Semi-Supervised Hybrid Windowing Ensembles for Learning from Evolving Streams

Sean Louis Alan FLOYD

Thesis submitted in partial fulfilment of the requirements for the

Master of Science in Computer Science

School of Electrical Engineering and Computer Science

Faculty of Engineering

University of Ottawa

Abstract

In this thesis, learning refers to the intelligent computational extraction of knowledge from data. Supervised learning tasks require data to be annotated with labels, whereas for unsupervised learning, data is not labelled. Semi-supervised learning deals with data sets that are partially labelled. A major issue with supervised and semi-supervised learning of data streams is late-arriving or missing class labels. Assuming that correctly labelled data will always be available and timely is often unfeasible, and, as such, supervised methods are not directly applicable in the real world. Therefore, real-world problems usually require the use of semi-supervised or unsupervised learning techniques. For instance, when considering a spam detection task, it is not reasonable to assume that all spam will be identified (correctly labelled) prior to learning. Additionally, in semi-supervised learning, "the instances having the highest [predictive] confidence are not necessarily the most useful ones" [41]. We investigate how self-training performs without its selective heuristic in a streaming setting.

This leads us to our contributions. We extend an existing concept drift detector to operate without any labelled data, by using a sliding window of our ensemble's prediction confidence, instead of a boolean indicating whether the ensemble's predictions are correct. We also extend selective self-training, a semi-supervised learning method, by using all predictions, and not only those with high predictive confidence. Finally, we introduce a novel windowing type for ensembles, as sliding windows are very time consuming and regular tumbling windows are not a suitable replacement. Our windowing technique can be considered a hybrid of the two: we train each sub-classifier in the ensemble with tumbling windows, but delay training in such a way that only one sub-classifier can update its model per iteration.

We found, through statistical significance tests, that our framework is (roughly 160 times) faster than current state of the art techniques, and achieves comparable predictive accuracy. That being said, more research is needed to further reduce the quantity of labelled data used for training, while also increasing its predictive accuracy.

Acknowledgements

The research conducted for this thesis has been financed by the Province of Ontario and the University of Ottawa.

I would particularly like to thank my supervisor, Dr. Herna L. Viktor, for her guidance, incredible patience, and, her support.

Finally, I could not have completed this thesis without the support, and encouragement of my friends and family.

Many thanks to each and every one of you.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ADWIN	Adaptive Windowing
ANOVA	Analysis of Variance
AI	Artificial Intelligence
Bagging	Bootstrap Aggregation
CPU	Central Processing Unit
CSV	Comma Separated Values
DDM	Drift Detection Method
FHDDMS	Fast Hoffding Drift Detection Method for evolving data Streams
GB	Gigabyte
IoT	Internet of Things
kNN	k Nearest Neighbours
LB	Leveraging Bagging
LED	Light Emitting Diode
LESS-TWE	Learning from Evolving Stream via Self-Training Windowing Ensembles
ML	Machine Learning
MFHDDMS	Modified Fast Hoffding Drift Detection Method for evolving data Streams
MOA	Massive Online Analysis
NB	Naive Bayes
NN	Neural Network
OS	Operating System
RAM	Random Access Memory

SEA	Streaming Ensemble Algorithm
SGD	Stochastic Gradient Descent
SSD	Solid State Drive
SSL	Semi-Supervised Learning
TN	True Negative
TP	True Positive
UCI	University of California, Irvine
VFDT	Very Fast Decision Tree
WEKA	Waikato Environment for Knowledge Analysis

Chapter 1

Introduction

Analytical models have been studied and developed by researchers, with increasing intensity over the last two decades, to intelligently and computationally extract knowledge from data [7]. The quantity and size of data sets have grown exponentially over that time, requiring new algorithms to respect new constraints. Data streams are a latest data format that researchers are developing techniques for, and, are characterised by velocious and continuous flows of data [49]. This format requires algorithms to update their models to adapt to potential changes to the underlying concepts that the stream represent over time. Techniques have been developed to explicitly detect these changes to help models adapt more quickly in order to stay relevant and accurate. Researchers have shown a continuing interest in the development of algorithms to extract knowledge from evolving streams [35, 37, 39, 48, 49, 86, 95]. Another current topic of interest is the development of ensembles, which are defined as an amalgamation of any number of analytical models. Ensembles are considered as one of the most promising research directions nowadays [43, 48, 67, 73, 79, 98].

An abundance of algorithms have been proposed in the literature to use ensembles to learn from evolving streams, that do so either online (read instance-by-instance) or in chunks. These algorithms typically learn in batches by training new classifiers on each incoming chunk, and either use some weighting scheme or replacement strategy that relies on the use of timely and correctly labelled data. Alternatively, algorithms can learn online by using sliding windows of a variable, or fixed, size to summarise the data and appropriately update their models from them.

Drift detection methods have also mainly relied on the use of labelled data by detecting changes in the accuracy of a classifier over time. On the other hand drift detectors that are unsupervised, mostly rely on statistical tests [24, 32, 85, 88].

1.1 Motivation

A major issue with supervised and semi-supervised learning of data streams is late-arriving or missing class labels. For example, a bank cannot know if a loan will default before several months have passed, if not years. Assuming that correctly labelled data will always be available and timely is unreasonable, and, as such, supervised methods are not usually applicable in the real world. Therefore, real-world problems usually require the use of semi-supervised or unsupervised learning techniques. To the best of our knowledge, no research has been conducted to develop semi-supervised techniques to learn from evolving streams without clustering unlabelled data, which is computationally expensive [49]. Furthermore, semi-supervised drift detecting algorithms for streams

are more applicable to real-world applications, but only one article has been published as of yet [40].

1.2 Thesis Objective

The purpose of this thesis is to contribute to narrow the gap in research as it pertains to semi-supervised learning of evolving streams, without clustering unlabelled data.

Therefore, we combined our contributions to introduce a framework, LESS-TWE (Learning from Evolving Streams via Self-Training Windowing Ensembles). Our framework employs self-training, a novel windowing technique exclusive to ensembles, a new weighted soft voting strategy, and an extension of Fast Hoeffding Drift Detection Method for evolving Streams (FHDDMS) to work without labelled data.

Research has not yet been conducted to study how selective self-training, a semi-supervised learning (SSL) algorithm, performs when removing the selective heuristic, thereby predicting labels for unlabelled data then training on it. Selective self-training learns from labelled data that was originally unlabelled. To do so, it annotates previously unlabelled data with labels for which the classifier has predicted with high confidence [103]. This presents one of the objectives for this thesis.

By introducing the novel windowing technique, we aim to investigate if savings relating to the execution time can be achieved while maintaining comparable predictive accuracy. Additionally, we can observe if delaying the training of some of the classifiers in the ensemble affects concept drift detection.

Finally, we introduce a weighted soft voting scheme for ensembles, in conjunction with our extension to FHDDMS to detect drifts without relying on labelled data.

1.3 Thesis Organisation

This thesis is organised as follows.

In chapter 2 we review the background work surrounding data stream mining, ensemble learners, and concept drift detection.

Following the review, we will present the contributions made by this thesis to the literature in chapter 3.

As stated above, this will cover improvements to an existing concept drift algorithm to reduce its dependency on ground truth, improvements to a simple voting classifier to also further reduce its dependency on ground truth and finally a novel windowing technique that combines sliding and tumbling techniques.

In chapter 4, we will present the experimental design for testing our contributions and present the results and discuss them in chapter 5.

Finally we conclude in chapter 6.

Chapter 2

Background Work

In this chapter, we will briefly cover machine learning algorithms, then go over published works as they pertain to data stream mining. This will consist of various algorithms, and the current research problems being addressed.

We expect the audience of this thesis to be comfortable in the field of computer science and to have briefly read about or been introduced to machine learning.

2.1 Summary

This chapter presents the fundamentals of machine learning and the various challenges of extracting knowledge from data streams. We introduce classification as a type of machine learning, for which the goal is to extract knowledge, computationally, from labelled data. Algorithms are developed to model the relationship between the data and the class labels. Semi-supervised learning is also a classification task but with the added constraint of learning from a data set that is not entirely labelled. In this thesis, the data we learn from come from data streams, which are voluminous, volatile and velocious. As such, constraints on

time and memory usage must be respected, and a mechanism is required to forget "old data" safely.

We cover specific classifiers: baseline classifiers are usually relatively simple and used as a baseline for comparing classifier performance. We also cover the algorithms that we employed in our framework, as well as the state of the art: Naive Bayes, Stochastic Gradient Descent, Hoeffding Trees and Leveraging Bagging.

We define concept drifts as an evolution in the probability distribution of classes and/or attributes. We describe the main types of concept drifts that can occur, regardless of how self-explanatory they are named: abrupt, gradual and recurring. We then review drift detection strategies presented in the literature, including FHDDM/S which we extend in this thesis. Our review shows a gap in research as it pertains to semi-supervised drift detection.

Next, we define ensembles as an amalgamation of a number of classifiers. As all classifiers must output a prediction, ensembles must as well; we present existing techniques to map these multiple outputs to a single one. Having defined ensembles, we review those that were proposed in the literature using two criteria: the processing method and whether or not they were designed to deal with evolving concepts. The processing method distinguishes if instances are analysed online (one-by-one) or in batches. Our review shows a gap in research as it pertains to semi-supervised learning from evolving streams.

2.2 Machine Learning Fundamentals

Within the vast domain of Artificial Intelligence (AI) lies the field of Machine Learning (ML) whose goal is to extract knowledge from data automatically.

Given a data set, a model is constructed using a machine learning algorithm to extract knowledge from the attributes in the data. Classification is, among others, a method of knowledge extraction called supervised learning. Each item belonging to a given data set must have a nominal class label and a series of pairs of attributes and values. The class label is almost always constrained to a predefined set. A machine learning algorithm is used to model the relationship between the class labels and the attribute/value pairs. Once a model is created, it can predict the class label of any unlabelled data instance (or labelled, as long as the model does not use its value while predicting).

Semi-supervised learning (SSL) differs from traditional supervised learning, by learning from a data set that is not fully labelled, in that not all samples in a data set are labelled.

Initially, classification tasks dealt with static, small, homogeneous data sets. Therefore models created were also static and did not need to be updated, or did not support it. Big data, a sub-field of machine learning, arose due to the need to analyse even more voluminous data sets. This required new knowledge extraction techniques to be developed to handle large swaths of data while specifically focusing on keeping computation time and memory usage as low as possible. More recently, with the advent of the Internet of Things (IoT), more and more data is being generated and streamed online from low-powered (both computationally and due to battery capacity) internet-connected devices. This has developed an interest in the research community to design machine learning algorithms that can learn from data streams, due to the new challenges brought about by this new data medium.

2.3 Data Stream Classification

2.3.1 Data Streams

Krempl et al., in their paper discussing open challenges in data stream mining, use three Vs to describe the main challenges that must be dealt with: volume, volatility and velocity [49]. The difficulty with mining data streams is that these three challenges are present simultaneously, while only a subset must typically be dealt with when mining static data sets.

These challenges can be broken down into the following requirements: limiting time taken to process and model incoming data; modelling over a single pass, in an incremental fashion to keep memory bounded and minimal; establishing a mechanism to safely forget "old data" over time, to adapt to changes in the underlying concepts to be modelled [35, 37, 39, 48, 49, 86, 95].

2.3.2 Baseline Classification

Majority Class

This is one of the simplest classifiers. It predicts the class label of a new instance as the currently most frequent class. The majority class algorithm is mostly used as a baseline to determine if another classifier performs better or worse than simply predicting the most frequent class. Due to its simplicity, it satisfies the time requirement stated above by running very quickly, and requires very little maintenance and memory: only an array of counters are needed for each class [10].

No-Change

This is another very simple classifier. It predicts the class of a new instance as the real class of the previously seen instance. This classifier requires even less maintenance than majority class, as it only requires a variable to keep track of the last seen class label [10].

2.3.3 Naive Bayes (NB)

This algorithm applies the Bayes' theorem with naive feature independence assumptions. Zhang, in his paper discussing the optimality of Naive Bayes, says that it "is one of the most efficient and effective inductive learning algorithms" [100]. He also proves, that although the feature independence assumption is usually violated in real-world applications, that it can perform optimally if the "dependences distribute evenly in classes, or if the dependences cancel each other out" [100].

There are various versions implemented depending on the distribution of the data to be modelled, such as Gaussian NB, Multinomial NB, and Bernouilli NB.

A downside to using NB in an online setting is its inability to learn new concepts unless they are explicitly added to its training set with their new class label.

2.3.4 Stochastic Gradient Descent (SGD) Classifier

Bottou, in his paper discussing tricks for stochastic gradient descent, explains that a loss function measures the cost of predicting $\hat{y}$ when the actual class was y, and that the objective is to seek a function, which is parameterized by a weight vector, that minimises that loss. SGD is a simplification of gradient descent, that

estimates the gradient of the empirical risk instead of computing it. The empirical risk measures the training set performance. The estimation of the gradient is based on a randomly selected example from the data at each iteration [14].

The particular version of the SGDClassifier selected, from scikit-learn [69], implements a simple yet very efficient method to learn a logistic regression model under convex loss functions. The advantages, as listed by the authors of this particular implementation, are its efficiency, as previously stated, and its simple implementation. They also list, as its disadvantages, the hyperparameters that the classifier requires and its sensitivity to feature scaling. However, they note that it can "easily scale to problems with more than 10^5 training examples and more than 10^5 features" [83]. The authors also cite Bottou and his SGD SVM [15][1] and other published works [91, 84] as inspiration for their implementation.

2.3.5 Hoeffding Trees / VFDT

The main issue with building decision trees in a streaming setting is reusing instances to determine the best splitting criterion for attributes, which is not possible due to the memory requirements for storing all of the data [23]. Domingos and Hulten suggest a Very Fast Decision Tree algorithm for streaming data (VFDT), called Hoeffding Tree in [23]. This tree waits for new instances to arrive instead of reusing data for computing splits. Most interestingly, they find that the resulting tree is almost identical to the one modelled by an offline batch learning algorithm, given enough data to learn from. The algorithm (1) is based on the Hoeffding Bound and chooses, as a confidence interval for the entropy at a node, the value:

[1]SVMSGD: https://leon.bottou.org/projects/sgd

$$\epsilon = \sqrt{\frac{R^2 \ln 1/\delta}{2n}} \tag{2.1}$$

where R is the range of the random variable, δ is one minus the desired probability of choosing the correct attribute at any node, and n is the number of examples collected at a node. Lastly in algorithm 1, G is the heuristic measure to choose test attributes, and could be information gain as seen in C4.5 or some other heuristic [10, 23]

Algorithm 1: Hoeffding Tree [10]

```
1 function HoeffdingTree(Stream, δ)
    Data: a stream of labelled examples, confidence parameter δ
2   let HT be a tree with a single leaf node
3   init counts n_ijk at root
4   for each example(x,y) in Stream do
5     HTGrow((x,y), HT, δ)
1 function HTGrow((x, y), HT, δ)
2   sort(x, y) to leaf l using HT
3   update counts n_ijk at leaf l
4   if examples seen so far at l are not all of the same class then
5     compute G for each attribute
6     if G(best attribute) - G(second best) > sqrt(R^2 ln 1/δ / 2n) then
7       split leaf on best attribute
8       for each branch do
9         start new leaf and initialize counts
```

2.3.6 Leveraging Bagging

Leveraging Bagging (LB), also called Leverage Bagging, is a classifier ensemble that improves upon the online Oza Bagging algorithm in terms of accuracy, at the cost of higher memory utilisation and an increased execution time [6]. In offline bagging, M models are trained, each "with a bootstrap sample of size N

created by drawing random samples with replacement from the original training set" [6]. Each of the model's training sets should contain the original training set K times.

In online bagging, each sample is assigned a weight according to $Poisson(1)$, instead of sampling with replacement. Bifet, in a study comparing online bagging to online boosting, found that the former was the best method in terms of accuracy [12], but at the cost of a high memory utilisation and execution time, just like LB as we stated above. Bifet also claims that adding more random weights to all instances seems to improve accuracy more than if only adding to the misclassified instances. For this reason, they proposed their online leveraging bagging algorithm with randomisation improvements: increasing the weights of the input samples and adding randomisation to the output of the ensemble via output codes.

The improvements in Leveraging bagging [6] therefore come in the form of: first, increasing re-sampling by using a higher parameter for the Poisson distribution, to attribute a wider range of weights to the incoming samples (training samples), thereby increasing the input-space diversity; secondly, through the use of output detection codes: they assign an n bit binary code to each class label, and one of n classifiers are trained on one of those bits in order to correct, to a certain extent, misclassifications. Finally, to deal with concept drift, LB uses one instance of the ADWIN algorithm for each n classifier. When a drift is detected, the classifier associated with the ADWIN detector that has the highest variance in its sliding window is reset (calculated as the average of the numeric values kept in the window). Algorithm 2 shows the pseudocode for LB.

Algorithm 2: Leveraging Bagging for M models [6]

Initialise base models h_m for all $m \in \{1, 2, \ldots, M\}$
Compute colouring $\mu_m(y)$
for ***all*** *training example(x,y)* **do**
    for $m = 1, 2, \ldots, M$ **do**
        Set $w = Poisson(\lambda)$
        Update h_m with the current example with weight w and class $\mu_m(y)$
    if *ADWIN detects change in error of one of the classifiers* **then**
        Replace classifier with higher error with a new one
    anytime output: return hypothesis:
    $h_{fin}(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} I(h_t(x) = \mu_t(y))$

2.4 Evolving Concepts

In this section, we first define concept drifts and cover the different kinds of drifts that can occur. Strategies presented in the literature are reviewed thereafter.

2.4.1 Concept Drift Preliminaries

We consider two types of streams: those that are stationary and those that are not. The former consists of those where examples are generated randomly from a stationary but unknown probability distribution. The latter consists of streams where the distribution of classes and/or attributes can evolve over time. These evolutions in probability distributions are called concept drifts and occur after periods of stability [35, 48]. As concept drifts occur, a classifier's model learned from previous data becomes out of date with the current probability distribution and, as a result, its accuracy deteriorates over time. The classifier must then forget its model and start anew. Ergo, the primary challenge is to detect at which point in time concepts occur. Finally, to detect drifts, an algorithm must "combine robustness to noise with sensitivity to concept change" [35].

TABLE 2.1: How real drifts differ from virtual drifts, and noise

	Persistent	**Consistent**
Real drift	✓	✓
Virtual drift	×	✓
Noise	×	×

Real examples of domains with concept drifts include, among others, surveillance systems, telecommunication systems, sensor networks, categorising spam, financial fraud detection and weather predictions.

Gama formalises concept drift as a change in the joint probability $P(\vec{x}, y) = P(y|\vec{x}) \times P(\vec{x})$ [35], that is consistent and persistent.

A consistent concept is defined as a change in the state of the target function over a predetermined threshold between two distinct points in time, whereas a concept is qualified as persistent if measured as consistent over a given time frame.

This definition, based on cause and effect, allows us to distinguish two types of drift, as well as noise, which table 2.1 illustrates.

In practice, the learning model must be updated regardless of whether a drift is real or virtual.

The most common way to categorise concept drifts, however, is based upon the manner in which changes occur [10], as figure 2.1 depicts.

Sudden drifts are characterised by an abrupt change in the distribution generating the data, meaning that a given distribution is suddenly replaced by another distribution instantaneously [92].

Gradual drifts, as their name suggests, present "a slower rate of change". The distribution generating the data is slowly replaced over a longer period of time.

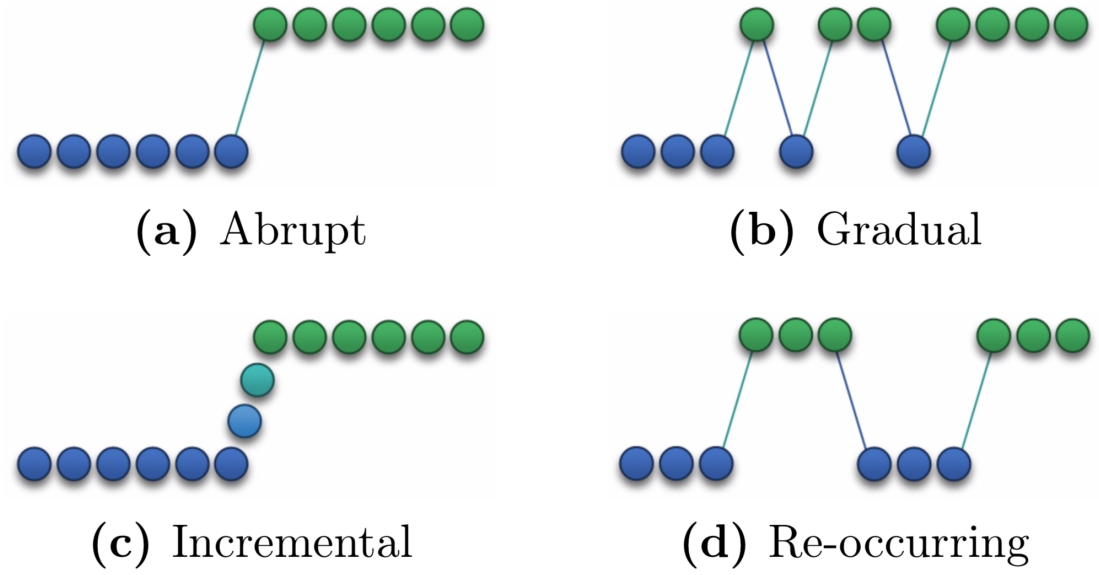

FIGURE 2.1: Types of drifts [70]

In [58], two types of gradual drifts are identified. The first, simply called gradual drifts, samples back and forth from two different distributions by slowly increasing its probability of sampling from the new distribution all the while decreasing from the one being replaced. The second transitions from two distributions by slowly morphing from one to the other, thereby slowly incorporating incremental changes. During this transition phase, data cannot be attributed to one distribution or another, but rather to a mix of both. These drifts are qualified as incremental.

The last commonly identified type of drift is the recurrent, or re-occurring, drift. Concepts can reappear after some length of time, and may or may not do so in a cyclical fashion meaning that concepts could reappear in a specific order [10, 48, 92].

2.4.2 Concept Drift Detection Methods

Methods to detect drifts do so based on information collected from a classifier's performance or directly from the incoming data. They warn that a drift may be occurring or that one has occurred, which is usually followed by updating, retraining or replacing an old classifier by a new one.

As we previously stated, fully supervised learning of data streams is not very suitable for real-world applications of data stream mining. It logically follows that drift detection methods that rely on a classifier's predictive performance, through the use of class labels, would also not be suitable for real-world applications.

3 metrics are usually considered to measure a drift detector's performance, according to [48]: the number of correct positive drift detections (real drifts detected), the number of false alarms (no drift present, but detected one erroneously), and the drift detection delay (time between a drift and its actual detection). As is usually the case, trade-offs are typically made between these three metrics. However, some aggregated measures are proposed in the literature. One such measure is the *acceptable delay length*, proposed by Pesaranghader and Viktor in [72], that determines if a drift is a true positive by measuring the distance between where a drift is detected and its actual location.

Gama categorises drift detection methods into 4 groups [34], which we will briefly cover in the upcoming paragraphs. These four groups are *Statistical Process Control* methods, *Sequential Analysis* methods, methods that *monitor distributions of 2 different time windows* and *contextual approaches*.

Drift Detection Method (DDM), proposed by Gama in [38], is the most well-known method for the first category. The main idea behind DDM is to monitor the number of errors a classifier makes when predicting. They use the assumption, which they back through statistical theory, that the error-rate should decrease if the distribution generating the incoming data does not change. Consequently, a change in distribution will increase the error rate. They set distinct thresholds for warnings and for drifts. If and when the warning level is reached, incoming instances are to be stored in a special window to be used to retrain the classifier if it ever reaches the drift level. If the drift level is not reached and the error rate drops below the warning level, the window is forgotten, and the drift warning is considered as a false alarm. DDM is not an optimal candidate to be extended for semi-supervised tasks.

Early Drift Detection Method (EEDM) [2] extends DDM to improve the detection of gradual drifts. Instead of monitoring the *number* of misclassifications, they monitor the *distance between* misclassifications.

Krawczyk, in [48], states that sequential probability ratio tests, such as the Wald test, are the basis for detectors that use a *sequential analysis* method. The Cumulative Sum approach (CUSUM) proposed by Page in [68], computes a weighted sum of the last k examples to detect significant changes in a specified parameter of the probability distribution.

Methods using Hoeffding's (HDDM) and McDiarmid's inequalities were proposed in [31]. These methods use moving averages and weighted moving averages to detect when a statistic (p) deviates "statistically far" from its expected value. The authors use a normal distribution example where the warning would be at 95% (p differs from $\mu \pm 2\sigma$) and the drift level at 99% (p differs from $\mu \pm 3\sigma$). The algorithm generalises to any distribution but requires a desired significance

level α_W for warnings and α_D for drifts. The authors state that their two methods are independent of classifier choice, have $O(1)$ complexity for time and applicable to scenarios where "labels are harder to obtain".

Adaptive Windowing (ADWIN2) proposed by Bifet [5] maintains a window of variable size containing bits or real numbers. The size of the window grows when no change is apparent and shrinks when change is apparent. ADWIN requires $O(\log(W)$ update time and memory for a window size of W. Change is detected when two distinct sub-windows have a different average. The split index between these two sub-windows is considered as an indication of a drift. ADWIN is a very well known drift detection method, and the best-known method for *monitoring distributions of 2 different time windows*.

As we have previously stated, these drift detection methods rely on timely access to labels which is not a realistic assumption in the real world. Active-Learning is an approach that has not yet been extensively researched.

Unsupervised detection of drifts can rely on, according to [88], non-parametric statistical tests such as the CNF Density Estimation test [24], the multivariate version of the Wald–Wolfowitz test [32]. A two-sample Kolmogorov–Smirnov test, a Wilcoxon rank sum test, and a two-sample t-test can also be used to detect drifts in data distribution according to [85, 88].

SAND is a semi-supervised framework for learning from evolving streams, proposed by Haque in [40]. SAND trains k-NN estimators (an unsupervised learner, read clustering technique), which regroups k pseudopoints, to detect drifts using each model's confidence. The confidence is averaged from two calculated heuristics for each model: association and purity. Association decreases the further a data point is from a cluster, and purity indicates if a cluster contains

mostly one class or a mix of classes. SAND uses a semi-supervised change detector, that consists of storing the confidences in a sliding window, and comparing means from two sub-windows. Drifts are detected when the the mean of the newest confidence values drops below 95% of the mean of older confidence values ($m_a \leq 0.95 \times m_b$, where m_a is the mean for the sub-window containing the newest confidence values). The ensemble maintains a good accuracy using a limited amount of labelled data, but none of the experiments specify its execution time.

Our thesis contributes to narrowing the gap in research dealing with semi-supervised drift detection by extending FHDDM to work with unlabelled data, which we cover in the following section.

FHDDM and FHDDMS

Pesaranghader, in [70, 71, 72], proposes two concept drift detection algorithms, with similar characteristics. The first uses a single sliding window, whilst the second uses two sliding windows: one short and the other longer. The sliding window stores a boolean value indicating if the classifier predicted the class properly. His drift detection algorithms keep track of the maximum frequency value seen of correct predictions as well as the current frequency of correct predictions over the sliding window(s), then computes the difference between the maximum and current frequency. Hoeffding's inequality is used to determine the maximum desired difference between an empirical mean and a true mean of n random independent variables, without making any assumptions on the distribution of the data. A drift is detected by using Hoeffding's inequality to detect when a significant change occurs between the two frequency measures, meaning that the difference surpasses a given threshold. The author found that FHDDM

and FHDDMS were able to detect drifts with smaller delays and greater accuracy when compared to the state-of-the-art. Refer to algorithm 3 for the implementation of FHDDM.

Algorithm 3: Fast Hoeffding Drift Detection Method (FHDDM) [70, 71, 72]

```
function init(window_size, delta)
    (n, δ) = (window_size, delta);
    ε_d = sqrt((1/2n) ln(1/δ));
    reset();
function reset()
    w=[];
    μ^m = 0;
function detect(p)
    if w.size = n then
        w.tail.drop();
    w.push(p);
    if w.size < n then
        return False;
    else
        μ^t = w.count(1) / w.size();
        if μ^m < μ^t then
            μ^m = μ^t;
        Δμ = μ^m − μ^t;
        if Δμ ≥ ε_d then
            reset();
            return True
        else
            return False
```

2.5 Ensembles

Highlighted in review articles [43, 48, 67, 73, 79, 98], ensembles are considered as one of the most promising research directions nowadays. Ensembles, also

known as combined classifiers, multiple classifier systems (MCS), classifier ensembles, classifier fusion, and hybrid classifiers, can also be found reviewed in books [3, 52, 78, 82], and machine-learning handbooks [1, 25].

A supervised ensemble amalgamates any given number of classifiers [48, 65, 73, 77]. As an ensemble encapsulates other learning techniques, it needs a strategy to consider each of the classifiers' outputs to make a final prediction. What follows are examples of some such strategies, after which we review supervised ensembles as they pertain to data streams.

2.5.1 Ensemble voting strategies

When an ensemble predicts a class label for a new instance, each classifier comprised within must predict a class label for the instance and return it to the ensemble. The ensemble then needs to map these multiple, potentially different, predictions to a single output prediction. In order to do so, several functions have been proposed that apply weights to any permutation of the classifiers and/or of their predictions, and then apply a combination rule to these un/weighted values to finally vote on the final output of the ensemble.

Zhou, explains the three following voting techniques in [102, pp. 72-75].

Majority Voting

This voting scheme is reportedly the most popular. The method is as simple as it sounds. Each classifier in the ensemble votes for the class label it predicts and the winning class label is the one with at least half the votes. If none of the class labels obtain more than half of the votes, a "rejection option" can be given

and no prediction is made by the ensemble. In the case of binary classification, with n classifiers, the winning class must have $\lfloor \frac{n}{2} + 1 \rfloor$ votes. Equation 2.2 shows the formula for majority voting. $h_i^j(x) \in [0, 1]$ and takes value 1 if classifier h_i predicts class label c_j. l is the number of class labels.

$$H(x) = \begin{cases} c_j, & \text{if } \sum_{i=1}^{n} h_i^j(x) > \frac{1}{2} \sum_{k=1}^{l} \sum_{i=1}^{n} h_i^k(x) \\ rejection, & \text{otherwise} \end{cases} \tag{2.2}$$

Plurality voting

This technique is almost identical to majority voting, with the slight difference that it does not require the final class label to obtain more than half of the votes; the final class label obtained the most votes. Ties are broken arbitrarily, and plurality coincides with majority voting in the case of binary classification. Equation 2.3 shows the formula for plurality voting. $h_i^j(x) \in [0, 1]$ and takes value 1 if classifier h_i predicts class label c_j.

$$H(x) = c_{arg_j max \sum_{i=1}^{n} h_i^j(x)} \tag{2.3}$$

Soft voting

Soft voting differs from plurality voting in that it requires each of the classifiers in the ensemble to output a confidence value (usually in $[0, 1]$) for their prediction for each class value or output the probabilities that an instance belongs to a given class label for all class labels. In the case of *simple soft voting*, the average probability for each class label is computed over the predictions of all classifiers. The probability of the final class label is given by equation 2.4. Again,

$h_i^j(x) \in [0,1]$ and takes value 1 if classifier h_i predicts class label c_j. L is the set of class labels, l here is any label in L.

$$H(x) = max(\frac{1}{n}\sum_{i=1}^{n} h_i^l(x)\ \forall l \in L) \tag{2.4}$$

There are variations of soft voting where a weight is applied to either each of the classifiers, or to each class, or to each instance. However, Zhou states that in practice, instance weights are typically not used as it may involve a large number of weight coefficients.

2.5.2 Review of Ensemble Classifiers

Krawczyk in his survey reviewing ensemble learning for data streams [48], categorises ensemble learning from data streams for supervised classification using two criteria. The first is the data processing method: online, as in one by one, or in chunks. The second one deals with the ensembles' ability to deal with stationary or non-stationary streams, meaning without or with concept drifts. We re-use these two criteria for our review.

As our contributions deal with non-stationary streams, we will focus on reviewing techniques for those.

Chunk-based ensembles for stationary streams

Learn^{++} builds incremental neural networks (NN) and uses majority voting to combine their outputs [74, 48]. However, this technique is unsuitable for large streams, as all NN are retained, memory used continuously increases.

AdaBoost RAN-LTM combines AdaBoost.M1 and Resource Allocating Network with Long-Term Memory. A predetermined number of base classifiers are trained then incrementally updated on new instances [45, 48]. The forgetting factor is "suppressed" to more efficiently weight voting combinations. Same as for Learn^{++}, infinite streams may cause memory usage issues.

Growing Native Correlation Learning (Growing NCL) aims to co-train an ensemble using neural networks. Their algorithm allows for a trade-off between the forgetting rate and the ability to adapt to new data. Experimental results seemed to indicate that fixed size ensembles have a better generalisation ability whereas the growing size may "easily overcome the impact of too strong forgetting" [57, 48]

Bagging^{++} was shown to be faster than Learn^{++} and NCL, with comparable results [101, 48]. It improves upon Learn^{++} through the use of bagging to build new models from incoming data. The ensemble is composed of diverse classifiers selected from a set of four base classifiers.

Online ensembles for stationary streams

OzaBag allows the use of bagging in an online setting by replicating newly arriving instances by updating the classifier with k copies of these incoming instances, where $k \sim Poisson(1)$ distribution [66, 48].

Leveraging Bagging (LB), as explained above, adds more randomisation to the input and output of the ensemble, by increasing sampling from $Poisson(1)$ to $Poisson(\lambda)$ where λ is a parameter for LB [6, 48].

OzaBoost (also known as Online Boosting) uses a fixed amount of classifiers, which are all trained sequentially on all incoming instances [66, 48]. When a

classifier misclassifies an instance, the instance's weight is increased to boost its importance to the subsequent classifiers. The first weight is the highest possible value: $\lambda = 1$. Then, classifiers are trained $k = \text{Poisson}(\lambda)$ times. Each classifier keeps a count of instances it misclassified (ϵ). If a classifier classifies correctly, then $\lambda = \lambda * \frac{1}{2(1-\epsilon)}$, otherwise $\lambda = \lambda * \frac{1}{2\epsilon}$. The new weight λ is then used to train the next classifier.

Ultra Fast Forest of binary Trees (UFFT) makes use of an ensemble of Hoeffding Trees [36, 48]. The split criterion used can only be applied to binary classification problems, but binary decomposition allows for multi-class problems to be handled. Each pair of classes has its own binary tree, which is updated when a new instance has a true class label for one of its two classes.

Ensemble of Online Sequential Extreme Learning Machines (EOS-ELM) [54, 48] is an improvement on the existing OS-ELM technique. EOS-ELM is composed of initially diverse online neural networks (NN) which are initially trained by using random weights for their single hidden layer. The outputs of the NNs are averaged to obtain the ensemble's output. Incoming data is used to train all NNs in the ensemble sequentially. The authors do not, however, indicate how the classifiers in the ensemble maintain their diversity while learning from the stream.

Other online ensemble techniques for stationary streams include, among others, Ensemble of adaptive-size Hoeffding trees [9], Online Random Forest [21, 80], Online Mondrian Forest [53], and Hoeffding Option Trees [35].

Chunk-based ensembles for non-stationary streams

Many chunk-based ensembles for non-stationary streams use the following approach:

1. Given a chunk of data D obtained from a stream, evaluate ensemble classifiers C_j on D using evaluation measure $E(C_j)$
2. Train a new classifier C_i on D
3. Add C_i to the ensemble if its size permits it, or replace a given classifier in the ensemble

The following algorithms use this approach to varying degrees.

Street and Kim, in [90], propose the well-known Streaming Ensemble Algorithm (SEA). SEA follows the approach stated above, with a precision to be added in step three. If the ensemble size has not been reached, C_i is added to the ensemble. Otherwise, C_i, and all other classifiers in the ensemble predict class labels for the next training chunk, and the classifier with the worst quality is replaced with C_i. In order to promote diversity and avoid over-fitting, SEA favours classifiers that correctly classifies instances for which the other classifiers in the ensemble are "nearly undecided" [90]. Finally, it uses majority voting to map the output predictions of the classifiers for an ensemble prediction. Again, due to the ensemble's reliance on prequential accuracy (and therefore true and timely class labels) to replace the lowest quality classifier, it is not the most suited to deal with late-arriving labels or missing labels. By that same logic, it is neither a prime candidate to be extended to deal with semi-supervised problems. Other potential issues include determining good chunk and ensemble sizes, that old classifiers can outweigh newer ones thereby "slowing down adaptation to newer

concepts" [48]. SEA does present the following qualities: it uses approximately constant memory, runs quickly, and can adjust quickly to concept drift.

Accuracy Weighting Ensemble (AWE), proposed by Wang et al. in [94], follows the typical approach, with a variation on the replacement strategy. The idea behind AWE is to weight each classifier using a particular variant of the mean square error on the newest chunk using cross-validation. The weight of a classifier is reversely proportional to the estimation of its prediction error. Classifiers are pruned if they predict randomly or worse, or by only keeping a subset of those with the highest weights. This removes classifiers that do not model the current concept well or makes room for newer classifiers to learn new concepts. However, just like previous classifiers, AWE relies on the prequential accuracy for its pruning strategy, and therefore may have difficulty with missing or late-arriving class labels, and requires a different pruning strategy for it to be extended to deal with semi-supervised data. Additionally, the use of cross-validation, for the computation of weights, increases AWE's execution time.

Chu and Zaniolo proposed the Adaptive Boosting ensemble (Aboost) [18], that mixes boosting with a chunk-based input. To detect concept drifts, each time a chunk is received, the ensemble's error is computed. If a concept drift is detected, the entire ensemble is completely reset; otherwise, each instance of the chunk is assigned a weight based on the ensemble's error. This weight is then used to train a new classifier from the weighted chunk, which is added to the ensemble if it is not full; otherwise, it replaces the oldest classifier in the ensemble. Soft voting is used to map the classifier's predictions to a single output for the ensemble. In their experimental evaluation, the authors found that their approach outperformed SEA [90] and AWE [94] in terms of predictive accuracy. Their technique is also faster, uses less memory and is more adaptive to concept

drifts. False flags for concept drifts may cause issues, however, as the entire ensemble is reset. Moreover, as for SEA and AWE, prequential accuracy is relied upon, in this case, to compute weights, making this algorithm susceptible to failure when dealing with missing or late-arriving class labels.

Elwell and Polikar also proposed a chunk-based approach inspired by boosting [27] called Learn^{++}.NSE, that as its name suggests improves upon Learn^{++} to deal with non-stationary environments. The typical approach is again used, with each incoming chunk being used to train a new classifier in the ensemble. Learn^{++}.NSE is similar to Adaptive Boosting but differs in that the weights are assigned based on how a classifier predicts as compared to the entire ensemble. A higher weight is assigned if a classifier predicts correctly when the ensemble does not, and a lower weight is assigned in the opposite case. The authors allow their ensemble to learn from new instances while reinforcing existing and still relevant knowledge by giving more importance to recent misclassifications to promote the learning of newer concepts. Furthermore, Learn^{++}.NSE uses a weighted majority voting strategy to combine the classifier's predictions, allowing them to assign a minuscule weight to old classifiers representing old concepts. This allows the algorithm to disable a classifier when it does not predict well and to re-enable it when the concept recurs. However, as Learn^{++}.NSE allows for an unlimited number of classifiers, the memory used keeps increasing. Again, it presents the same issues for late or missing labels.

The following strategies use an alternative approach, that differs from the typical one stated at the start of this section.

Knowledge-Based Sampling (KBS), a boosting-like method, is proposed by Scholz and Klinkenberg in [81]. For each new chunk, depending on which has the best

resulting accuracy, a new classifier is trained, or the last trained classifier is updated with the new data. In a boosting-like style, weights are assigned to each instance of a chunk, and weights are also attributed to each classifier based on its predictive accuracy for the new chunk. These weights allow for the pruning of poorly performing classifiers, as well as quickly detecting concept drifts. KBS has been shown to not perform well when dealing with a recurring concept right after an abrupt drift. The authors state that KBS is computationally efficient, and empirically showed that it can "outperform more sophisticated adaptive window and batch selection strategies" [81]. However, as all of the other algorithms seen, it relies on prequential accuracy, in this case for its pruning strategy, making it susceptible to failure when dealing with missing or late-arriving class labels.

Batch Weighted Ensemble (BWE) was proposed by Deckert [20] to deal with abrupt and gradual concept drifts. BWE consists of an ensemble with an embedded simple Batch Drift Detector (BDDM). BDDM builds a simple regression model on accumulated accuracy values from each incoming instance, and will only process batches with decreasing trends. BDDM outputs warning or drift flags. When a drift is possible (warning or drift flag), a new classifier is trained on the new batch, and any classifiers that perform worse than a random classifier are discarded. Again, we can see a reliance on prequential accuracy within the ensemble, and therefore it might cause issues when class labels are missing or late-arriving.

Weighted Aging Ensemble (WAE), proposed by Wozniak in [97], is inspired by AWE [94], and extended with two modifications. The first is that classifiers are weighted based on their prequential accuracy, but also on how much time they

spend within the ensemble. The last modification adds classifiers to the ensemble based on their diversity measure.

Other alternative approaches include Accuracy Updated Ensemble (AUE2) by Brzezinksi [17], and Ensemble Tracking (ET) which deals specifically with recurring concepts by Ramamurthy [75].

Online ensembles for non-stationary streams

Nishida, in [62], proposes an enhanced version of Adaptive Classifier Ensembles (ACE) that adds a pruning method and improves the voting method. ACE comprises one online classifier, a set of batch classifiers, and a drift detection mechanism. The online classifier is trained on every incoming instance, and a fixed-sized buffer keeps instances that were recently seen. When the buffer is full or a drift is detected, a new batch classifier is created to summarise the data for that time period, the buffer is emptied, and the online classifier is reinitialised. A weighted majority vote is used to compute the output for the ensemble, for which the classifier weights are determined by the predictive accuracy of the classifiers over the last W seen instances. To manage the number of classifiers in the ensemble, the oldest classifier is usually pruned, unless it has good predictive accuracy for the last seen W instances, making use of older data when concepts recur. The size of the buffer kept does not seem to affect ACE's ability to detect abrupt concept drifts, but determining the size of the sliding window containing the W most recently seen instances might prove to be difficult. Finally, ACE requires timely and correct class labels in order to function correctly, which means it is not a prime candidate to be extended to deal with semi-supervised tasks, or that it be particularly applicable for real-world applications where labels can arrive late, or not at all.

In [12], Bifet proposes Adaptive Windowing (ADWIN) Bagging, which is merely the result of adding an ADWIN drift detector to online bagging (briefly explained in section 2.3.6. ADWIN is responsible for replacing the worst performing classifier in the ensemble with a new classifier when a drift is detected.

Other algorithms that learn using online ensembles over non-stationary streams include [16, 46, 47, 51, 59, 89, 99], as well as the SAND semi-supervised framework [40].

In summary, only one of the articles reviewed that dealt with non-stationary streams did not rely on the prequential accuracy, whether for their weighting scheme, their drift detection, or some other reason, any attempt to extend these algorithms to deal with semi-supervised learning tasks could prove arduous. This constitutes a gap in research which we narrow with our contributions in the next chapter.

Chapter 3

Learning from Evolving Streams via Self-Training Windowing Ensembles (LESS-TWE)

In this chapter, we introduce our methodology, entitled Learning from Evolving Streams via Self-Training Novel Windowing Ensembles (LESS-TWE). First, we provide the reader with an overview of our framework, then discuss our contributions. These include a weighted soft voting scheme, a novel windowing technique that combines sliding and tumbling techniques, a non-selective self-training method, and the extension of an existing concept drift algorithm to remove its dependency on labelled data. Finally, we use a toy example to explain how data is processed from a stream.

3.1 Overview of LESS-TWE

Figure 3.1 illustrates how our contributions fit together and operate in one iteration of an interleaved test-then-train loop (defined in section 4.4.2).

When predicting labels, data can arrive in chunks or one-by-one. In this example, data arrives one instance at a time. The first step is for the ensemble to predict a label for the new data instance; the classifiers in the ensemble each predict a label and a weighted soft vote scheme selects a winning label for the ensemble. The prediction confidence for that label is used to detect drifts. If a drift is detected, a subset of classifiers in the ensemble are first reset, and subsequently trained on a sliding window of previously seen labelled data (stored in the backup window). If no drifts were detected, then the label predicted replaces the real label value in the novel window, thereby self-training. This window stores N times the number of instances processed at a time (in this example, one), where N is the number of classifiers in the ensemble. A single classifier is trained per iteration of the interleaved test-then-train loop, in a cyclical manner, meaning that each classifier in the ensemble is trained only once every N loop iterations. That is, a different classifier from the ensemble is trained at the next loop iteration.

Next, we provide a summary of our contributions.

3.1.1 Summary of Contributions

We propose a weighted soft voting scheme that uses a hyperbolic tangent function. We choose the *tanh* function, as it allows us to approximate a logistic function while also being less computationally expensive [55, p. 10].

Next, we propose a novel windowing technique, exclusive to ensembles. Our technique allows every classifier in the ensemble to train from each data point in the stream **exactly once** similar to tumbling windows, as opposed to sliding windows where a data point is trained on **at least once**. As such, our method

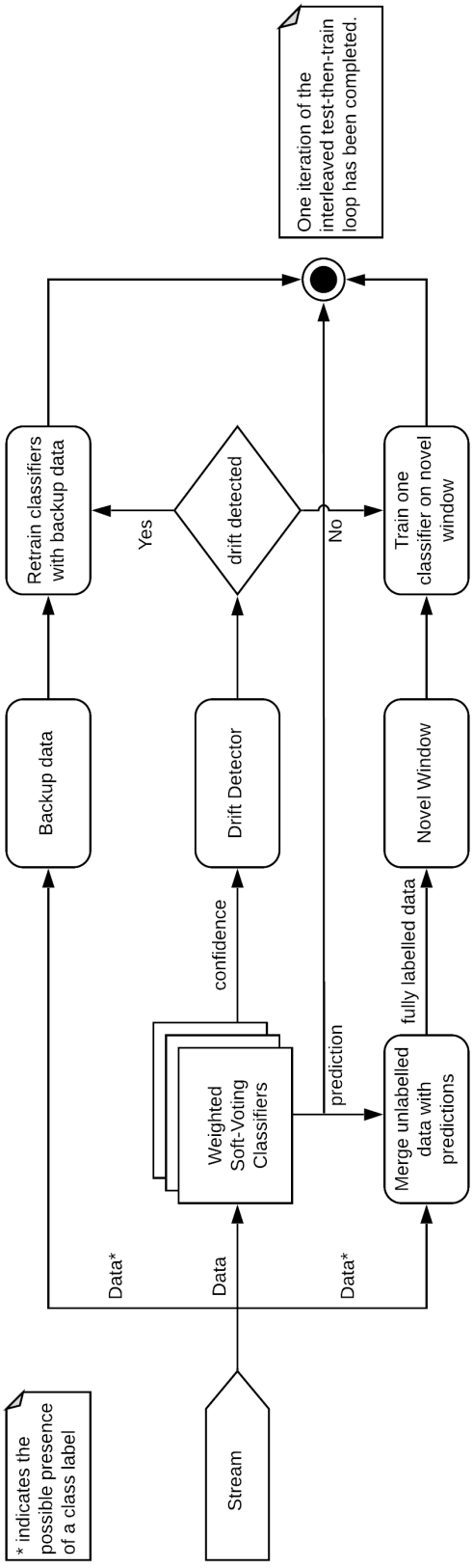

FIGURE 3.1: High-Level Overview of the LESS-TWE methodology

presents a trade-off between accuracy and execution time. Only one classifier per batch is trained on the window, therefore each classifier trains on a specific data instance at different times, resulting in delayed training. This means that only one classifier in the ensemble is trained on the newest data before the ensemble predicts labels for the subsequent batch. The motivation for this technique is to determine if we can spend less execution time training the classifiers and to investigate how progressively delaying training of some of the classifiers in the ensemble affects concept drift detection and classification performance.

Our next contribution relates to selective self-training. In this semi-supervised learning algorithm, a classifier assigns a label it predicts to unlabelled data for future learning, if it is highly confident it is correct. There have not yet been any attempts, to the best of our knowledge, to investigate how self-training performs if labelling all unlabelled data, regardless of the classifier's confidence in its predicted label.

Finally, our last contribution is an extension of the Fast Hoeffding Drift Detecting Method for evolving data Streams (FHDDMS), which detects drifts using the prequential accuracy. In our extension, we propose three schemes, one of which makes use of the average of the ensemble's classifiers' confidences. Our extension allows FHDDMS to run without any labelled data, therefore making it an unsupervised drift detector.

3.2 Voting Classifier: Weighting and Voting Schemes

The reader may refer to section 2.5 for background on ensembles and voting classifiers.

A voting classifier was utilised from the mlxtend library[1], developed by Sebastian Raschka. Mlxtend is an extension of the scikit-learn python machine learning library, to handle classification in a streaming setting.

The mlxtend voting classifier implements two voting strategies: the first being "soft" voting and the second being "hard" (majority) voting; refer to section 2.5.1 for their definitions.

We implemented two additional soft voting schemes. These schemes require its classifiers to output a probability or confidence in the class label prediction. The first method makes use of a logistic function with a sigmoid curve to weight the probability, centred around a confidence level. Let α be the parameter that defines the confidence level, let $\alpha = 50\%$. The second method, explained soon thereafter is equivalent to the function of the first method while being less computationally intensive.

Logistic functions are defined by the function in equation 3.1

$$f(x) = \frac{L}{1 + e^{-k(x-x_0)}} \tag{3.1}$$

where e is the natural logarithm base, x_0 is the x-value of the sigmoid's midpoint, L is the curve's maximum value, and k is the steepness of the curve.

[1] http://rasbt.github.io/mlxtend/user_guide/classifier/EnsembleVoteClassifier/

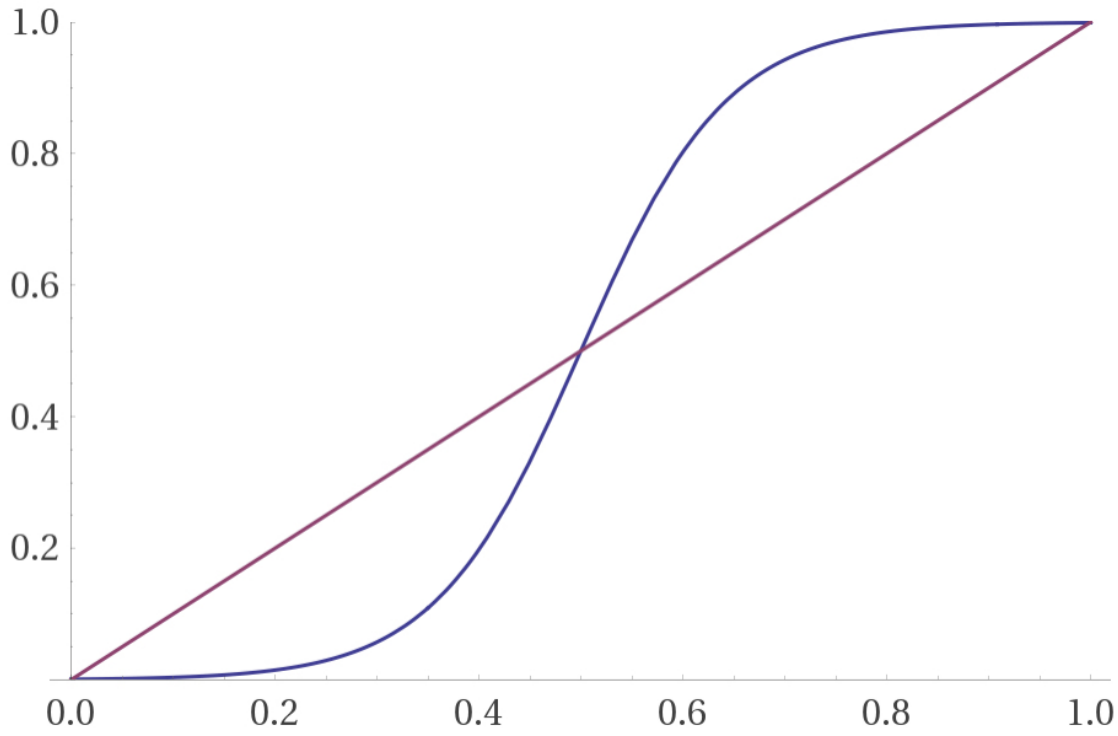

FIGURE 3.2: Voting classifier weighting functions

The logistic function that we initially selected had the following values for the variables listed above: $x_0 = 0.65$, $L = 1$, and $k = 14$. These values were selected based on the resulting graph for values of $x \in [0, 1]$ and $f(x) \in [0, 1]$. As figure 3.2 illustrates, predictions with less than approximately forty percent probability are essentially treated as predictions with zero probability; predictions with under a seventy percent probability are diminished in importance, and those over seventy percent are boosted. Refer to table 3.3 to see how the values change by five and ten percent increments. The reasoning motivating this choice is our intuition that we should put more trust in predictions with over seventy percent (70%) probability, put less trust in predictions with a probability of under seventy percent (70%), and virtually none in predictions with a probability of under forty percent (40%).

TABLE 3.1: Experimental test of weighting function parameters

Parameters	**Accuracy**	κ	κ_t	κ_m
no weighting	81.95	61.47	61.10	52.78
10, 0.65	83.56	64.77	64.57	57.00
14, 0.45	84.28	66.26	66.12	58.88
14, 0.50	**84.77**	**67.34**	**67.17**	**60.15**
14, 0.55	**84.72**	**67.25**	**67.06**	**60.02**
14, 0.65	83.75	65.12	64.98	57.49
14, 0.6	84.22	66.13	65.98	58.71
14, 0.75	81.73	60.86	60.63	52.21
14, 0.85	80.51	58.65	57.99	49.01
5, 0.65	82.77	63.03	62.85	54.91
7, 0.65	83.04	63.56	63.44	55.62

However, after extensive experimentation, of which the results are found in tables 3.1 and 3.2, we found that setting $x_0 = 0.50$ and $k = 14$ was more optimal than the values we initially selected through intuition for the x_0 and k parameters.

The same test as above was repeated, but, over four separate data sets (SEA, SINE1, CIRCLES, MIXED which are covered in section 4.3) for the two best combinations of parameters and the one from our intuition ($k = 14$, and $x_0 \in [0.5, 0.55, 0.65]$), and ran only once. Table 3.2 shows, again, that $k = 14$, $x_0 = 0.50$ seems to be the optimal set of parameters as it obtains the best results for three out of four data sets.

We must note that $\forall x \in [0, 1], f(x) \in [0, \frac{1}{1+e^{-7}} \approx 0.9991]$ for the function in equation 3.1 , we could then divide the function by its value for the maximum value for $p_i(X)$ (which is 1) to ensure that our function covers all values in $[0, 1]$, but this is unnecessary as all values will be in the same range, and it adds another calculation. Refer to figure 3.2 for the plot of equation 3.2.

TABLE 3.2: Experimental test of weighting function parameters on 4 datasets

Dataset	**Parameters** ($k = 14$)	**Accuracy**	κ	κ_t	κ_m
sine1	$x_0 = 0.5$	84.30	68.60	68.69	68.73
	$x_0 = 0.55$	84.15	68.30	68.39	68.44
	$x_0 = 0.65$	83.79	67.59	67.67	67.72
mixed	$x_0 = 0.5$	79.95	59.90	60.07	60.12
	$x_0 = 0.55$	79.89	59.78	59.96	60.01
	$x_0 = 0.65$	79.60	59.20	59.38	59.43
circles	$x_0 = 0.55$	79.72	59.45	59.44	59.15
	$x_0 = 0.65$	79.43	58.87	58.87	58.57
	$x_0 = 0.5$	77.72	55.45	55.45	55.13
SEA	$x_0 = 0.5$	85.20	68.27	68.10	61.28
	$x_0 = 0.55$	84.25	66.25	66.05	58.80
	$x_0 = 0.65$	83.41	64.38	64.25	56.60

TABLE 3.3: Probabilities after weighting

Probability (%)	0-10	20	30	40	50	60	65	70	75	80-85	90-100
Weighted (%)	0	1	6	20	50	80	89	94	97	99	100

$$\frac{1}{1 + e^{-14(p_i(X) - 0.50)}} \tag{3.2}$$

Our proposed voting strategy therefore computes the average of this logistic function using the prediction of each classifier for each class label. See the following equation:

$$\frac{1}{n} \sum_{i=1}^{n} \frac{1}{1 + e^{-14(p_i(X) - 0.50)}} \tag{3.3}$$

where n is the number of classifiers in the voting ensemble, and $p_i(X)$ is the probability of classifier i predicting that the tuple in question belongs to class X.

We propose another weighting equation to determine if it is possible to achieve similar results by using a different function that resembles the logistic sigmoid function presented above while being less computationally intensive. As LeCun states in [55, p. 10], "hyperbolic tangent functions often converge faster than the

TABLE 3.4: Weighting function benchmark results

Sigmoid function	**Hyperbolic tangent**
1441.05 ns/class/loop	407.40 ns/class/loop

standard logistic function". Our hyperbolic tangent weighting function is seen in equation 3.4 and the sum is seen in equation 3.5. The plot of equation 3.4 is completely identical to the plot of equation 3.1.

$$f(x) = \frac{1 - tanh(3.5 - 7x)}{2} \tag{3.4}$$

$$\frac{1}{n}\sum_{i=1}^{n}\frac{1 - tanh(3.5 - 7 \times p_i(X))}{2} \tag{3.5}$$

The calculation of averages presented in equations 3.3 and 3.5 are calculated for each class label. The class label with the highest average is thereafter selected as the winner in the vote. We aim to test the hypothesis that weighting the predictions by an exaggeration of their probability will increase the accuracy of our voting classifier.

In order to confirm that the new weighting function was more computationally efficient, we compared the total running time to evaluate each weighting function ten million times on a random value in [0, 1]. The benchmark showed, as seen in table 3.4, that the logistic function takes almost 3.5 times longer to compute than the hyperbolic tangent function, which will, therefore, be used for the remainder of this thesis.

Algorithm 4 shows the implementation of this new voting scheme using the hyperbolic tangent function. Figure 3.2 plots the proposed weighting function and an unweighted prediction represented by $f(x) = x$, for $x \in [0, 1]$. This figure shows how much a value is boosted or reduced compared to its original value.

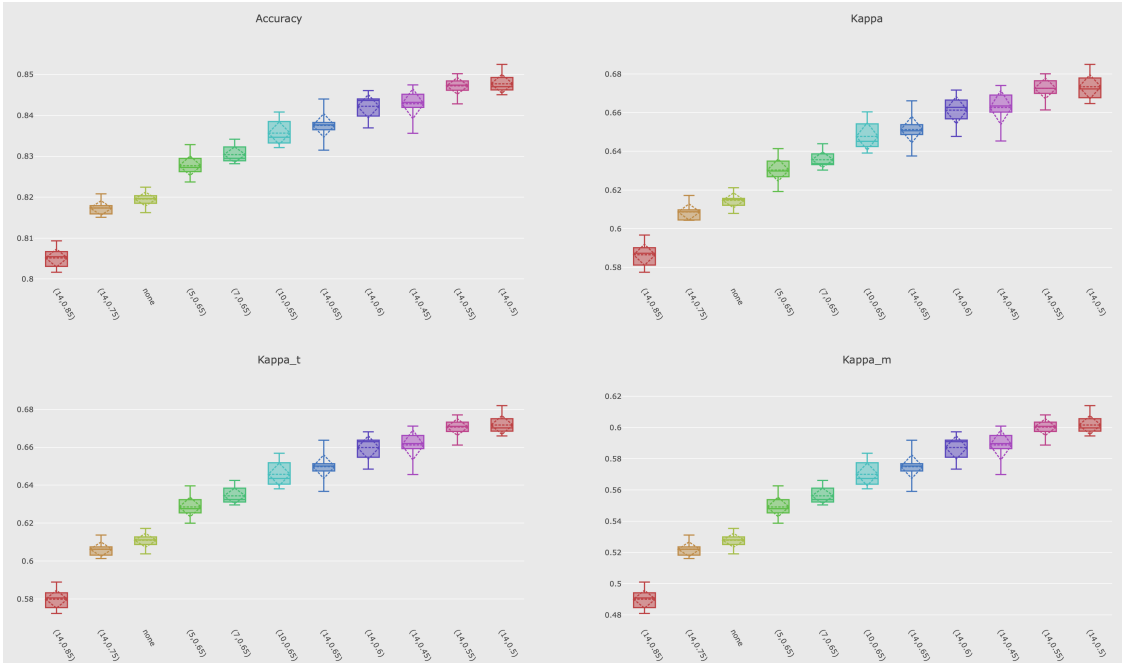

FIGURE 3.3: Boxplots of experimental test for logistic function parameters

Given that the hyperbolic tangent function selected is equivalent to the logistic sigmoid function, it is not necessary to test its parameters experimentally as they give the exact same output (difference between the two functions at any given point does not exceed much more than 10^{-16}). Figure 3.3 shows the boxplots of table 3.1. The whiskers correspond to the minimum and maximum values over 10 runs with each parameter. The dotted diamond corresponds to the standard deviation, the dotted line in the box corresponds to the mean, while the solid line corresponds to the median.

3.3 Hybrid sliding-tumbling windows

In [48], Krawczyk mentions that one of the many issues in data stream mining is execution time, in the sense that our algorithm must learn faster than tuples

Algorithm 4: tanh weighting scheme for voting classifier

```
1  function voting_classifier.predict(X)
       /* The probabilities are stored in a 1 × 2 matrix (using binary
          classification for simplicity) where each column represents a
          class, and each row represents a sub-classifier            */
2      weighted_probabilities = [];
3      foreach classifier ∈ voting.classifiers do
4          array = classifier.predict();
5          w_p = [weight(array[0]), weight(array[1])];
6          weighted_probabilities.append(w_p);
       // map sub-classifier probabilities to single probability
7      avg = weighted_probabilities.avg_over_columns();
8      prediction, prediction_probability = max(avg), class_max_value(avg);
9      return prediction, prediction_probability;
10 function weight(p)
11     return (1 − tanh(3.5 − 7p)) / 2;
```

can arrive. In our case, we aim to determine if we can delay training of some classifiers in our ensemble, and how it affects execution time and classifying performance, as well as drift detection performance.

We propose the following algorithm, implemented in algorithm 5, titled *Sliding-Tumbling Windows for Training Ensembles*. Let the number of tuples (single tuple or chunk) used in the interleaved test-then-train loop iterations be *number_of_tuples*, and let *number_of_classifiers* be the number of classifiers in the ensemble. The ensemble will have a window size of *number_of_tuples* $\times$ *number_of_classifiers*. At every iteration of the interleaved test-then-train loop, we will append the new tuples to the ensemble's window and train a single classifier in the ensemble on that window. For the next *number_of_classifiers - 1* iterations, we will train the remaining *number_of_classifiers - 1* classifiers in the ensemble. We do this so that from the point of view of the ensemble, we are training using sliding windows. However, from the point of view of each classifier in the ensemble, we are training them using tumbling windows.

While not used for the same purpose, the same sliding batches were used to improve CDC-Stream in [22, 26]. Figure 3.4, taken from D'Ettore's thesis, shows how the algorithm works for three (3) classifiers in the ensemble with a batch size of one (1) and a window size of three (3). Each classifier will only learn from the same coloured batch; meaning that at time t, only a single classifier has enough tuples to learn from, but the others will learn at time t+1 and finally at time t+2. Each classifier will be learning from what essentially is a tumbling window, from their point of view, just not all from the same one, or at the same time.

To clarify the example, (using figure 3.4) at time t_1, our ensemble will receive only one instance, add it to the sliding window and train classifier c_1 on the window. At time t_2, another instance will be added to the window and the ensemble will train classifier c_2 on the new window containing two tuples. At time t_3, the ensemble will train classifier c_3 on the first purple chunk (the window now has its maximum of 3 instances). At time t_4, c_2 will learn from the first blue chunk. And at time t_5, c_3 will learn from the first green chunk. This process loops indefinitely.

The motivation for this technique is to determine if we can spend less execution time training the classifiers. We also aim to investigate how progressively delaying training of some of the classifiers in the ensemble affects concept drift detection and classification performance while also hopefully reducing execution time.

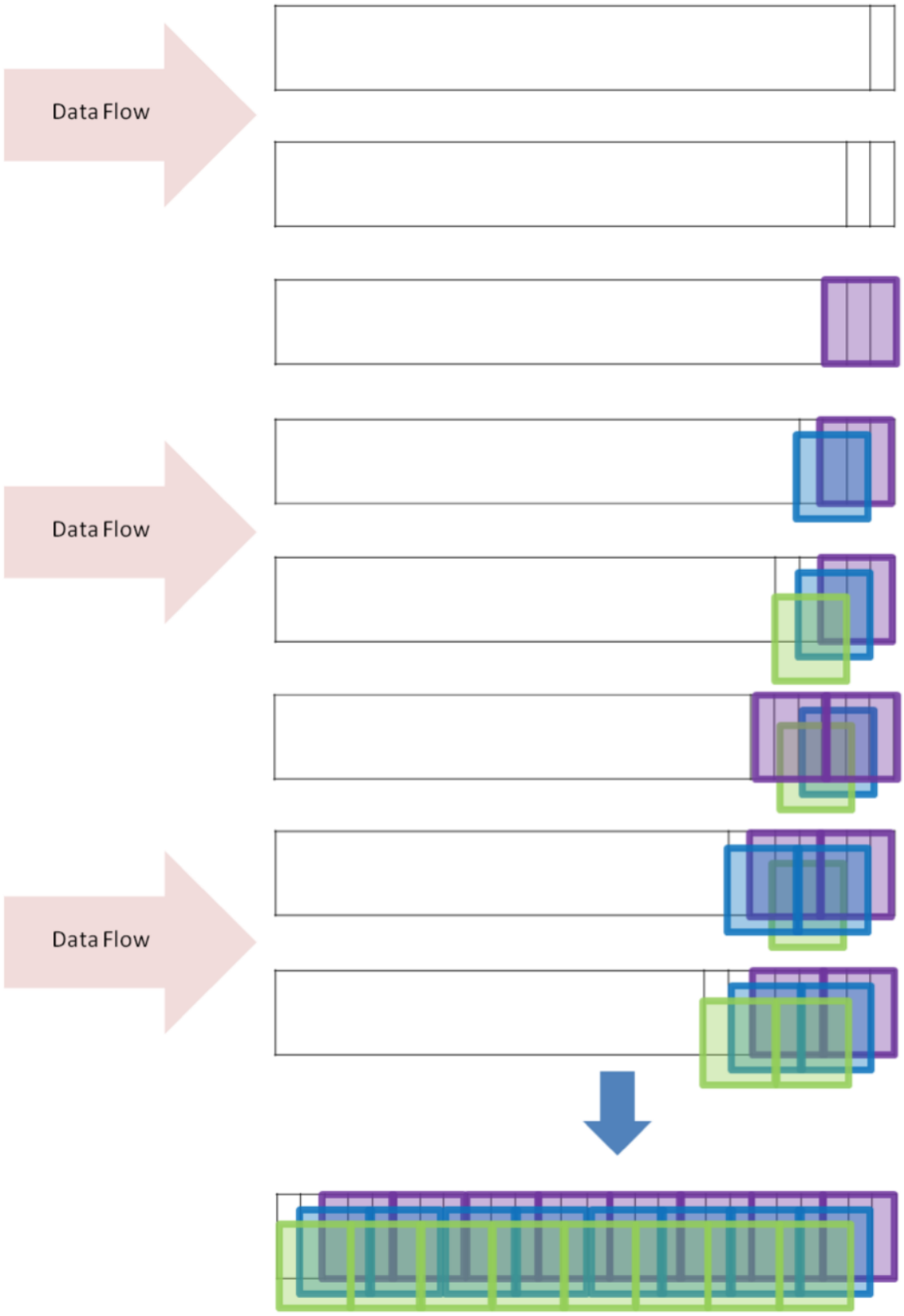

FIGURE 3.4: Sliding Tumbling windows [22].

Algorithm 5: Sliding-Tumbling Windows for Training Ensembles

Result: at least one classifier in the ensemble was trained

```
// voting_ensemble stores a classifier list, its count, and, the
   index of the current classifier to train
function ensemble.train(X, y)
    classifier_to_train = classifier_list[index];
    index = (index + 1) modulo (number_of_classifiers);
    classifier_to_train.partial_train(X, y);
```

3.4 Non-Selected Self-Training

The interleaved test-then-train methodology in a data streaming setting has a rather significant flaw if we consider a real-life scenario: we are assuming that we obtain the ground truth immediately after testing. This means that we assume that the ground truth is always available in a fraction of a second after testing our models. For the vast majority of cases, this approach is not realistic, again, in a real-world setting.

Therefore, we aim to determine if we could re-use the idea behind self-training in an offline setting to reduce our dependency on labelled data in the online streaming setting for the interleaved test-then-train method. We propose an approach that is, as previously stated, similar to selective self-training in that we use the classifier's prediction, as opposed to correctly labelled instances, when training.

However, using only the predictions to train our model in an online setting is a recipe for disaster. The known cons to using self-training in an offline setting are that it can reinforce classification errors; it is, therefore, logical for us to conclude that we will encounter the same risks in porting this idea to a streaming setting.

Given the restriction of limiting reinforcing misclassification errors, our goal is to determine at what ratio of predictions to ground-truth our voting ensemble's

accuracy would decline and by how much.

The algorithm behind this idea is very straightforward: it consists in duplicating the class label array and replacing at random a given fraction of true labels with predictions from the ensemble, if a drift was not detected right before. See algorithm 6 for the pseudocode (function *swap ground truth with predictions*).

Algorithm 6: Online self-training

```
1  while stream.has_more_instances() do
2      X, y = stream.get_next_tuples(number_of_instances_to_fetch);
3      predictions, probabilities = voting_ensemble.predict(X);
4      drift_detected = voting_ensemble.detect_drift(predictions, probabilities);
5      if percentage ≠ 100 and drift not detected then
6          y = swap_ground_truth_with_predictions(y, predictions, percentage);
7      voting_ensemble.train(X, y);
   // This algorithm only shows the steps required to modify the ground
      truth array
8  function swap_ground_truth_with_predictions(y, predictions, percentage)
9      for index = 0 ; index < length(y) ; index+ = 1 do
10         if random_number_between(0, 100) > percentage then
11             y[index] = predictions[index];
12     return y;
```

3.5 Extending FHDDM/S to function without labelled data

Additionally, this thesis builds upon Pesaranghader's work described in section 2.4.2

FHDDMS, the drift detection algorithm proposed by Pesaranghader in [72] relies on the immediate knowledge of labelled data. The drift detection mechanism in FHDDMS relies on storing, in a sliding window, whether or not the classifier correctly predicted the class. This method only applies to a select few

domains where the correct label becomes available almost instantaneously after the data arrives, and to all synthetic data streams of course. For the vast majority of domains, this method is not applicable.

It is for that reason that we have set out to study if FHDDM/S is still able to detect drifts when completely removing its dependency on any knowledge of the ground truth in an online streaming setting.

In order to do so, we have opted to extend FHDDM and FHDDMS such that its sliding window(s) now store(s) either a numerical value for a prediction probability, or a different boolean.

Our extension, if using prediction probabilities, requires classifiers that can output such values. When predicting the class for a data point, each classifier in the ensemble outputs its probability for each class label. For example, a classifier could output the following class probabilities for a non-binary classification task $\{A : 24\%, B : 11\%, C : 65\%\}$.

We implemented three approaches. The first two use multiple drift detectors, one per classifier in the ensemble. In addition, the third uses a single drift detector by averaging values from each classifier. The drift detector in question is our extended version of FHDDM/S.

The first approach stores boolean values in its detectors' sliding windows. The boolean value indicates if a classifier predicted the class that was voted as the winner for the ensemble.

In the second, each window stores its classifier's probability for the class that won the vote. These values may or may not be weighted, using the hyperbolic tangent function from section 3.2.

TABLE 3.5: Testing whether to use weighted probabilities for detecting drifts

Dataset	**Weighted/Unweighted**	**Accuracy**	κ	κ_t	κ_m
SEA	✓	84.53	66.77	66.66	59.53
	×	84.42	66.51	66.42	59.23
circles	×	78.35	56.71	56.70	56.39
	✓	78.25	56.51	56.50	56.19
mixed	✓	80.00	60.00	60.18	60.23
	×	79.97	59.94	60.11	60.16
sine1	✓	84.35	68.71	68.79	68.84
	×	84.30	68.60	68.68	68.73

The last method also stores class probabilities, but stores the *average* probability from all classifiers in the ensemble into a single window in a single drift detector. As for the previous approach, the probabilities averaged are those for the winning class. Furthermore, the values may or may not be weighted using that same function, as for the previous approach.

After extensive experimentation, results indicate that the weighted probabilities were no more useful than the unweighted values for detecting drifts as the difference in performance between the two is negligible. To illustrate, a SEA dataset was generated with 10% noise with one-hundred thousand instances, with four concepts and three abrupt concept drifts at every twenty-five thousand instances. The *CIRCLES*, *MIXED* and *SINE*1 datasets were also used (refer to section 4.5). When drifts are detected, all of the classifiers in the ensemble were completely reset, and one drift detector for the entire ensemble was used. In order to determine whether to use unweighted or weighted probabilities for the drift detector, we looked at the accuracy and kappa statistics of the entire stream (covered in section 4.5). Table 3.5 shows these results. The values in the table are averages over ten runs.

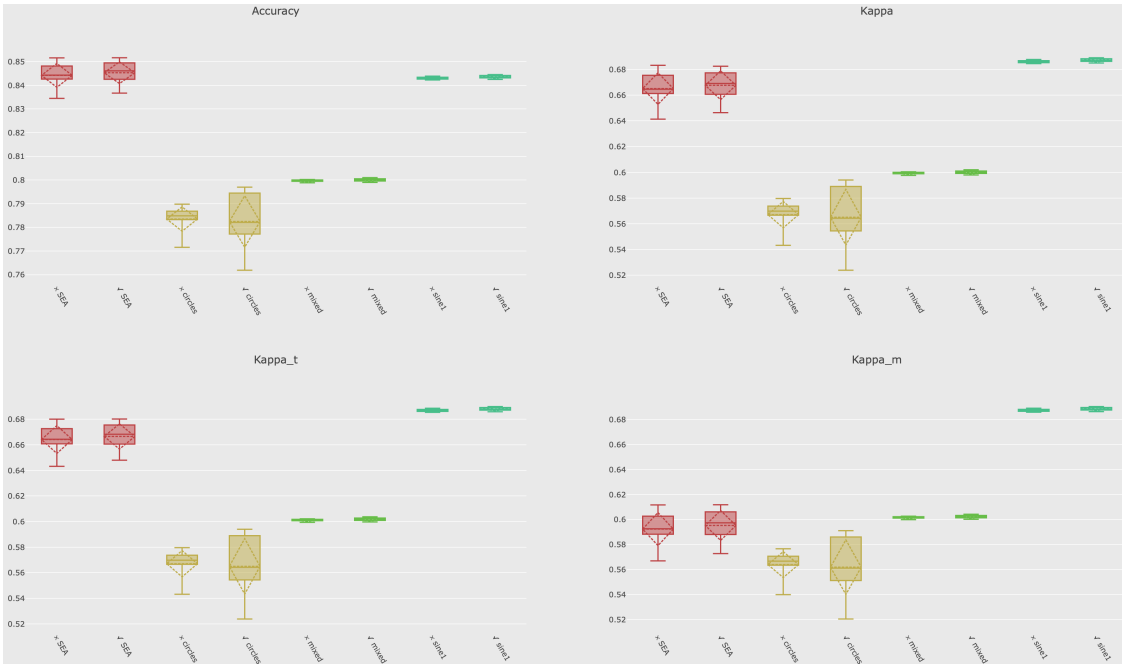

FIGURE 3.5: Boxplots showing performance over 4 datasets (using un/weighted probabilities for drift detection)

Figure 3.5 illustrates the results of experimentation in table 3.5 through the use of boxplots. The weighted predictions cause performance to increase ever so slightly. However, in the case of the *CIRCLES* dataset, the unweighted probabilities seem to perform better than the weighted ones.

Our modified drift detectors using the probabilities will be called Modified FHDDM/S. Algorithm 7 shows the implementation of MFHDDM. No changes were necessary to implement MFHDDM/S with the different boolean, since this is similar to the current version.

3.6 A toy example

In this section, we explain how an instance from the stream is processed from beginning to end. The reader should refer to algorithm 8 and figures 3.6 and 3.1.

Algorithm 7: Modified Fast Hoeffding Drift Detection Method (MFHDDM)

```
function init(window_size, delta, use_probability)
    (n, δ, p) = (window_size, delta, use_probability);
    ε_d = sqrt((1/2n) ln(1/δ));
    reset();
function reset()
    w=[];
    μ^m = 0;
function detect(p)
    if w.size = n then
        w.tail.drop();
    w.push(p);
    if w.size < n then
        return False;
    else
        if use_probability then
            μ^t = w.average();
        else
            μ^t = w.count(True) / w.size();
        if μ^m < μ^t then
            μ^m = μ^t;
        Δμ = μ^m − μ^t;
        if Δμ ≥ ε_d then
            reset();
            return True;
        else
            return False;
```

The first step in the algorithm is to pre-train the algorithm so that it can start with a**n** adequate model of the data. Therefore a parameter is used to dictate the number of instances that the ensemble is trained on before starting the interleaved test-then-train loop.

Next, the interleaved test-then-train loop begins. For this toy example, we use a batch size of one. This batch (or chunk) contains a single instance, which is referred to as X, and its true class value is referred to as y.

The ensemble is first tasked with predicting the class value, denoted $\hat{y}$, of X. In order to do so, the ensemble requires each classifier it contains to assign a probability that X belongs to a class, for each possible value that the class can take. We use binary classification as example, therefore $y \in \{0, 1\}$. So in our example, each classifier needs to predict $P(X, y = 0)$, and $P(X, y = 1)$ given its model of the data. The ensemble then keeps a copy of these predictions and applies a weighting function to the original values. The weighting function, as previously stated, reduces values in $]0, 0.5[$ and increases values in $]0.5, 1[$. Finally, the ensemble averages the probabilities for each class class value (again, for binary classification). It continues in the same manner for the weighted probabilities. The ensemble now has four values: $p_0, p_1 = \frac{1}{n}\sum_{i=1}^{n} P_i(X, y = Y)\ \forall\ Y$, $w_0, w_1 = \frac{1}{n}\sum_{i=1}^{n} w(P_i(X, y = Y))\ \forall\ Y$, where n is the number of classifiers in the ensemble, $w(x)$ is the weighting function seen in section 3.2, $P_i(X, y = 0)$ is the probability that classifier i assigns X to class 0. Y is the set of possible class values. The maximum of p_0 and p_1 determines the class the ensemble predicts for X. If p_0 is the maximum value, then $\hat{y} = 0$, and in the opposite case where p_1 is the maximum value then $\hat{y} = 1$.

The next step in the interleaved test-then-train loop is dedicated to drift detection. In the case where a drift is detected, a sliding window keeps the previously seen instances for retraining the classifiers. The size of this window is the same as the size of the window used for the training step, therefore it is size three in our example. MFHDDMS is used to detect drifts, when its sliding window it full. The average probability for the winning class, is passed to the drift detector. There are two sliding windows: one short to detect abrupt drifts, and a longer one to detect gradual concept drifts. The sliding window, therefore, keeps track, as discussed in section 3.2, of either probabilities or booleans. In the case of the

probabilities, the Hoeffding bound is used to detect if the average probability drifts too far from the maximum seen average probability. In the case of the boolean values, a drift is detected, using the Hoeffding bound again, when the classifier in question predicts less often on average according to the "winning" vote. When a drift is detected, the drift detector's windows are emptied, the classifiers in the ensemble are either all reset or a subset of them are reset. If a drift is detected, the sliding window of previously seen instances is then used to retrain all of the classifiers in the ensemble.

If no drift is detected, there is a chance (set by a parameter) that the y value (the real class value of X) will be replaced by $\hat{y}$ (the predicted class value of X). This serves as the self-training step, where the parameter would force X to be unlabelled.

The final step in the interleaved test-then-train loop is to train the ensemble on the $< X, y >$ tuple. In order to do so, a hybrid sliding-tumbling window (as seen in section 3.3) is used within the ensemble to train the classifiers contained within it. In this case, since $< X, y >$ is the first instance seen by the ensemble during the interleaved test-then-train loop (as in this is the first iteration of the prequential evaluation loop) it is added to the hybrid window. This hybrid window has a size equal to the batch size (in this case, one) times the number of classifiers in the ensemble (three). Only one classifier in the ensemble is trained on the hybrid window's tuples. In order to do so, an index is kept inside the ensemble. This index points to the classifier to train and is incremented at each $ensemble.train()$ call, and a modulus is applied on the index to keep it within the range of classifiers in the ensemble. Thus, classifier c_0 is selected to train on the hybrid window, containing a single tuple at the moment.

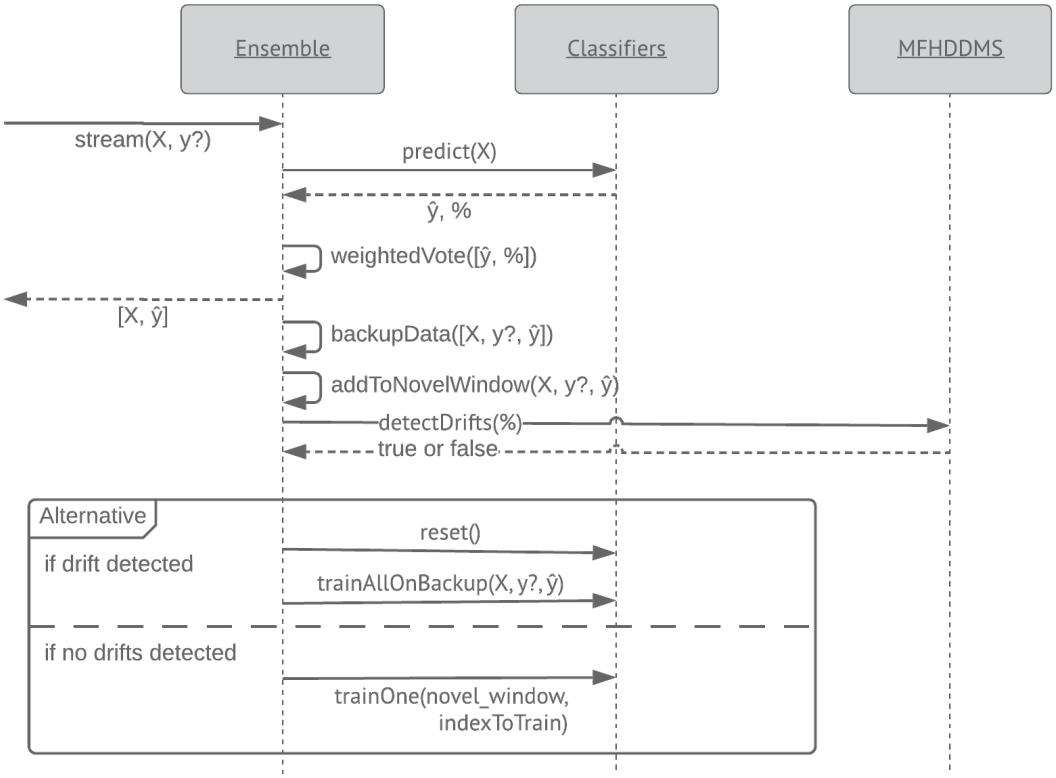

FIGURE 3.6: Simplified Sequence Diagram illustrating algorithm 8

At this point, the first iteration of the interleaved test-then-train loop has terminated, and the framework computes the global metrics required (accuracy, kappa statistics, etc.) as well as over a sliding window of the last 200 instances.

3.7 Conclusion

In this chapter, we have introduced the components of our LESS-TWE methodology. The first contribution was to propose a new weighting function for our ensemble classifier. The second was a novel windowing technique for classifiers within an ensemble that delays their training. The third was the non-selective self-training on a portion of the training data (for which the label was ignored).

Algorithm 8: Data processing pipeline

```
function evaluate_prequential(ensemble, pretrain_size, batch_size, window_size)
    ensemble.train(pretrain_size);
    window = fifo_queue(size = window_size);
    while stream.has_more_samples() do
        X, y = stream.next_sample(batch_size);
        window.add([X,y]);
        prediction, probabilities, weighted_probabilities = ensemble.test(X);
        drift_detected = drift_detection(prediction or probabilities or
          weighted_probabilities);
        if random() > ground_truth_percentage and not drift_detected then
            replace_y(y, prediction);
        if drift_detected then
            ensemble.reset(window);
        ensemble.train(X, y);
function ensemble.train(X, y)
    // instance variables clfs stores the classifiers in ensemble &
       mod stores the current index of the classifier to train
    ensemble.clfs[ensemble.mod].train(X, y);
    ensemble.mod = (ensemble.mod + 1) modulus (length(ensemble.clfs));
function ensemble.test(X)
    p, w_p = [], [];
    for classifier ∈ ensemble.classifiers do
        prob = classifier.predict();
        p.add(prob), w_p.add([weight(prob[0]), weight(prob[1]));
    // map subclassifier probabilities to single probability
    avg, w_avg = p.avg(), w_p.avg();
    prediction = class_of_max(w_avg);
    return prediction, avg, w_avg;
function weight(prediction_probability)
    return (1 − tanh(3.5 − 7 × prediction_probability)) / 2;
function drift_detection(value, use_probability)
    window.push(value);
    if window.size > max_window_size then
        window.tail.drop();
        if use_probability then
            μ^t = window.average();
        else
            μ^t = window.count(True) / window.size();
        if μ^m < μ^t then
            μ^m = μ^t;
        Δμ = μ^m − μ^t;
        if Δμ ≥ ϵ_d then
            return True and reset();
    return False;
```

The fourth and final contribution was to extend FHDDM/S to operate without any knowledge of the ground truth. We described the components of our methodology and their algorithms, and illustrated the framework by means of a toy example. Next, we explain our experimental evaluation.

Chapter 4

Experimental Design

In this chapter, we describe our experimental design.

The data sets used for our analysis in the upcoming chapter are SEA [90], CIRCLES, SINE1 and MIXED [50], which all containing noise, and either abrupt or gradual concept drifts.

The estimation technique we use is prequential evaluation, also known as interleaved test-then-train, which consists of infinitely executing a loop where a classifier first predicts labels for new data (without its label), then adapts its model for said data, with the correct label.

The performance measures that we use are the execution time, measured in seconds, as well as the κ-temporal statistic to evaluate a classifier's predictive performance, also called κ^+ or κ_t. This κ statistic compares our classifier to a no-change classifier and takes into account temporal dependence in the data.

Using the mean values for the entire stream for both of the metrics mentioned above, we use statistical tests to determine whether or not the differences observed are statistically significant and not due to simple coincidence. When comparing two classifiers across multiple data sets, we use the Wilcoxon test,

and when comparing more than two classifiers, we use the Friedman test, coupled with the post-hoc Nemenyi test.

We conduct the following experiments:

- examine the impact of each parameter value on the mean of each metric,
- rank the results of each parameter combination in order to establish any trend regarding parameter values across the metrics,
- compare the top ranking parameter combinations to the state of the art.

4.1 Software and Hardware specifications

In order to ensure that our experiments are reproducible, we specify the specifications of the hardware and software used to run them. All experiments were run on a MacBook Pro model *11,4*. This machine was running macOS 10.14.4 on a quad-core Intel i7 2.2GHz processor (**3.4GHz** TurboBoost, 8 virtual cores), with a 256 GB SSD, and **16GB of RAM**. During the course of the experiments, the laptop was plugged in and fully charged.

Python 3.7.3 was installed, with the following dependencies:

- *sortedcontainers* v2.0.5
- *numpy* v1.15.3
- *scipy* v1.1.0
- *scikit-learn* v0.20.0
- *pandas* v0.23.4
- *mlxtend* v0.13.0

- *scikit-multiflow* (latest commit from master rebased onto our branch) ***#f18a433***

4.2 Scikit-multiflow

Scikit-multiflow [60] is a Python framework, that complements scikit-learn, an offline, batch learning Python library. As it currently stands, scikit-multiflow implements stream generators, machine learning algorithms, drift detection algorithms and evaluation methods for data streams. The authors describe scikit-multiflow as "an open-source framework for multi-output / multi-label and stream data mining", and takes its inspiration from MOA [11] and MEKA [76]. The former is the most popular data stream mining framework, implemented in Java, which includes a collection of machine learning algorithms, and evaluation tools. The latter is another project implemented in Java, but for multi-label learning and evaluation.

We selected this framework as it is backed by Bifet, implemented in Python: a popular language among data scientists, open-source, and very recent.

4.3 Data sets

The four data sets that we select to conduct our experiments on are well-known benchmarks often used for evaluating algorithms in evolving streams [2, 5, 38, 63, 64]. Lu, in a review on concept drift, states that "[as] instances are generated using predefined rules [...], synthetic data sets [are] a good option for evaluating the performance of learning algorithms in different concept drift scenarios" [56, pp. 12-13].

Real-world streaming data sets can present real concept drift, or drift that is synthetically generated. In the case of the latter, they have been referred to as semi-real-world streaming data sets [70, p. 42]. While real data sets can provide realistic benchmarking for different drift detection algorithms, some researchers have put forward that the location and/or presence of concept drift for real-world streaming data sets are unknown [5, 12, 31, 42]. As such, real-world data sets make it difficult for algorithms to "understand the drift, and could introduce bias when comparing different machine learning models" [56, p. 13]. Further, as real-world data is time-consuming to correctly label, streaming data sets that contain real data are minuscule compared to synthetic data sets. It is not uncommon for real data sets to contain as little as five to ten thousand instances, which is a far cry from being realistic. As Bifet states in [12], "demonstrating systems only on small amounts of data does not build a convincing case for capacity to solve more demanding data stream applications".

Therefore, we opt to conduct our experiments on the widely-used synthetic benchmarking data sets described below. Having said that, further research should be conducted to investigate the performance of this framework on real, or semi-real streaming data sets.

4.3.1 Synthetic Data Sets

For our experiments, four synthetic data sets are used, three of which ($CIRCLES$, $SINE1$, and $MIXED$) were generated by my colleague for his paper [72] where

TABLE 4.1: *CIRCLES* data set concepts

center	(0.2, 0.5)	(0.4, 0.5)	(0.6, 0.5)	(0.8, 0.5)
radius	0.15	0.2	0.25	0.3

he proposed FHDDMS, the drift detection algorithm. These data sets were generated with ten percent (10%) class noise[1] to test the robustness of his drift detection algorithm against noisy data streams. The data sets contain one hundred thousand rows belonging to one of two classes, and present either gradual or abrupt concept drift. To generate concept drift, Bifet's approach is used, where a sigmoid function defines "the probability that every new instance of the stream belongs to the new concept after the drift" [12]. A parameter, W, indicates the length of the concept drift, starting at the point of change t_0. Pesaranghader sets $W = 500$ for generating gradual concept drifts, and $W = 50$ for abrupt drifts. While the specific data sets we use were generated by Pesaranghader, the genesis of these synthetic data sets can be traced back to [50] and were further used in, among others, the following papers [2, 5, 38, 63, 64].

CIRCLES

As stated by Gama et al. in [38], this data set is composed of two relevant numerical attributes: x and y, which are uniformly distributed in [0, 1]. There are four different concepts in this data set, each representing whether or not a point is within a circle given x, and $y >$ coordinates for its centre and its radius r_c. This data set contains gradual concept drifts that occur at every twenty-five thousand (25 000) instances. The four pairs of $< (x, y), r_c >$ defining each concept are given in table 4.1.

[1] To generate class-noise, the correct label of an instance is replaced by another (incorrect) label.

SINE1

As stated by Gama et al. in [38], this data set contains abrupt concept drifts, with noise-free examples. It has only two relevant numerical attributes, for which the values are uniformly distributed in [0, 1]. Before the concept drift, all instances for values below the curve $y = sin(x)$ are classified as **positive**. Then, after the concept drift, the rule is reversed; therefore the values below the curve become **negative**. The drifts were generated at every twenty thousand (20 000) instances.

MIXED

As stated by Gama et al. in [38], this data set contains abrupt concept drifts and uses four relevant attributes. Two of which are boolean, let them be v and w; and the other two attributes are numerical, in [0, 1]. Instances belong to the positive class if two of three conditions are met: v is true, w is true, $y < 0.5 + 0.3 \times sin(3\pi x)$. For each concept drift, the conditions are reversed, meaning that if the conditions are met, it will be a positive instance, then after the drift, it will be a negative instance. The abrupt concept drifts occur at every twenty thousand (20 000) instances.

Streaming Ensemble Algorithm (SEA) generator

First described in [90] by Street and Kim, the Streaming Ensemble Algorithm (SEA) generates streams with abrupt concept drift. It is composed of three numerical attributes of values in [0, 10], and only the first two attributes are relevant. For each instance, the class is determined by checking if the sum of the two relevant attributes passes a threshold value. Let f_1 and f_2 be the two numerical relevant attributes, and θ the threshold. An instance belongs to class *one* if

$f_1 + f_2 \leq \theta$. As in Street's paper, our stream has four concepts, with the threshold values for each being 8, 9, 7 and 9.5. We generate streams of one hundred thousand instances, from zero to twenty percent noise, in ten percent increments ($\{0; 10; 20\%\}$). Drifts, therefore, occur at every twenty-five thousand (25 000) instances, and the parameter dictating the rate of change of the concept drift is set to $W = 0$, meaning that the concept changes abruptly from one instance to the next.

4.4 Estimation techniques

In an offline setting, with static data, the most common evaluation method used is called cross-validation. However, given how little time a model has to learn from each instance due to the velocity of data streams and the risk of concept drift, cross-validation is not suited for an online setting. The two following techniques are, though.

4.4.1 Holdout

Two distinct data sets are required for this technique: a training data set to train a model, and another to test it. Cross-validation is typically used in offline static data mining, but is too computationally heavy and/or can be too time-consuming in a streaming setting, and therefore the validation step is skipped, and performance is measured against a single holdout set [7].

This technique is most useful when the training and testing sets have already been defined, as it makes comparing results from different studies possible. In

order to track performance, the model is evaluated periodically, but not too frequently to avoid negatively impacting performance.

The holdout data set can be sourced from new tuples arriving from the data stream, and can also be added later to the training set in order to optimise instance use.

It should be sufficient to safely use a single static holdout set if we make the assumption that there is no concept drift [7].

4.4.2 Interleaved test-then-train, or prequential

Another evaluation method is prequential evaluation, also known as the interleaved test-then-train method. The evaluation method consists of first testing the classifier on a given set of instances, then training the classifier on that same set. The evaluation strategy, therefore, ensures that the model has not previously seen testing tuples, and no holdout testing set is necessary, and the accuracy of the model is therefore incrementally updated. This method also allows us to use all of the data for both testing and training. As more data is tested than in the holdout method, each instance used to assess the accuracy and performance of the model weighs less than it would have in a smaller holdout test set [7].

All experiments are done using the interleaved test-then-train evaluation technique.

TABLE 4.2: Confusion Matrix

	Actual Positive	Actual Negative
Predicted Positive	TP	FN
Predicted Negative	FP	TN

4.5 Performance measures

4.5.1 Accuracy & Confusion Matrix

Let us assume that we are dealing with a binary classification problem (with two classes) with a **P**ositive and **N**egative class.

A confusion matrix keeps track of the correctness of our classifying model by using the following four measurements [44, pp. 77-79].

- False Positive (FP): number of instances incorrectly classified as positive
- False Negative (FN): number of instances incorrectly classified as negative
- True Positive (TP): number of instances correctly classified as positive
- True Negative (TN): number of instances correctly classified as negative

It should then be clear that all of the positive class instances (P) are in the combined groups of FN and TP, and that all of the negative class instances (N) are in the remaining FP and TN groups.

A confusion matrix helps us to visualise these measurements by presenting them in a matrix (two-by-two grid in the case of a binary classification problem). Each column represents all of the instances for an actual class, whereas each row represents the instances predicted for a given class. Table 4.2 shows a confusion matrix.

Given our four measurements above, we can calculate the following metrics:

- The *sensitivity* and *specificity* (or *recall*) indicate the completeness for a given class of instances retrieved that belong to that class [44, p. 96]; in other words the percentage of correctly classified instances of a given class that were found overall.

$$\frac{TP}{TP+FN} = \frac{TP}{P} \text{ and } \frac{TN}{TN+FP} = \frac{TN}{N} \tag{4.1}$$

- The *accuracy*, while not a reliable metric, indicates the number of correct predictions over all cases to be predicted [44, p. 86].

$$sensitivity \times \frac{P}{P+N} \times specificity \times \frac{N}{P+N} = \frac{TP+TN}{P+N} \tag{4.2}$$

- *Precision* indicates the exactness of predictions for a given class [44, p. 99]. In other words, the percentage of instances the classifier predicted as positive that were actually positive,

$$\frac{TP}{TP+FP} \tag{4.3}$$

- *F-measure*, which is the harmonic mean of precision and recall [44, p. 103].

$$2 \times \frac{precision \times recall}{precision + recall} \tag{4.4}$$

4.5.2 Kappa (κ) statistics

κ statistic

Bifet and Frank argue in [4] that prequential accuracy is not well suited for classifying unbalanced data in a stream whereas the Kappa (κ) statistic proposed by Cohen in [19] is better adapted when dealing with changing class distributions. They therefore proposed a sliding-window kappa as defined by equation 4.5

$$\kappa = \frac{p_0 - p_c}{1 - p_c} \tag{4.5}$$

where p_0 is the classifier's prequential accuracy and p_c is the probability that a chance (or no-change) classifier makes a correct prediction. Refer to [4] for the equations used to calculate p_0 and p_c.

Values of κ are in [0, 1]. If the classifier always classifies instances correctly, then $p_0 = 1$ and $\kappa = 1$; if the classifier correctly classifies instances as often as the chance classifier, then $p_0 = p_c$ and $\kappa = 0$; and if the chance classifier is always right, then $p_c = 1$ and $\kappa = 1$.

κ^+ statistic

The κ^+ statistic is an improvement upon the regular κ statistic that takes into account temporal dependence in order to better evaluate the performance of a given classifier. When there is no temporal dependence in the data and that classes are balanced, then κ^+ is equal to κ. While we could solely rely on κ^+, the authors however recommend using both κ and κ^+ statistics in order to obtain a

more thorough evaluation. κ^+ is defined in equation 4.6

$$\kappa^+ = \frac{p_0 - p'_e}{1 - p'_e} \tag{4.6}$$

where p_0 is the classifier's prequential accuracy and p'_e is the accuracy of a no-change classifier. The no-change classifier predicts that the next class label will be the same as the last seen class label [8]. Just like the original κ statistic, $\kappa^+ \in [0, 1]$. Refer to [13] for the equations relating to calculating p_0 and p'_e. κ^+ is also sometimes referred to as κ-temporal (κ_t) [104].

κ_m statistic

The κ_m statistic is another improvement upon the regular to indicate whether a classifier is performing better than a majority-class classifier [8]. Its definition is given in equation 4.7.

$$\kappa_m = \frac{p_0 - p_m}{1 - p_m} \tag{4.7}$$

where p_0 is the classifier's prequential accuracy and p_m is the prequential accuracy of a majority-class classifier. If the classifier always predicts correctly, then $\kappa_m = 1$. If the classifier predicts correctly as often as the majority-class one, then $\kappa_m = 0$.

4.5.3 Testing for Statistical Significance

While the measures presented in 4.5 are useful for evaluating the performance of our classifiers, it is insufficient to rely solely on them to fully evaluate performance differences between classifiers. Statistical significance tests are used

to determine whether or not the differences that were observed are statistically significant and not due to simple coincidence [44].

Popular choices for statistical tests in the machine learning community for comparing multiple classifiers across multiple cases are the Analysis of Variance (ANOVA) [28] and the Friedman test [33].

The ANOVA test assumes a normal data distribution whereas the Friedman test does not. It is for this reason, coupled with the fact that the Friedman test is also non-parametric, that we have chosen the latter as our choice for a statistical significance test. However, the Friedman test is not recommended when only comparing two algorithms on multiple domains [29, p. 355]; for that reason, we will be using the Wilcoxon test [96] in those scenarios.

The Friedman test

The test ranks each algorithm separately for each data set and computes a test statistic using the variation within the ranks and the variation within the error variation. The Friedman statistic χ_F^2 is defined in equation 4.8, taken from [44];

$$\chi_F^2 = \frac{SS_{Total}}{SS_{Error}} = \left[\frac{12}{n \times k \times (k+1)} \times \sum_{j=1}^{k} (R.j)^2\right] - 3 \times n \times (k+1) \tag{4.8}$$

where n is the number of data sets, k is the number of algorithms considered, and R simply denotes ranking. And finally $(k-1)$ is known as the degrees of freedom.

The Friedman statistic (χ_F^2) is then looked up in the χ^2 distribution table to obtain a *probability value* (p-value), signifying $P(\chi_{k-1}^2 \geq \chi_F^2)$. This p-value is widely used in null-hypothesis testing by measuring it against a threshold value called

a significance level. A lower p-value means a higher statistical significance. Traditionally, the significance level is either 0.1 or 0.5, and denoted as α. The null hypothesis is accepted if the p-value is greater than α; otherwise, H_1 is accepted, and the null hypothesis is rejected.

When the null hypothesis is rejected following a Friedman test, post hoc Nemenyi tests are recommended by Japkowicz and Shah in [44], as well as Flach in [29, p. 355].

Wilcoxon's Signed-Rank test

This test is a non-parametric version of the matched-paired t-test. It is used to test if two paired samples come from the same distribution, by measuring the difference in their mean ranks.

The test procedure is as follows [44, pp. 233-235], [29, p. 354]:

N will be set as the sample size, meaning that there are $2N$ data points, and since data are paired, $[x_{1,i}, x_{2,i}]$ represent the measurements for pairs where $i \in [1; N]$.

We test the following null hypothesis.

H_0: difference between the pairs follows a symmetric distribution around zero
H_1: does not follow a symmetric distribution around zero

- Calculate $d_i = (x_{1,i} - x_{2,i}) \forall i \in [1; N]$.
- Pairs for which the difference is zero are excluded, let N_r be the reduced sample size.
- Rank the N_r pairs by their absolute difference in increasing order, let R_i denote rank. In the case of ties, assign the pairs with the average of their ranks.

- Two sum of ranks are then calculated:

  $W_{s1} = \sum_{i=1}^{N_r} I(d_i > 0)\,\text{rank}(d_i)$,

  $W_{s2} = \sum_{i=1}^{N_r} I(d_i < 0)\,\text{rank}(d_i)$.

- Calculate statistic $T_{wilcox} = \min(W_{s1}, W_{s2})$.

- For smaller values of N, which is the case in our thesis, look up tabulated critical values of T (according to N and statistical significance level) and reject the null hypothesis if the statistic is inferior to the tabulated value.

Post-hoc test: the Nemenyi test

The Nemenyi test [61] is used in order to determine which algorithms compared in the Friedman test actually differ. It computes a statistic, q, as defined in equation 4.9 for a pair of classifiers f_{j1} and f_{j2}. q gives us the difference between ranks of the two algorithms, and can be also expressed as a critical difference (CD) [29, p. 356]. In order compare the rankings, the Nemenyi test computes the mean rank for a given classifier using equation 4.10 where R_{ij} is the rank of classifier f_j on data set S_i [44, pp. 256-257].

$$\overline{R}_{.j} = \frac{1}{n}\sum_{i=1}^{n} R_{ij} \tag{4.9}$$

$$q = \frac{\overline{R}_{.j1} - \overline{R}_{.j2}}{\sqrt{\frac{k(k+1)}{6n}}} \tag{4.10}$$

4.6 Experimental Setup

For these experiments, the following classifiers were used: Gaussian Naive Bayes (G_NB), Leveraging Bagging, Stochastic Gradient Descent (SGD), Multinomial

Naive Bayes (M_NB), and our Voting Ensemble composed of three sub-classifiers. The three classifiers inside the voting ensemble are the SGD, M_NB, and G_NB. Finally, we also used a no-change classifier as well as a majority-class classifier as our baselines.

Readers familiar with classification algorithms [93] might know that some advantages of NB are its scalability, high accuracy, ability to use prior knowledge, and the fact that it has comparable performance to decision trees and neural networks. NB is incrementally more confident, and easy to implement. However, the downsides are its significant compute costs, and that using conditional dependencies reduces accuracy due to the real dependency between attributes. SGDs have generally high predictive accuracy, generalise well meaning that they are robust and suitable for noisy domains, and good for incremental learning. However, they are known to have a slow training time, may fail to converge, and output a black box model which means that results cannot be interpreted. Leveraging bagging [6] is an improvement over the Online Bagging technique of Oza and Russel. They introduce more randomisation to the online bagging algorithm to achieve a more accurate classifier. However, the authors noted that subagging, half subagging, and bagging with-out replacement ran faster but were marginally less accurate.

Unless stated otherwise, our default parameters for the experiments are listed in table 4.3 and the default parameters for the classifiers are listed in table 4.4. These values were obtained through extensive experimentation.

Finally, each experiment is run on five different examples each sourced from three synthetic data sets (5 examples of $SINE1$, another 5 of $CIRCLES$, etc.) and three streams generated by the SEA generator with levels of noise in increments of ten percent (from 0% to 20%), for a grand total of eighteen (18) streams of one

TABLE 4.3: Default parameters for the experiments

Parameter	Value
Classifier	Ensemble Voting Classifier
Pretrain size	1 000
Max samples	100 000
Batch size	25
Window size	75
Window type	sliding tumbling hybrid
Ground truth %	100
Drift reset type	none

hundred thousand (100000) instances.

Our voting ensemble takes seven (7) different parameters, some of which are conditional upon the value of others. The parameters and the values they can take are shown in table 4.5. Some combinations of parameters are not allowed such as combining boolean voting with non-boolean drift detector content, or using probability voting with weighted probability drift content. In the worst case, when counting the illegal combinations, there are $4 \times 2 \times 3 \times 5 \times 4 \times 3 \times 2 = 1880$ combinations. There are 1110 permitted parameter combinations.

We will run our algorithm for each permitted combination of the parameters stated above, and will measure κ_t and the execution time for each of the eighteen data sets. We will then evaluate how each parameter affects the execution time and κ_t.

We will test the null hypothesis that all values of a given parameter will perform similarly in regards to a given measure when we set all other parameters. We will repeat this experiment over all combinations of the "other" parameters.

For parameters that have only two values, we will use the Wilcoxon test to determine if the values of the parameters lead to statistically significant differences in the metrics measured. We will also use the internal intermediate results of

TABLE 4.4: Our default parameters for all classifiers

Classifier	Parameters
Leveraging Bagging	base_estimator=HoeffdingTree() n_estimators=10 w=6 delta=0.002 leverage_algorithm=leveraging_bag random_state=None
Hoeffding Tree	max_byte_size=33.5MB split_criterion=information gain split_confidence=0.0000001 binary_split=False remove_poor_atts=False no_preprune=False leaf_prediction=Naive Bayes Adaptive
Gaussian NB	priors=None var_smoothing=1e-9
Multinomial NB	alpha=1.0 fit_prior=True class_prior=None
Stochastic Gradient Descent	loss=log penalty=l2 alpha=0.0001 l1_ratio=0.15 fit_intercept=True max_iter=1000 tol=1e-3 shuffle=True epsilon=0.1 n_jobs=-1 random_state=None learning_rate=optimal eta0=0.0 power_t=0.5 early_stopping=False validation_fraction=0.1 n_iter_no_change=5 average=False n_iter=None

TABLE 4.5: Voting ensemble parameters

Parameter	**Values**
Voting type	boolean, probability, avg. w. probability, w. avg. probability
Window type	sliding, hybrid
Chunk size	25, 75, 100
Ground truth	60%, 70%, 80%, 90%, 100%
Drift reset type	no drift detection, blind, partial, complete
Drift content	boolean, probability, weighted probability
Drift detector count	1 or many

the Wilcoxon test to rank the parameter values, again depending on the metric measured.

Otherwise, for parameters that can take more than two values, we will use the Friedman test; and when the Friedman test rejects the null hypothesis, we will use a post-hoc Nemenyi test to determine which pairs of values lead to statistically significant differences in the measured metrics.

We will be comparing the values of each parameter across all permitted parameter combinations. We will then be able to assess if a parameter value is often ranked better and if it seems to influence the measures even when changing the "other" set parameters. In other words, we aim to determine if a parameter is likely to frequently affect our measured metrics no matter of the other parameters.

After we assess the impact of the parameter values on the measured metrics, we will use the ranking algorithm from the post-hoc Nemenyi test to rank each parameter combination to determine the top ranking combination for each measured metric and over both metrics.

Once we obtain the parameter combinations that lead to the best results, we will compare them to the state of the art algorithm (leveraging bagging). We will, also, be comparing these results with no-change and majority-class classifiers

(trained with 100% labelled examples and sliding windows). The leveraging bagging classifier is implemented with a built-in ADWIN drift detector. As we will be comparing at least 3 algorithms, we will employ the Friedman test in conjunction with the post-hoc Nemenyi test where appropriate to determine which pairs of algorithms differ.

In summary, we:

1. compare each value of one parameter (while setting other parameters to a given value), multiple times and each time changing the other parameter combination to assess the impact of that one parameter value on the measured metrics
2. rank the results of each parameter combination
3. use those rankings to compare the top parameter combinations to the state of the art

In the following chapter, we present the results of our experiments, analyse these findings and discuss their significance.

Chapter 5

Experimental Evaluation and Discussion

Recall from the previous chapter that in order to evaluate our contributions, we measure how our algorithm performs over four different data sets. We evaluate our ensemble using the κ_t metric, as well as the execution time. Finally, we also consider the percentage of labelled data used to train our ensemble. In this chapter, we investigate how each parameter influences each measured metric using Wilcoxon and Friedman tests, and in the case of the latter, post-hoc Nemenyi tests to further confirm which pairs of algorithms differ in performance. Next, we will compare all of the parameter combinations together by ranking them by each metric. This is done to find any trends that lead to better performance. Finally, we will compare our approach to the state of the art.

TABLE 5.1: Statistically significant percentage of parameter combinations found via the Wilcoxon test

Parameter	Measure	0.05	0.01	0.001	Total
Drift detector count	execution time	5%	6%	82%	93%
	κ_t	4%	1%	1%	7%
Window type	execution time	1%	6%	89%	97%
	κ_t	7%	8%	44%	60%

Figure 5.1 shows that as the execution time increases, κ_t stays roughly constant. The majority of the changes seem to be due to the various parameter combinations. This is a good sign as it suggests that the predictive accuracy of our model is at most very loosely tied to the execution time of its algorithm.

5.1 Investigating how each parameter influences each metric

5.1.1 Wilcoxon tests

Drift Detector Count

As indicated by table 5.1, the parameter *Drift Detector Count* seems to heavily influence the execution time of the algorithm, regardless of the other parameter values. When it comes to the κ_t metric, only 7% of combinations proved to have statistically significant differences in predictive performance.

When we dig deeper into the *Drift Detector Count* parameter, according to table 5.2 and figure 5.2, for the execution time metric, the parameter value *1 for ensemble* is evidently the best choice as it ranks best across 92% of parameter combinations, and best across 99% of statistically significant different results.

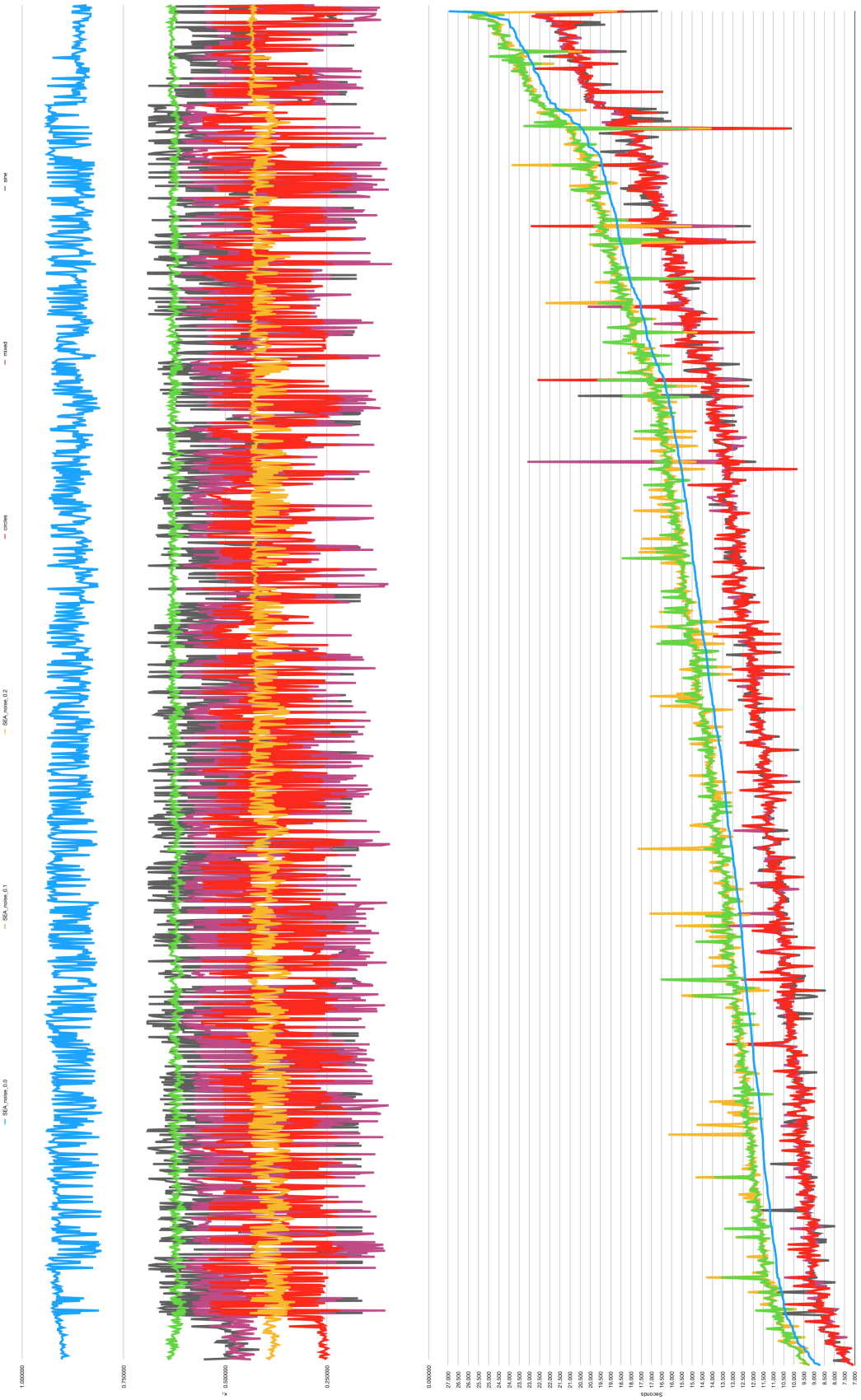

FIGURE 5.1: κ_t in relation to time, across all parameter combinations

TABLE 5.2: Statistically significant percentage of parameter combinations by parameter value for Drift Detector Count found via the Wilcoxon test

Measure	**1 for ensemble**				**1 per classifier**			
	significant		**insignificant**		**significant**		**insignificant**	
	count	**%**	**count**	**%**	**count**	**%**	**count**	**%**
execution time	447	99%	22	73%	4	0%	8	26%
κ_t	19	59%	206	48%	13	40%	217	51%

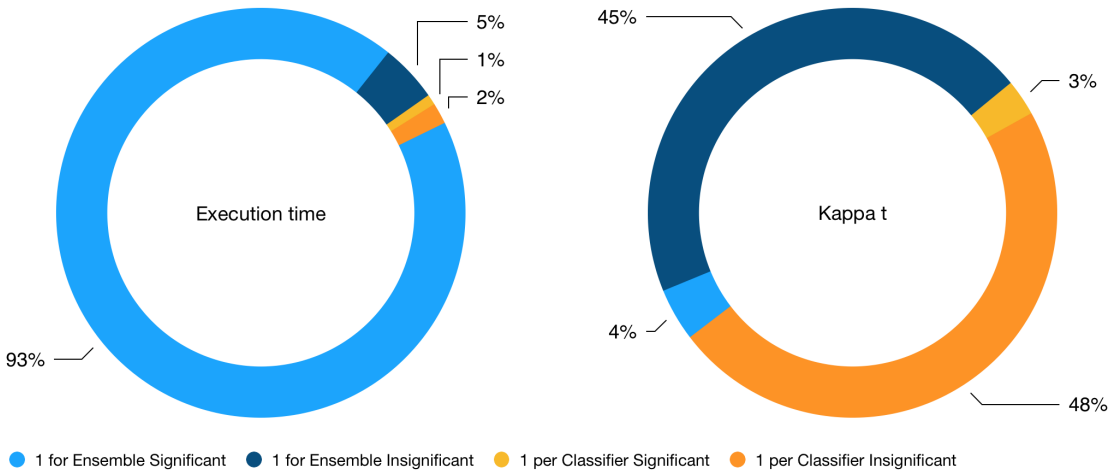

FIGURE 5.2: Pie chart illustrating table 5.1

For the κ_t metric, it is not as clear cut; both *1 for ensemble* and *1 per classifier* rank best among about 50% of the time.

The results we found lead us to believe that choosing the *1 for ensemble* value for the *Drift Detector Count* parameter is very beneficial in reducing the execution time of the algorithm. While this finding might not hold across other data streams, it is likely that choosing this value for the parameter will decrease execution time significantly. Logically, it makes perfect sense that choosing *1 for ensemble* over *1 per classifier* leads to lower execution time because the implementation of the former is such that it performs only a fraction of the operations of the latter. Additionally, we also found that none of the parameter values consistently outranked the others in predictive accuracy, and we, therefore, cannot say if choosing a particular parameter value will result in better values for the κ_t metric. We will later be considering the raw values to determine if a particular drift detector count leads to better predictive accuracy.

Window Type

As for the *Window Type* parameter, 97% of parameter combinations show a significant statistical difference in the execution time depending on the value of the window type. This indicates that the parameter value heavily influences the execution time of the algorithm, independently of other parameter values. When it comes to the κ_t metric, over half (60%) of combinations proved to show a statistically significant difference in predictive performance. This suggests that the *Window Type* parameter could have some non-negligible influence over the κ_t metric.

TABLE 5.3: Statistically significant percentage of parameter combinations by parameter value for Window Type found via the Wilcoxon test

Measure	**Hybrid**				**Sliding**			
	significant		**insignificant**		**significant**		**insignificant**	
	count	**%**	**count**	**%**	**count**	**%**	**count**	**%**
execution time	514	99%	6	54%	5	0%	5	45%
κ_t	9	2%	86	40%	319	97%	125	59%

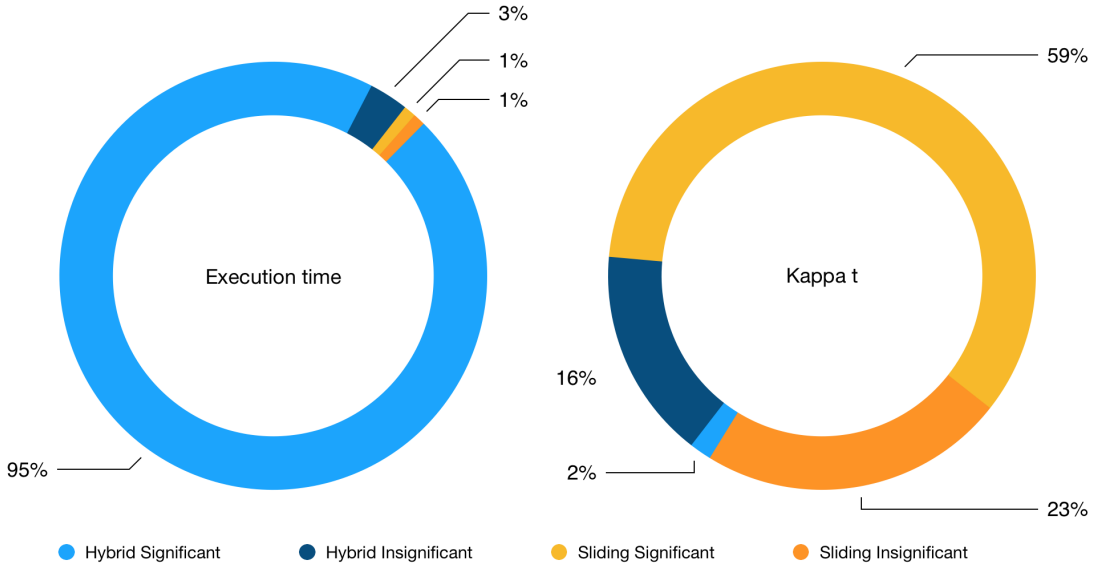

FIGURE 5.3: Pie chart illustrating table 5.3

When we dig deeper into the *Window Type* parameter, as table 5.3 and figure 5.3 illustrates, for the execution time metric, *hybrid* is evidently the best choice as it ranks best across 98% of parameter combinations. For the κ_t metric, it is just as obvious as *sliding* ranks best across about 82% of the parameter combinations.

The *Window Type* parameter ranking results suggest that *hybrid windowing* has a significant impact on the execution time of the algorithm, independently of other parameters values. This suggests that choosing the *hybrid windowing* value for the *Window Type* parameter could be very beneficial in reducing the execution time of the algorithm. Again, this is exactly as expected: the implementation of *hybrid windowing* is such that each sub-classifier in the ensemble only trains on each instance once, whereas the *sliding windowing* technique is implemented such that each sub-classifier in the ensemble trains on each instance at least once. This, logically, leads to fewer operations, and therefore a reduction in execution time for the algorithm. This should hold true across all data streams, and therefore we recommend anyone who chooses to run this algorithm with the intent of reducing the execution time to choose the *hybrid windowing* parameter value. However, we also found that *sliding windowing* outperforms *hybrid windowing* across most parameter combinations, with or without a significant statistical difference. Again, using the explanation above for the reduction in execution time, since each sub-classifier training using *sliding windowing* trains on each instance multiple times, it is logical that it is better able to fit the data instance in its model. This means that the algorithm presents a trade-off between execution time and predictive accuracy when considering the *Window Type* parameter. Whoever runs the algorithm must choose which metric they value more, and choose a windowing type accordingly.

As other parameters can take more than two values, we must use the Friedman

TABLE 5.4: Percentage of parameter combinations that showed statistically significant differences from the post-hoc Nemenyi test

Parameter	Measure	Significant	%	Insignificant	%
Batch size	κ_t	$\frac{1}{330}$	0.3%	$\frac{229}{330}$	99.70%
	execution time	$\frac{287}{330}$	86.97%	$\frac{43}{330}$	13.03%
Drift reset type	κ_t	$\frac{1}{330}$	0.30%	$\frac{329}{330}$	99.70%
	execution time	$\frac{64}{330}$	19.39%	$\frac{266}{330}$	80.61%
Ground truth	κ_t	$\frac{57}{198}$	28.79%	$\frac{141}{198}$	71.21%
	execution time	$\frac{168}{198}$	84.85%	$\frac{30}{198}$	15.15%
Voting type	κ_t	$\frac{0}{180}$	0%	$\frac{180}{180}$	100.00%
	execution time	$\frac{154}{180}$	85.56%	$\frac{26}{180}$	14.44%

TABLE 5.5: Breakdown of unique statistically significant different rankings of parameter combinations from table 5.4

Parameter	Measure	Significant	Insignificant
Batch size	κ_t	$1 \rightarrow 1$	$229 \rightarrow 6$
	execution time	$287 \rightarrow 2$	$43 \rightarrow 2$
Drift reset type	κ_t	$1 \rightarrow 1$	$229 \rightarrow 6$
	execution time	$64 \rightarrow 6$	$266 \rightarrow 6$
Ground truth	κ_t	$57 \rightarrow 3$	$141 \rightarrow 6$
	execution time	$168 \rightarrow 22$	$30 \rightarrow 16$
Voting type	κ_t	$0 \rightarrow 0$	$180 \rightarrow 4$
	execution time	$154 \rightarrow 2$	$26 \rightarrow 3$

test in combination with the post-hoc Nemenyi test in order to test our null hypotheses.

5.1.2 Post-hoc Nemenyi tests

Batch Size

As we can observe from table 5.4, only one combination of parameters among 330 were significantly statistically different when considering the κ_t metric for the batch size parameter, which is a negligible amount. As for the execution

TABLE 5.6: Rankings for batch size and parameter combination counts

Metric	Stat. Sig.	Ranks	Stat. Sig. values	%
κ_t	✓	25 / 75 / 100	25 / 100	0.30%
	×	100 / 25 / 75		16.36%
		100 / 75 / 25		10.30%
		25 / 100 / 75		16.67%
		25 / 75 / 100		27.58%
		75 / 100 / 25		10.00%
		75 / 25 / 100		18.79%
Execution time	✓	100 / 75 / 25	100 / 25	71.52%
		75 / 100 / 25	75 / 25	15.45%
	×	100 / 75 / 25		6.67%
		75 / 100 / 25		6.36%

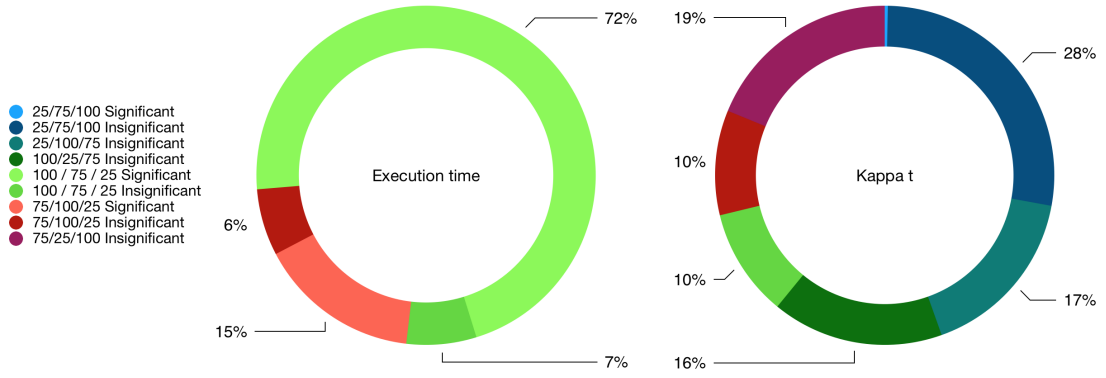

FIGURE 5.4: Pie chart illustrating table 5.6

time metric, almost 87% of parameter combinations proved to show a significant statistical difference. This confirms our expectations that changing only the batch size has a significant impact on the execution time of the algorithm but no significant impact on its predictive accuracy. The first row of table 5.5 indicates that there are seven (one plus six) unique batch size rankings for the κ_t metric; the second line indicates that there are four unique rankings for execution time: two unique rankings of parameters that were significantly statistically different and two unique rankings for batch sizes that were insignificant. These values are illustrated in table 5.6 and figure 5.4.

Let us examine the pie charts illustrating the rankings of the *Batch size* parameter values. We will first examine the results for the execution time. At first glance, we notice that *100* clearly ranks first over the majority of parameter combinations, followed by *75* over a minority of combinations. For the κ_t metric, the rankings are less clear. *25* seems to rank first more frequently (across 45% of parameter combinations), whereas *75* and *100* rank first across 26% of parameter combinations each. These rankings do not tell us how much the difference is across parameter combinations. By this, we mean that while a parameter value may rank first across a larger percentage of parameter combinations, it may actually perform worse than another value over a smaller percentage of parameter combinations. In any case, the results indicate that there is another trade-off to be made in regards to the batch size parameter between predictive accuracy and execution time. Again, we expected these results as a batch size of 100 allows the ensemble to learn from more examples at once, which reduces the number of operations that would have to be repeated were we to use a batch size of 25 for example, where each operation down the line would be run 4 times instead of only 1 time. As for predictive accuracy, *25* has a slight advantage over the other parameter values possibly because there is a better chance that a drift wreaks havoc on a sub-classifier's ability to train on a batch of *75* or *100* than on a batch of *25*. For a batch of *25*, the sub-classifiers can be reset sooner, and the next batch may be easier to model.

Drift Reset Type

Let us now investigate the Drift Reset Type parameter. In table 5.4, only a single out of 330 parameter combinations showed a significant statistical difference

in parameter values when considering the κ_t metric, whereas about 20% of parameter values showed a statistically significant difference in parameter values when considering the execution time metric. This suggests that the value of the drift reset type parameter does not have a significant impact over other values on the measured metrics. Table 5.5, lists the number of different rankings found in table 5.7 over the measured metrics, depending on whether results showed a statistical significant difference.

Let us examine table 5.7 or better yet, the pie charts shown in figure 5.5 illustrating the rankings of the *Drift Reset Type* parameter values. Again, we will first examine the results for the execution time. At first glance, we notice that *blind resets* clearly ranks first the least over the parameter combinations. *Reset all* ranks first across half of the parameter combinations and *partial resets* over about a third of them. This is a good sign, and expected. Since sub-classifiers must be reset more frequently, they must be completely re-trained on data, which also prevents them from learning from a larger set of older data. This might be good for streams with very frequent drifts, but those where they infrequently appear, blindly resetting the classifier prevents it from remembering possibly useful historical data.

For the κ_t metric, the rankings are not as clear. Again, we must remember that these rankings do not tell us the difference across parameter combinations, meaning that while a parameter value may rank first across a more significant percentage of parameter combinations, it may perform less well than another value over a smaller percentage of parameter combinations. In any case, the results indicate that each parameter value ranks first across a third of parameter combinations. This does not allow us to conclude much, other than blind resets may perform just as well as resetting every sub-classifier or only a minority

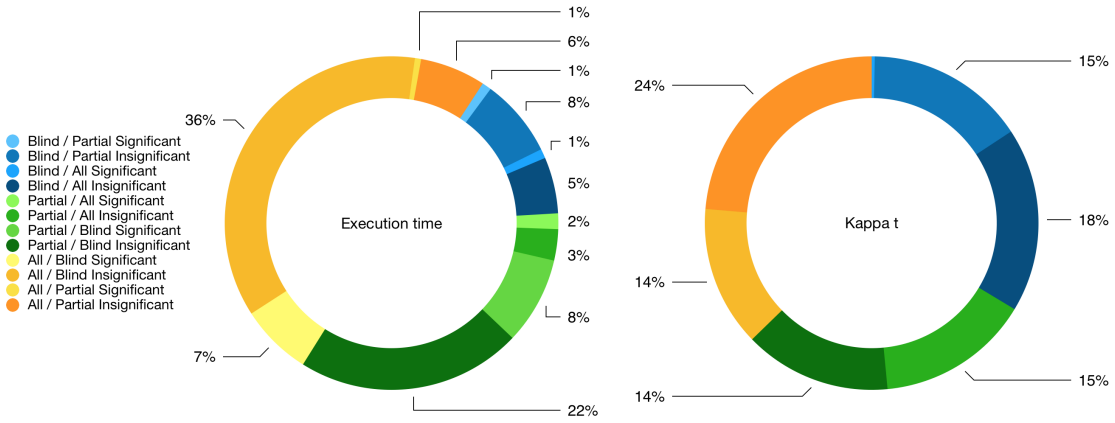

FIGURE 5.5: Pie chart illustrating table 5.7

of them. It could be that the sub-classifiers are not very apt at learning from the data, or that resetting the sub-classifiers is not very useful. Another reason could be attributed to the way partial drift resets work, in that each sub-classifier has a chance of being reset, meaning that there is a chance that the offending sub-classifier that is not correctly adapting to the concept drift is not reset. A more thorough investigation is unfortunately outside of the scope of this work, due to time constraints. It could also be that the predictive accuracy cannot improve much after a drift due to the limitations of the sub-classifiers to model the data effectively. Either way, this is truly unfortunate, as it suggests that changing how often and how many sub-classifiers are reset does not seem to affect the κ_t metric. These results indicate that blind resets perform almost as well as our other reset strategies, which may suggest that our drift reset strategy needs further investigation.

Ground Truth

Let us now investigate the Ground Truth parameter (which indicates the percentage of labelled instances used to train our ensemble). Table 5.4 shows that

TABLE 5.7: Rankings for drift reset type and parameter combination counts

Metric	**Stat. Sig.**	**Ranks**	**Stat. Sig. values**	**%**
κ_t	✓	BLIND / ALL / PARTIAL	BLIND / PARTIAL	0.30%
	×	ALL / BLIND / PARTIAL		23.64%
		ALL / PARTIAL / BLIND		13.64%
		BLIND / ALL / PARTIAL		15.45%
		BLIND / PARTIAL / ALL		17.88%
		PARTIAL / ALL / BLIND		14.24%
		PARTIAL / BLIND / ALL		14.85%
Execution time	✓	ALL / PARTIAL / BLIND	ALL / BLIND	6.97%
		ALL / BLIND / PARTIAL	ALL / PARTIAL	0.61%
		BLIND / PARTIAL / ALL	BLIND / ALL	0.91%
		BLIND / ALL / PARTIAL	BLIND / PARTIAL	0.91%
		PARTIAL / BLIND / ALL	PARTIAL / ALL	1.52%
		PARTIAL / ALL / BLIND	PARTIAL / BLIND	8.48%
	×	ALL / BLIND / PARTIAL		6.36%
		ALL / PARTIAL / BLIND		36.36%
		BLIND / ALL / PARTIAL		7.58%
		BLIND / PARTIAL / ALL		5.45%
		PARTIAL / ALL / BLIND		21.82%
		PARTIAL / BLIND / ALL		3.03%

less than one-third of parameter combinations show a statistically significant difference for the κ_t metric; but over 84% for the execution time metric. This suggests that the value of the ground truth parameter has a measurable impact on the execution time, across a large portion of the other parameters, (almost) no matter their values.

Table 5.8 shows that, for the κ_t metric, there are multiple possible options for ranking the parameter values. For those with a statistically significant difference, 100 performs better than 60. 100 ranks first across all parameter combinations over 98% of the time, as expected. This is normal, as we expect our ensemble to better learn to model the streamed data from ground truth rather than from its own predictions of class label values, considering that it can enable our ensemble to learn from erroneously labelled data. It is a good sign, however, that there is no statistically significant difference for parameter values other than 100 and 60. It is logical that selecting a higher percentage of ground truth will lead to higher predictive accuracy, but we can allow for a trade-off between predictive accuracy and time.

In table 5.8, there are different combinations for ranks and statistical significant different values for the execution time metric. Combinations with 60% ground truth rank first across 74.31% of parameter combinations, and 100% ground truth ranks first across only 17.71% of parameter values. When we exclude insignificant results, our previous observation remains true, but across a smaller percentage of parameter combinations, but only by 5% to 10%. We can conclude that using 60% ground truth is more likely to decrease execution time. This is an unexpected finding, as we expected that using less ground truth would cause an increase in execution time by causing more drift detection events and therefore more model resets and retrains. However, it is possible that by using less

ground truth, there are no drift detection events because the model is not able to properly learn from the data, so the model is never reset or retrained which might also explain why this parameter value also ranks as one of the worst in predictive accuracy.

When looking at the raw values, as depicted in figure 5.6, we notice that the κ_t metric does not change much over the parameter combinations, but rises noticeably starting when using 80% ground truth. The regular peaks and valleys within each set of ground truth values (separated by black vertical lines) represent how other parameters affect κ_t. We also notice that the amount of ground truth has a strong effect over the sine1, mixed and particularly on the circles data sets. It may be that it is easier to model the data from SEA than from the other data sets, and therefore it does not need as much ground truth to build an accurate model. Not much can be said when looking at the raw values for the execution time metric other than that there may be fewer valleys in the section dedicated for the 60% ground truth parameter.

Voting Type

Finally, we investigate the Voting Type parameter. As table 5.4 shows, none of the 180 parameter combinations proved to show a significant statistical difference for the κ_t metric. However, a large portion of the parameter combinations showed a significant statistical significance for the execution time parameter (85.56% of combinations). This leads us to believe that the voting type does not have a significant influence on predictive accuracy. However, for execution time, there is a statistically significant difference present across a good majority of parameter combinations, leading us to believe that the significance remains true across a good portion of other parameter values.

TABLE 5.8: Rankings for ground truth and parameter combination counts

Metric	**Stat. Sig.**	**Ranks**	**Stat. Sig. values**	**%**
κ_t	✓	100 / x / x / x / 60	100 / 60	28.80%
	×	100 / x / x / x / 60		65.15%
		100 / x / x / 60 / 70		4.55%
		80 / 100 / 90 / 70 / 60		1.52%
Execution time	✓	100 / 60 / x / x / 70	100 / 70	9.10%
		100 / 60 / 90 / 80 / 70	100 / 70 60 / 70	0.51%
		100 / 60 / 90 / 70 / 80	100 / 80	2.02%
		100 / 60 / 80 / 70 / 90	100 / 90	0.51%
		60 / x / x / x / 100	60 / 100	14.16%
		60 / x / x / x / 70	60 / 70	1.02%
		60 / x / x / x / 80	60 / 80	50.51%
		60 / 70 / 100 / 80 / 90	60 / 80 60 / 90	0.51%
		60 / 80 / 70 / 100 / 90	60 / 90	1.01%
		70 / 90 / 80 / 100 / 60	70 / 60 90 / 60	0.51%
		90 / x / x / x / 100	90 / 100	2.03%
		90 / 70 / 80 / 100 / 60	90 / 60	3.03%
	×	100 / 60 / x / x / 90		1.02%
		100 / 60 / x / x / 70		1.52%
		100 / 60 / 90 / 70 / 80		3.03%
		60 / x / x / 80 / 90		1.52%
		60 / x / x / x / 80		3.55%
		60 / 100 / 90 / 80 / 70		0.51%
		60 / x / x / 80 / 100		1.52%
		90 / x / x / x / 100		2.03%
		90 / 70 / 80 / 100 / 60		0.51%

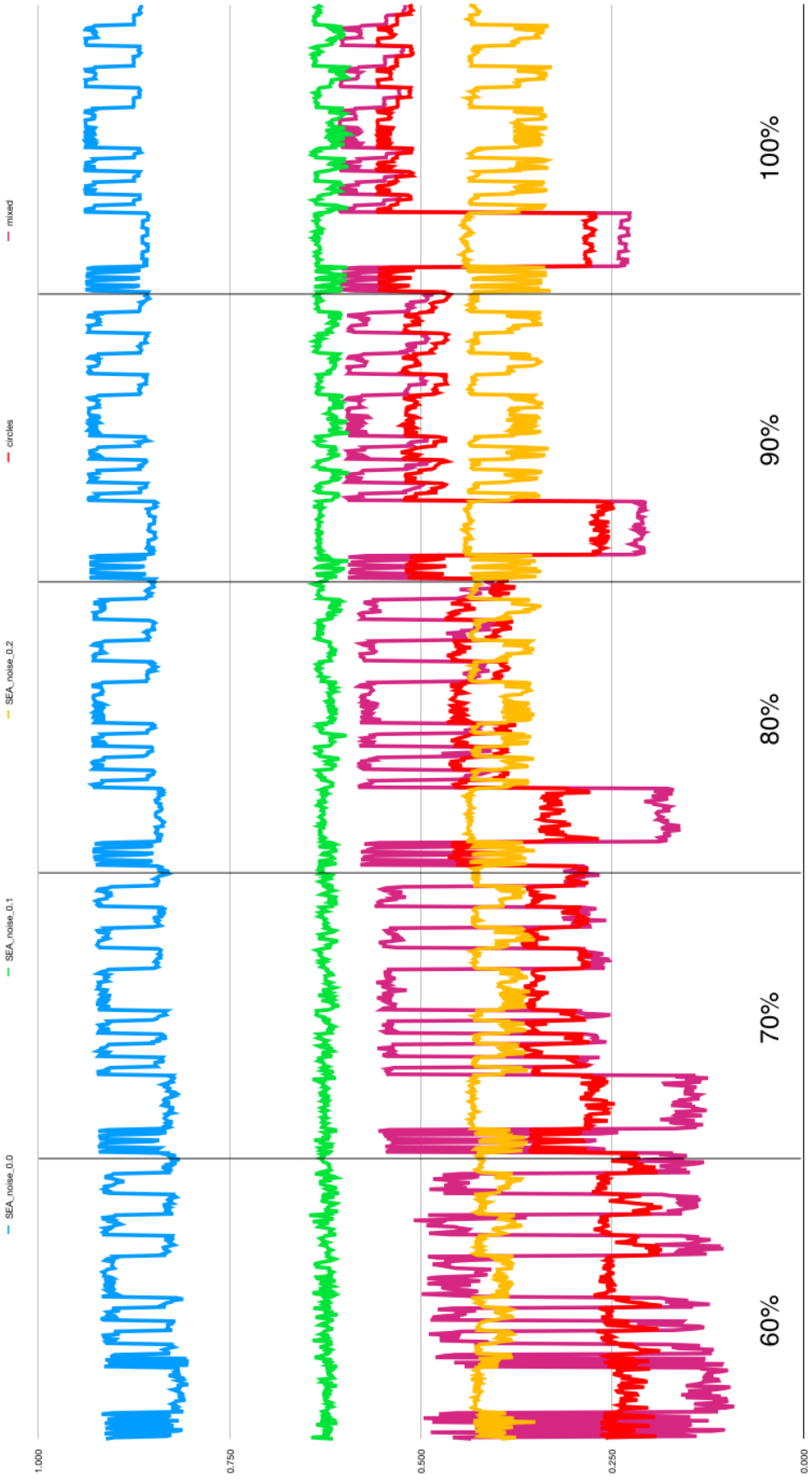

FIGURE 5.6: κ_t across all parameter combinations, ordered by ground truth

According to table 5.5, there are not very many different possible rankings for both metrics. Indeed, there are four possible rankings for κ_t and 5 for the execution time metric.

Let us now examine table 5.9, or better yet the pie charts from figure 5.7 for the *Voting Type* parameter. We will first examine the results for the execution time. At first glance, it is clear that *probability voting* ranks first amongst the majority of parameter combinations (90% of them), while *weighted averaged probability voting* ranks first amongst the remaining 10% of parameter combinations. We can say with the utmost certainty that *averaged weighted probability voting* is not a good choice for a parameter value if the goal is to keep execution time low, as it ranks last over 99% of parameter combinations. It makes sense that probability voting ranks first in execution time over most parameter combinations as its implementation is such that the two other voting schemes add to the operations computed for probability voting. In other words, for the other two voting schemes, the results obtained through probability voting is an intermediate result.

For the κ_t metric, *probability voting* ranks first across about 69% of parameter combinations, and *weighted averaged probability voting* ranking first across the remaining fraction of parameter combinations. Probability voting appears to rank better than the other two voting schemes, but this could be due to the noise that we add when we apply the weights to the prediction probabilities. Furthermore, the ranks do not allow us to determine by how much the voting scheme impacts the actual metrics. Then, to answer why *weighted averaged probability voting* performs better than *averaged weighted probability voting*, it could be that there is less noise introduced by first averaging the prediction probabilities then applying the weight, than performing those operations in the opposite way.

When we consider at the raw results from our simulations, and order them by

TABLE 5.9: Rankings for voting type and parameter combination counts

Metric	Stat. Sig.	Ranks	Stat. Sig. values	%
κ_t	×	PROBA / AVG W / W AVG		1.11%
		PROBA / W AVG / AVG W		67.78%
		W AVG / AVG W / PROBA		1.11%
		W AVG / PROBA / AVG W		30.00%
Execution time	✓	PROBA / W AVG / AVG W	PROBA / AVG W	78.33%
		W AVG / PROBA / AVG W	W AVG / AVG W	7.22%
	×	PROBA / AVG W / W AVG		0.56%
		PROBA / W AVG / AVG W		11.11%
		W AVG / PROBA / AVG W		2.78%

voting type (then by ground truth), as seen in figure 5.8, we can notice that there is no significant difference in the κ_t values between probability voting and weighted averaged probability voting. However, we can notice a significant difference between probability voting and averaged weighted probability voting.

The best choice here appears to be *probability voting*, but we will be able to better determine the veracity of this statement when ranking all parameter combinations and comparing their raw metric values in the next section. It does not matter whether execution time or predictive accuracy matters more, probability voting is more likely to better model the data, predict new instances all the while taking the least amount of time to do so, as opposed to the other parameter values.

5.1.3 Summary

The above results suggest that it is preferable to use the following parameter combination to obtain higher κ_t values: [*sliding*, *probability*, *25*, 100%, *all*, *1e*]. Otherwise, to minimise the execution time, the results suggest to use [*hybrid*, *probability*, *100*, 60%, *all*, *1e / 1c*].

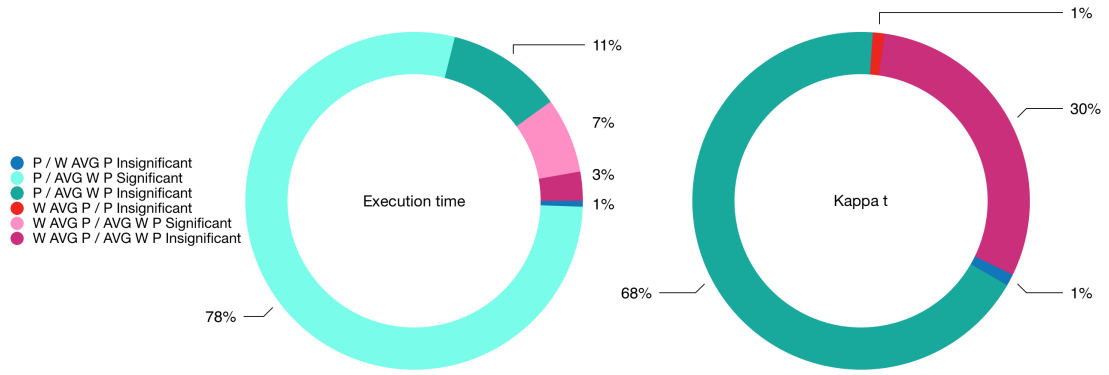

FIGURE 5.7: Pie chart illustrating table 5.9

The differences lie with the window type, the batch size, the ground truth used and partially the drift detector count. The results above indicated that using probability voting, one detector per ensemble to detect drifts, and resetting all classifiers when drifts occur would lead to better κ_t values and a lower execution time. For the batch size, window type, and the drift detector count, it is entirely logical that choosing one value over another would change the execution time as they were, at least partially, implemented as time-saving measures.

In the following section, we will rank the parameter combinations to determine if the ones listed two paragraphs above are indeed top ranking.

5.2 Comparing all parameter combinations

In this section, we aim to rank all 1110 parameter combinations to determine which parameters perform the best, and those that perform the worst. Recall that in the previous section, we looked at how parameter values performed individually, and this section investigates how they interact with each other. This will be done in three different configurations:

1. over κ_t

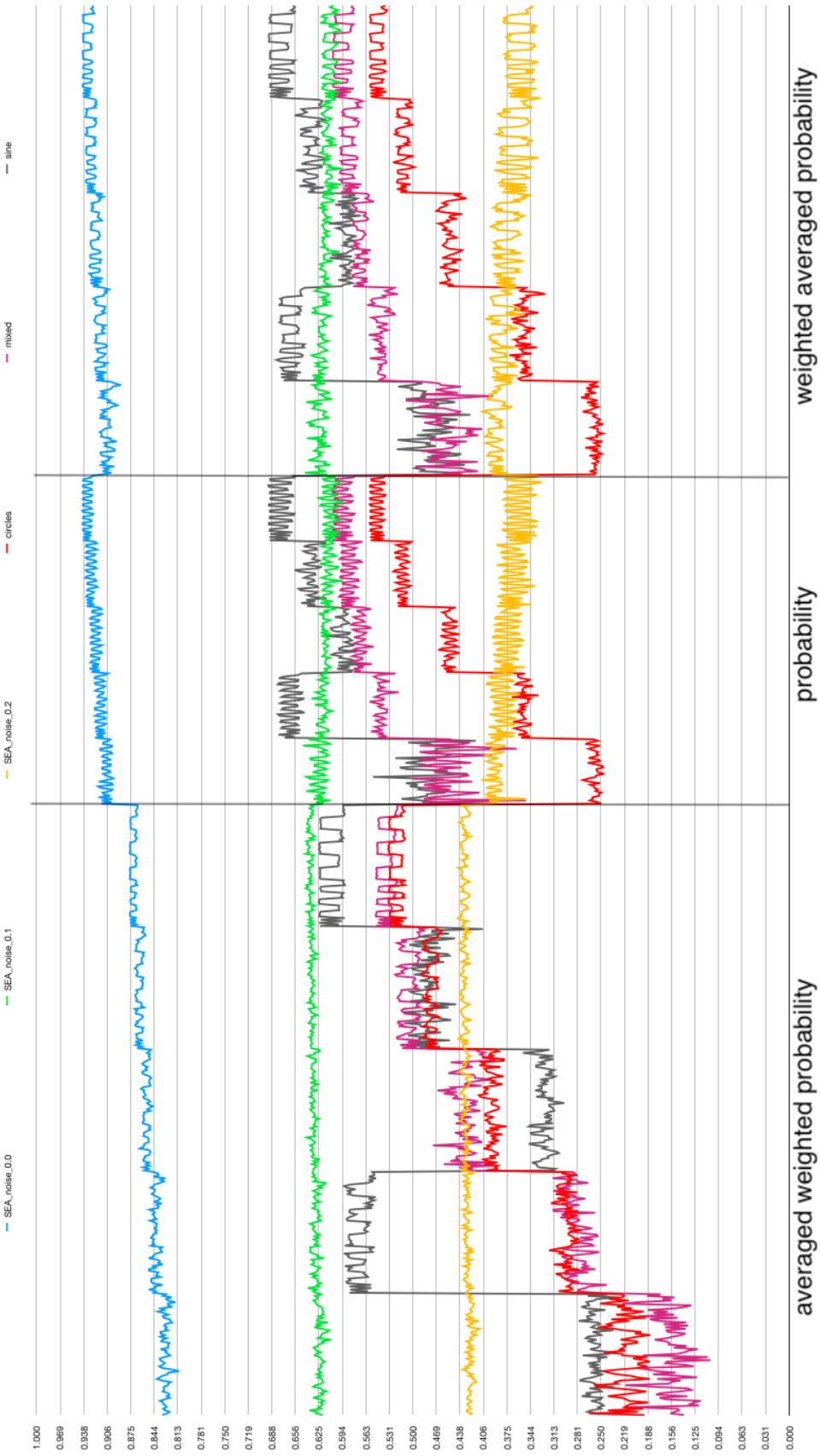

FIGURE 5.8: κ_t across all parameter combinations, ordered by voting type

TABLE 5.10: Mapping shortened parameter values with full name

Parameter	**Value**	**Real Value**
Window type	s	Sliding
	h	Hybrid
Voting type	aw	Averaged weighted probability
	b	Boolean
	p	Probability
	wa	Weighted averaged probability
Drift reset type	a	All
	b	Blind
	n	None
	p	Partial
Detector count	1c	1 per classifier
	1e	1 for ensemble
Detector content	b	Boolean
	p	Probability
	wp	Weighted probability

2. over execution time

3. over both metrics

Due to the sheer number of parameter combinations ranked, in contrast to the few data sets used to compare them, we cannot rely on statistical significance tests. We can, however, use the algorithms in these tests to rank the combinations. We use the Nemenyi test's ranking algorithm to find the top ranking parameter combinations for our algorithm.

Parameter values are shortened to fit in tables, and their values are split by vertical lines. See table 5.10 for the mapping between shortened and real parameter values. The order of the parameters in the figures are the following: [*window type, voting type, ground truth, batch size, drift reset type, detector count, detector content*].

TABLE 5.11: Top and Bottom Ten ranked parameter combinations for κ_t

Top Ten							**Bottom Ten**						
s	aw	100	25	a	1c	p	h	b	60	75	b	1e	b
s	aw	100	75	a	1e	p	h	b	60	75	p	1e	b
s	aw	100	75	a	1c	p	s	b	60	75	b	1e	b
s	aw	100	100	a	1e	wp	h	aw	60	75	p	1e	wp
s	aw	100	25	b	1e	wp	h	b	60	100	a	1e	b
s	aw	100	75	b	1c	wp	s	b	60	100	p	1c	b
s	aw	100	75	p	1e	wp	h	aw	60	75	a	1c	wp
s	aw	100	100	b	1e	p	s	aw	60	75	b	1e	wp
s	aw	100	100	a	1e	p	s	b	60	25	none	none	none
s	aw	100	100	b	1c	p	s	b	60	75	none	none	none

5.2.1 Ranking over κ_t

Table 5.11[1] lists the top and bottom ten ranking parameter combinations (best and worst in decreasing order, or, in other words, the combinations in the first row are the best and the worst), and table 5.12 indicates how frequently each parameter occurred in the best 50 ranked parameter combinations for κ_t. These results suggest that parameter combinations with the following values score higher on the κ_t metric: [*sliding*, *, 100%, *75 or 25*, *partial*, *1c or 1e*, *probability*]. The asterisk is used to indicate any value for that parameter. These findings do not precisely match up with the results found in the previous section. For example, we found that probability voting was more likely to be top ranking in the previous section, whereas we found in this section that weighted averaged or averaged weighted probability voting was more frequently in the 50 top ranked parameter combinations. The same can be said for the drift reset type parameter value.

[1]Window type (sliding, hybrid), Voting type (averaged weighted, boolean, probability, weighted averaged), Ground Truth, Batch size, Drift reset type (all, blind, none, partial), Detector count (1 per classifier, 1 for ensemble), Detector content (boolean, probability, weighted probability)

TABLE 5.12: Breakdown of parameter value frequency in the top 50 ranked parameter combinations for κ_t

Parameter	**Value**	**Breakdown**
Window type	hybrid	10%
	sliding	90%
Voting type	averaged weighted	36%
	probability	26%
	weighted averaged	38%
Ground truth	90	38%
	100	62%
Batch size	25	34%
	75	38%
	100	28%
Drift reset type	all	26%
	blind	26%
	none	12%
	partial	36%
Detector count	1 per classifier	42%
	1 for ensemble	46%
	none	12%
Detector content	probability	68%
	weighted probability	20%
	none	12%

TABLE 5.13: Top and Bottom Ten ranked parameter combinations for execution time

Top Ten							**Bottom Ten**						
h	p	60	100	b	1e	p	s	aw	60	25	b	1c	p
h	p	60	100	p	1e	p	s	aw	100	25	b	1c	wp
h	wa	60	100	a	1e	wp	s	aw	60	25	b	1c	wp
h	wa	60	100	b	1e	p	s	aw	100	25	b	1c	p
h	p	60	100	a	1e	p	s	aw	60	25	b	1e	wp
h	p	60	100	none	none	none	s	aw	60	25	b	1e	p
h	wa	60	100	p	1e	p	s	aw	70	25	b	1c	b
h	wa	60	100	a	1e	p	s	aw	80	25	b	1c	b
h	p	60	75	a	1e	p	s	aw	80	25	p	1c	wp
h	wa	60	100	b	1e	wp	s	aw	100	25	b	1e	wp

5.2.2 Ranking over execution time

Table 5.13[2] lists the top and bottom ten ranking parameter combinations, and table 5.14 indicates how frequently each parameter occurred in the best 50 ranked parameter combinations for the execution time. These results suggest that parameter combinations with the following values have lower values for the execution time metric: [*hybrid, probability, 60%, 100, partial, 1c or 1e, probability*]. These results almost completely match the results from the previous section, confirming our findings.

5.2.3 Ranking over both metrics

By filtering the raw values from the simulations we ran, in combination with the rankings obtained in the two previous subsections, the best performing parameter combinations will be determined in this subsection.

[2]Window type (sliding, hybrid), Voting type (averaged weighted, boolean, probability, weighted averaged), Ground Truth, Batch size, Drift reset type (all, blind, none, partial), Detector count (1 per classifier, 1 for ensemble), Detector content (boolean, probability, weighted probability)

TABLE 5.14: Breakdown of parameter value frequency in the top 50 ranked parameter combinations for execution time

Parameter	**Value**	**Breakdown**
Window type	hybrid	78%
	sliding	22%
Voting type	averaged weighted	0%
	boolean	4%
	probability	56%
	weighted averaged	40%
Ground truth	60	80%
	70	4%
	80	4%
	90	4%
	100	8%
Batch size	25	0%
	75	42%
	100	58%
Drift reset type	all	42%
	boolean	14%
	partial	36%
	none	8%
Detector count	1 per classifier	28%
	1 for ensemble	64%
	none	8%
Detector content	boolean	4%
	probability	74%
	weighted probability	14%
	none	8%

TABLE 5.15: Data set filtering conditions

Data set	**Condition**
SEA 0% noise	$\leq$ 11 seconds
circles	$\leq$ 9.2 seconds
sine	$\leq$ 9.2 seconds
mixed	$\leq$ 9.1 seconds

Figure 5.9 shows the raw values for each data set for both measured metrics. The values are ordered by an average of the ranks for κ_t and execution time. To give more weight to the κ_t metric in the ordering, the equation for the average is the following:

$$\frac{3 \times rank_{\kappa_t} + rank_{execution\ time}}{4} \tag{5.1}$$

In order to find the right balance, we started to filter out the parameter combinations with the conditions found in table 5.15, and then filtered out those whose averaged rank was below 340. We should note that the κ_t ranks are in the range of $[228.8, 947.3]$ and that of the execution time in the range of $[1, 1108.2]$. Figure 5.10 shows the remaining parameter combinations, with their raw values for both measured metrics. These filters were selected by intuition and through extensive exploration and inspection, in order to remove very high execution times as well as poor κ_t values. We also chose to include the parameter combinations that led to the best overall κ_t average metric, and the one with the best average execution time metric.

For the best resulting predictive accuracy, [*sliding window, probability voting*, 100% *ground truth, 100 batch size, partial reset, one drift detector, probability drift content*] proved to be the best parameter combination. As for the execution time, the following combination of parameters proved to be the best: [*hybrid window, probability*, 60% *ground truth, 100 batch size, blind reset, one drift detector, probability content*].

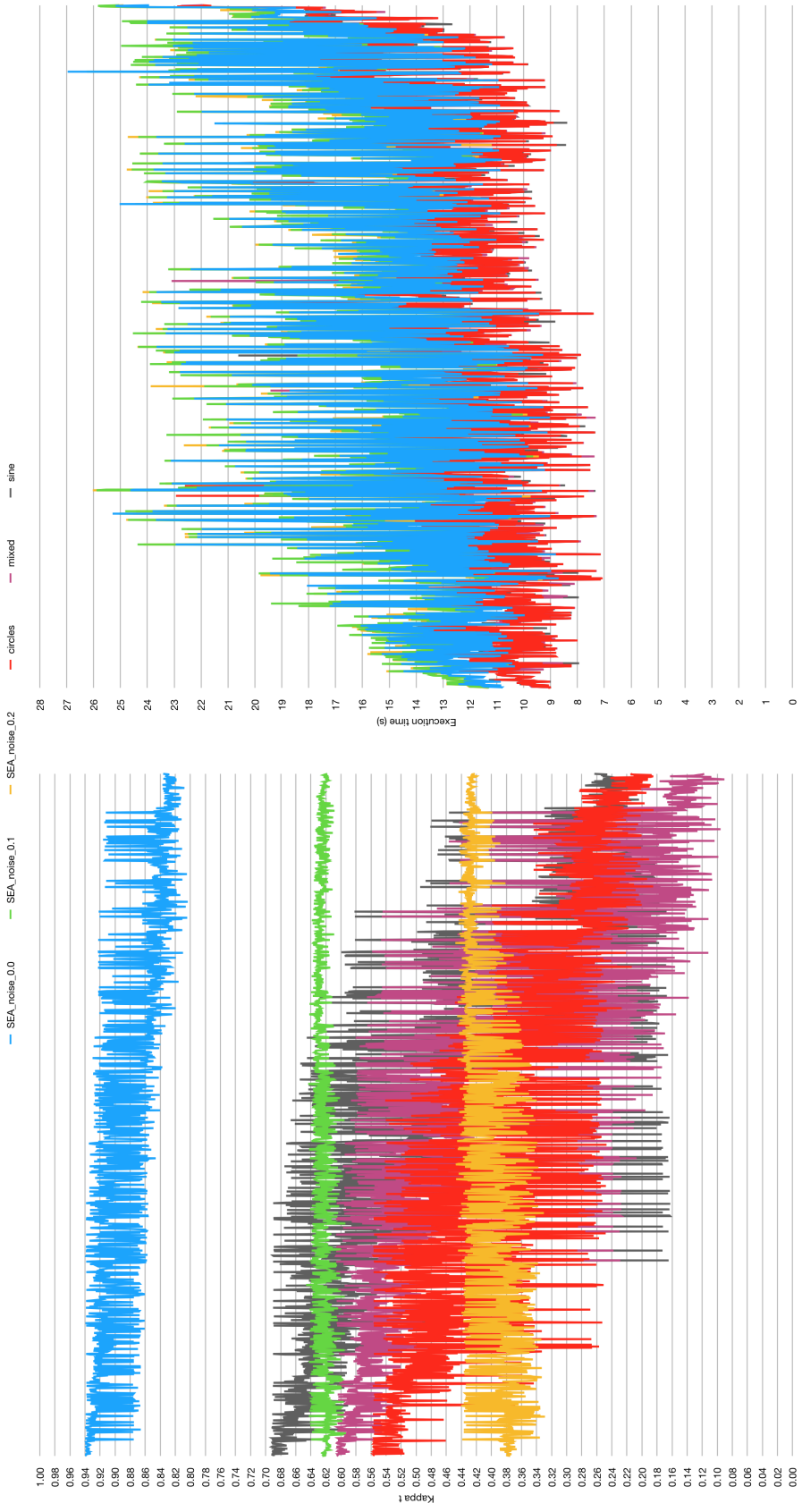

FIGURE 5.9: Raw κ_t and execution time values ordered by averaged ranks

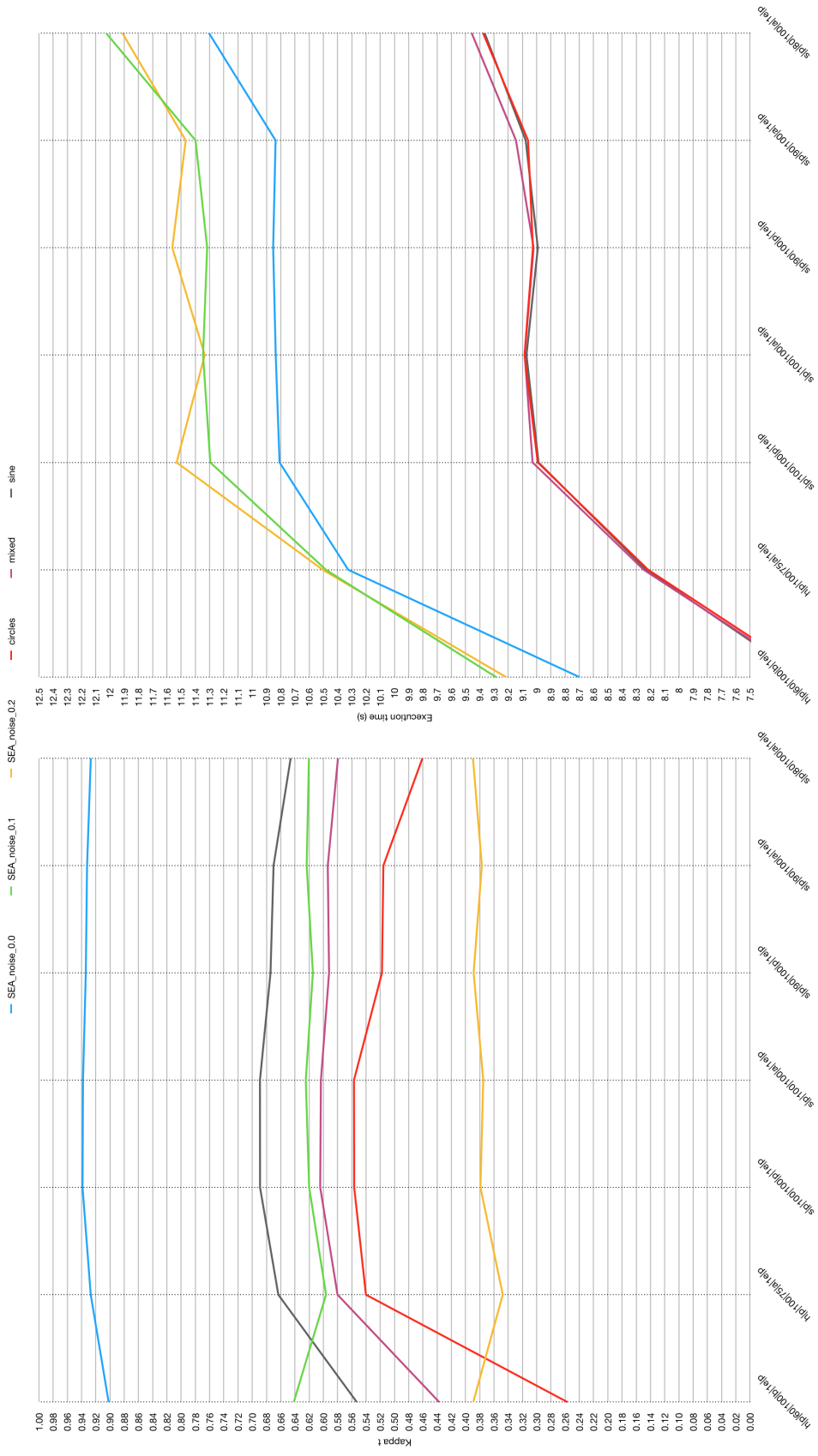

FIGURE 5.10: Remaining parameter combinations and their raw metric values after filtering

However, if we can accept a 1 to 4% reduction for κ_t values, then significant time savings can be achieved by using [*hybrid window, probability voting*, 100% *ground truth, 75 batch size, reset all, one drift detector, probability content*]. Indeed, we can reduce execution time by up to 9.5%.

5.2.4 Effects of training with less labelled data

As stated in section 3.4, we aim to determine the percentage of labelled data used at which our voting ensemble's κ_t metric declines and by how much. Figure 5.11 shows the predictive performance and execution times for parameter combinations with varying amounts of ground truth (ten percent increments starting at 60%). The combinations were selected by inspection after having been ranked with the averaging equation (5.1) shown above. Table 5.16 lists the global predictive accuracy and the κ_t values for each data set for some of the parameter values used in figure 5.11. This table indicates that the global predictive accuracy of our ensemble is not significantly reduced by training with less ground truth. However, the predictive performance of the ensemble, as measured by κ_t, differs between 20% and 54% when examining κ_t when using 60% and 100% of ground truth. The κ_t results indicate that as the ensemble trains with less labelled data, it performs more similarly to a baseline no-change classifier (refer to section 4.5.2).

One finding that we find particularly odd is that as the use of ground truth diminishes, the predictive accuracy increases for the SEA generated data sets with noise. We are led to believe that for increasing levels of noise in a data set, reducing the ground truth used (to an extent) to train a model increases its predictive performance. Further research, outside the scope of this thesis, is

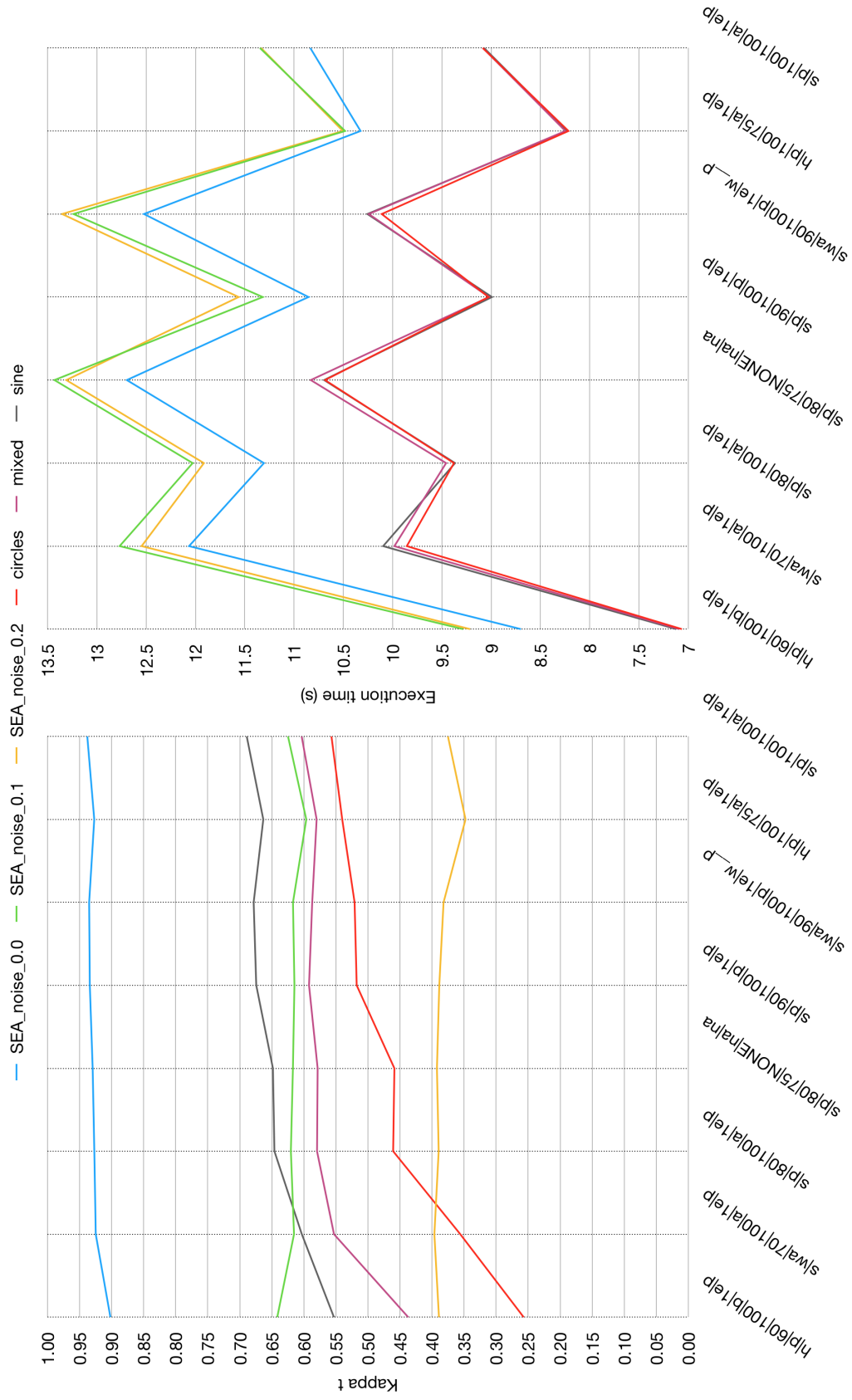

FIGURE 5.11: κ_t and execution times for parameter combinations using varying amounts of ground truth

TABLE 5.16: Accuracy (%) and κ_t when training with varying percentages of labelled data for the parameter combinations in figure 5.11.

GT	CIRCLES		MIXED		SINE1		SEA 0%		SEA 10%		SEA 20%	
	Acc	κ_t	**Acc**	κ_t	**Acc**	κ_t	**Acc**	κ_t	**Acc**	κ_t	**Acc**	κ_t
100	78	0.56	80	0.60	84	0.69	97	0.94	82	0.62	70	0.38
90	76	0.52	79	0.59	84	0.68	97	0.94	82	0.62	70	0.38
80	73	0.46	79	0.58	82	0.65	96	0.93	82	0.62	70	0.39
70	68	0.36	77	0.55	80	0.60	96	0.92	82	0.62	71	0.40
60	63	0.26	72	0.44	77	0.55	95	0.90	83	0.64	70	0.39

needed to ascertain the veracity of this finding. We posit that data generated by SEA is much easier to model than data from CIRCLES, for example, therefore requiring much less labelled data to model.

From figure 5.11 and table 5.16, we find that our ensemble does not suffer a drastic reduction in its global predictive accuracy when training with only 60% ground truth (in other words, training with 40% less labelled data). However, the κ_t statistic indicates that training our ensemble using non-selective self-training on a data set containing less than 70% labelled data results in a predictive performance that is not drastically better than a no-change classifier (for the most part, as results for SEA and the other data sets highly differ). Additionally, we can see that the CIRCLES data set is harder to model with access to less labelled data, which is perfectly logical given what the class label represents. Indeed, very specific data instances are required to model the class well, given that it represents whether or not a data instance resides within the radius of a predefined circle.

It is clear that the reduction in κ_t is dependant on both the data set being modelled as well as how much of that data set is labelled. Overall, we find that training our ensemble with a data set that is 80% or 90% labelled leads to a good predictive accuracy (as measured by κ_t).

5.3 Comparing to the State of the Art

Now that we have determined which parameter combinations worked particularly well, we can compare them to the State of the Art.

As previously mentioned, the algorithms which we will be comparing our voting ensemble to will be mainly the Leveraging Bagging algorithm. As we explained in the chapter 2, the Leveraging Bagging (section 2.3.6) will be comprised of 10 Hoeffding Tree (section 2.3.5) estimators, each with its own ADWIN drift detector. We will also be using a regular Hoeffding Tree using the default parameters without any drift detector. Finally, we will also include a no-change and a majority class classifier in our comparison.

5.3.1 Choosing a window size for State of the Art algorithms

For these algorithms, we made sure to use 100% ground truth for the training, sliding windowing and only modified the window size. However, changing the window size did not change the execution times or κ_t by much more than 1% or 1 second as can be seen in figure 5.12. For this reason, we chose to keep one example for each algorithm, that ranked better with a given window size than the others. We should note that applying the Friedman test and Nemenyi tests showed that the window size resulted in confirming the null hypothesis that all window sizes led to similar results for each classifier for the no change, majority voting, and SGD classifiers. Therefore, we took the best overall ranking window size. For Hoeffding Trees, a window size of 25 showed a significant statistical difference to 100 but only for the execution time. For Leveraging Bagging, a window size of 25 showed a significant statistical difference to 100 but only for the κ_t metric.

FIGURE 5.12: κ_t and execution times of State of the Art algorithm with varying window sizes

The resulting chosen window sizes are as follows: no change (25), majority voting (25), SGD (75), Hoeffding Tree (25), Leveraging Bagging (25).

We have opted to compare these algorithms to our voting ensemble with six different parameter combinations (1 for each increment of ground truth used, and an additional one using 100% ground truth).

5.3.2 Visual comparison

Finally, we can compare our voting ensemble to the state of the art. Figure 5.13 shows the raw results, to better visualise how each algorithm, and its parameter combinations affect the data sets that they are trying to model.

As we can see from these two graphs, Leveraging Bagging (LB) does achieve the best predictive accuracy but the worst execution time. While the difference in predictive accuracy between LB and the other algorithms is noticeable, it is not glaring. However, when it comes to execution time, we were required to use a logarithmic scale to show its run time while also showing the run times of other algorithms. LB takes more than two orders of magnitude longer than the Voting Ensemble, and 1.5 orders of magnitude longer than a Hoeffding Tree (HT). Given that LB is comprised of 10 HTs, it makes perfect sense that LB takes so much longer to run.

However, our findings from the graph do not have the weight of a proper statistical analysis, which follows in the next section.

5.3.3 Statistical Analysis

In this final section, we will test the following two null hypotheses:

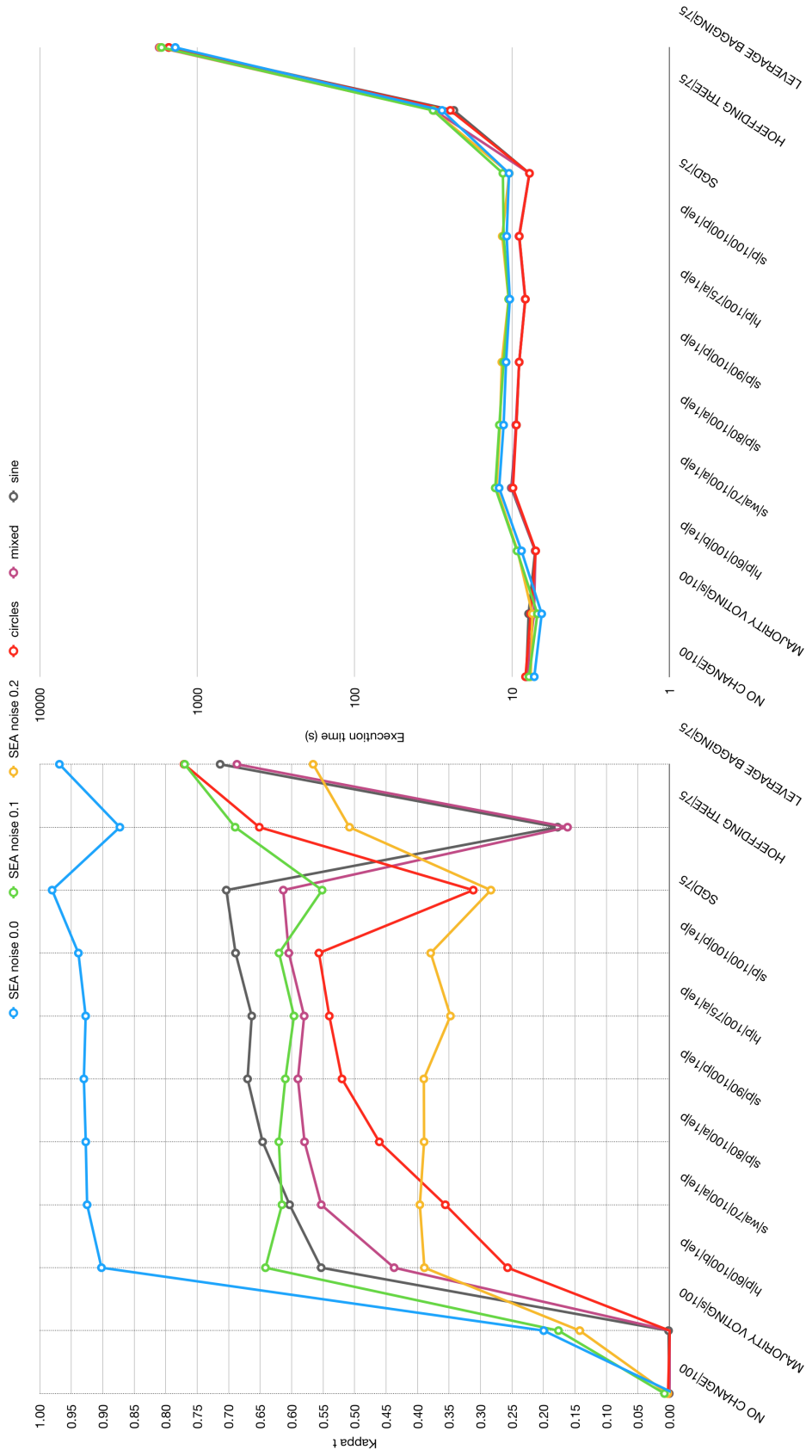

FIGURE 5.13: κ_t and execution times when comparing our Voting Ensemble to the State of the Art

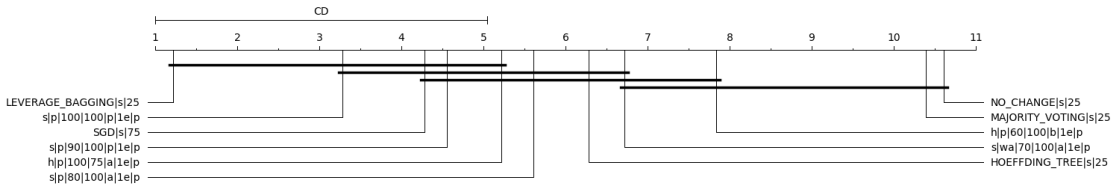

FIGURE 5.14: Nemenyi graph ranking κ_t for various algorithms

1. all algorithms, with their respective parameters predict classes equally well (κ_t)

2. all algorithms, with their respective parameters run in an equal amount of time.

For κ_t

We will start with the first, using the κ_t metric. Again, the Friedman test was used, with a significance level of 0.05. We found that $p < 2.1 \times 10^{-23}$, thus rejecting the null hypothesis. To determine which pairs of algorithms actually differ, we used the post-hoc Nemenyi test, yet again. The results can be seen in figure 5.14, where a lower rank means a better predictive accuracy (a better κ_t).

A Nemenyi graph shows a ranking of algorithms on a scale from 1 to N (typically the number of algorithms compared). A bar labelled critical difference (CD) is shown above the scale, which is the minimum rank length for two algorithms to not show a significant statistical difference in rank. Additionally, there may be horizontal bars that link ranked algorithms. Any algorithms that share the same horizontal bar are not significantly statistically different. Pairs of algorithms that are further apart than the CD bar are significantly statistically different.

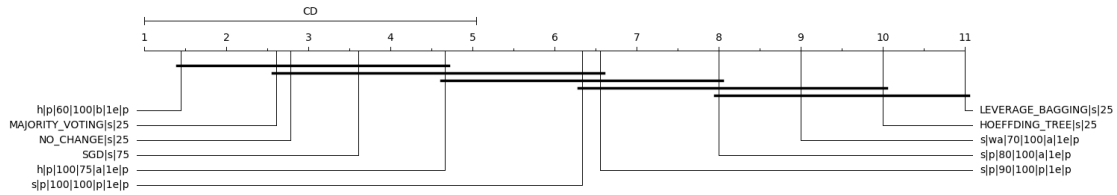

FIGURE 5.15: Nemenyi graph ranking execution times for various algorithms

The graph shows that there is no significant statistical difference between LB, our Voting Ensemble using our best overall parameter combination, an SGD classifier, our Voting Ensemble using our hybrid windowing approach, and our Voting Ensemble using only 80% ground truth when training. This supports the rejection of the null hypothesis for κ_t. It is also a very good sign, because all of the above-mentioned algorithms were statistically significantly better than a single Hoeffding Tree.

Note that there is a significant statistical difference between both the majority voting and no change classifiers with all algorithms, aside from our Voting Ensemble training with 70% of ground truth or less.

Therefore, this test showed that our Voting Ensemble, using our preferred parameter combinations, did not perform better or worse, statistically speaking, than Leveraging Bagging.

Considering execution time

For the final measure, execution time, we use the Friedman test, with a significance level of 0.05. We found that $p < 1.68 \times 10^{-31}$, thus leaving no doubt as to the rejection of the null hypothesis. The post-hoc Nemenyi test is used to determine which pairs of algorithms differ. The Nemenyi graph is shown in figure 5.15, where a lower rank means a lower, thus better, execution time.

The graph indicates that Leveraging Bagging ranks last, and Hoeffding Trees ranks second last, which is as expected given the raw values that we saw above. The graph also shows that there is a significant statistical difference between Leveraging Bagging (the State of the Art algorithm we are comparing), and our Voting Ensemble (except when using 70% or 80% ground truth). Given that Leveraging Bagging runs in over two orders of magnitude longer than our Voting Ensemble, this result is not surprising in the least. It is, however, comforting to have our visual analysis backed by this statistical test.

Therefore, this test statistically showed that our Voting Ensemble runs significantly faster than Leveraging Bagging.

Refer to appendix A.1 for additional statistical significance heat-maps and Nemenyi graphs.

5.3.4 Discussion

First of all, our statistical significance tests show that our Voting Ensemble is able to outperform the State of the Art *Leveraging Bagging* algorithm in execution time, and that it is able to perform on par with *Leveraging Bagging* in regards to the κ_t measured metric. It also indicated that the predictive performance of our ensemble when trained with only 90% ground truth does not present a statistical significant difference to that of *Leveraging Bagging*. When using a p value of 0.01, however, *Leveraging Bagging*'s predictive performance does not present a difference that is statistically different to those of our ensemble trained on a data set containing only 80% labelled data.

Our algorithm therefore predicts comparably to Leveraging Bagging and brings outstanding time savings in algorithm execution-time, running approximately 160 times faster.

5.4 Summary

A preliminary examination of our results shows that our framework's execution time is, at most, very loosely tied to its predictive accuracy.

An investigation on the impact of each parameter value on the mean of each metric leads us to determine how to roughly maximise κ_t or minimise the execution time. The differences lie with the window type, the batch size, the percentage of labelled data used and, partially, the drift detector count. Our results indicate that using probability voting, one detector per ensemble to detect drifts, and resetting all classifiers when drifts occur leads to better κ_t values and a lower execution time. For the batch size, window type, and the drift detector count, it is entirely logical that choosing one value over another would change the execution time as they were, at least partially, implemented as time-saving measures.

The findings above are confirmed, and parameter values that tend to rank well are identified by ranking the parameter combinations for each metric. By examining the paired rankings (time and predictive performance), we propose an alternative parameter combination[3] that achieves significant time savings over the one with the best predictive performance, while only reducing κ_t by one to four percent.

[3] hybrid windowing, probability voting, 100% ground truth, batch sizes of 75 instances, one drift detector for the ensemble tracking its confidence, and completely resetting the ensemble when drifts are detected

Our framework runs roughly 160 times faster than the state of the art *Leveraging Bagging* algorithm, and it is comparable in terms of the measured κ_t metric. Training with only 90% labelled data does not compromise our framework's predictive accuracy in comparison to that of *Leveraging Bagging*, in that no statistical significant difference is observed. Our results also indicate that the predictive accuracy of our ensemble when training on data sets containing less than 80% labelled data approaches the traditional threshold for statistical significance, and presents no significant difference when using a p-value of 0.01.

Practically, this means that our ensemble should definitely be considered when execution time is an important metric, since the predictive performance is comparable to that of Leveraging bagging.

Chapter 6

Conclusion

This thesis focused on improving semi-supervised learning from evolving streams without the use of clustering techniques, which are computationally expensive [49]. The goal of this study was to design fast algorithms to work with fewer labelled examples, and to extend an existing algorithm to detect drifts without relying on ground truth. Experiments were conducted in order to compare the performance of our framework against that of the state of the art in terms of predictive accuracy and execution time, while considering the percentage of labelled instances used at each stage of learning. This chapter discusses our contributions and presents opportunities for future work.

6.1 Contributions

A great deal of research is being conducted to develop algorithms for supervised learning of evolving streams. These techniques are usually ensemble-based, and typically make use of either boosting or bagging. This research is necessary to understand how well an algorithm can learn from an evolving stream, but the use of supervised training is to online learning as training wheels are to learning

how to ride a bicycle. In the real-world, the assumption that true labels will arrive on time is unequivocally unreasonable. We introduce our semi-supervised hybrid-windowing ensembles for learning in evolving streams as our solution to deal with this issue without using clustering techniques.

We first proposed a voting ensemble that uses a modified soft voting approach, by weighting each classifier's predictive confidence. Next, we developed a novel windowing type for ensembles, as sliding windows are very time consuming and regular tumbling windows are not a suitable replacement. Our windowing technique can be considered a hybrid of the two: we train each sub-classifier in the ensemble with tumbling windows, but delay training in such a way that only one sub-classifier can update its model per iteration. We also extended selective self-training by ignoring its heuristic: training classifiers using all predicted examples, and not only those with high predictive confidence. Finally, we extended an existing concept drift detector to successfully operate without any labelled data, by using a sliding window of our ensemble's prediction confidence, instead of a boolean value indicating whether, or not, the ensemble predicted correctly.

6.2 Future Work

Our framework took very little execution time, far below that of a current state of the art technique, and achieved comparable predictive accuracy to that of state of the art techniques that trained on fully-labelled data sets, while ours only trained on a subset of those labels, and ignoring them completely for detecting drifts. We believe that a higher predictive accuracy can be achieved without

significantly impacting the execution time, even when training with less labelled data.

In order to reduce the quantity of labelled data used for (pre-)training, we can introduce specialised domain knowledge into the learning cycle. An interactive active learning approach can be used, as seen in [30], where an intuitive online interface is used to request labels, from an oracle, for any number of uncertain data instances to train on instances that are representative of the stream, or those that classifiers find difficult.

Furthermore, our framework can be improved by incorporating ideas that have been proven to work such as intelligent window sizes, and replacing classifiers with a bad predictive accuracy streak. To deal with recurring concepts, we can incorporate weighted summarising classifiers as seen in Learn^{++}.NSE [27], however, the memory issue would need to be addressed.

Additionally, our framework does not guarantee diversity among the classifiers in the ensemble. A possible approach would be to replace the cyclic training from hybrid windows with a stochastic method. In other words, instead of training the classifiers in the ensemble in the same cyclical sequential manner, a classifier would be chosen at random to be trained from the new hybrid window.

Another area to be explored is how our framework performs when dealing with mixed concept drifts, and perform a study using a real-world data set.

Finally, our framework can be extended to better deal with highly imbalanced datasets. This would increase its applicability to real-world problems such as intrusion detection, and fraud detection.

Appendix A

Graphs

A.1 Nemenyi Graphs

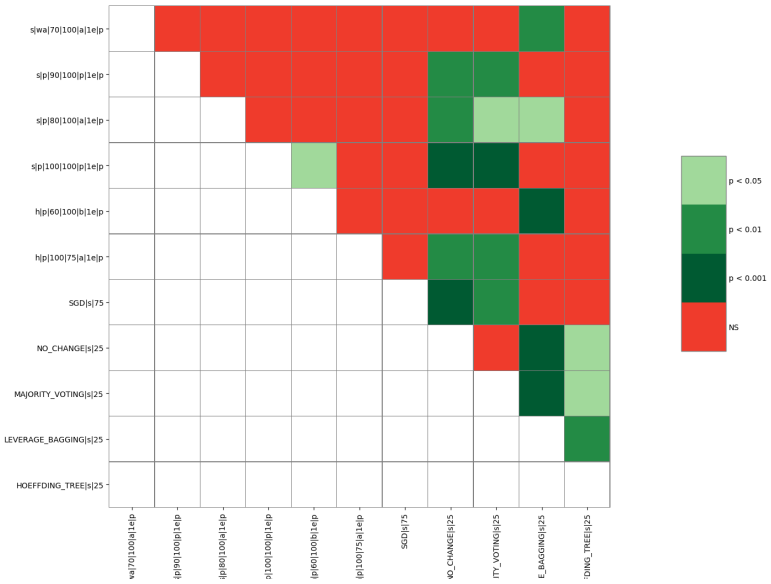

FIGURE A.1: State of the Art comparison: κ_t heatmap

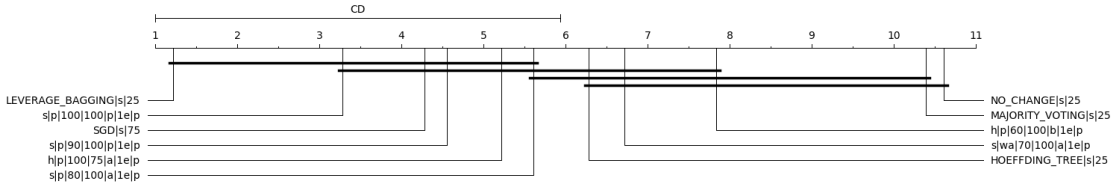

FIGURE A.2: State of the Art comparison: post-hoc Nemenyi graph for κ_t, $\alpha = 0.01$

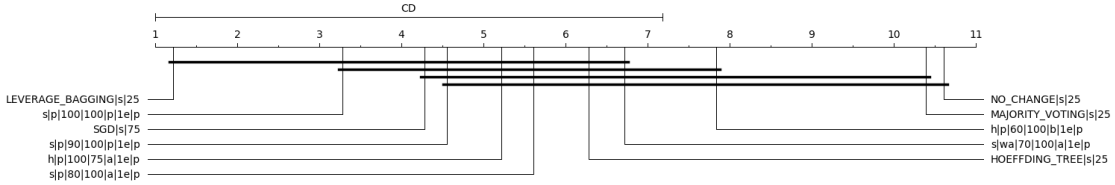

FIGURE A.3: State of the Art comparison: post-hoc Nemenyi graph for κ_t, $\alpha = 0.001$

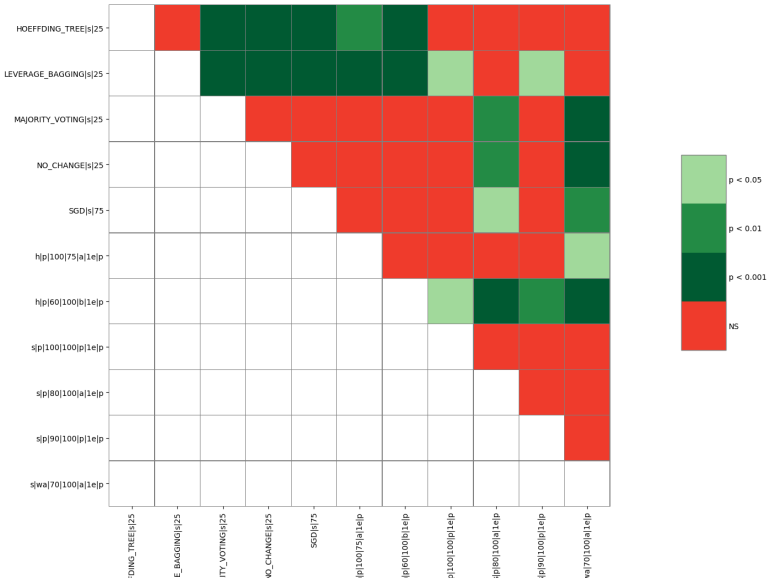

FIGURE A.4: State of the Art comparison: execution time heatmap

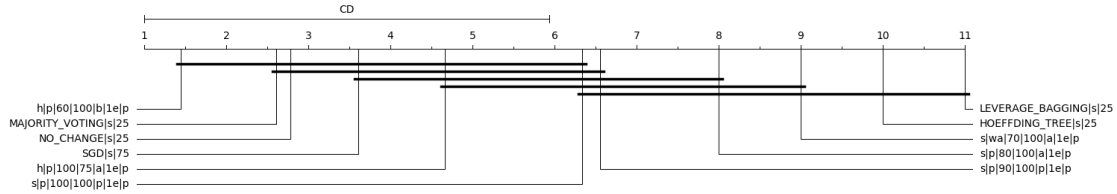

FIGURE A.5: State of the Art comparison: post-hoc Nemenyi graph for execution time, $\alpha = 0.01$

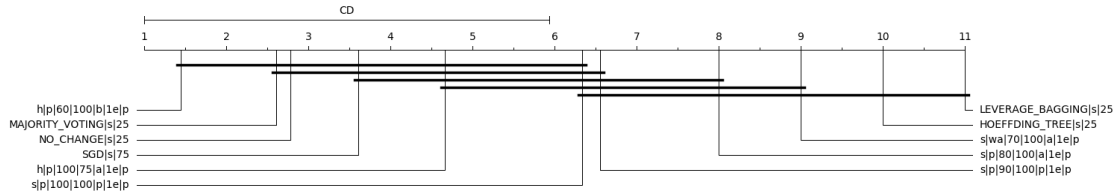

FIGURE A.6: State of the Art comparison: post-hoc Nemenyi graph for execution time, $\alpha = 0.001$

Appendix B

Summaries

As recommended by Silyn in [87, p. 151], we have created this chapter to collect all of the chapter summaries.

B.1 Chapter 2: Background Work

This chapter presents the fundamentals of machine learning and the various challenges of extracting knowledge from data streams. We introduce classification as a type of machine learning, for which the goal is to extract knowledge, computationally, from labelled data. Algorithms are developed to model the relationship between the data and the class labels. Semi-supervised learning is also a classification task but with the added constraint of learning from a data set that is not entirely labelled. In this thesis, the data we learn from come from data streams, which are voluminous, volatile and velocious. As such, constraints on time and memory usage must be respected, and a mechanism is required to forget "old data" safely.

We cover specific classifiers: baseline classifiers are usually relatively simple and used as a baseline for comparing classifier performance. We also cover the algorithms that we employed in our framework, as well as the state of the art: Naive Bayes, Stochastic Gradient Descent, Hoeffding Trees and Leveraging Bagging.

We define concept drifts as an evolution in the probability distribution of classes and/or attributes. We describe the main types of concept drifts that can occur, regardless of how self-explanatory they are named: abrupt, gradual and recurring. We then review drift detection strategies presented in the literature, including FHDDM/S which we extend in this thesis. Our review shows a gap in research as it pertains to semi-supervised drift detection.

Next, we define ensembles as an amalgamation of a number of classifiers. As all classifiers must output a prediction, ensembles must as well; we present existing techniques to map these multiple outputs to a single one. Having defined ensembles, we review those that were proposed in the literature using two criteria: the processing method and whether or not they were designed to deal with evolving concepts. The processing method distinguishes if instances are analysed online (one-by-one) or in batches. Our review shows a gap in research as it pertains to semi-supervised learning from evolving streams.

B.2 Chapter 3: Contributions

In this chapter, we introduce our methodology, entitled Learning from Evolving Streams via Self-Training Novel Windowing Ensembles (LESS-TWE).

We propose a weighted soft voting scheme that uses a hyperbolic tangent function. We choose the *tanh* function, as it allows us to approximate a logistic function while also being less computationally expensive [55, p. 10].

Next, we propose a novel windowing technique, exclusive to ensembles. Our technique allows every classifier in the ensemble to train from each data point in the stream **exactly once** similar to tumbling windows, as opposed to sliding windows where a data point is trained on **at least once**. As such, our method presents a trade-off between accuracy and execution time. Only one classifier per batch is trained on the window, therefore each classifier trains on a specific data instance at different times, resulting in delayed training. This means that only one classifier in the ensemble is trained on the newest data before the ensemble predicts labels for the subsequent batch. The motivation for this technique is to determine if we can spend less execution time training the classifiers and to investigate how progressively delaying training of some of the classifiers in the ensemble affects concept drift detection and classification performance.

Our next contribution relates to selective self-training. In this semi-supervised learning algorithm, a classifier assigns a label it predicts to unlabelled data for future learning, if it is highly confident it is correct. There have not yet been any attempts, to the best of our knowledge, to investigate how self-training performs if labelling all unlabelled data, regardless of the classifier's confidence in its predicted label.

Finally, our last contribution is an extension of the Fast Hoeffding Drift Detecting Method for evolving data Streams (FHDDMS), which detects drifts using the prequential accuracy. In our extension, we propose three schemes, one of which makes use of the average of the ensemble's classifiers' confidences. Our extension allows FHDDMS to run without any labelled data, therefore making

it an unsupervised drift detector.

Finally, we will cover all of our contributions by using a toy example to explain how data is processed from a stream.

B.3 Chapter 4: Experimental Design

In this chapter, we describe our experimental design.

All experiments are conducted on a MacBook Pro model *11,4* running Python 3.7.3.

The data sets used for our analysis in the upcoming chapter comprise of generated SEA data with [0, 10, 20] percent noise as well as CIRCLES, SINE1 and MIXED data sets, which contain 10% noise. Our data sets contain either abrupt or gradual concept drifts.

The estimation technique we use is prequential evaluation, also known as interleaved test-then-train, which consists of infinitely executing a loop where a classifier first predicts labels for new data (without its label), then adapts its model for said data, with the correct label. The prequential evaluation loop is provided by scikit-multiflow, a Python framework backed by Bifet.

The performance measures that we use are the execution time, measured in seconds, as well as the κ-temporal statistic to evaluate a classifier's predictive performance, also called κ^+ or κ_t. This κ statistic compares our classifier to a no-change classifier and takes into account temporal dependence in the data.

Using the mean values for the entire stream for both of the metrics mentioned above, we use statistical tests to determine whether or not the differences observed are statistically significant and not due to simple coincidence. When comparing two classifiers across multiple data sets, we use the Wilcoxon test, and when comparing more than two classifiers, we use the Friedman test, coupled with the post-hoc Nemenyi test.

We conduct the following experiments:

- examine the impact of each parameter value on the mean of each metric,
- rank the results of each parameter combination in order to establish any trend regarding parameter values across the metrics,
- compare the top ranking parameter combinations to the state of the art.

In the following chapter, we will present the results of our experiments, analyse these findings and discuss their significance.

B.4 Chapter 5: Experimental Evaluation and Discussion

A preliminary examination of our results shows that our framework's execution time is, at most, very loosely tied to its predictive accuracy.

An investigation on the impact of each parameter value on the mean of each metric leads us to determine how to roughly maximise κ_t or minimise the execution time. The differences lie with the window type, the batch size, the percentage of labelled data used and, partially, the drift detector count. Our results indicate that using probability voting, one detector per ensemble to detect drifts,

and resetting all classifiers when drifts occur leads to better κ_t values and a lower execution time. For the batch size, window type, and the drift detector count, it is entirely logical that choosing one value over another would change the execution time as they were, at least partially, implemented as time-saving measures.

The findings above are confirmed, and parameter values that tend to rank well are identified by ranking the parameter combinations for each metric. By examining the paired rankings (time and predictive performance), we propose an alternative parameter combination[1] that achieves significant time savings over the one with the best predictive performance, while only reducing κ_t by one to four percent.

Our framework runs roughly 160 times faster than the state of the art *Leveraging Bagging* algorithm, and it is comparable in terms of the measured κ_t metric. Training with only 90% labelled data does not compromise our framework's predictive accuracy in comparison to that of *Leveraging Bagging*, in that no statistical significant difference is observed. Our results also indicate that the predictive accuracy of our ensemble does not suffer until train on data sets containing less than 80% labelled data.

Practically, this means that our framework should definitely be considered when execution time is an important metric. However, for the applications that strictly require high predictive accuracy, Leveraging Bagging would still be the preferred choice.

[1] hybrid windowing, probability voting, 100% ground truth, batch sizes of 75 instances, one drift detector for the ensemble tracking its confidence, and completely resetting the ensemble when drifts are detected

Bibliography

[1] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[2] Manuel Baena-García et al. "Early drift detection method". In: (2006).

[3] Bruno Baruque and Emilio Corchado. *Fusion methods for unsupervised learning ensembles*. Springer, 2011.

[4] Albert Bifet and Eibe Frank. "Sentiment knowledge discovery in twitter streaming data". In: *International conference on discovery science*. Springer. 2010, pp. 1–15.

[5] Albert Bifet and Ricard Gavalda. "Learning from time-changing data with adaptive windowing". In: *Proceedings of the 2007 SIAM international conference on data mining*. SIAM. 2007, pp. 443–448.

[6] Albert Bifet, Geoff Holmes, and Bernhard Pfahringer. "Leveraging bagging for evolving data streams". In: *Joint European conference on machine learning and knowledge discovery in databases*. Springer. 2010, pp. 135–150.

[7] Albert Bifet and Richard Kirkby. "Data stream mining a practical approach". In: (2009).

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

[9] Albert Bifet et al. "Improving adaptive bagging methods for evolving data streams". In: *Asian conference on machine learning*. Springer. 2009, pp. 23–37.

[10] Albert Bifet et al. *Machine learning for data streams: with practical examples in MOA*. MIT Press, 2018.

[11] Albert Bifet et al. "MOA: Massive Online Analysis". In: *Journal of Machine Learning Research* 11 (2010), pp. 1601–1604. URL: http://portal.acm.org/citation.cfm?id=1859903.

[12] Albert Bifet et al. "New ensemble methods for evolving data streams". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2009, pp. 139–148.

[13] Hendrik Blockeel et al., eds. *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part I*. Vol. 8188. Lecture Notes in Computer Science. Springer, 2013. ISBN: 978-3-642-40987-5. DOI: 10.1007/978-3-642-40988-2. URL: https://doi.org/10.1007/978-3-642-40988-2.

[14] Léon Bottou. "Stochastic gradient descent tricks". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.

[15] Léon Bottou and Olivier Bousquet. "Learning using large datasets". In: *Mining Massive DataSets for Security* 3 (2008).

[16] Dariusz Brzezinski and Jerzy Stefanowski. "Combining block-based and online methods in learning ensembles from concept drifting data streams". In: *Information Sciences* 265 (2014), pp. 50 –67. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2013.12.011. URL: http://www.sciencedirect.com/science/article/pii/S0020025513008554.

[17] Dariusz Brzezinski and Jerzy Stefanowski. "Reacting to different types of concept drift: The accuracy updated ensemble algorithm". In: *IEEE Transactions on Neural Networks and Learning Systems* 25.1 (2014), pp. 81–94.

[18] Fang Chu and Carlo Zaniolo. "Fast and light boosting for adaptive mining of data streams". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2004, pp. 282–292.

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

[20] Magdalena Deckert. "Batch weighted ensemble for mining data streams with concept drift". In: *International Symposium on Methodologies for Intelligent Systems*. Springer. 2011, pp. 290–299.

[21] Misha Denil, David Matheson, and Nando Freitas. "Consistency of online random forests". In: *International conference on machine learning*. 2013, pp. 1256–1264.

[22] Sarah D'Ettorre. "Fine-Grained, Unsupervised, Context-based Change Detection and Adaptation for Evolving Categorical Data". PhD thesis. Université d'Ottawa/University of Ottawa, 2016.

[23] Pedro Domingos and Geoff Hulten. "Mining high-speed data streams". In: *Kdd*. Vol. 2. 2000, p. 4.

[24] Anton Dries and Ulrich Rückert. "Adaptive concept drift detection". In: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 2.5-6 (2009), pp. 311–327.

[25] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[26] Sarah D'Ettorre, Herna L Viktor, and Eric Paquet. "Context-Based Abrupt Change Detection and Adaptation for Categorical Data Streams". In: *International Conference on Discovery Science*. Springer. 2017, pp. 3–17.

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

[28] Ronald A Fisher. "Statistical methods and scientific inference." In: (1956).

[29] Peter Flach. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012. DOI: 10.1017/CBO9780511973000.

[30] Sean Floyd et al. "An Active Learning System for Text Classification". In: (2017). Accessed: 2019-04-08. URL: https://github.com/SeanLF/CSI4900/wiki/report.pdf.

[31] Isvani Frías-Blanco et al. "Online and non-parametric drift detection methods based on Hoeffding's bounds". In: *IEEE Transactions on Knowledge and Data Engineering* 27.3 (2015), pp. 810–823.

[32] Jerome H Friedman and Lawrence C Rafsky. "Multivariate generalizations of the Wald-Wolfowitz and Smirnov two-sample tests". In: *The Annals of Statistics* (1979), pp. 697–717.

[33] Milton Friedman. "The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance". In: *Journal of the American Statistical Association* 32.200 (1937), pp. 675–701. DOI: 10.1080/01621459.1937.10503522. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1937.10503522. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1937.10503522.

[34] João Gama et al. "A Survey on Concept Drift Adaptation". In: *ACM Comput. Surv.* 46.4 (Mar. 2014), 44:1–44:37. ISSN: 0360-0300. DOI: 10.1145/2523813. URL: http://doi.acm.org/10.1145/2523813.

[35] Joao Gama. *Knowledge discovery from data streams*. Chapman and Hall/CRC, 2010.

[36] João Gama, Pedro Medas, and Pedro Rodrigues. "Learning decision trees from dynamic data streams". In: *Proceedings of the 2005 ACM symposium on Applied computing*. ACM. 2005, pp. 573–577.

[37] João Gama et al. "A survey on concept drift adaptation". In: *ACM computing surveys (CSUR)* 46.4 (2014), p. 44.

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

[39] Mohammed Ghesmoune, Mustapha Lebbah, and Hanene Azzag. "State-of-the-art on clustering data streams". In: *Big Data Analytics* 1.1 (2016), p. 13.

[40] Ahsanul Haque, Latifur Khan, and Michael Baron. "Semi Supervised Adaptive Framework for Classifying Evolving Data Stream". In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Tru Cao et al. Cham: Springer International Publishing, 2015, pp. 383–394. ISBN: 978-3-319-18032-8.

[41] Sajad Homayoun and Marzieh Ahmadzadeh. "A review on data stream classification approaches". In: *Journal of Advanced Computer Science & Technology* 5.1 (2016), pp. 8–13.

[42] David Tse Jung Huang et al. "Drift detection using stream volatility". In: *Joint European conference on machine learning and knowledge discovery in databases*. Springer. 2015, pp. 417–432.

[43] Anil K Jain, Robert P. W. Duin, and Jianchang Mao. "Statistical pattern recognition: A review". In: *IEEE Transactions on pattern analysis and machine intelligence* 22.1 (2000), pp. 4–37.

[44] Nathalie Japkowicz and Mohak Shah. *Evaluating learning algorithms: a classification perspective*. Cambridge University Press, 2011.

[45] Takuya Kidera, Seiichi Ozawa, and Shigeo Abe. "An incremental learning algorithm of ensemble classifier systems". In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. IEEE. 2006, pp. 3421–3427.

[46] Jeremy Z. Kolter and Marcus A. Maloof. "Using Additive Expert Ensembles to Cope with Concept Drift". In: *Proceedings of the 22Nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: ACM, 2005, pp. 449–456. ISBN: 1-59593-180-5. DOI: 10.1145/1102351.1102408. URL: http://doi.acm.org/10.1145/1102351.1102408.

[47] J.Z. Kolter and M.A. Maloof. "Dynamic weighted majority: An ensemble method for drifting concepts". In: *Journal of Machine Learning Research* 8 (2007). cited By 336, pp. 2755–2790. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-37749050180&partnerID=40&md5=c08b89c1a7bbc036470dd8be9f66f57e.

[48] Bartosz Krawczyk et al. "Ensemble learning for data stream analysis: A survey". In: *Information Fusion* 37 (2017), pp. 132 –156. ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2017.02.004. URL: http://www.sciencedirect.com/science/article/pii/S1566253516302329.

[49] Georg Krempl et al. "Open challenges for data stream mining research". In: *ACM SIGKDD explorations newsletter* 16.1 (2014), pp. 1–10.

[50] Miroslav Kubat and Gerhard Widmer. "Adapting to drift in continuous domains (Extended abstract)". In: *Machine Learning: ECML-95*. Ed. by Nada Lavrac and Stefan Wrobel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 307–310. ISBN: 978-3-540-49232-0.

[51] Ludmila I Kuncheva. "Classifier ensembles for changing environments". In: *International Workshop on Multiple Classifier Systems*. Springer. 2004, pp. 1–15.

[52] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2004.

[53] Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. "Mondrian forests: Efficient online random forests". In: *Advances in neural information processing systems*. 2014, pp. 3140–3148.

[54] Yuan Lan, Yeng Chai Soh, and Guang-Bin Huang. "Ensemble of online sequential extreme learning machine". In: *Neurocomputing* 72.13-15 (2009), pp. 3391–3395.

[55] Yann A LeCun et al. "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[56] Jie Lu et al. "Learning under Concept Drift: A Review". In: *IEEE Transactions on Knowledge and Data Engineering* (2018).

[57] Fernanda Li Minku, Hirotaka Inoue, and Xin Yao. "Negative correlation in incremental learning". In: *Natural Computing* 8.2 (2009), pp. 289–320.

[58] Leandro L Minku, Allan P White, and Xin Yao. "The impact of diversity on online ensemble learning in the presence of concept drift". In: *IEEE Transactions on knowledge and Data Engineering* 22.5 (2010), pp. 730–742.

[59] Leandro L Minku and Xin Yao. "DDD: A new ensemble approach for dealing with concept drift". In: *IEEE transactions on knowledge and data engineering* 24.4 (2012), pp. 619–633.

[60] Jacob Montiel et al. "Scikit-Multiflow: A Multi-output Streaming Framework". In: *CoRR* abs/1807.04662 (2018). URL: https://github.com/scikit-multiflow/scikit-multiflow.

[61] Peter Nemenyi. "Distribution-free multiple comparisons". In: *Biometrics*. Vol. 18. 2. INTERNATIONAL BIOMETRIC SOC 1441 I ST, NW, SUITE 700, WASHINGTON, DC 20005-2210. 1962, p. 263.

[62] KYOSUKE Nishida and Koichiro Yamauchi. "Adaptive classifiers-ensemble system for tracking concept drift". In: *2007 International Conference on Machine Learning and Cybernetics*. Vol. 6. IEEE. 2007, pp. 3607–3612.

[63] Kyosuke Nishida and Koichiro Yamauchi. "Detecting concept drift using statistical testing". In: *International conference on discovery science*. Springer. 2007, pp. 264–269.

[64] M Kehinde Olorunnimbe, Herna L Viktor, and Eric Paquet. "Intelligent adaptive ensembles for data stream mining: a high return on investment approach". In: *International workshop on new frontiers in mining complex patterns*. Springer. 2015, pp. 61–75.

[65] David Opitz and Richard Maclin. "Popular ensemble methods: An empirical study". In: *Journal of artificial intelligence research* 11 (1999), pp. 169–198.

[66] Nikunj C Oza. "Online bagging and boosting". In: *2005 IEEE international conference on systems, man and cybernetics*. Vol. 3. Ieee. 2005, pp. 2340–2345.

[67] Nikunj C Oza and Kagan Tumer. "Classifier ensembles: Select real-world applications". In: *Information Fusion* 9.1 (2008), pp. 4–20.

[68] Ewan S Page. "Continuous inspection schemes". In: *Biometrika* 41.1/2 (1954), pp. 100–115.

[69] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[70] Ali Pesaranghader. "A Reservoir of Adaptive Algorithms for Online Learning from Evolving Data Streams". PhD thesis. Université d'Ottawa/University of Ottawa, 2018.

[71] Ali Pesaranghader, Herna Viktor, and Eric Paquet. "Reservoir of diverse adaptive learners and stacking fast hoeffding drift detection methods for evolving data streams". In: *Machine Learning* 107.11 (2018), pp. 1711–1743.

[72] Ali Pesaranghader and Herna L Viktor. "Fast hoeffding drift detection method for evolving data streams". In: *Joint European conference on machine learning and knowledge discovery in databases*. Springer. 2016, pp. 96–111.

[73] Robi Polikar. "Ensemble based systems in decision making". In: *IEEE Circuits and systems magazine* 6.3 (2006), pp. 21–45.

[74] Robi Polikar et al. "Learn++: An incremental learning algorithm for supervised neural networks". In: *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)* 31.4 (2001), pp. 497–508.

[75] Sasthakumar Ramamurthy and Raj Bhatnagar. "Tracking recurrent concept drift in streaming data using ensemble classifiers". In: *Sixth International Conference on Machine Learning and Applications (ICMLA 2007)*. IEEE. 2007, pp. 404–409.

[76] Jesse Read et al. "MEKA: A Multi-label/Multi-target Extension to Weka". In: *Journal of Machine Learning Research* 17.21 (2016), pp. 1–5. URL: http://jmlr.org/papers/v17/12-164.html.

[77] Lior Rokach. "Ensemble-based classifiers". In: *Artificial Intelligence Review* 33.1-2 (2010), pp. 1–39.

[78] Lior Rokach. *Pattern classification using ensemble methods*. Vol. 75. World Scientific, 2010.

[79] Lior Rokach. "Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography". In: *Computational Statistics & Data Analysis* 53.12 (2009), pp. 4046–4072.

[80] Amir Saffari et al. "On-line random forests". In: *2009 ieee 12th international conference on computer vision workshops, iccv workshops*. IEEE. 2009, pp. 1393–1400.

[81] Martin Scholz and Ralf Klinkenberg. "An ensemble classifier for drifting concepts". In: *Proceedings of the Second International Workshop on Knowledge Discovery in Data Streams*. Vol. 6. 11. Porto, Portugal. 2005, pp. 53–64.

[82] Giovanni Seni and John F Elder. "Ensemble methods in data mining: improving accuracy through combining predictions". In: *Synthesis Lectures on Data Mining and Knowledge Discovery* 2.1 (2010), pp. 1–126.

[83] *SGDClassifier scikit-learn documentation*. https://scikit-learn.org/stable/modules/sgd.html. Accessed: 2019-03-28.

[84] Shai Shalev-Shwartz et al. "Pegasos: Primal estimated sub-gradient solver for svm". In: *Mathematical programming* 127.1 (2011), pp. 3–30.

[85] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[86] Jonathan A Silva et al. "Data stream clustering: A survey". In: *ACM Computing Surveys (CSUR)* 46.1 (2013), p. 13.

[87] Heather Silyn-Roberts. *Writing for science and engineering: Papers, presentations and reports*. Newnes, 2012.

[88] Piotr Sobolewski and Michal Wozniak. "Concept Drift Detection and Model Selection with Simulated Recurrence and Ensembles of Statistical Detectors." In: *J. UCS* 19.4 (2013), pp. 462–483.

[89] Kenneth O Stanley. "Learning concept drift with a committee of decision trees". In: *Informe técnico: UT-AI-TR-03-302, Department of Computer Sciences, University of Texas at Austin, USA* (2003).

[90] W Nick Street and YongSeog Kim. "A streaming ensemble algorithm (SEA) for large-scale classification". In: *Proceedings of the seventh ACM*

SIGKDD international conference on Knowledge discovery and data mining. ACM. 2001, pp. 377–382.

[91] Yoshimasa Tsuruoka, Jun'ichi Tsujii, and Sophia Ananiadou. "Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty". In: *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1-Volume 1*. Association for Computational Linguistics. 2009, pp. 477–485.

[92] Alexey Tsymbal. "The problem of concept drift: definitions and related work". In: *Computer Science Department, Trinity College Dublin* 106.2 (2004), p. 58.

[93] Herna L. Viktor. *Lecture notes in Computer Science: CSI 5387 - Data Mining and Concept Learning*. 2016.

[94] Haixun Wang et al. "Mining concept-drifting data streams using ensemble classifiers". In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. AcM. 2003, pp. 226–235.

[95] Gerhard Widmer and Miroslav Kubat. "Learning in the presence of concept drift and hidden contexts". In: *Machine learning* 23.1 (1996), pp. 69–101.

[96] Frank Wilcoxon. "Individual comparisons by ranking methods". In: *Biometrics bulletin* 1.6 (1945), pp. 80–83.

[97] Michał Woźniak. "Application of combined classifiers to data stream classification". In: *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer. 2013, pp. 13–23.

[98] Michał Woźniak, Manuel Graña, and Emilio Corchado. "A survey of multiple classifier systems as hybrid systems". In: *Information Fusion* 16 (2014), pp. 3–17.

[99] Shin-ichi Yoshida et al. "Adaptive online prediction using weighted windows". In: *IEICE TRANSACTIONS on Information and Systems* 94.10 (2011), pp. 1917–1923.

[100] Harry Zhang. "The optimality of naive Bayes". In: *AA* 1.2 (2004), p. 3.

[101] Qiang Li Zhao, Yan Huang Jiang, and Ming Xu. "Incremental learning by heterogeneous bagging ensemble". In: *International Conference on Advanced Data Mining and Applications*. Springer. 2010, pp. 1–12.

[102] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC, 2012.

[103] Xiaojin Zhu and Andrew B. Goldberg. "Introduction to Semi-Supervised Learning". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 3.1 (2009), pp. 1–130. DOI: 10.2200/S00196ED1V01Y200906AIM006. eprint: https://doi.org/10.2200/S00196ED1V01Y200906AIM006. URL: https://doi.org/10.2200/S00196ED1V01Y200906AIM006.

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