University of Applied Sciences Würzburg-Schweinfurt Faculty of Computer Science and Business Information Systems

Bachelor-Thesis

Evaluation and implementation of gradient descent algorithms on streaming data

submitted to the University of Applied Sciences Würzburg-Schweinfurt in the Faculty of Computer Science and Business Information Systems to achieve the Bachelor of Engineering degree in 'Information Systems'

Florian Hohn

Submitted on: 25.02.2020

First Reader: Prof. Dr. Frank-Michael Schleif
Second Reader: Moritz Heusinger

Abstract

TODO

Note of thanks

Contents

1 Introduction

The following chapter will give the reader a short explanation what inspired the work for this thesis, what its prupose is and what it wishes to achiev, as well as an explanation how this work is structured, to give the reader a better oriantation for how to navigate this work.

1.1 Inspiration for this Thesis

There is an ever growing amount of data that needs to be processed in real time to be of use nowadays, e.g. for use in selfdriving cars, from sensors that monitor processes in the industry to improve their quality [5] or from data streams originating in the Internet that can be used to detect epidemics and natural disasters that can be combined with statistics from official centers for disease and disaster control for better prevention [5]. The problem is this flood of data has outpaced our capability to process, analyze, store, and understand these datasets. Traditional Machine learning algorithms are not capable of learning and training from this flood of data in a acceptable amount time. The reason for this, is that the stream can suddenly change (so called "Concept drift"). Data is also rapidly and constantly generated, which means that the learning algorithm can not use the data more than once to learn from it. For this reasons we need algorithms that can deal with this amount of data in real-time while also needing only a reasonable amount of resources [5].

1.2 Purpose of this Thesis

In [13] was shown that, on average, the $RSLVQ_{Adadelta}$ performance better in nearly all aspects beside the computation time and the Kappa Performance in respect to the other versions of the RSLVQ that were tested in this work. It had further shown that with delayed settings the $RSLVQ_{RMSprop}$ performed better than the $RSLVQ_{Adadelta}$ and the $RSLVQ_{SGD}$ implementation. This work will compare another modfified RSLVQ version, that is based on the ADAM algorithm, with the other already tested RSLVQ

versions when they are used on streaming data. It will compare them in respect to their returned accuracy, performance and computation time on different synthetic and real data streams with different evaluation approaches. According to [15] the $RSLVQ_{Adam}$ should compare favourably in comparison towards the other adaptive learning algorithms and should therefore be returning the best results in the experiments.

1.3 Outline of the Thesis

The second Chapter of this thesis will give a short overview over the basics of Machine Learning that are relavant for the understanding of the premise for the thesis. It will give also a short describtion about what is special about Data stream in Machine learning algorithms and what is importend to take into account when working with them. It will also give a short introduction about the criterias with which the comparision of the different implementations of the Algorithm take place and why these criteria are used for comparision. In the last part of the first chapter will be the framework introduced that was used for implementation and comparision.

The next chapter will then continue to give a deeper look into the deeper parts of the subsection of Machine learning and the Algorithm that is the basis for the different variation of implementation for the Algorithms. It will also give a short introduction about the version that this algorithm was based on, as this is importend to later understand the varietions and differences in the other implementations. In the last part of that chapter will then explain the differnet methods and the implementations of the varieants in greater detail.

The fourth chapter will then list and describe the used synthetic and real world streaming data sets that where used to perform the experiments. Furthermore will be describe how the experiments where conductet and with wich methods the results where controlled.

Following this, the next chapter will be dedicated for the result evaluation what what we informations where hoefully gained from conducting these experiments for the comparision of thes implementations.

The last chapter will summarys all the work that was done and give a final evaluation to the algorithms.

2 Basics

The Goal of this Chapter is to give the Reader a basic understanding of how Machine Learning Algorithm work. Furthermore, it will describe what the difference is between unsupervised und supervised learning.

There will be also an introduction and description on how the Gradient Descent Algorithm, that is the basis for the Modified versions of the Algorithms that will be compared in this work, works and how it decides what to do. After this is done there will be also a short description on what Streaming Data is and on the framework that is used to realise the implementation of the Experiments.

Finally there will be a explanation on how the criteria for the comparision of the Algorithms where choosen and how theses criteria work.

2.1 Machine Learning

The term 'Machine Learning' stands for a collection of self-learning algorithms, that learn form input Data to make fast and correct decisions and predictions. It is importand to remember that it is not a real Artifical inteligence, but rather a sequence of statistical analyses on given data, with the goal to make gradualy improvments to the prediction models of the algorithm. A Machine can do this optimization alot faster and effectifly than a humans, especially if this optimization has to be done in real time on big data input sets. This give also a bigger flexability, as machine that can learn from a changing enviroment needs lesser redesigns, which leads to less time that gets wastet and can be used more productifly on other tasks.

All these are reasons why Machine Learning has an ever increasing importants in the field of computer science, but also an growing impact on everday life. It is thanks to Machine learning that there are intelligent assisstant softwares that can recognize voice commands and questions (Apples SIRI, Amazon Alexa or Microsofts Cortana), e-mail spam filter, nearly self-driving cars, reliable search-engines, faster and more precise weather forecasts and challenging game ai's nowadays

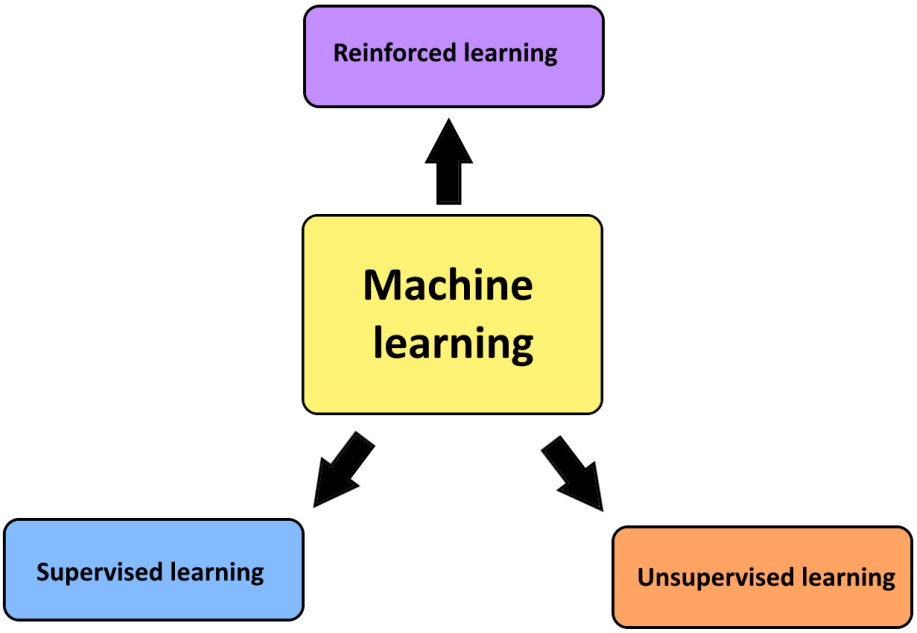

Figure 2.1: Overview of the Machine Learning Strategy's.

Machine Learning Algorithms can be categorisized depending on how they learn from the available training Data to create a model for the predictions. There are three main subdivisions, under wich these algorithms can fall. The first one is the Supervised learning strategy, then the Unsupervised learning strategy and the Reinforced learning strategy.

Only the Supervised Learning strategy will be explained in more detail on the following pages, as the focus of this thesis lays in the comparision of several different implementations of classification algortihms that are based on Supervised Learning.

[19]

2.1.1 Supervised Learning

The idea behind Supervised learning is to create a model from a already labeled training data set, that then can make prediction and decision on new or unknown data based on the learned model from the training set. The following example will help to understand this easier. Lets say a fruit farmer wants to automatically sort the harvested Fruits in either categorie A or B for the market. The sorting device is equiepped with 2 sensors to measure 2 features, one for colour and another one for the size of the fruit. It then decides to wich of the two classes the fruit belongs to. As this is a System that is capable of dividing features into a discrete number of classes the system can be called a *classifier*. This is a typical exampel of an *classifiecation task*. To configure the System in such a way that it correctly sorts the fruits into the right category, a specailist hand picks several fruits from a small test set and then sorts them depending on the already named features. In other words, the specailist classiefies them. Based on this by hand sortet training set the machine will then later decide how the other fruits should be sorted. It is used as a *model* for the classification. [10] Another type of Supervised learning is the *regression task*. The difference to the *classification task* is, that instead of a *discrete* number of classes that gets returned, a *continous* number of classes will be returned. An example of a *regressive task* is, for example, the prediction of the Stock market or the price of medications.

The Process as seen in figure 2.2 and as described as in the example of the farmer and his sorted fruits shows how the supervised learning works. The Algorithm (marked in green) learns with the help of the labeled training data. The result of this learning is then that the trained Algorithm, now called predictive model, can be used in a production enviroment. New, unlabeled, data can then be send into the predictiv model to get a predictied model as an output. [22]

To better understand how this predictiv model is formed, one has to understand how the classification task works. In the classification task the computer programm is asked to specify which of k catergories a input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathbb{R} \rightarrow 1, ..., k$. When $y = f(\boldsymbol{x})$, the model assigns an input described by vector $\boldsymbol{x}$ to a category identified by numeric code y. There are other variants of the classification task, for example, where f outputs a probability distribution over classes. An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values), and the output is a numeric code identifying the object in the image. [12]

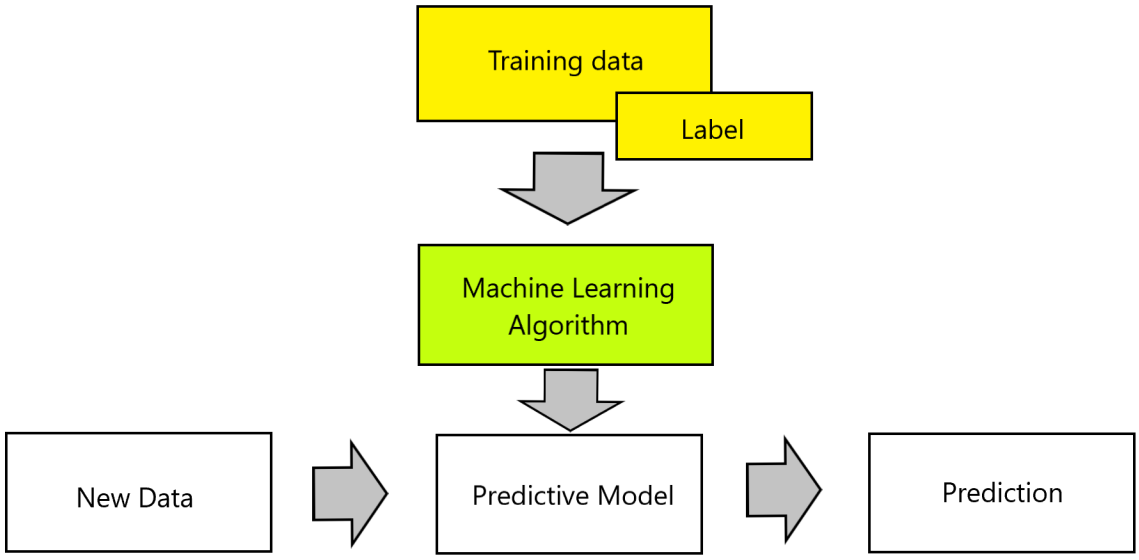

Figure 2.2: Process of the Supervised Learning. [22]

Then there must be also differentiated between the type of data that is put into the classification process. In batch or offine classification, a classifier-building algorithm is given a set of labeled examples. The algorithm creates a model, a classifier in this case. The model is then deployed, that is, used to predict the label for unlabeled instances that the classifier builder never saw. If we go into more detail, we know that it is good methodology in the first phase to split the dataset available into two parts, the training and the testing dataset, or to resort to cross-validation, to make sure that the classifier is reasonably accurate. But in any case, there is a first training phase, clearly separated in time from the prediction phase.

In the online setting, and in particular in streaming, like it is this thesis, this separation between training, evaluating, and testing is far less clear-cut, and is interleaved. We need to start making predictions before we have all the data, because the data may never end. We need to use the data whose label we predict to keep training the model, if possible. And probably we need to continuously evaluate the model in some way to decide if the model needs more or less aggressive retraining. [5]

2.1.2 Gradient Descent

The Gradient descent Algorithm is one of the most popular Algorithms to perform optimizations on and is by far the most commen way to optimize neural networks.This is also the reason why nearly every state-of-the-art Deep learning library has various forms of implementations to perform optimization on the gradient descent. The Problem, however is, that these Algorithms often perform as some form of black-box, because a practical explanation of what their stengths and weaknesses is are hard to find and explain. Gradient descent is a way to minimize an objective function $J(\theta)$ parameterizedby a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite directionof the gradient of the objective function $\nabla_\theta \in J(\theta)$ with respect to the parameters. The learning rate η determines the size of the steps wetake to reach a local minimum. In other words, we follow the direction of the slopeof the surface created by the objective function downhill until we reach a valley. [23]

The normal gradient descent, also known as batch-gradient descent, computes the gradient of the functionwith respect to the parameters θ for the entire training set:

$$\theta = \theta - \eta * \nabla_\theta J(\theta) \tag{2.1}$$

Since we need to calculate the gradients for the complete dataset to get just one update, batch gradient descent can get very slow and is intractable for dataset that do not fit into the memory. For this reason the batch gradient descent implementation can not be used in this comparison, as it is not able to perform updates "on-the-fly", as this is a necessarity for use on streaming Data.

In contrast to that, the Stocastic Gradient Descent (SGD) can perform a update for *each* training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta = \theta - \eta * \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{2.2}$$

While the bacht gradient descent performed redudant computations for large datasets, as it recomputes gradients for similar examples before each update, the SGD puts away this redundacy by performing one update at a time. Because of that it performce much faster and can be used on Streaming Data. The SGD performs frequent updates with a high variance, that causes the objective function to fluctuate heavily.

This fluctuation enables the SGD to jump to new and potetially better local minima, while the batch gradient only converges to the minimum of the basin the parameters are placed in. But this also ultimatly complicates the convergensto the exact minimum, as

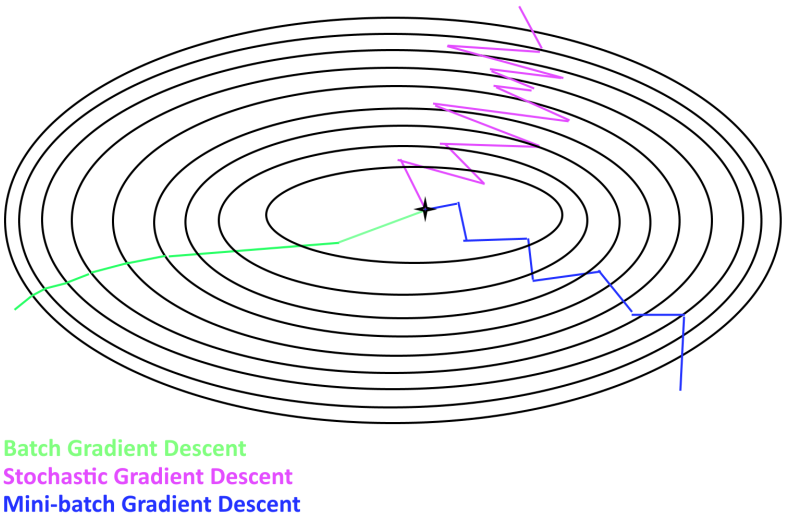

Figure 2.3: Visualisation differences of the three Gradient Descent Algorithms

the SGD keeps overshooting. This can be fixed by slowly decreasing the learning rate, as the SGD then shows the same convergeance behaviour as the batch gradient descent. It then almost certainly converges to a local or gloabl minimum for non-convex and convex optimization respectivly. [23] These advantages lead to the fact that the SGD is the most common Gradient descent algorithm.

Then there is also the Mini-batch gradient descent algorithm. It takes the best of both worlds and performs an update for every mini-batch on n training examples:

$$\theta = \theta - \eta * \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.3}$$

This way it reduces the variance of the parameter updates, which can lead to a more stable convergence. It can also make use of highly optimized matrix optimizations common for state-of-the-art deep learning libraries, that makes computing the gradient with respect to a mini-batch very efficient. Common mini-batch sizes range between 50 and 256 , but can vary for different applications. Typically the Mini-batch gradient descent is the algorithm of choice when there is a neural network to train and the term SGD is usually also employed even when a mini-batches are used. [23]

2.2 Streaming Data

Data streams are an algorithmic abstraction to support real-time analytics. They are sequences of items, possibly infinite, each item having a timestamp, and so a temporal order. Data items arrive one by one, and we would like to build and maintain models, such as patterns or predictors, of these items in real time. There are two main algorithmic challenges when dealing with streaming data: the stream is large and fast, and we need to extract information in real time from it. That means that usually we need to accept approximate solutions in order to use less time and memory. Also, the data may be evolving, so our models have to adapt when there are changes in the data. [5]

2.2.1 Increasing Volume of the Data

When the Data items keep arriving fast and in large quantatiy a algorithm is no longer able to process the data efficiently by using multiple passes. Instead, it has to be designed in such a way that it is able to process the data with one pass [1]. [5] describes five requirements that a algorithm has to fullfill, if it should be able to process streaming data successfull:

- Process an instance at a time, and inspect it (at most) once.
- Use a limited amount of time to process each instance.
- Use a limited amount of memory.
- Be ready to give an answer (prediction, clustering, patterns) at any time.
- Adapt to temporal changes (Concept drift).

The last point, "Adapt to temporal changes", has to do with evolving data streams and will be described further in the following section.

2.2.2 Drift in Streaming Data

Often there is a inherent temporal component to the stream mining process. This is because the data may evolve over time. This behaviour of a data stream is known as temporal locality[1] (also known as Concept drift)[5].Therefore, a straight forward adaptation of one-pass mining algorithms may not be an effective solution to the task. Thus, Stream mining algorithms need to be carefully designed with a clear focus on the

evolution of the underlying data, wich leads the the fith requirement (see enumeration 2.2.1) described in [5].

Now, to be more specific, the term Concept drift will be descript further. It does not mean that the items that are observed today are not exactly the same as those that were observed yesterday. A more reasonable notion is that statistical properties of the data change more than what can be attributed to chance fluctuations. To better understand this, it helps to assume that the data is in fact the result of a random process that at each time generates an item according to a probability distribution that is used at that exact time, and that may or may not be the same that is used at any other given time. There is no change (drift) when this underlying generating distribution remains the same. Drift occurs whenever it varies from one time step to the next [5].

Often, an additional independence assumption is implicitly or explicitly used: that the item generated at time t is independent of those generated at previous time steps. This assumption is often incorrect or false, because in many situations the stream has memory, and experiences bursts of highly correlated events. For example, a fault in a component of a system is likely to generate a burst of faults in related components [5].

Although Concept drift in the item distribution may be arbitrary, it helps to name a few generic types, which are not exclusive within a stream:

- *Sudden* drift occurs when the distribution has remained unchanged for a long time, then changes in a few steps to a significantly different one. It is often called *shift*.

- *Gradual* or *incremental* drift occurs when, for a long time, the distribution experiences at each time step a tiny, barely noticeable change, but these accumulated changes become significant over time.

- Drift may be *global* or *partial* depending on whether it affects all of the item space or just a part of it. In ML terminology, partial change might affect only instances of certain forms, or only some of the instance attributes.

- *Recurrent* concepts occur when distributions that have appeared in the past tend to reappear later. An example is seasonality, where summer distributions are similar among themselves and different from winter distributions. A different example is the distortions in city traffic and public transportation due to mass events or accidents, which happen at irregular, unpredictable times.

- In prediction scenarios, we are expected to predict some outcome feature Y of an item given the values of input features X observed in the item. *Real* drift occurs when $Pr[Y \mid X]$ changes, with or without changes in $Pr[X]$. *Virtual* drift occurs when $Pr[X]$ changes but $Pr[Y \mid X]$ remains unchanged. In other words, in

real drift the rule used to label instances changes, while in *virtual* drift the input distribution changes [5].

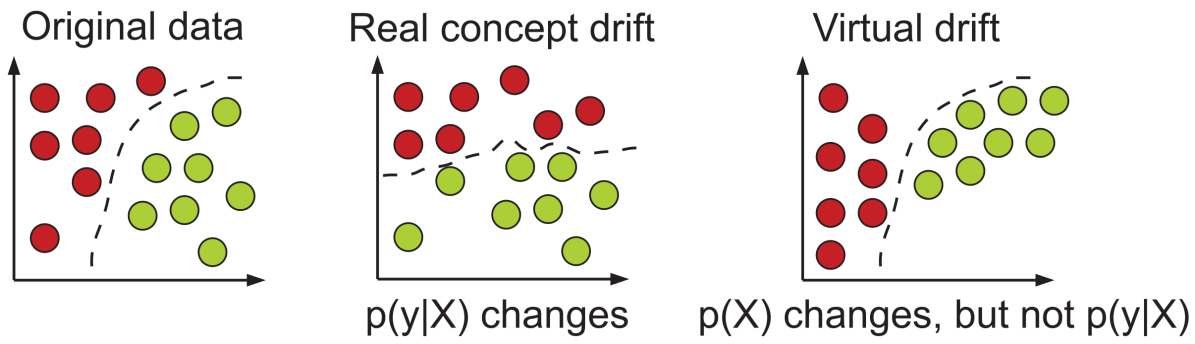

Figure 2.4: Visualization of types of Concept drift [11]

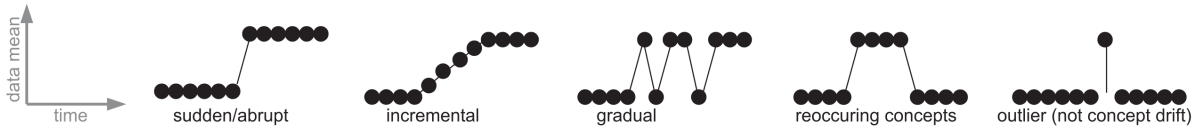

Figure 2.5: Visualization of real and virtual drift [11]

[11]

2.3 Criteria for the Comparision

This section will explain on what criteriy this thesis will base the comparsion of the different implementations. It will also descirbe what the difference between the cireria is and why this is importend to know. It will define what this thesis understands under the terms performance and accuracy, as well as what the differences between the used error-estimations are.

2.3.1 Evaluations for Data Streams

To evaluate a learning algorithmus, it must be know which examples were used to train the algorithmus and which are used to test the model output by the algorithms. Which procedure is used in batch learning has partly depended on data size. For small data sets with < 1000 examples the procedure of cross-validation was well suited, as it made

maximum use of the data that it could use. A Problem arose when the size of the data keept increasing, as this practicly created a time limit for procedures that keept repeating the training process to many times. For this reason it is commen practise to try to reduce the number of folds and repetitions for big data sets to allow experiments to conclude in reasonable time. One way to achieve this is to take several hundreds of thousend samples in a single batch and take them for a single holdout run, as this requires the least computational effort. The rationalization for this, beside the practical time issues, is that the reliability we may lose by not having repeated runs is compensated by the reliability gained by the sheer number of samples that are used. There are two main approaches that we have to consider for the evaluation process, the first one being Holdout and the second being Prequential. [2]
These two approaches will be now be described in more detail:

- **Holdout**: It is often acceptable to measure the performance with a single holdout set when the batch learning learning would otherwise reache a scale where cross-validation would be too time consuming. This can get very helpful, as the results of different studies can be directly compared to each other. To do this, the division between the train and test set have to be predefiened. To track model performance over time, the model can be evaluated periodically, for example, after every one million training examples. Testing the model too often has potential to significantly slow the evaluation process, depending on the size of the test set. A source for holdout dat sets are for example sets of data of the stream that have not yet been used to train the learning algorithm. These sets could be taken from "further ahead" in the stream to be used as test sets and could then later be used again to train the learning algortihm again after the testing is complete. This procedure would be recommandable in scenarios with concept-drift, as it would measure a model's ability to adapt to the latest trends in the data. In scenarios without drift, a single static holdout set should be sufficent, as this would avoid varying estimates between test sets. If it is assumed that the test set is independent and sufficently large in relation to the complexity of the target concept, it can be concluded that it then will provide an accurate measurement of generalization accuracy. [2]

- **Prequential**: An alternative approach to evaluate data stream algorithm is to interleave testing with training. Each individual set can be used to test the model before it is used for training and then the accuracy can be incremantally updated from this. When it is performed in this order the model will always be tested by sets that it has not seen yet. This has the advantage that no holdout set is needed to perform testing. This also results in a smooth plot of the accuracy over time, as each individual set will become increasingly more insignificant to the overall average. This approach also has its disadvantages, as it make it diffcult to accuratly seperate and measure training and testing times. It also obscures the true accuracy that the algorithm can achieve at agiven point, as the algorithm gets punished for mistakes that it made earlier regardless of the level of accuracy they

are eventually capable of. It also has to be said that this effect will diminish over time. With this approach the statistcs are updated with every set from the stream and can be recorded at this level od detail if desired. To increase the efficiency a sampling parameter can be used to reduce the required storage for the results, by recording only at periodic intervals like the holdout method. [2]

2.3.2 Performance Evaluation

In real data streams, the number of instances for each class may be evolving and changing. It may be argued that the prequential accuracy measure is only appropriate when all classes are balanced and have approximately the same number of examples. The Kappa statistic is a more sensitive measure for quantifying the predictive performance of streaming classifiers. [5] There are 3 different versions of the Kappa statistic that are relevant for the analysis of streaming data. The first Kappa statistc, Kappa κ, was introduced by Cohen in his work [7] and was defined as follows:

$$\kappa = \frac{p_0 - p_c}{1 - p_c} \tag{2.4}$$

The quantity $p0$ is the classifier's prequential accuracy, and p_c is the probability that a chance classifier—one that randomly assigns to each class the same number of examples as the classifier under consideration—makes a correct prediction. If the classifier is always correct, then $\kappa = 1$. If its predictions coincide with the correct ones as often as those of a chance classifier, then $\kappa = 0$ [5]. The second Kappa statistic, κ_m [4], is a measure that compares against a majority class classifier p_m, that is a classifier that stores a count for each of the class labels and predicts as the class of a new instance the most frequent class label, instead of a chance classifier:

$$\kappa_m = \frac{p_0 - p_m}{1 - p_m} \tag{2.5}$$

In cases where the distribution of predicted classes is substantially different from the distribution of the actual classes, the majority class classifier p_m can perform better than a given classifier while the classifier has a positive κ statistic [5]. The third Kappa statistic that exist, is the Kappa temporal statistic [6, 28]. It considers the presence of temporal dependencies in data streams and is defined as:

$$\kappa_{temp} = \frac{p_0 - p'_e}{1 - p'_e}, \tag{2.6}$$

where p'_e is the accuracy of the No-change classifier, that is a classifier that predict the last class in the data stream. It does this by exploiting autocorrelation in the label assignments. It is a simple and useful classifier when the same labels appear together in bursts. Statistic κ_{temp} takes values from 0 to 1. The interpretation is similar to that of κ: if the classifier is perfectly correct, then $\kappa_{temp} = 1$. If the classifier is achieving the same accuracy as the No-change classifier, then $\kappa_{temp} = 0$. Classifiers that outperform the No-change classifier fall between 0 and 1. Sometimes $\kappa_{temp} < 0$, which means that the classifier is performing worse than the No-change baseline [5].

Using κ_{temp} instead of κ_m,we can detect misleading classifier performance for data that is not $\prod D$. For highly imbalanced but independently distributed data, the majority class classifier may beat the No-change classifier. The κ_{temp} and κ_m measures can be seen as orthogonal, since they measure different aspects of the performance [5].

2.3.3 Accuracy Evaluation

Accuracy describes the ability of the classifier to correctly predict the data points in relation to the whole training set, which can be defined as:

$$Accuracy = \frac{Numberofcorrectpredictions}{Totalnumberofpredictions} \tag{2.7}$$

For a binary classification, accuracy can also be calculated in terms of positive and negatives as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{2.8}$$

with TP = True Positives, TN = True Negatives, FP = False Positives and FN = False Negatives. The accuracy is often used to get an measurement of how good the classifier is at returning correct predictions. It should be noted that the accuracy can give a misleading result for cases were ther is a high class imbalance. In this case the classifier may achieve a high accuracy, but because the model is to simple the result is to crude to be of use. A example would be, if the Label for A would occur in 99% of the cases, then the prediction that every case would be A would have a accuracy of 99%. This is called th Accuracy Paradox. For this reason measurements like Kappa, $Kappa_m$ and $Kappa_t emp$ are also required to get useful result of the performeance and accuracy of a classifier [5].

2.4 Framework

The following section will shortly introduce the python based scikit-multiflow framework, with which the the comperssion of the different implementations of the RSLVQ algorithm where done.

Scikit-multiflow is a framework that is inspired by the Massive Online Analysis (MOA), the most popular open source framework for machine learning on data streams, and MEKA, an open source implementation of methods for multi labeling. Scikit-multiflow is also inpired by scikit-learn, the most popular framework for machine learning in Python. Following the Scikit philosophy, scikit-multiflow is an open source machine-learning framework for multi-output/multi-label and stream data.

The Scikit-multiflow does not provide a graphical user interface like the MOA framework that it was inspired by. On the other hand can it be used within Jupyter notebooks, a popular interface in the data scientist community that is based on python. It is mainly a library that contains stream generators, learning methods, change detectors and evaluation methods for Machien learning algorithms on streaming data. Scilit-multiflow lays a special focus on its design, so that it is user friendly to new user and familiar for more experienced ones.[18]

It achieves this by having a simple workflow that can be easily describe in 4 steps:

1. Create a stream:
   A stream can be generate by use of either one of the predefiend Stream generators or by importing a Stream from a file.

2. Instantiate a classifier:
   In this step the user has to choose by which type of algorithm the classifikation process should be accomplished.

3. Setup the evaluator:
   Next the user has to choose if the evaluation process should be done presequential or by Hold-out method. Prequential-evaluation or interleaved-testthen-train evaluation, is a popular performance evaluation method for the stream setting only, where tests are performed on new data before using it to train the model. Hold-out evaluation is a popular performance evaluation method for batch and stream settings, where tests are performed in a separate test set. [18]

4. Run the evaluation:
   The evaluator will perform the following subtask in this final step, first, it will check if there are still samples in the stream and then it will pass the next sample

to the classifier, which will either test the sample with the `predict()` method or it willl update the classefier with the `partial_fit()` method of the clasifier.

To give better understanding how this exactly works, 2.1 will provide a small example where a synthetic waveform gets generated with the WaveformGenerator. This generator will create a synthetic stream that generates default samples with 21 numeric attributes and 3 target values, based on a random differentiation of some base waveforms. For the classifier is the Hoeffding Tree choosen with default parameters. After this we chose a prequential evaluator with the following parameters:

- `show_plot` = true, this will generate us a dynamic plot that will update as the classifier is trained.
- `pretrain_size` = 200, this parameter sets the number of samples that is used for the first training set.
- `max_sample` = 20000, this parameter set the maximum number of samples that should be used.

In the last step we then call evaluator method, that will then update the results and the plot. It is to note that at the time this thesis was written, the accuracy score stand for the performance in the framework. [18]

Listing 2.1: Simple example of the scikit-multiflow workflow

```
1
2  """1. Create a stream"""
3  stream = WaveformGenerator()
4  stream.prepare_for_use()
5
6  """2. Instantiate the HoeffdingTree classifier"""
7  ht = HoeffdingTree()
8
9  """3. Setup the evaluator"""
10 evaluator = EvaluatePrequential(show_plot=True,
11                                 pretrain_size=200,
12                                 max_samples=20000)
13
14 """4. Run evaluation"""
15 evaluator.evaluate(stream=stream, model=ht)
```

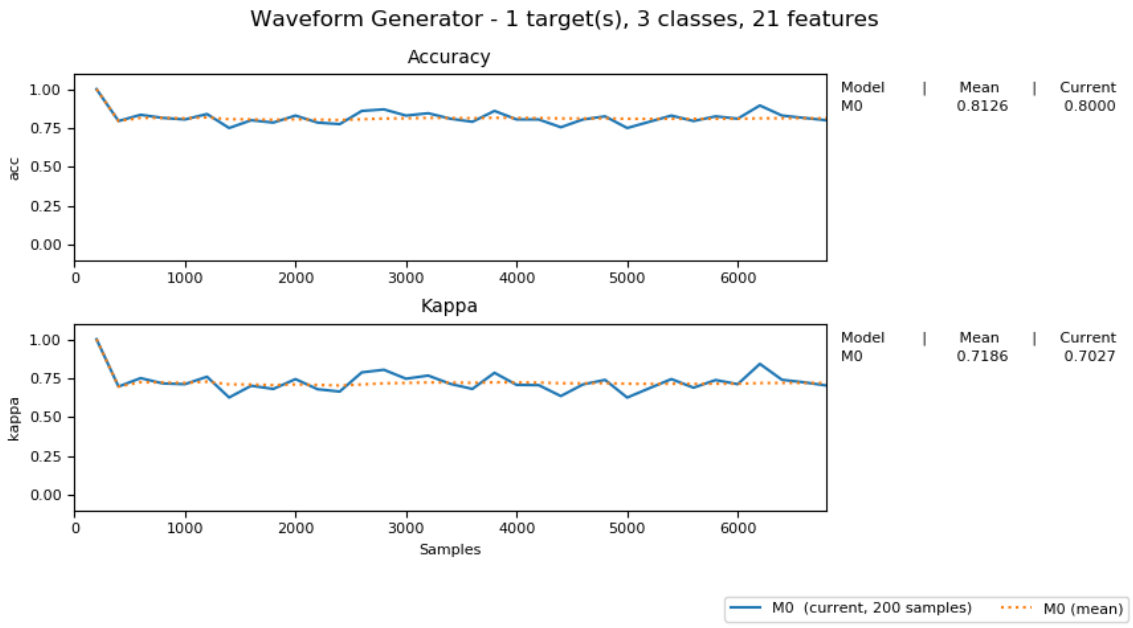

Figure 2.6: Result Table of the example in 2.1

3 Used Algorithms

3.1 Theoretical Work

This chapter will try to give a short explanation about how the Robust Soft Learning Vector Quantization (RSLVQ) works and will also explain the algorithm that it is based on. Furthermore will it explain what momentum based gradient descent algortihms there are and how they work. These are used in the different optimization implementation of the RSLVQ that will be compared in the later chapters.

3.1.1 Learning Vector Quantization

The first part of the theoretical work will give an introduction to the Learning Vector Quantization (LVQ), from that later the RSLVQ was derived from. The LVQ is a class of learning algorithms for nearest prototype classification (NPC).It stores a set of appropriatly chosen prototype vectors instead of storing all the data points of a training set. This change makes the method computationally more efficent, as it does not have to store a high number of items to which new data must be compared to for classifcation anymore. Instead, the current data point only has to be compared to the prototypes to achieve classification. This lead to a high popularity of the LVQ for real time applications, like i.e. speech recognition [16, 17], were it was used with good success.

A NPC consists of a set $\tau = (\theta_j, c_j)_{j=1}^{M}$ of labeled prototype vectors. The parameter $\theta_j \in \chi \equiv \mathbb{R}^D$ are vectors in the data space and $c_j \in \iota$, $i = 1, \ldots, N_y$ are their corresponding labels. Typically, the number of prototypes is larger than the number of classes, such that every classification boundary is determined by more than one prototype. The class y of a new datapoint x is determined

- by selecting the prototype θ_i which is closest to x,

$$y = \min_i d(x, \theta_i), \tag{3.1}$$

where d(.,.) is the distance between data point and prototype, and

- by assigning the label y of this prototype to the datapoint x.

A popular choice for d is the Euclidian distance

$$d(q,p) = \sqrt{\sum_{i\,=\,1}^{n}(q_i - p_i)^2}. \tag{3.2}$$

LVQ is a class of algorithm for the determination of prototype vectors for NPC, which make use of the distribution of the training data as well as their class labels when selecting useful set of prototype vectors. The LVQ 2.1, for example, selects for every pair (x, y), where x is the data point and y is the label, from the training set $S = (x_i, y_i)_{i=1}^{N}$, the two nearest prototypes θ_l, θ_m (see equation 3.1 for selection of one prototype) according to the Euclidian distance (see equation: 3.2). If the labels c_l and c_m are different and if one of them is equal to the label y of the data point, then the two nearest prototypes are adjusted according to:

$$\begin{aligned} \theta_l(t+1) &= \theta_l(t) + \alpha(t)(x - \theta_l), \quad c_l = y, \\ \theta_m(t+1) &= \theta_m(t) - \alpha(t)(x - \theta_m), \quad c_m \neg y. \end{aligned} \tag{3.3}$$

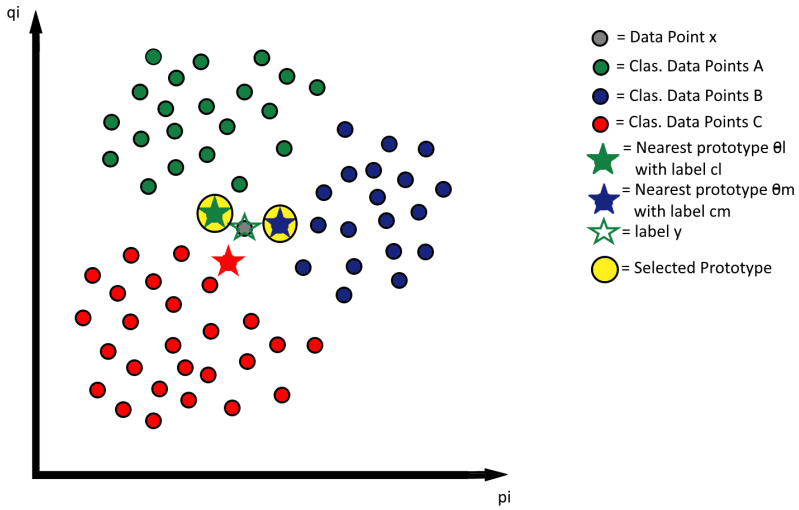

Figure 3.1: Graphic depiction of the LVQ classifcation task.

If the labels c_l and c_m are equal or both labels differ from the label y of the data point, no parameter update is being performed. The prototype, however, are changed are changed only if the data point x is close to the classification boundry, i.e. if it falls into a *window*

$$min(\frac{d(x,\theta_m)}{d(x,\theta_l)}, \frac{d(x,\theta_l)}{d(x,\theta_m)}) > s, \quad wheres = \frac{1-\omega}{1+\omega}, \tag{3.4}$$

of relative width $0 < \omega \leq 1$. This "window rule" had to be introduced, as otherwise the prototype vectors would have diverged. [24]

3.1.2 Robust Soft Learning Vector Quantization

The LVQ algorithm may be a popular adaptive nearest prototype classifier for multiclass classification, but the algortihms from this familiy had only been propsed on heuristic grounds. Seo and Obermayer then derived the Robust Soft Learning Vector Quantization (RSLVQ varient) in their work [24]. It is a LVQ that is based on a Gausian Mixture ansatz. It proposes an objective function which is based on a likelihood ratio and derives the learning rule from gradient descent.

They asume that the following objective function is maximized:

$$\begin{aligned} L_r &= \prod_{k=1}^{N} \frac{p(x_k, y_k \mid \tau)}{p(x_k, y_k \mid \tau) + p(x_k, \bar{y}_k \mid \tau)} \\ &= \prod_{k=1}^{N} \frac{p(x_k, y_k \mid \tau)}{p(x_k \mid \tau)} \stackrel{!}{=} max. \end{aligned} \tag{3.5}$$

The ratio $\frac{p(x,y|\tau)}{p(x|\tau)}$ is bounded by 0 from below and bounded by 1 from above. Because of this, instead of optimizing the likelihood ratio, they then optimize the logarithm of the ratio L_r like this,

$$logL_r = \sum_{k=1}^{N} log \frac{p(x_k, y_k \mid \tau)}{p(x_k \mid \tau)} \stackrel{!}{=} max, \tag{3.6}$$

were Stochastic gradient descent leads to the following learning rule:

$$\theta_l(t+1) = \theta_l(t) + \alpha(t)\frac{\partial}{\partial\theta_l}[log\frac{p(x,y \mid \tau)}{p(x \mid \tau)}], \tag{3.7}$$

where $\alpha(t)$ is learning rate with $\sum_{t=1}^{\infty}\alpha(t) = \infty$ and $\sum_{t=1}^{\infty}\alpha^2(t) < \infty$. If the conditional density functions $p(x \mid j)$ are of the normalized exponential form

$$p(x \mid j) = K(j)expf(x,\theta_j), \tag{3.8}$$

the learning rule becomes

$$\begin{aligned}\frac{\partial}{\partial\theta_l}[log\frac{p(x,y \mid \tau)}{p(x \mid \tau)}] &= \delta(c_l = y)(P_y(l \mid x) - P(l \mid x))\frac{\partial f(x,\theta_l)}{\partial\theta_l} \\ &- \delta(c_l \neq y)P(l \mid x)\frac{\partial f(x,\theta_l)}{\partial\theta_l}.\end{aligned} \tag{3.9}$$

where $P_y(l \mid x)$ and $P(l \mid x)$ are assigment probabilities,

$$\begin{aligned}P_y(l \mid x) &= \frac{p(l)expf(x,\theta_l)}{\sum\limits_{j:c_j=y} p(j)expf(x,\theta_j)}, \\ P(l \mid x) &= \frac{p(l)expf(x,\theta_l)}{\sum\limits_{j=1}^{M} p(j)expf(x,\theta_j)}.\end{aligned} \tag{3.10}$$

$P_y(l \mid x)$ describes the (posterior) probability that the data point x is assigned to the component l of the mixture , given that the data point was generated by the correct class. $P(l \mid x)$ describes the (posterior) probability that the data point x is assigned to the component l of the complete mixture using all classes. Using the gradient given by the equation 3.7, following learning rule is obtained:

$$\theta_l(t+1) = \theta_l(t) + \alpha(t)\begin{cases}P_y(l \mid x) - P(l \mid x)[\frac{\partial f(x,\theta_l)}{\partial\theta_l}], & c_l = y, \\ -P(l \mid x)[\frac{\partial f(x,\theta_l)}{\partial\theta_l}], & c_l \neq y.\end{cases} \tag{3.11}$$

It has to be noted that the factor $P_y(l \mid x) - P(l \mid x)$ is always positive. [24]

The Cost-function and the learning rule described until now were for training the RSLVQ for the general case, but Seo and Obermayer provided also a varient that uses a Gaussian mixture ansatz. In that version they choose a Gaussian mixture model whose components have similar width and strengths, i.e. $\sigma_l = \sigma, \forall l, and\ p(l) = \frac{1}{M}, l = 1, ..., M$. They obtain

$$f(x, \theta_l) = \frac{-(x - \theta_l)^2}{2\sigma^2}, \quad f(x, \theta_l) = \frac{1}{\sigma^2}(x - \theta_l). \tag{3.12}$$

By substituting the derivative of $f(x, \theta_l)$ into equation 3.11 they obtained

$$\theta_l(t+1) = \theta_l(t) + \alpha(t) \begin{cases} P_y(l \mid x) - P(l \mid x)(x - \theta_l), & c_l = y, \\ -P(l \mid x)(x - \theta_l), & c_l \neq y, \end{cases} \tag{3.13}$$

where $\alpha(t) = \frac{\alpha(t)}{\sigma^2}$ and

$$\begin{aligned} P_y(l \mid x) &= \frac{exp\frac{-(x-\theta_l)^2}{2\sigma^2}}{\sum_{j:c_j=y} exp\frac{-(x-\theta_j)^2}{2\sigma^2}}, \\ P(l \mid x) &= \frac{exp\frac{-(x-\theta_l)^2}{2\sigma^2}}{\sum_{j=1}^{M} exp\frac{-(x-\theta_j)^2}{2\sigma^2}}. \end{aligned} \tag{3.14}$$

The equations 3.13 and 3.14 implement the RSLVQ algorithm. Prototypes whose label is equal to the label of the data point are attracted, while prototypes with different labels are repelled. Using deterministic annealing of σ^2, which is a hyperparameter of the learning rule described in 3.13, the RSLVQ can be optimized. If the width σ of the Gaussian components goes to zero, both assignment probabilities (equation 3.14), become hard assignments. If th label c_q of the nearest prototype q is equal to the label y of the data point x then no update is being performed, because

$$P(l \mid x) = P_y(l \mid x) = 0, \quad \forall l \neq q, \quad P_y(q \mid x) - P(q \mid x) = 0. \tag{3.15}$$

Only if the label of the data point x differs from the label of the nearest prototype q, then

$$\begin{aligned} P(l \mid x) = 0, \forall l \neq q, \qquad & P_y(l \mid x) = 0, \forall l \neq q', \\ P(q \mid x) = 1, \qquad & P_y(q \mid x) = 1, \end{aligned} \tag{3.16}$$

where q' is the nearest prototype vector with the correct class label, and the nearest prototype vector and the nearest correct prototypevector are changed. In the "hard" version of the RSLVQ, prototypes are modified only by the data points that are not correctly classified. [24]

Seo and Obermayer then showed in their work [24] for LVQ 2.1 data points which are near the current class boundary and which are classified correctly have the strongest effect on the update of the prototypes. In contrast to that relies the RSLVQ on data points which are further away from the classification boundary and which are incorrectly classified, i.e. RSLVQ learns from mistakes. Additionaly, the RSLVQ does not require a "window rule" like the LVQ 2.1, as the prototypes do not diverge. [24]

3.2 Momentum Based Gradient Descent

The following section will outline three algorithms that are widely used in the Deep learning Community and that will later be used to optimize the RSLVQ Algorithm in hopes to improve its performence and accuracy.

3.2.1 Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another [26, 23], which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in figure.

Figure 3.2: Differences of a SGD with and without momentum.[1]

Momentum [21, 23] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in figure . It does this by adding a fraction γ of the update vector of the past time step to the current update vector

$$\begin{aligned} v_t &= \gamma v_{t\text{-}1} + \eta\theta - \eta\nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{3.17}$$

The momentum term γ is usualy set to 0.9 or a similar value.

In his Article, Ruder [23] describs it like if we would push a ball down a hill. The ball accumulates momentum as long as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity, if there is air resistance, i.e. γ ¡ 1). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

3.2.2 Adagrad

The first algorithm that will be described is the Adagrad [27]. It has to be noted that this algorithm will not be part of the comparision, but as this algorithm is the base from wich the Adadelta and later the RMSprop were developed from, it will be shortly explained nonteless. The Adagrad is an algorithm for gradient-based optimization that adapts to updates for each individual parameter to perform larger or smaller updates depending on their importance. The learning rate gets adapted to the parameters, were it performs larger updates if the parameters are more infrequent and smaller updates if they are frequent. Hence, it is well-suited to deal with sparse data. As other works

[1] Genevieve B. Orr

[8] have shown, this greatly improves the robustness of SGD, which lead to its wide use to train large Neural Networks. It was also used to train GloVe word embeddings as infrequent words require much larger updates than frequent ones [20, 23].

For the reason that Adagrad uses a different learning rate for every parameter θ_i at every time step t, Adagrad's per-parameter update is first shown and then vectorized. For brevity, we set $g_{t,\,i}$ to be the gradient of the objective function w.r.t. the parameter θ_i at the time step t:

$$g_{t,\,i} = \nabla_{\theta\ t} J(\theta_{t,\,i}) \tag{3.18}$$

The update for every SGD parameter θ_i at time t becomes to:

$$\theta_{t+1,\,i} = \theta_{t,\,i} - \eta \cdot g_{t,\,i} \tag{3.19}$$

In its update rule, Adagrad modifies the general learning rate η each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,\,i} = \theta_{t,\,i} - \frac{\eta}{G_{t,\,ii} - \epsilon} g_{t,\,i} \tag{3.20}$$

$G_t \in \mathbb{R}^{dxd}$ here is a diagonal matrix where each diagonal element $i,\ i$ is the sum of the squares of the gradients w.r.t. θ_i up to time step t [9], while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e$ - 8).

As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters θ along its diagonal, we can now vectorize our implementation by performing an element-wise matrix-vector multiplication $\odot$ between G_t and g_t:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{3.21}$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that. Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. [23]

3.2.3 Adadelta

One of the more often used momentum-based algorithm is the Adadelta approach. It is an extension of the Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size ω.

Rather than inefficently storeing ω previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t\text{-}1} + (1-\gamma) g_t^2 \tag{3.22}$$

We then set γ to a simliar value as the momentum, around 0.9. The SGD update will then be rewriten in terms of the parameter update vector $\Delta\theta_t$:

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t - \Delta\theta_t \end{aligned} \tag{3.23}$$

Because of this, the parameter update vector, that is base of the parameter update vector of the Adagrad (see equation 3.21), takes the form of:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{3.24}$$

In the next step the diagnoal Matrix G_t gets replaced with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{3.25}$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \tag{3.26}$$

As in [9] noted, the units in this update (as well as in the SGD, Momentum and Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another ecaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_t - 1 + (1-\gamma)\Delta\theta_t^2 \tag{3.27}$$

This leads to that the root mean squared error of the parameter updates is:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} \tag{3.28}$$

Because the $RMS[\Delta\theta]_t$ value is unknown to us, we try to approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t\text{-}1}$ leads to the final update rule for the Adadelta:

$$\begin{aligned} \Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t\text{-}1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{3.29}$$

In the Adadelta we do not need to set a learning rate η, as it was eliminated from the update rule. [23]

3.2.4 RMSprop

RMSprop is an adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class. [14] As a result of the need to resolve the Adagrad's radically diminishing learning rates, both the Adadelta and the RMSprop had been developed independetly from each other at the same time. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

$$E[g^2]_t = 0.9E[g^2]_{t\text{-}1} + 0.1g_t^2 \tag{3.30}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{RMS[g]_t} g_t \tag{3.31}$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001. [23]

3.2.5 Adam

As described in [15], the Adaptive Moment Estimation (ADAM) is another way to compute the adaptive learning rate for each parameter. Additionly to storing an exponentially decaying average of past squared gradients v_t like Adadelta or RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, simliar to momentum:

$$\begin{aligned} m_t &= \beta_1 m_{t\text{-}1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t\text{-}1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{3.32}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

To counteract these biases they compute a bias-corrected estimate of the first and second moment:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned} \tag{3.33}$$

These are then used to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.34}$$

The authors recommend in their work to use as default values of 0.9 for β_1, 0.999 for β_2

and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms. [23, 15]

Algorithm 1 Adam algorithm, as proposed in [15]. The authors recommend for default settings the value 0.9 for β_1, 0.999 for β_2 and 10^{-8} for ϵ as well as a stepsize α of 0.001.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
Initialize first moment vector: $m_0 = 0$
Initialize second moment vector: $v_0 = 0$
Initialize timestep: $t = 0$
while θ_t not converged **do**
t = t + 1
Get the gradients at timestamp t : $g_t \leftarrow \nabla_{\theta\ t} J(\theta_{t\text{-}1})$ (see equation 3.18)
Update first moment estimate: $m_t \leftarrow \beta_1 m_{t\text{-}1} + (1 - \beta_1) g_t$
Update second moment estimate: $v_t \leftarrow \beta_2 v_{t\text{-}1} + (1 - \beta_2) g_t^2$
Compute bias-corrected first moment estimate: $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
Compute bias-corrected second moment estimate: $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
Update the parameters: $\theta_{t+1} \leftarrow \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$

return Resulting parameters: θ_t

3.3 Implementation of the RSLVQ variants

THis chapter will show how the different RSLVQ varients where implemented, so that they can work with Streaming data. The implementation is based on the version that was used in the scikit-multiflow framework [18] and was then further modiefied by Moritz Heusinger in [13]. It has to be said that the implementation is nearly the same as in the previously mentioned work [13].

3.3.1 RSLVQ SGD

The main differences between the different implementations of the algorithms appear in the training phase, when the prototypes gets updated in the `_update_prototype` method. As the rest stays nearly the same, it will only be explained once during the explanation of how the $RSLVQ_{SGD}$ varient was implemented. In the scikit-multiflow framework the `partial_fit` method, shown in listing 3.1, gets executet everytime when an algorithm initiates the learning process or when it continous training process. It will

check if the algorithm has already completed the intial learning process or if it is the first time that the method gets executed. If one of those 2 cases is true, the method will continue by calling the internal `_validate_train_parms` method, that will then carry out further checks. If none of the 2 options is true, the method will raise an error that the algorithm has to intial the learning/training process. After this check, the method will call the `_optimize` method, that is shown in listing 3.2. It has to be noted that neither the `partial_fit`, the `_validate_train_parms` and the `_optimize` methods have been altered from how they have been implemented in the scikit-multiflow framework [18].

Listing 3.1: Method `partial_fit` from the scikit-multiflow framework

```
1
2  def partial_fit(self, X, y, classes=None, sample_weight=None):
3        """Fit the LVQ model to the given training data and parameters
    using
4        gradient ascent.
5
6        Parameters
7        ----------
8        X : array-like, shape = [n_samples, n_features]
9            Training vector, where n_samples in the number of samples
    and
10           n_features is the number of features.
11       y : numpy.ndarray of shape (n_samples, n_targets)
12           An array-like with the class labels of all samples in X
13       classes : numpy.ndarray, optional (default=None)
14           Contains all possible/known class labels. Usage varies
    depending
15           on the learning method.
16       sample_weight : Not used.
17
18       Returns
19       --------
20       self
21       """
22       if set(unique_labels(y)).issubset(set(self.classes_)) or
    self.initial_fit is True:
23           X, y = self._validate_train_parms(X, y, classes=classes)
24       else:
25           raise ValueError('Class {} was not learned - please declare
    all classes in first call of fit/partial_fit'.format(y))
26
27       self._optimize(X, y)
28       return self
```

The _optimize method will search for the the nearest correct and the nearest wrong prototype, based on their euclidiean distance towards the data point that has to be classified, as describe in 3.3 and visualized in 3.1. If the the correct prototype(prototype with the right label) is not the nearest to the data point, then each the model of the nearest wrong prototype as well as the model of the nearest correct prototype should be updated by calling the _update_prototype method

Listing 3.2: Method _optimize from the scikit-multiflow framework

```
1
2  def _optimize(self, X, y):
3        nb_prototypes = self.c_w_.size
4
5        n_data, n_dim = X.shape
6        prototypes = self.w_.reshape(nb_prototypes, n_dim)
7
8        for i in range(n_data):
9            xi = X[i]
10           c_xi = int(y[i])
11           best_euclid_corr = np.inf
12           best_euclid_incorr = np.inf
13
14           # find nearest correct and nearest wrong prototype
15           for j in range(prototypes.shape[0]):
16               if self.c_w_[j] == c_xi:
17                   eucl_dis = euclidean_distances(xi.reshape(1,
   xi.size),
18                                                  prototypes[j]
19                                                  .reshape(1,
   prototypes[j]
20                                                  .size))
21                   if eucl_dis < best_euclid_corr:
22                       best_euclid_corr = eucl_dis
23                       corr_index = j
24               else:
25                   eucl_dis = euclidean_distances(xi.reshape(1,
   xi.size),
26                                                  prototypes[j]
27                                                  .reshape(1,
   prototypes[j]
28                                                  .size))
29                   if eucl_dis < best_euclid_incorr:
30                       best_euclid_incorr = eucl_dis
31                       incorr_index = j
32
```

```
33              # Update nearest wrong prototype and nearest correct
    prototype
34              # if correct prototype isn't the nearest
35              if best_euclid_incorr < best_euclid_corr:
36                  self._update_prototype(j=corr_index, c_xi=c_xi, xi=xi,
37                                         prototypes=prototypes)
38                  self._update_prototype(j=incorr_index, c_xi=c_xi, xi=xi,
39                                         prototypes=prototypes)
```

The `_update_prototype`, as shown as in listing 3.3, is when the first real difference to the scikit implementation appears. The method will then compare if the prototype has the same label yi (here named as c_{xi}) as the data point xi. It will be then adjusted according to equation 3.13. $P_y(l \mid x)$ and $P(l \mid x)$ from 3.14 are then calculated by the `_p` method. It uses the internal method `_costf` to calculate the cost for each of the prototypes and it represents the $\frac{-(x-\theta_l)^2}{2\sigma^2}$ part of equation 3.14. Both the `_p` and the `_costf` method are unchanged from their implementation in [18]

Listing 3.3: Method `_update_prototype` as it was used in [13].

```
1
2   def _update_prototype(self, j, xi, c_xi, prototypes):
3         """SGD"""
4         d = xi - prototypes[j]
5
6         if self.c_w_[j] == c_xi:
7             # Attract prototype to data point
8             self.w_[j] += self.learning_rate * (self._p(j, xi,
    prototypes=self.w_, y=c_xi) -  self._p(j, xi, prototypes=self.w_)) *
    d
9         else:
10            # Distance prototype from data point
11            self.w_[j] -= self.learning_rate * self._p(j, xi,
    prototypes=self.w_) * d
```

3.3.2 RSLVQ Adadelta

The first steps of the Adadelta implementation are the same as in the SGD implementation. In the first step the method will again calculate the adjustment for the prototype like it was described in 3.13. It will also calculate the Posterior and prior in the same way as described in 3.14 with the `_p` and `_costf` methods. In next step is now were the differences between the implementation of the SGD and Adadelta start. First, it will calculate the sum of the past squared gradients according to equation 3.22. It will then

calculate the gradient update at step t according to the first equation from 3.29. As the $RMS[\Delta\theta]_t$ value is unknown to us, we try to approximate it with the RMS of parameter updates until the previous time step like this: $E[\Delta\theta^2] + \epsilon$. This gives us:

$$\Delta\theta_t = \frac{(E[\Delta\theta^2] + \epsilon)}{\sqrt{(E[\Delta\theta^2] + \epsilon)}} g_t \tag{3.35}$$

Next we store the squared parameter updates according to equation 3.27. Finally we apply the update prototype accoring to the second equation from 3.29:

$$\theta_{t+1} = \theta_t + \Delta\theta_t. \tag{3.36}$$

This implementation of the Adadelta was also used in [13].

Listing 3.4: Adadelta implementation from [13].

```
"""Implementation of Adadelta"""
      d = (xi - prototypes[j])

      if self.c_w_[j] == c_xi:
          gradient = (self._p(j, xi, prototypes=self.w_, y=c_xi) -
 self._p(j, xi, prototypes=self.w_)) * d
      else:
          gradient = - self._p(j, xi, prototypes=self.w_) * d

      # Accumulate past squared gradients
      self.squared_mean_gradient[j] = self.decay_rate
 *self.squared_mean_gradient[j] + (1 - self.decay_rate) * gradient **
 2

      # Compute update/step
      step = ((self.squared_mean_step[j] + self.epsilon) /
 (self.squared_mean_gradient[j] + self.epsilon)) **0.5 * gradient

      # Accumulate updates
      self.squared_mean_step[j] = self.decay_rate *
 self.squared_mean_step[j] + (1 - self.decay_rate) * step ** 2

      # Attract/Distract prototype to/from data point
      self.w_[j] += step
```

3.3.3 RSLVQ RMSprop

The implementation of the RMSprop is nearly the same as for the Adadelta, the difference is that we reintroduce the learning rate η, that was eliminated in the Adadelta and replaced by the root mean squared error of the parameter updates $RMS[\Delta\theta]_{t\text{-}1}$.

The steps until after the posterior and prior calculation are the exact same as in the SGD implementation. When the gradient is calculated, it will be added to the averge past squared gradients like in equation 3.30 described. If we take the value 0.9 for γ like Hinton suggested in [14], we get the following equation:

$$G_t = 0.9E[g^2]_{t\text{-}1} + 0.1g_t^2 \tag{3.37}$$

The prototype update can be calculated by equation 3.31, were the $-\frac{\eta}{RMS[g]_t}g_t$ part, that calculates the gradient update, can be substituted with $\Delta\theta$. Because the RMSprop is identical to the first update vector of the Adadelta, we will then get for $\Delta\theta$ the following eqaution:

$$\Delta\theta = -\frac{\eta}{G_t - \epsilon}g_t. \tag{3.38}$$

This equation is then used to calculate the gradient update before it is update is performed on the prototype in the final implementation [13].

Listing 3.5: Implementation of the RMSprop from [13].

```
1
2   """RMSprop"""
3         d = xi - prototypes[j]
4
5         if self.c_w_[j] == c_xi:
6             gradient = (self._p(j, xi, prototypes=self.w_, y=c_xi) -
     self._p(j, xi, prototypes=self.w_)) * d
7         else:
8             gradient = - self._p(j, xi, prototypes=self.w_) * d
9
10        # Accumulate gradient
11        self.squared_mean_gradient[j] = 0.9 *
     self.squared_mean_gradient[j] + 0.1 * gradient ** 2
12
13        # Update Prototype
```

```
14          self.w_[j] += (self.learning_rate /
    ((self.squared_mean_gradient[j] + self.epsilon) ** 0.5)) * gradient
```

3.3.4 RSLVQ Adam

As describe in chapter 3.2.5, the Adam implementation is simliar to the Adadelta and RMSprop implementation. This means for the implementation, that at the start it does again the same steps the SGD algorithm by computing the gradient. When the gradient is computed it continous by computing the average of past gradients m_t and the average of past squared gradients v_t like in equation 3.32. It has to be noted the parameters m_t and v_t needs to be initialized as vectors of 0's before the omputation process. This is done by modifying the `_optimize` method in the end like this:

Listing 3.6: Modification of the `_optimize` method to initialize m_t and v_t.

```
1
2   if self.initial_fit:
3     # Next two lines are Init for ADAM last m + v gradients
4     self.gradients_v_sqrt = np.zeros_like(self.w_)
5     self.gradients_m = np.zeros_like(self.w_)
6     self.initial_fit = False
```

In the next step will be then the bias corrected estimates for m_t and v_t, $\hat{m}_t$ and $\hat{v}_t$, like in equation 3.33. In the last step will then the update rule be applyied. Implemented is this rule like equation 3.34 suggested. The values β_1 and β_2 are set to 0.9 and 0.999 and ϵ is set to 10^{-8}, like it was suggested in [23, 15].

Listing 3.7: Implementation of Adam on basis of [13].

```
1
2   """Adam"""
3         d = (xi - prototypes[j])
4
5         """Calculate posterior"""
6         if self.c_w_[j] == c_xi:
7             gradient = (self._p(j, xi, prototypes=self.w_,y=c_xi) -
     self._p(j, xi, prototypes=self.w_))* d
8         else:
9             gradient = - self._p(j, xi, prototypes=self.w_)* d
10
11        """Compute gradients m """
12        self.gradients_m[j] = self.beta_1 * self.gradients_m[j] + (1 -
     self.beta_1) * gradient
```

```
13
14          """Compute squared gradients v """
15          self.gradients_v_sqrt[j] = self.beta_2 *
    self.gradients_v_sqrt[j] + (1 - self.beta_2) * gradient ** 2
16
17          """Compute m correction"""
18          m_corrected = self.gradients_m[j] / (1 - self.beta_1)
19
20          """Compute v correction"""
21          v_corrected = self.gradients_v_sqrt[j] / (1 - self.beta_2)
22
23          """Update prototype"""
24          self.w_[j] += - (self.learning_rate/(np.sqrt(v_corrected) +
    self.epsilon ))*m_corrected
```

4 Test realisation

4.1 Used Streaming Datat sets

In this section will be a short describtion of the data streams, synthetic and real world data, that where used to perform the experiments.

4.1.1 Synthetic data

- Sea Concepts: !not finsihed rework required!
  This dataset consists of 50000 instances with three attributes of which only two are relevant. The two class decision boundary is given by f1 + f2 = b, where f1, f2 are the two relevant features and b a predefined threshold. Abrupt drift is simulated with four different concepts, by changing the value of b every 12500 samples. Also included are 10% of noise. [25]

- Moving Squares:
  placeholder

4.1.2 Real world data

- Weather Dataset:
  This dataset contains eight different features like the minimum and maximum tempreature, visibility, wind speed, air pressure, etc. that were measured at the Offutt Air Force Base in Bellevue, Nebraska. These measuredments were measured in the period of 1949-1999. The goal of this dataset is to predict if it is going to rain on certains day or not. It contains 18159 instances with an imbalance towards no rain (69%) and was introduced by Elwell et al..

- Electicity Market Dataset:
  First described by Harris et al., it is often used for performance comparison, as

it is a good benchmark for concept drift classification. It holds the information of the Australian New South Wales Electricity Market, whose prices are affected by supply and demand. Each sample is characterized by teh following attributes: day of the week, time stamp, market demand, etc.. Each of the samples refers to a period of 30 minutes and the class label identifies the relative change compared to the last 24 hours. The used Dataset is the normalized version of the original dataset.

- Poker Dataset:
  To create this dataset, one million poker hands, each represented by five cards that got encoded with its suit and rank, were randomly drawn. The class is the resulting poker hand itself such as one pair, full house and so forth. Since the poker hand can not change by definition and each instance was randomly generated, this dataset had in its original form no drift in it. As this is the version as it was introduced by [3], virtual drift is present in this dataset. This was achieved by sorting the instances of the dataset by rank and suit. Duplicate hands were also removed. Additionly, this version was also normalized.

- Outdoor Objects:
  This Dataset was optained by images that were recorded by a mobile in a garden enviroment. The goal is to classify 40 different objects, each approached ten times under varying lighting conditions affecting the color based representation. Every ten images represent one approach and each approach is in temporal order within the dataset. The objects are encoded in a normalized 21-dimensional RG-Chromaticity histogram.

4.2 Implementation of the Comparision and Results

4.2.1 Experiment begrenzung

4.2.2 Experiment aufbau

4.2.3 Experiment durchlauf

Listing 4.1: Beispiel für einen Quelltext

```
1
2 public void foo() {
3     // Kommentar
4 }
```

5 Result Evaluation

6 Summary

Abbreviations

ADAM Adaptive Moment Estimation

LVQ Learning Vector Quantization

MOA Massive Online Analysis

NPC nearest prototype classification

RMS root mean squared

RSLVQ Robust Soft Learning Vector Quantization

SGD Stocastic Gradient Descent

List of Figures

List of Tables

Bibliography

[1] Charu C Aggarwal. *Data streams: models and algorithms*. Vol. 31. Springer Science & Business Media, 2007.

[2] Albert Bifet, Richard Kirkby, and Bernhardt Pfahringer. *DATA STREAM MINING A Practical Approach*. MIT Press, 2011.

[3] Albert Bifet et al. "Efficient data stream classification via probabilistic adaptive windows". In: *Proceedings of the 28th annual ACM symposium on applied computing*. 2013, pp. 801–806.

[4] Albert Bifet et al. "Efficient online evaluation of big data stream classifiers". In: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015, pp. 59–68.

[5] Albert Bifet et al. *Machine Learning for Data Streams with Practical Examples in MOA*. `https://moa.cms.waikato.ac.nz/book/`. MIT Press, 2018.

[6] Albert Bifet et al. "Pitfalls in benchmarking data stream classification and how to avoid them". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2013, pp. 465–479.

[7] Jacob Cohen. "A coefficient of agreement for nominal scales". In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46.

[8] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[9] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Mach. Learn. Res.* 12.null (July 2011), pp. 2121–2159. ISSN: 1532-4435.

[10] Wolfgang Ertel. *Introduction to Artifical Intelligence*. Springer, 2003. ISBN: 9780857292995.

[11] João Gama et al. "A Survey on Concept Drift Adaptation". In: *ACM Comput. Surv.* 46.4 (Mar. 2014). ISSN: 0360-0300. DOI: `10.1145/2523813`. URL: `https://doi.org/10.1145/2523813`.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[13] Moritz Heusinger, Christoph Raab, and Frank-Michael Schleif. "Passive Concept Drift Handling via Momentum Based Robust Soft Learning Vector Quantization". In: *Advances in Self-Organizing Maps, Learning Vector Quantization, Clustering and Data Visualization*. Ed. by Alfredo Vellido et al. Cham: Springer International Publishing, 2020, pp. 200–209. ISBN: 978-3-030-19642-4.

[14] Geoff Hinton. "Overview of mini--batch gradient descent". In: (). URL: 2http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[15] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014).

[16] Takashi Komori and Shigeru Katagiri. "Application of a generalized probabilistic descent method to dynamic time warping-based speech recognition". In: *[Proceedings] ICASSP-92: 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 1. IEEE. 1992, pp. 497–500.

[17] Erik McDermott and Shigeru Katagiri. "Prototype-based minimum classification error/generalized probabilistic descent training for various speech units". In: *Computer Speech & Language* 8.4 (1994), pp. 351–368.

[18] Jacob Montiel et al. "Scikit-Multiflow: A Multi-output Streaming Framework". In: *Journal of Machine Learning Research* 19.72 (2018), pp. 1–5. URL: http://jmlr.org/papers/v19/18-251.html.

[19] Nils J. Nilsson. *Introduction to Machine Learning*. 1998.

[20] Jeffrey Pennington, Richard Socher, and Christopher D Manning. "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[21] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks* 12.1 (1999), pp. 145–151. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6. URL: http://www.sciencedirect.com/science/article/pii/S0893608098001166.

[22] Sebastian Raschka. *Python MAchine Learning*. Packt Publishing, 2015. ISBN: 1783555130.

[23] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016.

[24] Sambu Seo and Klaus Obermayer. "Soft learning vector quantization". In: *Neural computation* 15.7 (2003), pp. 1589–1604.

[25] W. Nick Street and YongSeog Kim. "A Streaming Ensemble Algorithm (SEA) for Large-Scale Classification". In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '01. San Francisco, California: Association for Computing Machinery, 2001, pp. 377–382. ISBN: 158113391X. DOI: 10.1145/502512.502568. URL: https://doi.org/10.1145/502512.502568.

[26] Richard S. Sutton. *Two problems with backpropagation and other steepest-descent learning procedures for networks*. 1986.

[27] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *ArXiv* abs/1212.5701 (2012).

[28] Indrė Žliobaitė et al. "Evaluation methods and decision theory for classification of streaming data with temporal dependence". In: *Machine Learning* 98.3 (2015), pp. 455–482.

Statutory Declaration

Hiermit versichere ich, dass ich die vorgelegte Bachelorarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Florian Hohn, am February 13, 2020