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

There is an ever growing amount of Data in our everday lives that are produced by sensors when connected to the Internet of Things (IoT), used in self-driving cars or used to recognise our speech to take commands from us (e.g. Amazons Alexa or Apples Siri). But to be of use for us this Data first needs to be analyzied, and this becomes a real challenge, as traditinoal Machine Learning algorithm normaly take several runs through a training set before the learning process is complete. Thankfully there are already several Algorithm developed that can train and learn on data by only needing one run on these datasets and can therefor process this streamed Data. Moritz Heusinger looked already into different varients of how a Robust Soft Learning Vector Quantization (RSLVQ) algorithm can be modified to work on streaming data in [16] and compared them. This work will compare an additional varient of how the RSLVQ can be implemented to work on streaming data and will it compare to the other three varients of the RSLVQ in terms of accuracy, performance and computation time.

Note of thanks

I want to thank my familiy for always supporting me through the whole time I studied and worked on this thesis as well as the time I needed to finish my Bachelor of Engineering degree in 'Information Systems'. I also want to thank Prof. Schleif and Mr Heusinger, as I could always ask them when I was not sure about something while working on this thesis and they would then try to help me with my questions.

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 achieve, as well as an explanation how this work is structured, to give the reader a better oriantation for how to navigate the thesis.

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. from sensors that monitor processes in the industry to improve their quality [6], from data streams originating on the Internet that can be used to detect epidemics and natural disasters, which can be combined with statistics from official centers for disease and disaster control for better prevention [6] or for the use in selfdriving cars. 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 [6].

Heusinger compared for this reason several varients of how the RSLVQ Machine learning algorithm could be modified to make it work with stremed data in [16]. This work will now take the $RSLVQ_{Adam}$ that was suggested in [28] as possible modification for the RSLVQ and will compare it with the $RSLVQ_{SGD}$, $RSLVQ_{Adadelta}$ and the $RSLVQ_{RMSprop}$ varients, that Heusinger already had compared with each other in [16], in respect to their accuracy, performance and computation time.

1.2 Purpose of this Thesis

In [16] 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 paper. 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 [19] 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. This paper will provide a short describtion about what is special about Data stream in Machine learning algorithms and what is importent 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 the framework will be introduced which was used for implementation and comparision. The next chapter will then continue to give a deeper look into the more complex 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 importent to later understand the variations and differences in the other implementations. The last part of this chapter will then explain the differnet methods and the implementations of the variants 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 there will be a description how the experiments where conductet and with wich methods the results where controlled. The following chapter will then be dedicated to the evaluation of the results. The last chapter will summarize all the work that was done and give a final evaluation of 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, which 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 also be 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 an 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 a lot faster and more efficiently than a humans, especially if this optimization has to be done in real time on big data input sets. This gives also a bigger flexability, as a machine that can learn from a changing enviroment needs lesser redesigns, which leads to less time that gets wastet and can be used more productive on other tasks.

All these are reasons why Machine Learning has an ever increasing importance 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

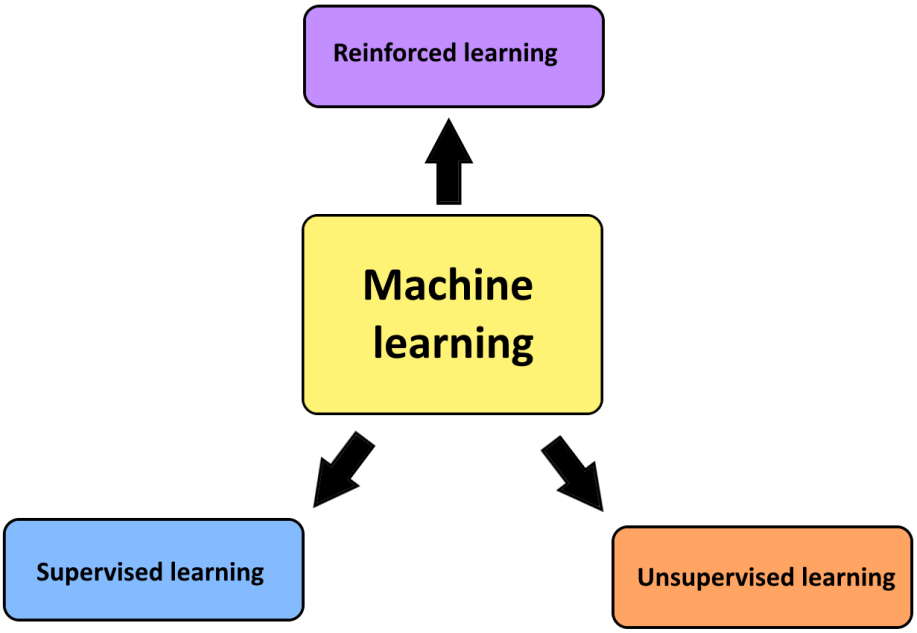

Figure 2.1: Overview of the Machine Learning Strategy's.

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 algorithms that are based on Supervised Learning [24].

2.1.1 Supervised Learning

The idea behind Supervised learning is to create a model from an already labeled training data set, that then can make predictions and decisions 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 *classification 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 classifies them. Based on this by hand sorted training set the machine will then later decide how the other fruits should be sorted. It is used as a *model* for the classification. [12] 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. [27]

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 $f : \mathbb{R} \to 1, ..., k$. 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. [15]

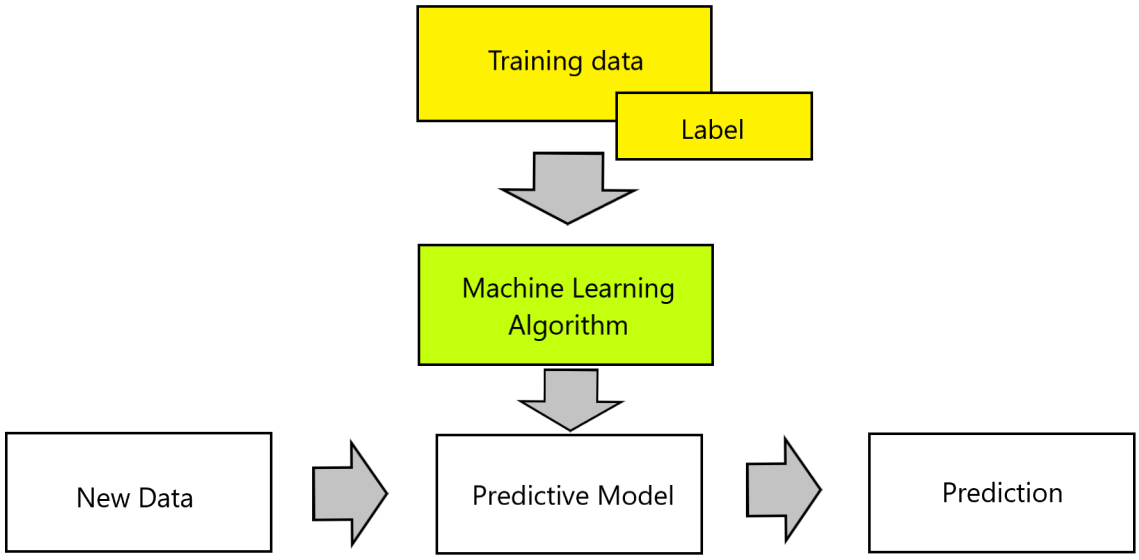

Figure 2.2: Process of the Supervised Learning. [27]

Then there must also be a diffenciation 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 from big data sets, like it is in this thesis, this separation between training, evaluating, and testing is far less clear-cut, as the algorithm has to work with the data that it receives right now and often can only use once, before new daat arrives that need processing. There is a need to start making predictions before we have all the data, because the data that arrives never ends. The data whose label was predicted needs to be keept to train the model, if possible. The model also needs to continuously be evaluated in some way to decide if it needs more or less aggressive retraining. [6]

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 sort of black-box, because a practical explanation of what their stengths and weaknesses is are hard to 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 we take to reach a local minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley. [28]

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 datasets 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 performs 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

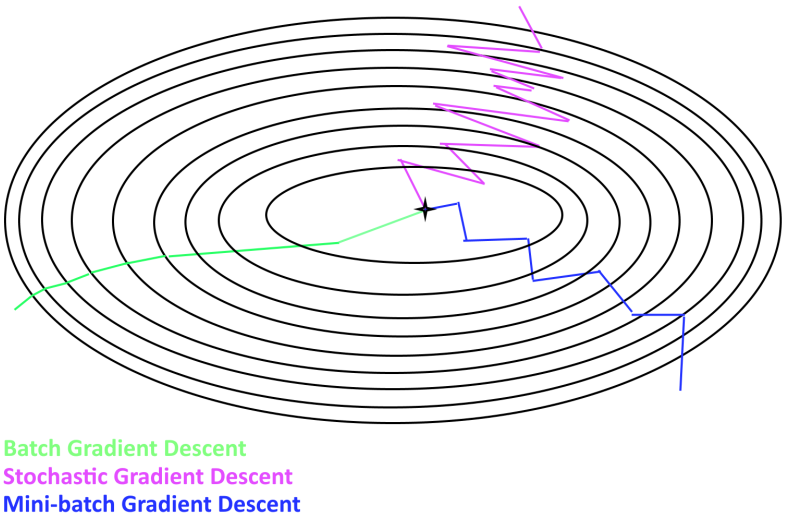

Figure 2.3: Visualisation differences of the three Gradient Descent Algorithms

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 respectively. [28] 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 mini-batches are used. [28]

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. [6]

2.2.1 Increasing Volume of the Data

When the Data items keep arriving fast and in large quantatiy an 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]. [6] describes five requirements that an algorithm has to fullfill, if it should be able to process streaming data successfull:

- Process an instance at a time, and inspect it (at least) 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)[6].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 [6].

The term Concept drift will now be descripted 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 each time generates an item according to a probability distribution which is used at that exact time, and that may or may not be the same as the one which 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 [6].

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 [6].

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 [6].

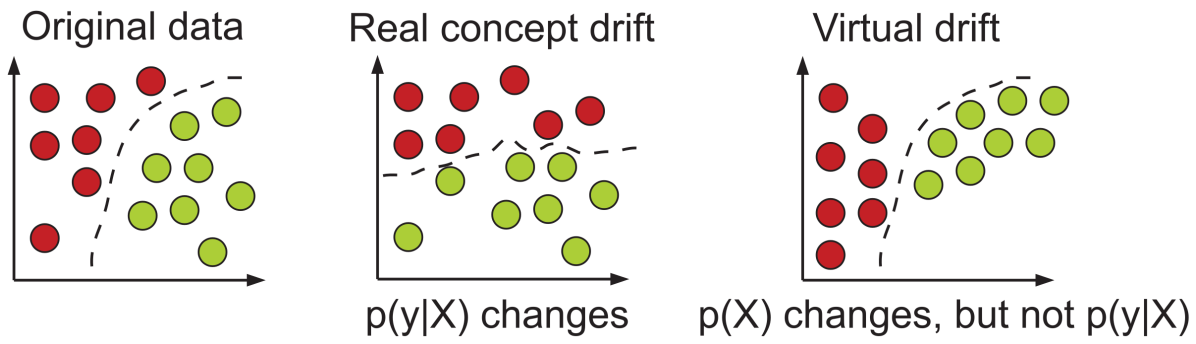

Figure 2.4: Visualization of types of Concept drift [14]

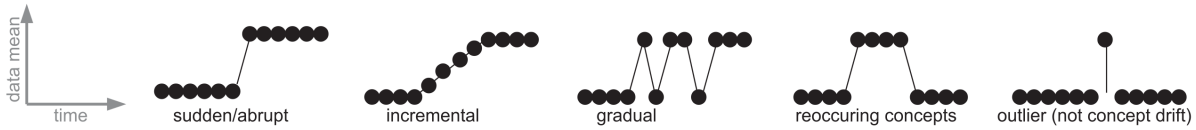

Figure 2.5: Visualization of real and virtual drift [14]

[14]

2.3 Criteria for the Comparision

This section will explain on what criteria this thesis will base the comparsion of the different implementations. It will also descirbed what the difference between the criteria is and why this is importent 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. [3]
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 predefined. 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 algorithm 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. [3]

- **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. [3]

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. [6] 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 [8] 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 $\kappa = 0$ [6]. The second Kappa statistic, κ_m [5], 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 [6]. The third Kappa statistic that exist, is the Kappa temporal statistic [7, 33]. 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 [6].

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 [6].

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 $Kappa_temp$ are also required to get useful result of the performeance and accuracy of a classifier [6].

2.4 Framework

The following section will shortly introduce the python based scikit-multiflow framework, with which the the comperison 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 more user friendly to new user and familiar for more experienced ones.[23]

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

1. Create a stream:
   A stream can be generate by use of either one of the predefined 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 up 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. [23]

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 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 [23].

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)
```

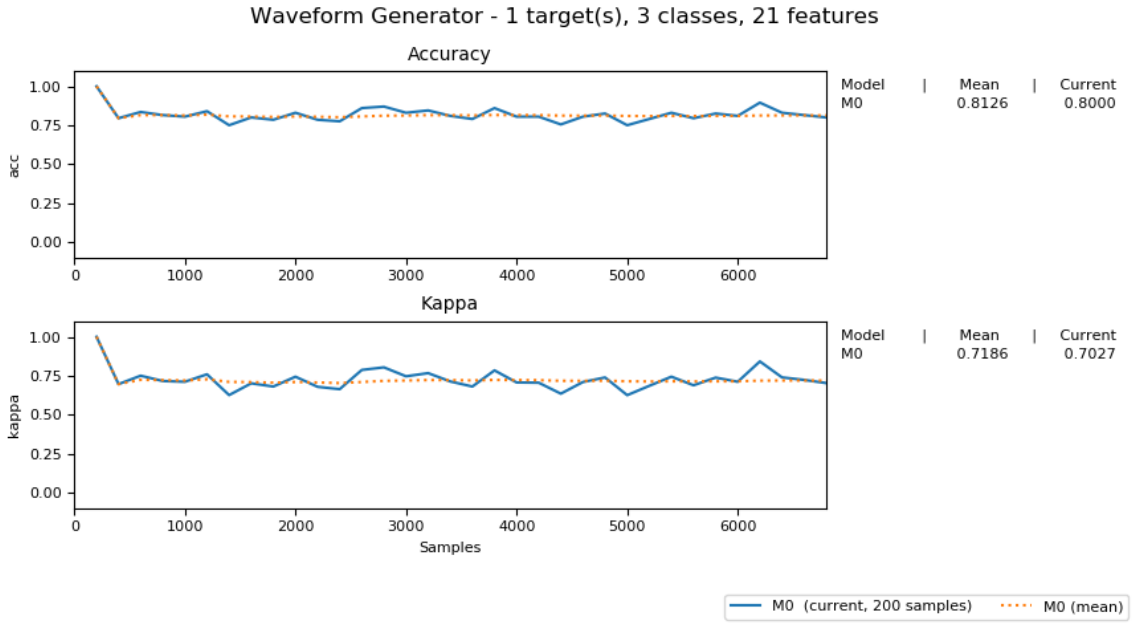

Figure 2.6: Result Table of the example in 2.1

3 Used Algorithms

3.1 Theoretical Work

This chapter will try to give a short explanation about how the RSLVQ works and will also explain the algorithm that it is based on. Furthermore it will explain what momentum based gradient descent algorithms 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 [20, 22], 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:

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

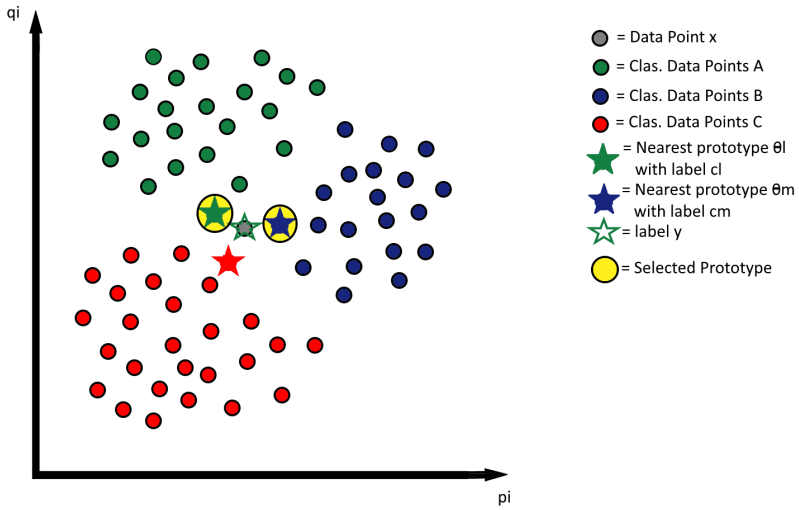

Figure 3.1: Graphic depiction of the LVQ classifcation task.

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, is 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. [29]

3.1.2 Robust Soft Learning Vector Quantization

The LVQ algorithm may be a popular adaptive nearest prototype classifier for multiclass classification, but the algorithms 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 [29]. It is a LVQ that is based on a Gausian Mixture. 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. [29]

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

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

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. [29]

Seo and Obermayer then showed in their work [29] 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. [29]

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 [31, 28], 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 3.2.

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

Momentum [26, 28] 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 [28] 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 [32]. 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 nontheless. 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

[9] 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 [25, 28].

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:

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

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 [10], 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. [28]

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 the fixed size ω.

Rather than inefficently storing 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 [10] 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 decaying 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. [28]

3.2.4 RMSprop

RMSprop is an adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class. [17] 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. [28]

3.2.5 Adam

As described in [19], 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. [28, 19]

Algorithm 1 Adam algorithm, as proposed in [19]. 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 [23] and was then further modiefied by Moritz Heusinger in [16]. It has to be said that the implementation is nearly the same as in the previously mentioned work [16].

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 [23].

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 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 [23]

Listing 3.3: Method `_update_prototype` as it was used in [16].

```
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 [16].

Listing 3.4: Adadelta implementation from [16].

```
1
2  """Implementation of Adadelta"""
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 past squared gradients
11       self.squared_mean_gradient[j] = self.decay_rate
    *self.squared_mean_gradient[j] + (1 - self.decay_rate) * gradient **
    2
12
13       # Compute update/step
14       step = ((self.squared_mean_step[j] + self.epsilon) /
    (self.squared_mean_gradient[j] + self.epsilon)) **0.5 * gradient
15
16       # Accumulate updates
17       self.squared_mean_step[j] = self.decay_rate *
    self.squared_mean_step[j] + (1 - self.decay_rate) * step ** 2
18
19       # Attract/Distract prototype to/from data point
20       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 [17], 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 [16].

Listing 3.5: Implementation of the RMSprop from [16].

```

"""RMSprop"""
      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 gradient
      self.squared_mean_gradient[j] = 0.9 *
 self.squared_mean_gradient[j] + 0.1 * gradient ** 2

      # Update Prototype
      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, at the start it repeats again the same steps of the SGD algorithm by computing the gradient. When the gradient is computed will it continue 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 computation 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.

```

if self.initial_fit:
  # Next two lines are Init for ADAM last m + v gradients
  self.gradients_v_sqrt = np.zeros_like(self.w_)
  self.gradients_m = np.zeros_like(self.w_)
  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. The last step will then apply the updated rule. Implemented is this like in 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 [28, 19].

Listing 3.7: Implementation of Adam on basis of [16].

```
"""Adam"""
      d = (xi - prototypes[j])

      """Calculate posterior"""
      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

      """Compute gradients m """
      self.gradients_m[j] = self.beta_1 * self.gradients_m[j] + (1 -
 self.beta_1) * gradient

      """Compute squared gradients v """
      self.gradients_v_sqrt[j] = self.beta_2 *
 self.gradients_v_sqrt[j] + (1 - self.beta_2) * gradient ** 2

      """Compute m correction"""
      m_corrected = self.gradients_m[j] / (1 - self.beta_1)

      """Compute v correction"""
      v_corrected = self.gradients_v_sqrt[j] / (1 - self.beta_2)

      """Update prototype"""
      self.w_[j] += (self.learning_rate/(np.sqrt(v_corrected) +
 self.epsilon ))*m_corrected
```

4 Test realisation

This section contains a short description of the synthetic and real datasets that were used to carry out the comparisions. It will also shortly describe the settings that were used and the reason for why these parameters were choosen. The Tests were carried out on a Microsoft Surface Pro(2017) with an Intel Core i5-7300U with 2 x 2,6 GHz and 8GB of RAM.

4.1 Used Datasets

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

4.1.1 Synthetic data

- Agrawal
  The generator was introduced by Agrawal et al in [2], and was a common source of data for early work on scaling up decision tree learners. The generator produces a stream containing nine features, six numeric and three categorical. There are ten functions defined for generating binary class labels from the features. Presumably these determine whether the loan should be approved. The features and functions are listed in the original paper [2, 23].

- Hyperplane
  Generates a problem of prediction class of a rotation hyperplane. It was used as testbed for CVFDT and VFDT in [18]. A hyperplane in d-dimensional space is the set of points x that satisfy $\sum_{i=1}^{d} w_i x_i = w_0 = \sum_{i=1}^{d} w_i$, where x_i is the i-th coordinate of x. Examples for which $\sum_{i=1}^{d} w_i x_i > w_0$, are labeled positive, and examples for which $\sum_{i=1}^{d} w_i x_i \leq w_0$, are labeled negative.

  Hyperplanes are useful for simulating time-changing concepts, because the change of the orientation and position of the hyperplane can be done in a smooth manner

by changing the relative size of the weights. Change can be introduced to this dataset by adding drift to each weight feature $w_i = w_i + d\sigma$, where σ is the probability that the direction of change is reversed and d is the change applied to every example [23].

- Sinegenerator
  This generator is an implementation of the data stream with abrupt concept drift, as described in Gama, Joao, et al [13] It generates up to 4 relevant numerical attributes, that vary from 0 to 1, where only 2 of them are relevant to the classification task and the other 2 are added by request of the user. A classification function is chosen among four possible ones:

  1. SINE1. Abrupt concept drift, noise-free examples. It has two relevant attributes. Each attributes has values uniformly distributed in $[0; 1]$. In the first context all points below the curve $y = sin(x)$ are classified as positive.

  2. Reversed SINE1. The reversed classification of SINE1.

  3. SINE2. The same two relevant attributes. The classification function is $y < 0.5 + 0.3sin(3\pi x)$.

  4. Reversed SINE2. The reversed classification of SINE2.

  Concept drift can be introduced by changing the classification function. This can be done manually or using the ConceptdriftStream method from the scikit-multiflow framework. Two important features are the possibility to balance classes, which means the class distribution will tend to a uniform one, and the possibility to add noise, which will, add two non relevant attributes [23].

- Transient Chessboard
  Virtual drift is generated by revealing successively parts of a chessboard. This is done square by square randomly chosen from the whole chessboard so that each square represents its own concept. Every time after four fields have been revealed, samples covering the whole chessboard are presented. This reoccurring alternation penalizes algorithms tending to discard former concepts. Only two instead of eight classes where used to reduce the impact of classification by chance [21].

- Moving Squares:
  Four equidistantly separated, squared uniform distributions are moving in horizontal direction with constant speed. The direction is inverted whenever the leading square reaches a predefined boundary. Each square represents a different class. The added value of this dataset is the predefined time horizon of 120 examples before old instances may start to overlap current ones. This is especially useful for dynamic sliding window approaches, allowing to test whether the size is adjusted

accordingly. [21]

- Sea Concepts:
  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. This exact Datas was also used in [30].

Dataset	Drift	Features	Classes	Targets	Samples
Agrawal	N	9	2	1	40 000
Agrawal Drift	Y	9	2	1	40 000
Hyperplane	Y	10	2	1	40 000
Hyperplane Drift	Y	10	2	1	40 000
Sine	Y	2	2	1	40 000
Sine Drift	Y	2	2	1	40 000
Chessboard	Y	2	8	1	200 000
Squares	N	2	4	1	200 000
Sea	Y	3	2	1	50 000

Table 4.1: Overview over the synthetic generated Datasets, the streams with "Drift" in name generate the drift with the $ConceptDrifStream()$ method from scikit-multiflow. (Y: yes, with Drift, N: no Drift)

4.1.2 Real world data

This section contains the description for datasets that were taken in the real world and that were already used in other publications.

- 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 measurments were taken 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 in [11].

- Electricity 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 the 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 that was used in [6].

- 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 [4], 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.

- Rialto Bridge Timelapse Dataset:
  Ten of the colorful buildings next to the famous Rialto bridge in Venice are encoded in a normalized 27-dimensional RGB histogram. The images were obtained from time-lapsed videos captured by a webcam with a fixed position. The recordings cover 20 consecutive days during may-june 2016. Continuously changing weather and lighting conditions affect the representation. The labels were generated by manually masking the corresponding buildings and excluded overnight recordings since they were too dark for being useful. This Dataset was originaly used in [21]

Dataset	Features	Classes	Targets	Samples
Weather	8	2	1	18 160
Electricity	8	2	1	45 312
Poker	10	10	1	829 201
Rialto	27	10	1	82 250

Table 4.2: Overview over the real Datasets

4.2 Test Settings

The test is run several times for each algorithm that is tested and also several times for each algorithm. The reason for this is that each algorithm is evaluated two times in different evaluation settings. Once it is evaluated with Holdout and once with Prequential (see section 2.3.1). In each of the evaluation processes each of the 4 algorithm performce a evaluation on the selected datastream. There are 13 different datastreams that will be used for the evaluation, which means that there will be $13 * 2 * 4 = 104$ runs

of the evalution. For each run will be a .csv file created and saved that will then later be condensed into 10 overview files for each of the evaluation metrics 4.3 for each of the evaluation settings. The metrics are represented in those list as percent, so a kappa score is then listed as 0.56, which means it has a value of 56 percent.

metrics
Accuracy
Kappa
$Kappa_m$
$Kappa_t$
Computation time

Table 4.3: Overview of the metrics that are recorded during the evaluation and that are then used for the comparision.

Each evaluation has a set maximum number of samples that it should use and after that it will stop and then start the next evaluation. This limit was set to 40 000, because nearly every datastream has at least 45 000 samples or more (there is 1 exception for this, and that is the weather dataset with only 18 000 samples) and the limitation should make it easier to directly compare the computation time between each of the algorithm even across different datastreams (with the exception for the weather dataset).

For the Setting of the hyperparameters of the Algorithms where mainly the recommended values from the litrature chosen. The parameters that exist or a shared between all/nearly all algorithms are set to the same start values. This was done so that each algorithm had the same starting conditions as the other ones for the comparision. The values are listed in table 4.4.

algorithm	σ	ϵ	$\gamma*$	$\eta*$	$\beta 1*$	$\beta 2*$	Sections
$RSLVQ_{SGD}$	1	$1e-8$	-	-	-	-	3.2.2
$RSLVQ_{Adadelta}$	1	$1e-8$	0.9	0.001	-	-	3.2.3
$RSLVQ_{RMSprop}$	1	$1e-8$	0.9	0.001	-	-	3.2.4
$RSLVQ_{Adam}$	1	$1e-8$	-	0.001	0.9	0.999	3.2.5

Table 4.4: Overview of the parameter settings for each of the algorithms. Parameters marked with a * only occured in some of the algorithms. (σ = number of prototypes, γ = decay rate, η = learning rate)

The evaluation was done in python with the scikit-multiflow framework described in section 2.4.

5 Result Evaluation

This chapter will list the results of the algorithm comparsions seperatly. First will be the results for the Holdout setting will be described and after that the results that were produced with the Prequential setting. The Result Tables on which these evaluations are based on can be found in the appendix of this work. (5 tables from the Holdout setting, 5 tables from the Prequential setting, 5 tables with ranks from the Holdout setting and 5 tables with ranks from the Prequential setting.)

5.1 Holdout settings

This section will provide the results of the algorithms based on their accuracy, performance measures, as described in section 2.3.2, and their computation time with Holdout evaluation setting

- **Accuracy**:
  Based on the Overall Accuracy score from Table 1 it can be seen that in the Holdout setting the normal $RSLVQ_{SGD}$ implementation performs the worst in comparision to the other three implementations. The $RSLVQ_{Adadelta}$ has the second best accuracy overall and the $RSLVQ_{RMSprop}$ has the best accuracy overall. The $RSLVQ_{Adam}$ implementation has performed worse than the $RSLVQ_{Adadelta}$ and the $RSLVQ_{RMSprop}$, but still performed better than the $RSLVQ_{SGD}$. Looking at the results of the synthetic and real Datastreams separetly, it can be seen that the the result are the same for the synthetic Streams, but on the datastream based on real world Data the $RSLVQ_{Adam}$ performed better then the $RSLVQ_{Adadelta}$ and the $RSLVQ_{RMSprop}$, as it was in fact the best performing implementation. The $RSLVQ_{Adadelta}$ was still the second best implementation, while the $RSLVQ_{RMSprop}$ was only the thrid best implementation. The $RSLVQ_{SGD}$ stayed the worst performing implementation accuracy wise.

- **Kappa**:
  In the next step the $Kappa$ score gets a closer look. The $Kappa$ score results are listed in the Table 2 and on their rank in Table 12. In these table can be similar result observed as in the results from the Accuracy metric. Overall the $RSLVQ_{SGD}$

again performed the worst out of all the implementations, while the $RSLVQ_{Adam}$ performed the third best again. The $RSLVQ_{Adadelta}$ and $RSLVQ_{RMSprop}$ switched their respectiv rank this time, with the $RSLVQ_{RMSprop}$ performing the best. A closer look to the results of the Synthetic and real world Data streams shows something similar to what was already observed in the the Accuracy score. While the ranking of the four implementation stayed the same for the synthetic Data streams, the $RSLVQ_{Adam}$ performed the best again for the real Data streams. The $RSLVQ_{RMSprop}$ performed the third best, while the $RSLVQ_{SGD}$ and the $RSLVQ_{Adadelta}$ stayed on their respectiv ranks of four and two. We can conclude from this that the $RSLVQ_{Adam}$ performs the best for the Kappa score when used on real streams with the holdout evalution.

- $Kappa_m$:
  In the Tables 3 and 13 are the results based on their $Kappa_m$ score listed. Based on the $Kappa_m$ score the algorithm performed the same as with the Kappa score. The $RSLVQ_{RMSprop}$ performed the best for the Overall result as well as for the synthetic streams only. While the $RSLVQ_{Adadelta}$ was second best in the overall, for synthetic and real data streams. The $RSLVQ_{Adam}$ was again the third best in the overall and synthetic only scores, while it was the best for real data streams. The $RSLVQ_{SGD}$ was again the worst in all three. Based on the results from the $Kappa_m$ score the best choice for the evaluation of real world data streams would be the $RSLVQ_{Adam}$ implementation with holdout evaluation settings.

- $Kappa_t$:
  The Tables 4 and 14 provide us with the metrics and the ranked values for the $Kappa_t$ score. This time the $RSLVQ_{Adam}$ performed the best in the Overall score and again on the real data streams, while performing the third best on the synthetic streams. The $RSLVQ_{Adadelta}$ and $RSLVQ_{SGD}$ implementations stayed again on the respectively second and last place for all three results. The $RSLVQ_{RMSprop}$ changed from the best performing implementation overall to only the third best performing implementation but still stayed the best performing algorithm for synthetic generated streams. Based on the performance of the $Kappa_t$ score the $RSLVQ_{Adam}$ is the best algorithm of the four compared for the evaluation of real data streams.

- **Computation Time**:
  The values for the Computation Time are listed in Table 5 and with ranks in Table 15. In it can be seen that the $RSLVQ_{SGD}$ implementation still performs the worst out of all 4 algorithms overall, but when used on real data streams it is the fastes computation wise. The fastes implementatin overall is the $RSLVQ_{Adadelta}$, followed $RSLVQ_{RMSprop}$ and the $RSLVQ_{Adam}$. If we take again a seprate look at the result for the synthetic and real data streams, we can see that on the synthetic stream the fastest algorithm was the $RSLVQ_{RMSprop}$, followed by the $RSLVQ_{Adam}$ and the $RSLVQ_{RMSprop}$ algorithms and with the $RSLVQ_{SGD}$ being the slowest one.

On the real data streams, as mentioned at the beginning, the $RSLVQ_{SGD}$ was the fastest, followed by the $RSLVQ_{RMSprop}$ and the $RSLVQ_{Adadelta}$ algorithms, the $RSLVQ_{Adam}$ algorithm being the slowest of the four. The computation time has shown that the other three algorithm may perform in all other metrics better than the normal $RSLVQ_{SGD}$ implementation, but this comes at the cost of the computation time, with all 3 taking longer than the $RSLVQ_{SGD}$.

5.2 Prequential settings

This section will provide the results of the algorithms based on their Accuracy, Preformance measures as described in section 2.3.2 and their Computation time with Prequential evaluation setting.

- **Accuracy**:
  The results and the rankings in respect to the Accuracy score for the algorithms are provided by the Tables 6 and 16 respectively. With prequential evalution enabled the $RSLVQ_{SGD}$ performed the best out off all four algorithms, followed by the $RSLVQ_{RMSprop}$ and the $RSLVQ_{Adadelta}$, with the $RSLVQ_{Adam}$ performing the worst on the accuracy score. A separate look at the synthetic and real data stream shows that for the synthetic streams the $RSLVQ_{RMSprop}$ performed the best, followed by the $RSLVQ_{Adadelta}$ and the $RSLVQ_{SGD}$ as well as the $RSLVQ_{Adam}$ performing the thrid best. The real data streams showed the same results as the overall results, with the $RSLVQ_{SGD}$ performing the best and the $RSLVQ_{Adam}$ performing the worst out of the four streams.

- **Kappa**:
  The Tables 7 and 17 provide us with the values and the rankes for the Kappa score that were taken with the prequentail evaluation. In the Overall result we can see that this time the $RSLVQ_{RMSprop}$ performed the best, followed by the $RSLVQ_{SGD}$ and $RSLVQ_{Adadelta}$ implementations. The $RSLVQ_{Adam}$ performed again as the worst out of the four tested implementations. The synthetic streams results show on the other hand that the $RSLVQ_{Adam}$ performed the best out of the four algorithms, followed by the $RSLVQ_{RMSprop}$ and the $RSLVQ_{Adadelta}$, with the $RSLVQ_{SGD}$ performing the worst on the Kappa score. From this we can conclude that the $RSLVQ_{Adam}$ algorithm should not be taken to evaluate real world data streams when used with prequentail evaluation settings. Instead of it the $RSLVQ_{RMSprop}$ should be taken according to the Kappa score results.

- $Kappa_m$:
  In the Tables 8 and 18 are the results listed for the $Kappa_m$ score with prequential evaluation settings. The results for the $Kappa_m$ score show a simliar ranking

for the overall results as the Kappa score had shown, with the $RSLVQ_{RMSprop}$ performing the best and the $RSLVQ_{Adam}$ performing the worst again. For the synthetic streams the $RSLVQ_{RMSprop}$ performed the best out of the four, followed by the $RSLVQ_{Adadelta}$ and the $RSLVQ_{Adam}$, with the $RSLVQ_{SGD}$ performing the worst. On the Real data streams the $RSLVQ_{SGD}$ performed the best with the $RSLVQ_{RMSprop}$ still being the second best performing algorithm and the $RSLVQ_{Adadelta}$ and $RSLVQ_{Adam}$ performing as the third and fourth best algorithms respectively.

- $Kappa_t$:
  The results for the $Kappa_t$ score are provided in the Tables 9 and 19. The $RSLVQ_{RMSprop}$ performed the best in the overall, synthetic and real data stream results in respect to the $Kappa_t$ score. The $RSLVQ_{Adadelta}$ performed in the overall results as well as in the results for the synthetic streams as the second best algorithm of the four, while still being the third best performing algorithm on the real data streams in regards to the $Kappa_t$ score. The $RSLVQ_{Adadelta}$ performed the worst on the synthetic streams, while still performing the third best on the real data streams and third best on the overall results. The $RSLVQ_{Adam}$ performed the worst of the four algorithms on the overal and real data stream results, while only performing the third best on the synthetic streams according to the $Kappa_t$. According to the results of the $Kappa_t$ score as well as the results of the other Kappa scores and the Accuracy score, it can be concluded that if the evaluation takes place with prequential settings, the algorithm that would return the best results would be the $RSLVQ_{RMSprop}$.

- **Computation Time**:
  The results for the Computation Time are stored in Table 10 and with ranking in Table 20. The overall fastest algorithm with prequential evaluation settings was the $RSLVQ_{Adam}$ algorithm, followed by the $RSLVQ_{SGD}$, the $RSLVQ_{Adadelta}$ and the $RSLVQ_{RMSprop}$ algorithm being the slowest of the four. Looking at the synthetic streams seperatly, it can be seen that the $RSLVQ_{SGD}$ was the fastest this time, followed by the $RSLVQ_{Adadelta}$ and the $RSLVQ_{Adam}$, with the $RSLVQ_{RMSprop}$ being the slowest again. On the real data streams it can be seen that the $RSLVQ_{Adam}$ was again the fastest implementation, while the $RSLVQ_{SGD}$ was the second fastest and the $RSLVQ_{RMSprop}$ being the third fastest. The computation time shows again that the algorithm that will return the best results will need some extra time to compute these results. It also showed that overall the $RSLVQ_{Adam}$ may have performed faster than under the Holdout settings, but because of that the results were worse than before.

6 Summary

The Test results have shown that the $RSLVQ_{Adam}$ does indeed perform Favourable in regards to the $RSLVQ_{Adadelta}$ and the $RSLVQ_{RMSprop}$ under Holdout evaluation settings, like it was described in [19]. It had also shown that the $RSLVQ_{Adadelta}$, $RSLVQ_{RMSprop}$ and the $RSLVQ_{Adam}$ all perform better towards the standard $RSLVQ_{SGD}$ under this setting. The only result where all three performed worse than the standard $RSLVQ_{SGD}$ implementation was the Computation time, but this can be explained by the fact the algorithms are more complex than the $RSLVQ_{SGD}$ and due to this taking more time to compute the results. The $RSLVQ_{Adam}$ also performed consistently good on the evaluation of real world data streams, when they were evaluated with Holdout settings. It should be noted that on synthetic streams the $RSLVQ_{Adam}$ performed worse than the $RSLVQ_{Adadelta}$ and the $RSLVQ_{RMSprop}$ implementations, so for the evaluation under holdout settings the other ones should be preferred for use rather than the $RSLVQ_{Adam}$.

If the evluation is done with prequential settings, the $RSLVQ_{Adam}$ performed worse than either the $RSLVQ_{Adadelta}$, $RSLVQ_{RMSprop}$ and the $RSLVQ_{SGD}$, while the $RSLVQ_{RMSprop}$ returned the best results under this setting. It should also be noted that if the evaluation is only done on a synthetic stream, the $RSLVQ_{Adam}$ could be taken as a possible alternative to the $RSLVQ_{RMSprop}$ or the $RSLVQ_{Adadelta}$, as it only performs a little bit worse than the other two in this case, but is faster in regards to the computation time in comparision to the other two.

Abbreviations

ADAM Adaptive Moment Estimation

IoT Internet of Things

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] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. "Database mining: A performance perspective". In: *IEEE transactions on knowledge and data engineering* 5.6 (1993), pp. 914–925.

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

[4] 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.

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] Ryan Elwell and Robi Polikar. "Incremental learning of concept drift in nonstationary environments". In: *IEEE Transactions on Neural Networks* 22.10 (2011), pp. 1517–1531.

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

[13] Joao Gama et al. "Learning with drift detection". In: *Brazilian symposium on artificial intelligence*. Springer. 2004, pp. 286–295.

[14] 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.

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

[16] 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.

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

[18] Geoff Hulten, Laurie Spencer, and Pedro Domingos. "Mining time-changing data streams". In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. 2001, pp. 97–106.

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

[20] 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.

[21] V. Losing, B. Hammer, and H. Wersing. "KNN Classifier with Self Adjusting Memory for Heterogeneous Concept Drift". In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. Dec. 2016, pp. 291–300. DOI: 10.1109/ICDM.2016.0040.

[22] 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.

[23] 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.

[24] Nils J. Nilsson. *Introduction to Machine Learning*. https://ai.stanford.edu/~nilsson/MLBOOK.pdf. 1998.

[25] 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.

[26] 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.

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

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

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

[30] 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.

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

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

[33] 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.

Eidesstattliche Erklärung

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 22, 2020

Appendices

1 Result Tables

1.1 Holdout results

- **Accuracy:**

Data Stream	$RSLVQ_{SGD}$	$RSLVQ_{Adadelta}$	$RSLVQ_{RMSprop}$	$RSLVQ_{Adam}$
Agrawal	0.4802	0.575	0.575	0.575
AgrawalDrift	0.5618	0.5828	0.5826	0.5828
Hyperplane	0.577	0.751	0.7783	0.7348
HyperplaneDrift	0.604	0.8108	0.7778	0.698
Sine		0.9824	0.9822	0.9826
SineDrift		0.9823	0.9822	0.9822
CessData	0.1335	0.1369	0.1232	0.1311
MoveSquares	0.318	0.291	0.3265	0.3097
SeaData	0.6206	0.7741	0.8492	0.7787
Synth. Mean	0.4707	0.654	0.6641	0.6417
Synth. Rank	4.0	2.0	1.0	3.0
Electric	0.5174	0.6468	0.6238	0.6637
Poker	0.2562	0.3176	0.2949	0.3815
Weather	0.5132	0.6778	0.6884	0.678
Rialto	0.1661	0.1739	0.1878	0.1922
Real Mean	0.3632	0.454	0.4487	0.4788
Real Rank	4.0	2.0	3.0	1.0
Overall Mean	0.4316	0.5925	0.5978	0.5916
Overall Rank	4.0	2.0	1.0	3.0

Table 1: Accuracy values for the Holdout evaluation.

- **Kappa:**

Data Stream	$RSLVQ_{SGD}$	$RSLVQ_{Adadelta}$	$RSLVQ_{RMSprop}$	$RSLVQ_{Adam}$
Agrawal	-0.0001	-0.0081	-0.0081	-0.0081
AgrawalDrift	0.0029	0.0128	0.0125	0.0128
Hyperplane	0.1541	0.5028	0.5571	0.47
HyperplaneDrift	0.2075	0.6215	0.5555	0.3946
Sine		0.9644	0.964	0.965
SineDrift		0.9643	0.9641	0.9641
CessData	0.0107	0.0134	0.0061	-0.0005
MoveSquares	0.0906	0.0547	0.102	0.0796
SeaData	0.2861	0.5323	0.6766	0.5403
Synth. Mean	0.1074	0.4065	0.4255	0.3798
Synth. Rank	4.0	2.0	1.0	3.0
Electric	0.0784	0.3499	0.2224	0.3425
Poker	0.0504	0.0705	0.0235	0.1189
Weather	0.06	0.1928	0.2217	0.1909
Rialto	0.0734	0.0821	0.0976	0.1025
Real Mean	0.0655	0.1738	0.1413	0.1887
Real Rank	4.0	2.0	3.0	1.0
Overall Mean	0.0922	0.3349	0.3381	0.321
Overall Rank	4.0	2.0	1.0	3.0

Table 2: Kappa values for the Holdout evaluation.

- $Kappa_m$**:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	-0.5705	-0.284	-0.284	-0.284
AgrawalDrift	-0.3004	-0.238	-0.2386	-0.238
Hyperplane	0.1255	0.4876	0.5647	0.4742
HyperplaneDrift	0.2025	0.6275	0.5478	0.3878
Sine		0.9675	0.9671	0.9681
SineDrift		0.9674	0.9672	0.9672
CessData	-0.0869	0.0838	0.0109	-0.0899
MoveSquares	0.0906	0.0547	0.102	0.0796
SeaData	-0.0353	0.3835	0.5883	0.396
Synth. Mean	-0.0821	0.3389	0.3584	0.2957
Synth. Rank	4.0	2.0	1.0	3.0
Electric	-0.2393	0.0931	0.034	0.1364
Poker	0.2399	0.3166	0.2938	0.3806
Weather	-0.638	-0.0841	-0.0485	-0.0834
Rialto	0.0734	0.0821	0.0976	0.1025
Real Mean	-0.141	0.1019	0.0942	0.134
Real Rank	4.0	2.0	3.0	1.0
Overall Mean	-0.1035	0.266	0.2771	0.2459
Overall Rank	4.0	2.0	1.0	3.0

Table 3: $Kappa_m$ values for the Holdout evaluation.

- $Kappa_t$**:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	-0.1761	0.0385	0.0385	0.0385
AgrawalDrift	0.0014	0.0493	0.0489	0.0493
Hyperplane	0.1568	0.506	0.5556	0.4649
HyperplaneDrift	0.188	0.6227	0.5603	0.3974
Sine		0.9632	0.9628	0.9639
SineDrift		0.9631	0.9629	0.9629
CessData	-1.473	-1.463	-1.5021	-1.4797
MoveSquares	0.318	0.291	0.3265	0.3097
SeaData	0.1838	0.514	0.6755	0.5239
Synth. Mean	-0.1144	0.2761	0.2921	0.2479
Synth. Rank	4.0	2.0	1.0	3.0
Electric	-2.0314	-1.2183	-1.3628	-1.1124
Poker	-2.5826	-2.2871	-2.3964	-1.9793
Weather	-0.5109	0.0001	0.0329	0.0006
Rialto	0.1661	0.1739	0.1878	0.1922
Real Mean	-1.2397	-0.8328	-0.8846	-0.7247
Real Rank	4.0	2.0	3.0	1.0
Overall Mean	-0.5236	-0.0651	-0.07	-0.0514
Overall Rank	4.0	2.0	3.0	1.0

Table 4: $Kappa_t$ values for the Holdout evaluation.

- **Computation time:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	31.75	30.48	30.53	30.46
AgrawalDrift	31.68	32.04	31.85	31.6
Hyperplane	28.88	27.37	27.92	29.29
HyperplaneDrift	29.69	28.43	28.69	31.42
Sine		25.88	26.13	27.6
SineDrift		30.86	31.9	28.01
CessData	76.06	78.81	88.77	84.39
MoveSquares	41.71	50.79	48.66	48.36
SeaData	31.02	30.64	29.95	30.85
Synth. Mean	38.6843	37.2556	38.2667	37.9978
Synth. Rank	4.0	1.0	3.0	2.0
Electric	29.32	28.98	29.47	30.45
Poker	106.03	110.32	108.92	111.37
Weather	11.82	12.44	12.07	12.21
Rialto	96.64	105.62	103.86	107.25
Real Mean	60.9525	64.34	63.58	65.32
Real Rank	1.0	3.0	2.0	4.0
Overall Mean	46.7818	45.5892	46.0554	46.4046
Overall Rank	4.0	1.0	2.0	3.0

Table 5: Computation Time for the Holdout evaluation.

1.2 Prequential results

- **Accuracy:**

Data Stream	$RSLVQ_{SGD}$	$RSLVQ_{Adadelta}$	$RSLVQ_{RMSprop}$	$RSLVQ_{Adam}$
Agrawal	0.5006	0.5801	0.5802	0.5089
AgrawalDrift	0.5051	0.5804	0.5803	0.4371
Hyperplane	0.606	0.782	0.7739	0.7685
HyperplaneDrift	0.6325	0.8406	0.8293	0.8398
Sine	0.8614	0.9844	0.9835	0.984
SineDrift	0.8628	0.984	0.9834	0.9843
CessData	0.5908	0.5791	0.5801	0.458
MoveSquares	0.9556	0.1357	0.2454	0.322
SeaData	0.6317	0.8277	0.814	0.8431
Synth. Mean	0.6829	0.6993	0.7078	0.6829
Synth. Rank	3.5	2.0	1.0	3.5
Electric	0.8284	0.8882	0.8955	0.7763
Poker	0.7637	0.6682	0.714	0.4577
Weather	0.638	0.6643	0.7308	0.6041
Rialto	0.7775	0.2715	0.2837	0.2656
Real Mean	0.7519	0.623	0.656	0.5259
Real Rank	1.0	3.0	2.0	4.0
Overall Mean	0.7042	0.6759	0.6919	0.6346
Overall Rank	1.0	3.0	2.0	4.0

Table 6: Accuracy values for the Prequential evaluation.

- **Kappa:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	-0.0048	-0.0028	-0.0027	0.0002
AgrawalDrift	0.0052	0.0032	0.0032	-0.0055
Hyperplane	0.212	0.5641	0.5477	0.5371
HyperplaneDrift	0.2649	0.6812	0.6586	0.6796
Sine	0.7209	0.9685	0.9669	0.9678
SineDrift	0.7238	0.9678	0.9666	0.9684
CessData	0.5319	0.5187	0.5197	0.3802
MoveSquares	0.9408	-0.1524	-0.0062	0.096
SeaData	0.2451	0.6264	0.5954	0.666
Synth. Mean	0.4044	0.4639	0.4721	0.4766
Synth. Rank	4.0	3.0	2.0	1.0
Electric	0.6479	0.7705	0.7854	0.5406
Poker	0.6002	0.457	0.5172	0.1243
Weather	0.2126	0.1742	0.3862	0.2131
Rialto	0.7527	0.1906	0.2042	0.184
Real Mean	0.5534	0.3981	0.4733	0.2655
Real Rank	1.0	3.0	2.0	4.0
Overall Mean	0.4502	0.4436	0.4725	0.4117
Overall Rank	2.0	3.0	1.0	4.0

Table 7: Kappa values for the Prequential evaluation.

- $Kappa_m$**:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	0.2596	0.3775	0.3776	0.2719
AgrawalDrift	0.2595	0.3724	0.3723	0.158
Hyperplane	0.2059	0.56	0.5502	0.5404
HyperplaneDrift	0.2607	0.6776	0.6532	0.6813
Sine	0.6972	0.9659	0.9641	0.9651
SineDrift	0.7006	0.9651	0.9638	0.9657
CessData	0.5251	0.5259	0.5269	0.3892
MoveSquares	0.9408	-0.1524	-0.0062	0.096
SeaData	0.4234	0.7302	0.7088	0.7543
Synth. Mean	0.4748	0.558	0.5679	0.5358
Synth. Rank	4.0	2.0	1.0	3.0
Electric	0.7045	0.8076	0.8201	0.6149
Poker	0.7476	0.6653	0.7115	0.4191
Weather	0.4728	0.5111	0.6079	0.4235
Rialto	0.7527	0.1906	0.2042	0.184
Real Mean	0.6694	0.5436	0.5859	0.4104
Real Rank	1.0	3.0	2.0	4.0
Overall Mean	0.5346	0.5536	0.5734	0.4972
Overall Rank	3.0	2.0	1.0	4.0

Table 8: $Kappa_m$ values for the Prequential evaluation.

- $Kappa_t$**:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	-0.1431	0.0389	0.039	-0.1241
AgrawalDrift	-0.1155	0.0509	0.0519	-0.2681
Hyperplane	0.2125	0.5609	0.5488	0.5362
HyperplaneDrift	0.2558	0.6825	0.6594	0.6802
Sine	0.7179	0.9682	0.9665	0.9675
SineDrift	0.7206	0.9676	0.9663	0.9681
CessData	0.0217	-0.006	-0.0038	-0.2956
MoveSquares	0.9556	0.1357	0.2454	0.322
SeaData	0.1927	0.6223	0.5924	0.656
Synth. Mean	0.3131	0.4468	0.4518	0.3825
Synth. Rank	4.0	2.0	1.0	3.0
Electric	-0.1827	0.2297	0.28	-0.5416
Poker	-0.0925	-0.5336	-0.322	-1.507
Weather	-0.1347	-0.0522	0.1562	-0.2408
Rialto	0.7775	0.2715	0.2837	0.2656
Real Mean	0.0919	-0.0211	0.0995	-0.506
Real Rank	2.0	3.0	1.0	4.0
Overall Mean	0.2451	0.3028	0.3434	0.1091
Overall Rank	3.0	2.0	1.0	4.0

Table 9: $Kappa_t$ values for the Prequential evaluation.

- **Computation time:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	35.97	36.08	35.66	36.96
AgrawalDrift	37.6	37.33	37.22	39.12
Hyperplane	36.02	34.57	34.41	34.28
HyperplaneDrift	37.29	36.05	36.04	34.89
Sine	34.16	32.41	31.38	32.25
SineDrift	34.08	32.94	33.54	33.75
CessData	89.9	89.82	101.15	93.2
MoveSquares	49.84	59.13	57.09	59.02
SeaData	37.72	36.0	35.9	33.66
Synth. Mean	43.62	43.8144	44.71	44.1256
Synth. Rank	1.0	2.0	4.0	3.0
Electric	35.85	35.31	34.96	35.16
Poker	120.73	122.72	121.62	117.75
Weather	17.3	17.67	17.55	16.51
Rialto	114.05	123.59	123.9	113.92
Real Mean	71.9825	74.8225	74.5075	70.835
Real Rank	2.0	4.0	3.0	1.0
Overall Mean	52.3469	53.3554	53.8785	52.3438
Overall Rank	2.0	3.0	4.0	1.0

Table 10: Computation Time for the Prequential evaluation.

2 Ranked Result Tables

2.1 Holdout results

- **Ranked Accuracy:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	4.0	2.0	2.0	2.0
AgrawalDrift	4.0	1.5	3.0	1.5
Hyperplane	4.0	2.0	1.0	3.0
HyperplaneDrift	4.0	1.0	2.0	3.0
Sine	4.0	2.0	3.0	1.0
SineDrift	4.0	1.0	2.5	2.5
CessData	2.0	1.0	4.0	3.0
MoveSquares	2.0	4.0	1.0	3.0
SeaData	4.0	3.0	1.0	2.0
Electric	4.0	2.0	3.0	1.0
Poker	4.0	2.0	3.0	1.0
Weather	4.0	3.0	1.0	2.0
Rialto	4.0	3.0	2.0	1.0

Table 11: Ranked Accuracy for the Holdout evaluation.

- **Ranked Kappa:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	1.0	3.0	3.0	3.0
AgrawalDrift	4.0	1.5	3.0	1.5
Hyperplane	4.0	2.0	1.0	3.0
HyperplaneDrift	4.0	1.0	2.0	3.0
Sine	4.0	2.0	3.0	1.0
SineDrift	4.0	1.0	2.5	2.5
CessData	2.0	1.0	3.0	4.0
MoveSquares	2.0	4.0	1.0	3.0
SeaData	4.0	3.0	1.0	2.0
Electric	4.0	1.0	3.0	2.0
Poker	3.0	2.0	4.0	1.0
Weather	4.0	2.0	1.0	3.0
Rialto	4.0	3.0	2.0	1.0

Table 12: Ranked Kappa for the Holdout evaluation.

- **Ranked** $Kappa_m$**:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	4.0	2.0	2.0	2.0
AgrawalDrift	4.0	1.5	3.0	1.5
Hyperplane	4.0	2.0	1.0	3.0
HyperplaneDrift	4.0	1.0	2.0	3.0
Sine	4.0	2.0	3.0	1.0
SineDrift	4.0	1.0	2.5	2.5
CessData	3.0	1.0	2.0	4.0
MoveSquares	2.0	4.0	1.0	3.0
SeaData	4.0	3.0	1.0	2.0
Electric	4.0	2.0	3.0	1.0
Poker	4.0	2.0	3.0	1.0
Weather	4.0	3.0	1.0	2.0
Rialto	4.0	3.0	2.0	1.0

Table 13: Ranked $Kappa_m$ for the Holdout evaluation.

- **Ranked** $Kappa_t$**:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	4.0	2.0	2.0	2.0
AgrawalDrift	4.0	1.5	3.0	1.5
Hyperplane	4.0	2.0	1.0	3.0
HyperplaneDrift	4.0	1.0	2.0	3.0
Sine	4.0	2.0	3.0	1.0
SineDrift	4.0	1.0	2.5	2.5
CessData	2.0	1.0	4.0	3.0
MoveSquares	2.0	4.0	1.0	3.0
SeaData	4.0	3.0	1.0	2.0
Electric	4.0	2.0	3.0	1.0
Poker	4.0	2.0	3.0	1.0
Weather	4.0	3.0	1.0	2.0
Rialto	4.0	3.0	2.0	1.0

Table 14: Ranked $Kappa_t$ for the Holdout evaluation.

- **Ranked Computation time:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	4.0	2.0	3.0	1.0
AgrawalDrift	2.0	4.0	3.0	1.0
Hyperplane	3.0	1.0	2.0	4.0
HyperplaneDrift	3.0	1.0	2.0	4.0
Sine	4.0	1.0	2.0	3.0
SineDrift	4.0	2.0	3.0	1.0
CessData	1.0	2.0	4.0	3.0
MoveSquares	1.0	4.0	3.0	2.0
SeaData	4.0	2.0	1.0	3.0
Electric	2.0	1.0	3.0	4.0
Poker	1.0	3.0	2.0	4.0
Weather	1.0	4.0	2.0	3.0
Rialto	1.0	3.0	2.0	4.0

Table 15: Ranked Computation Time for the Holdout evaluation.

2.2 Prequential results

- **Ranked Accuracy:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	4.0	2.0	1.0	3.0
AgrawalDrift	3.0	1.0	2.0	4.0
Hyperplane	4.0	1.0	2.0	3.0
HyperplaneDrift	4.0	1.0	3.0	2.0
Sine	4.0	1.0	3.0	2.0
SineDrift	4.0	2.0	3.0	1.0
CessData	1.0	3.0	2.0	4.0
MoveSquares	1.0	4.0	3.0	2.0
SeaData	4.0	2.0	3.0	1.0
Electric	3.0	2.0	1.0	4.0
Poker	1.0	3.0	2.0	4.0
Weather	3.0	2.0	1.0	4.0
Rialto	1.0	3.0	2.0	4.0

Table 16: Ranked Accuracy for the Prequential evaluation.

- **Ranked Kappa:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	4.0	3.0	2.0	1.0
AgrawalDrift	1.0	2.5	2.5	4.0
Hyperplane	4.0	1.0	2.0	3.0
HyperplaneDrift	4.0	1.0	3.0	2.0
Sine	4.0	1.0	3.0	2.0
SineDrift	4.0	2.0	3.0	1.0
CessData	1.0	3.0	2.0	4.0
MoveSquares	1.0	4.0	3.0	2.0
SeaData	4.0	2.0	3.0	1.0
Electric	3.0	2.0	1.0	4.0
Poker	1.0	3.0	2.0	4.0
Weather	3.0	4.0	1.0	2.0
Rialto	1.0	3.0	2.0	4.0

Table 17: Ranked Kappa for the Prequential evaluation.

- **Ranked** $Kappa_m$**:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	4.0	2.0	1.0	3.0
AgrawalDrift	3.0	1.0	2.0	4.0
Hyperplane	4.0	1.0	2.0	3.0
HyperplaneDrift	4.0	2.0	3.0	1.0
Sine	4.0	1.0	3.0	2.0
SineDrift	4.0	2.0	3.0	1.0
CessData	3.0	2.0	1.0	4.0
MoveSquares	1.0	4.0	3.0	2.0
SeaData	4.0	2.0	3.0	1.0
Electric	3.0	2.0	1.0	4.0
Poker	1.0	3.0	2.0	4.0
Weather	3.0	2.0	1.0	4.0
Rialto	1.0	3.0	2.0	4.0

Table 18: Ranked $Kappa_m$ for the Prequential evaluation.

- **Ranked** $Kappa_t$**:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	4.0	2.0	1.0	3.0
AgrawalDrift	3.0	2.0	1.0	4.0
Hyperplane	4.0	1.0	2.0	3.0
HyperplaneDrift	4.0	1.0	3.0	2.0
Sine	4.0	1.0	3.0	2.0
SineDrift	4.0	2.0	3.0	1.0
CessData	1.0	3.0	2.0	4.0
MoveSquares	1.0	4.0	3.0	2.0
SeaData	4.0	2.0	3.0	1.0
Electric	3.0	2.0	1.0	4.0
Poker	1.0	3.0	2.0	4.0
Weather	3.0	2.0	1.0	4.0
Rialto	1.0	3.0	2.0	4.0

Table 19: Ranked $Kappa_t$ for the Prequential evaluation.

- **Ranked Computation time:**

Data Stream	$RSLVQ_{\boldsymbol{SGD}}$	$RSLVQ_{\boldsymbol{Adadelta}}$	$RSLVQ_{\boldsymbol{RMSprop}}$	$RSLVQ_{\boldsymbol{Adam}}$
Agrawal	2.0	3.0	1.0	4.0
AgrawalDrift	3.0	2.0	1.0	4.0
Hyperplane	4.0	3.0	2.0	1.0
HyperplaneDrift	4.0	3.0	2.0	1.0
Sine	4.0	3.0	1.0	2.0
SineDrift	4.0	1.0	2.0	3.0
CessData	2.0	1.0	4.0	3.0
MoveSquares	1.0	4.0	2.0	3.0
SeaData	4.0	3.0	2.0	1.0
Electric	4.0	3.0	1.0	2.0
Poker	2.0	4.0	3.0	1.0
Weather	2.0	4.0	3.0	1.0
Rialto	2.0	3.0	4.0	1.0

Table 20: Ranked Computation Time for the Prequential evaluation.

2.3 List of contents on USB stick

-