VANDEWIELE USE CASE

How to detect drift on an edge device.

From the TETRA project MLOps4ECM

Contents

1 Introduction

This use case was introduced by Vandewiele, a company that produces machinery for the textile industry including weaving machines, tufting machines, and spinning machines. The focus of this use case is their weaving machines. For industrial machines, like the weaving machine, it is crucial to monitor and predict faults as fast as possible to reduce their downtime. That is why Vandewiele, together with Leap technologies have created a machine learning model for fault detection. They did this by creating a dataset, collecting the machine motion cycle data each time a fault happened in an R&D environment. This dataset is then used to create two machine learning models. The first one is a postprocessing model which can detect the exact moment a fault has happened, given data from a full weaving cycle. The other model is a real-time model which for every data sample in the cycle decides whether a fault has happened or not. Vandewiele wants to use these two machine learning models in their production environment. However, they first want to implement a monitoring system to know if their machine learning models are degrading in performance in the production environment. What makes this task more difficult is the fact that weaving machine cycles happen very quickly, therefore, the model should run on the machine itself, or right next to it on a small edge device, to avoid constantly sending data to the cloud and back. Therefore, the chosen monitoring technique also needs to run on the edge. This raises the question of not only which technique would work best, but also, which devices are needed to run the chosen techniques.

Figure 1: Example of a Vandewiele machine

2 Background drift detection

To monitor the machine learning model, we use a drift detection technique, which is a method that detects when there is a change or drift in the production environment, causing a loss in model performance. This change is called concept drift, where the concept that the model has learned is different from the current concept. There are two types of drift detection techniques. The first one is supervised drift detectors. These detectors use true labels to calculate the error rate or accuracy of the model. If these metrics start to worsen, there is concept drift and the model should be retrained. An example of a supervised drift detection use case is the detection of spam emails. There is a machine learning model that sorts emails into two categories: spam and no spam. With each email, the user confirms whether it was sorted correctly, since they can report when an email is sorted incorrectly. This can be seen as the true label. If a user has to report a lot of wrongly sorted emails, the machine learning model should be retrained, since its output differs significantly from the true label. Supervised drift detectors help determine when this difference is large enough to indicate there is drift.

Supervised drift detectors assume that there are always true labels available. However, in a production setting, this is often not the case. In such cases, unsupervised drift detectors can be used which do not need true labels to detect drift. Instead, they look at shifts or drift in input data distributions to see if the current data window deviates from a certain reference data distribution, often the training data. These techniques assume that a big change in the input data automatically causes concept drift. There are four main steps in the unsupervised techniques. The first one is the window selection where the reference and current data window are selected. Then the data in these windows are modeled into a different form, which is used to measure the dissimilarity between reference and current data in step three. The last step is to evaluate the dissimilarity measure against a criterion to detect whether there is drift or not. This test often consists of comparing against a threshold.

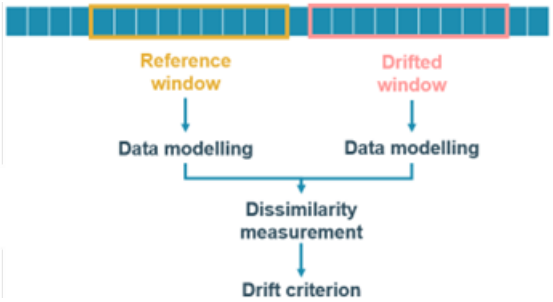

Figure 2: The four steps of unsupervised drift detection techniques

3 problem statement

3.1 problem setting

As mentioned in the introduction, two machine learning models were given that detect faults in a weaving machine. In addition to the machine learning models, two datasets were provided. The first dataset contains data from an R&D setup where they collected the training data of machine faults over the course of a couple of months, making sure to use different machine settings to create a generic dataset. This was the data used to train the machine learning models. The second dataset was collected from a production machine where first the sensors used to collect the data and a little later the brakes in the machine were changed. The data itself consists of sensor input data and some extra metadata information. Several types of labels are available as well. The first type are true labels, created by humans, which are only available for the training dataset. These are used for training the machine learning model. Other labels were also generated by two different non-machine learning algorithms which are now used to determine the fault location. This was available for most of the training dataset and for all of the second dataset. These outputs will be used as "true labels" during the supervised drift detection monitoring. A note to make here is that these values are also calculated from the input data which means that their results could also be affected by drift. However, since these are the only true labels available during the full dataset, these are the true labels we will use. The two datasets are used to make a single large streaming dataset which we will loop through sample by sample. At each step the data and model will be monitored using different drift detection techniques. In the next subsection we will explore the specific questions in more detail.

3.2 Research questions

The overall goal is to use the given datasets and machine learning models to create a drift detection monitoring system. Two main questions are examined. The first question is whether the change from R&D to production or the change of sensors and brakes have caused drift in the data or model performance. To answer this, several drift detectors are implemented and their results on the full dataset (training and production) examined to see if they detect any real drift. The second research question has to do with the size of the drift detectors. Since the machine learning model itself runs on the edge, it is best that the drift detection also happens on the machine so that drift is detected as quickly as possible without having to save and communicate all arriving data to the cloud. The question is then which type of edge device are suitable to deploy both the machine learning model and the drift detector. For this the memory usage and processing power of these detectors are evaluated and compared. Next to the self implemented drift detectors, we will also investigate the use of a monitoring tool on the edge. For this we will use Evidently, which is an open-source monitoring tool with which you can easily create different reports and test suites without requiring a cloud connection.

4 Research question 1

In this section we are going to investigate whether there is drift in the given data. First, we are going to explain the used method and the different drift detectors in sections 4.1 and 4.2. The experimental setup is explained in section 4.3, after which we move on to the experiment results and the conclusion of the first research question in section 4.4.

4.1 Method

The first question to answer is whether or not there is drift happening in the datasets. We were given three possible moments of drift. The first one is the change of R&D to production, the second one is the change of sensors, and the third is the change of brakes. These moments will be seen as our true drift labels. Using this, we will implement several drift detection methods from the state of the art and compare their results to these drift moments. We compare nine different drift detectors: four supervised and five unsupervised drift detectors. The specific detectors will be explained in Subsection 4.2.

Most drift detection techniques contain hyperparameters which can be tuned to the task. To optimize these hyperparameters for the detection of the three possible drifted points, hyperparameter optimization is used. This is implemented with the use of Optuna, a Python library which simplifies the optimization of hyperparameters of a certain algorithm. First, an objective function is chosen. In this case this includes running the chosen drift detector over the streaming dataset. Then the results of the drift detector are compared to the provided possible drift moments. Three metrics are calculated. The first one is the Mean Time to Detection (MTD). This is the time in samples between the moment of drift and its detection. This value should be as small as possible. The second metric is the Mean Detection Ratio (MDR). This ratio shows how many of the possible drifts are not detected over the total amount of true drifts. Again, a smaller value for this ratio implies better detection. Lastly, the number of False Detections (FD) is selected as the third metric. A false detection happens when the model detects drift but no drift has occurred since the last detection. A smaller number of false detections implies a better detector since we do not want a large number of false alarms. These metrics are calculated using the three possible true drift labels and the results of the drift detectors and are used for the multi-objective optimization, since we want to optimize three different metrics. We want to have a detector where all these metrics are as low as possible but are not NaN, which would happen if no drift was detected at all.

The result of a multi-objective optimization is a collection of possible hyperparameter solutions. These solutions performed best given the multiple objectives. Since there are multiple metrics to minimize, this front is not a single solution but a group of multiple optimal solutions, called the Pareto front. An example of a Pareto front can be seen in figure 3. The multi-objective optimization is implemented for the two provided machine learning models for both supervised and unsupervised drift detectors. For the supervised detectors we use the results of a rule-based algorithm. The choice of algorithm depends on the type of machine learning model evaluated. For the postprocess model we use a multiple insertions rule-based algorithm and for the realtime model we use a single insertion rule-based model as the true label of the model output.

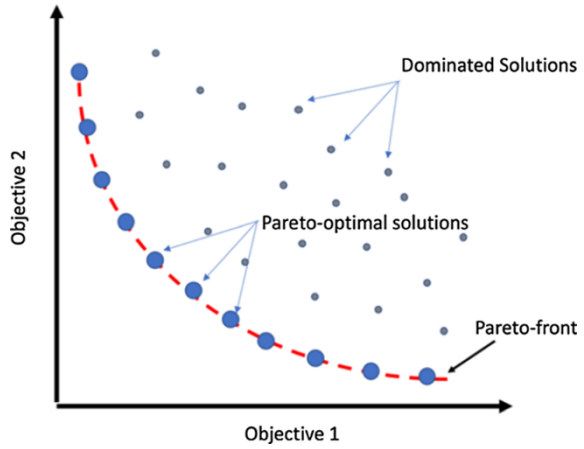

Figure 3: Image showing the Pareto front

4.2 Drift detection methods

In this section the different drift detection techniques used are explained. For each drift detection method the ranges for the hyperparameter search are provided.

4.2.1 DDM

Drift Detection Method (DDM), introduced by J. Gama et. al, was one of the first real concept drift detectors [3]. This supervised detector uses the error rate, which is the difference between predictions of the model and the true labels, to detect drift. If this error rate becomes larger over time, the model is performing worse, meaning that there is drift. To know when there is a statistically significant drift in the error rate, DDM keeps the distribution of the last n error rates as reference and then calculates the confidence interval of a new error rate belonging to this distribution. This is compared to two thresholds. These are a warning and a drift threshold. Drift is only detected when both thresholds are exceeded. If only the warning level was passed, the drift was seen as a false alarm. The warning level can also be used as the point to start collecting new data for retraining so when the drift level is exceeded, the retraining process can start almost immediately. Once drift is detected, the algorithm is reset and starts collecting new reference data. The hyperparameters for this method include the two thresholds and the length of the current window. Their chosen ranges are as follows:

- length of current window (integer) = 50 - 200 samples
- warning scale (integer) = 1-2 standard deviations = 84% - 95% confidence interval
- drift scale (integer) = warning scale - 4 standard deviations = 84% - 99.99% confidence interval

4.2.2 ADWIN

ADWIN or Adaptive Windowing [2], is a supervised drift detector which uses a sliding window of the model accuracy or error rate with an adaptable length for detecting drift. In that window it compares two subwindows, one containing older data and one containing new data. Using a statistical test, the means of the distributions of both subwindows are compared. The result of this test is compared to a certain threshold set by the hyperparameter δ. When the result is larger than the threshold, there is drift, otherwise there is none. This test is done for several subwindow lengths in the original window, if any of these tests indicate drift, it is considered a true detection. The size of the original window has a great effect on how fast drift is detected. Therefore, ADWIN looks at the mean of the window data to decide its length. If the mean in the window remains constant, the original window will grow. If this is not the case, the window will shrink, detecting drift more quickly. The hyperparameters for this technique are the number of samples between measurements, the δ, the maximum window length, the maximum subwindow length and the number of window combinations that should be tested. Their ranges during optimization are as follows.

- delta (float) = 0.0001 - 0.9
- maximum number of buckets (integer) = 3 - 10 ; decides the number of combinations tested
- measurement step (integer) = 16 - 128 samples
- minimum window length (integer) = 200 - 500 samples
- minimum sub window length (integer) = 50 - 199 samples

4.2.3 EDDM

The Early Drift Detection Method (EDDM) is a supervised detector developed to better detect gradual drift, compared to the original DDM. It looks at the average distance between errors from the machine learning model in a sliding window. When there is drift, the distance between errors tends to decrease when the distribution stays the same. It also works with a warning and drift threshold similar to the original DDM. There are three hyperparameters in this method. The first one is the number of error samples required to test whether a drift has occurred, which is the same as the length of the current window. The other two are the values of the warning and drift level thresholds. They have the following ranges during optimization.

- length of current window (integer) = 50 - 200 samples
- warning threshold (float) = 0.15 - 0.95 (confidence intervals = 1 - value)
- drift threshold (float) = 0.1 - warning threshold (confidence intervals = 1 - value)

4.2.4 STEPD

Nishida and Yamauchi introduced a supervised drift detection method using a statistical test of equal proportions to detect concept drift (STEPD) [7]. They detect drift by comparing the accuracy of the model of a current data window to the accuracy since the beginning of learning. They assume drift happens when the current accuracy is smaller than the overall accuracy and that there is no drift when the two accuracies are the same. They introduce the following test:

$$T(r_0, r_r, n_0, n_r) = \frac{|r_0/n_0 - r_r/n_r| - 0.5(1/n_0 + 1/n_r)}{\sqrt{\hat{p}(1-\hat{p})(1/n_0 + 1/n_r)}}$$

In this test there are two proportions. The first one is r_0/n_0 which is the number of correct classifications overall divided by the number of samples since the beginning of learning. The second proportion is r_r/n_r which is the number of correct classifications in the recent window over the number of samples in that window. This test is compared to the result of the test against a normal distribution which results in a significance level (p-value). This p-value is compared to a threshold. It indicates whether the null hypothesis was rejected or not. The null hypothesis is that $r_0/n_0 = r_r/n_r$, meaning that the number of correct classifications stays consistent over time. If the p-value is higher than a certain value, the null hypothesis holds and there is no drift. If the p-value is smaller than the threshold, the null hypothesis is rejected, and drift is detected. This method also works with a warning and drift level, similar to DDM and EDDM. The hyperparameters are the current window size, and the two thresholds, which have the following optimization ranges:

- window size (integer) = 50 - 200 samples
- warning alpha (float) = 0.7 - 0.9 confidence
- drift alpha (float) = warning - 0.99 confidence

4.2.5 Kolmogorov-Smirnov

The Kolmogorov Smirnov (KS) test compares the similarity between two cumulative distributions. It looks at the maximum absolute difference between these two distributions, as shown in figure 4 [1]. When this distance is larger, they are more distinct. In the unsupervised KS drift detector this test is used to compare the distribution of a certain reference window to a current detection window. Because the input consists of data samples and not a single value, the model outputs are used to compute the KS statistic. When the result of the KS test exceeds a predefined threshold, drift is detected. The hyperparameters for this test are the size of the reference window which matches the size of the current data window, and the threshold for drift detection. The ranges during the multi-objective search are as follows.

- window size (integer) = 50 - 300 samples
- threshold (float) = 0.7 - 0.95

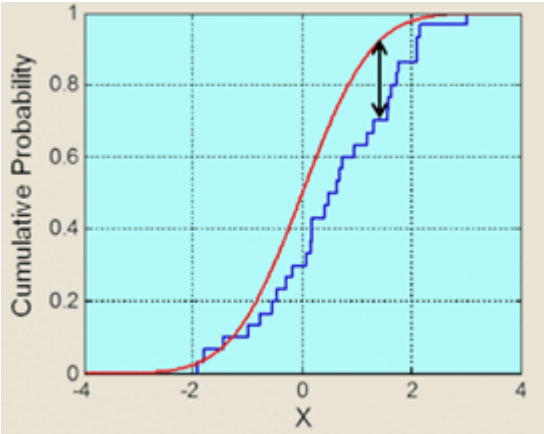

Figure 4: The distance measured by the KS test

4.2.6 D3

The discriminative drift detector (D3) uses a discriminative model to detect drift in an unsupervised way [4]. There are two sliding windows. One is the reference window, and the other is the detection or current window. Samples in the reference window are labeled zero and those in the detection window labeled one. Using this, a discriminative model is trained. If this model can correctly learn the difference between these two windows, there is drift. The metric to decide whether there is drift is the ROC area under the curve (AUC) of the machine learning model. The ROC curve is the curve plotted between the true positive and false positive rate of a machine learning model, an example is shown in figure 5. The AUC is a good way to tell if a model performs well in its learned task. An AUC closer to one means a better model. If this metric is higher than a certain threshold, the model can differentiate between the reference and current data which means that there is drift. There are again three hyperparameters for this detector. The first one being the size of the reference window. The second is the detection window size relative to the reference window. The last hyperparameter is the threshold value. They have the following hyperparameter ranges.

- w = reference window size (integer) = 100 - 200 samples
- ρ = current window size (in comparison to w) (float) = 0.1 - 0.4
- threshold (float) = 0.7 - 0.99

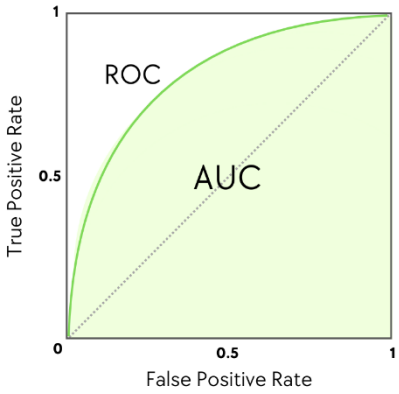

Figure 5: Image showing the area under the curve of the ROC curve

4.2.7 OCDD

The one-class drift detector (OCDD) creates a classifier using the first N samples in a data stream [5]. For each new sample, this classifier will predict whether it belongs to the same distribution as the input data or is an outlier. Then, a sliding window of these results is kept. If the number of outliers in this sliding window becomes larger than a certain threshold, there is drift. The size of the sliding window and the outlier threshold are the two hyperparameters for this drift detector.

- window size (integer) = 50 - 300 samples
- threshold (float) = 0.7 - 0.95

4.2.8 SPLL

The semi-parametric log-likelihood drift detector (SPLL) is an unsupervised drift detector which uses log-likelihood estimation to detect drift [6]. They first created a Gaussian mixture model based on the reference data window. This is a type of clustering where Gaussian distributions represent the clusters (Figure 6). Therefore, clustering the data should be possible for this method to be usable. This mixture model is then represented by a global covariance matrix, which shows the variances and covariances of the different Gaussian representations together. With this representation, the log-likelihood of each new data sample is estimated, using an upper bound for smaller computation needs. Drift is detected by comparing this upper bound to a threshold. The different hyperparameters used here are the number of clusters for the Gaussian mixture model, the size of the data windows and the threshold. They have the following ranges during optimization.

- window size (integer) = 50 - 400 samples
- number of clusters (integer) = 2 - 30
- threshold = 0.75 - 0.99

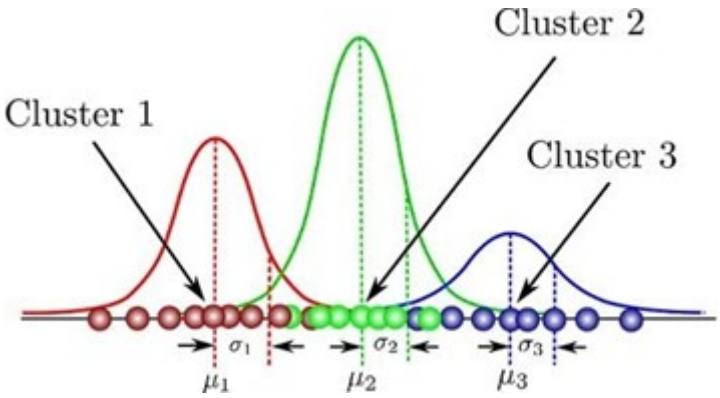

Figure 6: Clustering using the gaussian mixture models

4.2.9 BNDM

The Bayesian non-parametric drift detection method or BNDM, uses a Polya tree test, which is a non-parametric Bayesian test to detect drift [8]. First, they create Polya trees from the data of the reference and current data window. This type of tree takes a certain distribution and then creates a multi-resolution representation of the data as seen in Figure 7. This is done as follows. First the data distribution is taken in its whole, containing all data points, this is the first layer. Then the distribution is split into two where for each partition we count how many values there are. If you have for example data containing values from 0 to 10, then the split could be all data from 0 to 4 and from 5 to 10. This continues until a certain number of layers is reached. A typical way to split the data is based on the quartiles which means the data is evenly split until there are 8 partitions. These representations can be visualized in tree-form which is the Polya tree. The tree of the reference window is compared to the tree of the current window using a test called the Bayesian Factor. This is a certain statistical calculation which results in a value of one if both distributions are the same. If the Bayesian factor is lower than a certain threshold, drift is detected. The advantage of using these Polya trees is that they also show in which partitions the drift has happened. The hyperparameters for this detector are the size of the window, the threshold for the Bayesian factor, a constant needed for the creation of the Polya trees, and the depth of the Polya trees. They have the following ranges during optimization.

- window size (integer) = 50 -300 samples
- constant (float) = 0.1 - 0.95
- threshold (float) = 0.7 - 0.99
- maximum depth (integer) = 2 - 5 layers

4.3 experimental setup

In this section the setup of the experiments is explained. For our experiments both datasets were concatenated to simulate streaming data. We process this streaming data one batch at a time and at each batch a drift detection technique is run. At any given moment the last 200 data samples are saved to use as retraining data. If the drift detection technique detects drift, this data, along with the original training data, is used to retrain the provided machine learning model for 5000 loops. After training, the drift detection technique is reset and the drift detection is continued. Every time drift has been detected, this is saved in a list. This list is used afterwards to calculate the MTD, MDR and the number of false detections. Since each drift detection technique has different hyperparameters which can have an effect on the drift detection, Optuna is used for multi-objective hyperparameter optimization. The Pareto front of the multi-objective optimization is kept as

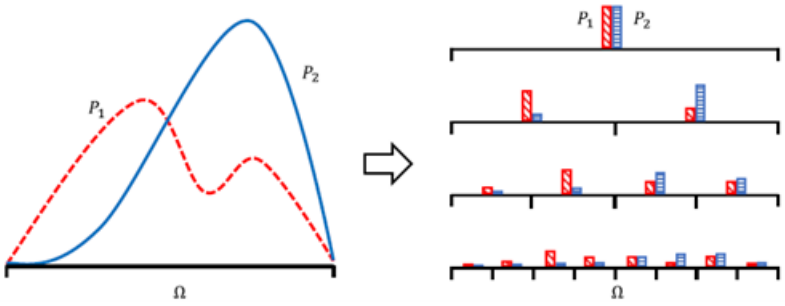

Figure 7: The multi-resolution of two distributions

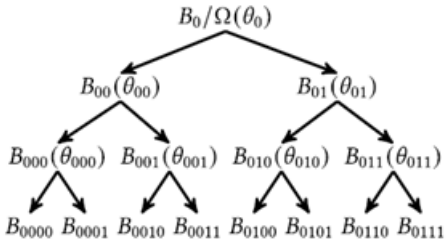

Figure 8: A Polya tree

the result. This whole optimization is done for every drift detection technique in two different ways. The first time we use the post-processed model as the model on which we want to find drift, and the true labels of the classification are provided from the multi insertion rule-based algorithm. The second time, the real-time model is used with labels derived from the single insertion rule-based algorithm. For each solution a figure of three graphs is created which can be used for evaluating the results. The discussion of the different results is done in the next subsection.

4.4 Results

For each drift detection technique we have created figures, each of which includes three graphs. The top graph shows the absolute difference between the predictions and the true labels during the online drift detection where the model is retrained. The second plot shows the same for the original, unchanged machine learning model, and the last graph shows when the drift detection technique has detected drift during the online detection. On all graphs three green lines can be seen. These correspond to the possible true drift moments. The first one is when the setup has changed from the R&D environment to the production environment. The other two lines come from when the sensors and then the brakes were changed. A few graph results are shown and discussed below from a few drift detection techniques, focusing on the realtime model. The conclusions of the results are similar for the postprocess model and the other techniques.

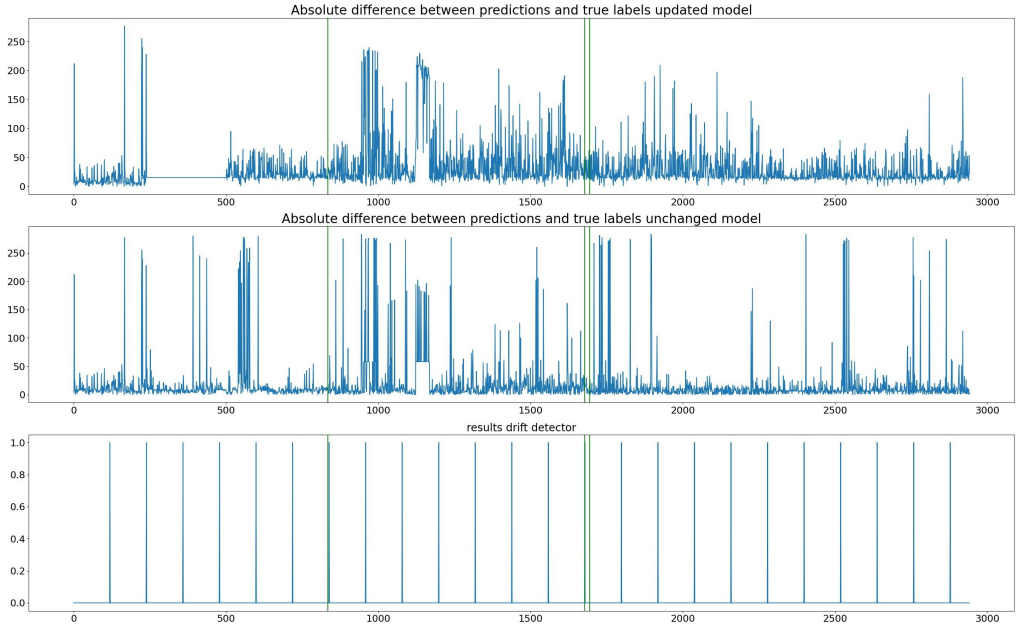

Figure 9: Monitoring results of BNDM (number of samples = 60, constant = 0.224, threshold = 0.76, maximum depth = 5)

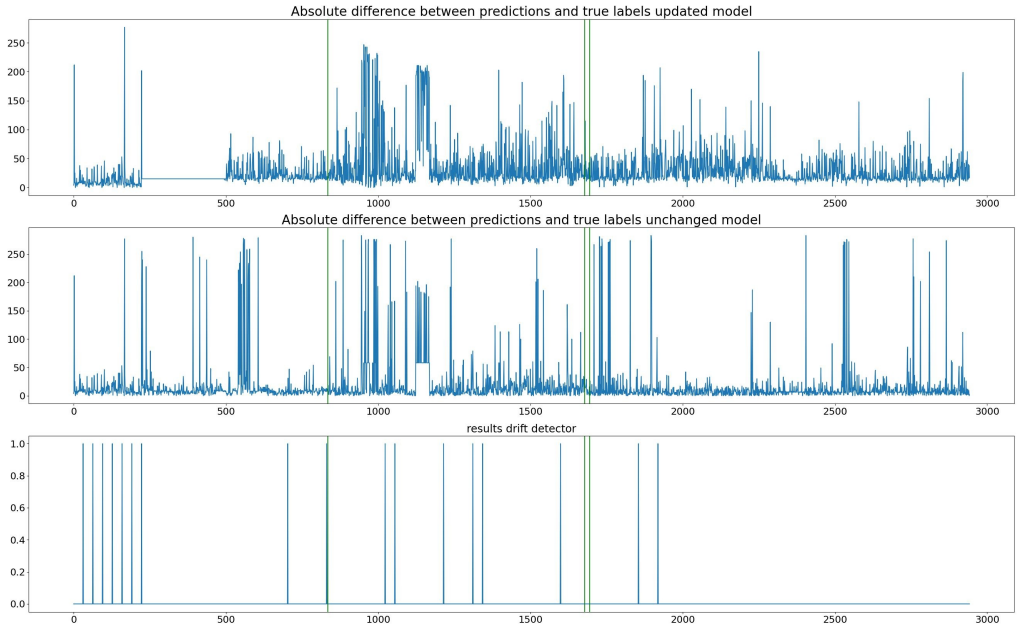

Figure 10: Monitoring results of ADWIN(Delta = 0.198, max buckets=3, min step size = 104, min window length = 487, min sub window length = 133).

When looking at the results, there are a few recurring solution types. The first type is the one where the drift detection technique is set too sensitive, which means that it detects drift every time it measures the data. This results in a drift detection graph which is almost always one or a graph that shows many peaks. This is a valid result for the Pareto front, since the mean detection ratio and mean time to detection will be zero. However, the number of false detections will be high. An example of this can be seen in Figure 9. You can see that each time there has been enough data gathered to create the tree in the BNDM drift detector, it detects drift, causing a reset of the drift detector. That is why you can see the very regular detections of drift in the bottom graph, which appear regularly spaced. If all resulting graphs had this type of result, this could mean

that there is no real drift since the detectors would only detect drift when they are too sensitive. However, since this is not the case, these graphs do not show us anything conclusive. There are some drift detection methods which always have this problem in this setting namely BNDM, KS, and SPLL. This shows us that the type of drift detector does affect when drift is detected. The other drift detection techniques do not have the detections in regular intervals. However, sometimes the amount of drifts detected are still too large to conclusively say that the three possible true drifts are actually drift, or if the drift detection technique is set too sensitive. This is for example the case for the ADWIN solution as seen in figure 10.

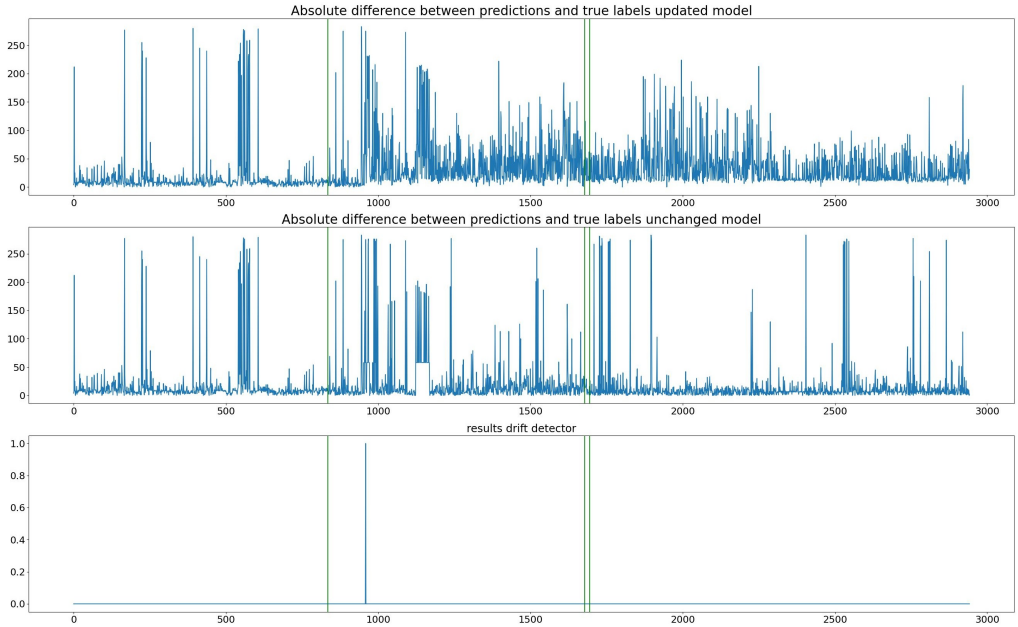

Figure 11: Monitoring results of DDM (warning scale = 2, N threshold = 154, and drift scale = 4).

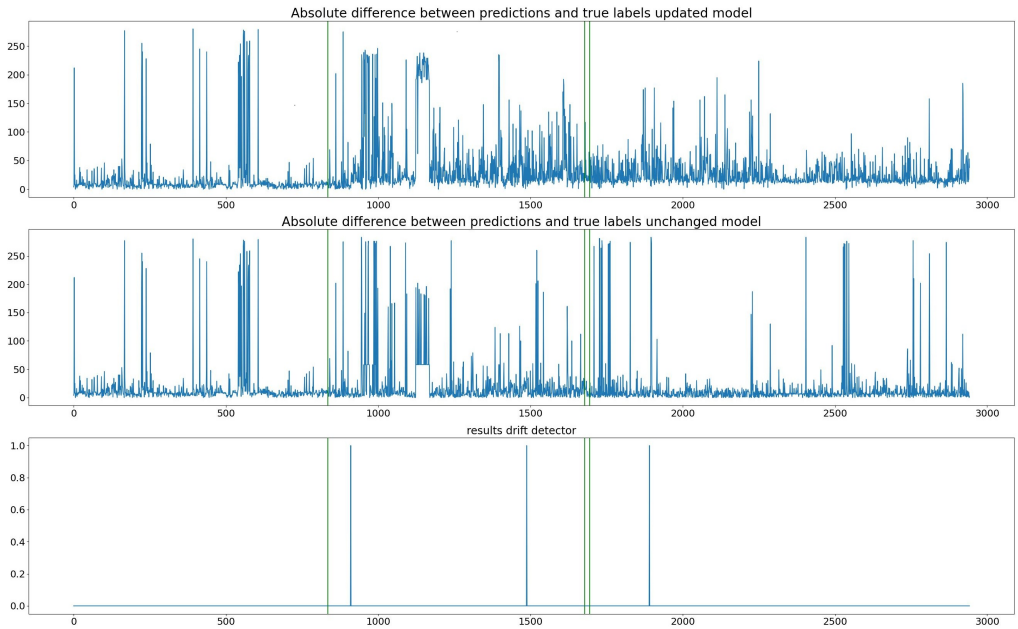

Figure 12: Monitoring results of OCDD (number of samples = 77 and threshold = 0.86).

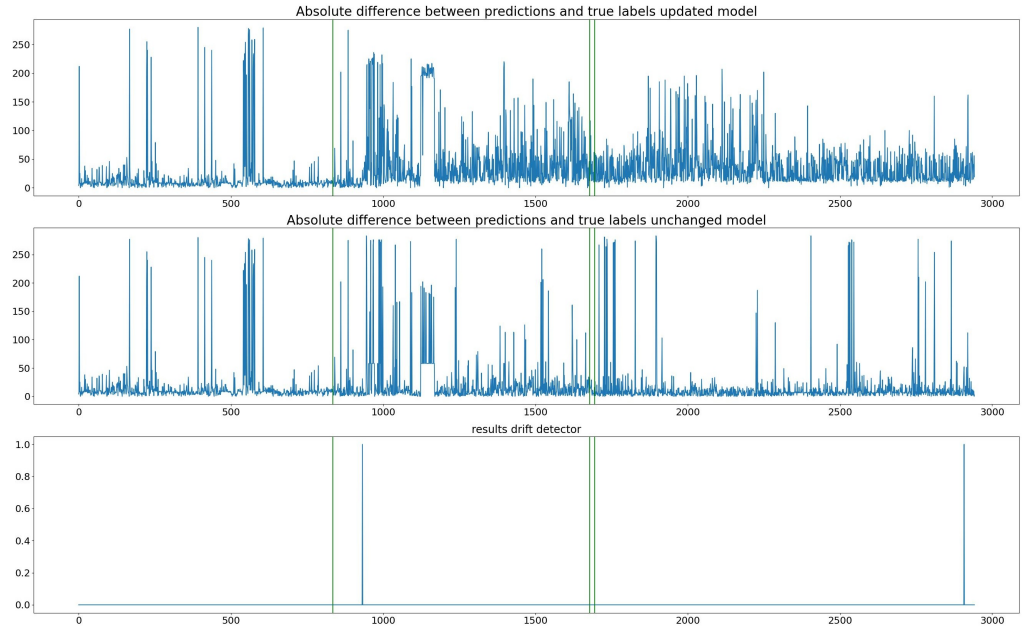

Figure 13: Monitoring results of EDDM (warning threshold = 0.83, N threshold=54, and drift threshold=0.218)

When looking at the other solutions there are some promising results. For example, when looking at the results shown in Figures 11 to 12. These are solutions from different drift detection algorithms which all show a similar drift detection point after the environment has been changed, suggesting there might be real drift happening there. However, there is a problem with these results which can be seen in the first two graphs of the figures. When comparing the retrained machine learning models to the original one, it is clear that the retrained machine learning model performs worse and when looking at the results of the original model, the original performance is also not that good. It is making large mistakes in a lot of places, even on its training data, as indicated by the large peaks in the graph. These phenomena can be explained by the fact that the provided data is very noisy. This is because of the fact that the provided data only contains faulty cases, which differ a lot from each other, showing little consistency, which means that the training data cannot be easily generalized and the model cannot learn the task without overfitting. This explains why the retrained model with the new data performs worse, because it is overfitted on the retraining data, and since this data is too different from the new, unseen data, the model does not show the expected improvement from retraining. This noisiness also explains why frequently the drift detection methods detect drift at every time they measure. No matter which reference dataset they use, it does not truly represent a stable concept since the faulty data is so different. Therefore they always detect drift, since they detect differences between fault graphs. The knowledge of this noisiness in the data makes it thus difficult to conclude that the detected drift points truly indicate drift because of the environment change, or if it is noise in the data.

4.5 conclusion

Looking at the different results, it is impossible to confidently conclude there is drift happening since the data, containing only faulty data at different time steps, is too noisy. To avoid this problem, it is more effective to use these drift detection models on the actual datastream measured by the sensors. This way, not only the noisy, faulty data is used for drift detection, but also the functional and likely more consistent operational data. This functional data is also measured at regular time steps, which makes it possible to have a consistent reference dataset. These results show that the type of data being evaluated and its preprocessing can have a strong effect on the results of the drift detection techniques. It is important to know what type of data your task expects. For drift detection this is a stable representation of the data which can be used as reference.

5 Research question 2

In this section we address the second research question: "Which edge devices are needed to run inference and drift detection on the edge?". In subsection 5.1 we discuss the experimental setup for measuring memory

and processing power. Next, in subsection 5.2, we show and discuss the results to conclude this question in subsection 5.3.

5.1 Experimental setup

The second research question involves the usability of drift detectors on edge devices. We measure this for two types of setups. The first setup uses Evidently, a monitoring tool, to detect drift using the mean absolute error and root mean square error between true labels and the model predictions. It is also used to save the resulting report in a JSON file since one of the main advantages of using Evidently over other techniques is their easy reporting. The second setup uses the different drift detectors to monitor the data. There are two main aspects to consider when working on the edge. The first one is the memory usage of the detector and the second one is the processing power needed to detect drift. The following two sections explain the experimental setup to evaluate both of these aspects for the different drift detection techniques.

5.1.1 Memory

Depending on the edge device used, there might not be a lot of memory available to keep track of the drift. Most methods compare a reference and a current data window. The storage of these data windows uses memory. Some methods train a machine learning model at each step which also uses memory space. Therefore, not all drift detectors are suitable for small edge devices of which some only have a few hundred Kilobytes available. It is therefore important to measure the memory usage of each detector and compare them. To do this, the tracemalloc() function in Python is used. This function can measure the peak memory usage between two points in the code. In our case it measures between the start and end of the monitoring loop. Between these points the drift detection method is initialized after which the drift detection method processes each data sample by executing the machine learning model and applying the drift detection algorithm. Retraining the model is excluded since this is not done on the edge device itself and it takes the same number of resources for each detector. The measurement stops when the detector has looped over each sample in the dataset. For each detector this measurement is done 35 times as each run may yield slightly different measurements. The results of the different detectors are then compared to one another using boxplots of the measurements. Boxplots show the median, quartiles and outliers for all results.

5.1.2 Processing power

To measure the processing power, a common metric is execution time. This is measured using the time library included in Python. This library includes the function perf_counter_ns() which captures the exact time at a certain point in the code in nanoseconds. This measurement is done twice for each sample in the monitoring loop. Once before the updating the detector with the new sample, and once after. In other words, for each sample the timing of the drift detector is measured. The average of these measurements serves as the final metric. Like the memory measurements, the processing power measurements are done 35 times for each drift detector and the results are compared using boxplots.

5.2 Experimental results

In this section the results of the memory and time measurements are shown and discussed. In section 5.2.1, we discuss the results of the Evidently setup. Afterwards, we discuss the memory results for the different drift detectors in section 5.2.2. We conclude with the timing results in section 5.2.3.

5.2.1 Evidently results

For the measured peak memory values and mean time values the boxplots can be seen in Figures 14 and 15 respectively. From these results, it is evident that the memory usage (104MB) and speed (50ms/loop) are pretty high. Evidently is not designed for measuring real-time drift detection as seen by the low speed. It can, however, be useful to create clear reports of several metrics which can be measured at regular time steps. These reports can be stored locally or transmitted to a cloud or server where they can be used in monitoring dashboards of the machine learning application. As can be seen in the memory boxplot, the Evidently loop uses a little more than 100MB of memory. This means that the type of edge device needed is in the category of microprocessors. This category includes devices like Linux boards, industrial PCs and PC-based PLC's. An important aspect of these microprocessors is that they have Linux or Windows installed since it is necessary to

run Python required to run the Evidently library. Some examples include the Jetson Nano and ARM Cortex A series.

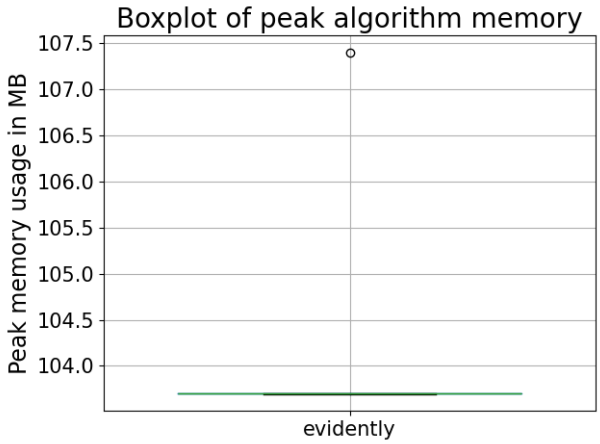

Figure 14: Boxplots of the peak algorithm memory usage for drift detection with Evidently.

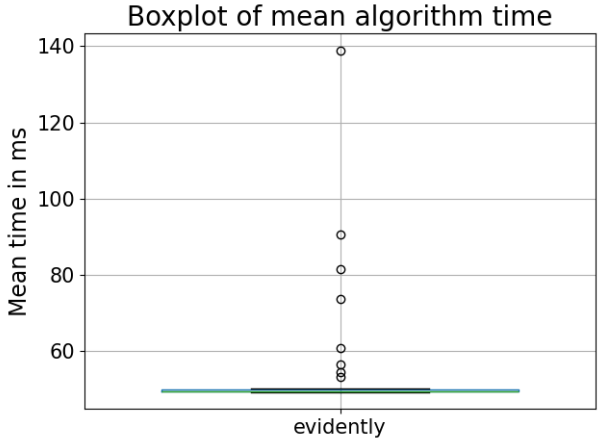

Figure 15: The boxplot of the mean algorithm time for drift detection with Evidently.

5.2.2 Memory detector results

From the measured peak memory values for each detector, a boxplot is created. The boxplots for all detectors can be seen in Figure 16. SPLL is the worst performing drift detector using more than 6000 KB (6MB) when running. Most of the other unsupervised techniques also use some more memory compared to the best result with their peak memory usage averaging between 2000 and 2500 KB. The fact that the supervised methods need less memory compared to the unsupervised methods is logical since they do not need to save the input data distributions or solve more complicated memory-intensive calculations like training a machine learning model to work. The KS-test is the only unsupervised drift detector that performs similarly well as the supervised methods, which makes sense since this method performs a relatively easy calculation compared to the other unsupervised methods. The peak memory boxplots of KS and the supervised methods can be seen in Figure 17. All these detectors need less than 1100 Kbytes for detecting drift which makes them more suitable for running on resource-constrained edge devices. Overall ADWIN was the best performing drift detector in terms of memory usage. In general these drift detectors can be implemented on smaller edge devices like microcontrollers. However, it is important to note that these microcontrollers cannot run Python, so the code must be translated to C or C++ to operate on these devices. Some example devices are the ESP32 and the ARM Cortex M series.

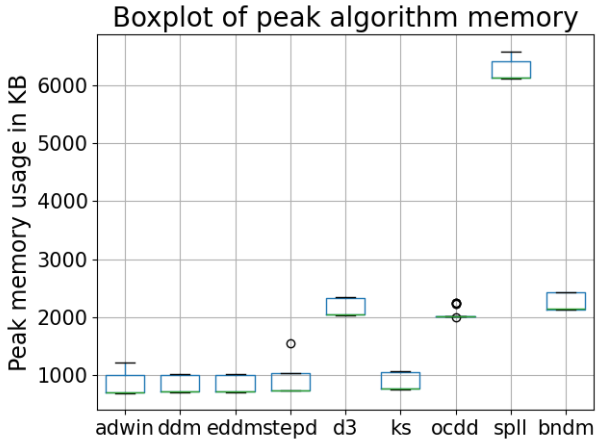

Figure 16: Boxplots of the peak algorithm memory usage for all the drift detectors, showing KS as the best unsupervised method and all supervised methods performing better then KS

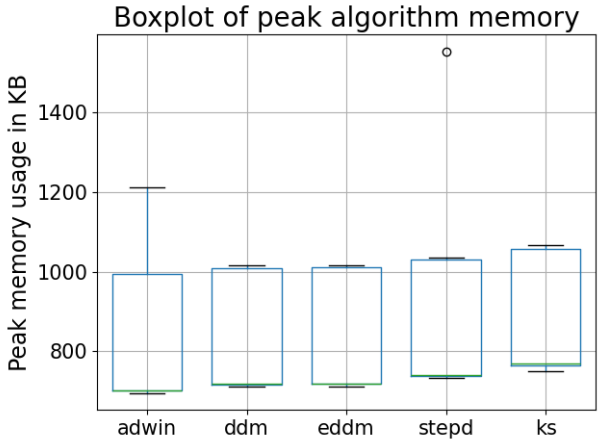

Figure 17: The boxplot of the peak algorithm memory usage for the best performing detectors, showing adwin as the best supervised method

5.2.3 Processing power detector results

The results from the time measurements are shown in Figure 18. You can see that the KS drift detection technique requires the least processing power. The worst performing drift detectors are SPLL with an average of around 65ms and BNDM with an average of around 22ms, but with an outlier near 95ms. Since it is important to detect drift in real time, the KS drift detector is looked at in more detail in Figure 19. There you can see that the average time is approximately 0.0072ms or 7.1μs. This proves KS to be the most suitable for real-time drift detection for time-sensitive tasks.

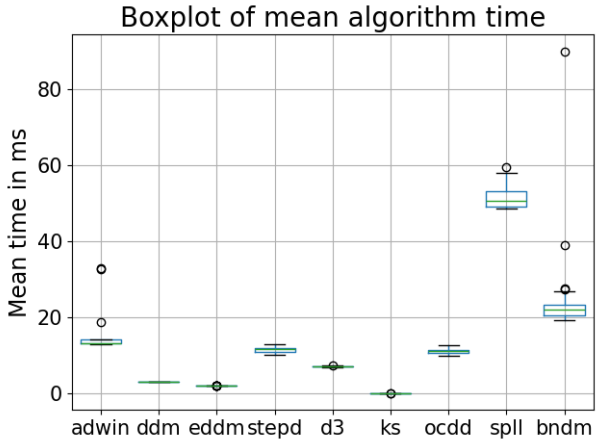

Figure 18: Boxplots of the mean algorithm time for all the drift detectors, showing KS as the best performing algorithm

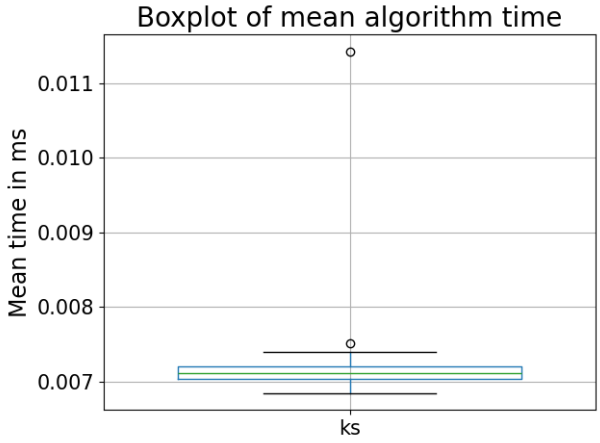

Figure 19: The boxplot of the mean algorithm time for the KS drift detection algorithm

5.3 conclusion

These results show us that it is possible to run the machine learning model on an edge device close to the machine. If drift detection has to happen at every step in realtime it is possible to use a microcontroller. However, the Python code must be translated to C code for it to work on these devices. If time is not as important, it is possible to use a tool like Evidently to easily create reports and test cases. However, a larger device like a microprocessor is needed.

6 Conclusion

In this report we discussed the use case of drift detection in the setup of fault detection in a weaving machine. We compared several drift detectors on the provided dataset to determine whether there is drift caused by environmental changes. From the results we can conclude that it is not possible to say whether or not there is drift in the data. This is due to the noise present in the data. the stability of the streaming data is crucial not only for the performance of the machine learning model but also for the effectiveness of the drift detection methods. We've also compared the memory and processing power of different drift detectors and Evidently. With this second set of experiments we can conclude that Evidently is best used for the creation of reports and test suites in non time-sensitive situations and its implementation would run best on a type of microprocessor which supports Python as needed for the Evidently library. If you want to detect drift in realtime it is best to use an implementation of one of the mentioned drift detectors, preferably KS since it is the fastest and uses limited memory. This implementation could even run on a microcontroller given that the code is translated to C/C++ first. However, before choosing a specific drift detector, it is important to evaluate its results on a

stable dataset which represents the use case to make sure it performs as expected.

References

[1] *Kolmogorov–Smirnov Test*, pages 283–287. Springer New York, New York, NY, 2008.

[2] Albert Bifet. Adaptive learning and mining for data streams and frequent patterns. *SIGKDD Explor. Newsl.*, 11(1):55–56, November 2009.

[3] João Gama, Pedro Medas, Gladys Castillo, and Pedro Rodrigues. Learning with Drift Detection. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Ana L. C. Bazzan, and Sofiane Labidi, editors, *Advances in Artificial Intelligence – SBIA 2004*, volume 3171, pages 286–295. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science.

[4] Omer Gözüaçik, Alican Büyükçakir, Hamed Bonab, and Fazli Can. Unsupervised Concept Drift Detection with a Discriminative Classifier. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2365–2368, Beijing China, November 2019. ACM.

[5] Omer Gözüaçik and Fazli Can. Concept learning using one-class classifiers for implicit drift detection in evolving data streams. *Artificial Intelligence Review*, 54(5):3725–3747, June 2021.

[6] Ludmila I. Kuncheva. Change Detection in Streaming Multivariate Data Using Likelihood Detectors. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1175–1180, May 2013.

[7] Kyosuke Nishida and Koichiro Yamauchi. Detecting concept drift using statistical testing. In Vincent Corruble, Masayuki Takeda, and Einoshin Suzuki, editors, *Discovery Science*, pages 264–269, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] Junyu Xuan, Jie Lu, and Guangquan Zhang. Bayesian nonparametric unsupervised concept drift detection for data stream mining. *ACM Trans. Intell. Syst. Technol.*, 12(1), November 2020.