# A Comparison of Online learning Naïve Bayes Classifier on RSS Feeds using SPARK

**Anil Palla**
pala@campus.tu-berlin.de

**Franziska Adler**
minza@mailbox.tu-berlin.de

## Abstract

text text text text text text text text text text text text text text text text text text text
text text text text text

# 1 Theoretical background

text text text

## 1.1 Motivation

In Text Classification Naïve Bayes Classifiers are popular due to their simplicity and find application in the field of spam email detection and discovery of specified web content [1, p. 225]. Classifier aim the prediction of a category like spam or ham on the basis of previous examples. This can be easily realised with a probabilistic classifier like Naïve Bayes assuming that the data distribution is static over time, all data is always accessible and query time does not play a major role. For adapting real world problems and their dynamics this might not be sufficient. For applications requiring immidiate processing of fresh data and answering queries in real-time, instead of traditional learning approaches, more real-time-tailored and more learning techniques involving sophisticated model updates have to be devised. The reason why model updates are necessary is the prevalence of shifts in the statistical distribution of the target classes in online-learning scenarios. This is called *concept drift*. Also aspects of scalability should be considered since concept drifts are common in situations of large data quantities arriving via streams with high data rates [2, p. 4]

In this project we use RSS feeds from BCC for classification on a distributed system. Focus is the evaluation of several Naïve Bayes classifiers which implement the online learning paradigm by using different model update techniques. The evaluation concentrates on the handling of concept drifts. First part of this report explores the theoretical foundations of the relevant concepts as well as the explanations of the conceptual challenges we face and the methodology we follow for implementation and the evaluation. The second part digs in more details of our implementation such as how the different classifier approaches are realised. Furthermore the evaluation of their performances are analysed and presented.

## 1.2 Naïve Bayes classifier

As mentioned Naïve Bayes classifiers are probabilistic classifiers which are simple but effective in text classification. They operate on the basis of the Bayes Theorem and assume independence of features. Here the feature values are normalised words frequencies occurring in a document. The probability of a word $w$ belonging to class $y \in Y$ is given by $P(y|w_i)$ and can be reformulated with Bayes Theorem to its conditional probability and a priori probability $P(y)$ of class $y$. Both can be derived from frequencies of documents and words.

$$P(y|w_i) \propto P(y)P(w_i|y)$$

1

The computation of the joint probability over a documents features will give the probability for a document $d$ belonging to a certain class $y$. Instead of multiplication the logarithm is used to avoid underflow by multiplying with zero:

$$P(y|d) \propto P(y) \prod_i P(w_i|y), \text{bzw.: } P(y|d) \propto logP(y)log \sum_i P(w_i|y),$$

Applying for all classes will give the decision rule which categorise a new text document according to its most likely class which is the one with the highest joint probability value.

$$\arg \max_y \{logP(y)log \sum_i P(w_i|y)\}$$

### 1.3 Challanges/ Concepts

### 1.3.1 Concept Drift

Most online learning scenarios including ours, news feed mining, is susceptible to the phenomenon called $concept\ drift$. Concept drift occurs when the statistical properties of the target variable to be predicted change over time. Concept drift can manifest itself in different ways as explained in [3, p. 5] . Firstly, shifts in the prior class probabilities or class-conditional probabilities could be the indicator of a concept drift. Likewise, if the relevance between a certain set of attribute values, clues, and certain class predictions change, this could signal a concept drift. Moreover, abrupt increase in the model complexity can be a sign of the concept drift as well for the applicable predictive models. Finally and most importantly, prediction accuracy serves as the most widely-used criterion for detecting the presence of a concept drift.

The sources of the concept drifts are diverse depending on the application. For example, in spam filtering applications, user changing his/her mind about what is spam and what is not is an example of a concept drift. In our classification scenario, news feed items categorisation, concept drifts can directly stem from the shift in the media attention which is naturally drawn to the emerging real-life events and its implications on the set of vocabulary news writers use to explain emerging phenomena. For example, a new music album titled "Anarchy in the UK" is introduced and sold millions of copies in short time hence became the topic of many news articles. Since the terms in the title suggests that the chances are that this item could be linked to the category 'politics', most probably the naive bayesian classifier will capture the statistical correspondence of these words to its corresponding what-it-would-be-categorized-as-in-past category and lead to false predictions.

Different ways of handling concept drifts are proposed in literature [3, p. 5]. These different approaches can be classified according to 4 different criteria. First criterium is whether instance-based or batch-based processing of the stream items are employed. Since, instance-based processing is mostly too costly in terms of CPU load, batch-based processing where the continuous stream is discretised as batches are mostly used. Secondly, concept drift handling differs by the mechanism of the drift detection. There are two common kinds of concept drifts detection mechanism namely explicit detection and implicit-detection. Explicit-detection mechanism takes a detection action upon the detection such as retraining the model based on the recent items. On the other hand, implicit-detection only continuously adjusts the weights of different classification parameters as a function of concept drift indicators such as error rate, accuracy, etc. These parameters could be the weights of the members of the ensemble classifiers letting the models trained with the obsolete data fade. Moreover, another criterium for the above-mentioned classification is whether classifier-specific or classifier-free action mechanism is employed. In the former, the detection and action mechanism depends on the nature of the classifier and cannot be applied to the all kinds of classifiers. In the latter, the detection is bound to the accuracy and the action is updating the training data so it works with any classifier. Finally, concept drifts can be handled by a single classifier or an ensemble of them. When an ensemble is employed, the prediction decision is jointly taken by a dynamic combiner logic that forgets some of the classifiers which performs bad. Furthermore, although the drift detection action is usually implicit in the ensembles, every now and then ensemble can drop a member by running 'replace the loser' policy.

For this project, we use three different stream learning approaches. They all do batch-based processing using a single classifier for explicit concept drift detection. In terms of classifier-dependency of the concept-drift handling methods, two of them implement classifier-free methods for detection

and taking action for the drifting concepts and one of them employs classifier-specific approach. Details regarding these different stream learning approaches we use are discussed in the Methods subsection.

### 1.3.2 Streaming

In order to deal with the continuous nature of the data, traditional programming primitives are not of much help and mostly distracting. In order to make programmerss job easier, libraries providing higher level abstractions for stream data are introduced. These libraries usually discretise the continuous input stream into batches and give the programmer a local view of the stream allowing him/her to grab only the data from one batch at a time. Hence, programmer writes code to handle one batch of the discretised stream and the streaming library run this piece of code on all the batches sequentially as they arrive. For instance, programmer wants to use a flatmap function and he writes only one line of code where the flatmap function is called and the streaming library executes this line many times for each arriving batch as if in a loop. This abstraction takes care of all the chore of updating the data structures storing batch elements and bookkeeping loop variables to implement repetitive execution logic, letting the programmer concentrate only on the streaming logic of the application.

### 1.3.3 Distributed Systems

With the Internet data availability seems not to constitute a problem anymore. Aspects of storing, processing and mining those data amounts were reconsidered in the past years and led to the raise of new technologies and the mostly unloved buzzword Big Data. A single CPU can not accomplish the processing of those data quantities, especially if the algorithms are computationally intensive like most of the prediction algorithms from the field of Machine Learning which are widely used in data mining. To handle that the parallel execution of calculations is spread over a cluster of machines with a underlying system responsible for scheduling, load balance and fault tolerance [4, p. 10] . The forerunner of this cluster model was Hadoop, now several frameworks which extend this work are developed and allow wider functionality and significant performance improvements. Even though those frameworks are centered around data processing and ease of use software developers need to be aware of the distributed nature of such systems and parallel computation execution. This might be clear to those coming from a Distributed Systems background but challenging for data engineers or developers related to data mining who are more used to sequential arrangement of data processing.

## 1.4 Methods

### 1.4.1 Learning Methods

As mentioned previously in the report, the major aim of the project is to compare different model update techniques implemented as the variants of the same Naive Bayesian Classifier in terms of their capability in adapting/capturing concept drifts. For this purpose, we evaluate one offline (as a baseline) and three online classifiers.

### 1.4.2 Offline Learning

Offline learning assumes the access to all training points at once. From this initial training set the model is constructed to classify new data instances. This means after the training phase the learning is finished and in case of naïve Bayes the probabilities which are the basis for the decision rule are fixed. In contrast online learning is performing model construction with new datapoints arriving.

### 1.4.3 Online Learning

We consider three different online learning approaches two of which are slight variations of each other and both implement explicit-detection of concept drifts. As for, the third one, it is considerably different and it features implicit-detection methods for concept drifts. However, all three have a set of features in common. To be more specific, they all are single-classifier approaches and all utilise a 'sliding window' of recent data points to be used for the potential updates of the predictive model. What first two approaches differ is the way predictive model updates are triggered. The first

approach, called the bruteforce update, rebuilds the model at fixed interval by using the data items stored in the sliding window. The second variation called, error-triggered updates, waits for accuracy rate to go below a certain threshold which is hardcoded. Third model, called incremental updates, implements a different model update scheme than first two. Instead of rebuilding the model with recent data, it incorporate the recent data coming from the last window into the model. While doing so, it scales the feature vectors of the recent data by the learning rate and the past data by the value that learning rate subtracted by one. Learning rate simply specifies the amount of adaptation to the recent items to be done and the mount of 'forgetting' of the past models. Use of a dynamic learning rate elevated by increasing rates of error enables the incremental updates approach to 'silently' detect concept drifts and take the recent data into more consideration than the past by scaling down the feature vectors of the past models and taking the feature-vectors of the recent items almost as they are. One crucial point worth mentioning about incremental-updates model, it continues the training of the classifier with every incoming batch of data no matter what the learning rate is at the moment. This makes the training and test set simply perfectly overlap onto each other. In other words, every incoming item is used for updating the model after the prediction is done and the immediate feedback on the outcome of the prediction is received.

All these three models perfectly exhibit the textbook definition of online learning behaviour: alternating between exploitation and exploration although the lengths and the data sets used for their respective exploitation and exploration intervals differ substantially due to the their different underlying update mechanism which are explained above.

## 2 Empirics

### 2.1 Data

Data we are using is basically RSS(Really simple syndication) items streamed from BBC. An RSS Item is combination of some tags describing the content of the feed item and some meta-data about it such as publication date, url, etc. in XML structure. An example RSS item is given below.

Listing 1: An Example RSS Item

```xml
<item>
    <title>VIDEO: Jaguar announces UK jobs boost</title>
    <description>Jaguar Land Rover says it will create 1,300 new jobs
        to build Jaguar's first sports utility vehicle (SUV) at its
        Solihull plant in the West Midlands.</description>
    <link>http://www.bbc.co.uk/news/business-30787327#sa-ns_mchannel=
        rss&amp;ns_source=PublicRSS20-sa</link>
    <guid isPermaLink="false">http://www.bbc.co.uk/news/business
        -30787327</guid>
    <pubDate>Mon, 12 Jan 2015 21:02:57 GMT</pubDate>
</item>
```

.

Here, we are only interested in what is in the publication date, description and title tags. Publication date is crucial for computing interval-based prediction accuracy metrics and in-batch ordering. Description and title constitute the document we want to categorise in our case as we want to mine RSS feed items as opposed to mining the news articles themselves. Therefore, there is no need for fetching the actual article's text.

One difficulty of working with data streams is the volatility of the data hence it inherently requires real-time data processing. In other words, data should be handled as soon as it arrives. Since it is only possible to process data after implementing the text classifier and get it running, there was no straightforward way of utilising the data streamed during the development of the project. As Data scarcity evidently would pose a serious problem for a text classifier especially considering that we aim to detect concept drifts, we were compelled to find a way to facilitate the data being streamed while developing the application. To this end, we decided to first archive the stream data, then broadcast/stream this data through one of the local ports on the development environment and finally let the streaming application consume this archived data broadcasted data once the development is

complete. This way, we could take the time needed for the development and meanwhile did not lose any data since the very first day we conceived the idea of mining news feeds items for this project.

## 2.2 Software

For realising the stated approaches we are using Apache Spark and MLlib. Apache Spark is a open source processing engine for parallel processing of large scale data. Spark works on top of a distributed storage and uses a cluster manager cluster manager like Yarn or Mesos. For the purpose of this project the locale storage was used as simulated distributed storage which is integrated in spark for developing and testing reasons.

While the Spark core handles scheduling and load balancing the on top working modules provide additional functionality for streaming, Machine Learning algorithms and graph computation. The main programming abstraction in Spark is called RDD (resilient distributed dataset), a collection of objects partitioned across different machines for parallel programming. Beside map and reduce parallel operations on RDDs like filter, collect and foreach etc. are provided. For shared variables broadcast variables and accumulators can be used.

TODO: Something about Streaming

TODO: MLlib if used

## 2.3 Implementation

Our implementation effort can be broken in two main parts namely initial streaming and the implementation of the stream learning algorithms.

### 2.3.1 Initial streaming

The starting point for the implementation was designing and implementing a 'hook' for stream data. Usually, it is trivial to feed Spark Streaming with a stream but in our case we had to be careful with the amount of data that we can process for the observations in the limited timeframe would be left after completing the implementation specially considering that we aim to detect concept drifts, we were compelled to find a way to facilitate the data being streamed while developing the application. To this end, we decided to first archive the stream data, then stream this data for the second time through one of the local ports on the development environment and finally let the streaming application consume this manually streamed data once the development is complete. This way, we could take the time needed for the development and meanwhile did not lose any data since the very first day we conceived the idea of mining news feeds items for this project. Below diagram describes how this initial handling of the data is realised.

In the figure, Spark Streaming discretises the incoming stream by breaking the data into batches to be processed in first in first out order. We simply run this flow which also does after doing in-batch ordering of items according to their timestamps to accumulate a txt file for the RSS archive to be stremed later. Additionally, this streaming also creates a dictionary, everytime it runs inserts new vocabulary into the dictionary along with their associated hashcode which serves as the unique identifiers for the terms. In the source code, implementation of the Initial Streaming tasks can be found in rss.categorizer.util package, RSS Fetcher.java and FeedRefiner in particular.

ADD: FIGURE (Initial Streaming)

### 2.3.2 Stream Learning Algorithms

For the implementation of the stream learning algorithms as the core of our feed categorised application, we first attempted to use Spark Streaming and NaiveBayesian classifier provided by Spark MLlib in combination to facilitate convenient streaming abstractions of Spark Streaming library and also use the NaiveBayesian classifier which is a very flexible and customisable implementation of the algorithm and allows the use of different feature vectors other than term-frequency. This approach failed due to a very strict restrictions on shared state variables in Apache Spark. More specifically, in order to implement stream learning algorithms which does explicit-detection for conceptual drifts, some state variable needs to maintained which the control flow of the stream algorithm depends on. This means that there needs to be a globally shared state variable which is broadcasted whenever the state variables are updated. Apache Spark does not allow this to happen. For example, in the

case of error-triggered update approach, there should be a boolean variable that becomes true whenever error rate, which can be maintained by means of an accumulator, goes below the threshold. A potential solution to this problem was to use the updateState function provided in Spark Streaming. Although this appears to be specifically designed for overcoming the difficulty of keeping state variables with the streaming abstraction which essentially treats every line of code as statements in a loop, this does not solve our problem with not being able to retain a single state variable. This is because, what updateState function offers is to join the current stream batch onto the past one and in this fashion build an incrementally accumulated and key-indexed RDDs which is not suitable model for storing a single-variable state.

The workaround we found was having the driver program done the heavy lifting of the tasks of model building or model updating after collecting the RDD items partitioned across the nodes of the cluster by using the 'collect' function. This way, a state variable can be stored in the driver program and can be updated easily. However, this only does not fully solve the whole problem. This is because, the function 'collect' can be only used for RDDs. However, with the streaming enabled, instead of RDDs, we only can get out hands on the 'sequence' of RDDs. After having tried hard to get this combination of three different libraries namely Spark Core, Spark Streaming and Spark MLlib and experienced the abovementioned technical difficulties which can be arguably caused by either the design choices of the libraries or the current incompatibilities between them, and also having considered that the main challenge of the project is supposed to be rather on the study of the relevant concepts and the experiments rather than a specific library, we decided to give up on using Spark Streaming and 'simulate' the streaming behaviour by traditional programming primitives such as a plain loop which draws a batch of the labeled points in every iteration from a file.

All four variants of the learning algorithm has a similar structure. For the explicit-detection approaches what changes from one to another is the triggering logic of the model update procedure. Although the forth variant, the incremental updates, looks pretty different, when the details are abstracted away, it follows the same steps as the other ones. Therefore, these all four variants could be depicted in one general picture.

ADD: Figure (Stream Learning)

In the figure, we see that RSS archive file which is streamed by the initial streamer application whose details are discussed in the previous section is read into a collection located at the driver program. And then, from this collection the RSS items are drawn in batches (one or multiple RSS items in one batch, parametric) at every iteration of some loop that simulates the Streaming logic. The batches here corresponds to the discretised batches in Spark Streaming. Then, for the training and prediction, in order to create data points, we have DataPointGeneration component. It convertS feed items into labeled data points where label is the category of the individual RSS Item which is obtained by the hardcoded mapping of Feed URL and their corresponding variables of double type that represents individual topic categories. Furthermore, the datapoint part of the labeled datapoint consists of a sparse vector of features indices and feature values which are both represented as an array. Feature indices are simply obtained from the dictionary which is basically a mapping between the dictionary term to their corresponding hashcode. For each term occurring in the title or the description of the feed item, its corresponding hashcode is fetched from the dictionary. As for the feature values, term-frequency of the items are used assuming feed items title and description constitute a document . Then we have a TrainingUDF which is pictured as black box in this abstract picture as this is the component where different variants of stream learners differ from each other. First three variant maintains a sliding window of a number of batches where this number is a parameter. Whenever a training is triggered, the training data in this sliding window is used. In the forth variant, incremental updates, this sliding window always contains always the most recent batch. It is worth mentioning that, this training UDF is only triggered by some state variables which are controlled by a logic which is different for every variant. So, when it is fired, the part of the collection of RSS Items whose indices falls within the sliding windows indices are paralleled via the 'parallelise' call provided by Spark Core and then used for model training or model update purposes whose details again up to the model update variant that is used. After the parallelisation of the training dataset, parallelised RDDs can be passed to the MLlib's Naive Bayesian model building functions for training. After the data point generation, the initial training is the same for every variant. However, after the initial model is create, when the trainingUDF is fired again, all the variants except for the the incremental model trivially retrains the model based on the data in the training window. As for the incremental model, it 'updates' the model by incorporating the recent items from the training window into the bayesian

model. However, instead of using the term-frequency for feature values, the forth variant uses TF multiplied by the learning rate for the new training points and also overwrites the feature values of the vectors of the training data by multiplying by learning rate subtracted from 1. This way the Bayesian model is incrementally updated. In the case, the trainingUDF is not fired, it just does not do any training although it still updates its training window as more items arrive. Following the other branch from the data generation point, for every data point the prediction method provided by the NaiveBayes model object is called and predictions are obtained. Then, there is predictionUDF that calculates the cummulative accuracy rate as it receives the predictions and labels by simply checking for equality. Moreover, predictionUDF manages a very crucial task: it dynamically controls all the state variables which are used for triggering of trainingUDF and changing the learning rate for the incremental updates variant.

## 2.4 State Variables

Each variant has a different implementation of TrainingUDF triggering functionality and it is controlled by a boolean state variable. In the batch model, there is no need for training after the initial training, hence there is no triggering of TrainingUDF. In the case of bruteforce update variant, predictionUDF fires the trainingUDF at the fixed intervals which is a parameter for the stream learner implementation. For the error-triggered updates variant, predictionUDF fires the trainingUDF whenever the cumulative accuracy it computes goes below a threshold which is again a parameter that can be tuned by hand. As for the incremental updates model, there is again no need for triggering of the training in the incremental updates model since TrainingUDF of it is always on, it always updates the NaiveBayesModel. This is a direct implication of the implicit concept drift detection behaviour of this variant.

Learning rate variable is used only for the incremental updates variant. For this, we designed a function that converges to 0 as the prediction accuracy improves and to 1 as it decreases. So, after every batch, we know the change in the cumulative accuracy and if there is an increase in the overall accuracy, we increase the current domain index of the learning rate function by a step size and read the range value which is the learning rate itself. The functions range interval is the [0,1] and domain interval is the Java representation of all positive rational numbers. The function is depicted below for the domain interval [0,50]

## 2.5 Evaluation Setup

### Evaluation Measurse

To detect concept drift different error measures like expected error rate - or in case of dealing with ensemble classifiers - error decomposition can be used to observe changes in the probabilities which might relate to concept drift.[5] Since we are using MLlib and it is hard to reach the inner properties of the computed model to compute elaborate measures we took cumulative accuracy as a very simple evaluation measure. Accuracy is the proportion of correctly predicted labels among all predictions. We therefore computed successively the accurary with every new data point as cumulative accuracy over all datapoints and the cumulative accuracy per interval respectivly within a single test set.

### Offline classifier

In the time of three weeks we received 3659 data points. This rather small amount of data restriced us in choosing training and test set size since we are oberserving the classification behavior over time. For each approach our initial training phase contains 600 data instances which equates to the first three days of streaming news feeds.

Our first approach - the offline model - is used as a reference model. After the training phase we have chosen three disjoint test sets with 1000 datapoints each as seen in the graphic below to retrieve possible performance changes. Our size of training and test set differs from other test setups which usually take a 50:50 to 80:20 ratio for set sizes. Of course the classifier performs better the more training data is available, but since we are using sliding window approaches for the online classification with limited capacity we have chosen this split for compatibility in the comparison with the different scenarios and with the larger testset to get a more accurate error estimation. Due

to this experimental setup we are not expecting a very high perfomance accunracy in general but to detect changes in performance which could indicate some sort of concept drift.



Figure 1: Offline setup

## Bruteforce

As mentioned the second approach - the bruteforce method - takes a sliding window of 600 data instances. We applied retraining phase three times and after every training a testphase followed to measure the performance of the previous training. The testsets in this scenario are the same sets as used in the offline classification and contain 1000 data points. Except the last test set which will not be taken into further consideration.
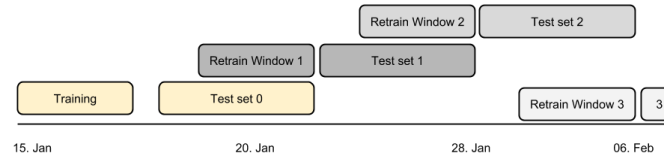


Figure 2: Bruteforce setup

## Error-triggered

As a variation of the bruteforce model with a fixed retraining interval we used a threshold triggered retraining which will rebuild the model out of the sliding window as soon as the accuracy of the model goes below a certain threshold. The window size of the error-triggered model consists as well of 600 data points like in the previous models. We set the accuracy threshold to 63% which seemed to be a reasonable choice due to our observations with the offline method. A sanity window of test points ensures that at least 300 new data points arrive before prematurely throwing out the rebuilt model. That means after every training at least 300 data points will be used to compute the performance and compare it to the threshold before on that basis the decision for retraining or not is made. Experimenting with smaller sanity windows lead to constant rebuilding. With the chosen sanity window rebuilding is executed six times.
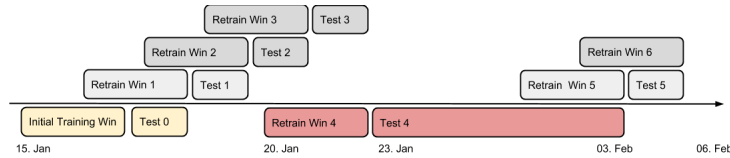


Figure 3: Error-triggered setup

## Incremental

In the incremental model, unlike the other update models, following an initial training-only phase, all the incoming stream items are always used for training right after the prediction. So, this way the immediate information whether the prediction was good or bad can be used for improving the model on the fly. In other words, predictive model building and testing phases perfectly overlap in incremental model update method. We used this to achieve a real-time response to the changes in the data arguably making the text mining application more resilient to the conceptual drifts. The initial training size of 600 is used for this model update method as well.

8

## 2.6 Evaluation results

The overall cumulative accuracy of the offline classifier is around 60% as visualised as a yellow line in the graphic below. The result is plausible since we did not deal with bias or variance and therefore weight the features nor preprocessed the data like using a stemmer. The per interval cumulative accuracy is shown alternating in green and blue. The first oscillations of the interval accuracys can not be taken into account which are approximately the first 250 - 300 test items since cumulative accuracy needs at least some prediction values to be reliable. As explained before the prediction model was computed once in the the first three days of receiving data points and applied to four disjoint testsets which were received in a later point of time. The training items are not included in the graphic, it starts with the first test point:
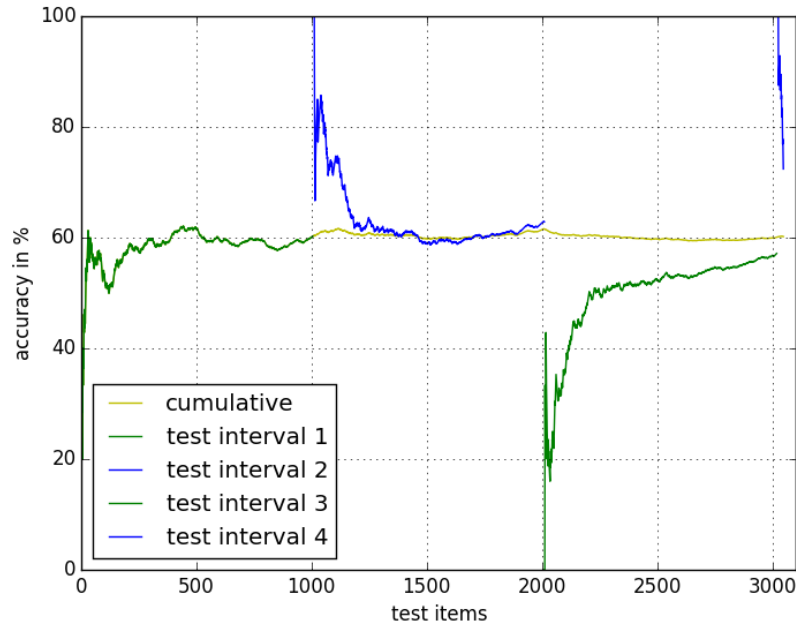


Figure 4: Performance of offline learning with received test items over time

While the first two intervals do not differ much the third interval shows a worse performance. This means the model which was computed in the beginning of our streaming does not fit very good to test items in a two weeks later point of time and might indicate some sort of concept drift.

The results for the bruteforce version which has the same testing setup but retrains the model after each test phase. The first test set will give the same results as the offline learning since they are both computed in the same way and the first retraining phase takes place after 1000 new instances arrived. The overall accuracies differ from the offline version by increased results for the bruteforce model after the first retraining around 63% to 64% of overall accuracy:

A one-to-one comparison of the different intervals of both models shows the following: The first retraining at data point 1000 improves the performance, the second retraining at test item 2000 does not perform as good as the first retraining on the following test set but is still better than the offline model. We assume that we observed concept drift, but the interval size might not have been chosen well since the accuracy drops after the second retraining. The second retraining might have taken place to soon, while the first retraining seemed approximately right in time. Nevertheless we can state that the bruteforce method does perfom better than the offline method even though retraining was chosen more or less randomly after 1000 test data points.
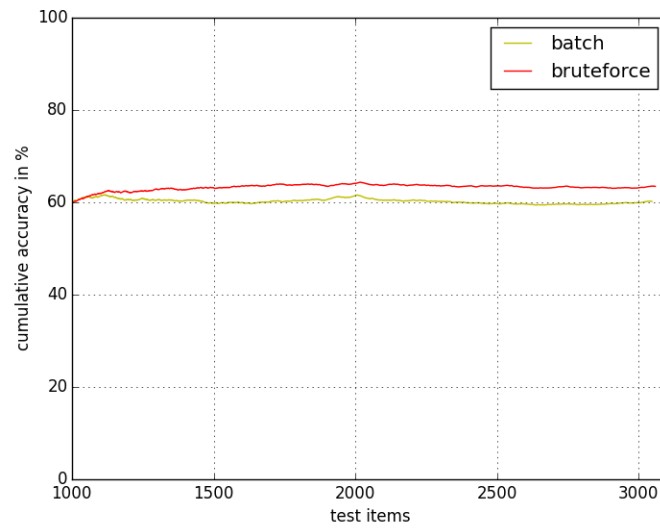
9

Figure 5: Overall accuracy for batch and bruteforce models after the first retraining
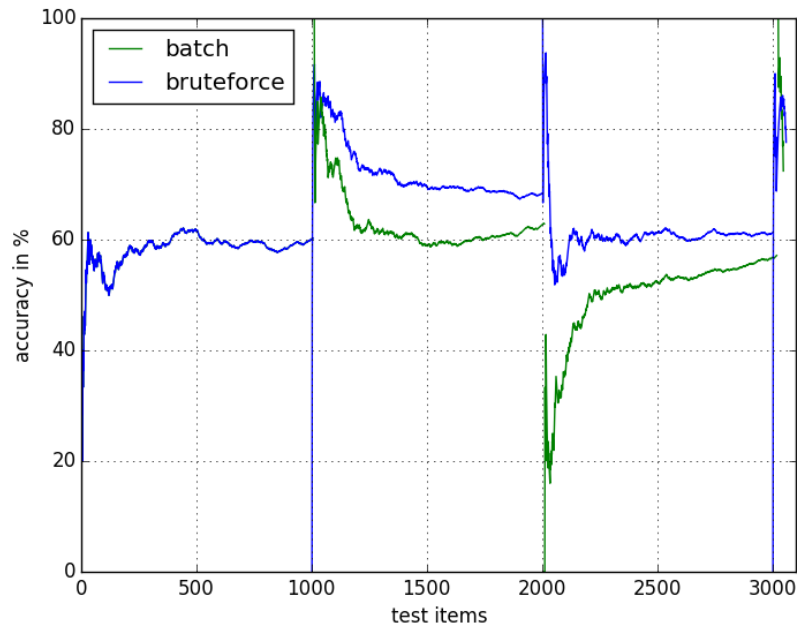


Figure 6: Per interval accuracy for offline and bruteforce method

10

To error-triggered model should handle this problem of fixed retraining phases and give us better estimation when retraining should be done. Applying a threshold of 63% percent as indication of retraining since we do not want to perform worse than the bruteforce method results in this graphic with the interval accuracies. It starts at 300 received testpoints which is the first retraining of the error-triggered model due to the sanity window:
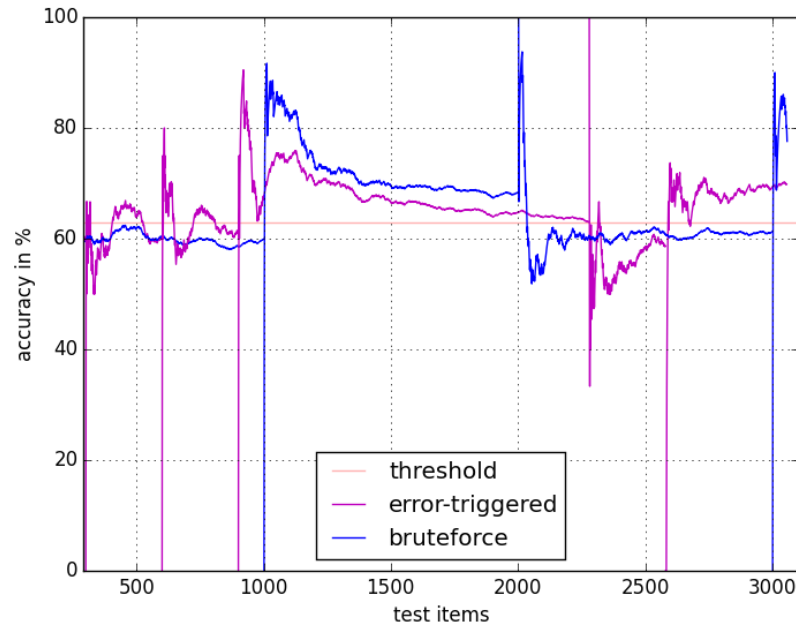


Figure 7: Per interval accuracy for error-triggered and bruteforce method

As seen here the retraining of the error-triggered approach takes place two more times at test items 600 and 900 which is the sanity window size. It followes by a stable model for over 1300 incoming test items and is retrained again two times. In this visualisation we have to consider that the test sets of the error-triggered models are much smaller than the bruteforce ones. However, the performance of the error-triggered models are not neccessarily much better than the bruteforce models except in the last interval starting at test items 2600. When applying smaller sanity windows constant retraining is performed due to the smaller test sets. The stable phase of the error-triggered approach starts at test item 900 just 100 data points before the bruteforce retraining is done but decreases shortly before 1000 nearly to the threshold. This drop is displayed as a less accumulative interval accuracy within this stable phase than the bruteforce approach. This means retraining the model in this point of time around receiving 1000 test items leads to a better prediction for a longer period. This might be caused by newsfeeds which are related to a certain topic more often and are included in training and test set. Out of this phase topic distribution might be broader and a classifier with 600 training items and 5 classes does just not perfom over 63% especially when data points are as sparse as those we used. For a profound analysis accuracy is also not the way to go.

Incremental as upper bound here?

## 2.7    Conclusion

- more training items, more items in general cause we are wanted to deal with large scale problems.. was not doable in the given time - parameter estimation important - tradeoff efficency execution of retraining and performance - combination, ensemble classifier consideration

## 2.8 Summary

what we wanted to do, which main problems we encountered, what we learned

## References

[1] Wolgang Ertel. *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. Vieweg + Teubner, Wiesbaden, 2008.

[2] Alexey Tsymbal. The problem of concept drift: Definitions and related work. Technical report, Department of Computer Science, Trinity College Dublin, Ireland, 2004. URL: https://www.cs.tcd.ie/publications/tech-reports/reports.04/TCD-CS-2004-15.pdf, last accessed on 21.01.2015.

[3] Ludmila I. Kuncheva. Classifier ensembles for detecting concept change in streaming data: Overview and perspectives. In *2nd Workshop SUEMA 2008 (ECAI 2008)*, pages 5–10, 2008.

[4] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[5] Jing Gao, Wei Fan, Jiawei Han, and Philip S. Yu. A general framework for mining concept-drifting data streams with skewed distributions. In *In Proc. SDM07*, 2007.