Toulouse School of Economics

Data Mining Project : Adaptive Methods for Concept Drift Detection

Master 2 Data Science for Social Sciences

Emma Bastien, Anaïs Boughanem, Coralie Sorbet

Project A : D3S - Session 1 - December 16

Teachers: Colombe Becquart, Anne Ruiz Gazen

Academic Year: 2024-2025

Contents

1 Introduction

This project is part of the Data Mining Course. The goal is to explain the Concept Drift phenomenon and implement methods to deal with it. Concept drift refers to changes in the underlying distribution of data over time, which can decrease the performance of predictive models. Understanding and managing Concept Drift is crucial in various dynamic applications such as financial forecasting, fraud detection, and adaptive learning systems.

In this project, we aim to study Concept Drift in depth and implement the Adaptive Windowing (ADWIN) algorithm, as described in the paper *"Learning from time-changing data with adaptive windowing"* (Bifet and Gavalda, 2007) [3]. ADWIN is a method designed to detect and adapt to Concept Drift by dynamically resizing a sliding window to account for changes in data distribution. Additionally, we will implement ADWIN2, an improved version of the original ADWIN algorithm, which provides enhanced performance in detecting Concept Drift more accurately and efficiently.

For our experiment, we will use the *elecNormNew* dataset, which is available on GitHub [4]. The dataset contains the following columns: **date**, which represents the date of the data entry, in a fractional format indicating the time within the year; **day**, denoting the day of the week with values ranging from 1 to 7, corresponding to Monday through Sunday; **period**, indicating the time during the day, in terms of hourly intervals or fractions of a day; **nswprice**, referring to the electricity price in New South Wales (NSW); **nswdemand**, representing the electricity demand in New South Wales (NSW); **vicprice**, referring to the electricity price in Victoria (VIC); **vicdemand**, representing the electricity demand in Victoria (VIC); **transfer**, denoting the electricity transfer between New South Wales (NSW) and Victoria (VIC); and **class**, a categorical variable with two modalities, *UP* and *DOWN*, identifying the change of the price relative to a moving average of the last 24-hour.

The report is organized as follows: we will begin by introducing the statistical methods and tools used in drift detection. Then, we will explain the concept drift phenomenon, emphasizing its importance and effects on predictive models. Following this, we will explore the *elecNormNew* dataset to identify any signs of data drift. We will also review various methods for detecting and addressing concept drift. Afterwards, we will describe our algorithm implementation and evaluate the performance using the *elecNormNew* dataset. Finally, we will conclude by analyzing the results to assess whether the implemented methods improved the predictions.

In this project, Coralie was responsible for writing the introduction, conducting the statistical analysis, and explaining the Concept Drift phenomenon. Anais contributed by reviewing the various methods used to detect and manage Concept Drift. Emma focused on coding the implementation, presenting the results, and drafting the conclusion.

2 Simple exploratory analysis

As explained in the introduction the dataset used in the project is *elecNormNew* dataset, which is available on GitHub [4]. Indeed, this dataset is ideal for studying Concept Drift, as its patterns can evolve due to factors like variations in demand and changes in pricing.

2.1 Cleaning of the dataset

Before starting the exploratory analysis, the *elecNormNew* dataset required several cleaning steps to make it suitable for analysis. First, we addressed missing column names by assigning appropriate labels to each column. Next, we transformed the **date** variable, which was originally represented as a fraction between 0 and 1 (indicating a point in the year). To do this, we set the first date to January 1st, 2022, and then incrementally added the subsequent dates starting from this reference point. Then, we created a new variable to represent the month. Similarly, we processed the **time** variable, expressed as a fraction of the day, and converted it into hours. Finally, we modified the **day** variable, which initially ranged from 1 to 7, by renaming the values to correspond to the days of the week (Monday to Sunday). The dataset had already been scaled to ensure consistency across variables, so we did not need to repeat this step. Finally, the dataset contained no missing values, likely due to prior treatment, such as mean imputation, which eliminated the need for further imputation and allowed us to proceed directly with the analysis. While the dataset does not have missing values, some months, show values continuously around the mean, likely due to imputed missing values. Despite this, the dataset is still suitable for studying Concept Drift, as patterns can change over time.

2.2 Exploratory analysis on all the variables

In this section, we conduct an exploratory analysis to better understand the dataset's structure and relationships between variables. This preliminary step is essential to uncover any patterns, dependencies, or irregularities that could impact our subsequent analysis and the implementation of Concept Drift detection algorithms.

The first aspect of this analysis focuses on the correlation matrix (Figure 1), which highlights the linear relationships between the numerical variables in the dataset.

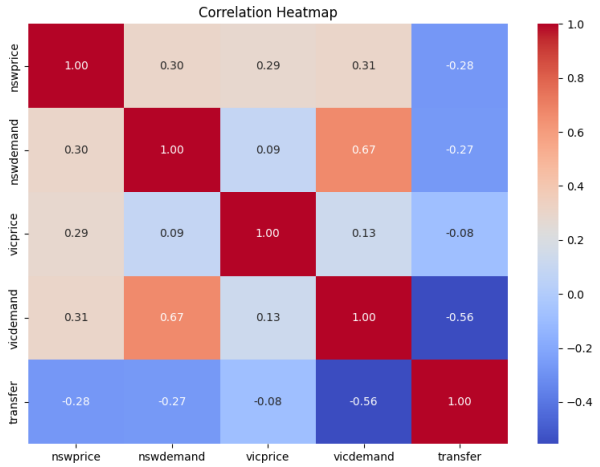

Figure 1: Correlation Matrix: Relationships Between Variables.

By visualizing the correlations, we can analyse the relationships between variables and evaluate whether multicollinearity might be a concern. For example, the moderate positive correlations between

nswprice and **nswdemand**, as well as **vicprice** and **vicdemand**, highlight the strong connection between electricity demand and pricing dynamics within each region. This observation aligns with expectations, as higher demand often drives up prices. On the other hand, the **transfer** variable shows minimal correlation with other variables, indicating that interregional electricity transfers are likely influenced by external factors, such as grid stability requirements or regulatory policies, rather than directly by market demand and pricing fluctuations. This distinction suggests that **transfer** may serve as a unique feature in subsequent analyses, unaffected by typical market dynamics.

In addition to examining variable relationships, we also analyze the distribution of the target variable, **class**, which represents the classification outcome for each observation.

Figure 2: Class Distribution: Frequency of Each Category.

The frequency of each category is shown in Figure 2, illustrating the relative distribution of the *DOWN* and *UP* classes within the dataset. The figure shows that while the *DOWN* class is slightly more frequent than the *UP* class, the distribution remains reasonably balanced. This balance is advantageous for evaluating the performance of Concept Drift detection algorithms, as it minimizes the risk of bias toward the majority class. A closer look at this distribution suggests that the dataset provides a fair representation of both categories, ensuring that the algorithms can reliably detect shifts in data distribution without being overly influenced by one class. This balance also enables a more accurate evaluation of the temporal trends associated with Concept Drift, which will be presented in the following section.

2.3 Exploratory analysis based on temporal trends

In this section, we delve deeper into temporal trends in the dataset. By visualizing how key metrics, evolve over different time intervals (hourly, daily, and monthly), we aim to identify patterns in the underlying data distribution. We mainly focused on electricity prices in New South Wales due to the greater variability observed in this region, as opposed to Victoria, where most values were clustered around the mean. This pattern suggests that earlier data treatments, such as missing value imputation, may have resulted in a less dynamic dataset for Victoria. By focusing on New South Wales, we gain a clearer understanding of the fluctuations in electricity prices and demand, which helps us uncover the factors driving these shifts. The goal is to better understand how these variables change over time and uncover potential factors driving shifts in electricity prices and demand, providing valuable insights for future modelling and analysis.

First, Figure 3 displays the hourly trend of average electricity prices in New South Wales.

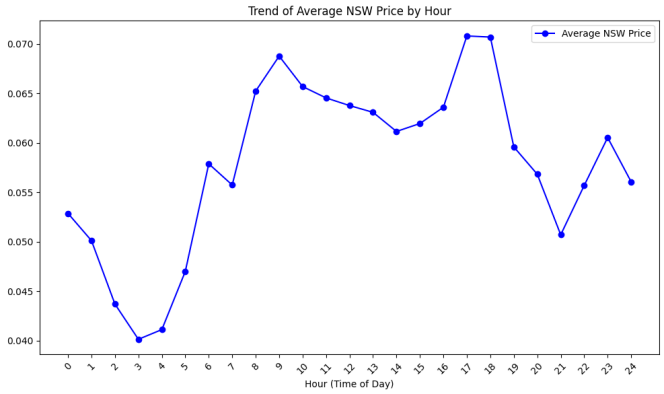

Figure 3: Hourly Trend of Average NSW Price.

Figure 3 illustrates the hourly trend of average electricity prices in New South Wales (NSW), showing distinct variations throughout the day. Prices are lowest during the early morning hours, which aligns with reduced electricity demand when most activities are minimal. They rise steadily during the morning as businesses and households begin their daily routines and peak again in the evening, likely driven by increased household usage after work hours. The evening peak is more pronounced than the morning peak, reflecting higher overall demand during this period. These patterns provide a clear picture of daily electricity usage behaviour.

Next, Figure 4 illustrates the daily trend of average NSW electricity prices over the days of the week.

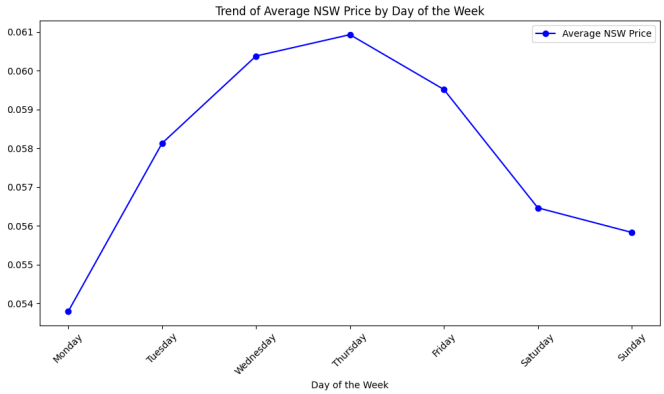

Figure 4: Daily Trend of Average NSW Price.

Figure 4 reveals a notable fluctuation in prices throughout the week. Monday has the lowest prices, which gradually increase until Thursday, probably due to the rising demand from businesses and industries during the workweek. After Thursday, prices begin to decrease until Saturday, reflecting a drop in demand as the weekend approaches. Sunday's prices are similar to Saturday's, suggesting consistent low demand over the weekend. These patterns align with typical behaviours of electricity consumption.

Finally, Figure 5 presents the monthly trend of average prices in NSW, offering a longer-term perspective on how prices change throughout the year.

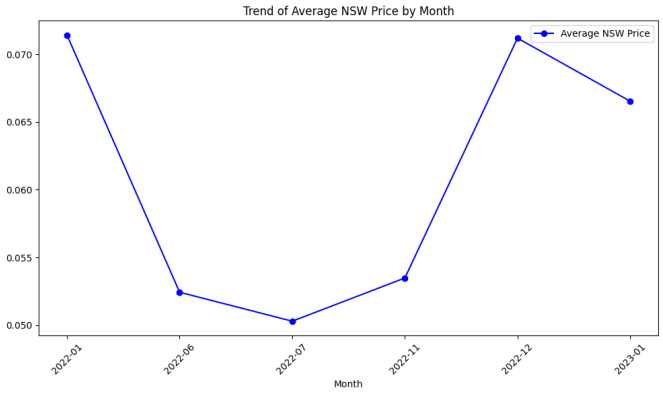

Figure 5: Monthly Trend of Average NSW Price.

Based on Figure 5, we can observe clear seasonal patterns in the data. Specifically, there is a higher demand during the winter months and lower demand in the summer. These trends are likely driven by seasonal changes in energy consumption.

Observing the evolution of trends over time is crucial for detecting Concept Drift, as shifts in these patterns may signal significant changes in demand dynamics or market conditions. These visualizations provide a detailed overview of temporal trends, serving as a foundation for detecting Concept Drift using adaptive algorithms like ADWIN, which will be introduced later.

3 The Concept Drift Phenomenon

3.1 The theory

Concept drift refers to the phenomenon where the statistical properties of data distributions change over time, leading to reduced performance of machine learning models trained on past data. This challenge often arises in dynamic environments, such as real-time data streams from social media or e-commerce platforms, where the relationships between features and labels evolve unpredictably (Hinder et al., 2023) [6].

Changes in data distributions can manifest themselves in different forms, each with distinct characteristics (Agrahari and Singh, 2021) [2]:

- **Real Drift:** This occurs when the relationship between input features and output labels changes, altering decision boundaries. For example, in fraud detection, new fraudulent patterns that are unrepresented in historical data might emerge.
- **Virtual Drift:** The input distribution changes, but the relationship between inputs and labels remains intact. For example, seasonal variations in user behaviour might shift input feature distributions without affecting class definitions.
- **Abrupt Drift:** A sudden and complete shift in data distribution, such as a sudden change in consumer preferences due to a global event.
- **Gradual Drift:** The transition between old and new distributions happens over time, as seen in market trends influenced by macroeconomic factors.
- **Recurrent Drift:** Old distributions reappear after a period, such as seasonal shopping patterns during holidays.

Below, figures illustrate these types of drift for better understanding (Agrahari and Singh, 2021) [2].

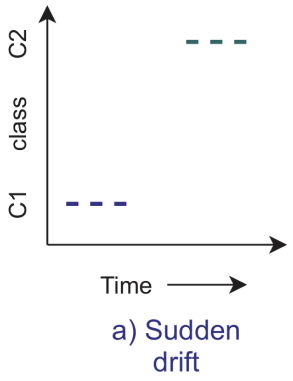

Figure 6: Illustration of sudden drift.

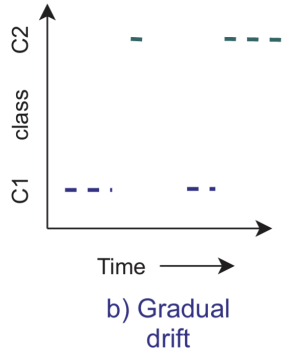

Figure 7: Illustration of gradual drift.

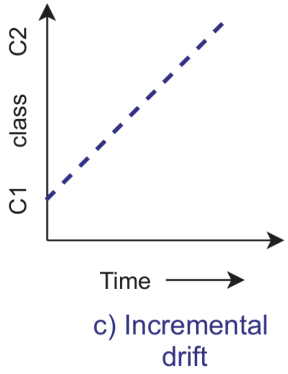

Figure 8: Illustration of incremental drift.

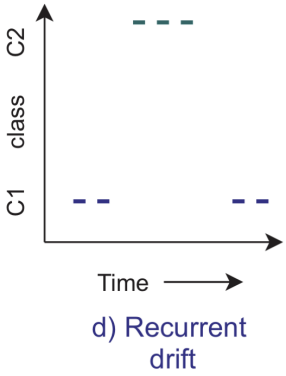

Figure 9: Illustration of recurrent drift.

Figure 6 shows an abrupt drift in which the distribution shifts suddenly, while Figure 7 demonstrates a more gradual transition. Similarly, Figure 8 illustrates incremental drift, where the distribution evolves in small gradual steps over time, while Figure 9 depicts recurrent drift, where previously observed distributions reappear cyclically or unpredictably.

Detecting and handling concept drift is key to keeping machine learning models accurate and reliable over time. As data evolves, techniques like adaptive windowing adjust the amount of data considered based on observed changes, with the ADWIN algorithm being one such example (Bifet and Gavaldà, 2007) [3]. Visualization methods, such as feature-based explanations, also play an important role in helping us pinpoint and understand the areas where these changes are occurring, making it easier to track how the data is shifting.

3.2 Detection in our dataset

Figure 10 below was implemented to detect some drift in the electricity price in New South Wales, in our dataset. We used the ADWIN method to do it, which will be presented in the following section.

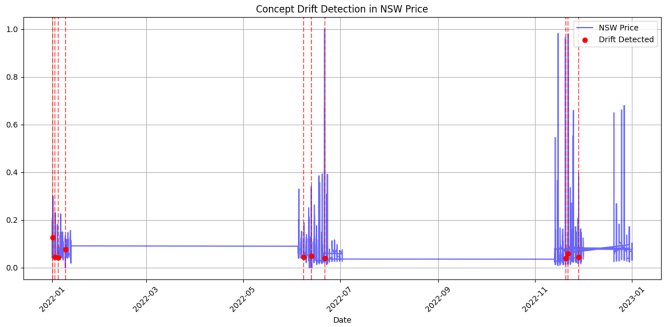

Figure 10: Concept Drift Detection in NSW Price.

It appears that there are at least ten drifts in the variable **nswprice**, suggesting significant changes in the distribution of electricity prices over time. These detected drifts indicate potential shifts in market dynamics, possibly influenced by factors such as supply-demand imbalances, policy changes, or external economic conditions. Detecting these drifts is crucial to adapting predictive models in real-time, ensuring their accuracy, and preventing outdated or erroneous predictions that could impact decision-making processes, such as pricing strategies, demand forecasting, or resource allocation in the electricity market.

3.3 How to Deal with Concept Drift

Concept drift, where the underlying data distribution changes over time, presents a significant challenge in data stream analysis. To address this, various methods have been developed to detect and adapt to concept drift. While ADWIN and ADWIN 2 provide a dynamic and statistically robust approach, other methods such as fixed-size windows and alternative drift detection algorithms have been widely used (Agrahari and Singh, 2021) [2]. The methods ADWIN and ADWIN2 will be explained in the next section as they correspond to the core idea that we want to implement in our project.

3.3.1 Fixed-Size Windows

The fixed-size window approach is one of the simplest strategies for handling concept drift. It involves maintaining a fixed-length sliding window that contains only the most recent data points. As new data arrive, older data points are discarded, ensuring that the model is trained on the most recent information.

Although computationally efficient and easy to implement, fixed-size windows have significant limitations:

- Sensitivity to Window Size: Choosing the optimal window size is critical, but challenging. A small window reacts quickly to changes but is prone to overfitting and noise, resulting in high variance. Conversely, a large window offers stability but may fail to detect rapid changes, leading to high bias.
- Inflexibility: Fixed-size windows cannot adapt to varying rates of change in the data, making them unsuitable for streams with dynamic behaviour.

3.3.2 Gama's Drift Detection Method (DDM)

Gama's Drift Detection Method (DDM) monitors the error rate of a predictive model to identify concept drift (Gama et al., 2004) [5]. It assumes that the error rate follows a binomial distribution under stable conditions. DDM calculates the mean (p_t) and standard deviation (s_t) of the error rate over time:

$$s_t = \sqrt{\frac{p_t(1-p_t)}{t}},$$

where p_t is the error rate at time t, and t is the number of samples. Drift is detected when the error rate exceeds a predefined threshold, indicating a significant change in the data.

While effective for detecting abrupt changes, DDM struggles with gradual drift, as smaller shifts may not trigger the error threshold. Furthermore, it relies on labelled data to compute error rates, which is not always feasible in real-time applications.

3.3.3 Exponentially Weighted Moving Average (EWMA)

The Exponentially Weighted Moving Average (EWMA) technique tracks the moving average of the error rate, giving more weight to recent observations. This makes EWMA particularly effective for detecting gradual concept drift, as it responds quickly to small, incremental changes in the data distribution. However, it may fail to detect sudden drift when the historical data is still weighted too heavily.

3.3.4 Ensemble Approaches

Ensemble methods combine multiple models to detect concept drift, making them particularly effective for handling both abrupt and gradual changes. Techniques like the Hybrid Forest, which combines Hoeffding Trees and Random Forests, maintain multiple models that are updated when drift is detected.

A Hoeffding Tree is a decision tree algorithm specifically designed for streaming data. It is particularly useful in environments where data arrives sequentially, and the model must adapt incrementally without storing the entire dataset.

The key feature of a Hoeffding Tree is its ability to make decisions on the best features for splitting nodes in the tree using partial information. Instead of waiting to see all the data, the algorithm decides when enough data has been observed to make a reliable decision, using Hoeffding's inequality to estimate the error of a feature's performance.

$$P\left(|\hat{\mu} - \mu| \geq \epsilon\right) \leq 2\exp\left(-2n\epsilon^2\right),$$

where:
- $\hat{\mu}$ is the estimated mean of the feature,
- μ is the true mean of the feature,
- n is the number of examples seen so far,
- ϵ is the maximum allowed difference between the estimated mean and the true mean.

The tree splits at the feature that minimizes the error, and it updates itself as new data arrives. This makes the Hoeffding Tree highly efficient in processing large, continuous data streams and adaptable to changing data distributions.

When combined in an ensemble, such as in the Hybrid Forest method, multiple Hoeffding Trees work together to provide a more robust prediction, each trained on different subsets of the data or in parallel, which improves overall performance in detecting drift.

4 Methodology to detect Concept Drift

4.1 Objective and Core Idea of ADWIN

Handling time-evolving data is a significant challenge in machine learning, as traditional algorithms assume static data distributions. ADWIN (ADaptive WINdowing) addresses this challenge by maintaining a variable-size window that adapts dynamically to the data stream. When no change is detected, the window grows to reduce variance; when a change is detected, it contracts to focus on the most recent data points while discarding outdated information. This allows ADWIN to efficiently handle concept drift without requiring manual parameter tuning, unlike traditional fixed-size windows.

ADWIN's core advantages lie in its ability to balance:
- Reaction time: Quickly adapting to changes by shrinking the window when concept drift is detected.
- Stability: Maintaining statistical reliability by expanding the window during stable periods.

The threshold for detecting changes, ϵ_{cut}, is determined by the formula:

$$\epsilon_{cut} = \sqrt{\frac{2}{m} \cdot \sigma_W^2 \cdot \ln\left(\frac{2}{\delta}\right) + \frac{2}{3} \cdot m \cdot \ln\left(\frac{2}{\delta}\right)},$$

where m is the harmonic mean of the sub-windows W_0 and W_1, σ_W^2 is the variance, and δ is the confidence parameter.

ADWIN can integrate with learning algorithms to track statistics (e.g., averages, counts) in real time, ensuring model accuracy as data evolves. However, the original ADWIN suffers from memory and time inefficiencies. ADWIN2 optimizes this by reducing memory usage and computational cost to $O(\log W)$, where W is the window size while maintaining the same performance guarantees.

4.2 Step-by-Step Methodology and Explanation

The ADWIN algorithm operates on a sliding window W, which is used to store the most recent observations from the data stream. When a new data point x_t arrives, it is added to the most recent sub-window, denoted as W_1. This sliding window continually updates as new data points come in. The window size is not fixed; it adapts based on changes detected in the data.

The mean of the window W is calculated as:

$$\mu_W = \frac{1}{n_W} \sum_{i=1}^{n_W} x_i,$$

where n_W is the number of data points in the window W and x_i represents each individual data point. This means is essential for tracking shifts in the data distribution over time.

Similarly, the variance of the window W is calculated using the formula:

$$\sigma_W^2 = \frac{1}{n_W} \sum_{i=1}^{n_W} (x_i - \mu_W)^2.$$

This variance helps quantify the spread or uncertainty of the data in the window, which is used to identify whether the data distribution is stable or has changed.

At regular intervals, ADWIN partitions the current window W into two sub-windows, W_0 and W_1. W_0 represents the older data, while W_1 contains the more recent data. The size of these two sub-windows is n_0 and n_1, respectively.

The algorithm calculates the means of both sub-windows, μ_{W_0} and μ_{W_1}, which are:

$$\mu_{W_0} = \frac{1}{n_0} \sum_{i=1}^{n_0} x_i \quad \text{and} \quad \mu_{W_1} = \frac{1}{n_1} \sum_{i=1}^{n_1} x_i.$$

Next, the algorithm compares the means of the two sub-windows W_0 and W_1. To determine whether the difference between these means is statistically significant, ADWIN uses a predefined threshold ϵ_{cut}. The threshold ensures that only significant changes are detected, avoiding false positives caused by random fluctuations in the data.

The threshold ϵ_{cut} is computed as follows:

$$\epsilon_{cut} = \sqrt{\frac{2}{m} \cdot \sigma_W^2 \cdot \ln\left(\frac{2}{\delta}\right) + \frac{2}{3} \cdot m \cdot \ln\left(\frac{2}{\delta}\right)},$$

where: - m is the harmonic mean of the sizes of the two sub-windows W_0 and W_1, calculated as:

$$m = \frac{1}{\frac{1}{n_0} + \frac{1}{n_1}}.$$

- σ_W^2 is the variance of the data in the window W, representing the spread of data points. - δ is the confidence parameter that controls the trade-off between sensitivity and false positives.

If the difference between the means $|\mu_{W_0} - \mu_{W_1}|$ exceeds the threshold ϵ_{cut}, the algorithm detects a change in the data distribution, known as concept drift.

When a change is detected, ADWIN reacts by shrinking the window. This is done by discarding the oldest data points from W_0, focusing on the more recent observations in W_1. The window size decreases to reflect the more recent data, which ensures that the model remains up-to-date with the latest distribution.

The number of data points removed from the window is determined by the difference between the sizes of the two sub-windows. The shrinking continues until the change is no longer significant, at which point the window size stabilizes.

ADWIN uses a hierarchical bucket structure to efficiently manage memory. Instead of storing each data point, the algorithm aggregates data into buckets. Each bucket holds statistics such as the sum and variance of the data points in that bucket. This approach reduces the memory requirements, especially for large data streams.

The mean of each bucket k is computed as:

$$\mu_k = \frac{S_k}{n_k},$$

where S_k is the sum of the data points in the bucket, and n_k is the number of data points in the bucket.

When the number of buckets in a row exceeds a predefined limit, the algorithm merges them into the next row. This logarithmic compression ensures that memory usage grows slowly concerning the window size, making ADWIN scalable for large data streams.

ADWIN continuously adjusts the size of the window as new data points arrive. During periods when no change is detected, the window expands to include more data points. This expansion helps reduce variance, providing a more stable estimate of the distribution. Conversely, when a change is detected, the window contracts to focus on the most recent data points, quickly adapting to the new distribution.

The adjustment of the window size is dynamic, and the number of data points in the window is updated as the window shrinks or grows:

$$n'_W = n_W - \text{removed points},$$

where n'_W is the new window size after shrinking.

This dynamic behaviour ensures that ADWIN remains responsive to concept drift while maintaining statistical stability.

The bucket structure and dynamic window resizing allows ADWIN to be memory-efficient and scalable. By storing aggregated statistics in buckets instead of individual data points, the algorithm can handle large-scale data streams without overwhelming memory resources. The logarithmic growth of memory usage concerning the window size ensures that ADWIN can scale efficiently, even in real-time, high-velocity data stream environments.

4.3 ADWIN2: An Optimized Version of ADWIN

While the original ADWIN algorithm is effective in detecting concept drift, it suffers from inefficiencies in time and memory usage as the window size grows. To address these issues, we introduce ADWIN2, an optimized version that reduces both memory and computational costs while preserving the core functionality and performance of ADWIN.

The key improvement in ADWIN2 is its ability to scale more efficiently. ADWIN2 uses a data structure that reduces memory usage to $O(\log W)$, where W is the window size, in contrast to the linear scaling of the original ADWIN.

Additionally, ADWIN2 reduces the time complexity of updates. While the original ADWIN algorithm processes updates in $O(n)$, ADWIN2 processes them in $O(\log W)$ time by using a logarithmic structure for its buckets, leading to faster updates and lower computational overhead.

Despite these optimizations, ADWIN2 retains the core features of ADWIN, such as partitioning the window into sub-windows and comparing their means to detect concept drift. The threshold for detecting changes (ϵ_{cut}) is calculated in the same way, ensuring the same performance guarantees.

ADWIN2's improved memory and time efficiency make it suitable for real-time, large-scale applications. It is particularly beneficial in scenarios where data streams are high-velocity and large-scale and in situations with limited system resources, such as embedded systems or mobile devices.

4.4 Theoretical Guarantees

ADWIN can detect concept drift with high accuracy while controlling the rates of false positives and false negatives, based on the statistical nature of the data.

ADWIN's performance is controlled using the confidence parameter δ, which regulates the statistical significance required to declare a change. This parameter balances sensitivity and specificity. The probability of a false positive (incorrectly detecting a change) is bounded by δ. Specifically, the probability that ADWIN incorrectly detects a change when no drift has occurred is no greater than δ:

$$P(\text{false positive}) \leq \delta.$$

Similarly, the probability of a false negative (failing to detect a real change) is also controlled, ensuring that a change will eventually be detected after it has occurred, even if it is small. The algorithm detects changes in a way that minimizes both false positives and false negatives, maintaining a reliable performance over time.

The window size adjustment process is theoretically guaranteed to strike an optimal balance between reaction time and variance (estimate uncertainty). ADWIN's window size adapts to changes in a way that minimizes statistical errors. Specifically, the window size adjusts so that reaction time and variance are balanced, providing timely detection without overreacting to noise.

The theoretical analysis ensures that ADWIN operates optimally even without knowing the rate of change in the data stream. This makes it highly flexible and capable of handling various types of data streams, from high-velocity streams to more stable ones.

4.5 Comparison with Fixed-Size Windows and Gama's Drift Detection

One of the key advantages of ADWIN is its ability to dynamically adjust the window size based on observed data, in contrast to traditional fixed-size windows. Fixed-size windows maintain a predefined number of recent observations, discarding older data. While simple to implement, this approach is sensitive to the window size, leading to high variance (false positives) if the window is too small, or high bias (false negatives) if it is too large. Fixed-size windows are unable to adapt to changes in the data stream, making them inefficient in dynamic environments.

ADWIN, on the other hand, dynamically adjusts its window size. When the data is stable, the window grows to improve statistical stability; when concept drift occurs, the window shrinks to focus on recent data, quickly adapting to changes. This flexibility makes ADWIN more efficient than fixed-size windows in handling evolving data streams.

Gama's Drift Detection method monitors model error rates and detects concept drift when the error exceeds a threshold. While effective, it is less sensitive to gradual shifts and does not adjust the window

size. As a result, it may be slower to respond to gradual concept drift compared to ADWIN, which detects even subtle changes and adjusts the window size accordingly.

Empirical comparisons show that ADWIN outperforms both fixed-size windows and Gama's method in flexibility and accuracy. ADWIN adapts to varying rates of change, detecting both abrupt and gradual concept drift more effectively.

5 Implementation of the methodology in the *elecNormNew* dataset

Now, in this part, our goal is to evaluate the performance of different concept drift detection methods presented in the last sections on a real dataset, the *elecNormNew* one, that was also presented earlier in this report. Indeed, we have decided to use this specific dataset, which was also the one used in Bifet and Gavaldà (2007) [3], in order not to exactly reproduce their results but to see if we could try to add some perspective and also feed the analysis with some different new elements.

For instance, as will be detailed later on, in addition to implementing ADWIN2 and Gama's Drift Detection Method (DDM) as it was done in the reference article, we will also implement the ADWIN algorithm to detect the drifts. Indeed, our main goal is to assess if the optimized version of ADWIN (ie. ADWIN2) is indeed preferable as it seems to be the case from all the theoretical guarantees that were detailed previously. In addition, we will also consider a static approach that does not detect drift in order to see if the accuracy of our classifier (a Naive Bayes one) will improve by using a drift detection method.

5.1 Key Considerations Before Implementation

In our analysis, we want to evaluate the performance of a Naive Bayes classifier using a prequential testing approach. The goal is to assess the impact of incorporating concept drift detection methods on the classifier's ability to adapt to changing data distributions and improve predictive accuracy. In our case, accuracy will measure how well the classifier correctly predicts whether the electricity price will go *UP* or *DOWN*, on average in the next 24 hours, for each instance in the dataset.

To do so, we are going to implement four different scenarios:

1. A **Static Naive Bayes** approach consisting of using a standard Gaussian Naive Bayes classifier with no drift detection mechanisms considered.
2. **Naive Bayes with ADWIN:** ie the use of a Naive Bayes classifier combined with the ADWIN drift detection algorithm.
3. **Naive Bayes with ADWIN2:** ie the use of a Naive Bayes classifier combined with the ADWIN2 drift detection algorithm.
4. Finally the **Naive Bayes with DDM:** ie the use of a Naive Bayes classifier combined with the DDM drift detection algorithm.

Before going on with the implementation of the methods on the dataset, let us define or precise some of the concepts we consider in our analysis:

5.1.1 Naives Bayes classifier

The **Naive Bayes classifier** is a supervised machine learning algorithm used for classification tasks. It is based on probability theory and belongs to the family of generative learning algorithms, which focus on modelling the distribution of input data for each class.

Naive Bayes classifiers are based on two main assumptions: **conditional independence**, where each feature is assumed to be independent of the others, and**equal contribution**, where all features are assumed to contribute equally to the outcome.

The variant used in our analysis, **Gaussian Naive Bayes**, is specifically designed for datasets with continuous variables. It assumes that the features follow a Gaussian (normal) distribution, and the model is trained by estimating the mean and standard deviation of each feature for each class.

5.1.2 Prequential Testing Approach

The prequential testing approach is a method used to assess machine learning models in dynamic data stream environments. It simulates the process of continuously receiving data and adjusting the model accordingly.

The approach involves two key phases for each data instance:

- **Testing Phase**: The model makes a prediction based on the current instance of data, simulating the model's real-time prediction capability.
- **Training Phase**: After the prediction, the true label of the instance is revealed. The model is then updated by incorporating this new information, allowing it to learn and adapt to potential changes in the data over time (which is the case in our analysis thanks to the drift detection methods used).

This method is particularly effective for evaluating models that must operate in scenarios where data is constantly evolving. By continuously testing the model's ability to predict unseen instances and updating it as new data becomes available, prequential evaluation provides a realistic measure of a model's performance. It ensures that the model adapts over time while maintaining predictive accuracy.

5.1.3 ADWIN and ADWIN2 algorithms

In this implementation part, we want to evaluate the performance of ADWIN and ADWIN2 concept drift detection methods, methods that were explained in detail in the last sections of this report. However, before implementing them, we precise the importance of two parameters of these algorithms, δ, the confidence parameter and W, the window size considered.

Indeed, according to Bifet and Gavaldà (2007) [3], δ is the confidence level parameter used to control how sensitive the drift detection mechanism is to changes in the data. It represents the probability of making a false detection of a concept drift when there is not. If δ is too large, the algorithm may detect too many drifts (false positives), whereas if δ is too small, it may miss actual changes in the data (false negatives). Therefore, it is important to test different values to find one that balances the sensitivity and specificity of the drift detection.

Now, the window size parameter W represents the size of the window used to calculate the moving accuracy. This sliding window is employed to smooth the accuracy results over time. Specifically, at each time point, the accuracy is calculated based on the last W instances. The window size in algorithms like ADWIN adapts dynamically based on the detected changes in the data. It can grow or shrink, depending on the detection of concept drifts, and controls how much past data the algorithm uses for learning.

5.1.4 Drift Detection Method (DDM)

Compared to ADWIN and ADWIN2, DDM from Gama (2004) [5] uses a fixed threshold for detecting drift. While this method works well for simple scenarios, it can be too rigid when handling complex data streams, as is the case in our analysis. Hence, we expect it to be less efficient than ADWIN and ADWIN2. That is what we want to assess in practice in the next sections.

5.1.5 Performance criteria considered

To assess the performance of the detection methods, we use three different criteria.

First, we use an **accuracy metric** that reflects the proportion of correct predictions made by the Naive Bayes classifier over the evaluated instances. We chose it because it allows us to monitor how well the model is performing over time while **accounting for recent changes in the data.** Indeed, if there is a drift in the data, the accuracy will change.

We also use a **number of drift-detected** criteria. Its goal is to track how many times a drift has been detected by each one of the drift detection methods. It helps to monitor how often the model has to adapt or retrain itself in response to changes in the data.

Finally, for each scenario considered, we compute its **total running time**. It tracks how computationally expensive each method is.

More precisely, it measures the total time the drift detection and model updating process takes to run (including predictions, accuracy calculation, and drift detection). Due to ADWIN2's specific characteristics and optimized structure compared to ADWIN, we expect it to be more efficient in terms of running time.

5.2 Implementation and results

5.2.1 Baseline approach ($\delta = 0.002$ and $W = 1000$)

In this baseline modelisation, we have used the four scenarios with, for ADWIN and ADWIN2, a δ equal to 0.002 and a window size equal to $W = 1000$. Also, we followed the approach of Bifet and Gavaldà (2007) [3] who trained all of their models on the last 48 samples received (still using a prequential testing method).

We obtained the following results:

Table 1: Performance Metrics for our 4 models (with $\delta = 0.002$ and $W = 1000$)

Model	**Mean Accuracy (Window 1000)**	**Total Running Time (seconds)**	**Drift Detection**
Static NB	0.7261	30.51	-
with ADWIN	0.7442	68.69	159
with ADWIN2	0.7462	62.78	188
with DDM	0.7139	56.38	51

So, we have that, for the simplest model (Gaussian Naive Bayes) that does not detect concept drift a mean accuracy of 72.61%. This is a decent result considering the absence of drift detection. Its running time is the smallest, due to its simplicity. It is thus quite efficient and fast for the task at hand.

Now, when using the ADWIN algorithm, it detects 159 drifts, which indicates that the model is actively tracking changes in the data distribution.
The mean accuracy is a little higher than the static approach, suggesting that the performance has improved over time.
However, the running time of 68.69 seconds is considerably longer (more than twice) than the Gaussian Naive Bayes. It confirms that the drift detection mechanisms do come with a computational cost.

Now, for ADWIN2, it detects 188 drifts, which is more than the 159 detected by ADWIN. This confirms that ADWIN2 is optimized and is more sensitive to changes in the data distribution. Its accuracy is slightly higher and its running time slightly shorter than ADWIN's. All the theory of ADWIN2 being

an improved version of ADWIN is confirmed here, with improved drift detection and a relatively lower computational cost.

Finally, DDM detects only 51 drifts, which is significantly fewer than ADWIN and ADWIN2. The mean accuracy is also lower, even lower than the Gaussian Naive Bayes method, suggesting that the algorithm fails in detecting the drifts efficiently in this case. Indeed, it only compares two statistical windows with a less sophisticated approach that does not adaptively adjust the window size as is the case in ADWIN and ADWIN2.

We also have plotted the evolution of the accuracy throughout all the instances for our four models as follows:

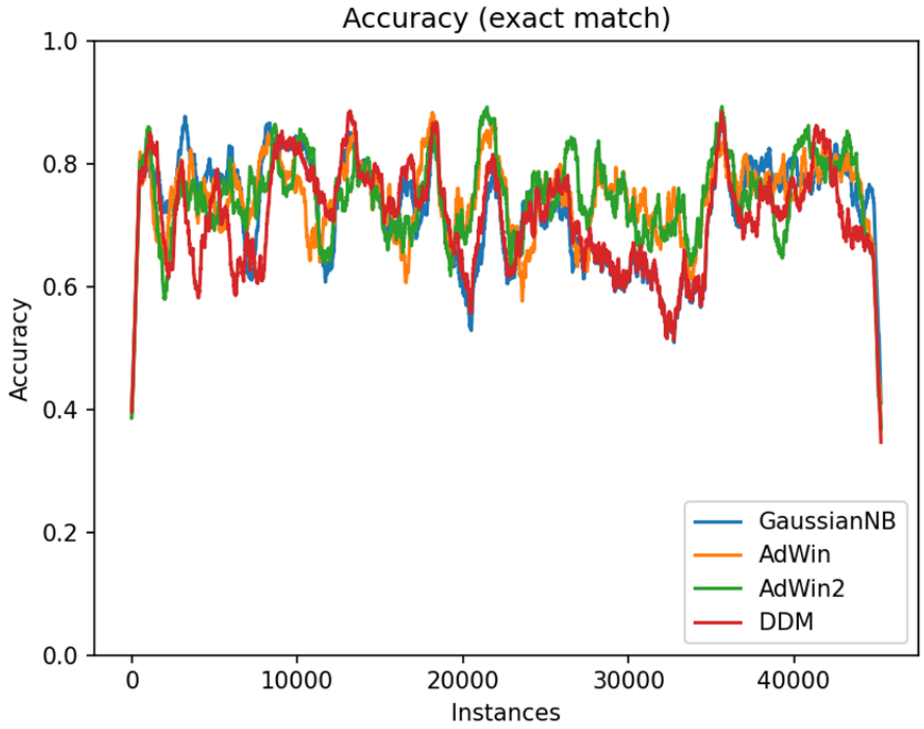

Figure 11: Evolution of the accuracy throughout instances for four models ($\delta = 0.002, W = 1000$)

Here, it is quite hard to distinguish which one of the models, amongst ADWIN and ADWIN2, performs well in terms of accuracy but it is overall, for both models, within a visual interval of 0.60 to 0.90. However, we clearly spot that DDM is the worst accuracy-based scenario.

5.2.2 Change in the δ parameter

Now in this setting, we consider the same four models but with a larger δ value now equal to 0.1 for ADWIN and ADWIN2 δ parameters. The window parameter W is still equal to 1000.

Here are the results we get :

Table 3: Performance Metrics for our 4 models (with $\delta = 0.1$ and $W = 1000$)

Model	**Mean Accuracy (Window 1000)**	**Total Running Time (seconds)**	**Drift Detection**
Static NB	0.7261	23.47	-
with ADWIN	0.7086	52.12	348
with ADWIN2	0.6889	69.42	451
with DDM	0.7139	66.87	51

The accuracy of the Static Gaussian Naive Bayes classifier remains consistent across both modelisations (72.61%), showcasing its stability. ADWIN and ADWIN2 experience notable drops in accuracy (70.86% and 68.89%, respectively), with the accuracy of ADWIN2 being even smaller than the one of ADWIN and of the Static Naive Bayes one. It suggests that increased sensitivity to drift in the new configuration might lead to over-adaptation.

Indeed, a larger δ in drift detection algorithms often correlates with more false positives detection. It is the case here with the number of drifts detected for ADWIN (equal to 348) and for ADWIN2 (equal to 451) being a lot bigger than in the previous setting. But here the higher detection does not translate into better accuracy.

For DDM, it detects the same number of drifts as before and maintains a similar accuracy (71.39%), demonstrating stable performance even with fewer drift detections.

We obtain the following accuracy measures plot :

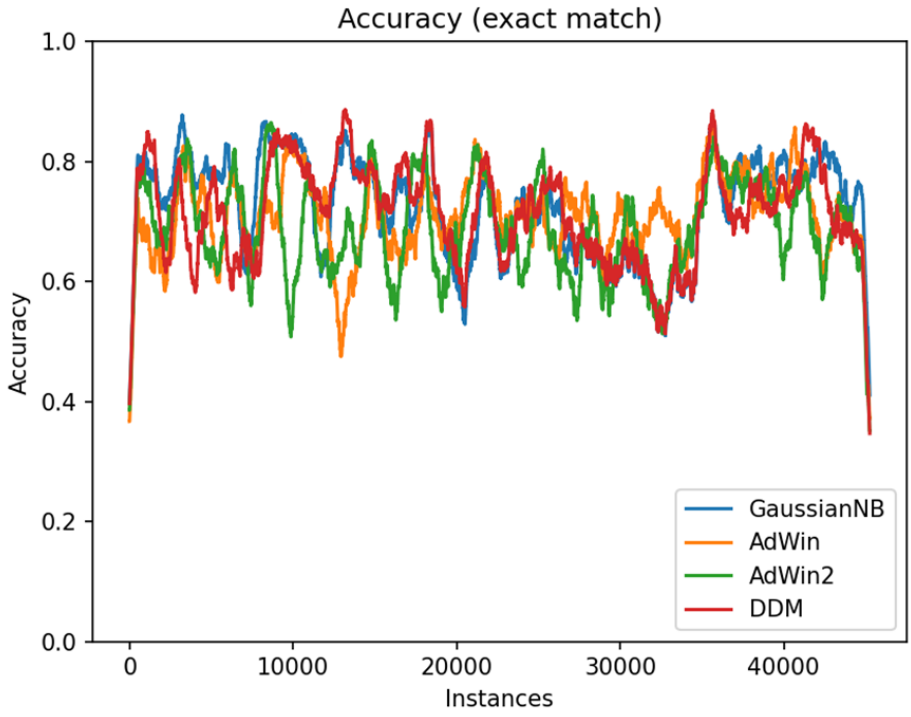

Figure 12: Evolution of the accuracy throughout instances for four models (delta = 0.1, W = 1000)

Here, compared to the previous setting, we cannot detect visually that classifiers using ADWIN and

ADWIN2 algorithms have a better accuracy overall. This confirms the results we had (lower mean accuracy than for Static Naive Bayes and DDM).

5.2.3 Change in the W parameter

Finally, in this last section we consider a setting with a δ parameter equal to 0.002 as in the first setting but this time with a window size equal to 100.

We obtain the results below :

Table 5: Performance Metrics for our 4 models (with $\delta = 0.002$ and $W = 100$

Model	Mean Accuracy (Window 100)	Total Running Time (seconds)	Drift Detection
Static NB	0.7301	24.16	-
with ADWIN	0.7480	45.40	159
with ADWIN2	0.7500	44.89	188
with DDM	0.7176	46.27	51

Compared to the first setting, we have, for all models, slightly better accuracies with shorter running times and the same number of drifts detected. ADWIN2 is the one performing best in terms of accuracy.

So, on paper, these results are ideal and seem to be the best case scenario. However, when we plot the accuracy over the instances as we did before, we get :

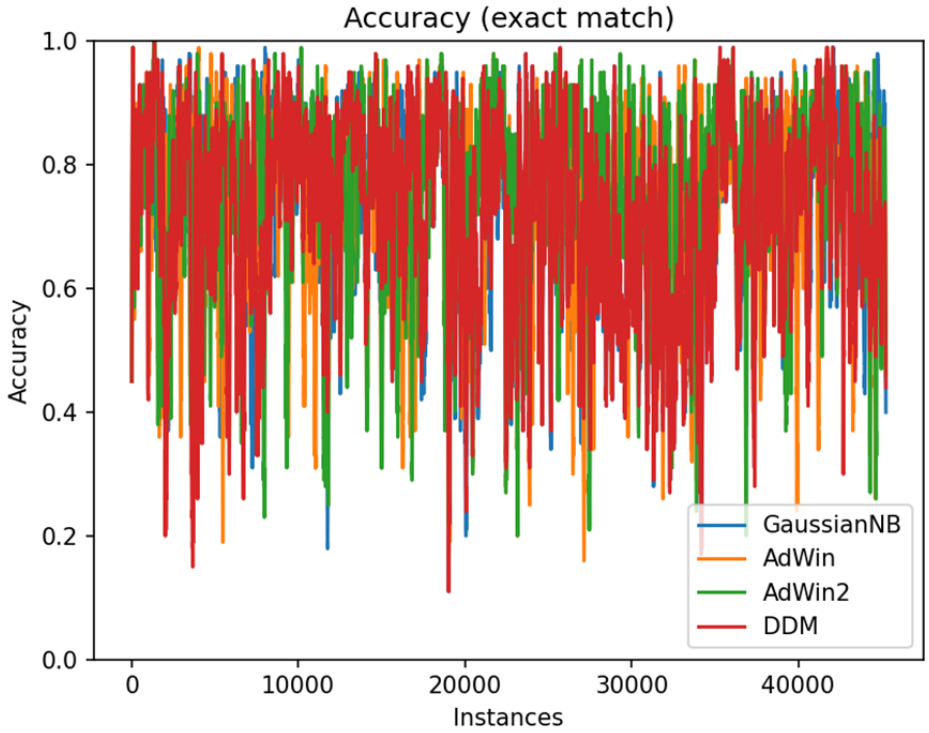

Figure 13: Evolution of the accuracy throughout instances for four models (delta = 0.002, W = 100)

The accuracy curves are showing higher variability across instances. This reflects the influence of smaller window sizes, where the models react more rapidly to changes in data but are also more sensitive to noise. Models such as ADWIN2 and ADWIN still perform better on average, but the increased noise makes it harder to observe on this plot.

All in all, a smaller window size allows for faster adaptation to changes but are also more prone to overreacting to noise or temporary fluctuations in the data while larger window size leads to more stable performance curves.

6 Conclusion

To conclude, we have conducted an in-depth exploration of the **concept drift phenomenon** and examined various **methods to address it**.
Among the existing drift detection techniques, we focused on the development and theoretical foundation of two closely related algorithms: **ADWIN and its optimized version, ADWIN2**, as detailed in Bifet and Gavalda (2007). These algorithms were selected for their ability to dynamically adapt to changes in data distributions over time, making them highly suitable for environments where data evolves continuously.

Through our implementation on the *elecNorm-New* dataset, we tested these algorithms alongside others, such as Drift Detection Method (**DDM**) explored by Gama (2004), using a **Gaussian Naive Bayes classifier** to assess their performance in detecting drift and improving prediction accuracy.

ADWIN2 achieved the highest accuracy under most settings, confirming its theoretical advantages. The static Naive Bayes classifier was less effective showcasing the importance of using drift-detection methods for accuracy improvement. DDM, due to its simplistic nature, performed the poorest in accuracy in most configurations.
In our baseline setting, ADWIN2 demonstrated slightly reduced runtime compared to ADWIN while maintaining superior drift detection sensitivity. It has also identified the most drifts.

One limitation of our implementation could be the use of the Naive Bayes classifier. Indeed, it relies on an assumption of feature independence, which often does not hold in real-world datasets. Additionally, Naive Bayes lacks interpretability in terms of feature importance, which can limit its usefulness.
One potential improvement could thus be to explore alternative classifiers like logistic regression, decision trees, or random forests that could provide better interpretability and performance.

Another limitation is the sensitivity of the results to hyperparameters, particularly the choice of δ and the window size (W) for the drift detection algorithms (ADWIN and ADWIN2). Without hyperparameter tuning, the performance of the models might not be fully optimized, leading to less reliable results. Hence, implementing grid search or other tuning methods for δ and W could help identify optimal configurations.

A Appendix

A.1 Python code for the exploratory analysis

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta

# Define column names
column_names = ["date", "day", "period", "nswprice", "nswdemand", "vicprice",
"vicdemand", "transfer", "class"]

# Read the CSV file and assign column names
df = pd.read_csv('concept-drift-master\data\elecNormNew.csv', header=None,
names=column_names)

# Display the first few rows
print(df)

# Define Start date (randomly)
start_date = datetime(2022, 1, 1)

# Convert 'period' (fraction of 24 hours) to readable time format (hh:mm)
df['total_hours'] = df['period'] * 24
df['hours'] = df['total_hours'].astype(int)
df['minutes'] = ((df['total_hours'] - df['hours']) * 60).astype(int)

# Calculate actual dates based on the fraction of the year
df['date'] = df['date'].apply(lambda x: start_date + timedelta(days=x * 365))

df['month'] = df['date'].dt.to_period('M')
df['week'] = df['date'].dt.to_period('W')

# Convert 'day' (integer) to weekday names
day_mapping = {1: "Monday", 2: "Tuesday", 3: "Wednesday", 4: "Thursday",
5: "Friday", 6: "Saturday", 7: "Sunday"}
df['day'] = df['day'].map(day_mapping)

# Display the transformed DataFrame
print(df)

# Heatmap of Correlations
correlation_matrix = df[['nswprice', 'nswdemand', 'vicprice', 'vicdemand',
'transfer']].corr()
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Heatmap')
plt.tight_layout()
plt.show()
```

```
# Class distribution
class_counts = df['class'].value_counts()
plt.figure(figsize=(8, 5))
class_counts.plot(kind='bar', color='blue')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.title('Distribution of Classes')

# Set x-axis labels horizontally
plt.xticks(rotation=0)

plt.tight_layout()
plt.show()

# Calculate the average by hour
df_avg = df.groupby('hours')['nswprice'].mean().reset_index()  # Calculate the
average by hour

# Ensure the 'hours' column is in a readable format for the X-axis
df_avg['hours'] = df_avg['hours'].astype(str)

# Create a plot of the average NSW price trend by hour
plt.figure(figsize=(10, 6))
plt.plot(df_avg['hours'], df_avg['nswprice'], label='Average NSW Price',
color='blue', marker='o')

# Add labels and a title
plt.xlabel('Hour (Time of Day)')
plt.title('Trend of Average NSW Price by Hour')

# Format the X-axis for better display of periods
plt.xticks(rotation=45)

# Add a legend
plt.legend()

# Use a compact layout to avoid overlap
plt.tight_layout()

# Display the plot
plt.show()

# Calculate the average by month
df_avg = df.groupby('month')['nswprice'].mean().reset_index()

# Ensure the 'month' column is in a readable format for the X-axis
df_avg['month'] = df_avg['month'].astype(str)

# Create a plot of the average NSW price trend by month
```

```
plt.figure(figsize=(10, 6))
plt.plot(df_avg['month'], df_avg['nswprice'], label='Average NSW Price',
color='blue', marker='o')

# Add labels and a title
plt.xlabel('Month')
plt.title('Trend of Average NSW Price by Month')

# Format the X-axis for better display of periods
plt.xticks(rotation=45)

# Add a legend
plt.legend()

# Use a compact layout to avoid overlap
plt.tight_layout()

# Display the plot
plt.show()

# Calculate the average by day
df_avg = df.groupby('day')['nswprice'].mean().reset_index()

# Ensure the 'day' column is an ordered category
day_order = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
"Sunday"]
df_avg['day'] = pd.Categorical(df_avg['day'], categories=day_order, ordered=True)

# Sort the data by day
df_avg = df_avg.sort_values('day')

# Create a plot of the average NSW price trend by day
plt.figure(figsize=(10, 6))
plt.plot(df_avg['day'], df_avg['nswprice'], label='Average NSW Price',
color='blue', marker='o')

# Add labels and a title
plt.xlabel('Day of the Week')
plt.title('Trend of Average NSW Price by Day of the Week')

# Format the X-axis for better display of days
plt.xticks(rotation=45)

# Add a legend
plt.legend()

# Use a compact layout to avoid overlap
plt.tight_layout()

# Display the plot
```

```
plt.show()
```

A.2 Python code for the concept drift detection plot

```
import matplotlib.pyplot as plt
from river.drift import ADWIN
import numpy as np

# Initialize the ADWIN drift detector
adwin = ADWIN()

# List to store drift points (timestamps where drift is detected)
drift_points = []
drift_dates = []

# Example of looping through a data stream (use your own data here)
for index, row in df.iterrows():  # assuming 'df' is your DataFrame
    value = row['nswprice']

    try:
        # Update the drift detector with the new data point
        adwin.update(value)

        # Check if drift is detected
        if adwin._drift_detected:  # Accessing the internal drift detection flag
            drift_points.append(value)  # Store the value of drift point
            drift_dates.append(row['date'])  # Store the date of drift point
            adwin._reset()  # Reset ADWIN after detecting drift
    except Exception as e:
        print(f"Error processing row {index}: {e}")

# Plotting the time series and drift points
plt.figure(figsize=(12, 6))
plt.plot(df['date'], df['nswprice'], label='NSW Price', color='blue', alpha=0.6)
plt.scatter(drift_dates, drift_points, color='red', label='Drift Detected', zorder=5)

# Add vertical lines for each drift point
for drift_date in drift_dates:
    plt.axvline(x=drift_date, color='red', linestyle='--', alpha=0.6)

plt.title("Concept Drift Detection in NSW Price")
plt.xlabel("Date")

plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()

# Show the plot
```

```
plt.show()
```

A.3 Python code for the method implementation

A.3.1 detector-classifier.py

```
"""Classifier that replaces the current classifier with a new one when a change is
detected in accuracy.
"""

# Authors: blablahaha
#          Alexey Egorov
#          Yuqing Wei
# Github link : https://github.com/blablahaha/concept-drift

from sklearn import clone
from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score


class DetectorClassifier(BaseEstimator):
    def __init__(self, clf, detection_method, classes):
        if not hasattr(clf, "partial_fit"):
            raise TypeError("Choose incremental classifier")
        self.clf = clf
        self.detection_method = detection_method
        self.classes = classes
        self.change_detected = 0

    def fit(self, X, y):
        self.clf.fit(X, y)
        return self

    def partial_fit(self, X, y):
        pre_y = self.clf.predict(X)
        if self.detection_method.set_input(accuracy_score(pre_y, y)):
            self.change_detected += 1
            self.clf = clone(self.clf)
            self.clf.partial_fit(X, y, classes=self.classes)
        else:
            self.clf.partial_fit(X, y)

    def predict(self, X):
        return self.clf.predict(X)
```

A.3.2 prequential.py

```
    """
Implementation of the prequential test
"""
```

```
# Authors: blablahaha
#          Alexey Egorov
#          Yuqing Wei
# Github link : https://github.com/blablahaha/concept-drift


from time import perf_counter
import numpy as np

def prequential(X, y, clf, n_train=1):
    """
    Prequential Evaluation: instances are first used to test, and then to train.
    :return: the label predictions for each test instance, and the associated
    running time.
    """
    row_num = y.shape[0]
    # Split an initial batch
    X_init = X[0:n_train]
    y_init = y[0:n_train]

    # Used for training and evaluation
    X_train = X[n_train:]
    y_train = y[n_train:]

    y_pre = np.zeros(row_num - n_train)
    time = np.zeros(row_num - n_train)

    clf.fit(X_init, y_init)

    for i in range(0, row_num - n_train):
        start_time = perf_counter()
        y_pre[i] = clf.predict(X_train[i, :].reshape(1, -1))
        clf.partial_fit(X_train[i, :].reshape(1, -1), y_train[i].reshape(1, -1).
        ravel())
        time[i] = perf_counter() - start_time

    return y_pre, time
```

A.3.3 adwin-bucket-row.py

```
    """
Within one bucket row, each bucket contains same number of original data.

New buckets are added at the end of bucket row.
When old buckets need to be removed, they are taken from the head of this bucket row.
"""
```

```
# For ADWIN parts
# Authors: blablahaha
#          Alexey Egorov
#          Yuqing Wei
# Github link : https://github.com/blablahaha/concept-drift

import numpy as np


class AdwinBucketRow:
    def __init__(self, max_buckets=5, next_bucket_row=None, previous_bucket_row=None):
        """
        :param max_buckets: Max bucket with one row
        :param next_bucket_row: Following bucket row
        :param previous_bucket_row: Previous bucket row
        """
        self.max_buckets = max_buckets

        # Current count of buckets in this bucket row
        self.bucket_count = 0

        self.next_bucket_row = next_bucket_row
        # Set previous bucket row connect to this bucket row
        if next_bucket_row is not None:
            next_bucket_row.previous_bucket_row = self

        self.previous_bucket_row = previous_bucket_row
        # Set next bucket connect to this bucket
        if previous_bucket_row is not None:
            previous_bucket_row.next_bucket_row = self

        # Init statistic number for each bucket
        self.bucket_sum = np.zeros(self.max_buckets + 1)
        self.bucket_variance = np.zeros(self.max_buckets + 1)

    def insert_bucket(self, value, variance):
        """
        Insert a new bucket at the end.
        """
        self.bucket_sum[self.bucket_count] = value
        self.bucket_variance[self.bucket_count] = variance
        self.bucket_count += 1

    def compress_bucket(self, number_deleted):
        """
        Remove the oldest buckets.
        """
        delete_index = self.max_buckets - number_deleted + 1
        self.bucket_sum[:delete_index] = self.bucket_sum[number_deleted:]
        self.bucket_sum[delete_index:] = np.zeros(number_deleted)
```

```
        self.bucket_variance[:delete_index] = self.bucket_variance[number_deleted:]
        self.bucket_variance[delete_index:] = np.zeros(number_deleted)

        self.bucket_count -= number_deleted

"""
Optimized bucket row for ADWIN 2.

Main changes :
- Optimization of the management of bucket rows
- simplification of the methods of adding (insert_bucket) and compressing
(compress_bucket)
- just affecting the infrastructure of buckets
"""

class Adwin2BucketRow:
    def __init__(self, max_buckets=5, next_bucket_row=None, previous_bucket_row=None):
        """
        :param max_buckets: Max bucket with one row
        :param next_bucket_row: Following bucket row
        :param previous_bucket_row: Previous bucket row
        """
        self.max_buckets = max_buckets

        # Current count of buckets in this bucket row
        self.bucket_count = 0

        self.next_bucket_row = next_bucket_row
        if next_bucket_row is not None:
            next_bucket_row.previous_bucket_row = self

        self.previous_bucket_row = previous_bucket_row
        if previous_bucket_row is not None:
            previous_bucket_row.next_bucket_row = self

        # Initialize bucket statistics (sum and variance)
        self.bucket_sum = np.zeros(self.max_buckets)
        self.bucket_variance = np.zeros(self.max_buckets)

    def insert_bucket(self, value, variance):
        """
        Insert a new bucket at the end.
        """
        if self.bucket_count < self.max_buckets:
            self.bucket_sum[self.bucket_count] = value
            self.bucket_variance[self.bucket_count] = variance
            self.bucket_count += 1
        else:
            raise ValueError("Cannot insert bucket: maximum bucket count reached.")
```

```
    def compress_bucket(self, number_deleted):
        """
        Remove the oldest buckets.
        """
        if number_deleted >= self.bucket_count:
            self.bucket_sum.fill(0)
            self.bucket_variance.fill(0)
            self.bucket_count = 0
        else:
            self.bucket_sum[:-number_deleted] = self.bucket_sum[number_deleted:]
            self.bucket_sum[-number_deleted:] = 0

            self.bucket_variance[:-number_deleted] =

            self.bucket_variance[number_deleted:]
            self.bucket_variance[-number_deleted:] = 0

            self.bucket_count -= number_deleted
```

A.3.4 adwin-buckets.py

```
    """
List of bucket rows.

Add new bucket row at head of window, remove old bucket row from tail of window.
"""

# For ADWIN parts
# Authors: blablahaha
#          Alexey Egorov
#          Yuqing Wei
# Github link : https://github.com/blablahaha/concept-drift

from concept_drift.adwin_bucket_row import AdwinBucketRow, Adwin2BucketRow

class AdwinRowBucketList:
    def __init__(self, max_buckets=5):
        """
        :param max_buckets: Max number of bucket in each bucket row
        """
        self.max_buckets = max_buckets

        self.count = 0
        self.head = None
        self.tail = None
        self.__add_to_head()

    def __add_to_head(self):
```

```
        """
        Init bucket row list.
        """
        self.head = AdwinBucketRow(self.max_buckets, next_bucket_row=self.head)
        if self.tail is None:
            self.tail = self.head
        self.count += 1

    def add_to_tail(self):
        """
        Add the bucket row at the end of the window.
        """
        self.tail = AdwinBucketRow(self.max_buckets, previous_bucket_row=self.tail)
        if self.head is None:
            self.head = self.tail
        self.count += 1

    def remove_from_tail(self):
        """
        Remove the last bucket row in the window.
        """
        self.tail = self.tail.previous_bucket_row
        if self.tail is None:
            self.head = None
        else:
            self.tail.next_bucket_row = None
        self.count -= 1

"""
Optimized bucket list for ADWIN 2.
Main changes :
- improving dynamic row management with the use of double-cahined lists
- allowing more efficient insertion and deletion
- to facilitate the merging of buckets and improve memory management
"""

class Adwin2RowBucketList:
    def __init__(self, max_buckets=5):
        """
        :param max_buckets: Max number of buckets in each bucket row
        """
        self.max_buckets = max_buckets

        self.count = 0
        self.head = None
        self.tail = None
        self.__add_to_head()

    def __add_to_head(self):
        """
```

```
        Add a new bucket row to the head of the list.
        """
        self.head = Adwin2BucketRow(self.max_buckets, next_bucket_row=self.head)
        if self.tail is None:
            self.tail = self.head
        self.count += 1

    def add_to_tail(self):
        """
        Add a new bucket row to the end of the list.
        """
        self.tail = Adwin2BucketRow(self.max_buckets, previous_bucket_row=self.tail)
        if self.head is None:
            self.head = self.tail
        self.count += 1

    def remove_from_tail(self):
        """
        Remove the last bucket row in the list.
        """
        if self.tail is not None:
            self.tail = self.tail.previous_bucket_row
            if self.tail is None:
                self.head = None
            else:
                self.tail.next_bucket_row = None
            self.count -= 1
```

A.3.5 adwin.py

```
    """
Implementation for paper:

Learning from Time-Changing Data with Adaptive Windowing
"""

# For ADWIN parts
# Authors: blablahaha
#          Alexey Egorov
#          Yuqing Wei
# Github link : https://github.com/blablahaha/concept-drift

from math import log, sqrt, fabs

from concept_drift.adwin_buckets import AdwinRowBucketList, Adwin2RowBucketList


class AdWin:
    def __init__(self, delta=0.002, max_buckets=5, min_clock=32, min_win_len=10,
```

```
min_sub_win_len=5):
    """
    :param delta: Confidence value
    :param max_buckets: Max number of buckets within one bucket row
    :param min_clock: Min number of new data for starting to reduce window
    and detect change
    :param min_win_len: Min window length for starting to reduce window
    and detect change
    :param min_sub_win_len: Min sub-window length, which is split from whole window
    """
    self.delta = delta
    self.max_buckets = max_buckets
    self.min_clock = min_clock
    self.min_win_len = min_win_len
    self.min_sub_win_len = min_sub_win_len

    # Time is used for comparison with min_clock parameter
    self.time = 0
    # Current window length
    self.window_len = 0
    # Sum of all values in the window
    self.window_sum = 0.0
    # Variance of all values in the window
    self.window_variance = 0.0

    # Count of bucket row within window
    self.bucket_row_count = 0
    # Init bucket list
    self.bucket_row_list = AdwinRowBucketList(self.max_buckets)

def set_input(self, value):
    """
    Main method for adding a new data value and automatically detect a possible
    concept drift.

    :param value: new data value
    :return: true if there is a concept drift, otherwise false
    """
    self.time += 1

    # Insert the new element
    self.__insert_element(value)

    # Reduce window
    return self.__reduce_window()

def __insert_element(self, value):
    """
    Create a new bucket, and insert it into bucket row which is the head of bucket
    row list.
```

```
    Meanwhile, this bucket row maybe compressed if reaches the maximum number of
    buckets.

    :param value: New data value from the stream
    """
    # Insert the new bucket
    self.bucket_row_list.head.insert_bucket(value, 0)

    # Calculate the incremental variance
    incremental_variance = 0
    if self.window_len > 0:
        mean = self.window_sum / self.window_len
        incremental_variance = self.window_len * pow(value - mean, 2) /
        (self.window_len + 1)

    # Update statistic value
    self.window_len += 1
    self.window_variance += incremental_variance
    self.window_sum += value

    # Compress buckets if necessary
    self.__compress_bucket_row()

def __compress_bucket_row(self):
    """
    If reaches maximum number of buckets, then merge two buckets within one row into
    a next bucket row.
    """
    bucket_row = self.bucket_row_list.head
    bucket_row_level = 0
    while bucket_row is not None:
        # Merge buckets if row is full
        if bucket_row.bucket_count == self.max_buckets + 1:
            next_bucket_row = bucket_row.next_bucket_row
            if next_bucket_row is None:
                # Add new bucket row and move to it
                self.bucket_row_list.add_to_tail()
                next_bucket_row = bucket_row.next_bucket_row
                self.bucket_row_count += 1

            # Calculate number of bucket which will be compressed into next
            bucket row
            n_1 = pow(2, bucket_row_level)
            n_2 = pow(2, bucket_row_level)
            # Mean
            mean_1 = bucket_row.bucket_sum[0] / n_1
            mean_2 = bucket_row.bucket_sum[1] / n_2
            # Total
            next_total = bucket_row.bucket_sum[0] + bucket_row.bucket_sum[1]
            # Variance
```

```
            external_variance = n_1 * n_2 * pow(mean_1 - mean_2, 2) / (n_1 + n_2)
            next_variance = bucket_row.bucket_variance[0] + bucket_row.bucket_variance[1] + e

            # Insert into next bucket row, meanwhile remove two original buckets
            next_bucket_row.insert_bucket(next_total, next_variance)

            # Compress those tow buckets
            bucket_row.compress_bucket(2)

            # Stop if the number of bucket within one row, which does not exceed
            limited
            if next_bucket_row.bucket_count <= self.max_buckets:
                break
        else:
            break

        # Move to next bucket row
        bucket_row = bucket_row.next_bucket_row
        bucket_row_level += 1

def __reduce_window(self):
    """
    Detect a change from last of each bucket row, reduce the window if there is a
    concept drift.

    :return: boolean: Whether has changed
    """
    is_changed = False
    if self.time % self.min_clock == 0 and self.window_len > self.min_win_len:
        is_reduced_width = True
        while is_reduced_width:
            is_reduced_width = False
            is_exit = False
            n_0, n_1 = 0, self.window_len
            sum_0, sum_1 = 0, self.window_sum

            # Start building sub windows from the tail of window (old bucket row)
            bucket_row = self.bucket_row_list.tail
            i = self.bucket_row_count
            while (not is_exit) and (bucket_row is not None):
                for bucket_num in range(bucket_row.bucket_count):
                    # Iteration of last bucket row, or last bucket in one row
                    if i == 0 and bucket_num == bucket_row.bucket_count - 1:
                        is_exit = True
                        break

                    # Grow sub window 0, while reduce sub window 1
                    n_0 += pow(2, i)
                    n_1 -= pow(2, i)
                    sum_0 += bucket_row.bucket_sum[bucket_num]
```

```
                    sum_1 -= bucket_row.bucket_sum[bucket_num]
                    diff_value = (sum_0 / n_0) - (sum_1 / n_1)

                    # Minimum sub window length is matching
                    if n_0 > self.min_sub_win_len + 1 and n_1 > self.min_sub_win_len
                    + 1:
                        # Remove oldest bucket if there is a concept drift
                        if self.__reduce_expression(n_0, n_1, diff_value):
                            is_reduced_width, is_changed = True, True
                            if self.window_len > 0:
                                n_0 -= self.__delete_element()
                                is_exit = True
                                break

                # Move to previous bucket row
                bucket_row = bucket_row.previous_bucket_row
                i -= 1
        return is_changed

    def __reduce_expression(self, n_0, n_1, diff_value):
        """
        Calculate epsilon cut value.

        :param n_0: number of elements in sub window 0
        :param n_1: number of elements in sub window 1
        :param diff_value: difference of mean values of both sub windows
        :return: true if difference of mean values is higher than epsilon_cut
        """
        # Harmonic mean of n0 and n1
        m = 1 / (n_0 - self.min_sub_win_len + 1) + 1 / (n_1 - self.min_sub_win_len + 1)
        d = log(2 * log(self.window_len) / self.delta)
        variance_window = self.window_variance / self.window_len
        epsilon_cut = sqrt(2 * m * variance_window * d) + 2 / 3 * m * d
        return fabs(diff_value) > epsilon_cut

    def __delete_element(self):
        """
        Remove a bucket from tail of bucket row.

        :return: Number of elements to be deleted
        """
        bucket_row = self.bucket_row_list.tail
        deleted_number = pow(2, self.bucket_row_count)
        self.window_len -= deleted_number
        self.window_sum -= bucket_row.bucket_sum[0]

        deleted_bucket_mean = bucket_row.bucket_sum[0] / deleted_number
        inc_variance = bucket_row.bucket_variance[0] + deleted_number * self.window_len *
        pow(
            deleted_bucket_mean - self.window_sum / self.window_len, 2
```

```
        ) / (deleted_number + self.window_len)

        self.window_variance -= inc_variance

        # Delete bucket from bucket row
        bucket_row.compress_bucket(1)
        # If bucket row is empty, remove it from the tail of window
        if bucket_row.bucket_count == 0:
            self.bucket_row_list.remove_from_tail()
            self.bucket_row_count -= 1

        return deleted_number


"""
ADWIN 2 Implementation: Optimized for better performance.
Main changes :

- now using the specific ADWIN2 bucket merge and compression strategies
- improving of the dynamic window management by removing obsolete data more efficiently
- reviews some maths computations used for drift detection (epsilon cut)
"""

class AdWin2:
    def __init__(self, delta=0.002, max_buckets=5, min_clock=32, min_win_len=10,
    min_sub_win_len=5):
        """
        :param delta: Confidence value
        :param max_buckets: Max number of buckets in one bucket row
        :param min_clock: Min number of new data for starting to reduce window
        :param min_win_len: Min window length to start checking for drift
        :param min_sub_win_len: Min sub-window length
        """
        self.delta = delta
        self.max_buckets = max_buckets
        self.min_clock = min_clock
        self.min_win_len = min_win_len
        self.min_sub_win_len = min_sub_win_len

        self.time = 0
        self.window_len = 0
        self.window_sum = 0.0
        self.window_variance = 0.0

        self.bucket_row_count = 0
        self.bucket_row_list = Adwin2RowBucketList(self.max_buckets)

    def set_input(self, value):
        """
        Add a new value and detect concept drift.
```

```
    """
    self.time += 1
    self.__insert_element(value)
    return self.__reduce_window()

def __insert_element(self, value):
    """
    Insert a new element into the window.
    """
    mean = self.window_sum / self.window_len if self.window_len > 0 else 0
    incremental_variance = self.window_len * pow(value - mean, 2) / (self.window_len
    + 1) if self.window_len > 0 else 0

    self.window_sum += value
    self.window_variance += incremental_variance
    self.window_len += 1

    self.bucket_row_list.head.insert_bucket(value, 0)
    self.__compress_bucket_row()

def __compress_bucket_row(self):
    """
    Compress buckets by merging them as needed.
    """
    bucket_row = self.bucket_row_list.head
    level = 0

    while bucket_row is not None:
        if bucket_row.bucket_count == self.max_buckets:
            if bucket_row.next_bucket_row is None:
                self.bucket_row_list.add_to_tail()
                self.bucket_row_count += 1

            n_1, n_2 = pow(2, level), pow(2, level)
            mean_1 = bucket_row.bucket_sum[0] / n_1
            mean_2 = bucket_row.bucket_sum[1] / n_2

            total_sum = bucket_row.bucket_sum[0] + bucket_row.bucket_sum[1]
            variance = bucket_row.bucket_variance[0] + bucket_row.bucket_variance[1]
            + (
                n_1 * n_2 * pow(mean_1 - mean_2, 2) / (n_1 + n_2)
            )

            bucket_row.next_bucket_row.insert_bucket(total_sum, variance)
            bucket_row.compress_bucket(2)

        bucket_row = bucket_row.next_bucket_row
        level += 1

def __reduce_window(self):
```

```
    """
    Reduce the window size by detecting concept drift.
    """
    if self.time % self.min_clock != 0 or self.window_len <= self.min_win_len:
        return False

    is_changed = False
    n_0, n_1 = 0, self.window_len
    sum_0, sum_1 = 0, self.window_sum

    bucket_row = self.bucket_row_list.tail
    level = self.bucket_row_count

    while bucket_row is not None:
        for i in range(bucket_row.bucket_count):
            n_0 += pow(2, level)
            n_1 -= pow(2, level)
            sum_0 += bucket_row.bucket_sum[i]
            sum_1 -= bucket_row.bucket_sum[i]

            if n_0 > self.min_sub_win_len and n_1 > self.min_sub_win_len:
                diff = fabs(sum_0 / n_0 - sum_1 / n_1)
                if self.__reduce_expression(n_0, n_1, diff):
                    self.__delete_element()
                    is_changed = True
                    break

        bucket_row = bucket_row.previous_bucket_row
        level -= 1

    return is_changed

def __reduce_expression(self, n_0, n_1, diff_value):
    """
    Calculate epsilon cut to detect drift.
    """
    m = 1 / (n_0 - self.min_sub_win_len + 1) + 1 / (n_1 - self.min_sub_win_len + 1)
    d = log(2 * log(self.window_len) / self.delta)
    variance = self.window_variance / self.window_len
    epsilon_cut = sqrt(2 * m * variance * d) + 2 / 3 * m * d
    return diff_value > epsilon_cut

def __delete_element(self):
    """
    Remove elements from the tail of the window.
    """
    bucket_row = self.bucket_row_list.tail
    deleted_count = pow(2, self.bucket_row_count)

    self.window_len -= deleted_count
```

```
        self.window_sum -= bucket_row.bucket_sum[0]

        deleted_mean = bucket_row.bucket_sum[0] / deleted_count
        variance_reduction = bucket_row.bucket_variance[0] + (
            deleted_count * self.window_len * pow(deleted_mean - self.window_sum
            / self.window_len, 2) / (deleted_count + self.window_len)
        )
        self.window_variance -= variance_reduction

        bucket_row.compress_bucket(1)
        if bucket_row.bucket_count == 0:
            self.bucket_row_list.remove_from_tail()
            self.bucket_row_count -= 1
```

A.3.6 DDM.py

```
    """ Drift detection method based in DDM method of Joao Gama SBIA 2004. """

# Authors: Wenjun Bai <vivianbai.cn@gmail.com>
#          Shu Shang <ignatius.sun@gmail.com>
#          Duyen Phuc Nguyen <nguyenduyenphuc@gmail.com>
# Github link : https://github.com/adolfoeliazat/drift-detection/blob/master/
drift_detector/adwin.py
# License: BSD 3 clause

import sys
import math


class DDM:
    """
    The drift detection method (DDM) controls the number of errors
    produced by the learning model during prediction. It compares
    the statistics of two windows: the first contains all the data,
    and the second contains only the data from the beginning until
    the number of errors increases.
    Their method doesn't store these windows in memory.
    It keeps only statistics and a window of recent errors data.".

    References
    ---------
    Gama, J., Medas, P., Castillo, G., Rodrigues, P.:
    "Learning with drift detection". In: Bazzan, A.L.C., Labidi,
    S. (eds.) SBIA 2004. LNCS (LNAI), vol. 3171, pp. 286{295. Springer, Heidelberg (2004)
    """

    def __init__(self):
        self.m_n = 1
        self.m_p = 1
```

```
    self.m_s = 0
    self.m_psmin = sys.float_info.max
    self.m_pmin = sys.float_info.max
    self.m_smin = sys.float_info.max
    self.change_detected = False
    self.is_initialized = True
    self.estimation = 0.0
    self.is_warning_zone = False

def set_input(self, prediction):
    """
    The number of errors in a sample of n examples is modelled by a binomial
    distribution.
    For each point t in the sequence that is being sampled, the error rate is
    the probability
    of mis-classifying p(t), with standard deviation s(t).
    DDM checks two conditions:
    1) p(t) + s(t) > p(min) + 2 * s(min) for the warning level
    2) p(t) + s(t) > p(min) + 3 * s(min) for the drift level

    Parameters
    ----------
    prediction : new element, it monitors the error rate

    Returns
    -------
    change_detected : boolean
                True if a change was detected.
    """
    if self.change_detected is True or self.is_initialized is False:
        self.reset()
        self.is_initialized = True

    self.m_p += (prediction - self.m_p) / float(self.m_n)
    self.m_s = math.sqrt(self.m_p * (1 - self.m_p) / float(self.m_n))

    self.m_n += 1
    self.estimation = self.m_p
    self.change_detected = False

    if self.m_n < 30:
        return False

    if self.m_p + self.m_s <= self.m_psmin:
        self.m_pmin = self.m_p;
        self.m_smin = self.m_s;
        self.m_psmin = self.m_p + self.m_s;

    if self.m_p + self.m_s > self.m_pmin + 3 * self.m_smin:
        self.change_detected = True
```

```
        elif self.m_p + self.m_s > self.m_pmin + 2 * self.m_smin:
            self.is_warning_zone = True
        else:
            self.is_warning_zone = False

        return self.change_detected

    def reset(self):
        """reset the DDM drift detector"""
        self.m_n = 1
        self.m_p = 1
        self.m_s = 0
        self.m_psmin = sys.float_info.max
        self.m_pmin = sys.float_info.max
        self.m_smin = sys.float_info.max
```

A.3.7 test.py

```
    """
This is a test script for the three drift detection algorithms (ADWIN, ADWIN2, DDM)
implemented in this project. The script takes the input dataset elecNormNew.csv.
The test is based on Prequential Evaluation for Naive Bayes estimators, and monitors 3
indicators of performance : accuracy, number of drifts detected and running time of
the algorithm.

"""

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import preprocessing
from sklearn.naive_bayes import GaussianNB
import time

from classifier.detector_classifier import DetectorClassifier
from concept_drift.adwin import AdWin, AdWin2
from concept_drift.DDM import DDM
from evaluation.prequential import prequential


def read_data(filename):
    df = pd.read_csv(filename)
    data = df.values
    return data[:, :-1], data[:, -1]


if __name__ == '__main__':
    n_train = 48
    X, y = read_data(r'C:\Users\emmab\Desktop\Data Mining\projet\A-Concept Drift (2)
    \A-Concept Drift\Project[1]\Project\concept-drift-master\data\elecNormNew.csv')
```

```
# Set x,y as numeric
X = X.astype(float)
label = ["UP", "DOWN"]
le = preprocessing.LabelEncoder()
le.fit(label)
y = le.transform(y)

w = 1000

clfs = [
    GaussianNB(),
    DetectorClassifier(GaussianNB(), AdWin(), np.unique(y)),
    DetectorClassifier(GaussianNB(), AdWin2(), np.unique(y)),
    DetectorClassifier(GaussianNB(), DDM(), np.unique(y))
]
clfs_label = ["GaussianNB", "AdWin", "AdWin2", "DDM"]

plt.title("Accuracy (exact match)")
plt.xlabel("Instances")
plt.ylabel("Accuracy")

# Iteration through each classifier
for i in range(len(clfs)):
    print("\n{}:".format(clfs_label[i]))

    start_time = time.time()

    # Run the prequential evaluation while mesuring time
    with np.errstate(divide='ignore', invalid='ignore'):
        y_pre, eval_time = prequential(X, y, clfs[i], n_train)

    end_time = time.time()

    #Total running time
    total_time = end_time - start_time
    print(f"Total running time for {clfs_label[i]}: {total_time:.2f} seconds")


    if clfs[i].__class__.__name__ == "DetectorClassifier":
        print("Drift detection: {}".format(clfs[i].change_detected))

    # Accuracy
    estimator = (y[n_train:] == y_pre) * 1
    acc_run = np.convolve(estimator, np.ones((w,)) / w, 'same')
    print("Mean acc within the window {}: {}".format(w, np.mean(acc_run)))

    # Plotting the accuracy
    plt.plot(acc_run, "-", label=clfs_label[i])

plt.legend(loc='lower right')
```

```
plt.ylim([0, 1])
plt.show()
```

References

[1] Adolfoeliazat. *GitHub - adolfoeliazat/drift-detection: Project of IoT data stream mining course: Implementation of Concept Drift Algorithm (Adwin, DDM, Stream Volatility)*. GitHub, 2017. URL: https://github.com/adolfoeliazat/drift-detection/tree/master (visited on 12/11/2024).

[2] Supriya Agrahari and Anil Kumar Singh. "Concept Drift Detection in Data Stream Mining : A literature review". In: *Journal of King Saud University - Computer and Information Sciences* (Dec. 2021). DOI: 10.1016/j.jksuci.2021.11.006.

[3] Albert Bifet and Ricard Gavaldà. "Learning from Time-Changing Data with Adaptive Windowing". In: *Proceedings of the 2007 SIAM International Conference on Data Mining* (Apr. 2007), pp. 443–448. DOI: 10.1137/1.9781611972771.42. (Visited on 03/27/2023).

[4] Blablahaha. *GitHub - blablahaha/concept-drift: Algorithms for detecting changes from a data stream.* GitHub, 2017. URL: https://github.com/blablahaha/concept-drift (visited on 12/11/2024).

[5] João Gama et al. "Learning with Drift Detection". In: *Advances in Artificial Intelligence – SBIA 2004* 3171 (2004), pp. 286–295. DOI: 10.1007/978-3-540-28645-5_29.

[6] Fabian Hinder et al. "Model-based explanations of concept drift". In: *Neurocomputing* 555 (Oct. 2023), p. 126640. DOI: 10.1016/j.neucom.2023.126640. URL: https://www.sciencedirect.com/science/article/pii/S0925231223007634 (visited on 01/11/2024).