University of Amsterdam
Faculty of Science

MSc Artificial Intelligence
Master Thesis

Detecting and Addressing Change in Machine Learning Data Pipelines

---

by
Bogdan Floris
12140910

July 21, 2020

Number of credits: 48 ECTS
November 2019 - July 2020

Supervisors:
prof. dr. Paul Groth
Jakub Zavrel

Assessor:
prof. dr. Paul Groth
prof. dr. Evangelos Kanoulas

$\zeta\vec{\alpha}$

Zeta Alpha

Abstract

Since the pace of machine learning innovation is ever-increasing, companies and practitioners have to constantly replace the components of their data pipelines in order to keep up with the state of the art. This issue takes a toll on both development time and compute resources, and so it would be desirable if the pipelines could detect that a change (i.e. concept drift), has occurred when a component is replaced in the pipeline and then adapt to that change. To investigate this problem, we focus on a small text classification pipeline where different word embeddings components act as the agent of change. We show that change can be detected fast and accurately when new embeddings representations are introduced in the pipeline, while highlighting two detectors from literature that proved the most reliable. We also found that the two models that we used in our experiments, an LSTM based neural network and a Naive Bayes one, behave quite differently when the embeddings are changed, with the LSTM model being more robust to perturbations in its input vector space than the Naive Bayes model. Finally, we take two approaches to addressing change in embeddings, trying to avoid completely retraining the model in order to save compute cost and time. On one hand, if the detected drift is small enough, we use a fine-tuning approach where the model in question is further trained with a fixed number of batches transformed with the new embeddings. We show that the LSTM model recovers its performance after only seeing 200 batches. On the other hand, if the detected drift is big, we use a mapping approach where we map the new embeddings to the old ones using both a linear mapping and a non-linear one. While the model recovers some of its performance, the method is crippled by some of the assumptions that we have made and by the dataset how we contructed the parallel corpus used to train the mapping. We hope that future work manages to solve this mapping issue and that this thesis paves the way for more research into the topic.

Acknowledgements

These past months working on this thesis have been very challenging and entertaining, and it allowed me to grow as a scientist and engineer. For making that happen, I would first like to thank the whole Zeta Alpha team, but especially Jakub Zavrel, for his guidance and endless stream of interesting ideas. Moreover, I would like to thank Paul Groth for showing me how research should be conducted, for answering all my questions, and for giving all the right advice. I am also grateful to my friends, brother, and Ramona for their patience and support. Finally, I have to thank my parents. For everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The machine learning field has seen tremendous improvement in the past decade, and it is continuing to adapt and advance at an ever growing pace. The AI Index Report 2019 (Perrault et al. (2019)) mentions that the number of AI papers produced each year has increased by more than 9x since 1996, with a lot of them primarily in the fields of natural language processing or computer vision. Every year, new state of the art solutions for all kinds of different research areas are introduced. One illustration of this trend are word embeddings, the driving factor behind the improvements on multiple natural language understanding tasks (Peters et al. (2018a)). A comprehensive overview of the history of word embeddings is presented in Section 2.3.

This trend also brings with it a cost to machine learning applications (Schelter et al. (2018)). Large scale ML pipelines are made out of many different components, and there is a constant need to continuously replace these components with better ones as more state of the art techniques are introduced. These changes come with repercussions on machine learning pipelines. Imagine that we have a pipeline where a pdf text document arrives, and we have different services that in order do the following things: extract the text from the pdf, split the text into chunks (sentences, words), transform the text into representations (embeddings), and at the end of the pipeline we have different models (classification, named entity recognition) that use the word embeddings to make predictions. We will call the service that receives text and transforms it to word embeddings a word embedder. Imagine that the current word embedder uses BERT (Devlin et al. (2018)), one of the state of the art methods in the field, but we wish to change it to a different variant that is perhaps trained on another dataset or has other characteristics. Suddenly, the models inside the pipeline see inputs from different embedding spaces and as such may degrade in performance. As we will see in Chapter 4, the models are affected in different ways depending on which new embedding is plugged in or on what machine learning model is being used. Thus, it would be interesting if the pipeline could detect these changes and adapt to them on its own.

This issue is also of particular interest to Zeta Alpha, a start-up and R&D lab that aims to build a platform for the scientific community using state of the art natural language understanding which researchers and practitioners can use to find relevant papers and organize their knowledge. Chapter 6 describes how the company uses the work discussed in this thesis in their own machine learning data pipeline. To address this issue, there is a two step process that needs to be performed. The first task is to monitor the model when a new update is pushed to the word embedder and detect how the model performs with the new word representations. This task is called anomaly detection (Chandola et al. (2009)) or concept drift detection (Gama et al. (2014)) and it has been studied quite extensively in literature (Chapter 2), resulting in a few concept drift detectors that work quite well for different drift types or for diverse datasets. Once a drift is detected, the second task the system needs to do is address the change. This

will be done according to the drift type detected, with the two methods that we use shown in Chapter 5.

Figure 1.1 shows the framework in which our experiments are performed. It displays how the different components of our pipeline interact with each other. Since the scope described above is too large to be tackled in this thesis, we decided to narrow it by building a simple text classification pipeline, where the agents of change are the different word embeddings components used to transform the text. The pipeline starts when the inputs arrive at it, with the dataset being used described in Chapter 3. The next step in the pipeline will take care of transforming the text data to a format which the models will understand, word embeddings, described in Chapter 2. As previously mentioned, changing between different word embeddings will be the main reason of drifts occurring in the pipeline. Then, the transformed inputs arrive at either one of the two chosen models, the Naive Bayes or the LSTM, which we discuss in Chapter 3. These models will make predictions, that together with the given labels, will be computed into metrics such as accuracy which will be fed to a change detector that is in charge of signalling drifts (chapters 2 and 5). Finally, once a drift is detected, we will try to address that change using two methods depending on the type of drift that is detected (Chapter 5).

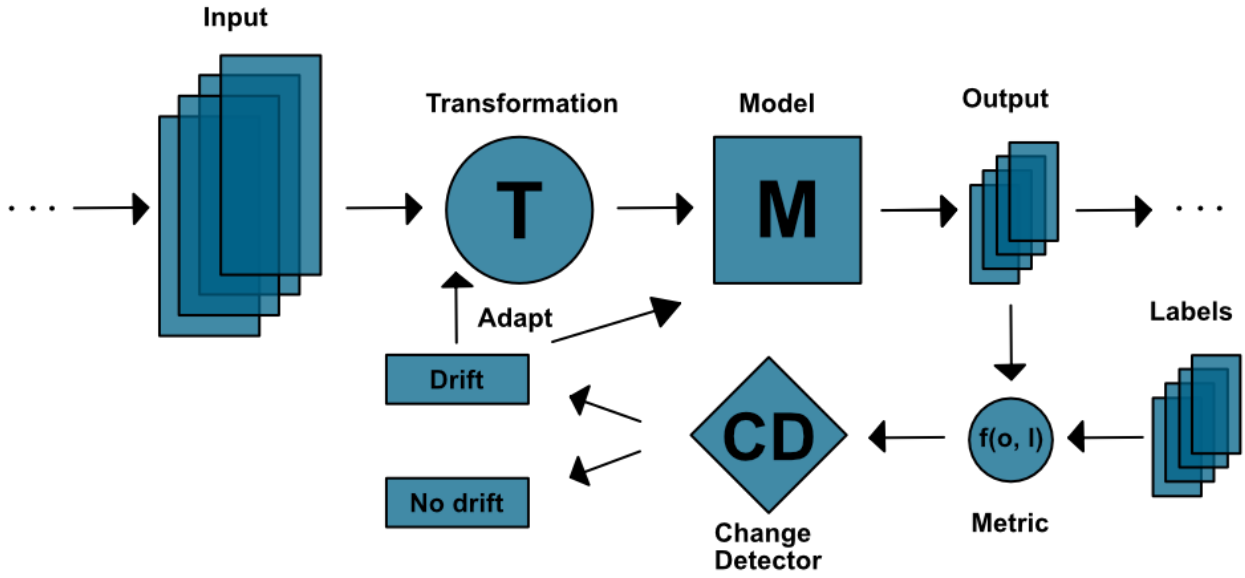


Figure 1.1: General overview of the architecture

1.1 Research Questions

The main goal of this thesis is to develop a general framework in which machine learning models that are part of streaming data pipelines can detect changes in their output distribution and adapt to them accordingly, while also successfully applying this framework on a more specific natural language processing use case. Specifically, we pose the following research questions:

How can we quickly detect that the output distribution of a model has changed given a continuous stream of data, and then report on the degree of change? The scope of this question is to be able to detect that the output of a machine learning model has changed from its initial distribution. This can be done by either checking the output distribution directly or checking the distribution of the metrics computed from the predictions

and the correct labels. We also wish to report on how much the change affects the performance of the model, since this has an impact on which method we need to employ to address the change.

Given that a change was detected, how can we adapt the model efficiently such that the performance is the same as it previously was before the change? Once we have detected that a change impacted the performance of the machine learning model, we also need to take measures to address that. Since every agent of change leads to a different drop in performance, there is no one size fits all method that can address this, and so this question will be split up into two based on how big the drift is. We characterize what small and big changes are later in the thesis.

- *Given a small change (gradual drift or small abrupt drift), can we adapt the model by feeding it with a few examples, compared to retraining it from scratch?*
- *Given a big change (big abrupt drift), can we adapt the model by finding a mapping between the old outputs and the new ones which is less expensive to compute than retraining the model from scratch?*

Chapter 2

Background

In this chapter, we introduce the background needed to support the work done in this thesis. First, we give a brief overview of the framework in which change can occur, a data pipeline, and present the concept of online learning. Then we go into more details about the work that has been done on detecting change in this framework, explaining concept drift. Finally, we present a small overview of the agent of change that acts on the pipeline we consider, word embeddings.

2.1 Data Pipelines and Online Learning

Since the turn of the century, as memory became increasingly cheaper and the internet more and more widespread throughout the world, companies have started to collect enormous amounts of data. According to Yaqoob et al. (2016), the amount of data doubles every two years, with the trend showing no signs of stopping. This phenomenon has given birth to the field of big data, which aims to develop techniques to process and analyze data that is too large or complex for traditional data analysis tools (Wikipedia contributors (2020a)). Big data has been forcing companies to slowly move away from traditional machine learning algorithms because these work in an offline setting. In offline machine learning, all data is available from the start and can be fed as batches to train the model, with each sample being used more than once potentially. This method is usually infeasible in the context of big data, so Fayyad et al. (1996) have proposed a few requirements for these big data machine learning pipelines:

- The training of a model should be done continuously and only on blocks of data or separate samples.
- Each sample should only be used once to train the model.
- We should assume that the samples are not stored after they have been seen by the model.

The third requirement can be relaxed in practice depending on how much storage is available, and most large scale data pipelines usually store at least a recent window of the data (Bifet and Gavaldà (2007)).

Online learning is the type of data analysis usually deployed in streaming data pipelines and is defined as a learner which attempts to solve an online decision task by fitting a model to a sequence of data that arrives one at a time (Hoi et al. (2018)). This is in contrast to offline (batch) machine learning, which requires that all data be present when training the model. The main advantage of online learning is that it is highly scalable in terms of both computing power and storage Liberty et al. (2020), which is suitable for big data, but it is also not as capable as batch learning since it only sees the samples once and not in an arbitrary order. According to Hoi et al. (2018), there are three types of online learning:

- Supervised online learning where feedback is immediately available.
- Online learning with limited feedback (e.g. only a few samples also have labels).
- Unsupervised online learning where there is no feedback.

The work done in this thesis will mainly focus on supervised online learning, while briefly touching on how unsupervised learning could be handled.

2.2 Concept drift

This section will present a brief introduction to concept drift in machine learning, mention how concept drift can act on data and on the models which were trained on the data, then touch on some work that has gone into concept drift detection algorithms.

2.2.1 Definition

Online data pipelines should in theory operate endlessly after being deployed, but the nature of data in general makes the environment in which they operate dynamic. One example would be the change in the distribution of the data samples that arrive at the pipeline. This phenomenon is known in theory as concept drift (Gama et al. (2014)). Take for example a set of data X with a target variable y, where we are trying to learn a function f, such that $y = f(X)$. Prediction models usually require and assume stationarity in the data (Wang and Abraham (2015a)), so if we make this assumption we can fit f once and assume that it works for all subsequent data coming to the pipeline. In practice, however, it might happen that the distribution of X changes in time, and so the relationship between X and y also does, which makes our estimate of f outdated.

We can formally define the problem addressed in this thesis as a concept drift issue. Let W be the initial, unprocessed dataset (which will be described in Section 3.1.1) with labels $\mathbb{C}$ that denote classes, $\mathbb{T}$ a set of transformation functions (for example a function that extracts features out of the initial dataset), and X the transformed input $\forall i : x_i = t(w_i)$, where $t \in \mathbb{T}$. We can then use Bayesian Decision Theory (Duda et al. (2001)) to state the classification task. The probability $P(c_k), c_k \in \mathbb{C}$ is known as the *a priori* and reflects the probability that a sample belongs to class c_k. The probability $P(X|c_k)$ is known as the *class conditional density function* and reflects the probability density of X when class c_k is known. When can use Bayes' Rule (Stone (2013)) to solve for the *posterior*:

$$P(c_k|X) = \frac{P(X|c_k)P(c_k)}{P(X)} \tag{2.1}$$

Concept drift occurs if for two time points t_1, t_2, where $t_2 > t_1$, $\exists x \in X : P_{t_1}(x|c_k) \neq P_{t_2}(x|c_k)$. For our purposes, we will assume that W remains stationary and the distribution shift happens because of a change in the transformation functions $\mathbb{T}$, induced by the pipeline switching from a transformation function to another. Naturally, W could also encounter an inherent distribution shift, not causes by the transformation functions, but by the data itself. The dataset that we will be using, described in Chapter 3, is actually static, but to make sure that change detection experiments cover all the concept drift types, we will be adding random noise to the data to simulate this inherent distribution shift.

2.2.2 Concept drift types

Concept drift is the shift in the distribution of the dataset as samples come to the pipeline, but of equal importance to both detection and overcoming change is the type of concept drift. Gama et al. (2014) describe five types of drift, but for our intents and purposes, we tackle the following three types:

- Big abrupt concept drift (Figure 2.1), which is signalled by a big instant change in the distribution that results in the model's accuracy degrading heavily. We define a big change as a drop in the model's accuracy of more than 0.2.
- Small abrupt concept drift (Figure 2.2), which is signalled by a small instant change in the distribution that results in the model's accuracy degrading just slightly. We define a small change as a drop in the model's accuracy of less than 0.2.
- Gradual concept drift (Figure 2.3), which is is signalled by small incremental changes in the distribution that in time have the same effect on the model as a big abrupt change but take to occur.

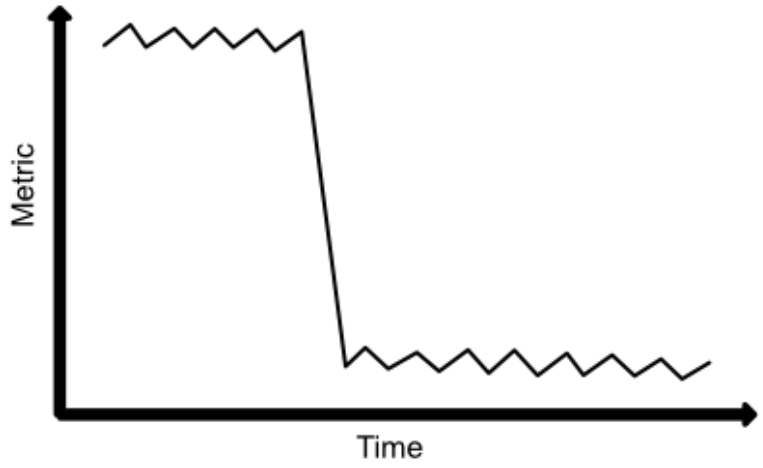


Figure 2.1: Big abrupt concept drift

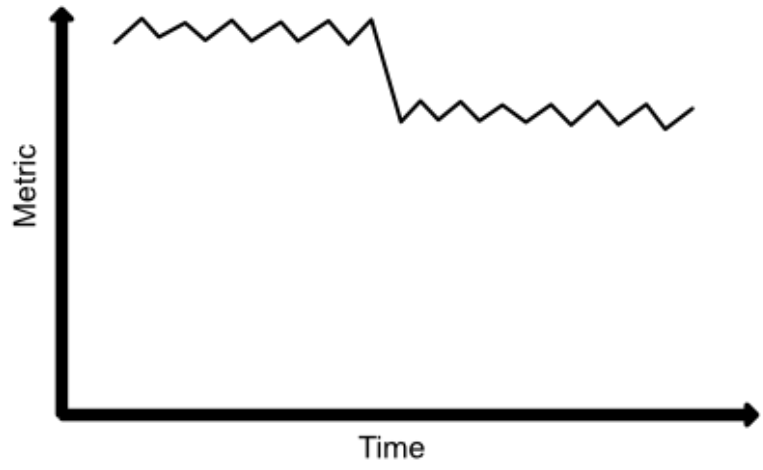


Figure 2.2: Small abrupt concept drift

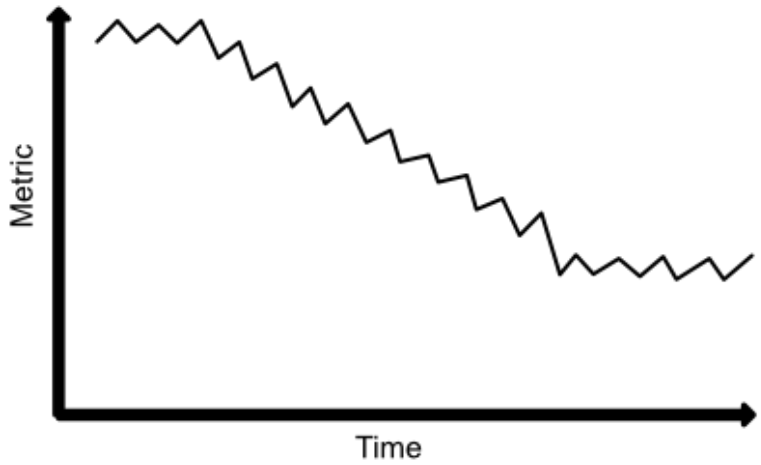


Figure 2.3: Gradual concept drift

The type of concept drift encountered has an impact of both detecting and addressing change. As emphasized in the next section, there are quite a few algorithms that are used in practice to detect concept drift, and most of them are better at detecting a specific type of drift, so there is no free lunch. As shown in the experiments in the next chapter, gradual drift is a little harder to detect since we do not want to make the change detector too sensitive because that could produce false positives. When in comes to addressing change, there are again a few options from which to choose. For example, a fine tuning approach might work for a small abrupt drift, but doing it for a big abrupt drift is akin to retraining the model from scratch, something we would like to avoid.

2.2.3 Concept drift detection algorithms

While there has been ample research involved in detecting anomalies in data which is at rest (Hodge and Austin (2004) provides a nice overview of the methodologies employed in anomaly detection and Zimek and Filzmoser (2018) comes with a data mining perspective on outlier detection), machine learning pipelines are anything but static. Data is continuously pushed through the system, so the metrics computed from the models' outputs are adjusted endlessly. Moreover, this uninterrupted flow in the system makes storing all these metrics infeasible, so the pipeline must be able to make a decision on whether or not a concept drift occurred based only on the most recent samples.

Kifer et al. (2004) wrote one of the pioneering papers on detecting change in data streams and came up with a novel solution that keeps two windows for the incoming stream and compares the two using a new distance metric that proves to be a good indicator for the degree of change, as it was specifically designed with this in mind. Their method could be improved by using a different method for saving samples, like the space saving algorithm for detecting heavy hitters, as described in Mitzenmacher et al. (2011). Rettig et al. (2019) uses more simple distance metrics like relative entropy and Pearson correlation, but the approach is tailored towards big data streams. ADWIN (Bifet and Gavaldà (2007)) improves on Kifer et al. (2004) by making the size of the window adaptable, resizing it according to the rate of change observed from the data in the window itself. Two methods that are especially suited for detecting drift in classification tasks using metrics are DDM (Gama et al. (2004)) and EDDM (Baena-García et al. (2006)). Both of them are based on the estimated distribution of the distances between classification errors, but DDM is more suitable for abrupt drift, while EDDM is better for gradual drift. A more state of the art paper, Yu et al. (2019), presents a novel framework for

hierarchical hypothesis testing, a concept drift detector named Hierarchical Linear Four Rates, which achieves great results on type I and II errors.

2.2.4 Concept drift adaptation

Concept drift adaptation is a newer topic of interest in the scientific community, but it is gaining more and more traction as models are becoming increasingly costly to retrain. Yu et al. (2019) mentions that there are two different approaches to address concept drifts in streaming data. *Continuing adaptation* does not include a concept drift detector, but assumes that the environment will change and either keeps updating the model parameters incrementally (Bifet and Gavaldà (2007)), or learns an ensemble of models on different windows in the stream (Elwell and Polikar (2011)). *Adaptation with drift detection* actually employs a detector, which signals that a change has occurred and then makes a decision on which adaption technique to use based on the degree of change (Wang and Abraham (2015b)). The work in this thesis will focus on the second approach, by only using adaptation techniques whenever a drift is signalled.

2.3 Word embeddings

We have mentioned in Section 2.2 that there is a set of transformation functions $\mathbb{T}$ that act as the agent of change on the initial dataset. The transformation agents that will be the main study of this thesis are word embeddings, which have been main driving force behind the improvements in natural language understanding in the past decade (Li and Yang (2017)). There were two main reasons that necessitated the development of word embeddings to represent text data:

- Solutions before the introduction of word embeddings were based on bag of words, where a word was represented by the number of times it appeared in the text. The bag of words method had the clear disadvantage that it produced huge and sparse vectors, which were unsuitable for models. Word embeddings, on the other hand, produce short and dense vectors, which improves both the computational complexity and the understanding of the text (Li and Yang (2017); Mikolov et al. (2013a)).

- Another issue with previous approaches was that the context of a word was not embedded into its representation in the vector space. Take for example these two sentences: *I am writing in my notebook* and *I am writing in my journal*. *Notebook* and *journal* have similar context and this should ideally be represented in their respective vectors. Their relationship is represented in theory by the cosine similarity (Mikolov et al. (2013a)):

$$\text{similarity}(w_1, w_2) = \cos(\theta) = \frac{w_1 \dot{w}_2}{\|w_1\|\|w_2\|} \tag{2.2}$$

  , where θ is the angle between the vectors of w_1 and w_2. The cosine similarity is not taken into account with the previous approaches, but word embeddings are optimized to improve it.

The breakthrough paper that introduced word embeddings to both the academic world and the industry was word2vec (Mikolov et al. (2013a)), which computes pre-trained word embeddings using a skip-gram model. Since then, the field has brought increasingly more improvements to non-contextualized word embeddings, with some examples being GloVe (Pennington et al. (2014)) and FastText (Joulin et al. (2016)).

Recently, since the introduction of BERT (Devlin et al. (2018)) and ELMo (Peters et al. (2018b)), the field has shifted to contextualized word embeddings, which have brought with

them great improvements to the state of art on a multitude of tasks (Devlin et al. (2018)). These embeddings are trained using a bi-directional transformer model by conditioning on both the left and right context of a word, thus creating better representations. Moreover, besides these different fundamental approaches to computing embeddings, each one of these techniques has dozens of pre-trained variations on different datasets or of different sizes (Wolf et al. (2019)). For example, BERT has seen variations that make it faster and cheaper computationally, like DistilBERT (Sanh et al. (2019)), or variations that make it work better for scientific text, like SciBERT (Beltagy et al. (2019)).

2.4 Conclusion

In conclusion, this chapter gives a brief introduction into the key concepts on which the work done in this thesis builds upon. The research will be done not in the traditional machine learning framework, that of offline learning, but in the more applicable situation of online learning, using a streaming data pipeline. The agent of change that will act on the pipeline will be a switch between different ways of computing word embeddings for text data, and we will be using concept drift detection algorithms to signal a change.

Chapter 3

Framework

In this chapter, we will focus on establishing the framework in which the experiments on detecting and addressing change will be conducted. Namely, we will discuss the dataset that was chosen for these experiments and how this dataset was fed into a prototype of a data pipeline. Moreover, we present what models were chosen and how they were trained.

3.1 Dataset

In this section, we will introduce the Web of Science dataset, make a case for its selection and then show how it was repurposed as a stream of data in our pipeline.

3.1.1 The Web of Science

Given the goal Zeta Alpha outlined in Chapter 1, their work involves the processing of a huge number of scientific papers which can then be explored by their users. This is why we have decided that for the purpose of the research done in this thesis, the best dataset to be used is the Web of Science (Kowsari et al. (2017)). Scientific papers are also an example of a domain that presents drift over time, since the number of papers in one field can increase or decrease based on the popularity of the field.

The Web of Science is a collection of abstracts that were extracted from scientific papers spanning many different fields (Kowsari et al. (2017)). The abstracts are labelled with their discipline according to a manually produced taxonomy (e.g. computer science, psychology). The dataset was designed as a hierarchical classification task, in which the purpose is to determine both the broader domain and a sub-domain of the paper (e.g. computer graphics is a sub-domain of computer science). Since our purpose in this thesis is not to solve the task, but to detect and address change, we will simplify the goal to a multi-class classification and only try to determine the sub-domain. The dataset is organized in three versions (table 3.1), each version having increasingly more samples, but also labels (Kowsari et al. (2017)), thus making the task more difficult.

Version	Samples	Subdomains	Domains
1	5,736	11	3
2	11,967	35	7
3	46,985	134	7

Table 3.1: Web of Science versions

To reiterate, the purpose of this thesis is not to achieve state of the art accuracy on this dataset, so we only need the Web of Science as a task for our experiments. Thus, we have

decided to pick the first version of the dataset to save both time and compute power. This choice should have no impact on the overall results of paper, since training on either of the other two versions would just give us a baseline model with different performance compared to the first version.

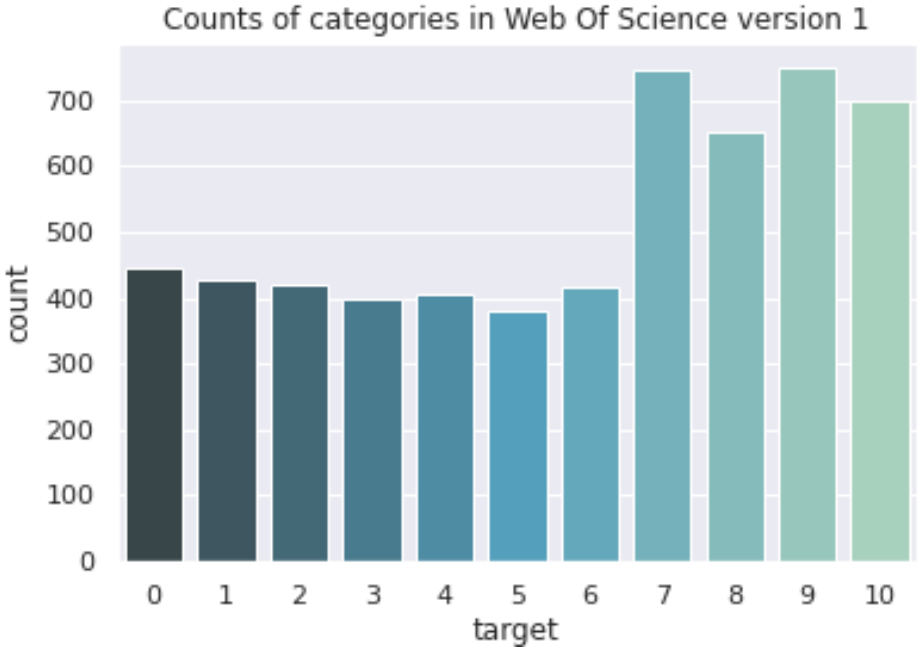


Figure 3.1: Web of Science counts for each label

We also visualize the distribution of labels in the first version of the Web of Science to check that all classes are represented properly. Guo et al. (2008) mentions that class imbalance happens when one class outnumbers the others 10 to 1, but we can see from Figure 3.1 that we do not have this problem, and so we can use the usual classification methods and metrics.

3.1.2 Data pipeline

Having chosen an appropriate dataset for our experiments, we now shift our focus to describing the data pipeline that will receive at its point of entry samples from the dataset. This pipeline will mostly follow the diagram shown in Figure 1.1.

The first requirement is to pick a suitable framework in which to design the pipeline. While frameworks like `Kafka` (Kreps (2011)) or `Flink` (Carbone et al. (2015)) are excellent for an industry setting where high performance and scalability are required, we chose a simpler implementation environment for our experiments. Thus, we have chosen to use `Scikit-Multiflow` (Montiel et al. (2018)), a data streaming library that builds on top of `Scikit-Learn` (Pedregosa et al. (2012)) and aims to provide the same algorithms as its parent library but adjusted to online learning. The library is a good choice since it allows us to keep the same challenges that come with using machine learning algorithms in data pipelines, thus making our experiments more robust, but also speeds up the implementation time since its API is compatible with `Scikit-Learn`, the most used machine learning library.

The first step in the pipeline is to pre-process the data, since text tends to contain information that is irrelevant to classification (e.g. tags, spaces, punctuation, miss-spellings) (Kowsari et al. (2019)). After cleaning the data, we can then feed it to one of the three chosen transformer (Wolf et al. (2019)) models (see Section 2.3) and their corresponding tokenizers to extract the word embeddings. To keep everything consistent throughout all experiments, we have chosen

to keep the embedding dimension to 768, to cut of all sequences of tokens to 512 if they are longer, to pad everything with zeros if the sequences of tokens are shorter than 512, and to keep the last layer of the transformer models (Vaswani et al. (2017)) as the inputs to the models. This results in a final dataset that is held in a three-dimensional tensor of shape $(N, 512, 768)$, where N is the number of samples.

The next step of the pipeline is to input these samples to the model (Section 3.2) to get a predictions that will then be used together with the labels to compute metrics which will be fed to the change detector (Chapter 4). The detector can then make a decision on whether a change has occurred and alert the system that the change must be addressed (Chapter 5).

3.2 Models

This section covers the next step in our data pipeline, the models. Two of them have been chosen for our experiments: Naive Bayes, which acts as a baseline, and an LSTM that will be the main object of our experiments. The LSTM was chosen because it has been shown in literature that it performs well at text classification (Sherstinsky (2020)), while the Naive Bayes model was chosen as a baseline against which we can compare the LSTM model. The comparison between the two models will be developed in this section and in the subsequent chapters and will serve to highlight how a probabilistic model and a neural network model behave under change. The next two subsections will present a more in depth look at the models (explanation on the methods, training process, etc.), but the important results averaged over 5 runs are shown in table 3.2.

	Loss	Train Accuracy	Test Accuracy	Precision	Recall	Macro F1
Naive Bayes	N/A	0.612	0.554	0.575	0.571	0.567
LSTM	0.087	0.955	0.898	0.894	0.895	0.892

Table 3.2: Metrics for the Naive Bayes and LSTM models averages over 5 runs (the first 2 columns are on the train set, while the rest are on the test set).

An important part of this thesis is to avoid a full retrain of the models, thus improving on the time and computational cost needed to adapt the model to the detected changes. Besides the metrics for the models, we also highlight the time needed to train the two models. The Naive Bayes is trained quickly in just 10.4 seconds, while the neural network based LSTM model is trained in 14 minutes and 12 seconds on a Tesla K80 GPU and 4 hours and 32 minutes on a CPU. As we will see in Chapter 5, the Naive Bayes model does not require an adaptation method since its training time is so slow. The LSTM, however, would benefit substantially from a quicker retraining model, and the mentioned chapter discusses that in detail.

3.2.1 Naive Bayes

The Naive Bayes classifier is a supervised machine learning algorithm that is quite popular in natural language processing (Rish (2001)) and is usually used as a baseline for other methods in text classification (Wang and Manning (2012)). The algorithm is based on Bayes' Rule (Stone (2013)) and the following equation is used to classify a sample:

$$P(y|x_1, ..., x_k) = \frac{P(x_1, ..., x_k|y)P(y)}{P(x_1, ..., x_k)} \tag{3.1}$$

$$= \frac{P(y)\prod_{i=1}^{k} P(x_i|y)}{P(x_1, ..., x_k)} \tag{3.2}$$

, where $x_{1..k}$ are features of a sample and y the label. Since the marginal probability remains equal throughout the computations, we can exclude it. Moreover, the Naive Bayes makes an important assumption that the features of a data point are conditionally independent given the label. Even though this assumption is not correct more often than not, it turns out that Naive Bayes usually works in practice on simpler classification tasks (Rish (2001)). In fact, a good predictor for the performance of the classifier is the amount of information lost due to the independence assumption, as the experiments in Rish (2001) show.

For our experiments, we use the implementation of Naive Bayes from the `Scikit-Learn` package, which also supports a `partial_fit` method, suitable for online learning, thus making the implementation compatible with our data pipeline. We have mentioned in Section 3.2 that a sample representation in our dataset is a 2D matrix with shape $(512, 768)$, but Naive Bayes only supports samples that are in one dimension. This leaves us with two options: we can either aggregate (taking the average or the maximum) over one of the dimensions, or we can transform the matrix into one big vector, by flattening it. We have found by trying all these methods that flattening the matrix is both more expensive and produces worse results than aggregating. The best performing method was taking the maximum over the first dimension in the matrix (i.e. max-pooling over all the words in the document), thus producing a sample of size 768 that contains all the information of a scientific paper abstract.

These samples were then fed to the data pipeline in batches of 32 for 10 epochs in order to train the Naive Bayes classifier, while keeping track of the training set accuracy. At the end of each epoch, we also computed four different metrics (accuracy, precision, recall, and macro F1) on a holdout testing set to make sure that the model was not influenced by a class imbalance problem. The results are shown in table 3.2 and in Figures 3.2 and 3.3. For these figures and all subsequent figures that present results, the time axis (x-axis) represents the number of batches that have been seen by the model at that point.

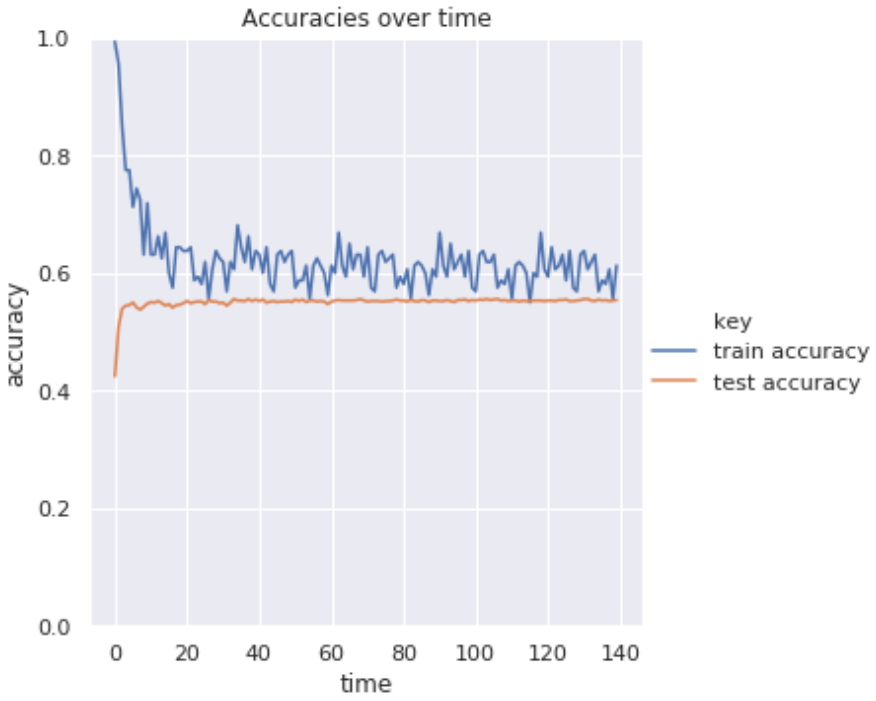


Figure 3.2: Naive Bayes accuracy over time using BERT embeddings

As we can see for the results, the Naive Bayes method performs decently enough for a multi-class classification problem with 11 classes (the 0.55 test set accuracy is substantially better than the accuracy of approximately 0.1 that should be achieved by a random guesser).

Moreover, we can see from Figure 3.3 that class imbalance is not a issue here since all the metrics that we have chosen have almost identical curves. Furthermore, the model converges quite fast and the curve shows no oscillations on the test set (Figure 3.2), so we can conclude that it is pretty robust on the dataset that it was trained on. However, as we will see in the next chapter (4), as soon as the features deviate just a little bit, the model performs much worse, so it is not robust to changes in data distribution.

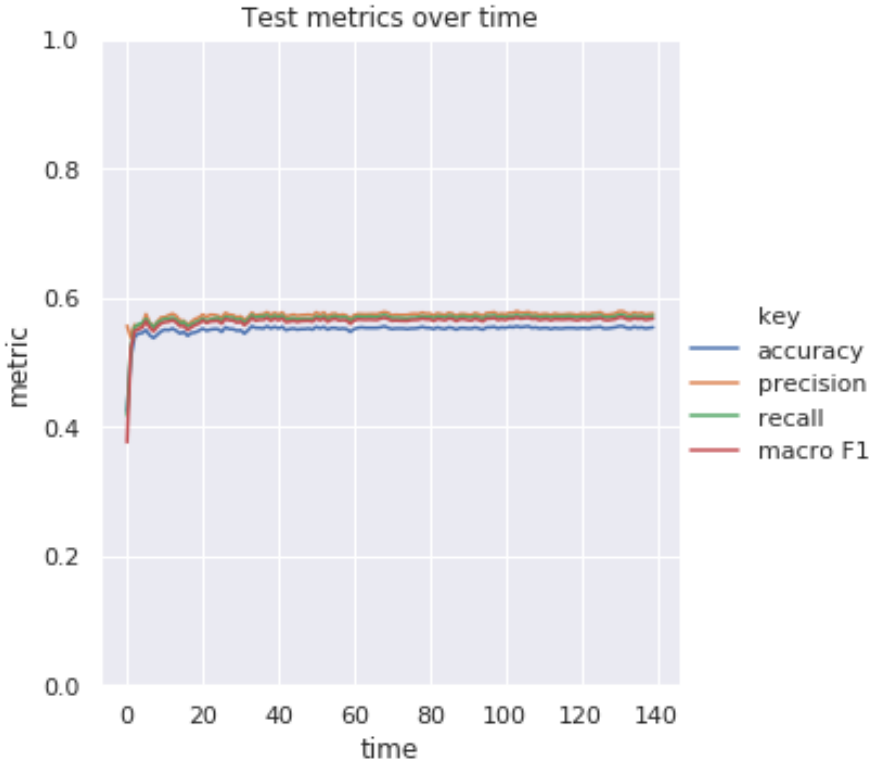


Figure 3.3: Naive Bayes metrics over time for the test set

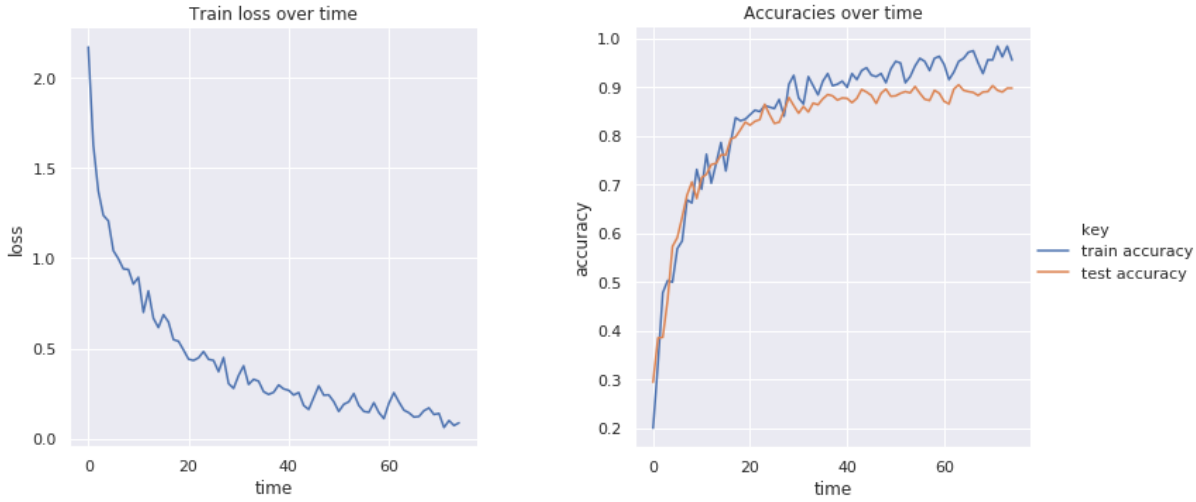
3.2.2 LSTM

Long Short Term Memory (LSTM) (Hochreiter and Schmidhuber (1997)) network is a type of artificial neural network that has been created to address a major problem with Recurrent Neural Networks (RNNs), that of the vanishing gradient. RNNs were designed to model continuity, meaning that the network could learn from past events. This is especially useful in the context of NLP, where we would like to model problems based on sequences of words. The vanishing gradient problem occurs when there are long-distance relationships between the elements of the sequence. By continuously performing derivations, the gradient will eventually approach 0 and thus we cannot perform a meaningful update. LSTMs are designed specifically to solve this problem, by making small modifications to the information by multiplications and additions through a mechanism called a cell state that allows for information to easily flow through the network (Sherstinsky (2020)). Cells make use of three gates that determine what information gets passed along to the next cell (Olah (2015)):

- Input gate, which decides how the input will affect the cell state
- Forget gate, which decides what information from the previous cell state is kept and what is forgotten
- Output gate, which determines the output hidden state that gets sent to the next cell

All these properties make LSTMs a very suitable choice as our model with which we can solve the text classification task. Neural network models are already online learning methods by their construction since they are using gradient descent on batches of an arbitrary number of samples. Thus, we can simply incorporate an LSTM into our data pipeline. The architecture will be kept simple since the task is not overly complex and we would like to save computation time. The network has two LSTM layers with a hidden dimension of 256 and a final fully connected layer that is used for classification. Unlike the Naive Bayes model, we had no issues feeding the 2D samples to the LSTM layers since the model is specifically designed to handle sequences. However, we encountered a similar issue before feeding the output of the LSTM layers to the fully connected layer for classification. Zhou et al. (2016) mentions that absolute value max pooling over the hidden dimension of the LSTM output will transform the vector to one dimension while preserving the most important information.

As we have done with the Naive Bayes model, we fed the samples to the data pipeline with an LSTM model in batches of 32 for 10 epochs, while keeping track of the train set loss and accuracy. We again computed the four metrics mentioned in the previous subsection at the end of each epoch on the test holdout test. The results are shown in table 3.2 and Figures 3.4 and 3.5.



(a) Loss over time

(b) Accuracy over time

Figure 3.4: LSTM loss and accuracy over time using BERT embeddings

As assumed, the results show us that the LSTM is a much better model for text classification since it does not assume independence between the features, but actually tries to learn that dependence. We obtain a test set accuracy of 0.898 averaged over 5 runs and while the model does not converge quite as fast as the Naive Bayes, it still manages to arrive at a good performance quite quickly, as can be seen from Figure 3.4. We also obtain the same results about the class imbalance problem as the Naive Bayes, with all four metric curves looking almost identical. The LSTM is not only very robust on the dataset that it was trained on, but as we will show in Chapter 5, it can also handle perturbations in its input distribution, with its performance declining much slower than the Naive Bayes model.

3.3 Conclusion

This chapter presented most of the preliminary work that went into this thesis before actually answering the research questions outlined in the introduction (Chapter 1). It presents the Web of Science dataset and how it was transformed to a data stream, and then goes on to

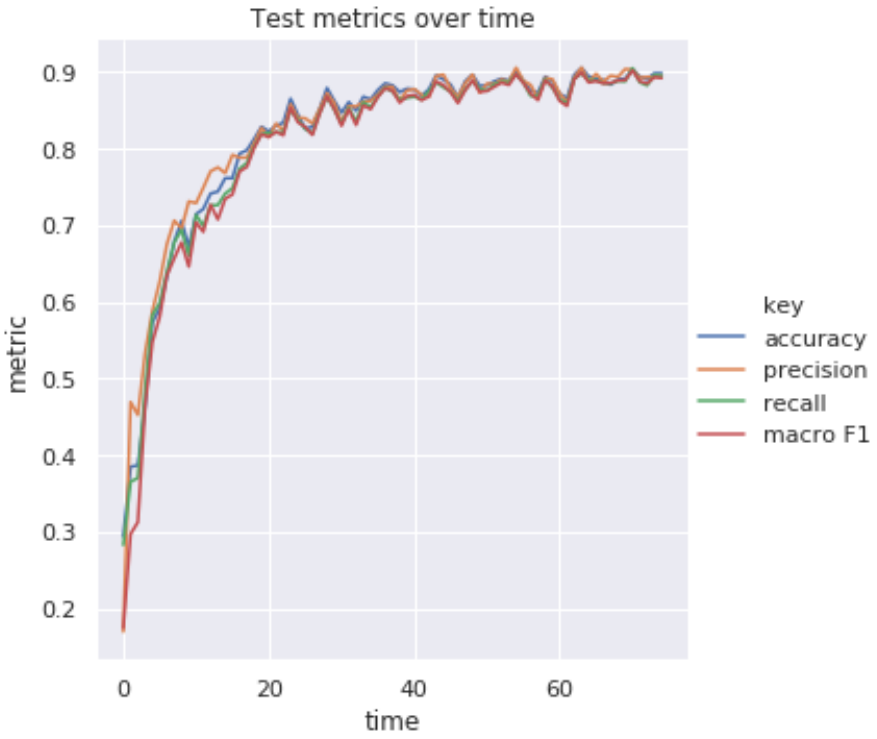


Figure 3.5: LSTM metrics over time for the test set

show how the two models that we have chosen (Naive Bayes and LSTM) were adjusted to our framework and trained, while showing their performance on the Web of Science classification task. Although this was not the purpose of this thesis, we have shown how the two models are quite different in terms of both robustness and performance, with the LSTM displaying great accuracy without any hyper-parameter tuning.

Chapter 4

Detecting change

Now that we have explained the framework in which the experiments in this thesis will be conduced, we can proceed to answer the first research question outlined in Chapter 1: *How can we quickly detect that the output distribution of a model has changed given a continuous stream of data, and then report on the degree of change?*.

This chapter, and every subsequent chapter after this, will be organized as follows. We will first explain how the experiments were setup and present the details in order to better understand the result, which will be shown after. Finally, we will interpret the results in a discussion.

4.1 Experimental Setup

We have outlined a few approaches to drift detection in Section 2.2.3, but there are too many algorithms to try all of them in practice, so we have decided to stick with a few that have proven themselves through time, or in the literature. Pesaranghader et al. (2017) mention that DDM, EDDM, and ADWIN have been frequently considered as benchmarks, while Page-Hinkley (Page (1954)) was a pioneer method in the field. We have experimented with all four before deciding on a suitable change detector. However, Page-Hinkley did not perform on par with the other three so we dropped it from our experiments.. Both DDM and EDDM are statistical analysis methods which attempt to detect drift based on statistical parameters such as the mean and standard deviation, so in theory they sacrifice detection accuracy for minimal storage and speed. ADWIN, however, is a window based method which needs to store two windows of the data stream, one older and another newer. If there is a significant difference between these two windows, than ADWIN signals a drift. This method is supposed to be more accurate, but it also needs to store two windows of data and compute a difference metric on those two windows, which makes it more computationally expensive. In practice, however, we have found that DDM and ADWIN perform on par on abrupt drift, while EDDM outshines the rest on gradual drift, which makes sense since it was specifically designed for this. Thus, we have decided to use DDM for the abrupt drift experiments, and EDDM for the gradual drift ones.

The experiments will be run as follows given either the Naive Bayes or the LSTM model trained in the previous chapter on BERT embeddings:

- We first push our whole dataset through the pipeline. The dataset will be transformed using the same embeddings as those on which the model was trained (i.e. BERT).
- We evaluate the predictions made by the model against the labels, producing an accuracy. This accuracy will then be the input to the change detector for each batch that runs

through the pipeline. Thus, the change detector can build an accurate mean and standard deviation for the dataset.

- After the whole dataset is finished, we are now ready to introduce change in the pipeline, by re-running the dataset from the beginning, but now transformed using either one of SciBERT or DistilBERT, or by adding random noise to the BERT embeddings. Again, the predictions will be evaluated against the labels, and the accuracies pushed to the change detector.

Note that the detector can signal a warning, meaning that the pipeline is close to deviating from the mean, or an actual drift. The detector will be queried for either a warning or a drift after each batch, both when running the stream the first time (to test that the detector is robust and does not signal false positives) and the second time (to detect changes).

The three agents of change were picked for the following reasons:

- SciBERT (Beltagy et al. (2019)) provides a completely different representation for the documents, since it was trained on scientific text. Thus, there should be no correlation between these embeddings and BERT, and so we can introduce SciBERT as an agent of big abrupt drift, a type of change that should be felt instantly by the pipeline and will result in huge degradation to the accuracy of the model.

- DistilBERT (Sanh et al. (2019)), on the other hand, was trained to be a faster and smaller version BERT, while retaining most of its language understanding capabilities. This should mean that models trained on BERT should perform on par when new samples arrive that have been transformed using DistilBERT. Thus, we can introduce this embeddings as an agent of small abrupt drift, which should also be felt instantly by the pipeline, but will result in minimal or no degradation in the performance.

- The last agent that we need to introduce needs to be able to simulate a gradual drift. For these experiments, we have to adjust our framework a bit by only running the stream once, but from a specific batch, we start to add random gaussian noise in an increasing fashion as follows: we keep the mean of the normal distribution to 0, and we pick a maximum standard deviation. Then, we make an evenly spaced interval of standard deviations between zero and the maximum. We then sample the random noise from the distribution, and thus every batch gets increasingly more noise added to it. Note that we will be running experiments with three maximum standard deviations: 1.0, 2.0, and 3.0.

The last point that we need to mention before showing the actual results is that we previously made the assumption that we possess labels for each abstract that passes through the pipeline. We use the labels to compute the metrics that are presented as input to the change detector. In real world data pipelines, this is usually not a valid assumption to make since the incoming data to the pipeline is not labelled most of time. This means that we also need to find a way in which to detect change when we have a trained model, but the data arriving at it does not have labels. One way to do this is to assume that the model in the current pipeline produces perfect predictions and use them as labels to produce metrics against the pipeline with a newly introduced agent of change. As an example, which we will see in the results section, we assume that the BERT embeddings produce perfect predictions (with accuracies of 1.0), and use those labels to compute metrics with the SciBERT predictions. One issue with this approach comes from the way change detectors work. Using the accuracies that are given as input, they compute a mean and standard deviation, and when a new accuracy is too far away from the mean given the standard deviation, it signals a drift. If we only input accuracies of 1.0, the change detector will compute a mean of 1.0 and a standard deviation of 0. Thus, any number other than 1.0

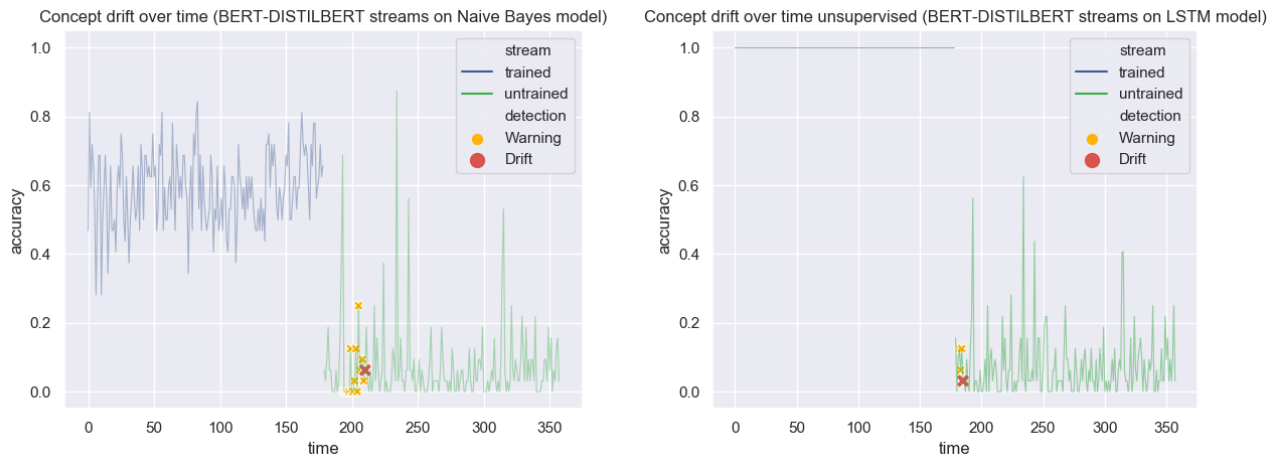
that arrives at the change detector will immediately trigger a drift, even if it is something that is very close to 1.0, such as 0.995. We have found from our experiments that using random uniform numbers between 0.9 and 1.0 as input to the change detector to build its statistics and tuning down its sensitivity (i.e. increasing the allowed deviation from the mean) solves the issue described above. It is also important to note that this approach only works in the setting outlined in this thesis, since in a real world production pipeline we do not know when we should stop feeding random numbers to the change detector and start computing the metrics using the previously stored predictions. This only works when we have a predetermined dataset, an approach that we use in practice in the Zeta Alpha pipeline, outlined in Chapter 6.

4.2 Results

Now that we have established how the experiments will be run, we can go on to present the results for both models. We will start of with Naive Bayes and then move on to the LSTM. Both of them will have the supervised and unsupervised results shown side by side for better clarity, and the Figures will have a consistent format: the blue stream will be the one on which the model was trained (i.e. BERT embeddings), while the green stream will be the stream impacted by change (i.e. SciBERT or DistilBERT). Moreover, warnings being the output of the change detector are visualized by a yellow cross, and the drift by a red cross. Note that for the random noise change detection experiments, there is only one stream, the blue one, which is slowly being impacted by change.

4.2.1 Naive Bayes model

We first change embeddings from BERT to DistilBERT (i.e. simulating small abrupt drift) for both the supervised (left) and unsupervised (right) case and showcase the results in Figures 4.1a and 4.1b. DistilBERT embeddings are supposed to preserve the embeddings space of BERT, but it seems that the Naive Bayes model is fitted very closely to the dataset that it was trained on, and it is not robust to small changes in its inputs. Except for a few outliers, we can see that the performance of the model makes a large drop. The change detector however did its job quite well and outputted a few warnings very fast before finally triggering a drift. Moreover, we can see that the places where the outliers are can be found in both the supervised and the unsupervised variants and thus it seems that our method for computing metrics is working so far. Note however that the change is triggered much faster than in the supervised case, mostly because the supervised case has an average accuracy of about 0.6, while in the unsupervised case we are using as input to the change detector random numbers between 0.9 and 1.0.

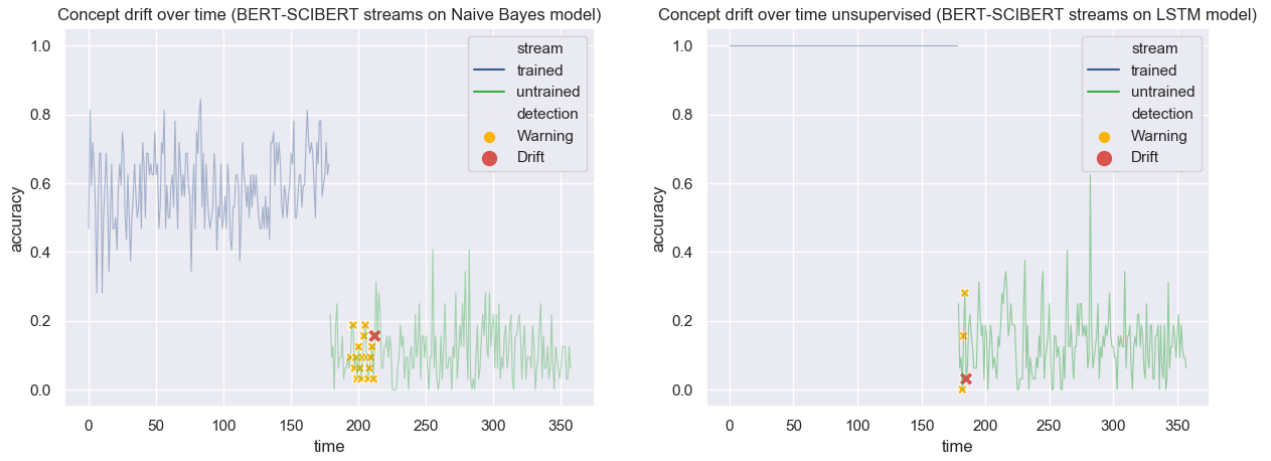


(a) Supervised change detection

(b) Unsupervised change detection

Figure 4.1: Detecting change using different embeddings (BERT-DISTILBERT) in the Naive Bayes model

Figures 4.2a and 4.2b present the results for the experiments when we are changing from BERT to SciBERT (i.e. simulating big abrupt drift). Given that the previous Figures also showed a big change, it is not surprising that these results showcase the same phenomenon. Again, performance degrades instantly, but in this case, we do not even have the outliers in the green stream. This is expected, however, since SciBERT is a completely different embedding from BERT and so the performance should degrade. The change detector also outputs the change quite fast, more so in the unsupervised case, for the reasons explained previously.



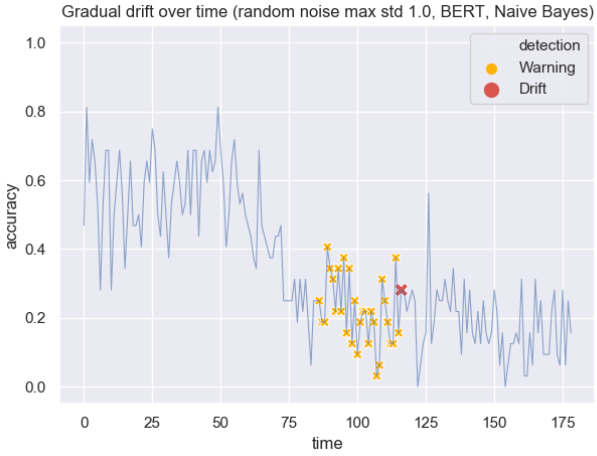
(a) Supervised change detection

(b) Unsupervised change detection

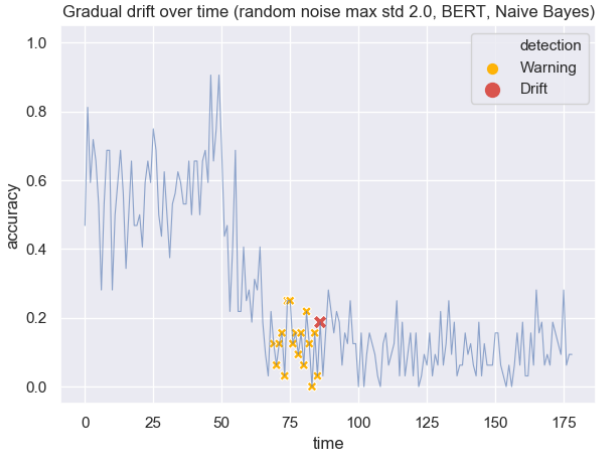
Figure 4.2: Detecting change using different embeddings (BERT-SCIBERT) in the Naive Bayes model

Now we can move on the third agent of change, which is adding random gaussian noise to the BERT embeddings to simulate gradual drift. As explained in the previous subsection, after a few dozen batches, we start adding this noise in increasing fashion by sampling from a normal distribution with mean zero and a standard deviation between 0 and a predetermined maximum. We run this experiments for three maximum standard deviations: 1.0 (Figure 4.3a), 2.0 (Figure 4.3b), and 3.0 (Figure 4.3c). Again, we start to notice the same pattern for the Naive Bayes model, that even slightly small changes to the input distribution results

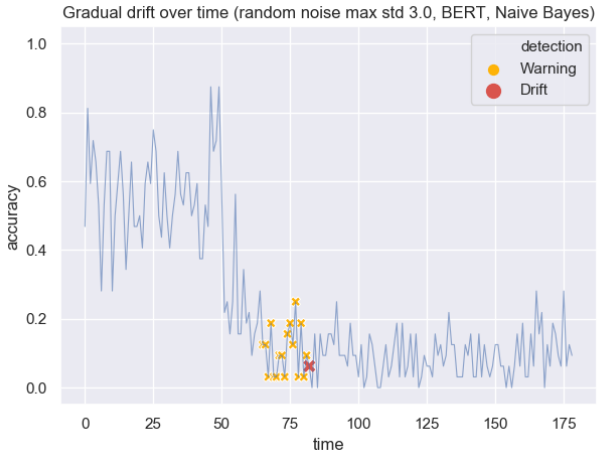
in significant drops in its performance. For the maximum standard deviations of 2.0 and 3.0, the drop to the performance of a random guesser is almost instant, with the change detector reacting accordingly. For the standard deviation 1.0, it takes a few batches for the performance to drop, with the change detector being a little slower as was expected.



(a) Gradual with std 1.0



(b) Gradual with std 2.0



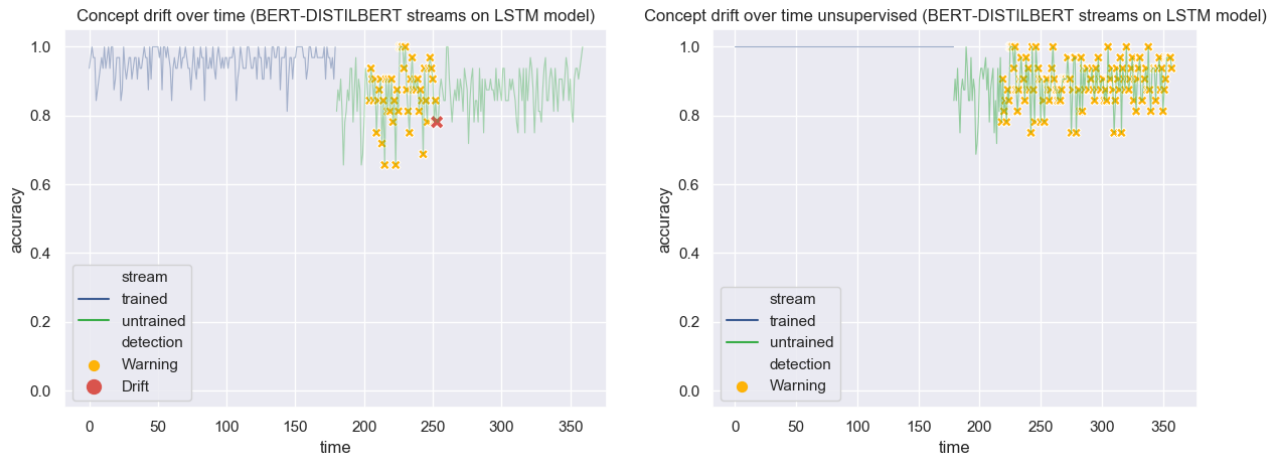
(c) Gradual with std 3.0

Figure 4.3: Detecting gradual change by adding random noise with different stds in the Naive Bayes model

4.2.2 LSTM model

Now that we have shown how the Naive Bayes model behaves under change, we can move on to a more interesting neural network based model which is supposed to provide a much better approximation for the input distribution (i.e. the BERT embeddings space), the LSTM. The results are presented in the same order as in the previous subsection, starting with changing embeddings from BERT to DistilBERT in order to simulate small abrupt drift when we assume labels and when the assumption is not made. These results are shown in Figures 4.4a and 4.4b and they confirm our expectations that the LSTM model behaves almost the same when changing from BERT to DistilBERT since they were purposefully designed to be interchangeable. We see from Figure 4.4a that the performance only drops slightly when switching streams, with only an approximate drop of 0.5 in accuracy on average on the green stream compared to the blue one. Moreover, the change detector correctly stays in the warning zone for a few dozen batches to see if the performance recovers before finally signalling a drift when about half the

stream has been run. When labels are not available (Figure 4.4b), we see that the green curves in both Figures look quite similar so the solution that we have come up with in which we replace the labels with the predictions of the first stream seems to be working quite well. However, since the random accuracies (between 0.9 and 1.0) that were fed to the change detector in the unsupervised case are a bit lower on average than the accuracies produced when we do have labels, the detector does not consider that the deviation is high enough to signal a drift, so it stays in the warning zone until the end of the stream. This could have been fixed by adjusting the sensitivity of the change detector, but for the purpose of the experiments, we have kept all parameters the same between the experiments. These results show that the DDM detector is robust enough to also detect change that is quite small, and even if it does not, it stays in a perpetual warning zone that can help identify drifts nonetheless. One improvement that we could bring to DDM is to also introduce a parameter that controls the length of the warning zone (i.e. if we are in a warning zone for a fixed duration, we signal an actual drift).

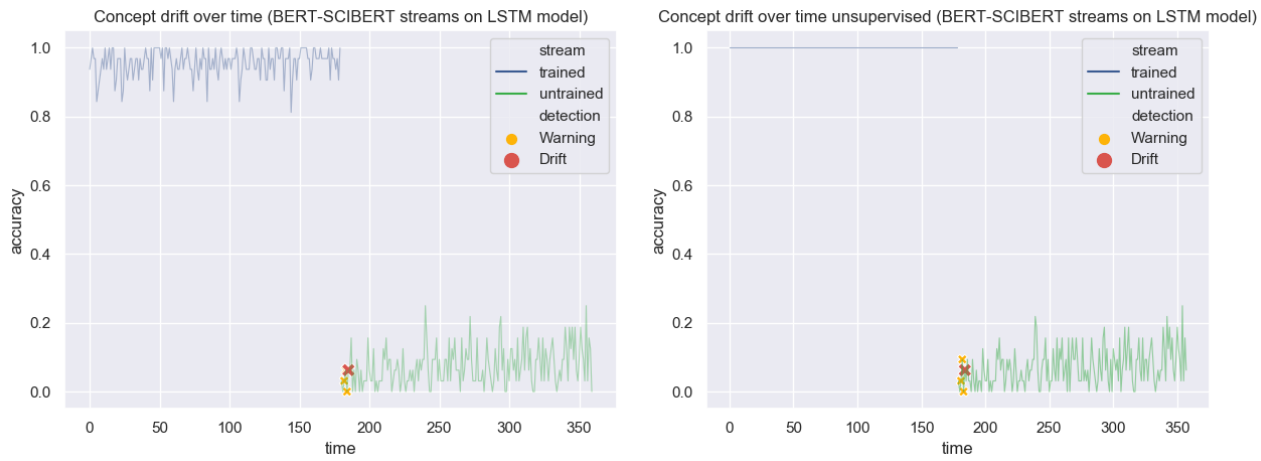


(a) Supervised change detection

(b) Unsupervised change detection

Figure 4.4: Detecting change using different embeddings (BERT-DISTILBERT) in LSTM model

Moving on to simulating big abrupt drift using a switch from BERT embeddings to SciBERT, the results are showcased in Figures 4.5a and 4.5b. These results were again as expected, with a huge downgrade in the performance of the LSTM happening instantly since the SciBERT embedding space is very different from the BERT space. The change detector signals a drift immediately as well, producing only two warnings as output in the supervised variant, and three in the unsupervised one. This is even faster than the results in the Naive Bayes model and this makes sense since the drop in performance is from almost perfect to random. The unsupervised case also performs exactly on par with the supervised, further proving that the method for detecting change when we have no labels works.

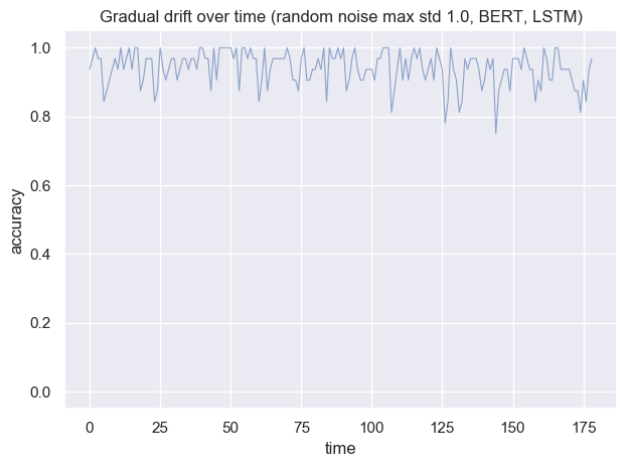


(a) Supervised change detection

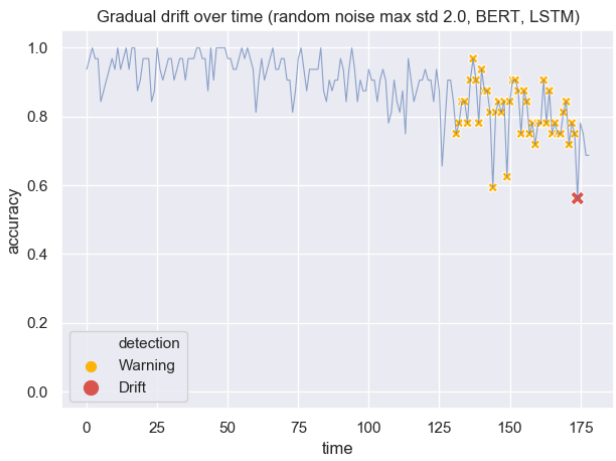
(b) Unsupervised change detection

Figure 4.5: Detecting change using different embeddings (BERT-SCIBERT) in LSTM model

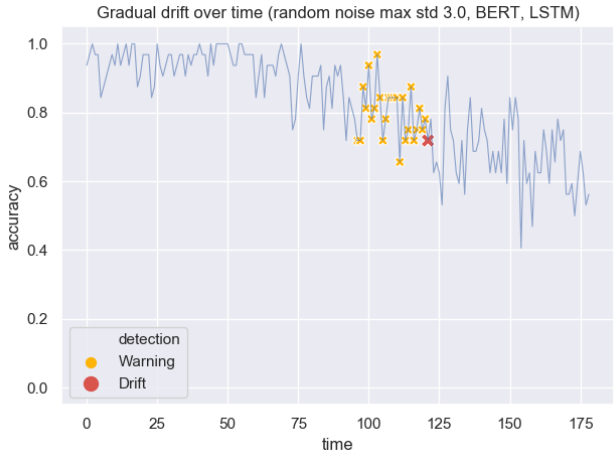
Finally, Figures 4.6a, 4.6a, and 4.6a present the results for adding gradual noise samples from a gaussian distribution with maximum standard deviations of 1.0, 2.0, and 3.0 respectively to the BERT embeddings to simulate a gradual drift. These results again prove that the LSTM model is much more robust then the Naive Bayes one, because even when small random noise is added to the inputs that the model uses, it still retains its performance. This can be clearly seen in the figures, where for a maximum standard deviation of 1.0, the model does not even have a performance drop for the whole duration of the stream, for a deviation of 2.0, it only enters the warning zone towards the end of the stream with a drift being signalled right at the end, and for a deviation of 3.0, the drift is detected after the middle of the stream.



(a) Gradual with std 1.0



(b) Gradual with std 2.0



(c) Gradual with std 3.0

Figure 4.6: Detecting gradual change by adding random noise with different stds in the LSTM model

4.3 Discussion

The results that were presented in the previous section showcase two important points:

- There is a clear difference in robustness between the two models that we have chosen. The Naive Bayes model *overfits* generalizes much worse than the LSTM model, and thus any small perturbation in its input leads to a heavy degradation in the performance of the model. This can clearly be seen from the change to the DistilBERT embeddings, which are trained to be almost equivalent to the BERT ones, where the model performed much worse than expected. The LSTM, however, did not have these issues, only showing the expected slight degradation in its performance when switching to the DistilBERT embeddings.

- The second takeaway is that the change detectors work quite well for all the models and even when we do not have labels with which to compute the accuracies. The results in this chapter can be improved, however, if the user of the pipeline knows beforehand what type of change they expect. If gradual drift is expected, EDDM is the preferred method since it signals change faster than the normal DDM. For abrupt drift, DDM works very well in the supervised, while for the unsupervised case, it could be a little faster. If no

labels are available, the sensitivity of the detector should be increased by decreasing the threshold for which a warning or a drift is signalled.

Chapter 5

Addressing change

After having discussed the problem of detecting change in a data stream when the embeddings space is altered, this chapter now turns to addressing the issues caused by that change. Specifically, it attempts to answer the second research question outlined in the introduction (Chapter 1), together with its two sub-questions: *Given that a change was detected, how can we adapt the model efficiently such that the performance is the same as it previously was before the change?*. The work will be presented in the next two sections, with the first one focusing on addressing small abrupt drift, where we can probably recover the original accuracy just by fine-tuning the model with a few batches from the new embeddings space. The second section in this chapter focuses on the much harder task of adapting to big abrupt drift, where the idea is to find a mapping between the two embedding spaces.

It is also worth mentioning that since one of the big motivations of this thesis was to save both the time and the compute cost associated with training big neural network models, we will only be attempting to address the changes that happen to the LSTM model, not the Naive Bayes one. This is because the Naive Bayes model requires almost no time and computation cost to train, and so adapting it is similar in time to full retraining.

5.1 Fine-Tuning

Starting of with the first part of second research question: *Given a small change (gradual drift or small abrupt drift), can we adapt the model by feeding it with a few examples, compared to retraining it from scratch?*, we address the small performance drop in the LSTM model (as seen in Figure 4.4) using a fine-tuning approach. Transfer learning has been an active area of research in deep learning for a few years now, keeping on trend with the need to make models cheaper and faster to train (Pan and Yang (2010)). It is a technique that aims to exploit the knowledge gained in one type of problem on another related task or domain. Typically, the first layers of a neural network contain low level information about the task (such as edges or colors in the convolutional layers of a CNN according to Pan and Yang (2010)), such that one can retrain only the last few layers of the network for the new task on which adaptation is required. This method is called fine-tuning, and although it still requires some amount of learning, it is much faster than a full retrain.

Since the BERT and DistilBERT embeddings are so closely related by construction (Sanh et al. (2019)), we expect we can use the fine-tuning approach in order to adapt the model from one embedding space to the other. Hashmani et al. (2019) mention that they use a similar approach to adapt an image dataset that suffers from concept drift for a classification task.

5.1.1 Experimental Setup

The approach taken to adapt the LSTM model to the DistilBERT embeddings is based on how convolutional neural networks are adapted to new datasets. Remember from Section 3.2.2 that the architecture of the model was made out of two LSTM layers followed by a final fully connected layer that was used for classification. Since the LSTM layers learn the structure of text by determining the long term dependencies between the words (Olah (2015)), we will keep these layers frozen, and only retrain the final classification layer. Since the fully connected layer only has a few parameters (2816 to be precise), the model can be adapted much faster than the time required to also retrain the two LSTM layers.

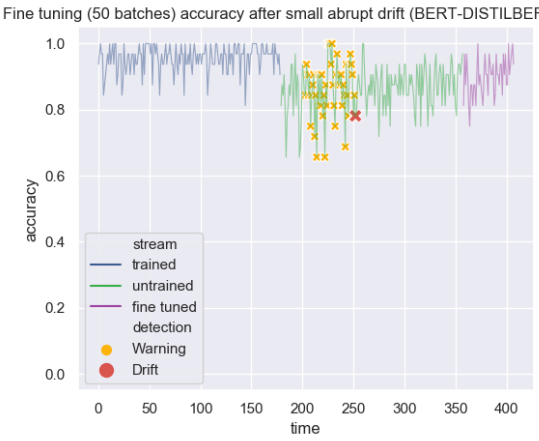
The experiments will take place in the same framework as those in the previous chapter, and they will have the same structure right before the adaptation part. We will start of with running the pipeline on the stream on which the model was trained on to build the statistics of the change detector, then we switch the embeddings while feeding the accuracies to the change detector and waiting for it to trigger warnings or drifts, and after the changed stream is finished, we finally move to the adaptation part. This final step of these experiments consist of re-running the stream with DistilBERT embeddings (this time with no change detector), and using the batches to train the classification layer of the LSTM while keeping track of the training accuracies to see if the model improves.

The adaptation part of the experiments will be run for 50, 100, and 200 batches of 32 samples (see Section 3.2) to see how many batches are needed for the model to adapt properly. Moreover, the batches will be fed in random order as opposed to the other steps where the batches came in deterministically. This is to make sure that the model didn't see only batches that belong to just a few classes, but actually receives samples that are representative of the whole dataset.

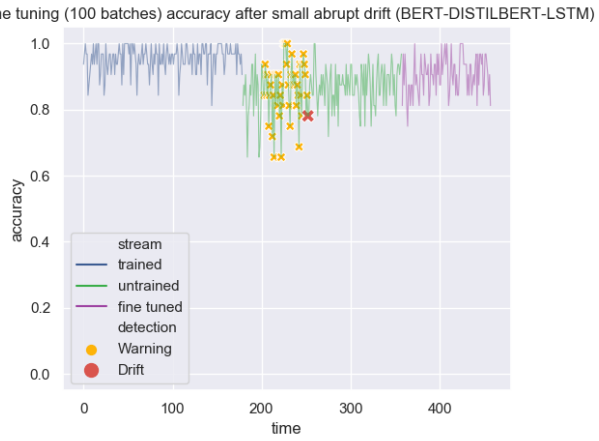
5.1.2 Results

The results for the fine-tuning approach when switching from BERT to DistilBERT embeddings, thus creating a small drop in performance, are shown in Figure 5.1. As mentioned above, we ran these experiments with three numbers of batches: 50, 100, and 200, which are shown in Figures 5.1a, 5.1b, and 5.1c respectively. The Figures show that, indeed, the fine-tuning approach manages to recover the accuracy of the model without needing to retrain the LSTM layers. While 50 seems to be too low of a number of batches for recovery, 100 already shows signs of clear improvement, while finally towards the end of the 200 batches it is quite clear that the performance of the model has recovered.

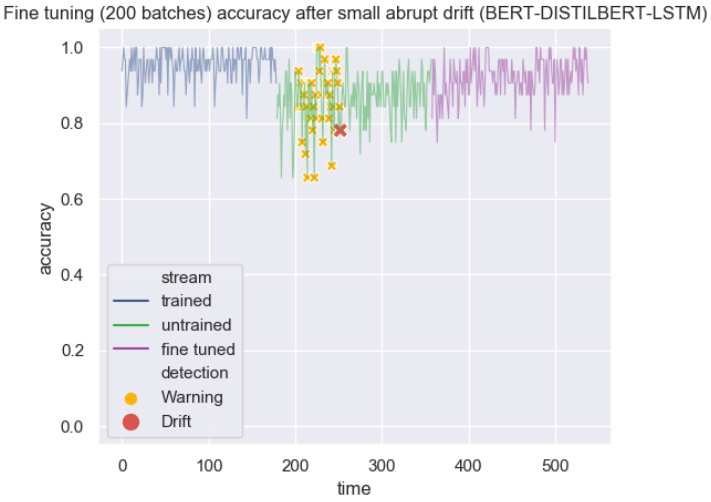
It is important to mention that the number of batches that need to be fed to the model in order for it to adapt is completely dependent on the actual model. While 200 batches work for this particular LSTM model, a different model may require a different number of batches. In most of the cases, however, fine-tuning should always manage to adapt the model quite fast, without needing to retrain from scratch.



(a) Fine-tuning experiments using 50 batches



(b) Fine-tuning experiments using 100 batches



(c) Fine-tuning experiments using 200 batches

Figure 5.1: Fine-tuning experiments

5.2 Mapping

Part two of the second research question turns our attention to addressing a substantial drop in the performance of the model and answering the following: *Given a big change (big abrupt drift), can we adapt the model by finding a mapping between the old outputs and the new ones which is less expensive to compute than retraining the model from scratch?*. As mentioned in the question above, this complicated problem will be approached by trying to find a mapping from the new embeddings (SciBERT) to the old ones (BERT), thus keeping the model trained on the BERT embeddings frozen and adding a new component to our framework 1.1 through which all inputs will be going such that they are switched from SciBERT to BERT so the model can understand them.

The Natural Language Processing field has been studying this mapping idea for a few years now (since Mikolov et al. (2013b)), but in the different context of trying to learn cross-lingual word embeddings. These methods attempt to find a mapping between embeddings for a source language and embeddings for a target language such that a model trained on the source language embeddings can also work with inputs that come from the target language. Mikolov et al. (2013b) first suggested this approach as an extension to their seminal word2vec paper (Mikolov et al. (2013a)), in which they construct a parallel corpus of words, each with a representation in the source embeddings space and the target embeddings space. The source matrix will be denoted as S, while the target matrix as T. They found that learning a simple linear mapping between S and T achieves similar results to more complicated solutions such as a multilayer neural network. Both Xing et al. (2015) and Artetxe et al. (2016) then found that enforcing an orthogonality constraint on the mapping matrix achieves better results on the same tasks. By putting this constraint in the problem, we transform it to the Orthogonal Procrustes problem (Schönemann (1966)). This states that given two matrices S and T, we would like to find a orthogonal matrix M that maps S to T as closely as possible. This boils down to the following equation:

$$M = \arg\min_{M'} ||M'S - T||_F \tag{5.1}$$

, where $||\cdot||_F$ is the Frobenius norm (Wikipedia contributors (2020b)) and $M'^T M' = I$ (I being the identity matrix). The closed-form solution can be found using Singular Value Decomposition on the matrix product between T and S^T, such that $U\Sigma V^T = \text{SVD}(TS^T)$ and $M = UV^T$. Both of the papers mentioned above rely on the assumption that there are bilingual dictionaries or parallel corpora readily available. The MUSE framework (Conneau et al. (2017)) manages to overcome this issue by employing a GAN (Goodfellow et al. (2014)) to train the mapping, where the generator acts as the actual mapping and the discriminator tries to guess whether a sample comes from the actual target embeddings or the mapped ones. The generator is then used a means to construct the bilingual dictionary which can finally be used to construct the linear mapping. This method manages to achieve results that are comparable with the Procrustes mapping when parallel data is actually available.

What all these methods described above have in common is that they assume that the embeddings used are non-contextualized (e.g., word2vec, glvoe), meaning that they only output one vector of embeddings per each word in the text, making it quite straightforward to construct a one to one mapping between the source and target embeddings. This is contrary to contextualized embeddings (e.g., BERT, ELMo), which can output multiple tokens per word thus producing multiple embedding vectors. Take for example the word *contextualized*. Its token representation when using BERT is $\begin{bmatrix}101 & 6123 & 8787 & 3550 & 102\end{bmatrix}$, while its representation when using SciBERT is $\begin{bmatrix}102 & 11695 & 645 & 103\end{bmatrix}$, and all these tokens have their respective vector representation. As we can see, both the number of tokens is different and also the tokens do not correspond between the two representations, thus it is impossible to construct a one to one

mapping between the words in this manner. Both Schuster et al. (2019) and Liu et al. (2019) have recently explored this problem in their work, but again in the context of trying to learn cross-lingual contextualized word embeddings. They define a word *anchor* as the aggregated representation of all the tokens in the word, where an aggregation can be either the average or the maximum of the token embeddings. They argue that a mapping learned using the anchors should hopefully generalize to the token-level embeddings as well. Schuster et al. (2019) arrive at this conclusion because they plotted the token representations of a few different words and found that the tokens for a word tend to cluster together in a point cloud.

5.2.1 Experimental Setup

Having given the necessary background into mappings between different embedding spaces, we can now explain how we are going to run our experiments. Before even talking about mappings, we first need construct a suitable dataset from our existing one, since token-level embeddings cannot be aligned, as explained above. Thus, we have decided to look through the whole Web of Science dataset and find the most common 10000 words there since picking all the words might skew the mapping because of noise. Then for all of the picked words, we can use both BERT and SciBERT to tokenize and subsequently embed them. Finally, we use the two aggregation methods described above (namely the maximum and the average) to get one embeddings representation per word resulting in the parallel dataset on which we can train a mapping.

Two methods will be tried in our experiments. The first is the classic Procrustes linear mapping, where the source matrix will be the SciBERT embeddings and the target matrix the BERT embeddings, both of them for the most common 5000 words. We expect this mapping to be very fast to train since it's a simple SVD on a matrix product, but not maximally accurate because it cannot model non-linear structure in the data. The second mapping that we picked was a simple multilayer perceptron with two fully connected layers that sandwich a ReLU activation function, such that the mapping is non-linear. We expect this one to be a little slower to train because even though it's a small network it still is a neural network trained using gradient descent, but it should be more flexible than the linear mapping.

We will put these assumptions to test in two ways. First, we want to measure how well the source matrix is actually mapped to the target and so we will showcase this by computing the mean-squared error between the mapped source matrix and the target and visualizing the embedding spaces using both PCA (Jolliffe (1986)) and t-SNE (van der Maaten and Hinton (2008)). Second, we also have to check how these mappings actually translate to a potential recovery in the performance of the LSTM model. For this purpose, we again turn to our pipeline to run the Web of Science stream three times: first, using the data transformed using BERT to build the statistics of the change detector, then we change the embeddings to SciBERT and let the detector do its work, and finally we again run the SciBERT stream but this time mapped to the BERT space using either the linear or the non-linear mapping.

Finally, we note that the linear mapping will be trained on the most common 5000 words, while the non-linear mapping will be trained on the most common 10000 since the MLP needs more training data.

5.2.2 Results

We first present the results for the linear Procrustes mapping. As expected, it was trained very fast, in just 2.34 seconds, on the 5000 most common words parallel dataset. Before the mapping, the MSE score between the source and the target is equal to 0.7274. After mapping, that score is reduced to 0.3134, which is quite good for an inflexible linear mapping, but still

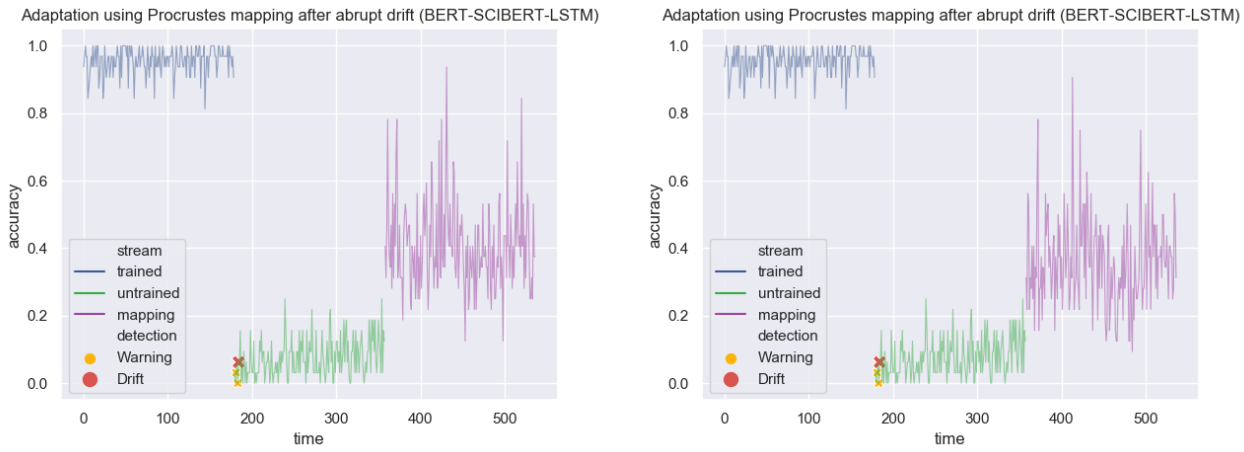
far from a robust mapping. This can also be seen from the PCA (Figure 5.2a) and the t-SNE (Figure 5.2b) visualizations. The red points in the visualizations are the source (SciBERT) embeddings, the green points are the target embeddings (BERT), and finally the blue ones are the mapped embeddings. If the mapping was perfect, the green and blue point clouds should be the same, but as we can see from figures, the point clouds are still pretty far from each other.

(a) Visualization using PCA

(b) Visualization using t-SNE

Figure 5.2: Visualizations for BERT, SciBERT, and the mapping between the two done using Procrustes

However, the Procrustes mapping still yields a decent improvement in the performance of the model, as can be seen from Figure 5.3. The mapping manages to achieve a decent average of 0.43 accuracy with a few outliers that also reach the performance of the model with the initial embeddings.The results are quite similar when using either the maximum or the average aggregation methods, with the maximum slightly outperforming the average.



(a) Results using **max** embeddings aggregation in the dataset

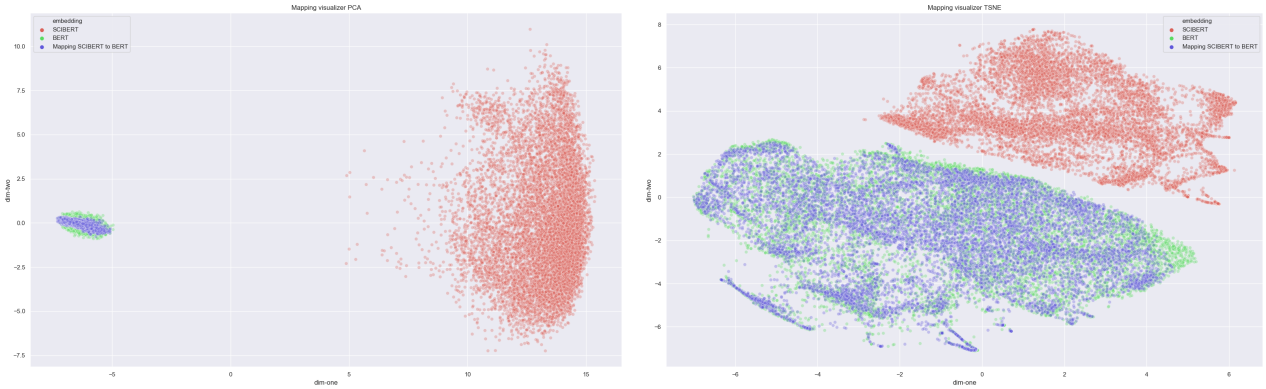
(b) Results using **average** embeddings aggregation in the dataset

Figure 5.3: Results for adaptation using the Procrustes mapping

Moving on to the MLP mapping, we can see from Figures 5.4a and 5.4b that the BERT embeddings represented by the green point cloud and the mapped embeddings represented by the blue point cloud overlap substantially, which is to be expected since we introduced a non-linearity between the two fully connected layers that reduces the bias of the model. This is verified by the MSE scores, with a 0.7205 before the mapping and a 0.1149 after. The training

time is much longer than the linear mapping, with 31.76 seconds, but which is still much quicker than a full retraining of the model.

The full summary for the scores and training times can be seen in table 5.1.

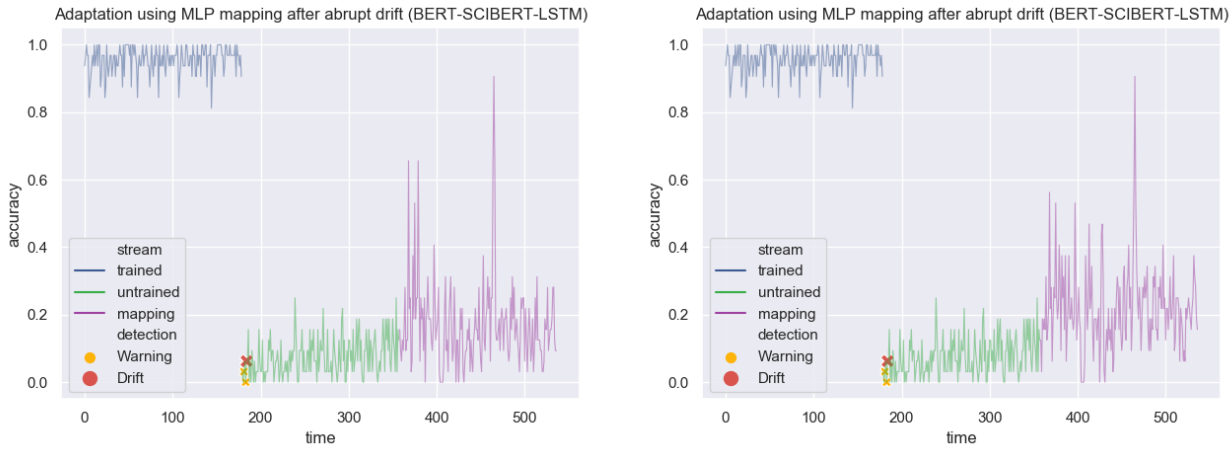


(a) Visualization using PCA

(b) Visualization using t-SNE

Figure 5.4: Visualizations for BERT, SciBERT, and the mapping between them done using MLP

Even though the MLP mapping manages to make the two point clouds of our dataset overlap, this however does not translate to good performance on our adaptation task. As can be seen from Figure 5.5, there is some accuracy improvement compared to the non-mapped stream, but it is small, and more importantly, it performs worse than the linear mapping. There is also a clear difference between the two aggregation methods here, with the average one performing better than the max one. These Figures also contain a few outliers, suggesting that some tokens are mapped well.



(a) Results using **max** embeddings aggregation in the dataset

(b) Results using **average** embeddings aggregation in the dataset

Figure 5.5: Results for adaptation using the MLP mapping

	MSE before mapping	MSE after mapping	Train time
Procrustes (5000 words)	0.7274	0.3134	2.34s
MLP (10000 words)	0.7205	0.1149	31.76s

Table 5.1: MSE scores and training times for both the Procrustes and the MLP mappings

5.3 Discussion

We can conclude from Section 5.1 that the first part of the second research question is answered by the fine-tuning approach to adapt from small abrupt concept drift. This approach can also be used to adapt from gradual drift by just using every sample to fine-tune the model continuously throughout the runtime of the pipeline, as mentioned in Hashmani et al. (2019). The fine-tuning can be triggered automatically whenever the model enters the warning zone, because if it is started only after a drift, the model might have diverged too far for a fine-tuning approach to work (as seen in Figures 4.6b and 4.6c).

As for the second part of that research question, we believe that the Procrustes mapping performs as well as it can given its linear nature, but the MLP mapping presents quite a paradox. On one hand, it maps the dataset that we constructed well, but on the other hand, it does not manage to adapt the model to the new SciBERT embeddings. We believe that this happens because of how we constructed the dataset using the anchors. Since the MLP is non-linear, it manages to produce a mapping with a low MSE score but that means that it also does not generalize to the non-anchored dataset which is used in the LSTM model. The Procrustes method on the other hand does not have such a low MSE score, but this translates to less overfitting on our anchored dataset and more generalization on the initial on. We have tried to fix this generalization problem by adding a dropout layer (Srivastava et al. (2014)) to the MLP for less overfitting, but that did not fix the issues. We also believe that a more complex neural network model would not help here since the issue is not about the flexibility of the model, but about the dataset. Thus, we conclude that the second part of the adaptation research question is not answered for now and we discuss a few ideas for future work in Chapter 7.

Chapter 6

Prototype Implementation

The work done in this thesis is geared more towards a real world optimization of machine learning data pipelines, and it should be of use to practitioners which operate in the NLP field that need to keep up to date with state of the art techniques but also want to reduce their costs in terms of both time and computational cost. Thus, we want to dedicate this chapter to describe a prototype implementation of the framework outlined in this thesis for the Zeta Alpha data pipeline.

Considering that Zeta Alpha's purpose is to extract scientific knowledge and make it easily accessible to both researchers and industry practitioners from around the world, the company requires an efficient and fast way to derive information from hundreds of thousands of scientific papers and blog posts. This is done using a data pipeline built using state of the art frameworks like Kafka [1] and MongoDB [2]. A scientific paper is put into the pipeline by a so-called *ingester*, then goes through a number of *converters* that extract a representation from the paper, before finally being stored in databases like ElasticSearch [3] or DGraph [4]. Examples of representations that can be extracted by the converters are: text out of a pdf document, citations out of a pdf document, splitting into sentences or words, cleaning the text, transforming the text to word embeddings, applying a named entity recognizer on the text, etc. Hence, we can think of the data pipeline as a sequence of nodes through which a document can go from the ingester until being stored in a database. In the current implementation of the pipeline, each one of those converters has an input and a deterministic output.

Given the current state of the pipeline, we have decided that a useful prototype of the work done in this thesis will only involve a *change detector* that will act as more of an integration test for the current version of the system. Since the outputs of the converters are deterministic, we can construct a test dataset out of a few dozen scientific papers for which we know the correct outputs and store it in a database. This dataset will be sent as a probe through the pipeline at specific time intervals to check that everything still performs according to specification. For each node/converter, we check the output that it produces for the given document, and compare it against the record that we have stored in the database. If the two are not equal, then the change detector will signal a drift.

When actual classification or regression machine learning models will be part of the pipeline, the plan is to extend the change detector such that it will more closely resemble the detector as it is presented in Chapter 4. Figure 6.1 shows how the service should interact with the pipeline once it's implemented. The leftmost node represents the ingester, while the rightmost ones are the databases. Everything in the middle is a converter, with the red ones being actual

[1] Kafka: `https://kafka.apache.org`
[2] MongoDB: `https://www.mongodb.com`
[3] ElasticSearch: `https://www.elastic.co/elasticsearch/`
[4] DGraph: `https://dgraph.io`

machine learning models that can access the change detector REST endpoint. The service will store one detection algorithm such as DDM or ADWIN for each model that uses it. Thus, each time a document passes through a model converter, a metric will be computed once the prediction is made and that metric will then be the input to the change detector, which in turn will communicate back to the model if it detects a warning or a drift.

Once the detection part is completed, each model converter will also need to handle the drop in performance if a drift is signalled. At first, this will be done by a simple retraining or fine-tuning using the newly introduced embeddings, but if the performance of the mapping improves in the future, that should also be implemented in the pipeline.

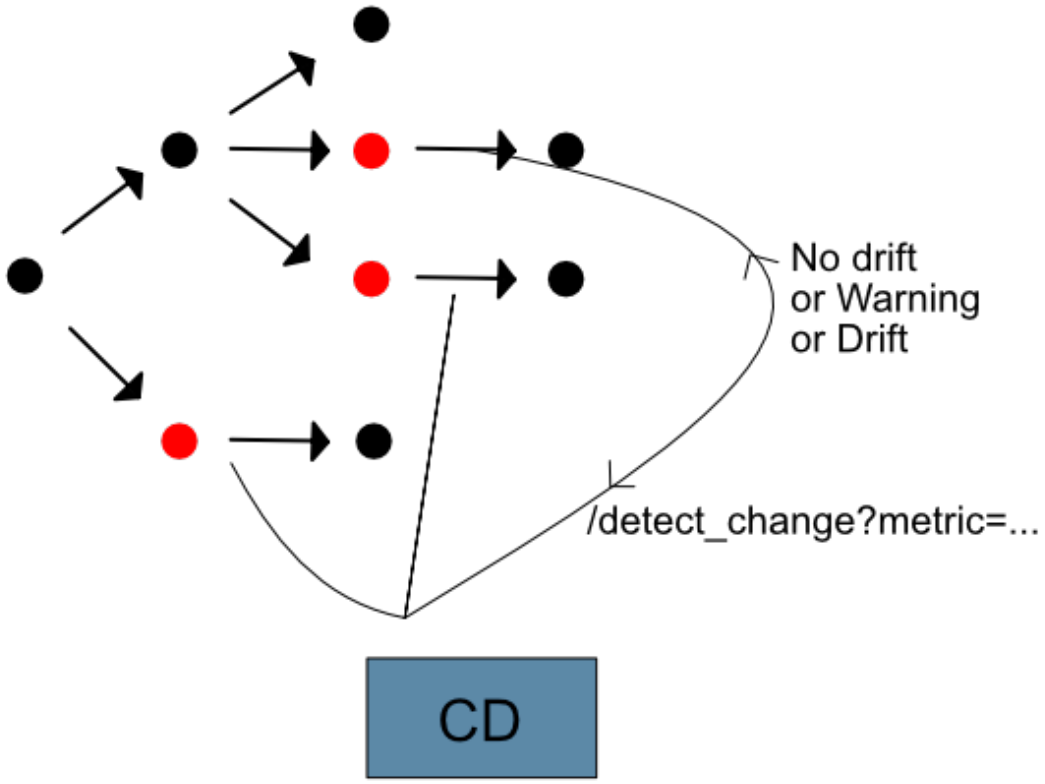


Figure 6.1: Change detector interaction with the Zeta Alpha pipeline

This chapter outlines a first implementation of the work done in this thesis for a real world machine learning pipeline. It shows that the work is applicable and provides a plan for future improvements to the pipeline once the prerequisites are implemented and the performance of the mapping improves.

Chapter 7

Conclusion

Throughout this thesis, we worked on making machine learning data pipelines both more autonomous and more efficient by having them detect drops in performance automatically and try to recover the lost efficacy without needing to retrain the models from scratch. Due to the large number of different configurations that a machine learning pipeline can have, be it because of the dataset or the flavor of the chosen model, we narrowed the extensive scope of our work to the natural language processing field. We set on to show how changing word embeddings can affect both a Naive Bayes and an LSTM model and then worked on addressing the issues caused by the changes. The decision to narrow the scope to this area was made because the natural language processing field has seen a lot of change this past decade, both in terms of models and word embeddings. Moreover, we chose the Web of Science dataset because the scientific literature is also affected by change in time as a result of the community's adjustment to trends.

We concluded that change can be detected promptly in both a supervised and an unsupervised framework, while finding that different embeddings have different impacts, SciBERT producing a big abrupt change and DistilBERT only slightly dropping the performance of the model. Chapter 4 also showed that changing embeddings does not affect all models equally, the LSTM model being much more robust and resistant to perturbations in the embeddings space than the Naive Bayes one. Moving on to addressing the change, Chapter 5 proves that a fine-tuning approach can adapt the model to the new embeddings much quicker than a full retrain when the pipeline only encounters a small abrupt drift. However, a big drop in the model performance of the model requires a different approach. We took inspiration from cross-lingual word embeddings to try and find a mapping between the new embeddings and the old ones, such that the model does not have to be adjusted at all. While the results show a significant recovery in the model's accuracy, the performance is only halfway what it used to be before the change and so this method is not yet good enough to be used in practice. Finally, Chapter 6 outlines how the ideas discussed in this thesis can be applied to a production level machine learning data pipeline.

7.1 Future Work

There are several avenues that can be pursued based on the work done in this thesis. First, there is a clear need for improvement in the mapping method that is used to address big abrupt drift, since the results obtained in Chapter 5 are not enough for the approach to be used in practice. We believe that pursuing a different model of constructing the mapping is not the way to go since the MLP already maps the aggregated dataset well, which can be seen from both the MSE score and Figure 5.4. Instead, we propose that future work should focus its attention on coming up with a different dataset, either by finding a different way to aggregate

the embeddings for a single word, or by actually searching for a way to align the actual tokens of contextualized word embeddings. The second option should result in much better results with any mapping method, since the actual embeddings are mapped, not some aggregations of them. Moreover, even if the mapping cannot achieve the desired results, but at least get close, a combined approach of mapping and fine-tuning can then get the performance of the model back to the initial state.

As previously mentioned, the framework that we set up in Chapter 1 is rather flexible, and hence future work can attempt to expand it by working with other models, finding different agents of change other than word embeddings, or even change the field entirely to something other than natural language processing.

Bibliography

[1] Artetxe, M., Labaka, G., and Agirre, E. (2016). Learning principled bilingual mappings of word embeddings while preserving monolingual invariance.

[2] Baena-García, M., Campo-Ávila, J., Fidalgo-Merino, R., Bifet, A., Gavald, R., and Morales-Bueno, R. (2006). Early drift detection method.

[3] Beltagy, I., Lo, K., and Cohan, A. (2019). Scibert: A pretrained language model for scientific text.

[4] Bifet, A. and Gavaldà, R. (2007). Learning from time-changing data with adaptive windowing. In *SDM*.

[5] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink™: Stream and batch processing in a single engine.

[6] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3).

[7] Conneau, A., Lample, G., Ranzato, M., Denoyer, L., and Jégou, H. (2017). Word translation without parallel data. *arXiv preprint arXiv:1710.04087*.

[8] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding.

[9] Duda, R. O., Hart, P. E., and Stork, D. G. (2001). *Pattern Classification*. Wiley, New York.

[10] Elwell, R. and Polikar, R. (2011). Incremental learning of concept drift in nonstationary environments. *IEEE Transactions on Neural Networks*, 22:1517–1531.

[11] Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R., editors (1996). *Advances in Knowledge Discovery and Data Mining*. American Association for Artificial Intelligence, USA.

[12] Gama, J., Medas, P., Castillo, G., and Rodrigues, P. (2004). Learning with drift detection.

[13] Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, H. (2014). A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)*, 46.

[14] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial networks.

[15] Guo, X., Yin, Y., Dong, C., Yang, G., and Zhou, G. (2008). On the class imbalance problem.

[16] Hashmani, M. A., Jameel, S. M., Alhussain, H., Rehman, M., and Budiman, A. (2019). Accuracy performance degradation in image classification models due to concept drift. *International Journal of Advanced Computer Science and Applications*, 10(5).

[17] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

[18] Hodge, V. and Austin, J. (2004). A survey of outlier detection methodologies. *Artificial Intelligence Review*.

[19] Hoi, S. C. H., Sahoo, D., Lu, J., and Zhao, P. (2018). Online learning: A comprehensive survey.

[20] Jolliffe, I. T. (1986). *Principal Component Analysis and Factor Analysis*, pages 115–128. Springer New York, New York, NY.

[21] Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2016). Bag of tricks for efficient text classification.

[22] Kifer, D., Ben-David, S., and Gehrke, J. (2004). Detecting change in data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 180–191. VLDB Endowment.

[23] Kowsari, K., Brown, D. E., Heidarysafa, M., Jafari Meimandi, K., , Gerber, M. S., and Barnes, L. E. (2017). Hdltex: Hierarchical deep learning for text classification.

[24] Kowsari, K., Jafari Meimandi, K., Heidarysafa, M., Mendu, S., Barnes, L. E., and Brown, D. E. (2019). Text classification algorithms: A survey. *Information*, 10(4).

[25] Kreps, J. (2011). Kafka : a distributed messaging system for log processing.

[26] Li, Y. and Yang, T. (2017). Word embedding for understanding natural language: A survey.

[27] Liberty, E., Karnin, Z., Xiang, B., Rouesnel, L., Coskun, B., Nallapati, R., Delgado, J., Sadoughi, A., Astashonok, Y., Das, P., Balioglu, C., Chakravarty, S., Jha, M., Gautier, P., Arpin, D., Januschowski, T., Flunkert, V., Wang, Y., Gasthaus, J., and Smola, A. (2020). Elastic machine learning algorithms in amazon sagemaker.

[28] Liu, Q., McCarthy, D., Vulić, I., and Korhonen, A. (2019). Investigating cross-lingual alignment methods for contextualized embeddings with token-level evaluation. In *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*, pages 33–43, Hong Kong, China. Association for Computational Linguistics.

[29] Mikolov, T., Chen, K., Corrado, G. S., and Dean, J. (2013a). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

[30] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality.

[31] Mitzenmacher, M., Steinke, T., and Thaler, J. (2011). Hierarchical heavy hitters with the space saving algorithm. *CoRR*, abs/1102.5540.

[32] Montiel, J., Read, J., Bifet, A., and Abdessalem, T. (2018). Scikit-multiflow: A multi-output streaming framework.

[33] Olah, C. (2015). Understanding lstm networks.

[34] Page, E. S. (1954). Continuous inspection schemes. *Biometrika*, 41(1-2):100–115.

[35] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359.

[36] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Müller, A., Nothman, J., Louppe, G., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Édouard Duchesnay (2012). Scikit-learn: Machine learning in python.

[37] Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation.

[38] Perrault, R., Shoham, Y., Brynjolfsson, E., Clark, J., Etchemendy, J., Grosz, B., Lyons, T., Manyika, J., Mishra, S., and Niebles, J. C. (2019). The ai index 2019 annual report.

[39] Pesaranghader, A., Viktor, H., and Paquet, E. (2017). Mcdiarmid drift detection methods for evolving data streams.

[40] Peters, M., Neumann, M., Zettlemoyer, L., and Yih, W.-t. (2018a). Dissecting contextual word embeddings: Architecture and representation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1499–1509, Brussels, Belgium. Association for Computational Linguistics.

[41] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018b). Deep contextualized word representations.

[42] Rettig, L., Khayati, M., Cudré-Mauroux, P., and Piórkowski, M. (2019). Online anomaly detection over big data streams. In *Applied Data Science*.

[43] Rish, I. (2001). An empirical study of the naive bayes classifier.

[44] Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter.

[45] Schelter, S., Bießmann, F., Januschowski, T., Salinas, D., Seufert, S., and Szarvas, G. (2018). On challenges in machine learning model management. *IEEE Data Eng. Bull.*, 41:5–15.

[46] Schuster, T., Ram, O., Barzilay, R., and Globerson, A. (2019). Cross-lingual alignment of contextual word embeddings, with applications to zero-shot dependency parsing.

[47] Schönemann, P. (1966). A generalized solution of the orthogonal procrustes problem. *Psychometrika*, 31(1):1–10.

[48] Sherstinsky, A. (2020). Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306.

[49] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958.

[50] Stone, J. V. (2013). *Bayes' Rule: A Tutorial Introduction to Bayesian Analysis*. Sebtel Press.

[51] van der Maaten, L. and Hinton, G. E. (2008). Visualizing data using t-sne.

[52] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

[53] Wang, H. and Abraham, Z. (2015a). Concept drift detection for streaming data. *2015 International Joint Conference on Neural Networks (IJCNN)*.

[54] Wang, H. and Abraham, Z. (2015b). Concept drift detection for streaming data. *2015 International Joint Conference on Neural Networks (IJCNN)*.

[55] Wang, S. and Manning, C. (2012). Baselines and bigrams: Simple, good sentiment and topic classification. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 90–94, Jeju Island, Korea. Association for Computational Linguistics.

[56] Wikipedia contributors (2020a). Big data — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Big_data&oldid=958939793`.

[57] Wikipedia contributors (2020b). Matrix norm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Matrix_norm&oldid=968061260`.

[58] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., and Brew, J. (2019). Huggingface's transformers: State-of-the-art natural language processing.

[59] Xing, C., Wang, D., Liu, C., and Lin, Y. (2015). Normalized word embedding and orthogonal transform for bilingual word translation. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1006–1011, Denver, Colorado. Association for Computational Linguistics.

[60] Yaqoob, I., Hashem, I., Gani, A., Mokhtar, S., Ahmed, E., Anuar, N., and Vasilakos, A. (2016). Big data: From beginning to future. *International Journal of Information Management*, 36.

[61] Yu, S., Abraham, Z., Wang, H., Shah, M., Wei, Y., and Príncipe, J. C. (2019). Concept drift detection and adaptation with hierarchical hypothesis testing. *Journal of the Franklin Institute*, 356(5):3187–3215.

[62] Zhou, P., Qi, Z., Zheng, S., Xu, J., Bao, H., and Xu, B. (2016). Text classification improved by integrating bidirectional lstm with two-dimensional max pooling.

[63] Zimek, A. and Filzmoser, P. (2018). There and back again: Outlier detection between statistical reasoning and data mining algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(6):e1280.