In[•]:= tfidf[chapter9TermFrequency["intermittence"],
  originDocumentFrequency["intermittence"], 15]
```

Finally, there are words that occur a number of times in the smaller unit (chapter in this case) but are relatively rare in the document overall. They are the most important for figuring out what the smaller unit is about, and thus should have the highest TF-IDF score.

```
In[•]:= {originDocumentFrequency["tapir"], chapter9TermFrequency["tapir"]}

In[•]:= tfidf[chapter9TermFrequency["tapir"], originDocumentFrequency["tapir"], 15]
```

Computing TF-IDF scores for every word in chapter 9

Now we simply map the TF-IDF function across the list of words in chapter 9 and store the results in an association.

```
In[•]:= chapter9TFIDFs = Association[
    # → tfidf[chapter9TermFrequency[#], originDocumentFrequency[#], 15] & /@
     Keys[chapter9TermFrequency]];
```

The fifty terms in chapter 9 that have the highest TF-IDF scores are listed below, in order of descending score. I have used the GatherBy command to group all of the terms with the same score. Take a moment to study this list and see if you can figure out what chapter 9 of *Origin* might be about.

```
In[•]:= Framed /@ Keys /@ GatherBy[TakeLargestBy[Normal@chapter9TFIDFs, Last, 50], Last] //
     Column
```

So what is chapter 9 of *Origin* about?

Look again at the high TF-IDF terms from Chapter 9: 'teleostean', 'pebbles', 'decay', 'conchologists', 'wear', 'tear', 'mineralogical', 'levels', 'grinding', 'gravel', 'sand', 'sedimentary', 'wears', 'wearing', 'watermark', 'tidal' ... The general sense is of geology, water, erosion, and fossils. When we look at Darwin's own summary for the chapter, we can see that this impression is accurate.

```
In[•]:= StringTake[originChapter9, 538]
```

In our example we used terms with high TF-IDF to figure out what Chapter 9 of *Origin of Species* is about. When used with a group of documents (a *corpus*), TF-IDF can help to automatically determine what each document is about and whether or not it is *relevant* to a particular search query.

Other ways of computing TF-IDF

The Wikipedia page on TF-IDF lists about a half a dozen options each for calculating TF and IDF and there is a large scholarly literature on extensions and adaptations of the method. A good place to start exploring is the classic text by Manning, Raghavan and Schütze, *Introduction to Information Retrieval*, which is available online.

Rapid automatic keyword extraction (RAKE)

TF-IDF has been in wide use for many decades because it is an excellent way of scoring documents for relevance given a query. (Think of the query as a very short text whose terms are being compared to the TF-IDF scores for the same words in all of the documents in the corpus.) One of the drawbacks of TF-IDF, however, is that you need to be able to compute document frequencies for a whole corpus, and that is often not possible or feasible. If you have a single webpage, you can't compute document

frequencies for the whole web. If you have a Wikipedia page, in theory you could compute document frequencies for all of Wikipedia to compare it with, but with ~6M articles in the English edition, that is usually not feasible. What you need instead is a way of extracting and ranking keywords that doesn't depend on being able to measure aspects of the whole corpus. Enter RAKE (Rose et al 2010).

Sample text

As our sample text, we are going to use the Wikipedia page for the subject "information overload".

```
In[•]:= sampleText = WikipediaData["information overload", "ArticlePlaintext"];
```

```
In[•]:= viewData@sampleText
```

Here is a list of unique lowercase words in our sample text.

```
In[•]:= sampleTextLowercaseWordList = Union@ToLowerCase@TextWords[sampleText];
```

Candidate generation

First the RAKE algorithm splits the text into sentences and generates a list of keyword candidates. Each of these candidates is a sequence of words delimited by stopwords or punctuation. The *stoplist* function below creates a list of these delimiters, and the *rakeSentenceToCandidates* function generates the candidates.

```
In[•]:= stoplist =
  WordBoundary ~~ # ~~ WordBoundary & /@ Complement[WordData[All, "Stopwords"],
    Join[CharacterRange["B", "H"], CharacterRange["J", "Z"],
     CharacterRange["b", "h"], CharacterRange["j", "z"], ToString /@ Range[0, 9]]];
```

```
In[•]:= rakeSentenceToCandidates[s_, stops_] :=
 DeleteCases[StringTrim /@ StringSplit[StringReplace[s,
     {PunctuationCharacter, stops} → " || ", IgnoreCase → True], " || "], ""]
```

Here is the first sentence in our sample text.

```
In[•]:= TextSentences[Snippet[sampleText, 10]][[1]]
```

Here are the keyword candidates.

```
In[•]:= rakeSentenceToCandidates[TextSentences[Snippet[sampleText, 10]][[1]], stoplist]
```

This next function generates the candidate list for the whole text.

```
In[•]:= rakeCandidateList[text_, stops_] :=
 TextWords /@
  Flatten[rakeSentenceToCandidates[#, stops] & /@ TextSentences[ToLowerCase@text]]
```

We apply it to our sample article and save the results.

```
In[•]:= sampleTextCandidateList = rakeCandidateList[sampleText, stoplist];
```

```
In[•]:= Short[sampleTextCandidateList, 3]
```

Pruning the candidates

For a relatively long document, some of the candidate keywords will be quite long. Here we remove ones that are five or more words in length.

```
In[•]:= sampleTextCandidateList = Complement[sampleTextCandidateList,
          Select[sampleTextCandidateList, Length[#] ≥ 5 &]];
```

Candidate scores

The next step is to compute two properties for each word, its frequency and its degree. You are already familiar with frequency. The degree is a measure that "favors words that occur often and in longer candidate keywords" (Rose et al 2010:7).

```
In[•]:= sampleTextWordFreqs = KeySort[Counts[Flatten@sampleTextCandidateList]];
```

```
In[•]:= sampleTextWordDegs = KeySort[
          Merge[Rule @@@ Flatten[Map[Thread, MapThread[List, {sampleTextCandidateList,
                Length /@ sampleTextCandidateList}]], 1], Total]];
```

Given these two measures, each word is scored as follows, and the output stored in an association.

```
In[•]:= sampleTextWordScores = N[sampleTextWordDegs / sampleTextWordFreqs];
```

```
In[•]:= Short[sampleTextWordScores, 3]
```

Now each keyword phrase can be scored, and the results stored in an association. Here are the 25 highest ranking keywords for the Wikipedia article on information overload. Look through the list and compare it with the original article.

```
In[•]:= sampleTextPhraseScores =
          ReverseSort[Association[MapThread[Rule, {sampleTextCandidateList,
              Table[Total[sampleTextWordScores /@ c], {c, sampleTextCandidateList}]}]]];
```

```
In[•]:= sampleTextPhraseScores[[1 ;; 25]] // Normal // TableForm
```

The RAKE function

Having worked through the steps we need to perform to extract keywords with RAKE, it will be more convenient to bundle all of the code into a function that returns an association of keyword phrase scores.

```
In[•]:= rake[txt_, stops_] :=
 Block[{clist, wfreqs, wdegs, wscores},
  clist = rakeCandidateList[txt, stops];
  wfreqs = KeySort[Counts[Flatten@clist]];
  wdegs = KeySort[Merge[Rule @@@
      Flatten[Map[Thread, MapThread[List, {clist, Length /@ clist}]], 1], Total]];
  wscores = N[wdegs / wfreqs];
  Return[ReverseSort[
    Association[MapThread[Rule, {clist, Table[Total[wscores /@ c], {c, clist}]}]]]]]
```

Abstracts and summary texts

RAKE really shines for shorter texts like summaries or abstracts.

```
In[•]:= sampleTextSummary = WikipediaData["information overload", "SummaryPlaintext"]
```

```
In[•]:= rake[sampleTextSummary, stoplist][[1 ;; 15]] // Normal // TableForm
```

Keyword entities

Persons

Having extracted keywords, you can try to match them to *Mathematica* entities. The expression below takes the high scoring RAKE keywords from the Wikipedia article on information overload, uses StringRiffle to turn lists of words into strings, then passes TextContents over each of the strings and returns Persons.

```
In[•]:= sampleTextPersons = Flatten[
  TextCases[#, "Person"] & /@ StringRiffle /@ Keys[sampleTextPhraseScores][[1 ;; 25]]]
```

Some of these persons have associated *Mathematica* entities.

```
In[•]:= DeleteCases[Interpreter["Person"] /@ sampleTextPersons, _Failure]
```

All of them have Wikipedia articles.

```
In[•]:= (WikipediaSearch /@ sampleTextPersons)[[All, 1]]
```

Get short summaries.

```
In[•]:= Table[Framed[StringRiffle@
    TextSentences[WikipediaData[per, "SummaryPlaintext"]][[1 ;; UpTo[2]]],
  {per, (WikipediaSearch /@ sampleTextPersons)[[All, 1]]}] // TableForm
```

Of course, you can also use persons or other entities to search for linked open data via SPARQL.

Learning more

Guide: Text Analysis

Sources of text; visualization, parsing and comparison; content extraction and analysis; text normalization.

https://reference.wolfram.com/language/guide/TextAnalysis.html

References

Manning, Christopher D., Prabhakar Raghavan and Hinrich Schütze. *Introduction to Information Retrieval*, Cambridge University Press. 2008.
https://nlp.stanford.edu/IR-book/information-retrieval-book.html

Rose, Stuart, Dave Engel, Nick Cramer and Wendy Cowley. "Automatic Keyword Extraction from Individual Documents." *Text Mining: Applications and Theory*, pp 1-20, Wiley. 2010.
https://onlinelibrary.wiley.com/doi/10.1002/9780470689646.ch1

ch16

Lesson 16. Word and Document Vectors

Information retrieval, continued

In the previous lesson we focused on the automatic extraction of keywords. Once we know how to extract keywords, we can make the process part of a workflow for batch downloading or crawling, and we can index keywords in documents and files and use them for both local and online searching.

In this lesson we continue our exploration of information retrieval, focusing on the measurement of similarity between words and documents. Many powerful information retrieval and text mining tools are based on the idea that aspects of a text can be represented in the form of vectors.

Speaking somewhat loosely, a *vector* is a mathematical quantity that has more than one component. To describe the distance along a ruler, you only need one measurement (say 2.5 centimeters from one end). To describe a position on a sheet of graph paper, however, you need two measurements (say three squares down and seven squares to the right of the upper left-hand corner). In *Mathematica*, we can write the resulting vector using list notation as {3, 7}. Note that the vector {7, 3} denotes a different position on our sheet of graph paper. Note also that your vectors are relative to an *origin*, a position where all of the dimensions are zero {0, 0}. In the graph paper example, the origin is the upper left-hand corner.

The document vector model

Clearing all definitions and defining *viewData*

In[•]:=
```
Clear["Global`*"]
```

```
In[•]:= viewData[x_] :=
         Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Bags of words and simple document vectors

Suppose we have three very short texts: “the cat is grey and the dog is black”, “my cat is fast and grey” and “my cat and my dog are friends”. In text mining and information retrieval, texts are often represented as *bags of words*. A bag of words is simply the list of words that occur in the text, with frequency information for each. We can create bag-of-words representations for our texts with WordCounts.

```
In[•]:= t1 = WordCounts["the cat is grey and the dog is black"]
```

```
In[•]:= t2 = WordCounts["my cat is fast and grey"]
```

```
In[•]:= t3 = WordCounts["my cat and my dog are friends"]
```

Here are all of the words used in all of the texts.

```
In[•]:= allwords = Union[Keys[t1], Keys[t2], Keys[t3]]
```

The Lookup command is used in the next expression to check the number of times each word in *allwords* occurs in the first text (*t1*). The word ‘and’ occurs once, the word ‘are’ occurs zero times, and so on, up to the word ‘the’ which occurs twice.

```
In[•]:= v1 = Lookup[t1, #, 0] & /@ allwords
```

Note that the output is a vector. Here are the vectors for the other two texts.

```
In[•]:= v2 = Lookup[t2, #, 0] & /@ allwords
```

```
In[•]:= v3 = Lookup[t3, #, 0] & /@ allwords
```

Here is a figure showing the vectors for all three texts.

```
In[•]:= TextGrid[Transpose[MapThread[List, {allwords, v1, v2, v3}]], Frame → All]
```

We can use the columns of this figure to compare the three texts. They all contain the word ‘and’. Only the third text contains the word ‘friends’ but the other two texts contain the word ‘grey’.

Intuitively, if two rows in the figure are identical then the two texts contain the same words with the same frequencies, but not necessarily in the same order. The texts "man bites dog" and "dog bites man" have the same bag-of-words representations but mean completely different things. Whenever we create a new representation of a text, we have to be aware of what information is being lost (word order, in this case).

The *allwords* list contains 11 columns. Each of these is a *dimension* of a space, and each of our text vectors in this example is a vector in an 11-dimensional space.

```
In[•]:= Length[allwords]
```

Cosine distance

There is a mathematical operation called the *cosine distance* that tells us how similar or different two vectors are, on a scale of 0 (identical) to 1 (completely different). By this measure *t1* and *t2* are most

similar to one another, *t2* and *t3* are a little more different, and *t1* and *t3* are the least similar.

```
In[•]:= N[CosineDistance[v1, v2]]
```

```
In[•]:= N[CosineDistance[v2, v3]]
```

```
In[•]:= N[CosineDistance[v1, v3]]
```

Scaling up

Now imagine that we want to compare a bunch of texts. In principle we could use exactly the same method to compare any group of texts: we represent each as a bag-of-words vector in the multidimensional space of the union of all words in all of our texts, and use cosine distance to measure similarity or difference. This will work, but we can make some refinements to make it work better. First, recall that the most frequent words in most texts are stopwords, and that these do very little to distinguish texts from one another. So instead of using a space of all words in all the texts, we might want to delete stopwords before we do our processing. Second, recall that some words do more to distinguish texts from one another than others, and that we used TF-IDF to measure the contribution of a given word to the distinctiveness of a particular text. So instead of using a space of all words (or all words once we've deleted stopwords), we might choose to use a space of high TF-IDF words. A third kind of refinement has to do with the similarity measure: we might find that other vector-based measures work better for assessing similarity and difference for particular applications.

We will also run into some problems as we scale up. One of these has to do with adding new texts. If we want our system to be dynamic, we have to deal with the possibility that new texts will require new dimensions to describe and that our multidimensional space will have to change as a result. Another problem has to do with texts of different lengths. A text like "man bites dog" will have zeros for all dimensions except three (and fewer if any of those three words is not a dimension), whereas a text like *War and Peace* will have thousands of non-zero dimensions. Comparing long texts to short texts may require more sophisticated methods. A third problem has to do with the overall number of dimensions. Comparing English language texts could conceivably require a space with hundreds of thousands of dimensions. We will need to do some sophisticated math to reduce the dimensionality of these spaces. Fortunately, *Mathematica* can handle these refinements and problems relatively easily.

Feature extraction and dimension reduction

Mathematica has high level commands for handling the process of extracting TF-IDF vectors for texts and then reducing the dimensions of the resulting space. In machine learning, a 'feature' is an individual property that can be measured for your data. Here the features are TF-IDF scores.

A small corpus of texts

In order to have some texts of varying lengths to work with, we are going to create a small corpus of Wikipedia articles. I've chosen these texts so that there are significant overlaps amongst the topics they cover, but also important differences.

```
In[•]:= wikipediaArticleCorpusTitles = {"stylometry", "corpus linguistics",
          "plagiarism", "forensic linguistics", "etymology", "trademark",
          "patent", "intellectual property", "fair use", "public domain"};
```

Next we create list of pairs. Each pair contains the title of the article and one of the sentences extracted from the article.

```
In[•]:= wikipediaArticleCorpus =
          Flatten[Table[List[w, #] & /@ TextSentences[WikipediaData[w]],
            {w, wikipediaArticleCorpusTitles}], 1];
```

```
In[•]:= First@wikipediaArticleCorpus
```

There are a few thousand sentences in all of the articles.

```
In[•]:= Length[wikipediaArticleCorpus]
```

Training and using a feature extractor function

Next we train a feature extractor function on the sentences. We specify that we want it to use TF-IDF features, then to reduce the number of dimensions of the resulting space.

```
In[•]:= wikipediaArticleExtractor =
         FeatureExtraction[foobWA[[All, 2]], {"TFIDF", "DimensionReducedVector"}]
```

The final step is to create a small function to query our system. This function needs the corpus of texts we are using, the feature extractor function, the query string, and the number of results we want.

```
In[•]:= queryWikipediaArticles[corpus_, fe_, qstr_, n_] :=
         MenuView[Rule @@@
           Table[corpus[[i]], {i, Nearest[fe[corpus[[All, 2]]] → Automatic, fe[qstr], n]}],
          ImageSize → Scaled[0.8]]
```

When we call the query function, it returns a sorted list of sentences in decreasing order of relevance. We use the pull down menu on the viewer to see each of the results.

For example, here is how we would ask for the top 10 results for the word ‘attribution’. Note that the results come from a variety of articles.

```
In[•]:= queryWikipediaArticles[wikipediaArticleCorpus,
         wikipediaArticleExtractor, "attribution", 10]
```

If we have a more elaborate query in mind, we can simply enter it as a string.

```
In[•]:= queryWikipediaArticles[wikipediaArticleCorpus,
         wikipediaArticleExtractor, "property rights", 10]
```

Word embeddings

A distributed representation for words

We have seen that it is possible to represent a text as a vector in a space where each dimension corresponds to one feature, whether the features are words, high TF-IDF terms, or some other linguistic unit. It is also possible to represent words themselves as vectors of real numbers, a technique known as word embedding. These representations are typically distributional, which means that the word is represented by a distribution of weights across dimensions. An example might make this more clear, so I have adapted an explanation given by Adrian Colyer (2016). His explanation is based, in turn, on the work of Mikolov et al (2013).

Say we are interested in the meanings of four terms: “king”, “queen”, “man” and “woman”. The meaning of each of these words has many dimensions, in the sense that the words “king” and “man” are more closely related to the concept of MASCULINITY than “queen” and “woman” are, and the latter words are more closely related to the concept of FEMININITY. “King” and “queen” are more closely related to ROYALTY than “man” and “woman”. As we imagine other words and other relationships, the network expands rapidly. Suppose we lay out a table of words (the columns) and dimensions (the rows), with an indication of whether a weight on a particular dimension is relatively high or low (say greater or less than 0.5 since we are talking about real numbers). We might end up with something like the following.

```
TextGrid[ ... ]
```

Each column is a vector that represents the meaning of a word as a distribution of real number valued weights across a space of conceptual dimensions. In word embeddings, however, the dimensions are not explicit or labelled. Instead, they arise implicitly as a result of training neural network models on text. A *lot* of text. It is not unusual to train with corpora that contain billions of tokens.

Summing up, Ben Schmidt (2015) writes “The exact operations of the new vector-space models aren’t always easy to figure out, but it’s easy to understand their central goals. 1. Word embedding models try to reflect similarities in usage between words to distances in space. ...
2. Word embedding models try to reflect similar relationships between words with similar paths in space.” We now look at examples of both.

Word vectors

The FeatureExtraction command of Mathematica includes access to semantic word vectors that come from a neural network trained on a large contemporary corpus. To continue our example, we call the function like this.

```
In[•]:= simpleWordVectors = FeatureExtraction[
          {"boy", "girl", "king", "man", "princess", "queen", "woman"}, "WordVectors"]
```

Words are represented in this space by real number valued vectors.

```
In[•]:= simpleWordVectors["boy"]
```

The space also includes representations of words we didn’t use when we called the command.

```
In[•]:= simpleWordVectors["boys"]
```

We can measure the distance between vectors in the space using geometric measures like EuclideanDistance.

```
In[•]:= EuclideanDistance[simpleWordVectors["boy"], simpleWordVectors["boys"]]
```

These vectors have many dimensions, so if we want to visualize them we need to reduce the number of dimensions. We create a function to reduce each vector to two dimensions so we can plot some of our words.

```
In[•]:= simpleWord2DVectors = Table[Labeled[DimensionReduce@oneclassic[a], a],
          {a, TextWords@{"king queen man woman"}}]
```

Now we use ListPlot to visualize.

```
In[•]:= ListPlot[simpleWord2DVectors[[1, 1]] → simpleWord2DVectors[[1, 2]],
         ImageSize → Scaled[.5]]
```

Algebraic operations

One of the surprising results that Mikolov et al (2013) discovered was that simple algebraic operations could be performed on word vectors. For example, they showed "*vector("King") – vector("Man")* + *vector("Woman")* results in a vector that is closest to the vector representation of the word *Queen*."

We can demonstrate this by doing the same operation then plotting the result. First we do the algebraic operation on the vectors.

```
In[•]:= calculatedMeaning =
          simpleWordVectors["king"] - simpleWordVectors["man"] + simpleWordVectors["woman"];
```

Next we define a dimension reducing function that we can use on the vector we just calculated.

```
In[•]:= simpleWordDimensionReducer = DimensionReduction[
          Flatten[Table[simpleWordVectors[w], {w, TextWords@{"king queen man woman"}}], 2]]
```

```
In[•]:= simpleWordDimensionReducer[calculatedMeaning]
```

Now we can Append the new data point and its label to the plot to visualize it in the space.

```
In[•]:= ListPlot[Append[simpleWord2DVectors[[1, 1]],
           First@simpleWordDimensionReducer[calculatedMeaning]] →
         Append[simpleWord2DVectors[[1, 2]], "King-Man+Woman"], ImageSize → Scaled[.5]]
```

Plurals

Word embeddings can also be used to reason by analogy. For example, king : kings :: queen : ?

Here, regularities of the singular/plural relation are captured by a relatively constant vector offset. We can visualize this with a plot.

```
In[•]:= plural2DVectors
```

```
In[•]:= plural2DVectors = Table[Labeled[DimensionReduce@simpleWordVectors[a], a],
           {a, TextWords@{"king kings queen queens"}}];
```

In[ ]:= Show[{…} +]

Historical word embeddings

Hamilton et al (2018) created word embeddings with a variety of historical corpora, leading to language models that were specific to a particular period. Projected into a lower dimensional space, these vectors could be used to visualize changes in word meaning and to study the rates of change in meaning over decades.

Here is a figure from their research.

In[ ]:=
```
Import[
 "https://web.archive.org/web/20190618203723/https://nlp.stanford.edu/projects/
   histwords/images/wordpaths.png"]
```

Reading these plots, we can see, for example, the word “gay” shifting from meaning something akin to “cheerful” or “flaunting” in the 1900s, to “witty” and “bright” in the 1950s, to its contemporary meaning of “homosexual”. The other two figures show the changes in meaning of “broadcast” and “awful” over the past 150 years. Their article, code and datasets are available online at

https://nlp.stanford.edu/projects/histwords/

Global vectors for word representation (GloVe)

A pretrained neural net

GloVe is another word embedding model that has been designed to capture statistical information regarding the co-occurrence patterns of words. There some pre-trained GloVe neural nets in the Wolfram Neural Net Repository. Here we download one that encodes 400K words using 100-dimensional word vectors. It was trained on a corpus of 6B word tokens.

https://resources.wolframcloud.com/NeuralNetRepository/resources/GloVe-100-Dimensional-Word-Vectors-Trained-on-Wikipedia-and-Gigaword-5-Data

Here is how we retrieve the pretrained net.

In[ ]:=
```
gLoVeNet = NetModel[
  "GloVe 100-Dimensional Word Vectors Trained on Wikipedia and Gigaword 5 Data"]
```

Visualizing features

Here is a an example from the documentation. It uses a FeatureExtractor to visualize words for animals and words for fruit. Note that the animals and fruits are clustered with one another, that common pairs like “cat” and “dog” or “crocodile” and “alligator” are near one another, and that there is a little cluster of tropical fruit.

```
In[•]:= animals = {"Alligator", "Ant", "Bear", "Bee", "Bird", "Camel",
    "Cat", "Cheetah", "Chicken", "Chimpanzee", "Cow", "Crocodile",
    "Deer", "Dog", "Dolphin", "Duck", "Eagle", "Elephant", "Fish", "Fly"};

In[•]:= fruits = {"Apple", "Apricot", "Avocado", "Banana", "Blackberry", "Blueberry",
    "Cherry", "Coconut", "Cranberry", "Grape", "Turnip", "Mango", "Melon",
    "Papaya", "Peach", "Pineapple", "Raspberry", "Strawberry", "Ribes", "Fig"};

In[•]:= FeatureSpacePlot[Join[animals, fruits], FeatureExtractor → gLoVeNet]
```

Neighboring words and word analogies

We can use the net model to find words that are near to one another in the space. Again, most of these examples come from the documentation.

```
In[•]:= gLoVeWords = NetExtract[gLoVeNet, "Input"][["Tokens"]];

In[•]:= gLoVeVectors = Normal@NetExtract[gLoVeNet, "Weights"][[1 ;; -2]];

In[•]:= word2vec = AssociationThread[gLoVeWords → gLoVeVectors];
```

Here are the eight words that are closest to “king”.

```
In[•]:= Nearest[word2vec, word2vec["king"], 8]
```

The eight words closest to “fish”.

```
In[•]:= Nearest[word2vec, word2vec["fish"], 8]
```

Here is the algebraic relation we looked at before. This time we retrieve the five nearest words to our calculated meaning.

```
In[•]:= Nearest[word2vec, word2vec["king"] - word2vec["man"] + word2vec["woman"], 5]
```

Here is an analogy problem France : Paris :: Germany : ?

```
In[•]:= Nearest[word2vec, word2vec["paris"] - word2vec["france"] + word2vec["germany"], 5]
```

Learning More

DH Transformations

Ben Schmidt, “Text Classification” and “Word Embeddings”

Blog post: New in the Wolfram Language: FeatureExtraction

https://blog.wolfram.com/2016/12/02/new-in-the-wolfram-language-featureextraction/

HistWords: word embeddings for historical text

Includes embeddings, link to paper, code on Github and data

https://nlp.stanford.edu/projects/histwords/

References

Colyer, Adrian. “The amazing power of word vectors.” *The Morning Paper* (Apr 21, 2016).
https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/

Hamilton, William L., Jure Leskovec and Dan Jurafsky. “Diachronic word embeddings reveal statistical laws of semantic change.” ACL 2016. *Arxiv.Org* (Oct 25, 2018).
https://arxiv.org/pdf/1605.09096.pdf

Mikolov, Tomas, Wen-tau Yih and Geoffrey Zweig. “Linguistic regularities in continuous word space representations.” *Proceedings of NAACL-HLT 2013*, pp 746-751 (2013).
https://www.aclweb.org/anthology/N13-1090

Schmidt, Ben. “Vector space models for the digital humanities.” *Ben’s Bookworm Blog* (Oct 25, 2015).
http://bookworm.benschmidt.org/posts/2015-10-25-Word-Embeddings.html

Widdows, Dominic. *Geometry and Meaning*. CSLI Publications, 2004.
http://www.puttypeg.net/book/
Chapter 5 “Word Vectors and Search Engines” is freely available online.

ch17

Lesson 17. Citations

The foundation of scholarship

Every scholar builds on the work of others and must acknowledge their contributions. Here we look at some ways that *Mathematica* can facilitate the handling of references and the study of citation. These include the use of *Mathematica*’s own entities and the retrieval of bibliographic data via web services and SPARQL. We explore the connection between crawling, search and citation and look at some possibilities for bibliometric analysis.

References and citation

Clearing all definitions and defining *viewData*

In[•]:=
```
Clear["Global`*"]
```

In[•]:=
```
viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Data about notable books and authors

Let’s start with *Mathematica* entities. There is metadata about more than 10K notable books which can be retrieved in Dataset form with an expression like the following.

In[•]:= Bleak House BOOK ["Dataset"]

Using the EntityClass command, we can extract a class of all books that meet some criterion. Here are the entities for notable books published during WWI.

In[•]:=
```
wwIBooks = EntityClass["Book",
  "FirstPublished" → Interval[{DateObject[{1914}], DateObject[{1918}]}]]
```

Here are the fields we can retrieve.

In[•]:=
```
wwIBooks["Properties"]
```

The EntityList command retrieves the entities in the class.

In[•]:=
```
EntityList[wwIBooks] // viewData
```

Here are the authors, titles and publication information for books in this class.

In[•]:=
```
wwIBookRefs = EntityValue[EntityList[wwIBooks],
   {"Author", "Name", "Publisher", "FirstPublished"}] // Dataset
```

Citation styles

One common scholarly task is to prepare citations according to a particular style. In the humanities, citations often have to be presented in the Notes and Bibliography style of the *Chicago Manual of Style*.

https://www.chicagomanualofstyle.org/tools_citationguide/citation-guide-1.html

In that style, bibliography entries look like this:

Grazer, Brian, and Charles Fishman. *A Curious Mind : The Secret to a Bigger Life*. New York : Simon & Schuster, 2015.
Smith, Zadie. *Swing Time*. New York : Penguin Press, 2016.

In the social sciences, the Author-Date style is more common.

https://www.chicagomanualofstyle.org/tools_citationguide/citation-guide-2.html

In that style, bibliography entries look like this:

Grazer, Brian, and Charles Fishman. 2015. *A Curious Mind: The Secret to a Bigger Life*. New York: Simon & Schuster.
Smith, Zadie. 2016. *Swing Time*. New York: Penguin Press.

There are literally thousands of different styles for various disciplines and journals, but the basic idea is always the same. You have bibliographic metadata that needs to be presented in a particular style for publication. Sometimes you have to format or edit entries by hand, but automation can get you 90-95% of the way to a clean bibliography.

Generating XML from a template

In *Mathematica* you can use the XMLTemplate command to format text that is generated from fields. For example, here are five books from the class we created above.

```
In[•]:= wwIBookRefs[[4 ;; 8]]
```

We need to process each record to get it into the correct format. The following function creates an association where each field name is a key. The surname of the first author entity is extracted and rearranged, and if there is no publisher, the notation “N.p.” is used.

```
In[•]:= processBiblioRecord[rec_] :=
 Block[{a = StringSplit[EntityValue[First@Normal[rec[[1]]], "Name"]],
   p = ToString@Normal[rec[[3]]]},
  Association[
   List[
    "author" → StringJoin[{Last[a], ", ", StringRiffle[Most[a]]}],
    "title" → rec[[2]],
    "publisher" → If[StringTake[p, 7] == "Missing", "N.p.", p],
    "date" → DateString[Normal@rec[[4]]]]]]
```

Here is a sample record in our dataset.

```
In[•]:= Normal[wwIBookRefs[[4]]]
```

Here is the record after it has been processed.

```
In[•]:= mccutcheon1914 = processBiblioRecord[wwIBookRefs[[4]]]
```

Next we need a template that specifies how to apply Chicago Notes and Bibliography style formatting and arrange the elements with the correct punctuation. Here we will format the references to display on the web, but we could easily use other XML tags for import into another program if that is what we wanted to do.

```
In[•]:= chicagoNBTemplate = XMLTemplate[
    "<li><wolfram:slot id='author'/>. <em><wolfram:slot id='title'/></em>.
      <wolfram:slot id='publisher'/>, <wolfram:slot id='date'/>.</li>"];
```

We apply it to a single reference like this. The output is an HTML fragment.

```
In[•]:= mccutcheon1914Ref = TemplateApply[chicagoNBTemplate, mccutcheon1914]
```

Here we apply the template to a list of references and assign the resulting list to a symbol. We also sort by first author’s last name at this point.

```
In[•]:= wwIBiblioEntries =
   Sort[Normal[TemplateApply[chicagoNBTemplate, processBiblioRecord[#]] & /@
      wwIBookRefs[[4 ;; 23]]]];
```

We’ve formatted each entry, but we didn’t format the page that will contain our bibliography. This also adds the hanging indents.

```
In[•]:= biblioTemplate = XMLTemplate["<!DOCTYPE html>
<html lang=\"en\">
<head>
<meta charset=\"utf-8\">
<title>Bibliography</title>
\t
<style media=\"all\">
ul, li {margin: 0; padding:0; list-style: none;}
li {padding-left:20px; text-indent: -20px;}
</style>
\t
</head>
<body>
<ul>
<wolfram:sequence values='#entries'><wolfram:slot id='1'/></wolfram:sequence>
</ul>
</body>
</html>"];
```

Finally, we use TemplateApply again to build an HTML string containing our entire bibliography.

```
In[•]:= wwIBiblio =
  TemplateApply[biblioTemplate, Association["entries" → wwIBiblioEntries]];
```

The EmbeddedHtml command takes an HTML string and creates a button. When you click on the button in your *Mathematica* notebook it opens the HTML in your default web browser. I've used the ImageSize option to make the browser window narrow enough to see the hanging indent formatting.

```
In[•]:= EmbeddedHTML[wwIBiblio, ImageSize → {350, Automatic}]
```

At this point I should stress that this is a toy example, and that writing software for creating clean bibliographies is a much larger undertaking. The system here has problems with author initials (look at the two periods in the entry for H.G. Wells, for example). It won't necessarily handle surnames like 'von Trapp' correctly, or hyphenated names or multiple authors. It doesn't include place of publication, or use the annotation "n.d." for missing dates or deal with all of the dozens of other kinds of references like chapters, articles, webpages, etc.

Citing Wolfram Alpha and *Mathematica* data

If you do use built-in entities or other computable knowledge from Wolfram Research, here is the information that you need to cite it properly.

http://support.wolfram.com/kb/23498

http://www.wolfram.com/knowledgebase/source-information/

Web services

A web service or API (application program interface) is an online program that is designed to interact with other programs. Here we explore a couple of web services that provide bibliographic information.

OCLC Classify

Classify is a web service from the Online Computer Library Center (OCLC). If you send a book ISBN in a URL to the service, the system will respond with a file of bibliographic information marked up as XML. Here I am requesting information about a history of the Cold War written by John Lewis Gaddis.

```
In[•]:= Import["http://classify.oclc.org/classify2/Classify?isbn=0143038273&summary=true",
 "Elements"]
```

In order to see the raw XML, I am converting it to a string before passing it off to *viewData*.

```
In[•]:= bibliographyXMLstring = StringRiffle[Flatten[Import[
      "http://classify.oclc.org/classify2/Classify?isbn=0143038273&summary=true",
      "Data"]], "\n"];
```

```
In[•]:= viewData[bibliographyXMLstring]
```

The bibliographic metadata is as follows.

The *work* tag contains information about the book, mostly stored in attributes like *author*, *editions*, *format* and *title*.

The *author* and *authors* tags contain information about the authors, some of it redundant.

The *lc* and *viaf* attributes of the *author* tag hold Library of Congress and OCLC identifiers, respectively. We will make use of these later.

The *ddc* tag contains Dewey Decimal call numbers (576.82092). You would use these to look for a copy of the book in most North American public libraries.

The *lcc* tag contains Library of Congress call numbers (QH31.D2) which indicate where the book would be filed in most North American academic libraries.

As we've seen, if we import as Symbolic XML, we can easily parse the data out of the file.

```
In[•]:= bibliographySymbolicXML = ImportString[bibliographyXMLstring, "XML"]
```

Author and title

```
In[•]:= First@Cases[bibliographySymbolicXML,
   XMLElement["work", {___, "author" → a_, ___, "title" → t_, ___}, _] → {a, t},
   Infinity]
```

Library of Congress call numbers

```
In[•]:= Cases[bibliographySymbolicXML,
  XMLElement["lcc", _, {XMLElement["mostPopular", {___, "sfa" → callno_, ___}, _],
     ___}] → callno, Infinity]
```

Open Library API

To connect to a web service we can use the *Mathematica* ServiceConnect command. It returns a ServiceObject.

```
In[•]:= openLibrary = ServiceConnect["OpenLibrary"]
```

To demonstrate the API, we will ask for up to twenty items authored by Walter Lippmann (1889-1974).

```
In[•]:= PersonData[ Walter Lippmann PERSON , {"Image", "BirthDate", "DeathDate"}] // TableForm
```

```
In[•]:= TextSentences[WikipediaData["Walter Lippmann"]][[1 ;; 7]]
```

Note that *openLibrary* in this context refers to the ServiceObject we just created.

```
In[•]:= lippmannResults =
  openLibrary["BookSearch", {"Author" → "Walter Lippmann", "MaxItems" → 20}]
```

We can use our usual commands for querying a dataset. Record for one book.

```
In[•]:= lippmannResults[2]
```

```
In[•]:= lippmannResults[2]["FirstPublishYear"]
```

```
In[•]:= lippmannResults[2]["PublishDate"]
```

```
In[•]:= lippmannResults[All, {"Title", "FirstPublishYear", "EditionCount"}]
```

Separate editions of a work each have their own Open Library ID. Here are the OLIDs for *A Preface to Politics*.

```
In[•]:= lippmannResults[2]["EditionKey"]
```

Get information about the first edition listed.

```
In[•]:= openLibrary["BookInformation", {"BibKeys" → lippmannResults[2, "EditionKey"][[1]]}]
```

Get a summary of the book

```
In[•]:= openLibrary["BookSummary", {"BibKeys" → lippmannResults[2, "EditionKey"][[1]]}]
```

Many works in the Open Library have text that can be retrieved and analyzed.

```
In[•]:= lippmannResults[2, "EditionKey"][[1]]
```

```
In[•]:= essentialLippmannText =
    openLibrary["BookText", {"BibKeys" → {{"OLID", "OL13540642M"}}}][1];
```

```
In[•]:= WordCloud[essentialLippmannText]
```

Book cover

```
In[•]:= Import["https://covers.openlibrary.org/b/id/6269110-M.jpg"]
```

Bibliographic linked open data

Bibliographic data is also available in the form of RDF triples. Here are a few examples.

Library of Congress

At the Library of Congress you can download authorities and vocabularies of many different kinds.

http://id.loc.gov/download/

This example is adapted from one in the documentation.

http://www.wolfram.com/language/12/rdf-and-sparql/import-rdf-data.html?product=mathematica

We request an authority file of Library of Congress Children's Subject Headings.

```
In[•]:= Needs["GraphStore`"]
```

```
In[•]:= childrensSubjects = Import[
  "http://id.loc.gov/static/data/downloads/authoritieschildrensSubjects.ttl.
    madsrdf.zip", "authoritieschildrensSubjects.madsrdf.ttl"]
```

We can use SPARQLQuery to retrieve a list of properties used by the triple store.

```
In[•]:= childrensSubjects // SPARQLQuery["select distinct ?p where {[] ?p []}"] //
  Query[All, "p"] // viewData
```

If we want to extract authoritative labels, we can use a query like this.

```
In[•]:= authoritativeLabels = childrensSubjects // SPARQLQuery["
    select * where {
      ?subject
       <http://www.loc.gov/mads/rdf/v1#authoritativeLabel> ?label .
    }
    "];
```

Here is the first such label. It shows us that a given subject number corresponds to Ceratopsians (horned dinosaurs).

```
In[•]:= First@authoritativeLabels
```

There are just over a thousand children's subject headings.

```
In[•]:= Length@authoritativeLabels
```

If we want to find all of the subjects that include the word 'animal' we could use an expression like the following.

```
In[•]:= Select[Replace[#["label"], RDFString[s_, _] → s] & /@ authoritativeLabels,
  StringContainsQ[#, "animal" | "Animal"] &] // TableForm
```

British national bibliography as LOD

The British National Bibliography is also available as linked open data from the British Library.

http://bnb.data.bl.uk/flint-sparql

Here is the endpoint.

```
In[•]:= bnbEndpoint = "http://bnb.data.bl.uk/sparql";
```

Construct a query string. This is one of the examples from the site: “Which titles by detective writer Ian Rankin appear in the BNB?”

```
In[•]:= rankinQuery = "SELECT DISTINCT ?title WHERE {
          ?book <http://purl.org/dc/terms/creator>
        <http://bnb.data.bl.uk/id/person/RankinIan>;
          <http://purl.org/dc/terms/title> ?title;
}";
```

Call SPARQLExecute.

```
In[•]:= SPARQLExecute[bnbEndpoint, rankinQuery]
```

Here is another example. We search for the history of the Cold War by John Lewis Gaddis that we used as an example earlier.

```
In[•]:= gaddisQuery = "SELECT ?book ?bnb ?title WHERE {
       ?book <http://purl.org/ontology/bibo/isbn13> \"9780141025322\";
          <http://www.bl.uk/schemas/bibliographic/blterms#bnb> ?bnb;
          <http://purl.org/dc/terms/title> ?title.
}";
```

```
In[•]:= SPARQLExecute[bnbEndpoint, gaddisQuery]
```

Here is the information related to that particular book. We are requesting all of the triples that have that resource ID as the subject.

```
In[•]:= Dataset@SPARQLExecute[bnbEndpoint, "select distinct ?predicate ?object where
         {<http://bnb.data.bl.uk/id/resource/013562453> ?predicate ?object}"]
```

Citation networks

Bibliometrics is the study of publications using statistical methods. A number of bibliometric measurements naturally take the form of networks: graphs representing which publications or which scholars cite other ones, graphs representing which scholars collaborate or co-author works with one another, and so on.

Co-authorship

In *Mathematica*’s example data, there is network graph of co-authorships in the field of network science.

```
In[•]:= coauthorships = ExampleData[{"NetworkGraph", "CoauthorshipsInNetworkScience"}]
```

We will choose the largest connected component to demonstrate a bibliographic measurement.

```
In[•]:= coauthorshipsSubgraph = ConnectedGraphComponents[coauthorships][[1]]
```

PageRank

One of the measurements we can do on the network of coauthorships is to measure the PageRankCentrality of each vertex. Here the vertices are highlighted according to this measure. Suppose you were randomly introduced to a researcher in network science. S/he introduces you to randomly chosen co-author, who introduces you to a randomly chosen co-author. What are the chances you would end up meeting a given person? The vertices with the highest PageRank have the largest diameter in the figure below.

```
In[•]:= HighlightGraph[coauthorshipsSubgraph, VertexList[coauthorshipsSubgraph],
  VertexSize → Thread[VertexList[coauthorshipsSubgraph] → 10 * Rescale[
      PageRankCentrality[coauthorshipsSubgraph, 0.85]]], ImageSize → Scaled[.8]]
```

If we had a graph of citations, PageRank would measure the likelihood of ending up at any particular reference after following a random chain of citations. If the vertices in our graph were webpages and the edges were hyperlinks, the PageRank measure would tell us the likelihood of reaching a particular page by randomly clicking on links.

PageRank is used by Google to determine how to rank websites returned during searches. More important websites tend to have more inbound links than less important ones, but they also tend to be linked to by other highly ranked sites. The original impetus for the design of the algorithm actually came from citation analysis: we know a scholar is important not only if he or she is cited frequently, but also if he or she is cited by other scholars who are cited frequently.

Further examples

Periodicals

Mathematica also has periodical entities. Here is the current number.

```
In[•]:= Length[EntityList["Periodical"]]
```

Here is how we retrieve information about a particular periodical.

```
In[•]:= The Washington Post PERIODICAL ["Dataset"]
```

Find periodicals with highest estimated circulation.

```
In[•]:= EntityClass["Periodical", {"EstimatedCirculation" → TakeLargest[10]}] // EntityList
```

Learning more

Title capitalization

Mathematica has a number of algorithms for doing this, including *Chicago Manual of Style*.

http://www.wolfram.com/language/12/core-language/text-capitalization.html?product=mathematica

Tutorial: Citation Management

Mathematica on Windows can connect to EndNote or a BibTeX database. Unfortunately this feature is not yet available for free alternatives like Zotero and Mendeley, and not available on OS X.

https://reference.wolfram.com/language/tutorial/CitationManagement.html

ch18

Lesson 18. Natural Language

Natural language processing

One advantage of programming with a very-high-level language like *Mathematica* is that knows how to handle natural languages above the level of characters and strings. These abilities fall under the general heading of natural language processing (NLP). Here we look at support for other languages, including alphabets and scripts, transliteration and translation, and dictionaries. We then turn to computational linguistics, analyzing natural language at the level of characters, words, sentences and meaning. Finally we look at measurements of emotional valence.

Multilingual analysis

The focus in this text is on working in English, but *Mathematica* has significant built-in knowledge about many languages, and has Unicode support so that research projects can be undertaken with any natural language. We look at a few of these features first.

Clearing all definitions and defining *viewData*

In[]:=
```
Clear["Global`*"]
```

In[]:=
```
viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Alphabets

The CharacterName command can be used to identify both Latin characters and non-Latin ones.

In[]:=
```
characterList = {"à", "ă", "ç", "ë", "ð", "ő",
    "þ", "ȣ", "ƹ", "ɸ", "ɘ", "א", "ɓ", "И", "Ѧ", "ע", "ɻ", "ש"};
```

In[]:=
```
# → CharacterName[#, "CommonName"] & /@ characterList // TableForm
```

The Alphabet command gives the alphabet for a particular language. Some languages (like Mandarin) don't have alphabets. The Mohawk and Slovak alphabets both use the Latin script, whereas Devanagari does not.

In[]:=
```
Alphabet["Mohawk"]
```

In[]:=
```
Alphabet["Slovak"]
```

In[]:=
```
Alphabet["Devanagari"]
```

The LetterNumber and FromLetterNumber commands can be used to compare alphabets. What position is the letter "z" in various European languages? Notice the use of the Thread command here.

```
In[•]:= europeanLanguageList = {"Croatian", "Dutch", "English", "French",
          "Hungarian", "Latvian", "Polish", "Portuguese", "Romanian", "Spanish"};

In[•]:= Thread[europeanLanguageList → LetterNumber["z", europeanLanguageList]]
```

What is the 25th letter in the alphabet of each of these languages?

```
In[•]:= Thread[europeanLanguageList → FromLetterNumber[25, europeanLanguageList]]
```

The AlphabeticSort command can be used to sort textual strings properly in different languages.

```
In[•]:= AlphabeticSort[
         {"Arboga", "Alingsås", "Åmål", "Ängelholm", "Askersund", "Arvika"}, "Swedish"]
```

The AlphabeticOrder command returns a 1 if the first string comes before the second, a -1 if the second comes before the first, and a 0 if they have identical sorting order.

```
In[•]:= AlphabeticOrder["a", "j"]

In[•]:= AlphabeticOrder["ñ", "n", "Spanish"]
```

Unicode

The Unicode standard is an attempt to represent the texts of all of the world's languages, and *Mathematica* now supports the full range of more than 1M characters. These can be specified using codes of two digits written as **\.xx** or four-digits written as **\:yyyy** or six digits written as **\|zzzzzz**

```
In[•]:= FullForm["ש"]

In[•]:= CharacterName["ש", "UnicodeName"]
```

Try starting an input cell and typing \| followed by 01F602

See this page for more details.

Multilingual dictionaries

The DictionaryLookup command works for a number of different languages.

```
In[•]:= DictionaryLookup[All]
```

Look up all Spanish words beginning with "c", ending with "n", and with an "ñ" in between.

```
In[•]:= DictionaryLookup[{"Spanish", "c" ~~ ___ ~~ "ñ" ~~ ___ ~~ "n"}]
```

Adapting an example from the documentation, we can use the multilingual dictionaries to show how the distribution of word lengths varies across languages.

```
In[•]:= Table[Histogram[StringLength /@ DictionaryLookup[{l, All}],
          {Range[25]}, ImageSize → 300, PlotLabel → l,
          PlotRange → {{0, 25}, All}], {l, {"Hebrew", "Hungarian"}}]
```

The overall number of words in each dictionary varies greatly, too.

```
In[•]:= {Length[DictionaryLookup[{"Hebrew", All}]],
 Length[DictionaryLookup[{"Hungarian", All}]]}
```

Language identification

The Classify command can be used for identifying the language that a text is written in. Here we get the language codes for a sample of natural languages.

```
In[•]:= sampleLanguages = EntityValue["Language", "SampleEntities"]
```

```
In[•]:= #[EntityProperty["Language", "Codes"]][[1]] & /@ sampleLanguages
```

Choose a language from this list at random and get the text of the Wikipedia main page.

```
In[•]:= randomLanguage = RandomChoice[sampleLanguages]
```

```
In[•]:= randomLanguageCode = randomLanguage["Codes"][[1]]
```

```
In[•]:= randomWikipediaMainpage =
   Import["http://" <> randomLanguageCode <> ".wikipedia.org/wiki/Main_Page"];
```

```
In[•]:= Snippet@randomWikipediaMainpage
```

```
In[•]:= Classify["Language", randomWikipediaMainpage]
```

Word translation and transliteration

The WordTranslation command translates words from one language to another.

```
In[•]:= WordTranslation["translate", "Spanish"]
```

```
In[•]:= WordTranslation["traducir", "Spanish" → "Dutch"]
```

```
In[•]:= WordTranslation["pannekoek", "Dutch" → All]
```

The Transliterate command translates from one script (i.e., writing system) to another. Greek to Latin.

```
In[•]:= Transliterate["Αλφαβητικός"]
```

Hiragana to Latin.

```
In[•]:= Transliterate["しんぱし"]
```

Hiragana to Hebrew.

```
In[•]:= Transliterate["しんぱし", "Hiragana" → "Hebrew"]
```

There is also a command called TextTranslate which uses an external service (and thus requires paid service credits). See the documentation for more details. Service credits are discussed in the Web Services lesson below.

https://reference.wolfram.com/language/ref/TextTranslation.html

Computational linguistics

As we have seen, we can use computers to analyze and synthesize various aspects of natural language. Here we look at different linguistic levels: characters, words, sentences, meaning.

Character frequencies

We've worked a lot with word and word n-gram frequencies, but character frequencies can be interesting, too. For example, here is how we get a list of common words in English.

```
In[•]:= commonEnglishWords = WordList[];
```

```
In[•]:= Length@commonEnglishWords
```

The LetterCounts command shows us the frequency of all letters. If you've ever tried to break a simple substitution cipher, you probably made use of the fact that letters like "e", "t" and "a" are among the most common in English. There is a more detailed example in the "Further examples" section of this lesson.

```
In[•]:= allCharacters = LetterCounts[StringJoin[commonEnglishWords], IgnoreCase → True]
```

If we just look at the first characters of words or the last characters of words, we see different distributions.

```
In[•]:= firstCharacters = ToLowerCase@StringTake[commonEnglishWords, 1];
```

```
In[•]:= ReverseSort[Counts[firstCharacters]]
```

```
In[•]:= lastCharacters = ToLowerCase@StringTake[commonEnglishWords, -1];
```

```
In[•]:= ReverseSort[Counts[lastCharacters]]
```

Character n-grams

We can use a related technique to count n-gram sequences of characters, rather than words.

```
In[•]:= allCharacterBigrams =
          CharacterCounts[StringJoin@commonEnglishWords, 2, IgnoreCase → True];
```

Here are the twenty most common character bigrams.

```
In[•]:= TakeLargest[allCharacterBigrams, 20]
```

If we look at the first two characters, we see the effect of common prefixes (e.g., copilot, unfamiliar, incredible, refurbish, dehumidify). Some common three character prefixes like dis- and str- will also have an effect here (e.g., dismay, strength).

```
In[•]:= firstCharacterBigrams = ToLowerCase@StringTake[commonEnglishWords, UpTo[2]];
```

```
In[•]:= TakeLargest[ReverseSort[Counts[firstCharacterBigrams]], 20]
```

The effect of common suffixes can be seen in the frequencies of the last two characters (e.g., slowly, sorted, running, runner).

```
In[•]:= lastCharacterBigrams =
          ToLowerCase@StringTake[Select[commonEnglishWords, StringLength[#] > 1 &], -2];

In[•]:= TakeLargest[ReverseSort[Counts[lastCharacterBigrams]], 20]
```

Character substitution and rearrangement

There are a number of puzzles and games that require rearranging or substituting characters. For example, what 10 words are closest to the string ‘pickel’ in terms of substitutions? We start by using a Nearest function.

```
In[•]:= nearPickel = Nearest[DictionaryLookup[]]["pickel", 10]
```

We can use the NearestNeighborGraph to plot the resulting network.

```
In[•]:= NearestNeighborGraph[nearPickel, VertexLabels → "Name"]
```

What are the words that match the pattern “p_ck__”?

```
In[•]:= DictionaryLookup["p" ~~ _ ~~ "ck" ~~ _ ~~ _]
```

We can get more information from the Nearest function by requesting all available properties. Here we create a dataset showing the words, the edit distance of each from “pickel”, and their index in the list of English words.

```
In[•]:= englishWords = WordList[];

In[•]:= Nearest[englishWords → All, "pickel", 10] // Dataset

In[•]:= englishWords[[3224]]
```

Here is how we would find the three longest words in the list of English language words.

```
In[•]:= TakeLargestBy[englishWords → "Element", StringLength, 3]
```

A related example to character substitutions is the problem of finding anagrams. Two words are anagrams if the sorted lists of characters in each is the same.

```
In[•]:= Sort@Characters["backward"]

In[•]:= Sort@Characters["drawback"]

In[•]:= Sort@Characters["backward"] == Sort@Characters["drawback"]
```

Here is a small function that finds anagrams.

```
In[•]:= anagrams[s_String] :=
         Module[{c = Sort[Characters[s]]},
          DictionaryLookup[x__ /; Sort[Characters[x]] == c]]

In[•]:= anagrams["backward"]

In[•]:= anagrams["restful"]
```

These kinds of techniques can also be used for more serious research processes, like cleaning up OCR or figuring out likely words when deciphering bad handwriting.

Decomposing words

The linguistic study of words is called morphology, and because words are relatively easy to parse in the texts of many languages there are a lot of computational techniques for working with them. One common task is to find the word stem. This is often used in designing search applications so that you don't have to specify plurals and other suffixed forms of your search terms.

```
In[•]:= WordStem["running"]
```

```
In[•]:= WordStem["inconsiderable"]
```

Some word stems are dictionary words in their own right, and others are not. Here is an example from the documentation for the WordStem command. We create a list of random words, stem each, and remove the ones that are identical to their stemmed form.

```
In[•]:= randomWords = RandomWord[30];
```

```
In[•]:= randomStems = WordStem[randomWords];
```

```
In[•]:= randomWordList = DeleteCases[Transpose[{randomWords, randomStems}], {w_, w_}];
```

Now we can visualize the ones that have dictionary words for stems. The code involves a couple of commands that you may not have seen yet, so spend some time figuring out how it works.

```
In[•]:= randomWordList =
 Replace[randomWordList, {w_, sw_?DictionaryWordQ} :> {w, Style[sw, Blue]}, {1}]
```

Word frequency over time

Mathematica has data for the relative frequencies of words in contemporary usage.

```
In[•]:= WordFrequencyData[{"the", "king", "lilylivered"}]
```

It is also possible to get time series data to see how word frequencies have changed over time.

```
In[•]:= DateListPlot[
 WordFrequencyData["veteran", "TimeSeries", {1900, Now}], PlotRange → Full]
```

```
In[•]:= DateListPlot[
 WordFrequencyData[{"war", "peace"}, "TimeSeries", {1900, Now}], PlotRange → Full]
```

Here is another example.

```
In[•]:= DateListLogPlot[WordFrequencyData[{"Walkman", "Discman"},
   "TimeSeries", {1978, 2010}, IgnoreCase → True], Filling → Axis]
```

Zipf's Law

If we plot word frequencies from most to least frequent we see that the curve has a particular shape: the frequency of a word is inversely proportional to its rank order. This is now known as Zipf's law, after the linguist who identified it in the mid-20th century. We use *Alice in Wonderland* as an example.

```
In[•]:= aliceWordCounts = WordCounts[ExampleData[{"Text", "AliceInWonderland"}]];
```

```
In[•]:= ListLogPlot[Values[aliceWordCounts], PlotRange → All]
```

Note that many words occur only a single time in a given text.

Mathematica also has some sophisticated commands that can find distributions from data.

```
In[•]:= FindDistribution[Values[aliceWordCounts]]
```

Entropy of a text

In information theory, entropy is a measure of uncertainty. If you are flipping a fair coin, the sequence of heads and tails that you have recorded up to a particular time doesn't matter. The next flip has an equal chance of being a head or tail. We say that it carries one *bit* of information. We can simulate a fair coin flip with the RandomInteger command.

```
In[•]:= RandomInteger[]
```

If we measure the entropy of 1000 simulated coin flips, the result should be very close to 1 bit per character.

```
In[•]:= N[Entropy[2, RandomInteger[1, 1000]]]
```

If entropy in natural language were maximized at the level of the alphabet, any character would be equally likely to follow any other. At the level of words, any word would be equally likely to follow any other. Natural language has rules, however, which make it less entropic and more redundant. In English, the letter 'u' always follows 'q'. Word positions within phrases can only be occupied by particular parts of speech.

Here is a word that *Mathematica* doesn't recognize.

```
In[•]:= DictionaryLookup["breen"]
```

In this phrasal position, it is most likely to be a noun or adjective.

```
In[•]:= TextStructure["The breen dog", "PartsOfSpeech"]
```

One of the examples in the documentation is a moving average plot of the entropy per word of the US Declaration of Independence.

```
In[•]:= declarationWords = TextWords@ExampleData[{"Text", "DeclarationOfIndependence"}];
```

```
In[•]:= ListLinePlot[MovingAverage[Entropy /@ declarationWords, 20]]
```

We can see that the entropy increases at the end of the document, meaning that in that part of the text there is more uncertainty about what the next word will be. If we look at the text we see that region is a block of signatures.

```
In[•]:= viewData[declarationWords[[1300 ;; -1]]]
```

We can compare this with number of meanings for each word (polysemy), which goes down abruptly in the same region.

https://reference.wolfram.com/language/example/PerformStatisticalSemanticAnalysis.html

```
In[•]:= ListLinePlot[
         MovingAverage[Length[WordData[ToLowerCase[#]]] & /@ declarationWords, 20],
         Filling → Axis, PlotRange → {0, All}, Frame → True]
```

Compression

One of the consequences of redundancy in natural language is that texts can be compressed. In raw (ASCII) text format, a single character requires 8 bits to represent. In English text, however, characters carry less than 8 bits of information on average. Here are the measurements for *Alice in Wonderland*.

```
In[•]:= StringLength[aliceInWonderland]
```

```
In[•]:= N[Entropy[2, aliceInWonderland]]
```

When we compress the text, we end up with fewer characters with higher average entropy for each.

```
In[•]:= aliceCompressed = Compress[aliceInWonderland];
```

```
In[•]:= StringLength[aliceCompressed]
```

```
In[•]:= N[Entropy[2, aliceCompressed]]
```

Text structure and parts of speech

Many words can be used in different ways.

```
In[•]:= PartOfSpeech["light"]
```

```
In[•]:= WordDefinition["light"] // TableForm // viewData
```

The linguistic context in which a given word appears helps to establish how it is being used. *Mathematica*'s TextStructure command parses sentences. One of the phases of parsing is assigning part of speech tags.

```
In[•]:= TextStructure[
         "The light went out, but the bags we were carrying were light.", "PartsOfSpeech"]
```

Parts of speech are combined into larger linguistic phrases.

```
In[•]:= TextStructure["The light went out, but the bags we were carrying were light."]
```

Information about structures that are nested within one another is often presented in the form of a tree. The figure above is equivalent to the one below.

```
In[•]:= TextStructure["The light went out, but the bags we were carrying were light.",
         "ConstituentGraphs"]
```

Meaning

The branch of linguistics concerned with meaning is called semantics, and the subbranch devoted to word meaning is called lexical semantics. Here is an example. Words with similar and opposite meanings.

```
In[•]:= Synonyms["light"]
```

```
In[•]:= Antonyms["light"]
```

This code is adapted from the documentation. It uses the recursive function NestList to find the synonyms for “ignite”, then the synonyms of each of those words in turn, then plots the resulting graph. So “erupt” is a synonym for “ignite”, and “burst” is a synonym for “erupt”. When you are reviewing this lesson, try to unpack the code to the extent that you can copy and modify it to create related graphs. What does the related graph of antonyms look like, for example?

```
GraphPlot[Flatten[Rest[NestList[
    Union[Flatten[Thread[# → Synonyms[#]] & /@ Last /@ #]] &, {"" → "ignite"}, 2]]],
 VertexLabels → "Name", ImagePadding → 25, ImageSize → Scaled[0.8],
 GraphStyle → "ThickEdge",
 EdgeStyle → Directive[Opacity[0.2], Thickness[0.01], Pink],
 Method → "SpringEmbedding"]
```

Sentiment

As we have seen in an earlier lesson, it is possible to use *Mathematica*’s built-in classifiers to determine the topic of a text. We used the TextContents command and looked for sentences with a high probability of being about a particular topic.

```
In[•]:= TextContents["Man I can't wait for the Braves game.", "SportsTopic", "Probability"]
```

```
In[•]:= TextContents["I want to check out this vegetarian restaurant.",
 "FoodAndDrinkTopic", "Probability"]
```

We can use a different classifier to sort sentences beforehand.

```
In[•]:= Classify["FacebookTopic", "Man I can't wait for the Raptors game."]
```

```
In[•]:= Classify["FacebookTopic", "I want to check out this vegetarian restaurant."]
```

When working with online sources in a professional context, it is sometimes necessary to detect profanity. Here is an example adapted from the documentation, which compares a random text from Wikipedia with a page in Wiktionary that is devoted to English swear words. Since swearing in social media is often a sign of emotional intensity, classifying these expressions is a more sophisticated approach than simply filtering them out.

```
In[•]:= Classify["Profanity", Import["http://en.wikipedia.org/wiki/Special:Random"]]
```

```
In[•]:= Classify["Profanity",
 Import["https://en.wiktionary.org/wiki/Category:English_swear_words"]]
```

Setting aside profanity, the Classify command can also be used to assess the sentiment of a text as positive or negative. Here is a message from the Enron email corpus.

https://www.cs.cmu.edu/~./enron/

```
In[•]:= enronMessage =
          "Like you, I am getting very frustrated with this process. I am genuinely
            trying to be as reasonable as possible. I am not trying to \"hold up\"
            the deal at the last minute. I'm afraid that I am being asked to take
            a fairly large leap of faith after this company (I don't mean the two
            of you -- I mean Enron) has screwed me and the people who work for me.";
```

We can test the sentiment of the first sentence like this.

```
In[•]:= Classify["Sentiment",
         "Like you, I am getting very frustrated with this process.", "Probabilities"]
```

We can also map the classifier across all of the sentences and get the system's best guess for each.

```
In[•]:= Map[Classify["Sentiment", #] &, TextSentences[enronMessage]]
```

Further examples

Simple ciphers

The StringReplace command takes a string and a list of replacements. You can use it to make a simple cipher. Here we replace only a single character.

```
In[•]:= StringReplace["This is a message to be encrypted", {"a" → "c"}, IgnoreCase → True]
```

We could type in our whole list of rules for encryption, but we will build the list using some list commands and MapThread instead.

```
In[•]:= CharacterRange["a", "z"]
```

```
In[•]:= Join[CharacterRange["c", "z"], {"a", "b"}]
```

```
In[•]:= encryptionRules = MapThread[Rule,
          {CharacterRange["a", "z"], Join[CharacterRange["c", "z"], {"a", "b"}]}]
```

Here is how we encrypt a message now

```
In[•]:= StringReplace["This is a message to be encrypted",
         encryptionRules, IgnoreCase → True]
```

To decrypt a message, we need to reverse each rule in our *encryptionRules*

```
In[•]:= decryptionRules = Map[Reverse, encryptionRules]
```

Now we can decrypt our message

```
In[•]:= StringReplace["vjku ku c oguucig vq dg gpetarvgf", decryptionRules]
```

Simple cryptanalysis

Because letter frequencies tend to be fairly consistent from one long English text to another, you can use frequency analysis to break simple substitution ciphers (like the one above). Let's start by encrypting the first 1000 characters of *Alice in Wonderland*, using our rules above.

```
In[•]:= aliceInWonderland = ExampleData[{"Text", "AliceInWonderland"}];

In[•]:= aliceCipherText = StringReplace[
          StringTake[aliceInWonderland, 1000], encryptionRules, IgnoreCase → True]
```

Now suppose we don't know what rules were used for the encryption. We start by counting letter frequencies in our ciphertext.

```
In[•]:= LetterCounts[aliceCipherText]

In[•]:= ListPlot[LetterCounts[aliceCipherText]]
```

Since g is the most common letter, let's hypothesize that is "E". The next most common is "v", so let's replace that with "T", and the third most common is "c", which we will replace with "A". Here is what we have so far...

```
In[•]:= StringReplace[aliceCipherText, {"g" → "E", "v" → "T", "c" → "A"}]
```

Looking at the text above, we see "TjE" and "TjAT", so "j" must be "h".

```
In[•]:= StringReplace[aliceCipherText, {"g" → "E", "v" → "T", "c" → "A", "j" → "H"}]
```

Now we see "uHE" and "HEt". Those are probably "SHE" and "HER".

```
In[•]:= StringReplace[aliceCipherText,
         {"g" → "E", "v" → "T", "c" → "A", "j" → "H", "u" → "S", "t" → "R"}]
```

Now we see "SkSTER", "hEET", "qR" and "STARTEf".

```
In[•]:= StringReplace[aliceCipherText, {"g" → "E", "v" → "T", "c" → "A",
           "j" → "H", "u" → "S", "t" → "R", "k" → "I", "h" → "F", "f" → "D", "q" → "O"}]
```

"pi" is obviously "NG", "d" is "B", and "y" is "W"

```
In[•]:= StringReplace[aliceCipherText,
         {"g" → "E", "v" → "T", "c" → "A", "j" → "H", "u" → "S", "t" → "R", "k" → "I",
          "h" → "F", "f" → "D", "q" → "O", "p" → "N", "i" → "G", "d" → "B", "y" → "W"}]
```

Your turn

Try using frequency analysis to decrypt the following ciphertext.

```
In[•]:= secretCipher =
  "qknxulpr1rturtfrnrupsukrsgtolpfnjjernqigmbjldyldruknurnrftgyjlrwngrtgrxmfflfft\
    mgrmhrnrymmdrhmpusglrwsfurvlrtgrbngurmhrnrbthlrkmblolprjtuujlrigmbgruklrh\
    lljtgyfrmprotlbfrmhrfsqkrnrwngrwnervlrmgrktfrhtpfurlgulptgyrnrgltykvmspkm\
    mdruktfrupsukrtfrfmrbljjrhtcldrtgruklrwtgdfrmhruklrfsppmsgdtgyrhnwtjtlfru\
    knurklrtfrqmgftdlpldruklrptykuhsjrxpmxlpuermhrfmwlrmglrmprmuklprmhrukltpr\
    dnsykulpfrwerdlnprwp.rvlgglurfntdrktfrjnderumrktwrmglrdnerknolremsrklnpdru\
    knurgluklphtljdrxnpirtfrjlurnurjnfurwp.
   rvlgglurplxjtldruknurklrkndrgmurvsurturtfrpluspgldrfklrhmprwpf.
   rjmgyrknfrzsfurvllgrklplrngdrfklrumjdrwlrnjjrnvmsurturwp.rvlgglurwndlrgmrn\
    gfblprdmremsrgmurbngurumrigmbrbkmrknfrunilgrturqptldrktfrbthlrtwxnutlguje\
    remsrbngurumruljjrwlrngdrtrknolrgmrmvzlqutmgrumrklnptgyrturuktfrbnfrtgotu\
    nutmgrlgmsykrbkerwerdlnpremsrwsfurigmbrwpf.rjmgyrfnefruknurgluklphtljdrtfr\
    unilgrvernremsgyrwngrmhrjnpylrhmpusglrhpmwruklrgmpukrmhrlgyjngdruknurklrq\
    nwlrdmbgrmgrwmgdnertgrnrqkntflrngdrhmsprumrfllruklrxjnqlrngdrbnfrfmrwsqkr\
    dljtykuldrbtukrturuknurklrnyplldrbtukrwp.rwmpptfrtwwldtnuljeruknurklrtfrum\
    runilrxmfflfftmgrvlhmplrwtqknljwnfrngdrfmwlrmhrktfrflpongufrnplrumrvlrtgr\
    uklrkmsflrveruklrlgdrmhrglcurbllirbknurtfrktfrgnwlrvtgyjlertfrklrwnpptldr\
    mprftgyjlrmkrftgyjlrwerdlnprumrvlrfsplrnrftgyjlrwngrmhrjnpylrhmpusglrhmsp\
    rmprhtolrukmsfngdrnrelnprbknurnrhtglruktgyrhmprmsprytpjfrkmbrfmrkmbrqngrt\
    urnhhlquruklwrwerdlnprwp.rvlgglurplxjtldrktfrbthlrkmbrqngremsrvlrfmrutplfm\
    wlremsrwsfurigmbruknurtrnwruktgitgyrmhrktfrwnppetgy";
```

Entropy, again

In order to decrypt these simple substitution ciphers the cryptanalyst makes use of the predictability of natural language. Note that the entropy of the cipher text is exactly the same as the entropy of the original text.

```
In[•]:= N[Entropy[ToLowerCase@StringTake[aliceInWonderland, 1000]]]
```

```
In[•]:= N[Entropy[aliceCipherText]]
```

One way to make a substitution cipher more difficult to crack (at least by hand) is to increase the entropy of the text before applying the cipher. Here is what the ciphertext looks like if we compress it before applying the encryption rules.

```
In[•]:= aliceCompressedCipherText = StringReplace[
   ToLowerCase@Compress@StringTake[aliceInWonderland, 1000], encryptionRules]
```

The entropy is now higher.

```
In[•]:= N[Entropy[aliceCompressedCipherText]]
```

```
In[•]:= LetterCounts[aliceCompressedCipherText]
```

And the frequency information is less useful.

```
In[•]:= ListPlot[LetterCounts[aliceCompressedCipherText]]
```

And since the plaintext has been compressed, the obvious relationship between frequent ciphertext characters and frequent plaintext characters has been lost.

In[•]:=
```
StringReplace[aliceCompressedCipherText,
 {"p" → "E", "r" → "T", "c" → "A", "m" → "H", "v" → "S", "u" → "R", "j" → "I",
  "d" → "F", "y" → "D", "k" → "O", "s" → "N", "z" → "G", "h" → "B", "l" → "W"}]
```

Learning more

Examine characteristics of scripts

Different languages can use the same writing system (script) while still having different alphabets. This example explores some of the alphabets that are written with Latin characters.

http://www.wolfram.com/language/11/knowledgebase-expansion/examine-characteristics-of-languages-alphabets-and.html?product=mathematica

Train a neural net classifier to detect the language of a text

Mathematica now has this functionality built in, but here is the code for training a classifier for this task.

http://www.wolfram.com/mathematica/new-in-10/highly-automated-machine-learning/detect-the-language-of-a-text.html

http://www.wolfram.com/language/gallery/determine-the-language-of-a-text/

Wolfram Demonstration: Rhyme finder

Uses phonetic information from *Mathematica*'s dictionary to find rhyming words.

http://demonstrations.wolfram.com/RhymeFinder/

Zipf's Law

This example adds more detail to our discussion by fitting the observed distribution to the predicted one.

http://www.wolfram.com/language/11/text-and-language-processing/zipfs-law.html?product=mathematica

Compare the structure of sentences

Two sentences have the same structure if their constituent graphs are identical. In this example the distance between constituent graphs is used to find sentences in Wikipedia articles with matching structures.

http://www.wolfram.com/language/11/text-and-language-processing/compare-the-structure-of-sentences.html?product=mathematica

Define grammar rules

This example shows how to build your own natural-language processing interface, to parse descriptions of routes and plot them on a map.

http://www.wolfram.com/language/11/text-and-language-processing/define-grammar-rules.html?product=mathematica

Synonyms and Antonyms

This example includes using the recursive function NestList to create synonyms of synonyms.

http://www.wolfram.com/language/12/core-language/synonyms-and-antonyms.html?product=mathematica

ch19

Lesson 19. Web Services

Application programming interface (API)

An API (application programming interface) consists of a set of computational building blocks which may be called by another program to achieve some goal. When the system is web based, it is also known as a *web service*. In this lesson we look at a number of web services and see how they can be used to build tools that draw on external data and processing.

Web services are accessible at a variety of levels in *Mathematica*, including through dedicated commands (e.g., WikipediaData), through the service framework (e.g., Twitter, Facebook) and through URL calls (e.g., MusicBrainz or Spotify as of June 2019).

Entity network crawling

Clearing all definitions and defining *viewData*

In[•]:=
```
Clear["Global`*"]
```

In[•]:=
```
viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Wolfram Service Credits

In *Mathematica*, web services can be accessed at a number of levels. The most transparent of these are bundled into specific commands. The WikipediaSearch and WikipediaData commands that we have used a number of times, for example, exchange information with Wikipedia without the programmer or user having to manage the details.

There are a number of very convenient commands that require Wolfram Service Credits that you have

to purchase. These commands include WebSearch, WebImageSearch, TextTranslation, GeoImage and SendMessage.

http://www.wolfram.com/service-credits/

In this text, I avoid using commands that require purchasing service credits because I don't want to add any additional burden to students who are using the text for coursework or self-study. That said, the commands that do require credits can be very handy for research projects that have a budget. You can check how many service credits you have available with the following expression.

In[•]:=
```
$ServiceCreditsAvailable
```

If you do design a program or project that requires service credits, make sure to read this forum on StackExchange first.

https://mathematica.stackexchange.com/questions/140261/v11-1-effective-methods-to-use-paid-functions-economically

Wolfram Cloud Credits

It is also possible to use *Mathematica* to expose your own API or interactive document on the web. These deployments typically require that you purchase Wolfram Cloud Credits.

http://www.wolfram.com/cloud-credits/

You can check how many cloud credits you have available with the following expression.

In[•]:=
```
$CloudCreditsAvailable
```

If you need to know more about your cloud account, you can use the CloudAccountData command to get details.

In[•]:=
```
(*CloudAccountData[]*)
```

Service Framework

Mathematica also includes a sophisticated service framework for connecting to particular sites. For many of these services you need to have a free or paid account (depending on the service). When you first try to connect to the service, a web browser will open for authentication. Once you have authenticated your account, you are returned to your notebook. Sites that can be accessed through the service framework but require authentication include social media sites (Twitter, Facebook, LinkedIn, Instagram, Reddit), storage (Flickr, Dropbox), personal (Google Calendar, Google Contacts), health (Fitbit, Runkeeper), and business search and services (Google Analytics, Yelp, SeatGeek). There is more information about a number of these web services in the "Project Ideas" section of this book.

In the lesson on Citations we used the service framework to access the Open Library API, which doesn't require service credits or authentication.

OCLC WorldCat Identities

Large databases of named entities often include links between records. The WorldCat Identities database from the Online Computer Library Center (OCLC) has a summary page for every name in

WorldCat, about 30 million entities including persons, organizations, and fictional characters. Each page also has “Related Identities” section which links one entity with another. By starting with any entity and following links to related pages, we can compile a map of the network around that entity.

https://www.worldcat.org/identities/

Let’s start with Walter Lippmann, who we met in the lesson on Citations. *Mathematica* does not have a connection to OCLC WorldCat Identities in the service framework, so we will access the API through URLs.

```
In[•]:= oclcWorldCatIdentitiesBaseURL = "http://worldcat.org/identities/";
```

This is Lippmann’s Library of Congress ID number

```
In[•]:= lippmannLCCN = "lccn-n80019504";
```

Here are the elements we can retrieve from Lippmann’s page.

```
In[•]:= Import[oclcWorldCatIdentitiesBaseURL <> lippmannLCCN, "Elements"]
```

This is what the page looks like in a browser.

https://www.worldcat.org/identities/lccn-n80019504/

Here is a function that, given an ID, returns names and associated IDs. We use it to retrieve entities that are connected to Lippmann.

```
In[•]:= getWorldCatIdentitiesANamesIDs[id_] :=
 StringCases[Import[oclcWorldCatIdentitiesBaseURL <> id, "Source"],
  Shortest["href=\"http://worldcat.org/identities/" ~~
     aid__ ~~ "\"" ~~ __ ~~ ">" ~~ a__ ~~ " </a>"] → {aid, a}]
```

```
In[•]:= getWorldCatIdentitiesANamesIDs[lippmannLCCN]
```

Let’s get the information for the Council on Foreign Relations now.

```
In[•]:= cfrLCCN = "lccn-n81061396";
```

```
In[•]:= getWorldCatIdentitiesANamesIDs[cfrLCCN]
```

This function uses *Mathematica*’s graph features to construct a list of links from an entity to associated names, and from those names to each of their neighbors in turn. Starting with Lippmann, we construct the neighborhood immediately around him, and the neighborhoods of his neighbors. This is known as a two-hop network.

```
In[•]:= getWorldCatIdentitiesNeighborhood[id_, namestr_] :=
 (* version 1 June 2018 *)
 Module[{neighbor1, outgraph, nname, ntemp},
  neighbor1 = getWorldCatIdentitiesANamesIDs[id];
  outgraph = Map[namestr → #[[2]] &, neighbor1];
  Do[
   nname = n[[2]];
   ntemp = getWorldCatIdentitiesANamesIDs[n[[1]]];
   outgraph = Join[outgraph, Map[nname → #[[2]] &, ntemp]],
   {n, neighbor1}];
  Return[outgraph]]
```

```
In[•]:= lippmannIDNeighborhood =
   getWorldCatIdentitiesNeighborhood[lippmannLCCN, "lippmann"];
```

```
In[•]:= Short[lippmannIDNeighborhood, 10]
```

We can use the Graph command to visualize the network we just created. Because it is large, we put it in a Pane.

```
In[•]:= Pane[Graph[lippmannIDNeighborhood, VertexLabels → Placed["Name", Center,
     Panel[#, FrameMargins → 0, Background → White] &], GraphStyle → "SimpleLink",
   GraphLayout → {"SpringElectricalEmbedding", "RepulsiveForcePower" → -2},
   EdgeShapeFunction → GraphElementData["ShortFilledArrow", "ArrowSize" → 0.004],
   EdgeStyle → Gray, ImageSize → {2200, 1600}],
  {Full, 400}, Scrollbars → True, ScrollPosition → {700, 600}]
```

Finally, we can also find and visualize communities around Lippmann.

```
In[•]:= FindGraphCommunities[lippmannIDNeighborhood] // viewData
```

Here I have used the Tooltip command so that if you hover over a vertex in the graph, it will show you the name. I have also labelled each of the communities in Lippmann's network based on the bridging person(s) or institution(s).

```
In[•]:= CommunityGraphPlot[lippmannIDNeighborhood, VertexLabels → Placed["Name", Tooltip],
  CommunityLabels → {"Lippmann", "Whitney Hart Shepardson and William O. Scroggs",
    "University of Virginia Library Electronic Text Center",
    "Allan Nevins and Stern Collection of Lincolniana",
    "Council on Foreign Relations", "Ronald Steel",
    "William E. Leuchtenburg"}, ImageSize → Scaled[0.7]]
```

Publication search

We have already seen that it is possible to access the Open Library API through the service framework freely and without authentication. Here are a few more examples of publication search APIs.

CrossRef

The CrossRef API can be used through the service framework to search for journal articles and other scholarly works. Here is the documentation page.

https://reference.wolfram.com/language/ref/service/CrossRef.html

Open a connection.

```
In[•]:= crossref = ServiceConnect["CrossRef"]
```

Search for journal article by keyword, limit to those issued in a given date range and return the first item.

```
In[•]:= crossrefWorks = crossref["WorksDataset",
    {"Query" → "self driving cars", "TypeID" → "journal-article",
     "IssuedDate" → {DateObject[{2019, 2, 1}], DateObject[{2019, 3, 1}]},
     "SortBy" → "Score", MaxItems → 20}];

In[•]:= First@crossrefWorks
```

Create a timeline of dates indexed.

```
In[•]:= TimelinePlot[crossrefWorks[All, "Indexed"]]
```

Search members.

```
In[•]:= crossrefMembers = crossref["MemberDataset", {"Query" → "philosophy", MaxItems → 10}];

In[•]:= crossrefMembers[All, {"PrimaryName", "Location", "ID"}]
```

Search works by member.

```
In[•]:= crossref["WorksDataset", {"MemberID" → crossrefMembers[7, "ID"]}]
```

Use ServiceDisconnect to close the connection.

```
In[•]:= ServiceDisconnect[crossref]
```

ArXiv

The ArXiv API can be used to find scientific preprints in a number of disciplines. It has a service framework connection.

```
In[•]:= arXiv = ServiceConnect["ArXiv"]
```

Here we request a maximum of four preprints related to a specific query and look at the second item.

```
In[•]:= arxivArticles = arXiv["Search", {"Query" → "word wrap", MaxItems → 4}];

In[•]:= arxivArticles[[2]]
```

Look at the abstract.

```
In[•]:= arxivArticles[[2, 6]]
```

Use ServiceDisconnect to close the connection.

```
In[•]:= ServiceDisconnect[arXiv]
```

There are a number of other examples of queries on the documentation page.

https://reference.wolfram.com/language/ref/service/ArXiv.html

PubMed

For medical research, the PubMed API can be accessed through the service framework.

```
In[•]:= pubmed = ServiceConnect["PubMed"]
```

Here is how we do an author and title search and sort the results by publication date.

```
In[•]:= pubmed["PublicationSearch", "Author" → "jarvik",
  "Title" → "artificial heart", "SortBy" → "PublicationDate"]
```

Retrieve information about a particular article.

```
In[•]:= pubmed["PublicationSearch", "ID" → "7010587", "Elements" → "FullData"][[1]]
```

Retrieve an abstract.

```
In[•]:= viewData@pubmed["PublicationAbstract", "ID" → "6690950"]
```

Use ServiceDisconnect to close the connection.

```
In[•]:= ServiceDisconnect[pubmed]
```

There are a number of other examples of queries on the documentation page.

https://reference.wolfram.com/language/ref/service/PubMed.html

Archive.org

By contrast with the previous examples, there is no service framework connection to Archive.org, but the site maintains a large number of APIs that return JSON. We can access these through URL commands.

For example, the Book Cover API is documented here.

https://archive.readme.io/docs/book-covers

```
In[•]:= URLExecute["https://archive.org/services/img/theworksofplato01platiala"]
```

The Item Metadata API is documented here.

https://archive.readme.io/docs/item

```
archiveOrgMetadata =
  Association@URLExecute["https://archive.org/metadata/principleofrelat00eins"];
```

Item title

```
In[•]:= Lookup[Association[archiveOrgMetadata["metadata"]],
  {"title", "creator", "year"}] // Column
```

File names and sizes

```
In[•]:= Cases[archiveOrgMetadata["files"],
  {"name" → n_, ___, "size" → s_, ___} → {n, s}] // TableForm
```

xkcd

Here is a less academic example of publication search. The webcomic xkcd has an API that can return JSON.

Here is how you request metadata for a comic.

```
In[•]:= xkcd = URLExecute["https://xkcd.com/730/info.0.json"];
```

```
In[•]:= viewData@xkcd
```

And how to request the comic itself.

```
In[•]:= Association[xkcd]["img"]
```

```
In[•]:= Import[Association[xkcd]["img"]]
```

Manipulating JSON

Upgrade JSON to linked data (JSON-LD)

If you are working with an API that returns JSON, you can import it as linked data using the "JSONLD" importer. Here we will demonstrate with a JSON API from the US Library of Congress.

https://libraryofcongress.github.io/data-exploration/

One of the collections is World War I sheet music. We use the Import command to query the collection for "boy" and return the results as JSON. The "JSONLD" and "ExpandContext" parameters tell the importer to convert the results to linked data using Schema.org

```
In[•]:= sheetMusicStore = Import[
  "https://www.loc.gov/collections/world-war-i-sheet-music/?q=boy&at=results&fo=
    json", "JSONLD", "ExpandContext" → "http://schema.org"]
```

The output is a triple store that we can query with SPARQL. Here is how we would ask for the titles, for example.

```
In[•]:= sheetMusicStore // SPARQLQuery["
  prefix schema: <http://schema.org/>
  select ?title where {
    ?item schema:title ?title .
  }"]
```

This next query returns links to cover image URLs.

```
In[•]:= sheetMusicCoverURLs = sheetMusicStore // SPARQLQuery["
  prefix schema: <http://schema.org/>
  select ?image where {
    ?item schema:image ?image .
  }"];
```

The first result

```
In[•]:= sheetMusicCoverURLs[[1]]
```

```
In[•]:= Import@sheetMusicCoverURLs[[1]]["image"]
```

Dashboards

A data dashboard is an interactive application that allows you to explore and visualize up-to-date information. In business, dashboards are often used to display 'key performance indicators' that indicate how the company is performing on various metrics. Since *Mathematica* makes it easy to retrieve up-to-date information via RSS feeds and APIs and easy to make interactive tools, it can be used to build dashboards of varying complexity.

Here is a toy example. It retrieves the RSS feeds of recent additions to the Biodiversity Heritage Library, the latest Kirkus Reviews and the New York Times book reviews, creating word clouds of the content.

https://www.biodiversitylibrary.org/recent

https://www.kirkusreviews.com

https://archive.nytimes.com/www.nytimes.com/services/xml/rss/index.html

The MenuView command creates a dynamic interface with a pull-down menu.

```
In[•]:= MenuView[{
  "BHL" →
   WordCloud[StringReplace[StringRiffle[
      Flatten[Cases[Import["http://www.biodiversitylibrary.org/RecentRss/100",
         "SymbolicXML"], XMLElement["title", _, t_] → t, Infinity]]],
     Shortest["(added: " ~~ __ ~~ ")"] → ""]],
  "Kirkus" → WordCloud[StringRiffle[StringReplace[Flatten[
       Cases[Import[" https : // www.kirkusreviews.com / feeds / rss ", "SymbolicXML"],
        XMLElement["description", _, d_] → d, Infinity]],
      {"'s" → "s", Shortest["<" ~~ __ ~~ ">"] → ""}]]],
  "NYT" →
   WordCloud[StringRiffle[StringReplace[Flatten[
       Cases[Import["https://rss.nytimes.com/services/xml/rss/nyt/Books.xml",
         "SymbolicXML"], XMLElement["description", _, d_] → d, Infinity]],
      {"'s" → "s", Shortest["<" ~~ __ ~~ ">"] → ""}]]]
 },
 1]
```

Learning more

Guide: Accessing External Services and APIs

https://reference.wolfram.com/language/guide/AccessingExternalServicesAndAPIs.html

Create a Graph of Contextual Nouns from H. G. Wells's *The War of the Worlds*

Example uses OpenLibrary API, full text parsing and network visualization to explore words related to Martian invaders.

http://www.wolfram.com/language/11/external-services/create-a-graph-of-contextual-nouns-from-h-g-wellss.html?product=mathematica

Tutorial: Library of Congress Data Exploration Guide

This tutorial shows how to work with the Library of Congress JSON API using Python.

https://github.com/LibraryOfCongress/data-exploration/blob/master/LOC.gov%20 JSON %20 API.ipynb

ch20

Lesson 20. Databases

Structured data

Mathematica's Dataset is the most structured option for representing data, as it is based on a hierarchy of lists and associations. When we say that data is structured, we are referring to the fact that we have a *data model*: an idea of the characteristics that describe entities and their relationships in a particular application. For example, suppose we want to keep track of genealogical information. For each person, we would like to have his or her name, birthdate and birthplace, death date and death place (if deceased), familial relationships and optional information like notes, photos, stories, voice recordings or videos, etc. We might create a database to store this information, putting it in a number of tables. Think of a table like a spreadsheet, with rows and columns. Each row is a record (e.g., an individual), each column a field (e.g., his/her surname). We would have a table of individuals, a table of families, and so on. One principle that we might follow would be to try not to replicate information. For example, if we are keeping track of four generations of a family that lived at one address, we would not want to keep typing in the full address for each person. Instead, we might want to have a table of significant locations that contained the addresses of each, then link a particular person (from the table of individuals) to a particular particular location for a period of time. The process of removing redundancies from data is called normalization.

In *Mathematica*, we can use the Dataset both as a powerful data structure in its own right, and as a way to represent external databases of both relational (SQL-like) and hierarchical, object store (NoSQL) varieties. Here we will focus on using datasets within *Mathematica*. If you need to access an external database, use the DatabaseLink toolkit or the EntityStore command (see the "Learning More" section for details).

Parts and structural operations

Clearing all definitions and defining *viewData*

```
In[•]:= Clear["Global`*"]
```

```
In[•]:= viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Spans, keys and values

Let's start by using a sample dataset about the planets.

```
In[•]:= planets = ExampleData[{"Dataset", "Planets"}]
```

We use the Part command to extract a single row. Here is the row for Mercury.

```
In[•]:= planets[[1]]
```

We use the Span command to extract a group of rows. Here are Earth and Mars.

```
In[•]:= planets[[3 ;; 4]]
```

Each row is an association. Here is the information about Earth's moon and Mars' moons.

```
In[•]:= planets[[3]]["Moons"]
```

```
In[•]:= planets[[4]]["Moons"]
```

By using the Normal command, we can see that a row is an association. We can see that it contains other associations. This is an example of the hierarchical nature of datasets. Each row is an association, and may contain other associations in turn.

```
In[•]:= planets[[4]]["Moons"] // Normal
```

The radius of Deimos.

```
In[•]:= planets[[4]]["Moons"]["Deimos"]["Radius"]
```

To extract a column, we need to specify which rows we are interested in. Here is how we retrieve the mass of all of the planets.

```
In[•]:= planets[All, "Mass"]
```

This column is an association, too.

```
In[•]:= planets[All, "Mass"] // Normal
```

Sum the mass of all the planets.

```
In[•]:= Total@Values[planets[All, "Mass"]]
```

We can extract a subset of rows and columns like this.

```
In[•]:= planets[2 ;; 3, {"Mass", "Radius"}]
```

We can extract a list of planets with Keys.

```
In[•]:= Keys@planets
```

Creating datasets

We can build a dataset from a list of lists. If we do it this way, it will not have names for either rows or columns.

```
In[•]:= Dataset[{{1, "x", {1}}, {2, "y", {2, 3}}}]
```

If we start with a list of associations, we will have named columns.

```
In[•]:= Dataset[{Association[{"a" → 1, "b" → "x", "c" → {1}}],
   Association[{"a" → 2, "b" → "y", "c" → {2, 3}}]}]
```

If we start with an association of lists, we will have named rows.

```
In[•]:= Dataset[Association[{"a" → {1, 2}}, {"b" → {"x", "y"}}, {"c" → {{1}, {2, 3}}}]]
```

Starting with an association of associations results in a table with names for both rows and columns.

```
In[•]:= Dataset[Association[{"a" → Association[{"first" → {1, 2}, "second" → ""}],
    "b" → Association[{"first" → {"x", "y"}, "second" → ""}],
    "c" → Association[{"first" → {1}, "second" → {2, 3}}]}]]
```

Modifying datasets

```
In[•]:= testDataset =
 Dataset[Association[{"a" → Association[{"first" → {1, 2}, "second" → ""}],
    "b" → Association[{"first" → {"x", "y"}, "second" → ""}],
    "c" → Association[{"first" → {1}, "second" → {2, 3}}]}]]
```

The Insert command can be used to add information. The third argument indicates the position where insertion is to take place (before the row with the key “b”).

```
In[•]:= Insert[testDataset,
 {"ab" → Association[{"first" → {u, v}, "second" → {9, 11}}]}, Key["b"]]
```

Note that this output does not replace the value assigned to the symbol *testDataset*.

```
In[•]:= testDataset
```

If we want that to happen we have to be explicit about reassigning the symbol.

```
In[•]:= testDataset = Insert[testDataset,
   {"ab" → Association[{"first" → {u, v}, "second" → {9, 11}}]}, Key["b"]]
```

Now the dataset has changed.

```
In[•]:= testDataset
```

The Delete command can be used to remove a row.

```
In[•]:= testDataset = Delete[testDataset, 3]
```

The Append command puts a new row at the end of the dataset.

```
testDataset =
 Append[testDataset, {"d" → Association[{"first" → "", "second" → {13}}]}]
```

Examples of the Take command, returning parts of the dataset. First two rows.

```
Take[testDataset, 2]
```

Last two rows.

```
Take[testDataset, -2]
```

Examples of the Drop command, returning parts of the dataset. Drop first row.

```
Drop[testDataset, 1]
```

Drop last two rows. What is the equivalent expression using Take?

```
Drop[testDataset, -2]
```

Mathematica has a number of pairs of commands that work in a complementary fashion like Take and Drop. Spend a moment studying the following list commands.

```
Range[6]
```

```
First[Range[6]]
```

```
Rest[Range[6]]
```

```
Most[Range[6]]
```

```
Last[Range[6]]
```

Creating datasets with EntityValue

We can use the EntityValue command to create datasets of properties. Here classes of file formats that *Mathematica* knows about.

```
EntityValue["FileFormat", "SampleEntityClasses"]
```

The properties we can request.

```
audio file formats FILE FORMATS ["Properties"]
```

A dataset of properties for a particular format.

```
MP3 Audio FILE FORMAT ["Dataset"]
```

Here is a dataset of audio file formats, showing name, developer, filename extension and applications.

```
audioFormatExtensions = EntityValue[audio file formats FILE FORMATS,
  {"Name", "Developer", "Extension", "Applications"}, "Dataset"]
```

Importing datasets from spreadsheets

If we have spreadsheet data, we can import it into a dataset. Here we import a small table from a

Microsoft Excel file, indicating that the first line contains the names of columns.

```
In[•]:= cityTable =
         Import["ExampleData/cities.xlsx", {"XLSX", "Dataset", 1}, "HeaderLines" → 1]
```

Selections and transformations

Selecting

When we wanted to pull elements from a list that matched a particular query, we used a command called Select.

Here are the numbers from 1 to 6.

```
In[•]:= Range[6]
```

Here are the odd numbers in that list.

```
In[•]:= Select[Range[6], OddQ]
```

In SPARQL, when we wanted to retrieve triples matching a particular pattern, we used a command like

*select * where {?fruit a <http://example.org/fruit>.}*

In both cases, the Select statement starts with a data structure and returns elements that match the query. We can also use Select to query datasets in *Mathematica*, and if we are accessing an external database that uses the SQL language, we will use a SQL command called Select, too.

In the previous section of this lesson we created a dataset of audio file formats, developers and file-name extensions. Which of those file formats were developed by Sony?

```
In[•]:= audioFormatExtensions[Select[#Developer == "Sony" &]]
```

If we just want one or more columns, we take them after selecting the rows.

```
In[•]:= audioFormatExtensions[Select[#Developer == "Sony" &], "Extension"]
```

The SelectFirst command retrieves the first record matching our criteria.

```
In[•]:= audioFormatExtensions[SelectFirst[#Developer == "Audible" &]]
```

Which file formats are supported by Apple iTunes?

```
In[•]:= audioFormatExtensions[
         Select[MemberQ[#Applications, "Apple iTunes"] &], "Extension"]
```

Taking

Here is our city data table.

```
In[•]:= cityTable
```

Cities with greatest and least populations.

```
In[•]:= cityTable[MaximalBy[#Population &]]
```

```
In[•]:= cityTable[MinimalBy[#Population &]]
```

Shortest and longest country names.

```
In[•]:= TakeLargestBy[cityTable, StringLength[#Country] &, 1]
```

```
In[•]:= TakeSmallestBy[cityTable, StringLength[#Country] &, 1]
```

Sorting

We can use the SortBy or ReverseSortBy commands to sort on one of the columns.

```
In[•]:= cityTable[SortBy[#City &]]
```

```
In[•]:= cityTable[ReverseSortBy[#Country &]]
```

Grouping

The GroupBy command lets us partition a dataset based on the values in a column. Look carefully at the example below. The leftmost columns show the group. The developer 4-MP3 was responsible for one format. There are over 200 formats with no developer listed; these have been grouped together. Sony has developed over 20 formats, which we retrieved early with a Select statement.

```
In[•]:= audioFormatExtensions[GroupBy["Developer"]]
```

Note that we could assign this output table to a new symbol if we wished to do further processing with it.

Computations and querying

Summary statistics

Here is the table of city data again.

```
In[•]:= cityTable
```

We can compute summary statistics for the population field by applying expressions to the column.

```
In[•]:= Total@cityTable[All, "Population"]
```

If you don't want large numbers to display in scientific notation, you can use the AccountingForm command.

```
In[•]:= Total@cityTable[All, "Population"] // AccountingForm
```

```
In[•]:= Min@cityTable[All, "Population"] // AccountingForm
```

```
In[•]:= Median@cityTable[All, "Population"] // AccountingForm
```

```
In[•]:= Mean@cityTable[All, "Population"] // AccountingForm
```

```
In[•]:= Max@cityTable[All, "Population"] // AccountingForm
```

Counting

How many audio formats was each developer responsible for?

```
In[•]:= ReverseSort@audioFormatExtensions[Counts, "Developer"]
```

Relations

The most powerful operations that we will consider for structured data are relational ones. Since a dataset is a hierarchy of lists and associations, expressions that process lists or associations can be used to manipulate datasets. Instead of working with datasets here, however, we will demonstrate relational operations with associations, to keep things as simple as possible.

Here are the lineups of some classic rock groups from the 1960s-80s. The keys for each association have been chosen for demonstration purposes, and don't reflect what each musician actually did or could do.

```
In[•]:= beatles = <|"singer" → "John Lennon", "bass" → "Paul McCartney",
    "lead" → "George Harrison", "drums" → "Ringo Starr"|>;

In[•]:= who = <|"singer" → "Roger Daltrey", "bass" → "John Entwistle",
    "lead" → "Pete Townshend", "drums" → "Keith Moon"|>;

In[•]:= ledZeppelin = <|"singer" → "Robert Plant",
    "bass" → "John Paul Jones", "lead" → "Jimmy Page", "drums" → "John Bonham"|>;

In[•]:= electricLightOrchestra = <|
    "multi-instrumentalist" → {"Jeff Lynne", "Roy Wood"}, "drummer" → "Bev Bevan"|>;

In[•]:= travelingWilburys = <|"lead" → "George Harrison", "multi-instrumentalist" →
     {"Jeff Lynne", "Tom Petty"}, "singer" → {"Bob Dylan", "Roy Orbison"}|>;
```

Catenate

The Catenate command creates a list of all of the values in order that appear in a list of associations. In this case it gives us a list of artists. Note that George Harrison and Jeff Lynne appear twice since they were members of multiple groups.

```
In[•]:= Flatten@Catenate[{beatles, electricLightOrchestra, travelingWilburys}]
```

Merge

The Merge command combines values with identical keys. In this case, it creates a new association where the musicians of each type (e.g., singer) are grouped together.

```
In[•]:= Merge[{beatles, who, ledZeppelin}, Identity]
```

Primary and foreign keys

Earlier, I mentioned that data is usually normalized to prevent repetition. To demonstrate, suppose we are running an online shop and want to keep track of our customers and the orders that they place. We keep details about each customer (things like email, phone and address) in a table of customers. In order to make sure that we have unique keys, each customer gets a unique id. This is called the primary key. We happen to have two customers named G Harrison, so this helps us to keep them apart.

```
In[•]:= customersTable = {
    <|"CustomerID" → "1", "Name" → "G Harrison", "City" → "New York",
     "StateProvince" → "NY", "Country" → "USA", "Email" → "gharrison@example.org"|>,
    <|"CustomerID" → "2", "Name" → "J Lynne", "City" → "Edmonton", "StateProvince" →
      "AB", "Country" → "Canada", "Email" → "mrbluesky@example.org"|>,
    <|"CustomerID" → "3", "Name" → "G Harrison", "City" → "London",
     "StateProvince" → "", "Country" → "UK", "Email" → "quietbeatle@example.org"|>,
    <|"CustomerID" → "4", "Name" → "J Page", "City" → "Lilydale",
     "StateProvince" → "NY", "Country" → "USA", "Email" → "zoso@example.org"|>,
    <|"CustomerID" → "5", "Name" → "R Daltrey", "City" → "New York",
     "StateProvince" → "NY", "Country" → "USA",
     "Email" → "babaoriley@example.org"|>};
```

We can wrap this in a Dataset command.

```
In[•]:= Dataset[customersTable]
```

Our customers place orders and we keep the details for each in a separate table. Each order gets a unique id.

```
In[•]:= ordersTable = {
    <|"OrderID" → "1", "CustomerID" → "3", "Date" → "01/01/2019", "Amount" → "$250"|>,
    <|"OrderID" → "2", "CustomerID" → "4",
     "Date" → "01/01/2019", "Amount" → "$100"|>,
    <|"OrderID" → "3", "CustomerID" → "4", "Date" → "01/03/2019",
     "Amount" → "$200"|>,
    <|"OrderID" → "4", "CustomerID" → "1", "Date" → "01/05/2019",
     "Amount" → "$250"|>,
    <|"OrderID" → "5", "CustomerID" → "2", "Date" → "01/05/2019",
     "Amount" → "$650"|>,
    <|"OrderID" → "6", "CustomerID" → "4", "Date" → "01/06/2019", "Amount" → "$75"|>,
    <|"OrderID" → "7", "CustomerID" → "3",
     "Date" → "01/10/2019", "Amount" → "$175"|>,
    <|"OrderID" → "8", "CustomerID" → "6", "Date" → "01/14/2019", "Amount" → "$575"|>
   };
```

We can also wrap this in a Dataset command.

```
In[•]:= Dataset[ordersTable]
```

If we had kept detail about each customer in the order table, we would have three copies of J Page's address, email etc. We haven't lost that information, however, because we can still look up the informa-

tion in our customer table. The CustomerID in the orders table is known as the foreign key. It links the orders table to the customers table. Who is customer 4?

```
In[•]:= Dataset[customersTable][Select[#CustomerID == "4" &]]
```

Similarly, we can retrieve information about each customer's orders using the customer ID. What orders did customer 4 place?

```
In[•]:= Dataset[ordersTable][Select[#CustomerID == "4" &]]
```

JoinAcross

Now we come to the JoinAcross command, which allows us to specify a number of different relations based on values. This next expression uses the CustomerID field to join the two tables.

```
In[•]:= Dataset@JoinAcross[ordersTable, customersTable, Key["CustomerID"]]
```

Notice that we have one customer (R Daltrey) who hasn't placed an order yet, and s/he does not appear in the joined table. We also have one order (order 8) which doesn't match any customers in the customer table. Order number 8 doesn't appear in the joined table, either.

This behavior, which is the default, is technically known as an "Inner" join. Only matched entries from both tables are included.

If we want to include unmatched entries from the table on the right, we use a "Right" join.

```
In[•]:= Dataset@JoinAcross[ordersTable, customersTable, Key["CustomerID"], "Right"]
```

If we want to include unmatched entries from the table on the left, we use a "Left" join.

```
In[•]:= Dataset@JoinAcross[ordersTable, customersTable, Key["CustomerID"], "Left"]
```

If we want to include all unmatched entries from both tables we use an "Outer" join.

```
In[•]:= Dataset@JoinAcross[ordersTable, customersTable, Key["CustomerID"], "Outer"]
```

Learning more

Video: Breaking the Boundaries of Data Science (beginner)

Introductory video on 'multiparadigm' data science, using computation with data in ways that go beyond traditional statistics.

https://www.wolfram.com/wolfram-u/catalog/dat001/

Video: Using DatabaseLink with the Wolfram Language

This video shows how to use the DatabaseLink toolkit to work with an external SQL database.

https://www.wolfram.com/wolfram-u/using-databaselink/

Guide: Working with Information in Relational Databases

Data in external SQL databases is mapped into entity stores in *Mathematica*.

https://reference.wolfram.com/language/guide/WorkingWithRelationalDatabases.html

ch21

Lesson 21. Measuring Images

The uses of geometry

Two dimensional images, including photographs and other kinds of sensor scans, contain reliable information about physical objects and environments. Extracting three-dimensional measurements from two-dimensional images is known as photogrammetry. Here we consider some examples of measuring images, including a typical photogrammetric task of reconstructing the positions from which photographs of a city were taken. We also look at the related process of georectification, which is the geometric manipulation of map images so they can be aligned or superimposed. We then turn to handwriting, which presents problems of measuring shapes, and finally to the reconstruction of a 3D model of a human face from a photograph and its comparison with a 3D model digitized from a 3D object.

Photogrammetry

Photogrammetry refers to the technique of measuring objects represented in photographs for use in surveying, mapping and other tasks. Here we look at aligning photographs for re-photography (photographs taken from same place at different times) and panoramas (a series of photographs taken from the same spot around the same time).

Clearing all definitions and defining *viewData*

In[•]:=
```
Clear["Global`*"]
```

In[•]:=
```
viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Historical re-photography

This pair of photographs by Wolfgang Kumm/dpa shows the Berlin Wall near the Brandenburg Gate in 1989 and the same location in 2014.

In[•]:=
```
wallPhotos = Import[
  "https://web.archive.org/web/20180523201537/https://www.thelocal.de/userdata/
    galleries/4148/berlin1.jpg"]
```

We can use the ImageTake command to separate the two.

```
In[•]:= ImageDimensions[wallPhotos]

In[•]:= wallBeforePhoto = ImageTake[wallPhotos, 351]

In[•]:= wallAfterPhoto = ImageTake[wallPhotos, {352, 702}]
```

We make sure both images are the same size

```
In[•]:= ImageDimensions[wallBeforePhoto]

In[•]:= ImageDimensions[wallAfterPhoto]
```

Alignment for re-photography, Part 1: Keypoints

Here is a function for creating standardized versions of images with a maximum pixel width or height of 400 pixels

```
In[•]:= standardizeImage[im_] :=
         ImageResize[ColorConvert[im, "Grayscale"], {400}]

In[•]:= wallStandardizedImages = {standardizeImage[wallBeforePhoto],
            ImageAdjust@standardizeImage[wallAfterPhoto]};
```

The ImageCorrespondingPoints command uses computer vision techniques to find matching *keypoints* in the two images.

```
In[•]:= wallMatches = (ImageCorrespondingPoints[##] &) @@ wallStandardizedImages;

In[•]:= Length[wallMatches[[1]]]
```

Here are the first two keypoints.

```
In[•]:= MapThread[Show[#1, Graphics[{Black, Background → Lighter@Orange,
              FontSize → 16, MapIndexed[Inset[#2[[1]], #1] &, #2]}]] &,
         {wallStandardizedImages, Join[{wallMatches[[1, 1 ;; 2]], wallMatches[[2, 1 ;; 2]]}]}]
```

Alignment for re-photography, Part 2: Dynamic image explorer

Next we transform the images into the same space, overlay them and create a dynamic image explorer with the

```
In[•]:= wall3 = ImageAlign[wallBeforePhoto, wallAfterPhoto,
            TransformationClass → "Rigid", Method → Automatic];

In[•]:= Manipulate[ImageCompose[wallBeforePhoto, {wall3, t}], {t, 0, 1}]
```

ImageAlign for panoramas

The ImageAlign command can also be used to create panoramas from images taken at the same time. Here is an example adapted from the documentation.

In[•]:= cityImage1 =  ;

In[•]:= cityImage2 =  ;

In[•]:= ImageCompose[ImageAlign[ImagePad[cityImage1, 50], cityImage2], {cityImage1, .5}]

An alternate approach is to find the geometric transformation of the second image to align it with the first one.

In[•]:= {e, tr} = FindGeometricTransform[cityImage1, cityImage2]

Once we have the transformation, we can use it to transform the second image then compose it with the first.

In[•]:= {w, h} = ImageDimensions[cityImage2];

In[•]:= tmp = ImagePerspectiveTransformation[cityImage2, tr,
DataRange → Full, PlotRange → {{0, First@tr[{w, 0}]}, {0, h}}];

In[•]:= ImageCompose[tmp, {cityImage1, .7}, Round@({w, h} / 2)]

Georectification

Georectification is the process of warping one map or image so that it can be represented in the same space as another.

Dynamic comparison of two images

We start with an example from the documentation, which shows how to create an interface that allows side-by-side comparison of two satellite images.

http://www.wolfram.com/language/11/image-and-signal-processing/dynamic-comparison-of-two-images.html?product=mathematica

The images are of Venice. “The left display shows a radar image taken by the ESA satellite Sentinel-1 on May 10, 2016; the right is an optical photograph taken by Sentinel-2 on April 29, 2016.”

In[•]:= {venice1, venice2} = {Import[
"https://earth.esa.int/documents/257246/2546936/Venice-Sentinel-1-10052016.jpg"], Import[
"https://earth.esa.int/documents/257246/2546936/Venice-Sentinel-2-29042016.jpg"]};

The DynamicModule command creates an object that maintains the x and y position and the scale of our two photographs. The DynamicImage command supports panning (click and drag on either image), zooming (SHIFT+click and drag up or down), etc. The scrollbars can also be dragged to change cropping.

```
In[•]:= DynamicModule[{x, y, s},
 Row[{DynamicImage[venice1, ZoomCenter → Dynamic[Scaled[{x, y}]],
    ZoomFactor → Dynamic[s], AppearanceElements →
     {"Pan", "Scrollbars", "ZoomButtons", "Zoom"}, ImageSize → {240, 360}],
   DynamicImage[venice2, ZoomCenter → Dynamic[Scaled[{x, y}]],
    ZoomFactor → Dynamic[s], AppearanceElements →
     {"Pan", "Scrollbars", "ZoomButtons", "Zoom"}, ImageSize → {240, 360}]}]]
```

Georectification, Part 1: Historical and contemporary maps

There is a 1967 Soviet atlas in the David Rumsey Map Collection with a map of Berlin showing the location of the Berlin Wall: Chief Administration of Geodesy and Cartography under the Council of Ministers of the USSR. *The World Atlas*. Second Edition. Moscow. 1967.

We will be using a clipped portion that I archived

```
In[•]:= wallMap1967 = Import[
  "https://github.com/williamjturkel/Digital-Research-Methods/raw/master/soviet-
    atlas-berlin-wall-1967b.jpg"]
```

We can retrieve a contemporary map with the GeoGraphics command.

```
In[•]:= wallMapNow = Rasterize[GeoGraphics[GeoDestination[ Brandenburg Gate BUILDING ,
    GeoDisplacement[{2600 m, 90°}]], GeoRange → 6000 m, ImageSize → Full]]
```

Georectification, Part 2: Resizing

We want the two maps to share at least one dimension.

```
In[•]:= ImageDimensions[wallMap1967]
```

```
In[•]:= ImageDimensions[wallMapNow]
```

```
In[•]:= wallMapNowResized = ImageResize[wallMapNow, 890];
```

```
In[•]:= ImageDimensions[wallMapNowResized]
```

Georectification, Part 3: Keypoints

When we aligned images in the previous section, we used computer vision to select matching key-points. Here we use an interactive interface to select the keypoints by hand. These are identical locations that are visible in both maps. As you move the Locators, their pixel coordinates change.

```
In[•]:= DynamicModule[{pt = {{200, 200}, {220, 200}, {240, 200}}},
 {LocatorPane[Dynamic[pt], Show[wallMap1967], {{0, 0}, {890, 690}}], Dynamic[pt]}]
```

Here are the three points I chose.

```
In[•]:= HighlightImage[wallMap1967, {{221.4`, 321.`}, {662.`, 569.`}, {695.`, 237.`}}]
```

Now we find the corresponding points on the present-day map

In[∘]:=
```
DynamicModule[{pt = {{113.`, 430.5`}, {684.`, 760.`}, {730.`, 314.`}}}, {LocatorPane[
    Dynamic[pt], Show[wallMapNowResized], {{0, 0}, {890, 892}}], Dynamic[pt]}]
```

Georectification, Part 4: Warping

Now we warp the historical map as if it were on a rubber sheet. We do this using the FindGeometric-Transform command.

In[∘]:=
```
FindGeometricTransform[{{113.`, 430.5`}, {684.`, 760.`}, {730.`, 314.`}},
  {{221.4`, 321.`}, {662.`, 569.`}, {695.`, 237.`}}]
```

This is what it looks like after the transformation.

In[∘]:=
```
Clear[warpedMap]
warpedMap = ImagePerspectiveTransformation[wallMap1967, TransformationFunction[

  ( 1.3011513421967178`    -0.009222908757556828` | -172.11435345117795` )
  ( -0.00785947720878244`   1.3425922808798445`   |  1.2679660916026876` )],
  ( ------------------------------------------------------------------- )
  (          0                        0           |         1            )

  PlotRange → All, DataRange → Full, Masking → All, Background → None]
```

Georectification, Part 5: Image composition

Finally we create an interactive interface to fade between the historical and contemporary maps.

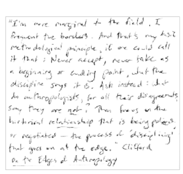
In[∘]:=
```
Manipulate[
  ImageCompose[wallMapNowResized, {warpedMap, t}, {0, 0}, {180, 0}], {{t, 0.7}, 0, 1}]
```

Handwriting

Earlier we learned about using optical character recognition (OCR) for detecting printed or typed words in page images. The recognition of handwritten words from page images is less well-developed, although continual progress continues to be made. In the meantime, if you need to work with scans of handwriting, one technique is word spotting (Manmatha & Srimal 1999; Rath & Manmatha 2002a, 2002b). The basic idea is to use image processing to find parts of the image that may represent a word of interest.

Finding rows of handwriting

Here is a sample of (my) handwriting. It is a quote from James Clifford's book *On the Edges of Anthropology: Interviews* (Prickly Paradigm Press, 2003).

In[∘]:=
```
handwritingSample =
```


```
;
```

The first thing we want to do is identify regions of the image that correspond to rows of handwriting. Suppose we rotate the image 90 degrees counterclockwise. Now imagine drawing a vertical line from the bottom edge to the top somewhere in the image. If the line runs through one of the rows of handwriting, it will hit a lot of dark ink. If it runs through one of the spaces between the rows of handwriting, it will hit mostly blank space.

```
In[•]:= Show[ImageRotate[handwritingSample], ImageSize → Small]
```

We use the following function to do exactly that. It runs through each column in the image and sums the lightness of pixels in that column.

```
In[•]:= columnPixelSum[img_] :=
         Sum[First /@ ImageData[img][[i]], {i, Length[ImageData[img]]}]
```

```
In[•]:= handwritingSampleColumnPixels = columnPixelSum[ImageRotate[handwritingSample]];
```

Next we smooth the resulting curve and find the peaks. That looks like this.

```
In[•]:= ListLinePlot[{GaussianFilter[handwritingSampleColumnPixels, 10],
           FindPeaks[GaussianFilter[handwritingSampleColumnPixels, 10]]},
         Joined → {True, False}, PlotStyle → {Automatic, PointSize[.02]}]
```

Each one of the peaks corresponds to the center of the blank space between rows of handwriting. Let's bundle our work so far into another function. This one takes an image of handwriting and returns a list of coordinate pairs. Each pair corresponds to the top and bottom of one row of handwriting in the image.

```
In[•]:= handwritingImageToRows[img_] :=
         Partition[
          FindPeaks[GaussianFilter[columnPixelSum[ImageRotate[img]], 10]][[All, 1]], 2, 1]
```

```
In[•]:= handwritingImageToRows[handwritingSample]
```

Here is how we extract a row of handwriting.

```
In[•]:= firstRow = ImageTake[handwritingSample, {1, 38}]
```

The second row.

```
In[•]:= ImageTake[handwritingSample, {38, 74}]
```

Finding word bounding boxes

Given a row of handwriting, the next step is to find the bounding boxes of all of the words in the row. We use the Erosion command to 'smear' the ink, so the individual characters run together.

```
In[•]:= Erosion[firstRow, DiskMatrix[4]]
```

We extract the morphological components to get the bounding boxes.

```
In[•]:= firstRowComponents = MorphologicalComponents[
            ColorNegate@Binarize@Erosion[firstRow, DiskMatrix[5]], Method → "BoundingBox"];
```

We can visualize this step with the Colorize command.

```
In[•]:= Colorize[firstRowComponents]
```

We can extract the actual bounding boxes with the next expression.

```
In[•]:= firstRowMeasurements = ComponentMeasurements[firstRowComponents, "BoundingBox"]
```

Visualize the results.

```
In[•]:= HighlightImage[firstRow, Rectangle @@@ Last /@ firstRowMeasurements]
```

A function for extracting words from page images

We can wrap these computations up into a single function. Given an image of a page of handwriting, the next function returns a nested list of word images (but not in their original order).

```
In[•]:= handwritingImageToWordImageList[img_] :=
 Module[{rowlist},
  rowlist = Table[ImageTake[img, row], {row, handwritingImageToRows[img]}];
  Return[Table[
    Table[ImageTake[r, a[[All, 2]], a[[All, 1]]], {a, Association[ComponentMeasurements[
        MorphologicalComponents[ColorNegate@Binarize@Erosion[r, DiskMatrix[5]],
         Method → "BoundingBox"], "BoundingBox"]]}], {r, rowlist}]]]
```

```
In[•]:= handwritingWords = handwritingImageToWordImageList[handwritingSample];
```

Here is what the output list looks like. I mapped the Framed command across the lists of word images to make it easier to see what was extracted.

```
In[•]:= Map[Framed, handwritingWords, {2}]
```

Comparing word shapes

Now that we have word images, we want some way of comparing their shapes. To do that, we need to extract features from the images that can be compared to one another. One way to do this is to use the same algorithm we used to find handwriting rows. We pass a vertical line through each column of the image and measure the pixel brightness.

Here are examples of the word ‘that’.

```
In[•]:= thatList = {handwritingWords[[2, 8]], handwritingWords[[4, 7]],
   handwritingWords[[9, 2]], handwritingWords[[11, 6]]}
```

When we compute the brightness of each column of pixels, we get a graph. Ideally, graphs for the same word are similar to one another.

```
In[•]:= ListLinePlot[columnPixelSum@#] & /@ thatList
```

Let’s compare the curves of the first and second instance of the word ‘that’. If we plot them on top of one another, we see that the shapes are similar although warped with respect to one another. As we write the same word again and again, some characters are stretched or compressed a bit.

```
In[•]:= thatOneCurve = columnPixelSum[thatList[[1]]];
```

In[•]:=
```
thatTwoCurve = columnPixelSum[thatList[[2]];
```

In[•]:=
```
ListLinePlot[{thatOneCurve, thatTwoCurve}]
```

The WarpingDistance command measures the similarity or difference of the two curves in terms of this stretching or compression. The lower the number, the closer they are to one another. These two are pretty close.

In[•]:=
```
WarpingDistance[thatOneCurve, thatTwoCurve]
```

The WarpingCorrespondence command finds the best correspondence between the two curves. We can see with a bit of adjustment, the two curves correspond nicely.

In[•]:=
```
{n1, n2} = WarpingCorrespondence[thatOneCurve, thatTwoCurve];
```

In[•]:=
```
ListLinePlot[{thatOneCurve[[n1]], thatTwoCurve[[n2]]}]
```

Here is a completely different word.

In[•]:=
```
discipline = handwritingWords[[6, 5]]
```

In[•]:=
```
disciplineCurve = columnPixelSum[discipline];
```

If we plot this curve with the first curve for 'that' we see they are very different.

In[•]:=
```
ListLinePlot[{thatOneCurve, disciplineCurve}]
```

The WarpingDistance between the two is much higher.

In[•]:=
```
WarpingDistance[thatOneCurve, disciplineCurve]
```

Attempts to find a correspondence between the two curves are unsatisfactory.

In[•]:=
```
{d1, d2} = WarpingCorrespondence[thatOneCurve, disciplineCurve];
```

In[•]:=
```
ListLinePlot[{thatOneCurve[[d1]], disciplineCurve[[d2]]}]
```

In the research papers that I cited, Manmatha and colleagues developed dozens of different word image features. Using these, they are able to automatically cluster related words and spot them in handwritten documents. We will learn more about clustering in a future lesson.

Facial 3D reconstruction

This example is adapted from one in the documentation.

http://www.wolfram.com/language/12/machine-learning-for-images/facial-3d-reconstruction.html?product=mathematica

It makes use of a pretrained neural net from the Wolfram Neural Net Repository. Given a 2D image of a face, the net reconstructs a 3D model.

https://resources.wolframcloud.com/NeuralNetRepository/resources/Unguided-Volumetric-Regression-Net-for-3D-Face-Reconstruction

We start by retrieving the pretrained net.

In[•]:=
```
net = NetModel["Unguided Volumetric Regression Net for 3D Face Reconstruction"]
```

Next we need a 2D photograph. We are going to use an 1863 portrait of Abraham Lincoln by Alexander Gardner from Wikipedia.

```
In[•]:= face = Import[
          "https://upload.wikimedia.org/wikipedia/commons/a/ab/Abraham_Lincoln_O-77_matte
            _collodion_print.jpg"];

In[•]:= Show[face, ImageSize → Small]
```

When we evaluate the net on the face image, it returns a 3D image. You can click and drag on this 3D model to rotate it around.

```
In[•]:= face3D = Image3D[net[face]]
```

Next we resize the model so that it has the same width and height as the 2D image.

```
In[•]:= reconstruction = Blur@ImageReflect[
          ImageResize[face3D, Append[ImageDimensions[face], 105]], Bottom → Front]
```

Starting at the tip of the nose and working our way into the head, we multiply each slice of the 3D model with the original image. This creates a series of slices that can be used to reconstruct a 3D image.

```
In[•]:= coloredSlices =
          Map[SetAlphaChannel[face, #] &, Image3DSlices[reconstruction, All, 2]];

In[•]:= Part[coloredSlices, {85, 75, 65, 55}]
```

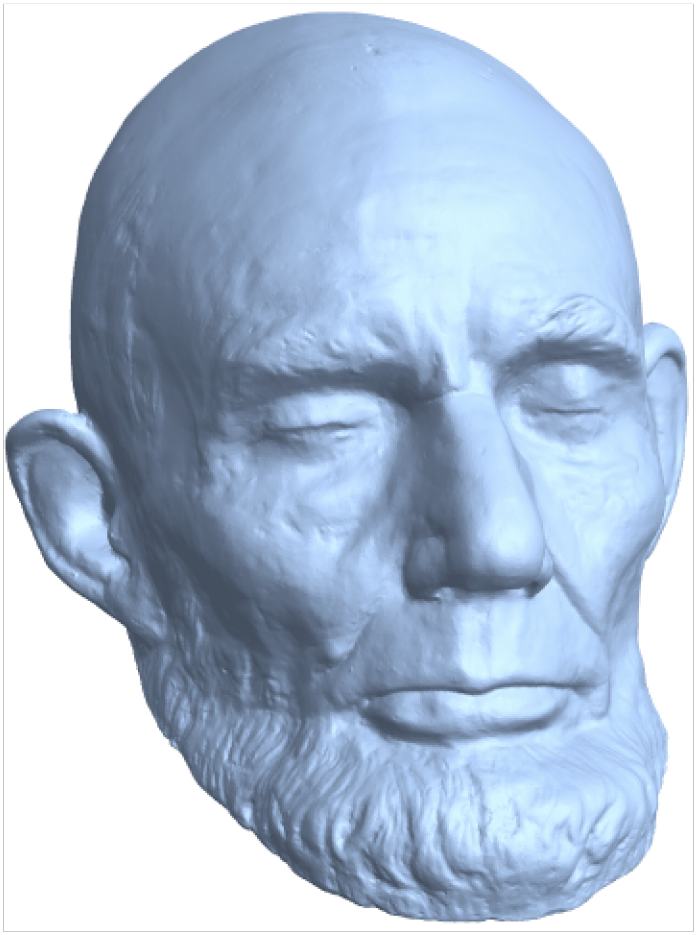
The final expression is the 3D facial reconstruction. Try clicking on the model and rotating it to see the three-dimensional relief.

```
In[•]:= ImageReflect[Image3D[coloredSlices, ViewPoint → {1, -2, 0},
          ViewAngle → 20 °, ImageSize → 300], Bottom → Front]
```

The reason I chose Lincoln for this example is that there is already 3D model that was digitized by the Smithsonian Institution from an 1865 life mask of Lincoln made by Clark Mills.

https://npg.si.edu/blog/lincoln-3-d

Here is a 2D screenshot picture of the model.



You can explore it in 3D in a web browser at the Smithsonian website.

https://3d.si.edu/model/fullscreen/p2b-1513178057393-1513869322319-0

You can also download the 3D model to explore in your notebook, and even print a copy on a 3D printer if you have access to one. Download the low-poly .STL file from the following page. It opens a dialogue where you accept terms and conditions and can optionally tell them about your use case. Once you have uncompressed the download, you can use the Import command to load it into your notebook.

https://3d.si.edu/explorer/abraham-lincoln-mills-life-mask #downloads

Learning more

Interactive image measurements

Another common task in photogrammetry is to trace lines and polygons in images to extract length, area, and other measurements. This can easily be done interactively in *Mathematica*.

http://www.wolfram.com/language/11/image-and-signal-processing/interactive-image-measurements.html?product=mathematica

Tutorial: Converting images into time series for data mining

A nice tutorial on using dynamic time warping to compare shapes of various kinds (e.g., skulls, leaves, handwriting).

https://izbicki.me/blog/converting-images-into-time-series-for-data-mining.html

References

Manmatha, R. and N. Srimal. "Scale space technique for word segmentation in handwritten manuscripts," *Proceedings of the Second International Conference on Scale-Space Theories Computer Vision* (Scale Space 99), pp. 22-33 (1999).
http://ciir.cs.umass.edu/pubfiles/mm-27.pdf

Rath, T and R. Manmatha. "Word Image Matching Using Dynamic Time Warping," *Proceedings of CVPR-03 conference*, vol. 2, pp. 521-527 (2002a).
http://ciir.cs.umass.edu/pubfiles/mm-38.pdf

Rath, T and R. Manmatha. "Features for Word Spotting in Historical Manuscripts," *Proceedings of ICDAR '03 conference*, vol.1, pp. 218-222 (2002b).
http://ciir.cs.umass.edu/pubfiles/mm-39.pdf

ch22

Lesson 22. Machine Learning

Supervised and unsupervised learning

Machine learning systems are divided into two broad categories, supervised and unsupervised. A supervised learner is given a set of labeled training data. If it is supposed to learn to tell the difference between French text and English text, for example, each of the texts that it receives during training has a label saying whether it is French or English. Once a supervised learner is trained, it is given unlabeled data and asked to provide a label for it. We will look at two broad classes of supervised learning, classification and prediction. The French and English text system is an example of classification. Given an unlabeled instance, the learner should be able to correctly classify it. A prediction system, on the other hand, is trained with instances and outcomes. For example, suppose you are trying to predict which students will be successful in college. Your training data consists of a large group of students where you have both their pattern of grades in required high school courses and an indication of whether they completed their college degree. Your trained predictor will take high school grades for a given student and predict whether s/he will complete a college degree.

For supervised learners to do very well, they often need a very large amount of labeled training data, which can be hard to come by for specialized domains. In transfer learning, you start with a learner that has been trained to do one task (say general purpose image classification), then you provide it with a much smaller amount of labeled data to focus it on a specific problem (say distinguishing

images of pre-cancerous cells from health ones).

Unsupervised clustering

Clearing all definitions and defining *viewData*

```
In[•]:= Clear["Global`*"]
```

```
In[•]:= viewData[x_] :=
 Framed[Pane[x, {Automatic, 200}, Scrollbars → True]]
```

Dictionary of Canadian Biography

For our example of unsupervised clustering we will be using the text of ten biographies from the first volume of the online *Dictionary of Canadian Biography*.

http://www.biographi.ca/en/

```
In[•]:= bioURL = "http://www.biographi.ca/EN/ShowBioPrintable.asp?BioId=";
```

Each biography has an ID. Here are the IDs.

```
In[•]:= biographyIDs = {"34182", "34214", "34223",
    "34224", "34352", "34425", "34427", "34231", "34488", "34663"};
```

We import each biography as symbolic XML and parse it to create an association where the keys are names and the values are biographies.

```
In[•]:= biographyAssociation =
  Module[{title, name, webpage},
   Association@Table[
     title = Import[bioURL <> id, "Title"];
     name = StringCases[title, "Biography - " ~~ n__ ~~ " - Volume" ~~ __ → n][[1]];
     webpage = Import[bioURL <> id, "XMLObject"];
     name →
      StringJoin@Flatten[Cases[Cases[webpage,
           XMLElement["div", {"id" → "content"}, {XMLElement["section",
               {"id" → "first", "class" → "bio"}, {}], bio__}] → bio, Infinity],
         XMLElement["p", {"class" → "NormalParagraph", _}, para_] → para] //.
        XMLElement[_, _, {e___}] → e], {id, biographyIDs}]];
```

Here are the names of the men whose biographies we will be studying.

```
In[•]:= Keys@biographyAssociation // TableForm
```

Clustering by hand

If we skim through the biographies, we can get a good idea of how these men were related to one another and group them together. When I chose the biographies, I had a few groupings in mind.

John and Sebastian Cabot, father and son, Italian explorers of the 1400-1500s.

```
In[•]:= jcabot = WikipediaData["John Cabot", "SummaryPlaintext"]
```

```
In[•]:= scabot = WikipediaData["Sebastian Cabot (explorer)", "SummaryPlaintext"]
```

Martin Frobisher, English explorer, and George Best, crew member and chronicler of Frobisher voyages (1500s).

```
In[•]:= frobisher = WikipediaData["Martin Frobisher", "SummaryPlaintext"]
```

```
In[•]:= best = WikipediaData["George Best (chronicler)", "SummaryPlaintext"]
```

Jolliet and La Salle, French explorers of the Mississippi river (1600s).

```
In[•]:= jolliet = WikipediaData["Louis Jolliet", "SummaryPlaintext"]
```

```
In[•]:= lasalle =
 WikipediaData["René-Robert Cavelier, Sieur de La Salle", "SummaryPlaintext"]
```

Brébeuf, Jogues and Le Jeune, Jesuit missionaries in New France (1600s).

```
In[•]:= brebeuf = WikipediaData["Jean de Brébeuf", "SummaryPlaintext"]
```

```
In[•]:= jogues = WikipediaData["Isaac Jogues", "SummaryPlaintext"]
```

```
In[•]:= lejeune = WikipediaData["Paul Le Jeune", "SummaryPlaintext"]
```

Talon, the First Intendant of New France (1600s), who doesn't fit immediately into any of the groups but has similarities with a number of the other men.

```
In[•]:= talon = WikipediaData["Jean Talon", "SummaryPlaintext"]
```

We will store the Wikipedia summaries in an association for further processing.

```
In[•]:= wikipediaAssociation = <|"CABOT, J" → jcabot, "CABOT, S" → scabot,
    "FROBISHER" → frobisher, "BEST" → best, "JOLLIET" → jolliet, "LA SALLE" → lasalle,
    "BREBEUF" → brebeuf, "JOGUES" → jogues, "LE JEUNE" → lejeune, "TALON" → talon|>;
```

Cluster analysis

Mathematica has a number of tools for automating this process of unsupervised clustering. Here is the output of the FindClusters command on both the Wikipedia summaries

```
In[•]:= Framed /@ FindClusters[wikipediaAssociation]
```

And the *DCB* biographies

```
In[•]:= Framed /@ FindClusters[biographyAssociation]
```

In both cases, Talon and La Salle are deemed to be by themselves in separate clusters. The Wikipedia summaries for Brebeuf and Jogues are clustered together (they mention the fact that the men were martyred together). Frobisher's Wikipedia summary is clustered separately from the others, but his biography is not.

Hierarchies of clusters

A number of commands create hierarchies of unsupervised clusters and present them as trees. The following expressions use TFIDF on the *DCB* biographies to create a hierarchy of clusters. The figures suggest that the biography of George Best is least like the others, followed by that of Martin Frobisher. The Cabots (1400s-1500s) are clustered together and distinguished from a branch of the tree whose members were in New France (1600s).

```
In[•]:= Dendrogram[biographyAssociation, Left, FeatureExtractor → "TFIDF"]
```

```
In[•]:= ClusteringTree[biographyAssociation, FeatureExtractor → "TFIDF"]
```

Compression distance

As we have seen, there are many different ways to define the similarity or dissimilarity of texts, and using TFIDF features is just one possibility. Rudi Cilibrasi and Paul Vitányi (2005) showed that compression can be used as a universal method for clustering, one that doesn't require any domain knowledge. It works remarkably well on natural language texts, music files, genomes, and many other kinds of representation. Their system is fairly easy to understand at an intuitive level, as long as you know that compressors are good at dealing with exact replications. If we use C[x] to denote the length of the compressed version of x, then for a reasonable compressor, C[xx] = C[x]. That is to say that once you compress two copies of x concatenated together, you should end up with something that is as long as the compressed version of x. If x and y are completely different, however, then C[xy] = C[x] + C[y].

Most interesting cases, however, will lie in between identity and complete difference. In these cases, C[x] < C[xy] < C[x] + C[y]. Intuitively, whatever x and y have in common makes C[xy] smaller than simply adding C[x] and C[y]. Or, to put it more provocatively, what the compressor 'learns' by compressing x can be put to use when it compresses y, and vice versa.

Here is a function that implements the Normalized Compression Distance. It returns a value close to zero when compressing identical copies of something, and the distance increases as the strings being compared diverge from one another.

```
In[•]:= normalizedCompressionDistance[x_String, y_String, compression_: "GZIP"] :=
 Module[{cx, cy, cxy},
  cx = StringLength[ExportString[x, {compression, "Text"}]];
  cy = StringLength[ExportString[y, {compression, "Text"}]];
  cxy = StringLength[ExportString[x <> y, {compression, "Text"}]];
  Return[N[(cxy - Min[cx, cy]) / Max[cx, cy]]]]
```

Let's try it out.

```
In[•]:= normalizedCompressionDistance["xxxxxxxxxx", "xxxxxxxxxx"]
```

```
In[•]:= normalizedCompressionDistance["xxxxxxxxxx", "xxxxxyyyyy"]
```

```
In[•]:= normalizedCompressionDistance["xxxxxxxxxx", "yyyyyyyyyy"]
```

Clustering with compression

Here are the ClusteringTree and Dendrogram expressions on the Wikipedia summaries using the normalized compression distance. Talon is now considered to be the outlier, followed by Frobisher. Brebeuf and Jogues are clustered together, as are the Cabots.

```
In[•]:= ClusteringTree[wikipediaAssociation,
         DistanceFunction → normalizedCompressionDistance]
```

```
In[•]:= Dendrogram[biographyAssociation, Left,
         DistanceFunction → normalizedCompressionDistance]
```

Here is the normalized compression distance clustering of the *DCB* biographies. Note that each of these three figures is displaying exactly the same data; only the visualization is changing. This is something to be aware of when working with exploratory data analysis.

```
In[•]:= ClusteringTree[biographyAssociation,
         DistanceFunction → normalizedCompressionDistance]
```

```
In[•]:= ClusteringTree[biographyAssociation, GraphLayout → "SpringEmbedding",
         DistanceFunction → normalizedCompressionDistance, ImageSize → Scaled[.6]]
```

Classify

In this example we are going to retrieve a number of Creative Commons licensed thumbnail images from Flickr and use them to train a classifier to distinguish pictures of leaves from pictures of grass. I have commented out (and iconized) the code that I used to retrieve the images and left copies of the thumbnails in the notebook since the Flickr page formatting is likely to change without warning.

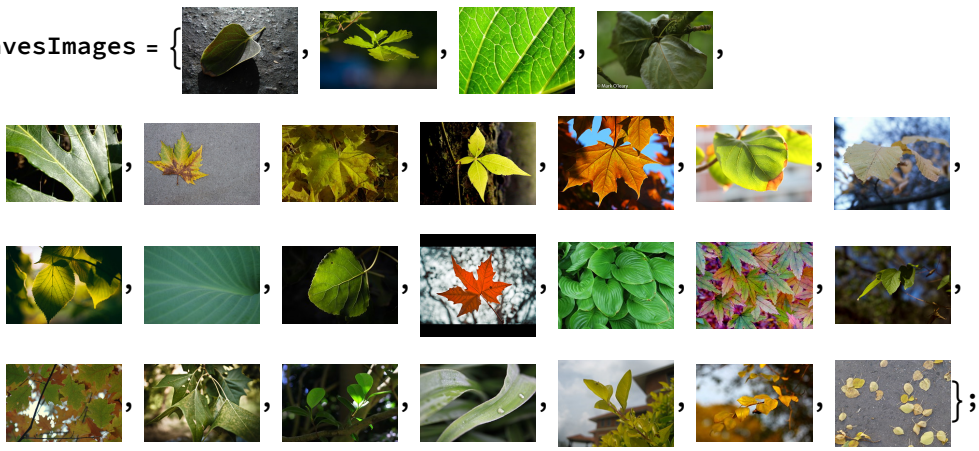
```
(* CompoundExpression[...] *)
```

Photo credits (name and title).

```
(* {...} *)
```

Here are our two data sets.

```
leavesImages = {
```



```
};
```

In[•]:=
```
grassImages = {
```

```
};
```

We will reserve the last four images of each set for testing.

In[•]:=
```
testImages = Join[leavesImages[[-4 ;; -1]], grassImages[[-4 ;; -1]]]
```

Now we create a training set by providing labels for the remaining images.

In[•]:=
```
trainingSet = Union[(# → "Leaves" &) /@ leavesImages[[1 ;; 21]],
    (# → "Grass" &) /@ grassImages[[1 ;; 21]]];
```

Each labeled instance looks like this.

In[•]:=
```
First@trainingSet
```

We train the classifier with a call to the Classify command.

In[•]:=
```
leavesOrGrass = Classify[trainingSet]
```

Now we test it on the test set. Remember, these are images that it has never seen before.

In[•]:=
```
leavesOrGrass[
```

```
]
```

Predict

Mathematica's example data contains a dataset of passengers on the *Titanic*. We will use these to demonstrate prediction based on machine learning. Each record gives the person's age, sex, the class of their berth on the ship, and indicates whether or not they survived.

In[•]:=
```
titanic = ExampleData[{"Dataset", "Titanic"}]
```

Number of rows.

In[•]:=
```
Length@titanic
```

Count the number of passengers with missing age.

In[•]:=
```
titanic[Count[_Missing], "age"]
```

Number of passengers travelling in each class.

In[•]:=
```
titanic[Counts, "class"]
```

Histogram of passenger ages, grouped by class.

```
In[•]:= titanic[GroupBy["class"], Histogram[#, {0, 80, 4}] &, "age"]
```

Retrieve 19 year old female passengers traveling in 2nd class.

```
In[•]:= titanic[Select[#class == "2nd" && #age == 19 && #sex == "female" &]]
```

Retrieve 19 year old male passengers traveling in 2nd class.

```
In[•]:= titanic[Select[#class == "2nd" && #age == 19 && #sex == "male" &]]
```

There were no 52 year old male passengers in 3rd class.

```
In[•]:= titanic[Select[#class == "3rd" && #age == 52 && #sex == "male" &]]
```

We can use the *Titanic* data to train a predictor. Given a particular age, sex and class, what were the chances of survival? First we need to create a list of rules, where an association of the first three fields is the key and the value is a 1. if the person survived and a 0. otherwise. We do that with the following expression (shown on the first three records of the dataset).

```
In[•]:= #[[1 ;; 3]] → 1. * Boole[#[[4]]] & /@ titanic[[1 ;; 3]] // Normal
```

Next we train the predictor. Here we are using the NearestNeighbors method, which bases a prediction on a small number of similar examples.

```
In[•]:= titanicSurvivalPredictor = Predict[
  #[[1 ;; 3]] → 1. * Boole[#[[4]]] & /@ titanic // Normal, Method → "NearestNeighbors"]
```

Now we can try our predictor. We have already seen that all of the 19-year-old females in 2nd class survived, and our predictor gives them a pretty good chance of survival.

```
In[•]:= titanicSurvivalPredictor[<|"class" → "2nd", "age" → 19, "sex" → "female"|>]
```

Only one 19-year-old male in 2nd class survived, and the predictor rates the chances of a similar passenger to be pretty dire.

```
In[•]:= titanicSurvivalPredictor[<|"class" → "2nd", "age" → 19, "sex" → "male"|>]
```

There were no 52-year-old males in 2nd class, but by inference from similar passengers, he would have had about a 25% chance of survival.

```
In[•]:= titanicSurvivalPredictor[<|"class" → "2nd", "age" → 52, "sex" → "male"|>]
```

Even that might be a bit optimistic!

```
In[•]:= titanic[Select[
  #class == "2nd" && (#age == 49 || #age == 50 || #age == 51 || #age == 54) && #sex == "male" &]]
```

Transfer learning

This example comes from the documentation. The Wolfram Neural Net Repository contains a large number of pre-trained neural nets for particular tasks.

http://resources.wolframcloud.com/NeuralNetRepository

These vary in size and speed of operation. The ResNet50 was trained to recognize 1000 classes of

objects using more than a million labelled images.

https://resources.wolframcloud.com/NeuralNetRepository/resources/ResNet-50-Trained-on-ImageNet-Competition-Data

We can obtain the model with the following command.

In[•]:=
```
resNet50 = NetModel["ResNet-50 Trained on ImageNet Competition Data"]
```

If you press the plus sign in the output above, it will expand your view of the properties of this neural net. In particular, we can see that the net consists of a number of layers. By replacing some of these layers and retraining the network with a small amount of data, we can create a new network that is specialized to a particular task.

The task that we have chosen for the new network is to learn to distinguish pictures of cats from pictures of dogs. Here is our training dataset.

In[•]:=
```
trainSet = {
```

```
-> "cat",
```

```
-> "cat",
```

```
-> "cat",
```

```
-> "cat",
```

```
-> "cat",
```

```
-> "cat",
```

```
-> "cat",
```

```
-> "dog",
```

```
-> "dog",
```

```
-> "dog",
```

```
-> "dog",
```

```
-> "dog",
```

```
-> "dog",
```

```
-> "dog"};
```

We take the convolutional part of the net with the NetTake command. If you use the plus sign on the output and compare this net with the original one, you will see that we have taken the middle portion of it.

In[•]:=
```
subNet = NetTake[resNet50, {"conv1", "flatten_0"}]
```

Even before training with our new data this net is capable of separating dog images from cat images. We can see this by using FeatureSpacePlot.

In[•]:=
```
FeatureSpacePlot[trainSet[[All, 1]],
 FeatureExtractor → subNet, LabelingSize → 100, ImageSize → Scaled[.8]]
```

We want the system to be able to classify new images, however, so now we add a new classification layer before the output. Instead of choosing among 1000 categories of object like the original net did, we are now choosing among two.

```
In[•]:= joinedNet =
         NetJoin[subNet, NetChain@<|"linear10" → LinearLayer[2], "prob" → SoftmaxLayer[]|>,
          "Output" → NetDecoder[{"Class", {"cat", "dog"}}]]
```

Next we add a layer near the input to perform data augmentation. The output is our new network after we have completed “net surgery”.

```
In[•]:= finalNet = NetPrepend[joinedNet, {"augment" → ImageAugmentationLayer[{224, 224}]},
          "Input" → NetExtract[joinedNet, "Input"]]
```

We retrain the network on our new training set.

```
In[•]:= trainedNet =
         NetTrain[finalNet, trainSet, LearningRateMultipliers → {"linear10" → 1, _ → 0}]
```

Now we can test it with an image it hasn’t seen.

```
In[•]:= trainedNet[
```

```
]
```

Learning more

Predict the survival of Titanic passengers

Here is another take on predicting the survival of *Titanic* passengers, this time using the Classify command.

http://www.wolfram.com/mathematica/new-in-10/highly-automated-machine-learning/predict-the-survival-of-titanic-passengers.html

Tutorial: Partitioning Data into Clusters

https://reference.wolfram.com/language/tutorial/PartitioningDataIntoClusters.html

Wolfram U Course: From Zero to AI in 60 minutes

https://www.wolfram.com/wolfram-u/machine-learning-zero-to-AI-60-minutes/

Wolfram U Course: Machine Learning Basics

https://www.wolfram.com/wolfram-u/machine-learning-basics/

Train a custom image classifier

Another example of transfer learning.

http://www.wolfram.com/language/12/machine-learning-for-images/train-a-custom-image-classifier.html?product=mathematica

Tutorial: Neural Networks in the Wolfram Language

https://reference.wolfram.com/language/tutorial/NeuralNetworksOverview.html

References

Cilibrasi, Rudi and Paul Vitányi, "Clustering by Compression," *IEEE Transactions on Information Theory* 51, no. 4 (2005): 1523-45.
https://homepages.cwi.nl/~paulv/papers/cluster.pdf

project-ideas

Project Ideas

This section contains ideas and leads for coming up with projects that build on the material in the lessons. Often the first step is simply replication: given the inputs used by someone else, can you get your code to produce the same outputs? Once you have replicated a particular example or application, then you can explore trying it on other sources, extending it in new ways, etc. Much of the code for the lessons of this book originated in this way. I saw something interesting and thought, "I wonder if I can implement that in *Mathematica*. And if I could, what could I do with it?"

Another approach is to start with a dataset or an API and write code to make use of it. Can you retrieve a single record? Extract fields? Can you compute descriptive statistics (min, max, total) for a group of records or the whole collection? Does it make sense to manipulate the data for reuse? What kinds of visualizations can help you to make sense of the whole collection or parts of it?

A third strategy is the mashup. Building on two or more examples or APIs, create a new project that uses the output of one as the input of the next.

Contents

- Geospatial
- Health
- Heritage
- Images
- Linguistics
- Linked open data
- Management science
- Military history
- Music
- Names
- Networks
- Neural nets
- Page images
- Programming
- Social media
- Social network analysis
- Social sciences
- Stylometry
- Text mining
- Time series
- Video
- Web services
- Workflow

pr-art-and-design

Art and design

3D Scanning and Printing

http://www.wolfram.com/language/gallery/make-a-3d-image-from-a-point-cloud/

Linked Art

Under development. Useful source of JSON-LD?

https://linked.art/index.html

MOMA

Wolfram Data Repository for Museum of Modern Art holdings.

https://datarepository.wolframcloud.com/resources/38f47cc0-667d-4988-8ebe-6c1bef23fdc6

The following example includes a very nice timeline of image collages as the final output.

http://www.wolfram.com/language/11/richer-knowledgebase-access/museum-of-modern-art-paintings-and-artists.html?product=mathematica

MOMA on Github

“This research dataset contains 137,643 records, representing all of the works that have been accessioned into MoMA's collection and cataloged in our database. It includes basic metadata for each work, including title, artist, date made, medium, dimensions, and date acquired by the Museum. Some of these records have incomplete information and are noted as ‘not Curator Approved.’” CSV and JSON.

https://github.com/MuseumofModernArt/collection

pr-biometrics

Biometrics

Project ideas

- Can the heart rate estimation example be applied to historical video or color film? There are lots of videos of people shouting into cameras on YouTube, so probably easiest to start there, then try on something from the Prelinger Archive or the Internet Archive TV news.

Heart rate estimation from blood flow to face

http://www.wolfram.com/mathematica/new-in-10/enhanced-sound-and-signal-processing/heart-rate-estimation-from-video.html

pr-databases

Databases

Project ideas

- Try implementing the *Programming Historian* tutorial with *Mathematica* instead of R. Or try using RLink to call the R Language from within your notebook.

Programming Historian: Getting started with MySQL and R

Datasets include CSVs of Welsh newspaper articles published during WWI.

https://programminghistorian.org/en/lessons/getting-started-with-mysql-using-r

pr-datasets

Datasets

Apache mail

Email archives of the Apache project available as downloadable MBOX files.

https://lists.apache.org/list.html?user@forrest.apache.org:2004-12

Computational / Data Journalism

https://www.propublica.org/nerds/

Digging into Data

List of data repositories. Not maintained, but many still work.

https://diggingintodata.org/repositories

Global events of organized violence

Geocoded dataset from the Uppsala Conflict Data Program, with fine temporal resolution for events.

https://datarepository.wolframcloud.com/resources/4befcf8d-d341-4196-accf-99edc4ba278d

Government of Canada Open Data

In Apr 2019, 99 datasets categorized as History and Archaeology.

https://open.canada.ca/en/open-data

Journal of Open Humanities Data

Has an RSS feed which could be used for an RSS example.

https://openhumanitiesdata.metajnl.com

Manned space missions

http://www.wolfram.com/language/11/time-series-processing/manned-space-missions.html?product=-mathematica

Mass shootings in America

Stanford University Libraries dataset, 1966-2016.

https://datarepository.wolframcloud.com/resources/393bc956-89b6-4775-8368-b3bce9e9a98d

Open government data

https://github.com/public-apis/public-apis #government

Example census.gov (US Census) has many APIs. Here we query the annual survey of entrepreneurs

https://www.census.gov/data/developers/data-sets/ase.html

http://api.census.gov/data/2014/ase/cscbo?get=GEO_TTL,NAICS2012_TTL,ASECBO,ASECBO_TTL,ACQBUS,ACQBUS_TTL,YEAR,OWNPDEMP,OWNPDEMP_PCT,OWNPDEMP_S,OWNPDEMP_PCT_S&for=us:*

Fields

http://api.census.gov/data/2014/ase/cscbo/variables.html

Programming Historian: Cleaning data with OpenRefine

Data set is messy TSV of freely available metadata about 75K objects in The Powerhouse Museum in Sydney.

https://programminghistorian.org/en/lessons/cleaning-data-with-openrefine

Programming Historian: Fetching and parsing data from the web with OpenRefine

Uses Shakespeare's Sonnets from Gutenberg and JSON from Chronicling America as examples. Also runs text against a free Sentiment API.

https://programminghistorian.org/en/lessons/fetch-and-parse-data-with-openrefine

Programming Historian: Understanding regular expressions

Example is to translate early 20thC public domain US public health reports from JSTOR collection in Internet Archive (Statistical Report of Morbidity and Mortality) into tabular data.

https://programminghistorian.org/en/lessons/understanding-regular-expressions

Public data sets

A large list of public datasets on GitHub.

https://github.com/awesomedata/awesome-public-datasets

SSHRC Competition statistics

Canadian Social Sciences and Humanities Research Council. Downloadable as Excel files.

http://www.sshrc-crsh.gc.ca/results-resultats/stats-statistiques/index-eng.aspx

See this example for working with similar data in a resource object.

http://www.wolfram.com/language/11/richer-knowledgebase-access/national-science-foundation-grants.html?product=mathematica

US presidential inaugural addresses

Complete text of inaugural addresses from 1789-2017.

https://datarepository.wolframcloud.com/resources/a8996b67-54d2-4de2-81f1-100d797fd90c

Wolfram Data Repository

The Wolfram Data Repository "is a public resource that hosts an expanding collection of computable datasets, curated and structured to be suitable for immediate use in computation, visualization, analysis and more." Many of the examples in this text are based on datasets in the WDR.

http://www.wolfram.com/language/11/cloud-storage-and-operations/access-the-data-repository.html?product=mathematica

https://reference.wolfram.com/language/workflow/UseDataFromTheWolframDataRepository.html

Lots of other options, like Agriculture, Demographics, Economics, Geography, Government, Health, Images, Meteorology

https://datarepository.wolframcloud.com/category/Culture/
https://datarepository.wolframcloud.com/category/Social-Media/
https://datarepository.wolframcloud.com/category/Text-Literature/

1911 Encyclopedia Britannica

https://datarepository.wolframcloud.com/resources/aab4a762-c163-4970-a3ce-f4f4e0bb7efc

UFO sightings 2015

https://datarepository.wolframcloud.com/resources/3e9c3364-be97-4ca7-90dd-ac9790a1ea1f

New Orleans Slave Sales, 1856-61

https://datarepository.wolframcloud.com/resources/f372fe55-1809-4c2d-bf0c-3fbae99d41cf

Mr Rogers' Sweater Colors

https://datarepository.wolframcloud.com/resources/f3c24818-7f88-4b31-bf54-9dc638730583

Video: Becoming a Data Curator

https://www.wolfram.com/wolfram-u/becoming-a-data-curator/

Video: Curating Data and Integrating the WDF

https://www.wolfram.com/wolfram-u/curating-data-integrating-data-framework/

pr-dates

Dates

Calculate with time zones

http://www.wolfram.com/mathematica/new-in-10/symbolic-dates-and-times/calculate-with-time-zones.html

Date formats

http://www.wolfram.com/language/11/units-and-dates/extensive-date-formats.html?product=mathe-

matica

http://www.wolfram.com/language/11/new-visualization-domains/flexible-date-formats.html?product=mathematica

http://www.wolfram.com/mathematica/new-in-10/symbolic-dates-and-times/extract-values-from-dates-and-times.html

http://www.wolfram.com/mathematica/new-in-10/symbolic-dates-and-times/calculate-with-dates-and-times.html

http://www.wolfram.com/mathematica/new-in-10/symbolic-dates-and-times/create-symbolic-date-and-time-expressions.html

http://www.wolfram.com/mathematica/new-in-10/domain-specific-language-interpretation/interpret-dates-expressed-in-natural-language.html

Create a moon phase calendar

http://www.wolfram.com/mathematica/new-in-10/time-series/create-a-moon-phase-calendar.html

DateHistogram

Example analyzes arrival time of airplanes, showing busiest and least busy days of the week.

http://www.wolfram.com/language/11/units-and-dates/date-analysis-in-datehistogram.html?product=mathematica

Event series with dates

Example shows how to extract day of week information from series of birthdays.

http://www.wolfram.com/language/12/probability-and-statistics/moving-map-on-event-series-with-dates.html?product=mathematica

Filter data by date

Get data values by date. Example is historical population data.

http://www.wolfram.com/mathematica/new-in-10/symbolic-dates-and-times/use-date-qualifiers-in-data-queries.html

Create an interactive TabView display of maps of volcanic activity by decade.

http://www.wolfram.com/mathematica/new-in-10/symbolic-dates-and-times/filter-data-by-date.html

Filter dates by decade

Example is TimelinePlot of Rembrandt paintings completed in the 1660s.

http://www.wolfram.com/language/12/units-dates-and-uncertainty/filter-dates-by-decade.html?product=mathematica

Refine knowledgebase searches using dates

Example is selecting recent movies based on release date.

http://www.wolfram.com/language/12/units-dates-and-uncertainty/refine-knowledgebase-searches-using-dates.html

Create a timeline

http://www.wolfram.com/language/11/units-and-dates/create-a-timeline.html?product=mathematica

TimelinePlot

http://www.wolfram.com/language/11/units-and-dates/visualize-date-objects.html?product=mathematica

http://www.wolfram.com/language/11/new-visualization-domains/visualize-time-events-with-timeline-plot.html?product=mathematica

http://www.wolfram.com/language/11/new-visualization-domains/add-graphical-labels-in-timeline-plot.html?product=mathematica

http://www.wolfram.com/language/11/new-visualization-domains/timeline-layout-and-label-placements.html?product=mathematica

Timeline from entity class

Timeline of Star Wars movie releases, using movie posters to represent each.

http://www.wolfram.com/language/11/units-and-dates/create-a-timeline-from-an-entity-class.html

Sapping Attention: Turning point years in history

In this pair of blog posts, Ben Schmidt analyzes years mentioned in titles of History dissertations over a 120-year period.

http://sappingattention.blogspot.com/2013/05/turning-point-years-in-history.html

http://sappingattention.blogspot.com/2013/05/what-years-do-historians-write-about.html

pr-email

Email

Mail datasets

Analyze email threads and networks

http://www.wolfram.com/language/12/mail-and-messaging/obtain-statistics-about-an-mbox.html?product=mathematica

http://www.wolfram.com/language/12/mail-and-messaging/analyze-email-threads.html?product=-

mathematica

http://www.wolfram.com/language/12/mail-and-messaging/analyze-relations-between-email-senders.html?product=mathematica

Programming Historian: Sentiment analysis for exploratory data analysis

Uses Enron E-mail corpus (423 Mb). Good discussion about scope (how much text to feed the analyzer at once).

https://programminghistorian.org/en/lessons/sentiment-analysis

https://www.cs.cmu.edu/~./enron/

pr-entertainment

Entertainment

Find theatres around a location

Uses *Mathematica*'s symbolic SPARQL expressions to find theatres in the center of the city of Madrid then plots them with GeoGraphics.

http://www.wolfram.com/language/12/rdf-and-sparql/find-theaters-around-a-location.html?product=mathematica

pr-entities

Entities

Project ideas

- Try creating an entity store for some domain of your own research.
- Try recreating the two-hop network of entities from the example titled "Expanded Relationships among Domains".

Entity store

Create your own entity store so you can work with new data as if it were built-in.

http://www.wolfram.com/language/11/richer-knowledgebase-access/create-a-fireball-meteor-entitystore.html?product=mathematica

Expanded relationships among domains

Example of a two-hop network of entities surrounding the astronaut Neil Armstrong.

http://www.wolfram.com/language/11/knowledgebase-expansion/expanded-relationships-among-domains.html?product=mathematica

pr-environment

Environment

Historical data in past images

Finding ecological and climatological data in historical sources

https://www.nature.com/naturejobs/science/articles/10.1038/nj7672-419

pr-faces

Faces

Project ideas

- There have been a number of notorious examples of facial recognition software misgendering trans people. Try creating a non-binary gender classifier.

Age estimation

Train a neural net to estimate people's ages.

http://www.wolfram.com/language/12/machine-learning-for-images/train-an-age-estimation-network.html?product=mathematica

Face interpolation

Uses facial landmarks to align faces and morph between them.

http://www.wolfram.com/language/12/machine-learning-for-images/perform-face-interpolation.html?product=mathematica

Gender

Example of training a gender classifier.

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/gender-classification.html

pr-forensics

Forensics

Project ideas

- One problem that sometimes occurs is to determine where a picture was taken. Suppose you have a photograph of a well-known landmark like a building or statue that includes the date and time it was taken (either in the EXIF data or as a watermark). Can you use sun shadows to determine the position of the camera? See the "Plot the path of shadows cast by the sun" example.

Capture the Flag: Forensics

A useful reference for getting started, from the Trail of Bits team.

https://trailofbits.github.io/ctf/forensics/

Plot the paths of shadows cast by the sun

Uses location and height of Sydney Opera House, position of sun, and trigonometry to plot the path of shadows cast by the building.

http://www.wolfram.com/mathematica/new-in-10/geographic-visualization/plot-the-paths-of-shadows-cast-by-the-sun.html

PDF format

PDF tricks

https://github.com/corkami/docs/blob/master/PDF/PDF.md

NSA report on hidden data and metadata in PDFs

http://www.itsecure.hu/library/file/Biztonsági%20 útmutatók/Alkalmazások/Hidden%20 Data %20 and %20 Metadata %20 in %20 Adobe %20 PDF %20 Files.pdf

Python library for analyzing PDFs

https://github.com/jesparza/peepdf

References

Farid, Hany. *Photo Forensics*. Cambridge, MA: MIT Press, 2019.

pr-geospatial

Geospatial

Project ideas

- Replicate the georectification process shown in the lesson on “Measuring Images” with the sources from the “Compare historic and contemporary maps” example, then compare the results with the ones shown in the example.
- Replicate as much as you can of the *Programming Historian* tutorial on Google Maps and Google Earth using resources within *Mathematica*.

Birth and death places of notable people

Example shows where notable people born in NY City died.

http://www.wolfram.com/language/12/cultural-and-historical-entities/plot-birth-and-death-places-of-notable-people.html?product=mathematica

Compare historic and contemporary maps

This example compares the original 1792 L’Enfant Plan of Washington DC with a street map from

today. It doesn't show how to do the georectification, but does use it by solving an equation with GeoPosition landmarks.

http://www.wolfram.com/mathematica/new-in-10/geographic-visualization/compare-historic-and-contemporary-maps.html

Connectivity of US counties

How many counties would you have to travel through if you drive from San Francisco to Manhattan?

http://www.wolfram.com/language/11/geo-data/connectivity-of-us-counties.html?product=mathematica

Distortion of country polygons in Mercator projection

http://www.wolfram.com/language/12/new-in-geography/country-polygon-distortion-in-the-mercator-projection.html?product=mathematica

Enhance curated datasets with built-in data

Example is mapping of Chicago Head Start schools.

http://www.wolfram.com/language/11/knowledgebase-expansion/enhance-curated-datasets-with-built-in-data.html?product=mathematica

Find the most interior point in the US

Try reimplementing with newer region methods.

http://www.wolfram.com/mathematica/new-in-10/entity-based-geocomputation/find-the-most-interior-point-in-the-united-states.html

Find the nearest broadcast stations

http://www.wolfram.com/mathematica/new-in-10/entity-based-geocomputation/find-the-nearest-broadcast-stations.html

Find the seas nearest a given location

http://www.wolfram.com/mathematica/new-in-10/entity-based-geocomputation/find-the-seas-nearest-a-given-location.html

Historical volcanic activity

Example is Pacific Ring of Fire in last 25 years.

http://www.wolfram.com/language/12/geographic-entities/view-historical-volcanic-activity-in-a-region.html?product=mathematica

Make a map of flight costs

http://www.wolfram.com/mathematica/new-in-10/semantic-data-import/make-a-map-of-flight-costs.html

Map the locations of shipwrecks

Example is the Mediterranean. Shows how to use Graphics to create a custom map marker.

http://www.wolfram.com/mathematica/new-in-10/entity-based-geocomputation/map-the-locations-of-shipwrecks.html

Neural network model to estimate geo location from image

http://www.wolfram.com/language/12/machine-learning-for-images/a-model-to-estimate-geo-location-from-an-image.html?product=mathematica

https://resources.wolframcloud.com/NeuralNetRepository/resources/ResNet-101-Trained-on-YFC-C100m-Geotagged-Data

Opacity and Color for heat maps

Example is 50 busiest airports in Europe.

http://www.wolfram.com/language/11/enhanced-geo-visualization/create-histograms-from-weighted-locations.html?product=mathematica

Plot locations on a map

Location of schools in a town

http://www.wolfram.com/mathematica/new-in-10/enhanced-visualization/plot-locations-on-a-map.html

Programming Historian: Introduction to Google Maps and Google Earth

This tutorial covers making a custom map in Google Maps, creating vector layers, viewing in Google Earth, georectifying, plotting data, KML, and adding scanned historical maps. Data from Jim Clifford's and Josh MacFadyen's research. Create global map of major exporters of fat to Britain in the mid-1890s.

https://programminghistorian.org/en/lessons/googlemaps-googleearth

Programming Historian: Extracting Keywords from Text with Gazateers

Adam Crymble includes a CSV of British placenames he extracted from Alumni Oxonienses, 1500-1714, mini biographies of individuals who went to Oxford during the reign of James I (1603-25).

https://programminghistorian.org/en/lessons/extracting-keywords

Programming Historian: Geocoding Historical Data

No good free sources of data for this?

https://programminghistorian.org/en/lessons/geocoding-qgis

Programming Historian: Geoparsing Text with the Edinburgh Geoparser

Extracts locations from English language text and plots on map.

https://programminghistorian.org/en/lessons/geoparsing-text-with-edinburgh

Programming Historian: Introduction to gravity models of migration and trade

This tutorial by Adam Crymble uses three kinds of regression modelling (simple linear, multivariate linear and negative binomial). Published dataset in Zenodo.

https://programminghistorian.org/en/lessons/gravity-model

Open dataset

https://zenodo.org/record/1217600

Segmentation of aerial photograph

Could use this for lineaments in urban landscape (from, e.g., former rail line).

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/segment-an-aerial-image.html

Size of parks and protected lands in US

Uses FilteredEntityClass, GeoListPlot and GeoBubbleChart.

http://www.wolfram.com/language/12/geographic-entities/compare-sizes-of-parks-and-protected-lands.html?product=mathematica

Visualization with tilted perspective projection

As if seen from a great height.

http://www.wolfram.com/language/11/enhanced-geo-visualization/visualization-with-a-tilted-perspective-projection.html?product=mathematica

pr-health

Health

Project ideas

- Visualize and map your own fitness activities.

Fitbit

Requires account.

http://www.wolfram.com/mathematica/new-in-10/curated-api-framework/visualize-fitness-data.html

RunKeeper

Requires account.

http://www.wolfram.com/mathematica/new-in-10/curated-api-framework/map-your-fitness-activities.html

pr-heritage

Heritage

Historical entity connections

Historical sites and events associated with Anne Frank.

http://www.wolfram.com/language/12/cultural-and-historical-entities/explore-connections-between-historical-entities.html?product=mathematica

pr-images

Images

Project ideas

- Use the Flickr API to search the Flickr Commons for images related to a research project.
- Can you train a neural net to distinguish between authentic period photographs and the output of a 'hipstamatic' filter used on a modern photograph?
- Get camera and software version information from historic WARC file then use SPARQL to get information about the timelines of those versions. Use to tell a story about how often institutions, particular sectors, etc. upgrade their technologies in particular periods.
- Test the output of the "Image synthesis" example with some of the photo forensics techniques of Hany Farid (see the Forensics project ideas for more information).
- Does multiscale image blending provide a more convincing way of presenting re-photography or georectification results like the ones we created in the lesson on "Measuring Images"?
- Use the "Interactive image measurements" example to measure architectural features from photographs, prior to building a 3D model of a heritage building (Bonnett 2003).
- Adapt the "Insect Identifier" to a class of entities you are interested in. One possibility would be to use a version of this code as the first stage of a multistage investigation. For example, say you are a collector of some kind of artifact like 1970s hi-fi equipment. The identifier can identify contemporary things like tape recorders, but not make fine distinctions within classes (like identifying a Braun reel-to-reel machine). Further steps would be to use transfer learning to train a neural net with a labelled

set of items; to create your own entity store for the training data, etc. Then visualize feature space of classified instances.

- Building on “Multi-view reconstruction” example, put an object on a turntable in front of green screen, rotate a few steps and photograph at each; then put in front of patterned background and move camera around photographing. Find best practices for photographing different kinds of objects; finally try 3D printing from *Mathematica*.

A model for single-image depth estimation

Neural network trained to reconstruct depth, given images labeled with a pair of points and distance to camera

http://www.wolfram.com/language/12/machine-learning-for-images/a-model-for-single-image-depth-estimation.html?product=mathematica

A model for super-resolution

Sharp upsampling from interpolated low-resolution images using machine learning.

http://www.wolfram.com/language/12/machine-learning-for-images/a-model-for-super-resolution.html?product=mathematica

An anomaly detector for images

When trained on normal images, detects ones that have been corrupted.

http://www.wolfram.com/language/12/high-level-machine-learning/train-an-anomaly-detector-for-images.html?product=mathematica

Color transformation

Render one image with the color palette of another.

http://www.wolfram.com/mathematica/new-in-9/advanced-image-processing-with-large-image-support/color-transformation.html

Convert images to vector graphics

Vectorize a bitmap.

http://www.wolfram.com/language/11/image-and-signal-processing/vectorize-a-bitmap-icon.html?product=mathematica

http://www.wolfram.com/language/12/new-in-image-processing/convert-images-to-vector-graphics.html?product=mathematica

Drawing tools

https://reference.wolfram.com/language/tutorial/EditingWolframLanguageGraphicsOverview.html

https://reference.wolfram.com/language/tutorial/InteractiveGraphicsPalette.html

Edge detection

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/detect-edges.html

EntropyFilter applied to images

Shows areas of higher information content, JPEG compression artifacts, etc.

https://reference.wolfram.com/language/ref/EntropyFilter.html

Feature-based Image inpainting and custom image inpainting with LearnDistribution

Could try this on portraits (like Lincoln example)

http://www.wolfram.com/language/11/image-and-signal-processing/feature-based-image-inpainting.html?product=mathematica

http://www.wolfram.com/language/12/high-level-machine-learning/create-a-custom-image-inpainter.html?product=mathematica

Original

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/repair-image-defects-by-inpainting.html

Find objects of a specific color

ChanVeseBinarize example.

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/find-objects-of-a-specific-color.html

Flickr

Service connection which requires a Flickr user account.

https://reference.wolfram.com/language/ref/service/Flickr.html

From images to graphs

MorphologicalGraph function

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-modeling/from-images-to-graphs.html

Gradient direction

http://www.wolfram.com/mathematica/new-in-9/advanced-image-processing-with-large-image-support/analyze-orientations-in-an-image.html

Example is rainbow colored thumb print.

http://www.wolfram.com/mathematica/new-in-9/advanced-image-processing-with-large-image-support/visualize-the-gradient-direction.html

'Hipstamatic'

Code gallery walks through an image processing pipeline for making a photograph look old.

http://www.wolfram.com/language/gallery/make-a-hipstamatic-filter/

Image denoising

http://www.wolfram.com/mathematica/new-in-9/advanced-image-processing-with-large-image-support/image-denoising.html

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/remove-noise-from-an-image.html

Image pyramids, feature detection and image salience

Pull features to detect most salient part of image

https://reference.wolfram.com/language/ref/ImageSaliencyFilter.html

http://www.wolfram.com/language/11/image-and-signal-processing/find-and-extract-salient-image-regions.html?product=mathematica

Also possible to do a multiscale version

http://www.wolfram.com/language/12/new-in-image-processing/imagepyramid-for-multiscale-image-representation.html?product=mathematica

http://www.wolfram.com/language/12/new-in-image-processing/multiscale-feature-detection.html?product=mathematica

https://reference.wolfram.com/language/ref/ImagePyramid.html

Image synthesis

Network-based method for determining optimal edges to cut a pair of photographs so that a new image can be synthesized from them.

http://www.wolfram.com/mathematica/new-in-9/enhanced-graphs-and-networks/image-synthesis.html

Insect identifier

Uses ImageIdentify and Entity properties with Bing Search.

http://www.wolfram.com/language/11/image-and-signal-processing/create-an-insect-identifier-tool.html?product=mathematica

Instagram

This example shows how to create a collage of media with a particular tag. Requires a user account.

http://www.wolfram.com/mathematica/new-in-10/curated-api-framework/make-a-collage-of-instagram-images.html

https://reference.wolfram.com/language/ref/service/Instagram.html

Interactive image measurements

Build an interactive interface for measuring features in an image.

http://www.wolfram.com/language/11/image-and-signal-processing/interactive-image-measurements.html?product=mathematica

JPEG

Importing meta information from JPEG images.

http://www.wolfram.com/mathematica/new-in-9/import-and-export-formats/import-meta-information-from-jpeg-images.html

Model an image as a function and generate random images

Network trained to reproduce image by treating it as a function from x,y pixels to r,g,b color values.

http://www.wolfram.com/language/11/neural-networks/model-an-image-as-a-function.html?product=-mathematica

Similar idea taken in a different direction: generating random images.

http://www.wolfram.com/language/11/neural-networks/generate-random-images.html?product=-mathematica

Multiscale Image Blending

Blend two different images at different levels of an image pyramid.

http://www.wolfram.com/language/12/new-in-image-processing/multiscale-image-blending.html?product=mathematica

Multi-view reconstruction

Recreate a 3D model from multiple 2D views.

http://www.wolfram.com/language/11/image-and-signal-processing/multi-view-reconstruction.html?product=mathematica

Extend to Kinect?

https://github.com/OpenKinect/libfreenect

Object recognition

ImageContents is easy to use and returns a dataset

http://www.wolfram.com/language/12/new-in-image-processing/built-in-object-detection-and-recognition.html?product=mathematica

http://www.wolfram.com/language/12/machine-learning-for-images/built-in-object-recognition.html?product=mathematica

Replace an image background

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/replace-the-image-background.html

Reverse web image search

Earlier example uses ImageIdentify, GoogleCustomSearch.

http://www.wolfram.com/language/11/external-services/reverse-web-image-search.html?product=mathematica

Replaced by new command which requires service credits.

https://reference.wolfram.com/language/ref/WebImageSearch.html

Semantic image segmentation

Using a neural net to identify people, animals, etc. and to paint regions of image with associated labels

http://www.wolfram.com/language/12/neural-network-framework/segment-images-semantically.html?product=mathematica

https://resources.wolframcloud.com/NeuralNetRepository/resources/Multi-scale-Context-Aggregation-Net-Trained-on-PASCAL-VOC2012-Data

Segmentation with Gaussian Mixture Model

http://www.wolfram.com/language/11/extended-probability-and-statistics/image-segmentation-with-gaussian-mixture-model.html?product=mathematica

SelectComponents and ComponentMeasurements in Segmentation

http://www.wolfram.com/language/11/image-and-signal-processing/updated-selection-and-formatting-for-component-ana.html?product=mathematica

Use smoothing filter for preprocessing

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/enhanced-segmentation-by-smoothing-an-image.html

Separate foreground from background

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/foreground-separation-and-removal.html

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/separate-foreground-from-a-similar-background.html

Similarity graph of images

http://www.wolfram.com/mathematica/new-in-9/advanced-image-processing-with-large-image-support/similarity-graph-of-images-using-earth-mover-dista.html

Texture-based segmentation

http://www.wolfram.com/mathematica/new-in-9/advanced-image-processing-with-large-image-support/texture-based-segmentation.html

There are also some built-in textures for testing

http://www.wolfram.com/mathematica/new-in-8/enhanced-2d-and-3d-graphics/built-in-texture-example-data.html

Use pre-trained neural net models to visualize features

http://www.wolfram.com/language/12/neural-network-framework/use-pre-trained-models-to-visualize-features.html?product=mathematica

Visualize stereogram images

This example creates red / green anaglyphs, so need 3D glasses to view.

http://www.wolfram.com/mathematica/new-in-9/import-and-export-formats/visualize-stereogram-images.html

References

Bonnett, John. “Following in Rabelais’ Footsteps: Immersive History and the 3D Virtual Buildings Project,” *Journal of the Association for History and Computing* vol 6, no 2 (September 2003). https://quod.lib.umich.edu/j/jahc/3310410.0006.202/--following-in-rabelais-footsteps-immersive-history-and-the-3d?rgn=main;view=fulltext

pr-linguistics

Linguistics

Project ideas

- Use the programmable linguistic interface to create a custom grammar / natural language interface for a specific limited vocabulary (e.g., parsing monetary amounts in business news articles or other sources.)

Build neural nets for any language

Byte-pair-encoding (BPE) sub-word tokenization for English and French.

http://www.wolfram.com/language/12/natural-language-processing/build-neural-nets-for-any-language.html?product=mathematica

Enhance sentiment analysis using transfer learning

Example uses Wikipedia word vectors for pretrained word-embedding layer (GloVe) and sentiment-labelled movie reviews to train for specific application.

http://www.wolfram.com/language/12/natural-language-processing/enhance-sentiment-analysis-using-transfer-learning.html?product=mathematica

Estimate word-length distributions of English parts of speech from HistogramDistribution

Calculate, for example, expected length of a preposition.

http://www.wolfram.com/mathematica/new-in-8/nonparametric-derived-and-formula-distributions/use-nonparametric-distributions-like-any-other-dis.html

Fill in missing words with SequencePredict

Example is: “The cat * on the mat”; given set of candidates and built-in sequence model for English, output ranked (is, was, sat, jumped, ..., avocado).

http://www.wolfram.com/language/12/natural-language-processing/fill-in-missing-words.html?product=mathematica

Generate text in English and represent text as sequence of vectors

GPT Transformer trained on BookCorpus data

https://resources.wolframcloud.com/NeuralNetRepository/resources/GPT-Transformer-Trained-on-BookCorpus-Data

BERT trained on BookCorpus and Wikipedia. Example includes sentence analogies.

https://resources.wolframcloud.com/NeuralNetRepository/resources/BERT-Trained-on-BookCorpus-

and-English-Wikipedia-Data

Model word lengths in different languages by binomial distributions

http://www.wolfram.com/mathematica/new-in-8/parameter-estimation-and-testing/model-word-lengths-by-binomial-distributions.html

Programmable linguistic interface

Create a custom grammar / natural language interface.

https://reference.wolfram.com/language/guide/ProgrammableLinguisticInterface.html

Programming Historian: Transliterating non-ASCII characters

Example is Russian human rights organization Memorial, which has a database of 3M entries for people who were arrested or executed during Stalin's regime. Good example for working with Unicode and taking advantage of *Mathematica*'s built-in knowledge of natural languages.

https://programminghistorian.org/en/lessons/transliterating

Data page

http://lists.memo.ru/d1/f1.htm

Query Wolfram Knowledgebase with natural language

Defines a grammar to recognize MusicAct and MusicAlbum entity types.

http://www.wolfram.com/language/11/text-and-language-processing/query-the-wolfram-knowledge-base-using-natural-lang.html?product=mathematica

Sapping Attention: Meaning chains with word embeddings

http://sappingattention.blogspot.com/2018/06/meaning-chains-with-word-embeddings.html

http://sappingattention.blogspot.com/2016/12/some-notes-on-corpora-for-diachronic.html

Transformer neural nets (GPT, BERT) for text

New for machine translation - includes link to paper Vaswani et al "Attention is all you need" (NIPS 2017)

http://www.wolfram.com/language/12/neural-network-framework/use-transformer-neural-nets.html?product=mathematica

Word sense disambiguation

ELMo trained on about 1B words. Given sentences with 'Apple', distinguishes between ones about the company and ones about the fruit.

http://www.wolfram.com/language/12/natural-language-processing/represent-word-semantics-in-

context.html?product=mathematica

pr-linked-open-data

Linked open data

Project ideas

- Replicate the *Programming Historian* tutorial using SPARQL, networks and timelines in *Mathematica*.
- The “Map of population distributions” example below uses entity data. Can you rewrite it to draw on SPARQL queries instead?

Map of population distributions

Map with pie charts showing child, adult and elderly populations for 25 largest countries in the world.

http://www.wolfram.com/language/12/geographic-visualization/population-distributions.html?product=mathematica

Programming Historian: Using SPARQL to access linked open data

Palladio example of a gallery of images and a timeline.

https://programminghistorian.org/en/lessons/graph-databases-and-SPARQL

pr-management-science

Management science

Cliques, covers and independent sets

Graphs for resource allocation problems.

https://reference.wolfram.com/language/guide/GraphCliquesCoversAndIndependentSets.html

Convert graphs to matrix representations

Adjacency, Incidence and Kirchhoff matrices.

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-analysis/convert-to-matrix-representations.html

Graphs and networks

Guide to graph and network commands.

https://reference.wolfram.com/language/guide/GraphsAndNetworks.html

Reliability of power grids

Simulate the effects of power station failure in a power grid.

http://www.wolfram.com/mathematica/new-in-10/enhanced-graphs-and-networks/reliability-of-power-grids.html

pr-military-history

Military history

Forces and casualties in historical military conflicts

Analyze forces and casualties of Second Punic War using Military Conflict entities and DateListStepPlot command.

http://www.wolfram.com/language/12/cultural-and-historical-entities/compare-forces-and-casualties-in-military-conflicts.html?product=mathematica

Historical battles and territorial changes

Locations of battles and territorial changes over the Hundred Years War using Military Conflict and Historical Country entities.

http://www.wolfram.com/language/12/cultural-and-historical-entities/visualize-historical-battles-and-territorial-changes.html?product=mathematica

Programming Historian: Visualizing data with Bokeh and Pandas

This is a tutorial on plotting historical data using the WWII THOR dataset of aerial bombings. (It also includes links to other suitable data including Scottish Witchcraft Trials, Civil Unrest Events since WWII and the Trans-Atlantic Slave Trade Database). Try replicating the analyses with *Mathematica*.

https://programminghistorian.org/en/lessons/visualizing-with-bokeh

pr-music

Music

Text recognition for guitar tablature

Extract Led Zeppelin's "Stairway to Heaven" from sheet music and play the MIDI file.

http://www.wolfram.com/language/12/machine-learning-for-images/text-recognition-to-read-music-sheets.html?product=mathematica

pr-names

Names

Project ideas

- Adapt the group portrait idea from the Signers of the Declaration of Independence example to portray musical acts that played Woodstock 1969 or Live Aid 1985.

- Take an American text that has a number of people's names in it, and use the historical popularity of names to predict the most likely date the text was written.

Age

Predict a person's age from their first name.

http://www.wolfram.com/mathematica/new-in-10/built-in-classifier-collection/predict-a-persons-age-from-their-first-name.html

```
In[•]:= Table[Plot[PDF[Predict["NameAge", n, "Distribution"], x], {x, 0, 100},
  PlotRange → All, Filling → Bottom, PlotLabel → n], {n, {"Juliet", "Romeo"}}]
```

Gender

Classifier finds the most likely gender of a given name and gives probabilities for ambiguous ones.

http://www.wolfram.com/mathematica/new-in-10/built-in-classifier-collection/determine-the-gender-of-given-name.html

Popularity

Analyze the historical popularity of given names throughout US history.

http://www.wolfram.com/language/11/new-visualization-domains/track-the-popularity-of-names.html?product=mathematica

http://www.wolfram.com/language/12/cultural-and-historical-entities/analyze-the-popularity-of-names.html?product=mathematica

Sherlock vs Watson

This example finds the number of times each character is mentioned in Arthur Conan Doyle's novels and plots the results.

http://www.wolfram.com/language/11/text-and-language-processing/sherlock-vs-watson.html?product=mathematica

Signers of the Declaration of Independence

Extracts names from Declaration of Independence, verifies that people were alive when it was signed, gathers their images, then assembles them into a group portrait.

http://www.wolfram.com/mathematica/new-in-10/domain-specific-language-interpretation/get-birth-dates-images-and-other-information-from.html

pr-networks

Networks

Analyze large and complex networks

Example is US railroad graph.

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-analysis/analyze-large-and-complex-networks.html

Clustering in small-world networks

Quantify network robustness with respect to perturbation for epidemiology, etc.

http://www.wolfram.com/mathematica/new-in-9/social-network-analysis/clustering-in-small-world-networks.html

Dog-friendly walkways

Uses network programming to find a walking tour that avoids bad dogs.

http://www.wolfram.com/mathematica/new-in-10/enhanced-graphs-and-networks/dog-friendly-walkways.html

Plan a trip in the London Underground

Shortest paths on the London Underground.

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-analysis/london-underground.html

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-analysis/trip-planning.html

pr-neural-nets

Neural nets

Machine learning

Language guide.

https://reference.wolfram.com/language/guide/MachineLearning.html

Aligning books and movies

Includes large book corpus used to create neural sentence embedding (BookCorpus). Paper and data available on website.

https://yknzhu.wixsite.com/mbweb

Avoid overfitting with a hold-out set

http://www.wolfram.com/language/11/neural-networks/avoid-overfitting-using-a-hold-out-set.html?product=mathematica

Find the optimal parameters of a classifier

Uses BayesianMinimization to find the best set of parameters for an SVM used on Titanic Data.

http://www.wolfram.com/language/11/improved-machine-learning/find-the-optimal-parameters-of-a-classifier.html?product=mathematica

Fooling neural networks

Create an image which looks one way to us and a different way to a neural net.

http://www.wolfram.com/language/12/machine-learning-for-images/fooling-neural-networks.html?product=mathematica

Neural network sensitivity map

Which features contribute to classification or misclassification of images?

http://www.wolfram.com/language/12/machine-learning-for-images/neural-network-sensitivity-map.html?product=mathematica

Obtain and use pre-trained models

This page has a lot of good examples.

http://www.wolfram.com/language/12/neural-network-framework/obtain-and-use-pre-trained-models.html?product=mathematica

Out of core image classification

Digit classifier task. Classify images directly from file paths: only small amounts of training data are in memory at any point in time.

http://www.wolfram.com/language/11/neural-networks/out-of-core-image-classification.html?product=mathematica

The overall technique for out-of-core could be very useful for other kinds of classification tasks

Train a classifier to differentiate between light and dark colors

https://reference.wolfram.com/language/workflow/TrainAMachineLearningClassifier.html

Train a convolutional network to classify both labels and sublabels of images

CIFAR-100 dataset

http://www.wolfram.com/language/11/neural-networks/multi-task-learning.html?product=mathematica

Train a daytime / nighttime classifier

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/daytime-nighttime-classification.html

http://www.wolfram.com/mathematica/new-in-10/highly-automated-machine-learning/distinguish-daytime-from-nighttime-pictures.html

pr-page-images

Page images

Deconvolve a blurred image

Document scans are occasionally blurry and need to be deblurred for OCR. This version assumes you know the point spread function.

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/deconvolve-a-blurred-image.html

Perform typo correction without a dictionary using Markov processes

In this example, typos are introduced by replacing 20% of the characters with an immediate neighbor on the QWERTY keyboard.
Could this be used to improve OCR?

http://www.wolfram.com/mathematica/new-in-10/enhanced-random-processes/perform-typo-correction-without-a-dictionary.html

Programming Historian: Cleaning OCR'd texts with regular expressions

Example is a page from Congressional Directory of 1887. Author writes a lot of regex to clean one page, but presumably it works for other pages in the same volume?

https://programminghistorian.org/en/lessons/cleaning-ocrd-text-with-regular-expressions

Programming Historian: Extracting illustrated pages from digital libraries

Examples from Hathi Trust API (includes metadata about which pages have pictures) and Internet Archive (Abbyy fine reader output includes XML which can be parsed for possible presence of a picture).

https://programminghistorian.org/en/lessons/extracting-illustrated-pages

Programming Historian: Generating an ordered data set from a text file

Parses raw OCR output, isolates metadata and generates a Python dictionary (in *Mathematica*, would use an association). Levenshtein distance, Roman to Arabic numerals, regular expressions, parsing dates, HTML output.

https://programminghistorian.org/en/lessons/generating-an-ordered-data-set-from-an-OCR-text-file

StackExchange: Image processing for pages

Make handwriting more legible by correcting for bleed through.

https://mathematica.stackexchange.com/questions/56137/automatically-lining-up-two-images-that-have-some-common-elements

Cleaning mildew from old documents.

https://mathematica.stackexchange.com/questions/17125/cleaning-mildew-from-old-documents-using-mathematica-image-processing

Text recognition to read rasterized plots

In this example, the axes, axes labels, and horizontal and vertical value ranges are automatically extracted from a scanned graph. Recognizing the plot itself would require further image processing.

http://www.wolfram.com/language/12/machine-learning-for-images/text-recognition-to-read-rasterized-plots.html?product=mathematica

Uneven illumination

Use segmentation with LocalAdaptiveBinarize

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/use-segmentation-to-improve-text-recognition.html

pr-programming

Programming

Project ideas

- Answer a question on the Mathematica Stack Exchange
- Look through the demonstrations, find one that you like, then analyze the code to determine what you would need to learn to adapt it to a project of your own and implement it
- Write up one of your projects as a Wolfram Demonstration or write up a function for the Wolfram Function Repository
- Use *Mathematica*'s rule-based expression transformation to implement a compiler for a 'little language' (Bentley 1986)
- Using the "File Format Pictures" below, try writing an importer for a new file format. This could be used for the basis of a project on Data Archaeology or Forensics (see entry)

Create a button to open another notebook

For a large project it sometimes makes sense to keep multiple notebooks rather than put everything in one. When I do this, I also create an index notebook with links to all of the other notebooks used in the project.

https://mathematica.stackexchange.com/questions/8881/creating-a-live-index-notebook-to-load-other-notebooks/8916#8916

File format pictures

Includes PDF and about a hundred others.

https://github.com/corkami/pics/tree/master/binary

Follow expression traversals

Use the Echo and EchoFunction commands to figure out which parts of an expression are visited during Cases, Select, etc.

http://www.wolfram.com/language/11/core-language/follow-expression-traversals.html?product=-mathematica

Hacker News

Has an API that returns JSON.

https://github.com/HackerNews/API

How To: Work with Rules

Mathematica contains a very powerful language for transforming expressions.

https://reference.wolfram.com/language/howto/WorkWithRules.html

Mathematica / Wolfram Language

There are a number of ways to use *Mathematica* to analyze itself.

http://www.wolfram.com/language/11/richer-knowledgebase-access/visualize-the-wolfram-language.html?product=mathematica

http://www.wolfram.com/language/11/richer-knowledgebase-access/wolfram-language-self-analysis.html?product=mathematica

http://www.wolfram.com/language/11/graphs-and-networks/network-of-wolfram-language-functions.html?product=mathematica

http://www.wolfram.com/language/11/graphs-and-networks/relationship-graphs.html?product=mathematica

R Language

Interface to R.

https://reference.wolfram.com/language/RLink/tutorial/UsingRLink.html

https://stat.ethz.ch/R-manual/R-patched/library/datasets/html/00Index.html

http://www.wolfram.com/mathematica/new-in-9/built-in-integration-with-r/

Slashdot

A statistical analysis of the Slashdot social network.

http://www.wolfram.com/mathematica/new-in-9/social-network-analysis/statistical-analysis-of-the-slashdot-social-networ.html

Store *Mathematica* expression in a QR code

QR code holds almost 3000 ASCII characters

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/evaluate-an-expression-stored-in-a-qr-code.html

Wolfram demonstrations

Interactive demos that can be explored online by anyone.

http://demonstrations.wolfram.com/topics.php

Wolfram Function Repository

Instant add-on *Mathematica* functions (with source code).

https://resources.wolframcloud.com/FunctionRepository/

References

Bentley, Jon. “Little Languages,” Programming Pearls, *Communications of the ACM* vol 29, no 8, pp 711-721.
http://staff.um.edu.mt/afra1/seminar/little-languages.pdf

pr-social-media

Social media

Automate social media analysis

Uses DocumentGenerator and DeliveryFunction (Cloud Deploy) for a Twitter Analysis that is tweeted daily.

http://www.wolfram.com/language/11/cloud-and-web-interfaces/automate-a-social-media-analysis.html?product=mathematica

Facebook

Example uses Facebook API to retrieve friends’ birth dates and send them a message wishing them a happy birthday.

http://www.wolfram.com/mathematica/new-in-10/curated-api-framework/wish-your-facebook-friends-a-happy-birthday.html

Communities in Facebook friend networks: try modifying this code with a tooltip, so you can mouse over nodes and see who each one is.

http://www.wolfram.com/mathematica/new-in-9/social-network-analysis/communities-in-facebook-friend-networks.html

Twitter

Sending messages from *Mathematica* notebooks directly to Twitter.

http://www.wolfram.com/mathematica/new-in-10/curated-api-framework/tweet-a-plot.html

A bot that automatically replies to tweets with a given hashtag.

http://www.wolfram.com/mathematica/new-in-10/curated-api-framework/automatically-reply-to-tweets.html

Create a data resource from personal Twitter data. See the section on the Wolfram Data Repository in the "Datasets" section for more information.

http://www.wolfram.com/language/11/cloud-storage-and-operations/create-a-data-resource.html?product=mathematica

pr-social-network-analysis

Social network analysis

Project ideas

- Replicate the *Programming Historian* tutorial on "Maps of correspondence" then extend the project to include some social network analysis.

Centrality

Comparing networks on centrality measures.

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-analysis/compute-the-betweenness-centrality.html

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-analysis/degree-centrality-in-social-networks.html

Graph components and connectivity

Guide to commands that may be used to explore limited connections between groups of people in a network.

https://reference.wolfram.com/language/guide/GraphComponents.html

Historian's Macroscope

Chapter on social networks for history in open access textbook.

http://www.themacroscope.org/?page_id=308

Historical Network Research website

This website contains a large number of resources for Social Network Analysis in historical research. Some particularly useful pages for getting started are the following:

http://historicalnetworkresearch.org/resources/first-steps/

http://historicalnetworkresearch.org/resources/external-resources/

http://historicalnetworkresearch.org/bibliography/

Homophily and assortativity mixing

Tendency of network members to associate with others who are similar to them in some way.

http://www.wolfram.com/mathematica/new-in-9/social-network-analysis/homophily-and-assortativity-mixing.html

Import and export graph formats

Mathematica supports GML, GXL, Pajek.

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-modeling/import-and-export-graph-formats.html

Programming Historian: Maps of correspondence

Mapping correspondence of Boston-area abolitionist women. Lesson folder includes CSV file of letters.

https://programminghistorian.org/en/lessons/using-javascript-to-create-maps

Programming Historian: Big data and network analysis

Example is network analysis of connections between national business elites in Canada in 1912.

https://programminghistorian.org/en/lessons/dealing-with-big-data-and-network-analysis-using-neo4j

Data

http://jgmackay.com/JGM/News/Entries/2017/1/16_Networks _of _Canadian _Business _Elites __Historical _Corporate _Interlock _Networks _circa _ 1912.html

Programming Historian: From hermeneutics to data to networks

Case study is first person narrative of Jewish survivor of the Holocaust. Includes link to text and

author's analysis.

https://programminghistorian.org/en/lessons/creating-network-diagrams-from-historical-sources

Programming Historian: Temporal (social) network analysis

Study development of a social network over time by adding begin and end dates (onset and terminus) to node and edge lists. Dataset is collaborations between French Gothic illuminated manuscript workshops between 1260 and 1320. Workshops are just numbered, however, instead of named.

https://programminghistorian.org/en/lessons/temporal-network-analysis-with-r

Programming Historian: Correspondence analysis

"Correspondence analysis (CA) produces a two or three dimensional plot based on relationships among two or more categories of data. These categories could be 'members and clubs,' 'words and books' or 'countries and trade agreements.'" AKA multi-dimensional scaling or bivariate network analysis. Uses R Library. Example is Canadian Parliamentary Committees under Harper and Trudeau. Data archived in Zenodo.

https://programminghistorian.org/en/lessons/correspondence-analysis-in-R

Social network analysis

Language guide and some visualization options.

https://reference.wolfram.com/language/guide/SocialNetworks.html

http://www.wolfram.com/language/11/graphs-and-networks/higher-fidelity-drawing.html?product=-mathematica

http://www.wolfram.com/mathematica/new-in-8/graph-and-network-analysis/analyze-social-networks.html

Social network modeling

Example that fits the shifted Gompertz distribution to Google Trends interest in Facebook over a decade.

http://www.wolfram.com/language/11/extended-probability-and-statistics/social-networks-modeling.html?product=mathematica

pr-social-sciences

Social sciences

Convert currencies with historical exchange rates

http://www.wolfram.com/mathematica/new-in-10/dimensional-variables/convert-currencies-using-historical-exchange-rates.html

Find effect of inflation on grocery prices

http://www.wolfram.com/mathematica/new-in-10/dimensional-variables/find-the-effect-of-inflation-on-grocery-prices.html

Historical socioeconomic data

Example is unemployment in Africa 2016

http://www.wolfram.com/language/12/financial-and-socioeconomic-entities/visualize-historical-socioeconomic-data.html?product=mathematica

pr-stylometry

Stylometry

The statistical analysis of variations in style between texts, authors or genres is called stylometry. Stylometric software often makes use of measurements that are not obvious to human readers when they read or write, no matter how familiar they are with the texts in question. One of the main uses of stylometry is to attribute a particular text to a particular author on the basis of literary style.

Project ideas

- Code for computing Mendenhall's Characteristic Curves of Composition is given in the section on Laramée's *Programming Historian* tutorial below. Try implementing the other two methods from his tutorial.
- The Laramée tutorial from the *Programming Historian* includes links to a number of articles on authorship attribution. Try implementing the methods from one of those papers and applying it to the disputed *Federalist Papers*.

Blog Posts: Revisiting the Disputed Federalist Papers and Automated Authorship Verification

These two blog posts by Daniel Lichtblau describe the application of a novel and sophisticated stylometric method to the disputed *Federalist Papers*, using *Mathematica*'s abilities to do string processing, dimension reduction, image processing and machine learning.

https://blog.wolfram.com/2018/10/11/revisiting-the-disputed-federalist-papers-historical-forensics-with-the-chaos-game-representation-and-ai/

https://blog.wolfram.com/2019/05/28/automated-authorship-verification-did-we-really-write-those-blogs-we-said-we-wrote/

Code Gallery: Determine the Author of a Text

Training a classifier to distinguish the works of Shakespeare, Oscar Wilde and Victor Hugo.

http://www.wolfram.com/language/gallery/determine-the-author-of-a-text/

Find which author wrote a text

Uses a Markov classifier on Gutenberg texts

http://www.wolfram.com/mathematica/new-in-10/highly-automated-machine-learning/find-which-author-wrote-a-text.html

Programming Historian: Introduction to Stylometry with Python

This tutorial by François Dominic Laramée shows three stylometric methods (Mendenhall's Characteristic Curves of Composition, Kilgariff's Chi-Squared Method and John Burrows' Delta Method) used on *The Federalist Papers*.

https://programminghistorian.org/en/lessons/introduction-to-stylometry-with-python

The *Federalist Papers* are available in the Wolfram Data Repository.

In[•]:=
```
(*ResourceObject["The Federalist Papers"]*)
```

Here we use titles to parse out the individual texts. We didn't clip any of the header information, however.

In[•]:=
```
(*federalistPapers=
 Dataset[Association/@StringCases[ResourceData["The Federalist Papers"],
    Shortest["FEDERALIST No. "~~no:{WordCharacter}..~~WordBoundary~~
       txt__~~"FEDERALIST"|EndOfString]→List[no→txt],Overlaps→True]]*)
```

We store authorship information in an association.

In[•]:=
```
(*federalistAuthorship=<|"Madison"→{10,14,37,38,39,40,41,42,43,44,45,46,47,48},
   "Hamilton"→
    {1,6,7,8,9,11,12,13,15,16,17,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,
     59,60,61,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85},
   "Jay"→{2,3,4,5},
   "Shared"→{18,19,20},
   "Disputed"→{49,50,51,52,53,54,55,56,57,58,62,63},
   "TestCase"→{64}|>;*)
```

Each of the texts written by a particular author is concatenated and stored in a dataset.

In[•]:=
```
(*federalistByAuthor=Dataset[Table[Association[k→TextWords@StringJoin@
       Flatten@Normal@Values@federalistPapers[federalistAuthorship[k]]],
   {k,Keys@federalistAuthorship}]]*)
```

This function computes Mendenhall's Characteristic Curves of Composition (1887).

In[•]:=
```
(*mendenhallCurve[authstr_,corpusassoc_]:=
 ListPlot[SortBy[Select[Tally[StringLength/@Normal[
       corpusassoc[Position[Keys@federalistAuthorship,authstr][[1,1]]][authstr]]],
    #[[1]]≤15&],First],Joined→True,PlotLabel→authstr]*)
```

Here is how we plot the curves.

In[•]:=
```
(*mendenhallCurve[#,federalistByAuthor]&/@Keys[federalistAuthorship]*)
```

Wolfram Demonstration: Comparing Writing Styles of Famous Texts

Demonstration of 11 different stylometric criteria.

http://demonstrations.wolfram.com/ComparingWritingStylesOfFamousTexts/

References

Laramée, François Dominic. “Introduction to stylometry with Python,” *The Programming Historian* 7 (2018). https://programminghistorian.org/en/lessons/introduction-to-stylometry-with-python

Mendenhall, T. C. “The Characteristic Curves of Composition.” *Science* (11 Mar 1887): Vol. ns-9, Issue 214S, pp. 237-246.
DOI: 10.1126/science.ns-9.214S.237
https://science.sciencemag.org/content/ns-9/214S/237

pr-text-mining

Text mining

Project ideas

- Replicate the *Programming Historian* tutorial on “Data mining the Internet Archive collection” then augment with Entities or SPARQL queries.

Programming Historian: Corpus Analysis with Antconc

The Programming Historian site contains a large number of tutorials for doing common tasks in the digital humanities. In this lesson, Heather Froehlich shows how to use off-the-shelf concordance software called Antconc to make comparisons between texts on a large scale. The Programming Historian lessons are developed with free tools, but the techniques described can be readily implemented in *Mathematica*. One way to develop your programming skills is to see how much of an analysis like this you can replicate with the techniques you are learning in this course.

https://programminghistorian.org/en/lessons/corpus-analysis-with-antconc

Programming Historian: Basic text processing in R

Examples are presidential State of the Union Addresses (which are available in *Mathematica*). Authors show how to do exploratory and stylometric analyses and document summarization.

https://programminghistorian.org/en/lessons/basic-text-processing-in-r

Programming Historian: Text mining through the HathiTrust Research Center’s Extracted Features dataset

Includes downloadable samples to work with. A lot of the material would fit into datasets (including database-like querying) and could make use of list plotting in *Mathematica*.

https://programminghistorian.org/en/lessons/text-mining-with-extracted-features

Programming Historian: Data mining the Internet Archive collection

Downloading and mining MARC records of the 7K Antislavery Collection (BPL) at the Internet Archive.

https://programminghistorian.org/en/lessons/data-mining-the-internet-archive

Sapping Attention: Google n-grams change over time

Need to have access to the Google n-grams database for examples?

http://sappingattention.blogspot.com/2017/07/what-is-described-as-belonging-to.html

http://sappingattention.blogspot.com/2012/02/poor-mans-sentiment-analysis.html

http://sappingattention.blogspot.com/2012/07/do-revolutionaries-really-read-history.html

Text analysis

Guide to sources of text, visualization, parsing, comparison, and content extraction and analysis.

https://reference.wolfram.com/language/guide/TextAnalysis.html

Text manipulation

Guide to commands for manipulating text, from characters and strings through higher level natural language processing.

https://reference.wolfram.com/language/guide/ProcessingTextualData.html

Text normalization

Before doing text analysis or mining, it is common to normalize texts (e.g., removing diacritics, forcing to consistent case, etc.) This guide provides an overview of options.

https://reference.wolfram.com/language/guide/TextNormalization.html

pr-time-series

Time series

Continuous and discrete

Options for representing both continuous and discrete time series.

http://www.wolfram.com/mathematica/new-in-10/time-series/represent-time-series-data-with-timeseries-and-eve.html

Create time series with dates

Use calendar dates as time stamps.

http://www.wolfram.com/mathematica/new-in-10/time-series/create-timeseries-with-dates.html

Day specification

TimeSeriesWindow and TimeSeriesResample allow you to specify “BusinessDay” or “Monday” etc.

http://www.wolfram.com/language/12/probability-and-statistics/use-day-specification-in-time-series-operations.html?product=mathematica

http://www.wolfram.com/language/12/probability-and-statistics/select-business-days-in-a-time-series.html?product=mathematica

Extract part of a time series

Use of the TimeSeriesWindow command.

http://www.wolfram.com/mathematica/new-in-10/time-series/extract-part-of-a-time-series.html

Extract time series from internet source

Daily mean temperatures for Great Lakes from NOAA.

http://www.wolfram.com/language/11/time-series-processing/data-from-internet-sources.html?product=mathematica

Filter a time series

Dataset is US accidental deaths.

http://www.wolfram.com/mathematica/new-in-10/time-series/filter-a-time-series.html

Missing data

Work with time series containing missing data.

http://www.wolfram.com/mathematica/new-in-10/time-series/work-with-time-series-containing-missing-data.html

Plot time and event series

Population growth of the US from 1970 to 2013.

http://www.wolfram.com/mathematica/new-in-10/enhanced-visualization/plot-time-and-event-series.html

Trends and seasonalities

Example is number of international airline passengers from 1949-60.

http://www.wolfram.com/language/11/time-series-processing/trends-and-seasonalities.html?product=mathematica

pr-video

Video

Project ideas

- Take the example of object recognition and tracking in video and see if you can get it to work for snippet of a historical film (from the Prelinger archives, for example)
- Try repurposing the video dynamics example for the Zapruder film. There are a lot of frame-by-frame samples on YouTube, and perhaps elsewhere (Rosenbaum 2013).

Augmented reality

Uses desktop or laptop camera and optical flow on hand gesture.

http://www.wolfram.com/language/11/image-and-signal-processing/augmented-reality-on-rotating-objects.html?product=mathematica

Detect rotating objects

Uses ImageDisplacements and Curl to capture dynamic features in video sequences.

http://www.wolfram.com/language/11/image-and-signal-processing/detect-rotating-objects.html?product=mathematica

Explore video dynamics

Example is of mitosis.

http://www.wolfram.com/language/11/image-and-signal-processing/dynamics-of-mitosis.html?product=mathematica

Keypoint-based video stabilization

Clean up shaky video.

http://www.wolfram.com/language/12/new-in-image-processing/keypoint-based-video-stabilization.html?product=mathematica

Object recognition and tracking in video

http://www.wolfram.com/language/12/machine-learning-for-images/object-recognition-and-tracking-in-videos.html?product=mathematica

http://www.wolfram.com/mathematica/new-in-9/advanced-image-processing-with-large-image-support/tracking-objects-in-an-image-sequence.html

http://www.wolfram.com/mathematica/new-in-8/comprehensive-image-processing-environment/-track-moving-objects.html

Green screen / chroma keying

http://www.wolfram.com/mathematica/new-in-10/enhanced-image-processing/chroma-key-composition.html

Programming Historian: Introduction to audiovisual transcoding etc.

https://programminghistorian.org/en/lessons/introduction-to-ffmpeg

Remove green screen

Uses color neighborhoods.

http://www.wolfram.com/language/12/new-in-image-processing/color-neighborhoods.html?product=-mathematica

Slow motion with optical flow

Interpolates warped images based on optical flow to slow down a video sequence.

http://www.wolfram.com/language/11/image-and-signal-processing/slow-motion-using-optical-flow.html?product=mathematica

References

Rosenbaum, Ron. “What does the Zapruder film really tell us?” *Smithsonian Magazine* (Oct 2013). https://www.smithsonianmag.com/history/what-does-the-zapruder-film-really-tell-us-14194/

pr-web-services

Web services

Project ideas

- Try implementing the *Programming Historian* tutorial on “Reshaping JSON” using *Mathematica*. Then try substituting the European 2016 Twitter data from the Wolfram Repository.
- Try implementing the *Programming Historian* tutorial on web APIs using *Mathematica* and Wolfram Cloud.

CORE

World’s largest collection of open access research papers with APIs for text mining (135M papers).

https://core.ac.uk

https://core.ac.uk/services/#access-to-raw-data

https://core.ac.uk/services/api/

Federal Reserve Economic Data

Service connection. Requires credentials.

https://reference.wolfram.com/language/ref/service/FederalReserveEconomicData.html

Open PHACTS

Pharmacological data accessible through the service framework.

https://reference.wolfram.com/language/ref/service/OpenPHACTS.html

Penguin

The publisher Penguin has an API that lets you retrieve authors, works, titles and events.

http://www.penguinrandomhouse.biz/webservices/rest/

Programming Historian: Reshaping JSON

In this tutorial complex JSON files are parsed into flat CSVs. Examples come from art museum API and small sample of public tweets (I could use Wolfram Data Repo Europe 2016 data instead).

https://programminghistorian.org/en/lessons/json-and-jq

Programming Historian: Creating web APIs with Python and Flask

Example of working API is retrieving material on Sensationalism and Historical Fires from Chronicling America Historical Newspaper API. The project is a distant reading API that has a MySQL database of science fiction books.

https://programminghistorian.org/en/lessons/creating-apis-with-python-and-flask

pr-workflow

Workflow

Project ideas

- Use IFTTT to archive tweets or other social media posts to a Google spreadsheet or to Dropbox then import and analyze in *Mathematica*
- Using chat, email and other tools, try setting up a Slack-style system for a small research team
- Explore a network of linked files for basins of attraction. Adapt the “Automate Web Tasks” example to do a random walk around a network, keeping track of all hops. Are there nodes that are visited frequently? Never visited?

Automate filling of forms on webpages

WebExecute - note that example uses JavaScript but WebExecute help file shows how to use XPath, CSS Selectors, HTML Tags, etc.

http://www.wolfram.com/language/12/external-system-integration/automate-filling-forms-on-webpages.html?product=mathematica

Automate web tasks

Example shows random clicking to switch to new linked pages using WebExecute.

http://www.wolfram.com/language/12/external-system-integration/automate-web-tasks.html?product=mathematica

Chat

Mathematica has extensive support for chat sessions. This workflow guide shows how to use them.

https://reference.wolfram.com/language/workflowguide/UsingChat.html

Download data from websites and display progress bar

http://www.wolfram.com/mathematica/new-in-9/full-range-of-web-access-support/display-a-progress-bar.html

Dropbox

You can download and upload data to your Dropbox account in *Mathematica*.

http://www.wolfram.com/mathematica/new-in-10/curated-api-framework/manage-files-in-dropbox.html

Email

There is extensive support for email in *Mathematica*. This page contains links to a number of examples.

http://www.wolfram.com/language/12/mail-and-messaging/?product=mathematica

Send emails

http://www.wolfram.com/language/12/mail-and-messaging/send-email-with-wolfram-language-objects.html?product=mathematica

Send automated messages to your phone

http://www.wolfram.com/language/12/mail-and-messaging/send-automated-messages-to-your-phone.html?product=mathematica

Visualize a personal email timeline

http://www.wolfram.com/mathematica/new-in-10/semantic-data-import/visualize-a-personal-email-

timeline.html

File format conversion works with audio, image and tabular files

Can also do batch jobs with FileConvert, FileSystemMap, FileSystemScan

http://www.wolfram.com/language/12/data-import-and-export/directly-convert-between-file-formats.html?product=mathematica

Iconizing datasets and programs

Allows long code expressions to be collapsed, hiding formatting options. Good for deploying initialization code and code used to create figures.

http://www.wolfram.com/language/12/notebook-interface/iconize-datasets-and-programs.html?product=mathematica

http://www.wolfram.com/language/12/notebook-interface/interactively-iconize-content.html?product=mathematica

IFTTT (if this, then that)

Using IFTTT monitor Twitter and send information to a Channel, in Wolfram Desktop session, see continuously updated WordCloud.

http://www.wolfram.com/language/11/channel-framework/create-a-dynamic-ifttt-recipe.html?product=mathematica

Programming Historian: Automated downloading with wget

Culminates in mirroring of activehistory.ca

https://programminghistorian.org/en/lessons/automated-downloading-with-wget

Programming Historian: Applied archival downloading with wget

Examples of recursive retrieval from LAC and Natl Archives of Australia

https://programminghistorian.org/en/lessons/applied-archival-downloading-with-wget

Programming Historian: Downloading multiple records using query strings

Searching for ‘negro’ and ‘mulatto’ in Old Bailey Online by deciphering URLs.

https://programminghistorian.org/en/lessons/downloading-multiple-records-using-query-strings

Programming Historian: Intro to beautiful soup

Example extracts data from online Biographical Dictionary of the US Congress, 1774-Present.

https://programminghistorian.org/en/lessons/intro-to-beautiful-soup

Programming Historian: Preserving your research data

Useful idea: someone should be able to look at your files and know what you did. That someone will probably be you a few months from now.

https://programminghistorian.org/en/lessons/preserving-your-research-data

Scheduled tasks

http://www.wolfram.com/language/11/cloud-and-web-interfaces/scheduled-task-status-summary.html?product=mathematica

Shorten and expand URLs

http://www.wolfram.com/mathematica/new-in-10/url-manipulation/shorten-and-expand-urls.html