SEMI-SUPERVISED ADAPTIVE CLASSIFICATION OVER DATA STREAMS

by

Ahsanul Haque

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Latifur Khan, Chair

Dr. Alvaro Cárdenas

Dr. Kevin W. Hamlen

Dr. Murat Kantarcioglu

This dissertation is dedicated

to my parents, without whom

I would not be able to pursue my dreams.

SEMI-SUPERVISED ADAPTIVE CLASSIFICATION OVER DATA STREAMS

by

AHSANUL HAQUE, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2017

ACKNOWLEDGMENTS

First, I would like to offer my wholehearted appreciation and gratitude to my PhD supervisor Professor Latifur Khan – a great researcher, educator, and a wonderful human being. Professor Khan provided precious suggestions, ideas, and support at difficult times in my research and beyond. He is a role model to me for characteristics of his amazing personality, such as diligence, patience, kindness, and tranquility. Thanks to Professor Khan for guiding me throughout my graduate student life.

I would like to thank Dr. Alvaro Cárdenas, Dr. Kevin W. Hamlen, and Dr. Murat Kantarcioglu for their interest in my research, and serving on my PhD committee. Their comments helped me greatly in refining this dissertation.

This research was supported in part by awards from the National Science Foundation (NSF), the Air Force Office of Scientific Research (AFOSR), and IBM. Any opinions, findings, conclusions, or recommendations expressed are those of the authors, and do not necessarily reflect the views of the NSF, AFOSR, or IBM.

I was fortunate enough to work with a bunch of very talented friends and colleagues – Solaimani, Khaled, Sayeed, Ahmad, to name a few. I spent a wonderful five years with them. I thank them for making my journey towards the PhD degree so enjoyable and memorable. Special thanks to Swarup, for his contribution and partnership as a co-researcher in many of my research works.

Last but not the least, I would like to thank my wife, Kazi Sabrina Sonnet, for her continuous support and sharing my journey as a graduate student. I am grateful to the Almighty for whatever I have achieved in my life.

October 2017

SEMI-SUPERVISED ADAPTIVE CLASSIFICATION OVER DATA STREAMS

Ahsanul Haque, PhD
The University of Texas at Dallas, 2017

Supervising Professor: Dr. Latifur Khan, Chair

Data streams are ubiquitous in today's digital world. Efficient extraction of knowledge from these streams may help in making important decisions in (near) real-time, and unveiling hidden opportunities. Traditional data mining techniques are inadequate on the streaming data due to its inherent properties, such as infinite length, concept drift, concept evolution, limited labeled data, and covariate shift between labeled training data and unlabeled test data. In this dissertation, we study the challenges posed in data stream classification due to these properties, and propose solutions to the challenges.

A data stream is essentially an infinite flow of data. The classifier in a streaming scenario needs to be updated regularly as the underlying class boundary may change and totally new classes may emerge in the stream over time, known as the concept drift and the concept evolution problems respectively. As labeled data instances are scarce in the real-world data streams, the classifier must be trained and updated under a semi-supervised setting in order to capitalize the large portion of data that are unlabeled. Due to the semi-supervised setting, covariate shift such as sampling bias may be introduced between the training and the test distribution. An efficient data stream classification approach would consolidate for this difference in distributions. In addition to addressing these challenges, a classifier in the streaming context must be scalable for addressing the additional challenges posed by any Big Data or Internet of Things (IoT) stream.

In this dissertation, we propose four paradigms for addressing challenges in classifying data streams, namely *ECHO*, *FUSION*, *SDKMM*, and *CASTLE*. First, we propose ECHO, which is a semi-supervised approach for addressing infinite length, concept drift and concept evolution using a limited amount of labeled data. Next, we study the consequences of covariate shift in data stream classification. A covariate shift between the training and test distribution may occur due to difficulty in collecting labeled data instance, often resulting in a sampling bias. In a streaming scenario, we consider two separate streams, where one of the streams provides only labeled training data, and the other stream provides unlabeled test data. These streams of data may have covariate shift and asynchronous concept drifts among them. The second approach proposed in this dissertation, referred to as FUSION, addresses challenges in the above scenario, also known as the *Multistream* classification problem. In the third and fourth approaches proposed in this dissertation, we propose two scalable paradigms for data stream classification. The third approach, SDKMM, is a sampling-based distributed approach for addressing covariate shift between the training and the test distribution. The last approach presented in this dissertation, referred to as CASTLE, is a hierarchical ensemble classification model for data streams, where individual classifiers in the hierarchy are trained in parallel and in a distributed fashion. We theoretically analyze various properties of the proposed approaches. Moreover, we evaluate each of the above approaches using benchmark datasets, and compare them with a number of baseline approaches. Empirical results indicate the effectiveness of the proposed approaches.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION[1]

In today's connected digital world, an enormous amount of data are being generated in the form of data streams from a variety of sources – social networks, online businesses, sensors, military surveillance to name a few. It is estimated that 2.5 quintillion bytes of data are being generated everyday (Wang et al., 2014). These streams of data are important sources of knowledge that can help in designing improved strategies, and unveiling hidden opportunities. This is why the necessity of having a robust and fast technique for classifying data streams is becoming increasingly important. However, data stream classification is a challenging task due to its inherent properties. In this dissertation, we study five major challenges in data stream classification, and propose solutions to address these challenges.

The first challenge we study is the infinite length problem. Data streams are essentially infinite flows of data. Due to the infinite length, data streams cannot be stored into main memory for analyzing, e.g., labeling. Existing techniques either divide the stream into fixed-sized chunks, e.g., (Parker and Khan, 2015; Masud et al., 2011, 2010), or use gradual forgetting, e.g., (Koychev, 2000, 2002; Klinkenberg, 2004), to address the problem of infinite length. In these approaches, the classifier typically is updated after processing a chunk of

[1]

data. However, since the underlying concepts often change in data streams, deciding the proper chunk size requires *a priori* knowledge about the time-scale of change, which is not available most of the time. So, these techniques suffer from a trade-off between performance during stable periods due to redundant retraining, or delayed response to a sudden drift of concepts. Existing dynamic sliding window based approaches, e.g., (Bifet and Gavaldà, 2007; Gama et al., 2004), determine chunk boundaries dynamically by tracking any major change in error rate of the classifier, but requires true labels of all data instances.

The second challenge we consider in this dissertation is the change of underlying concepts, also known as the concept drift problem (Gama et al., 2014). The classification model needs to be updated regularly to adapt to the most recent concepts (Parker and Khan, 2015). The vast majority of concept drift researches are supervised in nature, and assume that true labels of test instances will be readily available to update the classifier as soon as they are tested. However, typically labeling requires annotation by a human expert, which is a costly and time-consuming process. More often, true labels are available only for a limited amount of data. Therefore, supervised approaches suffer in this scenario (Masud et al., 2008).

The next major challenge we consider is the emergence of a totally new class, also known as the novel class detection or concept evolution problem. In real-world data streams, such as intrusion detection, text classification or fraud detection, the number of classes is not fixed, and concept evolution may occur at any time in the stream. If concept evolution is not addressed timely, the classifier classifies the instances from the new class as existing classes, thereby the classification error increases. Unfortunately, this challenge has been ignored by most state-of-the-art techniques.

The fourth challenge we discuss in this dissertation is the scarcity of labeled data in data streams. A vast majority of the existing techniques for data stream mining are fully-supervised in nature, as they assume that true labels for data instances would be available as soon as prediction is done. As mentioned before, providing true labels for data instances is a

time-consuming process, and requires human effort. Therefore, although data is abundant in data streams, typically most of them are unlabeled. Fully-supervised models do not utilize this enormous amount of unlabeled data, therefore are not suitable in data stream mining scenario.

The fifth and the final challenge we consider is the presence of sampling bias between the training and the test distribution. A fundamental assumption in data mining, known as the "stationary distribution assumption", is that both the training and test data represent the same data distribution (Zadrozny, 2004). This assumption may be violated in a real-world semi-supervised learning setting, where a few data instances are sampled for which true labels are provided. Traditional classifiers that are trained on the biased training data, without accounting for the sampling bias, greatly suffer in this setting. This important challenge also has not been addressed by the state-of-the-art data stream mining approaches.

In addition to the above challenges, scalability is very important in analyzing *Big Data* or IoT data streams. Big Data is characterized by four V's, i.e., Volume, Velocity, Variety, and Veracity. At present, the data produced per day is in quintillion bytes range and is supposed to increase even more in nearby future. This grand scale and the rise of data outstrip traditional data mining techniques (Zikopoulos et al., 2011). Any system dealing with Big Data Stream receives this high volume data continuously. Moreover, data in Big Data Stream may have different types of features and unstructured data. This is why, usually data needs to undergo normalization preprocessing before applying classification methods which is an extra overhead. Thus, traditional data mining systems fall short in classifying this huge amount of data that is constantly in motion.

In this dissertation, we present four frameworks, where the first two frameworks extensively address the five challenges in data stream classification. The latter two frameworks have been designed with scalability in mind to address the challenges posed by Big Data streams. Next, we briefly discuss the proposed frameworks.

ECHO. In the first approach, we address the challenges of infinite length, concept drift, and concept evolution using a limited amount of labeled data. More specifically, we propose a sliding window based framework, *ECHO* (Haque et al., 2016) that maintains an ensemble of classifier models, each trained on a dynamically determined chunk. Each of these models is a semi-supervised clustering based k-NN type model. As soon as a new data instance arrives in the stream, in addition to classifying the instance, ECHO estimates the confidence in the classification. We propose two estimators, namely *Association* and *Purity* to estimate confidence. ECHO detects any concept drift by tracking any significant change in confidence estimates using a change detection technique (CDT). Monitoring change in the distribution of classifier confidence is the bottleneck in ECHO in terms of execution time. Therefore, in order to improve performance, we propose a few strategies for selectively executing change detection. Moreover, we propose a recursive formula, and use dynamic programming to calculate change detection scores. These greatly reduce the execution time of our proposed framework, and make it practically usable.

Once a concept drift is detected, ECHO updates the classifier using a limited amount of labeled data. It uses the classifier confidence for selecting important instances for updating the model, and requests label for those instances only. ECHO also incorporates a novel class detector. It detects outliers in the data stream, and stores them in a buffer. The novel class detector periodically examines the buffer for detecting the presence of a novel class. It detects a novel class if the outliers have enough cohesion among themselves and separation from the existing class instances. Empirical results show that ECHO achieves competitive classification accuracy, if not better, compared to the fully-supervised baseline approaches, despite using a limited labeled data for training.

FUSION. We propose the next approach, referred to as *FUSION* (Haque et al., 2017), for handling covariate shift over data streams. In this work, we focus on the multistream classification problem, which involves two independent non-stationary data generating processes.

One of them is the source stream that continuously generates labeled data. The other one is the target stream that generates unlabeled test data from the same domain. The distribution represented by the source stream data is biased compared to that of the target stream. Moreover, these streams may have asynchronous concept drifts between them. The multistream classification problem is to predict the class labels of target stream instances by utilizing labeled data from the source stream. This kind of scenario is often observed in real-world applications due to the scarcity of labeled data problem. In this work, we propose an efficient solution for multistream classification by fusing drift detection into online data shift adaptation.

Concretely, we estimate a weight for each training instance, so that the weighted training distribution closely imitates the test distribution. FUSION models the instance weights using a *Gaussian* kernel model, and updates it online. It trains a classification model based on the weighted training instances and detects an asynchronous concept drift if there is a significant difference between the weighted training and the test distribution. FUSION updates the classification model following detection of any concept drift. We study the theoretical convergence rate and computational complexity of the proposed approach. Moreover, empirical results on benchmark datasets indicate significantly improved performance over the baseline methods.

SDKMM. As mentioned before, scalability is an important consideration while designing a classification technique for Big Data or IoT streams. In the third paradigm presented in this dissertation, referred to as *SDKMM* (Haque et al., 2016), we design a scalable and distributed approach for handling covariate shift between the training and the test distribution. As mentioned before, limited access to supervised information may forge scenarios in real-world data mining applications, where the distributions represented by training and test data are not same, but related by a covariate shift, i.e., having equal class conditional distribution

with unequal covariate distribution. This difference in the distributions must be accounted for in order to build an efficient classifier.

A number of approaches are available in the literature that address the covariate shift problem by estimating density ratio as the importance weight for the training instances. However, these approaches in general have high time complexity, which limits their application in real-time applications, such as data stream classification. In this work, we focus on one such method, referred to as Kernel Mean Matching (KMM). It has time complexity cubic in the size of training data, which is computationally impractical for large or streaming datasets. Our proposed approach, SDKMM is a sampling-based algorithm to address the limited scalability problem of KMM. In this approach, first, we generate a number of random sub-samples from the original training data. The number of samples is determined in such a way that each training instance is included in at least one of the sub-samples with a very high probability. In addition to sampling the training data, we also split the test data. Then, we consider each possible pair of training sample and test split as a train-test component. Next, we apply the KMM on each train-test component independently. Finally, we aggregate the weights for each training instance that are estimated from different train-test components, in order to estimate the final weight for that training instance.

Importantly, we show that the approach is highly parallelizable, and therefore propose a distributed algorithm for estimating training instance weights efficiently using Spark. Experiment results on benchmark datasets show that the proposed approach achieves competitive estimation accuracy within much lower execution time compared to the KMM algorithm. Moreover, it indicates that larger size of training data results into a higher accuracy with minimal effect on the execution time of the proposed approach.

CASTLE. In the last approach presented in this dissertation, we propose a scalable multi-tiered ensemble-based method for Big Data stream classification. We refer to this approach

as *CASTLE* (Haque et al., 2014). More specifically, CASTLE builds separate ensemble classifier for each class that is present in the stream. Each class-based ensemble consists of feature-based models. CASTLE induces a *Naive Bayes* model for each non-numeric feature, and an *AdaBoost* model for each numeric feature. This hierarchical structure is maintained and updated after the arrival of each chunk of data. As soon as a test instance arrives in the stream, first, votes from individual feature-based models are aggregated for collecting class-based ensemble votes. Finally, the class-based ensemble votes are aggregated for final prediction.

The bottleneck of this approach is building AdaBoost ensembles for each of the numeric features. Therefore, it faces scalability issue if the number of numeric features is large. However, we observe that the process of forming AdaBoost ensemble for a feature is completely independent of forming AdaBoost ensemble for other features. So, there is scope for parallel training and maintenance of these AdaBoost ensembles. In CASTLE, we propose three different MapReduce-based approaches to form AdaBoost ensembles for different numeric features in parallel. Each of the approaches executes only one MapReduce job for building all the AdaBoost ensembles needed per data chunk. First two approaches build all the feature-based AdaBoost ensembles under a particular class in the same Map task. The third approach does not have this constraint. In the third approach, the task of building feature-based AdaBoost ensembles is distributed among different Map tasks regardless of class information.

The rest of the dissertation is organized as follows: In Chapter 2, we present the necessary background information, and discuss some of the related work. We present our proposed approaches chronologically in Chapters 3 - 6. Finally, we conclude the discussion with a summary and future directions in Chapter 7.

CHAPTER 2

BACKGROUND AND RELATED WORK[1]

In this chapter, we formally define the data stream and the Multistream classification problem. Moreover, we briefly discuss related work in this domain.

2.1 Data Stream Classification

A data stream is a continuous sequence of data instances. Let the set of first t instances received from the data stream be $\left\{\left\langle \boldsymbol{x}^{(1)}, y^{(1)}\right\rangle, \ldots, \left\langle \boldsymbol{x}^{(t)}, y^{(t)}\right\rangle\right\}$, where $\left\langle \boldsymbol{x}^{(i)}, y^{(i)}\right\rangle$ is the i^{th} instance. Ignoring the index, in each data instance $\langle \boldsymbol{x}, y\rangle$, $\boldsymbol{x} \in \mathcal{D}^v$ denotes a v dimensional feature vector from domain $\mathcal{D}$. On the other hand, $y \in \{1, \ldots, l, \ldots, L\}$ denotes the true class label of the instance, where l is the number of classes observed so far in the stream, and L is the number of possible classes, which is unknown. Let $\hat{y}^{(t)}$ be the prediction for $\left\langle \boldsymbol{x}^{(t)}, y^{(t)}\right\rangle$, the prediction is correct if $y^{(t)} = \hat{y}^{(t)}$. $\left\langle \boldsymbol{x}^{(t)}, y^{(t)}\right\rangle$ is called a labeled data instance if $y^{(t)}$ is provided. The classification task for each new instance in the data stream, $\left\langle \boldsymbol{x}^{(i)}, \cdot\right\rangle$, is to predict the class label $y^{(i)}$ using the set of instances $\mathcal{L}^{(i-1)} \cup \mathcal{U}^{(i-1)}$, where $\mathcal{L}^{(i-1)}$ and $\mathcal{U}^{(i-1)}$ denote set of labeled and unlabeled instances from $\left\{\left\langle \boldsymbol{x}^{(1)}, y^{(1)}\right\rangle, \ldots, \left\langle \boldsymbol{x}^{(i-1)}, y^{(i-1)}\right\rangle\right\}$ respectively.

---

[1]

2.1.1 Related Work

The challenges in data stream classification stem from the inherent properties of data streams. In this section, we briefly discuss existing research works that address challenges in data stream classification, i.e., infinite length, concept drift, limited labeled data, and the emergence of novel classes.

Infinite Length

Data streams are divided into fixed-size chunks by most state-of-the-art approaches to address the infinite length problem (Aggarwal and Yu, 2010; Masud et al., 2011; Parker and Khan, 2015; Widmer and Kubat, 1996). These approaches use *abrupt forgetting* as only the latest chunk of data instances is kept in the memory. Typically to address concept drift, each fixed-size chunk is used to retrain or update the classifier as soon as all the instances in the chunk are labeled. However, setting the fixed size for chunks is very difficult in the context of an evolving data stream. Approaches using a fixed chunk size cannot capture the concept drift immediately if the chunk size is too large, or suffer from unnecessary frequent retraining during stable time periods if the chunk size is too small (Bifet and Gavaldà, 2007).

Gradual forgetting is used by (Koychev, 2000, 2002; Klinkenberg, 2004), which is a full memory approach for defining a window of instances for learning. In gradual forgetting, each example is associated with a weight rather than discarding it from the memory completely. The weight typically is assigned based on the age of that data instance assuming that importance of an instance should decrease with its age. Various decay techniques are used in the literature. For example, linear decay techniques are used in (Koychev, 2000, 2002), and an exponential decay is used in (Klinkenberg, 2004). However, finding the perfect decay function is a challenge if information on the time-scale of change is not available (Bifet and Gavaldà, 2007). In our proposed approach (ECHO), we determine the chunk size dynamically by using an explicit change detection technique (CDT).

Concept Drift Detection

In data stream mining, change detection techniques (CDTs) are used either to detect any change in the input data distribution, or to detect any change in the classifier feedback. Several methods, e.g., (Song et al., 2007; Kuncheva and Faithfull, 2012; Ross et al., 2011) exist to detect change of the input data distribution in a data stream. However, detecting a change of distribution in a multi-dimensional space is considered as a hard problem (Harel et al., 2014). It introduces error while finding changes in the multidimensional input space, hence not efficient in the context of data stream mining. In our proposed approach, we focus on detecting changes in one-dimensional classifier confidence.

Various CDTs have been proposed in the literature to detect concept drift from any significant change in the classifier feedback. *Adwin* (Bifet and Gavaldà, 2007) is a sliding window based technique which determines the size of the window according to the rate of change observed from the window data itself. The approach proposed in (Gama et al., 2004) detects a change when the error rate over the whole current window significantly exceeds the lowest error rate recorded. The approach proposed in (Cieslak and Chawla, 2007) exploits *Kruskal Wallis* analysis and *Kolmogorov Smirnov* tests to detect changes. An approach based on obtaining statistics from the loss distribution of the learning algorithm by reusing the data multiple times via re-sampling has been proposed in (Harel et al., 2014). Concept drift detection in (Alippi et al., 2013) contains two CDTs based on Intersection of Confidence Intervals (ICI), one to detect any change in the input data distribution and another to detect any change in the classifier error rate. Considering the high volume and speed of today's data streams, running two CDTs after testing each instance is expensive. Another ICI based approach is ACE (Nishida et al., 2005). All of the above CDTs detect changes in the classifier error rate, requiring true labels of all data instances be readily available. This assumption is not practical in the context of data streams (Masud et al.,

2008). Our proposed approach ECHO detects changes in classifier confidence, which does not require any supervised information.

Limited Labeled Data

Existing semi-supervised techniques use active mining, computational geometry, or random sampling for selecting important data instances for labeling. The approaches proposed in (Fan et al., 2004; Zhu et al., 2007, 2010; Masud et al., 2010) are examples that use active learning. The approach proposed in (Fan et al., 2004) estimates the error of a decision tree model based on a few statistics. If the estimated error is significantly high, it randomly samples a small number of instances from the data stream for labeling. The size of the sample depends on the cost for labeling. A classifier-ensemble based active learning framework has been proposed in (Zhu et al., 2007, 2010). The authors of these papers argue that the variance of an ensemble classifier is directly related to its error rates. Therefore, the instances that contribute most to the variance of the ensemble classifier are considered as important, and true labels are requested for only these instances. The approach proposed in (Masud et al., 2010) selects only those data instances for labeling for which the expected classification error is high. Though active learning approaches are useful for selecting important data instances, these typically add an extra overhead to the overall learning algorithm. Our proposed approach reuses the same confidence scores calculated during prediction to select instances for labeling without adding any extra overhead.

Apart from active learning techniques, the approaches presented in (Aggarwal, 2006; Efraimidis and Spirakis, 2006; Al-Kateb et al., 2007; Masud et al., 2008) investigate various random sampling based techniques over data streams. However, though random sampling is useful for reducing labeling cost to some extent, it often misses the instances important for accommodating change of class boundaries. An approach based on computational geometry, called COMPOSE, has been proposed in (Dyer et al., 2014). However, it can only address gradual (limited) drift, rather than abrupt drift.

Novel Class Detection

Various clustering-based novel concept detection techniques for data streams have been proposed in (Spinosa et al., 2008; Hayat and Hashemi, 2010). In these approaches, the instances which are not explained by the current decision model are labeled with an *unknown* profile. If a sufficient number of instances with the *unknown* profile can be found, clustering is applied on these instances. Valid clusters are evaluated as an extension of the normal class or a novelty. Therefore, these are *single class* novelty detection methods, where authors assume that there is only one *normal* class and all other classes are novel. Thus, it is not suitable for a multi-class environment.

A novel class detection algorithm in a multi-class environment has been proposed in (Masud et al., 2011), where each new test instance is considered as an outlier if it falls outside of decision boundary of the ensemble classifier. A novel class is detected if a sufficient number of outliers have high cohesion among themselves and enough separation from the instances of existing classes. However, this approach divides the data stream into fixed-size chunks, hence suffers from the trade-off discussed earlier. Unlike these approaches, our proposed framework detects novel classes in a multi-class environment using dynamically-determined chunk boundaries.

Three general strategies for transforming block-based ensembles into online learners are investigated in (Brzezinski and Stefanowski, 2014). However, it does not investigate strategies related to concept evolution. An evaluation methodology for multi-class novelty detection algorithms has been presented in (Faria et al., 2013). In this dissertation, we assume that only one novel class may appear at a time in the data stream. So, we use the traditional novel class detection performance evaluation metrics that have been used by most state-of-the-art approaches.

2.2 Multistream Classification

In this section, first we briefly discuss covariate shift adaptation. Next, we introduce the Multistream classification problem and point out the challenges in it. Finally, we briefly discuss the prior work on this problem.

2.2.1 Data Shift Adaptation

A fundamental assumption in data mining, known as the "stationary distribution assumption", is that both the training and test data represent the same data distribution (Zadrozny, 2004). However, it may be violated in real-world applications due to limited supervision, or lack of control over the data gathering process. Traditional techniques based on this assumption greatly suffer in this scenario.

Addressing an arbitrary difference between training and test distribution is a very difficult problem (Huang et al., 2006). Hence, most approaches addressing this challenging assume that the training and test data distributions, denoted by $P_{tr}(\cdot)$ and $P_{te}(\cdot)$ respectively, are related by a covariate shift assumption. More specifically, the relationship between the training and test data distributions is such that $P_{tr}(y|\boldsymbol{x}) = P_{te}(y|\boldsymbol{x})$ and $P_{tr}(\boldsymbol{x}) \neq P_{te}(\boldsymbol{x})$, where $\boldsymbol{x}$ and y denote the set of covariate values and label of a data instance respectively.

In general, covariate shift between training and test data distributions is accounted by computing an importance weight, $\beta(\boldsymbol{x}) = \frac{P_{te}(\boldsymbol{x})}{P_{tr}(\boldsymbol{x})}$, for each training instance $\boldsymbol{x}$, and using them in the learning process. KMM (Huang et al., 2006), KLIEP (Sugiyama et al., 2008), and uLSIF (Kanamori et al., 2009) are among the techniques that are available in the literature for handling covariate shift. However, these approaches work only on fixed-size training and test data. Although Kawahara and Sugiyama extended KLIEP for direct online density ratio estimation and sequential change-point detection (Kawahara and Sugiyama, 2012), it works on a single stream of data, where the set of recent training/reference and test data instances are determined by a sliding window. In this dissertation, we consider covariate shift in the

Multistream classification problem, which considers two streams of data, where new data instance may arrive arbitrarily at any stream.

2.2.2 Problem Statement

The vast majority of existing data stream classification techniques make two strong assumptions. First, true labels of data instances along the stream become available soon after prediction, which are then used for updating the existing classifier. In practice, labeled data are scarce as obtaining true labels is costly (Haque et al., 2016). Furthermore, it is assumed that the training and the test data represent the same distribution. As mentioned in Section 2.2.1, this assumption may also be violated due to scarcity of labeled data. This may induce a sampling bias between the training and test data distribution. A new problem setting, called *Multistream Classification*, has been introduced in (Chandra et al., 2016), where two data streams over the same domain are considered by relaxing the above assumptions.

Let us consider that two different but related processes generate data continuously from a domain $\mathcal{D}$. The first process operates in a supervised environment, i.e., all the data instances that are generated from the first process are labeled. On the contrary, the second process generates unlabeled data from the same domain. The stream of data generated from the above processes are called the *source stream* and the *target stream*, and are denoted by $\mathcal{S}$ and $\mathcal{T}$ respectively. Each data instance is denoted by $(\boldsymbol{x}, y)$, where $\boldsymbol{x} \in \mathcal{D}^d$ is the set of d covariates, and y is the true label of the instance. As mentioned before, only $\boldsymbol{x}$ for each instance is observed in $\mathcal{T}$, where $\mathcal{S}$ also provides y in addition to $\boldsymbol{x}$ for each instance. The *Multistream Classification* is defined as follows.

Definition 1. *Let $\mathbf{X}_S \in \mathcal{D}$ be a set of d-dimensional vectors of covariates and $\mathbf{Y}_S$ be the corresponding class labels observed on a non-stationary stream $\mathcal{S}$. Similarly, let $\mathbf{X}_T \in \mathcal{D}$ be a set of d-dimensional vectors of covariates observed on another independent non-stationary stream $\mathcal{T}$. Let P_S and P_T denote covariate distribution from $\mathcal{S}$ and $\mathcal{T}$ respectively. Data*

generated from S and T are related by a covariate shift, i.e., $P_S(y|\boldsymbol{x}) = P_T(y|\boldsymbol{x})$ and $P_S(\boldsymbol{x}) \neq P_T(\boldsymbol{x})$. Construct a classifier $\mathcal{M}$ that predicts class label of $\boldsymbol{x} \in \mathbf{X}_T$ using $\mathbf{X}_S$, $\mathbf{Y}_S$ and $\mathbf{X}_T$.

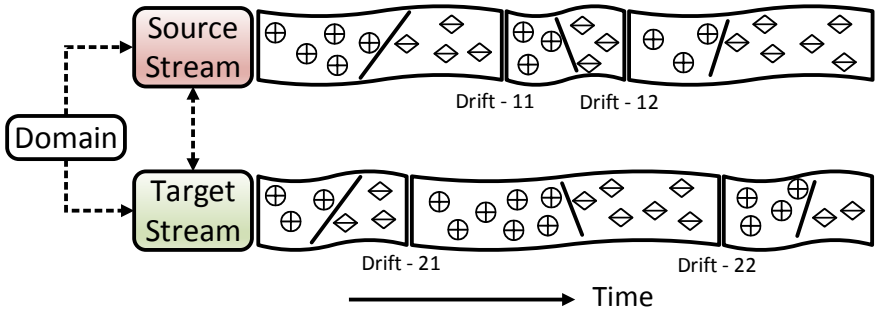


Figure 2.1: An example illustrating asynchronous data drifts

Challenges

As mentioned before, we assume that initially at time t, data distributions of $\mathcal{S}$ and $\mathcal{T}$ are related by a covariate shift, i.e., $P_S^{(t)}(y \mid \boldsymbol{x}) = P_T^{(t)}(y \mid \boldsymbol{x})$ and $P_S^{(t)}(\boldsymbol{x}) \neq P_T^{(t)}(\boldsymbol{x})$. However, this assumption may not be true at time $r > t$ due to the non-stationary nature of data streams. The reason is that, individually within each data stream, the conditional probability distribution may change over time due to concept drift, i.e. $P^{(t)}(y \mid \boldsymbol{x}) \neq P^{(r)}(y \mid \boldsymbol{x})$. Similarly, a *covariate shift*, i.e., a change in the covariate distribution, may also occur with time in each stream.

With two independent non-stationary processes generating data continuously from $\mathcal{D}$, the effect of a drift may be observed at different times on these streams, referred to as *asynchronous* drift. Figure 2.1 illustrates asynchronous data drifts between the source and the target stream. In this illustration, four independent data drifts occur at different times on $\mathcal{S}$ and $\mathcal{T}$. The drifts *Drift-11* and *Drift-21* are similar, and occur on the source and target stream respectively but at different times, which by definition is an asynchronous drift.

Similarly, *Drift-12* and *Drift-22* represent another example of an asynchronous concept drift. This type of scenario may occur in a real-world application when the factor causing a drift affects the streams at different times. To summarize, the main challenges in Multistream classification are handling data shift and asynchronous concept drift between the source and the target stream simultaneously and efficiently.

2.2.3 Prior Work

MSC (MultiStream Classifier) (Chandra et al., 2016) is the first framework that was proposed for Multistream classification. MSC uses Kernel Mean Matching (KMM) for covariate shift adaptation between the source and the target distributions by weighing labeled source instances in the learning process. However, since source or target stream may have asynchronous concept drifts, weights of the training instances may become outdated if there is a drift in any of these streams. To address this challenge, an ensemble of classifiers is maintained. The ensemble contains classifiers built on both source stream and target stream data. The ensemble is updated by calculating new weights for recent instances from the source data stream if a concept drift is detected in either of these streams of data. Concept drift is detected by monitoring any significant change in classifier feedback following similar approach proposed in (Haque et al., 2016). Since source stream generates all labeled data, a concept drift in it is detected by tracking any major change in classifier error rate. On the contrary, classifier confidences are monitored to detect concept drifts in the target stream generating only unlabeled data.

While being the first approach for addressing the challenges of Multistream classification, MSC suffers from a number of limitations. First, it uses an ensemble classifier consisting of models trained on both source and target data. Once there is a concept drift in either stream, the ensemble is updated using a model trained on data from the corresponding stream. As a result, the overall ensemble management is complex in MSC. Second, it executes change

detection algorithm for detecting concept drift after receiving any new data instance either in the source or target stream. The change detection algorithm used in MSC has time complexity cubic in the number of data instances in the current window, which makes MSC extremely slow. Third, once a concept drift is detected in any of the streams, MSC uses Kernel Mean Matching for covariate shift adaptation, which also has cubic time complexity. The above overheads adversely affect the performance of MSC. In this dissertation, we propose a framework FUSION for addressing the challenges in the Multistream classification problem efficiently.

CHAPTER 3

ECHO: DATA STREAM CLASSIFICATION USING LIMITED LABELED DATA[1]

Data stream classification is a challenging task because of its inherent properties as mentioned in Chapter 1. Due to the infinite length of data, it is not practical to store all the training data first, and then execute a multi-pass learning algorithm like traditional data mining. Instead, data stream classification demands single-pass limited-memory learning algorithm that can adapt to any change in the underlying concepts quickly. One strategy of updating the classifier could be dividing the stream of data into fixed-size chunks and updating the model periodically at the end of each chunk. This strategy suffers from important shortcomings. As time scale of change is not often available, based on the size of the chunks, the classifier may become outdated, or may experience unnecessary updates.

Therefore, it is clear that the best way of keeping the classifier updated is to detect any change in the underlying concepts, and update the classifier only when it is necessary. It again poses two important challenges, i.e., how to detect a concept drift efficiently, and which instances should be used to update the classifier. Please note that despite providing abundant data instances, labeled data instances are scarce in data streams. In this chapter, we propose a framework, referred to as *ECHO* ((Efficent Concept Drift and Concept Evolution Handling over Stream Data)) (Haque et al., 2016), which answers the above questions using only a limited amount of labeled data.

ECHO maintains an ensemble of classifier models, each trained on a dynamically determined chunk. However, unlike other existing sliding window techniques, it does not depend

[1] ©2016 Association for the Advancement of Artificial Intelligence. Portions Adapted, with permission, from A. Haque, L. Khan, and M. Baron, "SAND: Semi-Supervised Adaptive Novel Class Detection and Classification over Data Stream," AAAI Conference on Artificial Intelligence, North America, pp. 1652-1658, February 2016; ©2016 IEEE. Portions Adapted, with permission, from A. Haque, L. Khan, M. Baron, B. Thuraisingham, and C. Aggarwal, "Efficient Handling of Concept Drift and Concept Evolution over Stream Data," IEEE International Conference on Data Engineering (ICDE), pp. 481-492, May 2016.

on the error rate of the classifier, which requires true labels for all data instances. Rather, it estimates classifier confidence in classifying each test instance, stores in the sliding window, and tracks any significant change in confidence estimates using a change detection technique (CDT). A change in the classifier confidence indicates the occurrence of a concept drift. Therefore, if there is a significant change in confidence estimates, a concept drift is detected, a chunk boundary is determined, and the classifier is updated using a few labeled instances from the latest chunk.

In order to estimate the confidence of the classifier in ECHO, we propose two estimators, namely *Association* and *Purity*. These estimators are generic as these can be used to estimate the confidence of any clustering based model. We theoretically justify the use of these estimators, and show that confidence estimates decrease consistently in presence of a concept drift. A naive strategy is to invoking the change detection module after estimating classifier confidence on each test instance. A change detection score is calculated each time based on values currently stored in the sliding window. If this score is greater than a pre-fixed threshold, a concept drift is detected. This exhaustive invocation of the change detection module is computationally expensive. Instead of this, we propose a recursive formula, and use dynamic programming to calculate change detection scores. Along with this, we propose selective and sporadic execution of the change detection module based on confidence score on the most recent test instance. These greatly reduce the execution time of ECHO, and makes it practically usable.

ECHO uses only a limited amount of labeled data instances from the latest chunk for updating the classifier. Motivated by the principle of Uncertainty Sampling (Settles, 2009), it selects data instances for labeling based on the classifier confidence score calculated during testing. If the confidence in classifying an instance is low, the true label for that instance is requested. Otherwise, the label predicted by the classifier is accepted as the label of that instance. Thus, training data is formed only using a partially labeled data. A new model

is then trained on the training data to replace the oldest model in the ensemble. Thus, the ensemble is updated using a limited amount of labeled data without any extra overhead by simply using the confidence scores calculated for dynamic sliding window management.

ECHO also incorporates a novel class detector for the purpose of handling concept evolution. The detector assumes the appearance of a single novel class at a time in the data stream. Any instance falling outside the decision boundaries of all models in the ensemble is identified as an outlier. The framework interprets the presence of a sufficiently large number of outliers with strong cohesion among themselves as the emergence of a novel class.

Primary contributions of this work are as follows:

1. We present a technique to estimate the confidence of any clustering based classifier with theoretical justification.

2. We analytically show that classifier confidence decreases consistently in presence of a concept drift. Therefore, we design a suitable change detection technique (CDT) to detect any significant change in classifier confidence.

3. We use the confidence scores to select a limited number of data instances from the latest chunk for labeling. Following detection of a concept drift, the classifier is updated using this limited amount of labeled data instances without any extra overhead.

4. We present a semi-supervised framework (ECHO) that uses dynamically determined chunks both for classification and novel class detection. We use dynamic programming, and propose sporadic and selective execution of the change detection module to reduce the execution time of ECHO to a great extent.

5. We evaluate ECHO on several benchmark and synthetic datasets. Experiment results indicate that ECHO shows good performance in terms of both classification accuracy and execution time.

3.1 The Proposed Approach

High level workflow of ECHO is depicted in Figure 3.1. The framework has four modules, i.e., *Classification*, *Novel Class Detection*, *Change Detection*, and *Update*. ECHO maintains

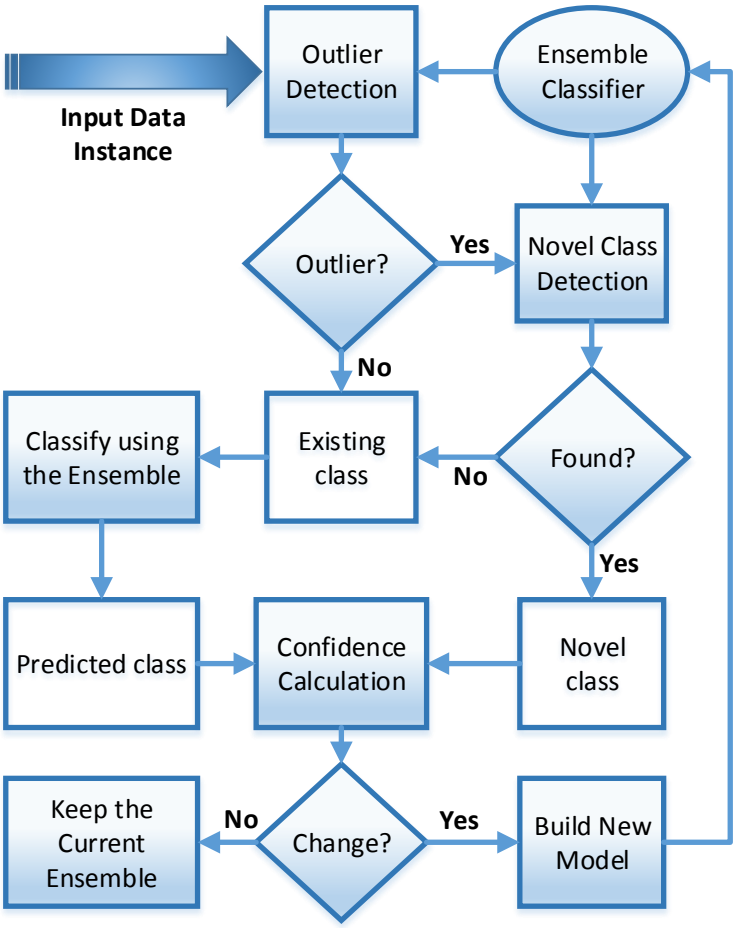


Figure 3.1: High level workflow of ECHO

an ensemble $\mathcal{M}$ of t classification models, and a dynamic window W containing classifier confidence estimates in predicting labels of recent test data instances. Let $\{M_1, ..., M_t\}$ be the models in the ensemble. At the beginning, the ensemble classifier contains models trained on the initial training data, also known as warm-up period data. Once the warm-up period

is over, each incoming instance in the data stream is first examined to determine whether it is an outlier or not. It detects an instance as an outlier if the instance falls outside of the decision boundary of the ensemble classifier. If the instance is not an outlier, it is classified as instance of an existing class using majority voting among the models in the ensemble. On the contrary, if the instance is an outlier, it is temporarily stored in a buffer. When there are enough instances in the buffer, the *Novel Class Detection* module is invoked. We define a class as *novel class* if none of the models in the ensemble has been trained with instances from that class. If a novel class is detected, the instances of the novel class are tagged accordingly. Otherwise, the instances in the buffer are considered as from existing classes and classified using the current ensemble classifier. As soon as any test instance arrives, ECHO along with predicting the label of the instance, also estimates the confidence in the prediction. It keeps monitoring classifier confidences on recent instances. ECHO detects a concept drift if there is a significant change in confidence scores. A new model is trained on the recent instances, and the ensemble classifier is updated if a concept drift is detected.

As mentioned before, ECHO uses a sliding window, denoted by W, for storing and monitoring classifier confidence on recent instances. Sliding window management of ECHO is depicted in Figure 3.2. As discussed before, if a test instance is inside the decision boundary of the ensemble classifier, or is not detected as an instance from a novel class, ECHO classifies it by taking the majority vote from the ensemble. At the same time, it estimates the confidence of each model in this classification using a few estimators. These model confidences are combined together to calculate the overall confidence of the ensemble classifier. ECHO stores recent instances along with corresponding confidence scores in the sliding window W. Following insertion of confidence estimates, ECHO selectively invokes the *change detection* module to search for any significant change of distribution among the scores saved in W. If there is such a change, ECHO detects a concept drift, and W is updated. Moreover, a few instances are selected based on the confidence scores for which the actual labels are

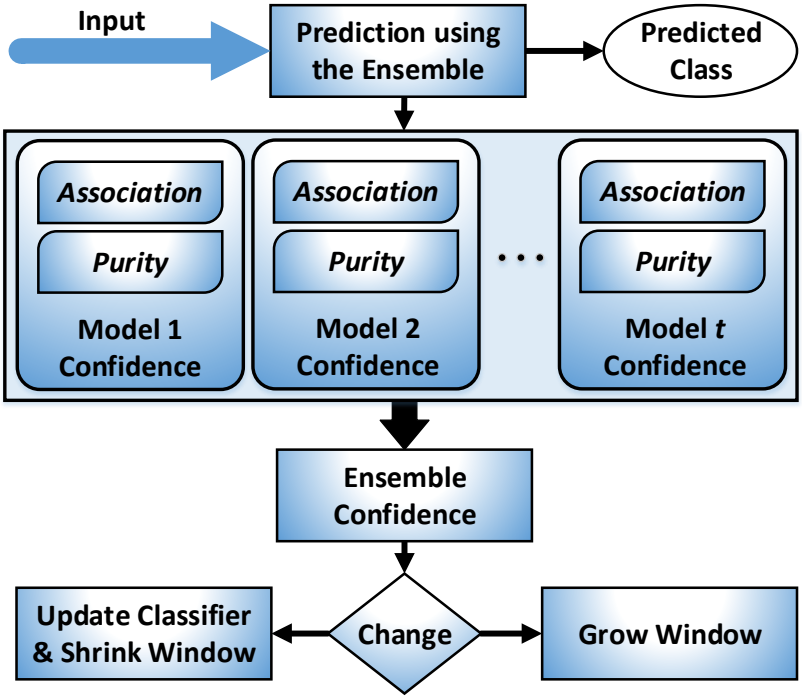


Figure 3.2: Sliding window management of ECHO

requested. Finally, the ensemble classifier is updated following a concept drift by training a new model on a limited amount of labeled data, and by replacing the oldest model in the ensemble by it. Based on the memory resource available, we set a maximum allowable size for W denoted by V_m. If W grows beyond V_m, a chunk boundary is determined, and both $\mathcal{M}$ and W are updated. Later in this chapter, we discuss classification, novel class detection, and change detection modules. Table 3.1 contains a list of frequently used symbols.

3.2 Training and Classification

Each model in the ensemble, M_i, $i \in 1 \ldots t$, is a k-NN type model. However, unlike k-NN, raw data points are not stored in the model. Rather, a number of clusters are built using any clustering algorithm, e.g., K-means, DBSCAN (Ester et al., 1996) etc. We use an impurity based K-means algorithm (discussed in Section 3.5) to build the clusters in our experiments.

Table 3.1: Commonly used symbols and terms

$\mathcal{M}$: The ensemble classifier	$\mathcal{C}^{(\boldsymbol{x})}$: Confidence in classifying $\boldsymbol{x}$
M_i : i^{th} model in $\mathcal{M}$	τ : Classifier confidence threshold
t : Number of models in the ensemble	$\mathcal{U}$: Set of unlabeled training data
$\mathcal{L}$: Set of labeled training data	$\mathcal{A}_i$: *Association* of M_i
h_{ip} : p^{th} pseudopoint in M_i	$\mathcal{P}_i$: *Purity* of M_i
$\boldsymbol{x}^{(k)}$: k^{th} instance in stream	$y^{(k)}$: True label of instance $\boldsymbol{x}^{(k)}$
$D_{ip}(\boldsymbol{x})$: Distance of instance $\boldsymbol{x}$ from h_{ip}	$\hat{y}_i^{(k)}$: Predicted label of instance $\boldsymbol{x}^{(k)}$ by M_i
$\mathcal{L}_{ip}(c)$: Frequency of class c in h_{ip}	W : Dynamic sliding window
$R(h_{ip})$: Radius of h_{ip}	V_m : Maximum allowable size for W

We assume that only a portion of data instances will be labeled. Once the clusters are created, the raw data points are discarded after summaries (mentioned as *pseudopoint*s) of the clusters are saved. Therefore, each model is a collection of K pseudopoints. Summary of a cluster, i.e., a pseudopoint contains the centroid, radius, and the number of data points belonging to each of the classes (referred to as frequencies). The radius is equal to the distance between the centroid and the farthest data point in the corresponding cluster.

Each pseudopoint corresponds to a "hypersphere" in the feature space with a corresponding centroid and radius. The *decision boundary* of a model M_i is the union of the feature spaces encompassed by all pseudopoints in M_i. The decision boundary of the ensemble $\mathcal{M}$ is the union of decision boundaries of all models $M_i \in \mathcal{M}$.

If a test instance $\boldsymbol{x}$ is inside the decision boundary of $\mathcal{M}$, it is classified using each $M_i \in \mathcal{M}$, $i \in 1 \ldots t$ as follows. Let $h \in M_i$ be the pseudopoint whose centroid is the nearest from $\boldsymbol{x}$. The predicted class of $\boldsymbol{x}$ is the class that has the highest frequency in h. The data point $\boldsymbol{x}$ is classified using the ensemble $\mathcal{M}$ by taking the majority vote among all classifiers.

3.3 Novel Class Detection

If a test instance $\boldsymbol{x}$ falls outside of the decision boundary of $\mathcal{M}$, it is declared as a filtered outlier, or *F-outlier*. The principle of the novel class detection is that an instance should be closer to instances from the same class (cohesion), and farther apart from instances from

other classes (separation). Since any *F-outlier* falls outside of the decision boundary, it is far away from the existing class instances. So, it satisfies the separation property. If it satisfies the cohesion property also, i.e., it is close to other F-outlier instances, ECHO declares emergence of a novel class. To examine this, such instances are stored in a buffer. This buffer is periodically examined by the *novel class detection* module to observe whether there are enough instances in the buffer that are close to each other. This is done by computing the q-Neighborhood Silhouette Coefficient, or q-NSC(Masud et al., 2011). This is defined based on q, c-neighborhood of an *F-outlier* $\boldsymbol{x}$ ($q, c(\boldsymbol{x})$ in short), which is the set of q instances from class c that are nearest to $\boldsymbol{x}$. Here q is a user-defined parameter.

Let $\bar{D}_{c_{out},q}(\boldsymbol{x})$ be the mean distance of an *F-outlier* $\boldsymbol{x}$ to its q-nearest *F-outlier* neighbors. Also, let $\bar{D}_{c,q}(\boldsymbol{x})$ be the mean distance from $\boldsymbol{x}$ to its $q, c(\boldsymbol{x})$, and let $\bar{D}_{c_{min},q}(\boldsymbol{x})$ be the minimum among all $\bar{D}_{c,q}(\boldsymbol{x})$, $c \in$ {Set of existing classes}. In other words, q, c_{min} is the nearest existing class neighborhood of $\boldsymbol{x}$. Then q-NSC (Masud et al., 2011) of $\boldsymbol{x}$ is given by:

$$q\text{-}NSC(\boldsymbol{x}) = \frac{\bar{D}_{c_{min},q}(\boldsymbol{x}) - \bar{D}_{c_{out},q}(\boldsymbol{x})}{max(\bar{D}_{c_{min},q}(\boldsymbol{x}), \bar{D}_{c_{out},q}(\boldsymbol{x}))} \tag{3.1}$$

q-NSC considers both cohesion and separation, and yields a value between -1 to $+1$. A positive value of q-NSC indicates that the *F-outliers* are closer to other *F-outlier* instances stored in the buffer (more cohesion), and farther away from the instances from existing classes (more separation). The q-NSC($\boldsymbol{x}$) value of an *F-outlier* $\boldsymbol{x}$ must be computed separately for each classifier $M_i \in \mathcal{M}$. ECHO declares emergence of a novel class if it finds at least $q' > q$ *F-outlier*s having a positive q-NSC score for all the classifiers $M_i \in \mathcal{M}$.

3.4 Calculation of Confidence Scores

ECHO employs two heuristics, i.e., *association* and *purity* to estimate confidence of each individual model $M_i \in \mathcal{M}$ in classifying any instance $\boldsymbol{x}$. These individual model confidences are then combined together to calculate confidence of the entire ensemble classifier. Let h_{ip}

be the p^{th} pseudopoint in M_i, and c_m be the class having highest frequency in h_{ip}. Assuming the closest pseudopoint from $\boldsymbol{x}$ in model M_i is h_{ip}, the heuristics are calculated as follows:

- *Association* is calculated by $R(h_{ip}) - D_{ip}(\boldsymbol{x})$, where $R(h_{ip})$ is the radius of h_{ip} and $D_{ip}(\boldsymbol{x})$ is the distance of $\boldsymbol{x}$ from h_{ip}. Therefore, smaller the $D_{ip}(\boldsymbol{x})$, higher the *association*.
- *Purity* is calculated by $\frac{|\mathcal{L}_{ip}(c_m)|}{|\mathcal{L}_{ip}|}$, where $|\mathcal{L}_{ip}|$ is the sum of all *frequencies* in h_{ip}, and $|\mathcal{L}_{ip}(c_m)|$ is the *frequency* of c_m in h_{ip}.

Association and *purity* of the model M_i are denoted by $\mathcal{A}_i$ and $\mathcal{P}_i$ respectively. We theoretically justify use of these heuristics in Section 3.5. Both of the heuristics contribute to model confidence according to their estimation capability. This capability is evaluated by the correlation between heuristic values and classification accuracy using the initial training data as follows. Heuristic values for M_i are calculated for each of the training instances. Let $\mathcal{H}_{ij}^k$ be the value of j^{th} heuristic in M_i's classification of instance $\boldsymbol{x}^{(k)}$, the k^{th} instance in the stream. Since we use two heuristics, $j \in \{1, 2\}$. Let $\hat{y}_i^k$ be the prediction of M_i on instance $\boldsymbol{x}^{(k)}$, and y^k be the true label of that instance. Let v_i is the vector containing v_i^k values indicating whether the classification of instance $\boldsymbol{x}^{(k)}$ by model M_i is correct or not. In other words, $v_i^k = 1$ if $\hat{y}_i^k = y^k$ and $v_i^k = 0$ if $\hat{y}_i^k \neq y^k$. Finally, a correlation vector r_i is calculated for model M_i. It contains r_{ij} values, which are *Point-biserial* correlation coefficients between $\mathcal{H}_{ij}$ and v_i for different j. Once the correlation coefficients, i.e., estimation capabilities are evaluated, it is retained for calculating model confidence in classifying future test instances.

To calculate the confidence of M_i in classifying a test instance $\boldsymbol{x}$ (denoted by $\mathcal{C}_i^{(\boldsymbol{x})}$), ECHO first calculates heuristic values $\mathcal{H}_i^{(\boldsymbol{x})}$. Next, $\mathcal{C}_i^{(\boldsymbol{x})}$ is calculated by taking the dot product of $\mathcal{H}_i^{(\boldsymbol{x})}$ and r_i, i.e., $\mathcal{C}_i^{(\boldsymbol{x})} = \mathcal{H}_i^{(\boldsymbol{x})}.r_i$. Similarly, ECHO calculates confidence scores for each of the models in the ensemble. These scores are then normalized between 0 and 1. Finally, ECHO takes the average confidence of the models towards the predicted class to estimate confidence of the entire ensemble $\mathcal{C}^{(\boldsymbol{x})}$.

3.5 Justification of Confidence Estimators

In this Section, we first define the objective function for semi-supervised K-means clustering that is used to build models in ECHO. Then, based on the objective function, we theoretically justify the choice of the heuristics for estimating classifier confidence.

3.5.1 Objective Function

As mentioned in Section 3.2, any standard clustering algorithm can be used in ECHO. In this chapter, we use the semi-supervised imputrity-based clustering method proposed in (Masud et al., 2008) for building the classification models in ECHO. Given a limited amount of labeled data, the goal of impurity-based clustering is to create K clusters by minimizing the intra-cluster dispersion, and at the same time by minimizing the impurity of each cluster. This is also referred to as K-means with *Minimization of Cluster Impurity* (MCI-Kmeans). A cluster is completely pure if all the labeled data points in that cluster belong to the same class. As all the clusters in this context are desired be pure with other unlabeled data instances, a term is added to the objective function for the clustering to penalize each cluster for being impure. Considering both of the intra-cluster dispersion and the cluster purity, the general form of the objective function used in MCI-Kmeans is as follows:

$$O_{MCIKmeans} = \sum_{i=1}^{K} \sum_{\boldsymbol{x} \in \mathcal{X}_i} ||\boldsymbol{x} - \mu_i||^2 + \sum_{i=1}^{K} \mathcal{W}_i * Imp_i \tag{3.2}$$

where $\mathcal{W}_i$ is the weight associated with cluster i, Imp_i is the impurity of cluster i, μ_i is the centroid of cluster i, and $\mathcal{X}_i$ is the set of all (both labeled and unlabeled) points in cluster i. To ensure that both the intra-cluster dispersion and cluster impurity are given the same importance, the weight associated with each cluster is chosen to be:

$$\mathcal{W}_i = |\mathcal{L}_i| * \bar{\mathcal{D}}_{\mathcal{L}_i} \Rightarrow \mathcal{W}_i = \sum_{\boldsymbol{x} \in \mathcal{L}_i} ||\boldsymbol{x} - \mu_i||^2$$

where $\mathcal{L}_i$ is the set of all labeled data points in Cluster i and $\bar{\mathcal{D}}_{\mathcal{L}_i}$ is the average dispersion from each of these labeled points to the cluster centroid.

Any impurity measure can be plugged in to Equation 3.2. Following (Masud et al., 2008), we use $Imp_i = ADC_i * Ent_i$ as the impurity measure, where ADC_i is the "Aggregated Dissimilarity Count" of cluster i and Ent_i is the entropy of cluster i. The Dissimilarity count $DC_i(\boldsymbol{x}, y)$ of a data point $\boldsymbol{x}$ in cluster i having class label y is the total number of labeled points in that cluster belonging to classes other than y. $DC_i(\boldsymbol{x}, y) = 0$ if if $\boldsymbol{x}$ is unlabeled (i.e., $y = \emptyset$). On the contrary, $DC_i(\boldsymbol{x}, y) = |\mathcal{L}_i| - |\mathcal{L}_i(c)|$, if $\boldsymbol{x}$ is labeled and its label $y = c$, where $\mathcal{L}_i(c)$ is the set of labeled points in cluster i belonging to class c. The "Aggregated Dissimilarity Count", denoted by ADC_i, is the sum of the dissimilarity counts of all the points in cluster i: $ADC_i = \sum_{\boldsymbol{x} \in \mathcal{L}_i} DC_i(\boldsymbol{x}, y)$. The entropy of a cluster i is computed as: $Ent_i = \sum_{c=1}^{C} (-p_c^i * log(p_c^i))$, where p_c^i is the prior probability of class c, i.e., $p_c^i = \frac{|\mathcal{L}_i(c)|}{|\mathcal{L}_i|}$.

After combining all the above terms, our objective function becomes:

$$O_{MCIKmeans} = \sum_{i=1}^{K} \sum_{\boldsymbol{x} \in \mathcal{X}_i} ||\boldsymbol{x} - \mu_i||^2 + \sum_{i=1}^{K} \sum_{\boldsymbol{x} \in \mathcal{L}_i} ||\boldsymbol{x} - \mu_i||^2 * \sum_{\boldsymbol{x} \in \mathcal{L}_i} (|\mathcal{L}_i| - |\mathcal{L}_i(c)|) * \sum_{c=1}^{C} \left(-\frac{|\mathcal{L}_i(c)|}{|\mathcal{L}_i|} * \log \frac{|\mathcal{L}_i(c)|}{|\mathcal{L}_i|} \right) \tag{3.3}$$

Let h_{ip} and h_{jq} be the closest pseudopoint from a test data point $\boldsymbol{x}$ in model M_i and M_j respectively. We define M_i will have higher confidence than model M_j in classifying a test data instance $\boldsymbol{x}$, if including $\boldsymbol{x}$ in h_{jq} increases the objective function more than including $\boldsymbol{x}$ in h_{ip}. We define the following terms:

$\Delta Disp_i^{(\boldsymbol{x})} \leftarrow$ increase in intra-cluster dispersion due to adding $\boldsymbol{x}$ to the closest pseudopoint in M_i.

$\Delta ADC_i^{(\boldsymbol{x})} \leftarrow$ increase in the "Aggregated Dissimilarity Count" due to adding $\boldsymbol{x}$ to the closest pseudopoint in M_i.

$\Delta Ent_i^{(\boldsymbol{x})} \leftarrow$ increase in entropy due to adding $\boldsymbol{x}$ to the closest pseudopoint in M_i.

Next, we justify the use of *association* and *purity* as confidence estimators.

3.5.2 Association

Consider two cases: a) $\boldsymbol{x}$ is inside only one of h_{ip} and h_{jq}. Without loss of generality, let assume that $\boldsymbol{x}$ falls inside h_{ip} but outside of h_{jq}. Therefore,

$$R(h_{ip}) > D_{ip}(\boldsymbol{x}) \Rightarrow R(h_{ip}) - D_{ip}(\boldsymbol{x}) > 0 \Rightarrow \mathcal{A}_i(\boldsymbol{x}) > 0$$

and

$$R(h_{jq}) < D_{jq}(\boldsymbol{x}) \Rightarrow R(h_{jq}) - D_{jq}(\boldsymbol{x}) < 0 \Rightarrow \mathcal{A}_j(\boldsymbol{x}) < 0$$

So, from the above equations, we get the following:

$$\mathcal{A}_i(\boldsymbol{x}) > \mathcal{A}_j(\boldsymbol{x}) \tag{3.4}$$

If $\boldsymbol{x}$ falls inside h_{ip} but outside of h_{jq}, h_{ip} has a greater *association* value than h_{jq}. Therefore, if other properties remain same, M_i will have greater confidence than M_j as expected. Moreover, since the point $\boldsymbol{x}$ falls outside of decision boundary of h_{jq} but inside of h_{ip}, $||\mu_{jq} - \boldsymbol{x}||^2 > ||\mu_{ip} - \boldsymbol{x}||^2 \Rightarrow \Delta Disp_i^{(\boldsymbol{x})} < \Delta Disp_j^{(\boldsymbol{x})}$, where μ_{ip} and μ_{jq} are centroids of h_{ip} and h_{jq} respectively. So, including $\boldsymbol{x}$ into h_{jq} increases the objective function value (Equation 3.3) more than that of h_{iq}. So, M_i will have more confidence than M_j in classifying $\boldsymbol{x}$.

b) Test instance $\boldsymbol{x}$ falls into the same side of both h_{ip} and h_{jq}. Without loss of generality, let assume that, h_{ip} and h_{jq} have similar properties (e.g., *centroid*, *radius* etc). Let us also assume that M_i has a higher *association* than M_j in the case of classifying $\boldsymbol{x}$. Therefore, we can deduce:

$$\begin{aligned}
&\mathcal{A}_i(\boldsymbol{x}) > \mathcal{A}_j(\boldsymbol{x}) \\
&\Rightarrow R(h_{ip}) - D_{ip}(\boldsymbol{x}) > R(h_{jq}) - D_{jq}(\boldsymbol{x}) \\
&\Rightarrow R(h_{ip}) + D_{jq}(\boldsymbol{x}) > R(h_{jq}) + D_{ip}(\boldsymbol{x}) \\
&\Rightarrow D_{jq}(\boldsymbol{x}) > D_{ip}(\boldsymbol{x}) \quad (\text{since } R(h_{ip}) = R(h_{jq})) \\
&\Rightarrow ||\mu_{jq} - \boldsymbol{x}||^2 > ||\mu_{ip} - \boldsymbol{x}||^2 \\
&\Rightarrow \Delta Disp_i^{(\boldsymbol{x})} < \Delta Disp_j^{(\boldsymbol{x})}
\end{aligned} \tag{3.5}$$

Therefore, in this case also, including $\boldsymbol{x}$ into h_{jq} increases the objective function more than including $\boldsymbol{x}$ into h_{ip}. So, in both cases, greater *association* leads to better confidence.

3.5.3 Purity

Assume that h_{ip} predicts $\boldsymbol{x}$ as an instance of c_m, and h_{iq} predicts $\boldsymbol{x}$ as an instance of c_n. Let us also assume that h_{ip} and h_{jq} share similar properties except that h_{ip} has a higher *purity* than h_{jq} in predicting data point $\boldsymbol{x}$. There can be two different cases:

a) Both of the pseudopoints contain an equal number of labeled points, i.e., $|\mathcal{L}_{ip}| = |\mathcal{L}_{jq}|$. Then,

$$\begin{aligned}
&\mathcal{P}_i(\boldsymbol{x}) > \mathcal{P}_j(\boldsymbol{x}) \\
&\Rightarrow \frac{|\mathcal{L}_{ip}(c_m)|}{|\mathcal{L}_{ip}|} > \frac{|\mathcal{L}_{jq}(c_n)|}{|\mathcal{L}_{jq}|} \\
&\Rightarrow |\mathcal{L}_{ip}(c_m)| > |\mathcal{L}_{jq}(c_n)| \quad (\text{Since } |\mathcal{L}_{ip}| = |\mathcal{L}_{jq}|) \\
&\Rightarrow -|\mathcal{L}_{ip}(c_m)| < -|\mathcal{L}_{jq}(c_n)| \\
&\Rightarrow |\mathcal{L}_{ip}| - |\mathcal{L}_{ip}(c_m)| < |\mathcal{L}_{jq}| - |\mathcal{L}_{jq}(c_n)| \text{ (as } |\mathcal{L}_{ip}| = |\mathcal{L}_{jq}|) \\
&\Rightarrow DC_{ip}(\boldsymbol{x}, c_m) < DC_{jq}(\boldsymbol{x}, c_n) \\
&\Rightarrow \Delta ADC_{ip}^{(\boldsymbol{x})} < \Delta ADC_{jq}^{(\boldsymbol{x})} \qquad (3.6)
\end{aligned}$$

b) The pseudopoints contain an unequal number of labeled points, i.e., $|\mathcal{L}_{ip}| \neq |\mathcal{L}_{jq}|$. Then,

$$\begin{aligned}
&\mathcal{P}_i(\boldsymbol{x}) > \mathcal{P}_j(\boldsymbol{x}) \\
&\Rightarrow \frac{|\mathcal{L}_{ip}(c_m)|}{|\mathcal{L}_{ip}|} > \frac{|\mathcal{L}_{jq}(c_n)|}{|\mathcal{L}_{jq}|}
\end{aligned}$$

Again since $p_{c_m}^i = \frac{|\mathcal{L}_{ip}(c_m)|}{|\mathcal{L}_{ip}|}$ and $p_{c_n}^j = \frac{|\mathcal{L}_{jq}(c_n)|}{|\mathcal{L}_{jq}|}$, the following holds:

$$\begin{aligned}
&p_{c_m}^i > p_{c_n}^j \\
&\Rightarrow \Delta Ent_{ip}^{(\boldsymbol{x})} < \Delta Ent_{jq}^{(\boldsymbol{x})} \qquad (3.7)
\end{aligned}$$

Equation 3.6 and Equation 3.7 show that in both cases, including $\boldsymbol{x}$ into the closest pseudo-point in model M_j will increase the value of the objective function more than that of model M_i. Thus, a higher value of *purity* leads to higher confidence.

3.6 Effect of Concept Drift on Classifier Confidence

In this Section, we show that classifier confidence decreases due to a concept drift. First, we will see the relationship between the intra-cluster dispersion in terms of sum of squared error (SSE) and *association*. Let SSE_p be the SSE of pseudopoint h_{ip}, μ_{ip} be the centroid of h_{ip}, and $\boldsymbol{x}$ be an arbitrary data point. Therefore, the SSE_i of a classification model is defined as follows:

$$\begin{aligned} SSE_i &= \sum_p SSE_p = \sum_p \sum_{\boldsymbol{x} \in h_{ip}} (\boldsymbol{x} - \mu_{ip})^2 \\ &= \sum_p n_{ip} \frac{\sum_{\boldsymbol{x} \in h_{ip}} (\boldsymbol{x} - \mu_{ip})^2}{n_{ip}} = \sum_p n_{ip} \bar{D}_{ip} \end{aligned} \tag{3.8}$$

where $\bar{D}_{ip}$ is the mean distance between a data point in h_{ip} and the centroid of h_{ip}, and n_{ip} is the number of instances in h_{ip}. Now the sum of *association* of the model M_i can be formulated as follows:

$$\begin{aligned} \mathcal{A}_i &= \sum_p \mathcal{A}_{ip} = \sum_p \sum_{\boldsymbol{x} \in h_{ip}} (R(h_{ip}) - (\boldsymbol{x} - \mu_{ip})^2) \\ &= \sum_p n_{ip} R(h_{ip}) - \sum_p \sum_{\boldsymbol{x} \in h_{ip}} (\boldsymbol{x} - \mu_{ip})^2 \\ &= \sum_p n_{ip} R(h_{ip}) - SSE_i \end{aligned} \tag{3.9}$$

Equation (3.9) concludes that total model *association* is inversely proportional to the model SSE because the pseudopoint radii are fixed (can be considered constant) on a given model.

Next, we show that a concept drift increases model SSE. We describe concept drift as the drift of the decision boundary, obtained by drifted pseudopoints. Assume, without loss of

generality, that concept drift is quantified by the amount of drift of the pseudopoints, which is obtained by δ_{ip} displacement of the centroid of h_{ip} for all p in the model. Assume that a new window (i.e., chunk) appears in the stream that exhibits the drift. Therefore, the new chunk can be obtained by moving the center of each pseudopoint h_{ip} in the current model by δ_{ip}, while keeping the same distribution of data points in each pseudopoint. Let the drifted pseudopoint be h'_{ip}. Now, if we randomly draw n_{ip} data points from h'_{ip}, and calculate the mean distance between each a point $\boldsymbol{x}'$ and the centroid μ_{ip} (of the old pseudopoint), the mean distance would be higher by δ_{ip} amount. Let SSE'_i be the SSE of the new model, and D'_{ip} be the mean distance between new data points and old pseudopoint. Therefore, we can derive that:

$$\begin{aligned} SSE'_i &= \sum_p n_{ip} D'_{ip} = \sum_p n_{ip} (\bar{D}_{ip} + \delta_{ip}) \\ &= \sum_p n_{ip} \delta_{ip} + \sum_p n_{ip} \bar{D}_{ip} \\ &= \sum_p n_{ip} \delta_{ip} + SSE_i > SSE_i \end{aligned} \tag{3.10}$$

Since $SSE'_i > SSE_i$, from equation (3.9) we can follow up that $A'_i < A_i$, meaning that *association*, and as a consequence, confidence in classifying the new data chunk will be lower because of the concept drift. The higher the amount of drift, the lower the confidence will be.

3.7 Change Detection

ECHO maintains a variable size window W to monitor confidence of the classifier on recent data instances. Confidence scores are generated in the range of $[0, 1]$ (discussed in Section 3.4). We observe from our experiments on various datasets that generated confidence scores tend to follow a *beta* distribution. We have carried out *Chi-Square* goodness of fit test on the generated confidence scores which also confirm that observation. Moreover, we

have shown in Section 3.6 that confidence scores decrease consistently in presence of a concept drift. Therefore, we propose a CUSUM (Baron, 1999)-type change detection technique (CDT) on beta distribution to use in this context. After inserting each confidence score, our CDT is used to detect any significant change of statistical properties within the values stored in W.

Algorithm 1 Detect-Change (η, γ, W)

Input: η: Sensitivity; γ: Cushion period size; W: The dynamic sliding window.
Output: The change-point if exists; -1 otherwise

1: $T_h \leftarrow -log(\eta)$, $n \leftarrow$size of W, and $\omega_n \leftarrow 0$.
2: **if** $n \leq V_m$ & $mean(W^{(1)} \dots W^{(n)}) > 0.3$ **then**
3: **for** $k \leftarrow \gamma : n - \gamma$ **do**
4: Estimate pre and post beta distributions, $beta(\hat{\alpha_0}, \hat{\beta_0})$ and $beta(\hat{\alpha_1}, \hat{\beta_1})$ from $W^{(1)} \dots W^{(k)}$ and $W^{(k+1)} \dots W^{(n)}$ respectively.
5: Calculate $S_{k,n}$ using Equation (3.11).
6: **end for**
7: Calculate ω_n using Equation (3.12).
8: **if** $\omega_n \geq T_h$ **then**
9: **Return** $kmax$, where $S_{kmax} = \omega_n$.
10: **else**
11: **Return** -1.
12: **end if**
13: **else**
14: **Return** n.
15: **end if**

Algorithm 1 sketches the proposed CDT. Let n be the size of W, if at any time overall mean confidence of the classifier falls below 0.3, or the window size exceeds the maximum allowable size, the proposed CDT returns n as the change-point. Otherwise, first it divides W into two sub-windows for each k between γ to $n-\gamma$. We know that each of the sub-windows contain a beta distribution, however the parameters are unknown. So, $W^{(1)} \dots W^{(k)}$ and $W^{(k+1)} \dots W^{(n)}$ constitute pre and post beta distributions respectively, where $W^{(i)}$ is the i^{th} element in W. We estimate the parameters using the method of moments. So, each sub-window should contain at least γ number of values to preserve statistical properties of a distribution. In the literature, $\gamma = 100$ is widely used, which is also called the *cushion*

period. The proposed CDT estimates the parameters at Line 4. Let $(\hat{\alpha_0}, \hat{\beta}_0)$ and $(\hat{\alpha_1}, \hat{\beta}_1)$ are the estimated parameters for pre and post beta distributions respectively. Then, sum of the log likelihood ratios is calculated at Line 5 using the following formula:

$$S_{k,n} = \sum_{i=k+1}^{N} log \left(\frac{f\left(W^{(i)} \mid \hat{\alpha}_1, \hat{\beta}_1\right)}{f\left(W^{(i)} \mid \hat{\alpha}_0, \hat{\beta}_0\right)} \right) \tag{3.11}$$

Where $f\left(W^{(i)}|\hat{\alpha}, \hat{\beta}\right)$ is the probability density function (PDF) of the beta distribution having parameters $\left(\hat{\alpha}, \hat{\beta}\right)$ applied on $W^{(i)}$. Next, the CUSUM process score ω_n for the values stored in W is calculated at Line 7 using the following formula:

$$\omega_n = \max_{\gamma \leq k \leq n-\gamma} S_{k,n} \tag{3.12}$$

Let $kmax$ is the value of k for which the algorithm calculated the maximum $S_{k,n}$ value where $\gamma \leq k \leq n - \gamma$. Finally, a change is detected at point $kmax$ if ω_n is greater than a pre-fixed threshold. We fix the threshold based on the value of the desired sensitivity η. In our experiments, we use $-log(\eta)$ as the threshold value.

3.8 Updating the Ensemble using Limited Labeled Data

A chunk boundary is detected as soon as a significant change is detected in the distribution of confidence scores. Subsequently, the current ensemble classifier is updated using the instances from that chunk as shown in Algorithm 2. However, instead of requiring true labels of all instances, ECHO intelligently selects a few instances for labeling using the classifier confidence scores. Since confidence scores are calculated using *association* and *purity* of the models, it provide useful insight to select important instances for updating the classifier. If the confidence in classifying an instance is below a threshold τ, ECHO requests for its true label and include in the labeled instance set (Line 1). On the contrary, if the confidence is above τ, ECHO uses the predicted label and includes the instance in the unlabeled instance

Algorithm 2 UpdateClassifier ($\mathcal{M}$, W, τ, cp)

Input: $\mathcal{M}$: The current ensemble classifier; W: The dynamic sliding window; τ: Classifier confidence threshold; cp: Estimated change-point.
Output: $\mathcal{M}_u$: The updated ensemble classifier

1: $\mathcal{L} \leftarrow \left\{ < \boldsymbol{x}, y >: \mathcal{C}^{(\boldsymbol{x})} \in W \text{ and } \mathcal{C}^{(\boldsymbol{x})} \leq \tau \right\}$ // y is the true label of $\boldsymbol{x}$
2: $\mathcal{U} \leftarrow \left\{ < \boldsymbol{x}, \hat{y} >: \mathcal{C}^{(\boldsymbol{x})} \in W \text{ and } \mathcal{C}^{(\boldsymbol{x})} > \tau \right\}$ // $\hat{y}$ is the predicted label of $\boldsymbol{x}$
3: $\mathcal{T} \leftarrow \mathcal{L} \cup \mathcal{U}$ // $\mathcal{T}$ is the training set
4: $M' \leftarrow$ TrainNewModel ($\mathcal{T}$)
5: $\mathcal{M}_u \leftarrow$ Update ($\mathcal{M}, M'$)
6: $\mathcal{T} \leftarrow \emptyset$
7: $W \leftarrow \left[W^{(cp+1)} \dots W^{(n)} \right]$

set (Line 2). Labeled and unlabeled set of instances together form the new training set to train a new model (Line 4) as discussed in Section 3.2.

Once a new model is trained, it replaces the oldest one among the existing models in the ensemble. This ensures that we have exactly t models in the ensemble at any given point of time. In this way, the infinite length problem is addressed as a constant amount of memory is required to store the ensemble. The concept-drift problem is addressed by keeping the ensemble up-to-date with the most recent concept. Finally, W is updated so that it contains only $W^{(cp+1)} \dots W^{(n)}$, where cp is the change-point.

3.9 Time and Space Complexity

ECHO has four modules, i.e., *Classification*, *Change Detection*, *Novel Class Detection*, and *Update*. Time complexity of invoking the *Novel Class Detection* module once is $O(KV_m)$, where K is the number of *pseudopoints*. This module is invoked only when the buffer contains q number of instances in it. Including this periodic call to the *Novel Class Detection* module, the total time complexity for classification is $O(KtV_m + tV_m + mKV_m)$, where $m = V_m/q$. Since $m >> Kt$, the total time complexity for the classification is $O(mKV_m)$. The time complexity for invoking *Change Detection* module for a whole chunk is $O(V_m^3)$. So, the overall time complexity for executing ECHO on a chunk of data is $O(mKV_m + V_m^3 + f(V_m))$,

where $f(V_m)$ is the time to train a new classifier with V_m training instances. Most often, $V_m >> m$ and $V_m >> K$, so the total time complexity is essentially $O(V_m^3 + f(V_m))$. We use four buffers, i.e., *filtered outlier buffer*, *training buffer*, *unlabeled data buffer* and *dynamic sliding window*. All the buffers can contain at most V_m instances, where V_m is the maximum allowable size for the dynamic sliding window. So, space complexity of our framework is $O(V_m)$.

3.10 Performance Improvement

As discussed in Section 3.9, time complexity of the *change detection* module of the proposed approach (ECHO) is $O(V_m^3)$ for one chunk of data. Therefore, change detection module adds significant overhead on ECHO. In this chapter, we propose strategies to improve performance of the proposed approach in terms of execution time.

3.10.1 Sporadic Execution

As shown in Equation 3.11 and 3.12, we calculate $S_{k,n}$ exhaustively for each k, $\gamma \leq k \leq n-\gamma$, to get CUSUM score ω_n on n observations. As stated before, confidence scores are expected to decrease when a concept drift occurs. Hence, we are interested about change-points only in the negative direction, i.e., change of distribution in decreasing confidence scores. We use this fact to avoid invoking computationally heavy CUSUM-type CDT after inserting each confidence score. Rather, we use a computationally inexpensive secondary algorithm to seek at least a certain amount of decrease in the population mean of confidence scores. Since the secondary algorithm can only detect a mere mean shift, it is assumed to produce frequent false alarms. Therefore, when it signals such a change, it is considered as a preliminary detection only. It will be then verified and confirmed by a more accurate but computationally heavier CUSUM-type CDT. In ECHO, we form the inexpensive secondary algorithm simply by comparing mean confidence scores of the sub-windows. It produces a warning and therefore

proposed CUSUM-type CDT is invoked only if η times decrease is observed between the mean scores of the sub-windows, where η is the sensitivity parameter introduced in Section 3.7.

3.10.2 Recursive Calculation

To calculate the CUSUM score ω_n, we need to calculate $S_{k,n}$ for each k, $\gamma \leq k \leq n - \gamma$. Hence, to develop the recursion, we divide the original problems into sub-problems based on the value of k and n. We propose the following recursion to calculate $S_{k,n}$:

$$S_{k,n} = \begin{cases} 0, & \text{if } 0 \leq n \leq 2\gamma \\ \sum_{i=k+1}^{n} \log \frac{f(W^{(i)}|\hat{\alpha}_1,\hat{\beta}_1)}{f(W^{(i)}|\hat{\alpha}_0,\hat{\beta}_0)}, & \\ \quad \text{if for all } m \leq n,\ I(k,m) = 0 & \\ S_{k,m} + \sum_{i=m+1}^{n} \log \frac{f(W^{(i)}|\hat{\alpha}_1,\hat{\beta}_1)}{f(W^{(i)}|\hat{\alpha}_0,\hat{\beta}_0)}, \text{ otherwise} & \\ \quad \text{where } m = \max\{g : g \leq n \ \&\ I(k,g) = 1\} & \end{cases} \tag{3.13}$$

Let current size of W be n. Since each of the sub-windows are required to contain at least γ number of observations, $S_{k,n} = 0$ for all $n \leq 2\gamma$. After receiving each test instance, ECHO predicts the label of the instance and inserts the confidence score into W. Since we only compute $S_{k,n}$ when a secondary algorithm gives a warning, for each $m \leq n$ we use $I(k,m)$ as an indicator function to indicate whether $S_{k,m}$ was computed or not. If there is no such $m \leq n$ for which $S_{k,m}$ was computed, we calculate $S_{k,n}$ using the original formula shown in Equation 3.11. These first two cases constitute the base of our recursive formula.

The third case applies if we have at least one $m \leq n$ for which $S_{k,m}$ was calculated before, i.e., $I(k,m) = 1$. If there exists multiple m like this, we find the largest m and reuse $S_{k,m}$ to calculate $S_{k,n}$ recursively. Parameters for pre and post-beta distributions are estimated as before from $W^{(1)} \ldots W^{(k)}$ and $W^{(k+1)} \ldots W^{(n)}$ respectively. However, in this case, we only calculate log-likelihood ratios for $W^{(m+1)} \ldots W^{(n)}$ and add with $S_{k,m}$ to get $S_{k,n}$. Since we do

not adjust for log-likelihood ratios for $W^{(k+1)} \dots W^{(m)}$, it introduces a loss in the estimation. On the other hand, it can be shown that mere calculating $S_{k,n}$ recursively reduces the time complexity of invoking change detection module for one chunk to $O(V_m^2)$. Empirical result also suggests that the above recursive calculation reduces execution time significantly while keeping very competitive accuracy.

Algorithm 3 Detect-Change-Revised (η, γ, W, $Stat$)

Input: η: Sensitivity; γ: Cushion period size; W: The dynamic sliding window; Ind: Data structure containing largest m for which $I(k, m)$=1; $Stat$: Data structure containing previous CUSUM calculations.

Output: The change-point if exists; -1 otherwise

$T_h \leftarrow -log(\eta)$, $n \leftarrow$size of W, and $\omega_n \leftarrow 0$.
if $n \leq V_m$ & $mean(W^{(1)} \dots W^{(n)}) > 0.3$ **then**
  for $k \leftarrow \gamma : n - \gamma$ **do**
    if Secondary algorithm (Section 3.10.1) produces a warning **then**
      Retrieve pre-beta distribution $beta(\hat{\alpha_0}, \hat{\beta_0})$ from $Stat[m]$ if $Ind[k] = m$: estimate the parameters explicitly if there is no such m.
      Estimate post beta distribution $beta(\hat{\alpha_1}, \hat{\beta_1})$ from $W^{(k+1)} \dots W^{(n)}$.
      Calculate $S_{k,n}$ using Equation (3.13).
    end if
  end for
  Calculate ω_n using Equation (3.12).
  Insert pre-beta distribution and $S_{k,n}$ at $State[n]$, and update $Ind[k] \leftarrow n$.
  if $\omega_n \geq T_h$ **then**
    Return $kmax$, where $S_{kmax} = \omega_n$.
  else
    Return -1.
  end if
else
  Return n.
end if

Algorithm 3 sketches the revised change detection algorithm using the above stated secondary inexpensive algorithm and recursion. We maintain two data structures *Ind* and *Stat*. *Ind* is used to implement the indicator function used in Equation 3.13. It returns the largest m for which $S_{k,m}$ has already been computed and -1 if no such m exists. On the other hand, *Stat* is used for memoization of previous CUSUM scores and distributions.

The algorithm starts with calculating the threshold and size of W. If there is a warning from the secondary algorithm, we use the proposed CUSUM-type CDT to verify it (Line 4). Pre and post beta distributions are estimated explicitly if *Ind(k)* returns -1, meaning there is no $m \leq n$ for which $S_{k,m}$ was calculated before. Otherwise, only post-beta distribution is estimated explicitly, and pre beta distribution is retrieved from *Stat* at Line 5. Next, $S_{k,n}$ is calculated using the Equation 3.13, and ω_n is calculated using Equation 3.12. *Ind* and *Stat* are updated using n and CUSUM calculations respectively. Finally, a change is detected if $\omega_n \geq T_h$.

3.10.3 Selective Execution

So far in Section 3.10.1 and 3.10.2, we have focused on efficiently detecting a change-point once the change detection module is invoked. To further reduce time complexity of ECHO, we focus on invoking the change detection module itself selectively instead of executing after inserting each confidence estimate. Since we intend to detect any significant decrease of confidence over a period of time, confidence scores calculated on the latest test instance could be used to decide whether the change detection needs to be executed or not. We can skip execution of the change detection module if the latest confidence score is high. On the contrary, we must execute the change detection if the latest confidence score is low. We examine the following two strategies based on this principle for selective execution of the change detection module:

1) First strategy is to use a pre-fixed sampling threshold on the classifier confidence to decide whether the change detection module will be invoked or not. We use the classifier confidence threshold τ (introduced in Section 3.8) as the sampling threshold also. If the classifier confidence on test instance $\boldsymbol{x}$, i.e., $\mathcal{C}^{(\boldsymbol{x})}$ is below τ, change detection module is invoked and vice versa. This version will be referred to as ECHO-F.

2) According to second strategy, The probability of executing the change detection module after inserting $\mathcal{C}^{(\boldsymbol{x})}$ is determined by $\mathrm{e}^{-\mathcal{C}^{(\boldsymbol{x})}}$. In other words, we calculate the probability of

invoking the change detection module based on the confidence score itself. Therefore, if the confidence in classifying an instance is high, the probability of invoking the change detection module will be low and vice versa. We will refer to this version as ECHO-D.

3.11 Evaluation

3.11.1 Datasets

Table 3.2 depicts the characteristics of the datasets. The *ForestCover* dataset is obtained from the UCI repository as explained in (Masud et al., 2011). It contains geospatial descriptions of different types of forests. The labeling task is to find the actual forest cover type for a given observation from US Forest Service (USFS) Region-2 Resource Information System (RIS) data. In order to prepare it for novel class detection, we arrange the data so that at most three and at least two classes co-occur at a similar time. In the second real-world

Table 3.2: Characteristics of datasets

Name of dataset	Num of Instances	Num of Classes	Num of Features
ForestCover	150,000	7	54
PAMAP	150,000	19	52
PowerSupply	29,927	24	2
HyperPlane	100,000	5	10
SynRBF@0.002	100,000	7	70
SynRBF@0.003	100,000	7	70

dataset used, referred to as *Physical Activity Monitoring (PAMAP)* (Reiss and Stricker, 2012), nine individuals were equipped with sensors that gathered a total of 53 streaming features whilst they performed activities. Nineteen total activities were identified as class labels - including one category for miscellaneous or transient activities. Without loss of generality, we use the first $150,000$ data instances from ForestCover and PAMAP datasets in our experiments. The third real-world dataset is *Powersupply* (Zhu, 2010), which contains

hourly PowerSupply of an Italian electricity company which records the power from two sources: PowerSupply from main grid and power transformed from other grids.

Besides using real-world datasets, we also use several synthetically generated datasets. HyperPlane (Zhu, 2010) is such a data stream which is generated using the equation: $f(x) = \sum_{j=1}^{d-1} a_j \frac{(x_j + x_{j+1})}{x_j}$, where $f(x)$ is the label of instance x and $a_j,\ j = 1, 2, .., d$, controls the shape of the decision surfaces. *SynRBF@X* are synthetic datasets generated using *RandomRBFGeneratorDrift* of MOA (Bifet et al., 2010) framework where X is the Speed of change of centroids in the model. We generate two such datasets using different X to check how efficiently different approaches can adapt to a concept drift.

We use ForestCover and PAMAP datasets for simulating both concept drift and novel classes. Rest of the datasets are used to test only concept drift handling capability of the considered approaches.

3.11.2 Experiment Setup

We implement two versions of the proposed framework, i.e., ECHO-F, ECHO-D to analyze performance as discussed in Section 3.10.3. We compare classification and novel class detection performance of our framework with *ECSMiner* (Masud et al., 2011). We have chosen ECSMiner since it addresses both concept drift and concept evolution. Other than that, we compare the performance of ECHO with *OzaBagAdwin* (OBA) and *Adaptive Hoeffding Tree* (AHT) implemented in MOA (Bifet et al., 2010) framework, since these approaches seem to have superior performance than others on the datasets used in the experiments. Both OBA and AHT use *ADWIN* (Bifet and Gavaldà, 2007) as the change detector. These approaches do not have novel class detection capability. So, we compare ECHO with these approaches only in terms of classification performance.

We evaluate each of the above classifiers on a stream by testing and then training with chunks of data in sequence. To evaluate ECSMiner, we use 50 pseudopoints, and ensemble

size 6 as suggested in (Masud et al., 2011). We use 100% labeled training data to evaluate ECSMiner, OBA and AHT. On the other hand, we use $t = 6$, $\tau = 0.90$, $\eta = 0.001$, $\gamma = 100$, and $V_m = 5000$ as the default setting for ECHO, if not specified otherwise. Percentage of labeled data in ECHO depends on the value of τ, and the frequency and intensity of concept drifts in the dataset used.

3.11.3 Performance Metrics

Let FN = total novel class instances misclassified as existing class, FP = total existing class instances misclassified as novel class, TP = total novel class instances correctly classified as novel class, Fe = total existing class instances misclassified (other than FP), N_c = total novel class instances, and N = total instances in the stream. We use the following performance metrics for evaluation:

1. *Error%*: Total misclassification error (percent), i.e., $\frac{(FP+FN+Fe)*100}{N}$.
2. M_{new}: % of novel class instances misclassified as existing class, i.e., $\frac{FN*100}{N_c}$.
3. F_{new}: % of existing class instances Falsely identified as novel class, i.e., $\frac{FP*100}{N-N_c}$.
4. F_2: F_β score provides the overall performance of a classifier by considering both *precision* and *recall*. We use $\beta = 2$, which gives us $F_2 = \frac{5*TP}{5*TP+4*FN+FP}$.
5. *Processing Time*: Average time required in milliseconds to process thousand instances.
6. *Processing Speed*: Average number of instances processed in one second.

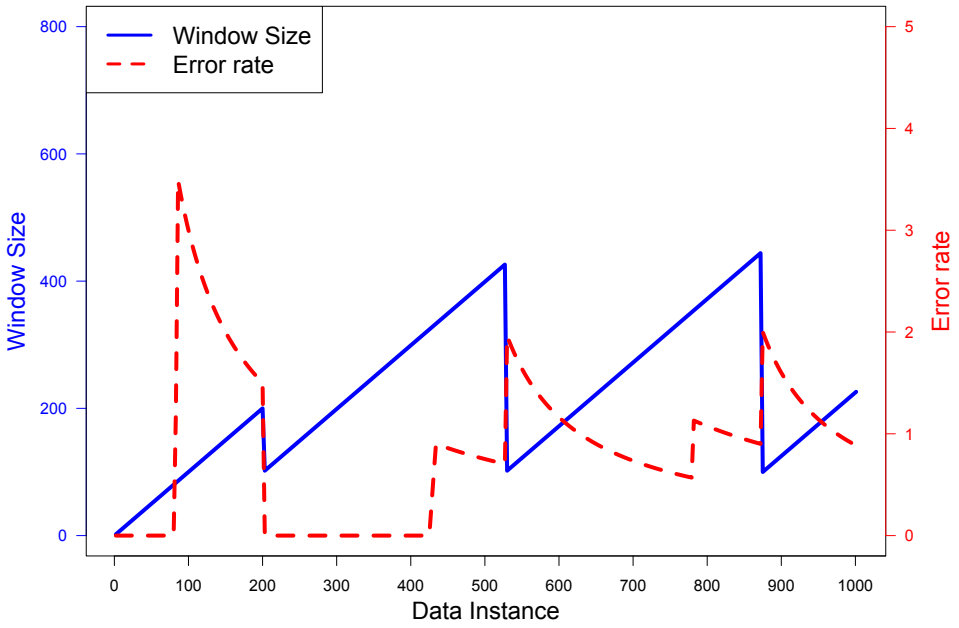
3.11.4 Classification

As discussed before, most techniques to classifying evolving data stream divide the stream into fixed-size chunks regardless of occurrence or intensity of concept drifts. On the contrary,

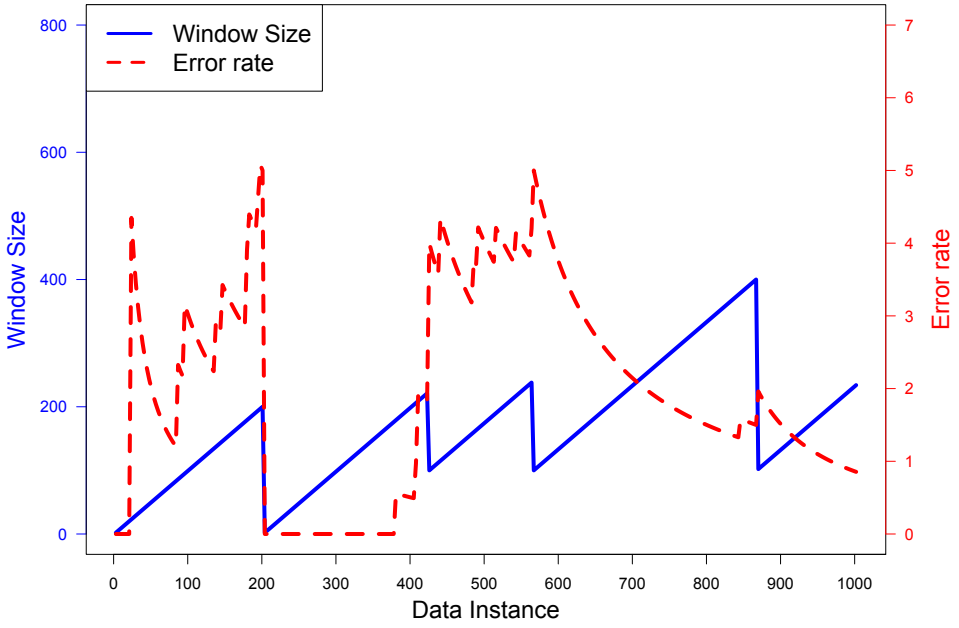
ECHO determines the chunk size dynamically based on any significant change in classifier confidence. Hence, ECHO avoids unnecessary training during the stable period and frequently update the classifier when needed. This characteristic is evident in Figure 3.3.

In this experiment, we have considered the first thousand instances of ForestCover and SynRBF@0.002 datasets, and plotted the window size and error rate as the stream progresses. In case of ForestCover dataset, we observe that whenever error rate of the classifier increases significantly, the proposed approach detects a concept drift, and reinitialize the sliding window W after updating the classifier. Increase in classifier error rate indirectly indicates a concept drift. Therefore, this supports the claim we made in Section 3.6 that though estimated without any supervision, classifier confidence decreases if there is a change in underlying concepts. Moreover, we also observe that following re-initialization of the sliding window, the error rate drops as expected. This indicates that the updated classifier reflects the current concepts, and thereby classification accuracy increases. Figure 3.3b shows similar behavior in case of SynRBF@0.002 dataset. However, we observe more frequent concept drift detection, which is expected as we increase frequency and intensity of concept drifts intentionally in this synthetic dataset. Overall, this experiment shows that the proposed semi-supervised approach works well in detecting concept drifts and updating the classifier timely.

Table 3.3 and Table 3.4 compare performance of ECHO using two different values for τ, with other baseline approaches in terms of classification accuracy. More specifically, Table 3.3 shows classification accuracy of ECHO using $\tau = 0.9$. In this case, ECHO outperforms the other approaches by a large margin on all the datasets considered. We also observe that, a high value of τ in general results in a large amount of labeled data. However, this is not always true. For example, in case of PowerSupply dataset, ECHO versions use labels for less than 10% of instances despite using high τ, still exhibit superior performance. As discussed in Section 3.8, ECHO only requests label for an instance if the confidence is below τ. It



(a) ForestCover



(b) SynRBF@0.002

Figure 3.3: Change of window size and error rate of ECHO-D as the stream progresses

implies that, in case of PowerSupply dataset, ECHO has high confidence in classifying most of the test instances. So, instead of requesting labels for those instances, ECHO uses the predicted labels while updating the classifier.

The number of instances ECHO requests labels for is controlled by τ. We show classification performance of ECHO using $\tau = 0.4$ in Table 3.4. Although, other approaches considered in this table use 100% labeled data, we have put the performance of these approaches for ease of comparison. We observe that ECHO versions use extremely low amount of labeled data because of using a low value for τ. However, ECHO versions still show competitive performance, if not better, than the other considered approaches in case of all the datasets. So, experiment result suggests that ECHO can be used in scenarios where labeled data is very scarce and expensive.

3.11.5 Novel Class Detection

ECHO stores the filtered outliers temporarily in a buffer until there are enough instances in the buffer to detect a novel pattern. As the stream progresses, more instances from the novel pattern arrive in the stream which makes it easier to detect the novel pattern. However, similar to ECSMiner, we also consider a maximum allowable time up to which the classifier can wait for enough instances from a emerging class to appear. We set this time constraint as *400* instances as suggested in (Masud et al., 2011). Table 3.5 compares the novel class detection performance of ECHO-D and ECHO-F using $\tau = 0.9$ with ECSMiner on ForestCover and PAMAP datasets. We observe that both versions of ECHO show competitive performance. ECHO-D shows the best F_{new} and competitive M_{new} and F_2 on ForestCover dataset despite using less labeled data than ECSMiner. In case of PAMAP dataset, ECHO-D shows better performance than ECSMiner in terms of all M_{new}, F_{new}, and F_2. If we consider overall performance (*Error %*, M_{new}, F_{new}, and F_2), ECHO clearly outperforms all the other baseline approaches.

Table 3.3: Summary of classification results

Name of dataset	ECHO-F ($\tau = 0.9$)		ECHO-D ($\tau = 0.9$)		ECSMiner	AHT	OBA
	Error%	% of labeled data	Error%	% of labeled data	Error%	Error%	Error%
ForestCover	3.95	96.54	**3.68**	95.16	4.55	22.89	18.06
PAMAP	4.75	93.64	**3.73**	94.7	35.26	8.76	7.27
PowerSupply	0.01	8.93	**0.01**	9.8	0.01	85.59	86.92
HyperPlane	2.79	99.76	**2.18**	100	3.73	46.24	48.55
SynRBF@0.002	23.98	99.79	**18.19**	99.62	63.43	38.75	37.04
SynRBF@0.003	25.60	99.51	**22.37**	99.8	65.39	48.65	46.86

Table 3.4: Comparison of classification performance using limited amount of labeled data

Name of dataset	ECHO-F ($\tau = 0.4$)		ECHO-D ($\tau = 0.4$)		ECSMiner	AHT	OBA
	Error%	% of labeled data	Error%	% of labeled data	Error%	Error%	Error%
ForestCover	7.96	39.26	**3.88**	35.33	4.55	22.89	18.06
PAMAP	4.77	69.56	**4.73**	68.62	35.26	8.76	7.27
PowerSupply	0.01	0.1	**0.01**	0.14	0.05	85.59	86.92
HyperPlane	2.99	32.6	**2.36**	26.9	3.73	46.24	48.55
SynRBF@0.002	42.65	38.4	38.17	38.88	63.43	38.75	**37.04**
SynRBF@0.003	45.51	36.51	**40.65**	54.48	65.39	48.65	46.86

Table 3.5: Novel class detection performance using $\tau = 0.9$

dataset	Method	M_{new}	F_{new}	F_2
ForestCover	ECHO-F	11.37	2.27	0.72
	ECHO-D	14.51	**2.11**	0.71
	ECSMiner	**8.42**	2.13	**0.88**
PAMAP	ECHO-F	0.05	4.22	0.78
	ECHO-D	**0.05**	**3.39**	**0.82**
	ECSMiner	0.05	37.53	0.45

3.11.6 Parameter Sensitivity

In this section, we examine parameter sensitivity of the proposed approach. More specifically, we vary parameters of ECHO, i.e., confidence threshold (τ), number of models in the ensemble (t), sensitivity parameter (η), and examine its effect on performance of ECHO. We consider the error rate, percentage of labeled data used, and processing time as performance metrics in this set of experiments. To evaluate ECHO in terms of processing time, we use two metrics, namely *Execution Time* and *Processing Speed*. Execution Time is defined as the average time taken by ECHO in milliseconds to process one thousand instances from a dataset. On the other hand, Processing Speed is defined as the number of instances processed by ECHO on average in one second. In this set of experiments, we apply ECHO-D on SynRBF@0.002 dataset, and observe the performance metrics by varying the parameters discussed above.

Confidence Threshold

Figure 3.4 shows performance of ECHO-D as the confidence threshold (τ) varies. We discussed in Section 3.8 that ECHO selects instances for labeling based on the confidence threshold (τ). So, the percentage of labeled data used to update the classifier depends on the value of τ. A higher value of τ should incur a larger amount of labeled data and vice versa. This is exactly what we observe from the figure. As τ increases, the percentage of labeled data increases, and percentage of error decreases. In the figure, we also compare

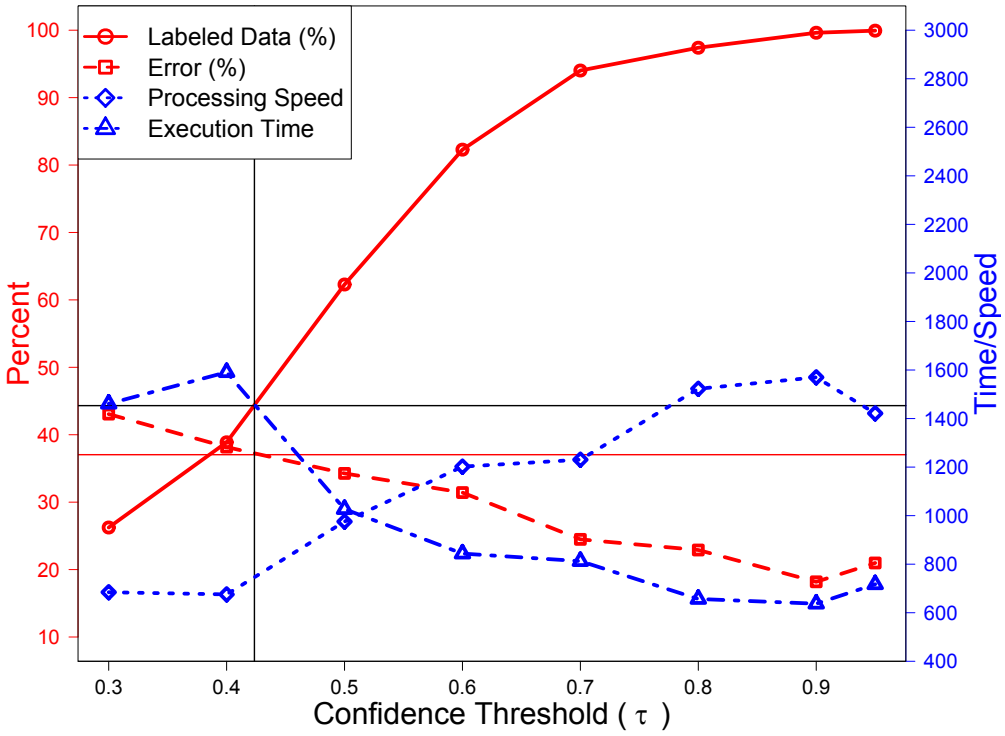


Figure 3.4: Sensitivity of ECHO-D to confidence threshold (τ) on SynRBF@0.002

the performance of ECHO with the best performing baseline method. From Table 3.3, we know that the *OzaBagAdwin* (OBA) approach performs best on SynRBF@0.002 in terms of classification accuracy among all the baseline methods. We observe from the figure that ECHO requires only around 45% labeled data to surpass that performance. This is impressive especially considering that OBA uses 100% labeled data. Figure 3.4 also shows effect of τ on Processing Speed and Execution Time. We observe that the Execution Time decreases, thereby Processing Speed increases with increasing τ.

Sensitivity Parameter

In the next experiment in this section, we examine sensitivity of ECHO-D to the sensitivity η and ensemble size t parameters. Figure 3.5 shows the performance of ECHO-D with increasing η. As discussed in Section 3.7, η controls the false alarm rate in change detection

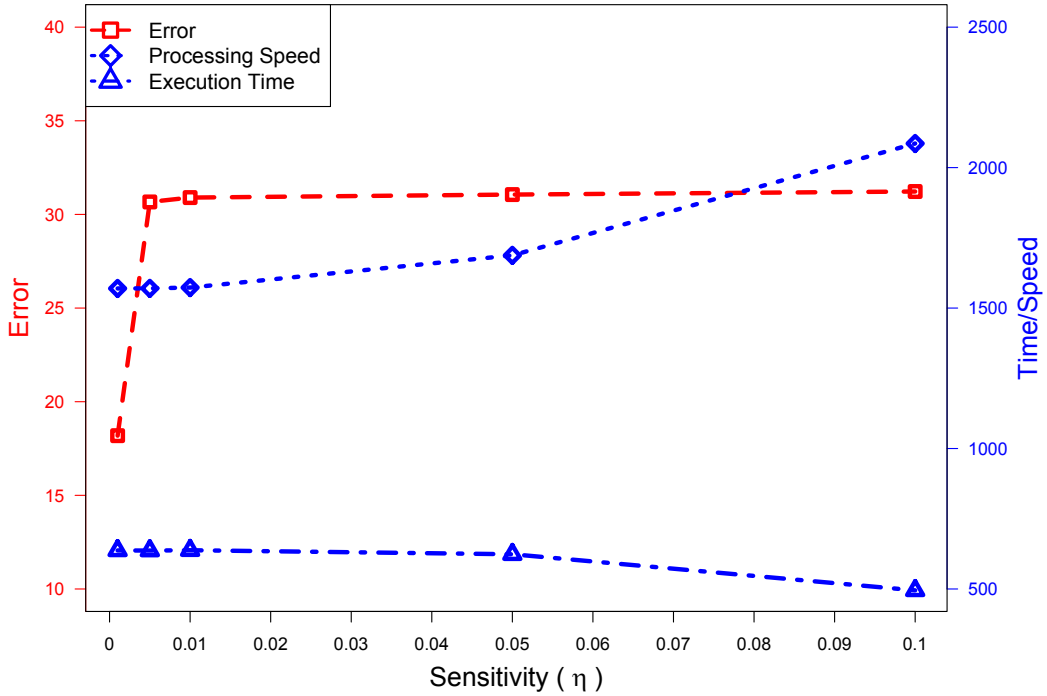


Figure 3.5: Performance of ECHO-D as sensitivity parameter (η) varies

algorithm. We observe that percentage of error increases with increasing η probably due to more false alarms in change detection. Moreover, increasing η results into a larger threshold in change detection. It means change detection becomes less sensitive and the average size of the sliding window increases with increasing η. As discussed in Section 3.9, the time complexity of ECHO depends on the size of the sliding window. Therefore, Execution Time decreases, thereby Processing Speed decreases with increasing η.

Ensemble Size

Figure 3.6 shows effect of varying ensemble size t on performance of ECHO-D using Syn-RBF@0.002 dataset. We observe that the error rate keeps decreasing with increasing until $t = 6$, due to the reduction of error variance (Tumer and Ghosh, 1996). However, if we keep increasing beyond $t = 6$, we observe an increase in the error rate. This behavior can be explained using the analysis presented in (Tumer and Ghosh, 1996), which states that

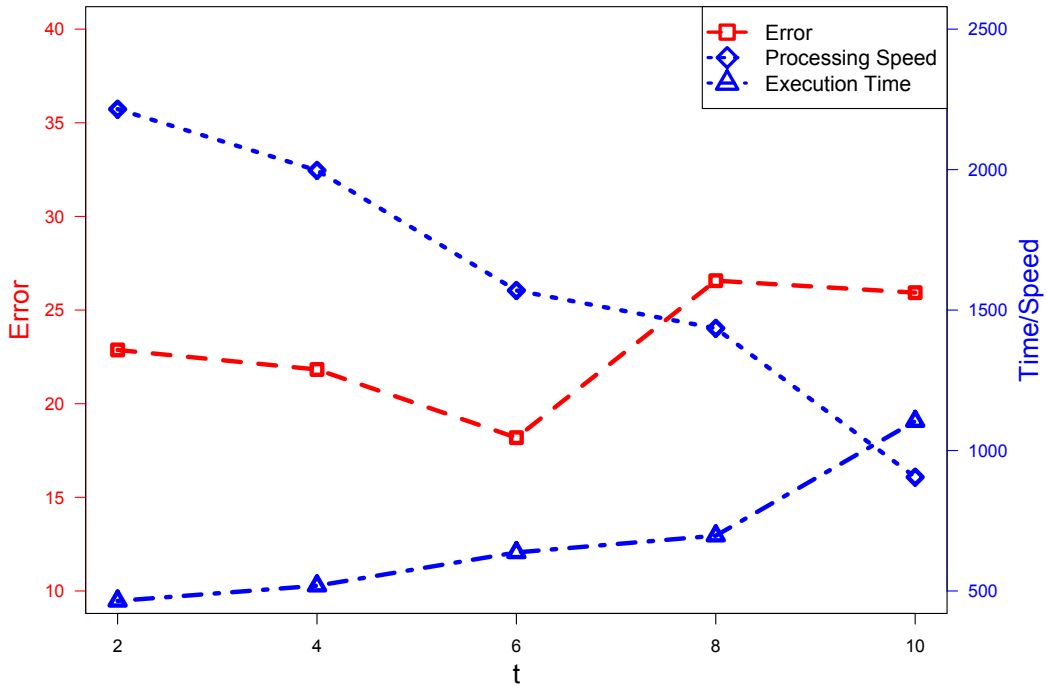
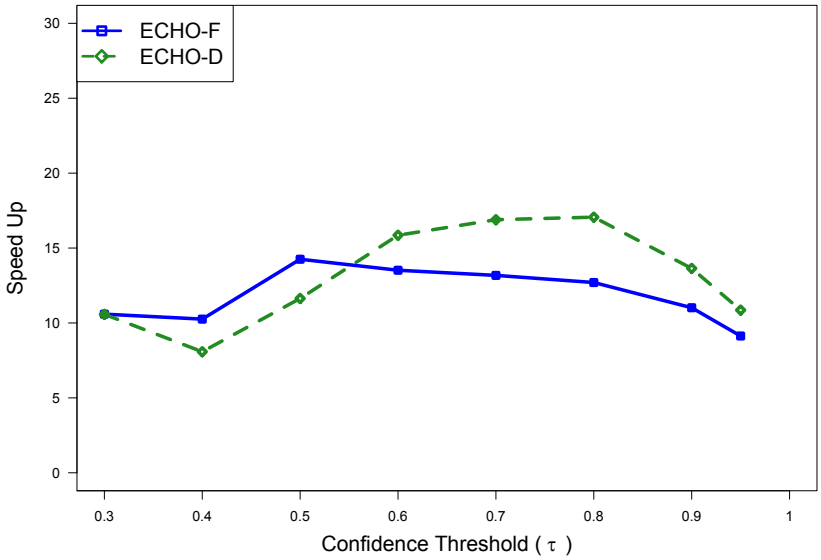


Figure 3.6: Sensitivity to ensemble size (t)

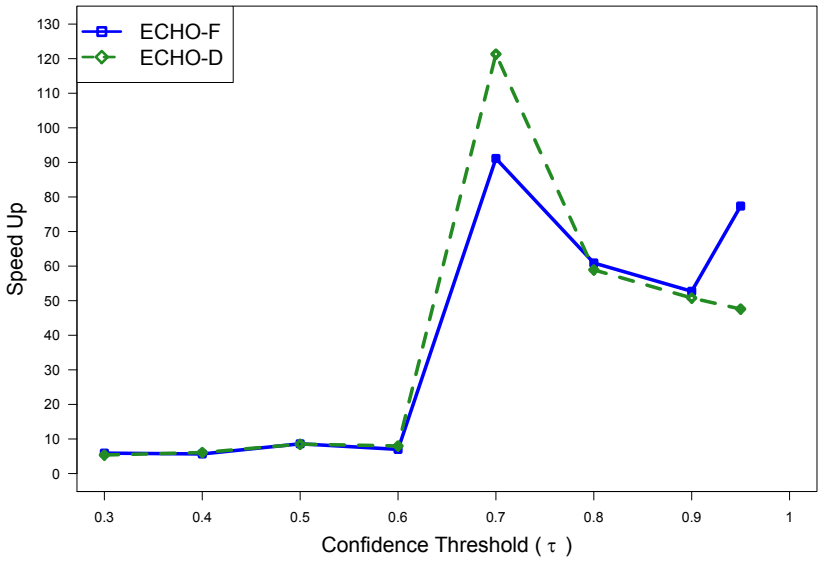
increasing ensemble size helps in decreasing error rate until individual model errors are correlated. Moreover, we observe that Execution Time increases and Processing Speed decreases with increasing t, due to more overhead in classification and ensemble maintenance.

3.11.7 Speed Up

In this set of experiments, we examine *Speed Up* achieved by ECHO using dynamic programming and other performance improvement techniques discussed in Section 3.10. To examine the improvement in terms of execution time, we compare the performance of ECHO-F and ECHO-D with basic ECHO implementation without any performance improvement. We define Speed Up as T_b/T_o, where T_b is the total time taken by the basic ECHO implementation and T_o is the total time taken by an improved version of ECHO. Figure 3.7 shows the Speed Up achieved by ECHO-F and ECHO-D on SynRBF@X datasets with increasing τ. As discussed before, a higher value of X indicates more frequent and more intense concept drift in



(a) SynRBF@0.002



(b) SynRBF@0.003

Figure 3.7: Confidence threshold (τ) vs Speed Up

SynRBF@X. We observe that improved versions of ECHO achieve as high as 18 times Speed Up than the basic implementation on SynRBF@0.002. In case of SynRBF@0.003, improved versions achieve as high as 120 times Speed Up which is very promising. This indicates the effectiveness of our proposed strategies for performance improvement.

CHAPTER 4

FUSION: AN ONLINE METHOD FOR MULTISTREAM CLASSIFICATION[1]

Data stream mining researchers have so far focused on mining a single stream of data. Even if data is received from more than one stream simultaneously, all of them are assumed to be generated from a non-stationary data generating process (Chandra et al., 2016). Any change in the data generating process would affect data distributions in these streams simultaneously. Therefore, all such streams can be combined into a single stream, as individual streams represent the same distribution. However, combining streams may not be effective in particular scenarios, especially if individual streams represent different distributions with asynchronous and independent concept drifts among them. This type of scenarios may arise if data is generated by two different, but related non-stationary processes.

For example, consider building a model for predicting sentiment of tweets (Kouloumpis et al., 2011). Typically, the sentiment is not provided as the ground truth along with a tweet. So, in order to collect training data, a few users may agree to provide tweets with sentiment label information. On the contrary, tweets on which the model needs to analyze the sentiment may come from any Twitter user. Users providing the training data may represent only a small portion of the population. Therefore, if we assume two streams of data, one from the Twitter users providing labeled data, another from the whole population of Twitter users, a sampling bias may exist between the distributions represented by these streams of data. This type of data shift is typically caused due to limited supervision, or lack of control over the data generating process (Chandra et al., 2016).

[1]

A new problem setting called *Multistream Classification* has been introduced in (Chandra et al., 2016) to address these scenarios. It involves two simultaneous streams of data. One of the streams, called the *source stream*, provides only labeled training data. The other stream, called the *target stream*, provides unlabeled test data. The classification task is to use the labeled data from the source stream for classifying unlabeled data from the target stream efficiently. As pointed out before, combining the two streams may result in a different overall distribution that inhibits available data patterns when they are considered individually. Moreover, independent and asynchronous concept drifts may occur in either of the streams over time. Therefore, traditional techniques for data stream mining may not be effective if applied to the combined stream.

The main challenge of Multistream classification is to address data shift and independent asynchronous concept drifts between the source and target streams. In this chapter, we propose an efficient approach for Multistream classification. We refer to this approach as FUSION (Efficient Multistream classification using Direct Density Ratio Estimation) (Haque et al., 2017). It uses two sliding windows for storing recent instances from the source and target streams. Data shift between the source and target stream is addressed by weighing each source instance based on the density ratio. Let $P_S(\cdot)$ and $P_T(\cdot)$ be the distributions represented by recent source and target data instances respectively. The density ratio for an instance $\boldsymbol{x}$ is defined by $\beta(\boldsymbol{x}) = \frac{P_T(\boldsymbol{x})}{P_S(\boldsymbol{x})}$. A Gaussian kernel model is used in FUSION for direct density ratio estimation. The model is updated online with incoming instances. An ensemble classifier is used for classification, where each model is trained on weighted source stream instances.

FUSION has an inherent capability of addressing asynchronous concept drifts in Multistream classification. In addition to addressing data shifts, FUSION uses density ratios estimated by the Gaussian kernel model for detecting any change between distributions represented by weighted source and target stream data. If a significant change is detected,

the Gaussian kernel model is updated. Subsequently, weights for the source stream labeled data are re-evaluated using the updated kernel model, and the ensemble classifier is updated. The efficiency of the proposed approach stems from the fact that it uses the same kernel model for addressing both data shift and asynchronous concept drift in Multistream classification. Empirical results on benchmark datasets show that FUSION outperforms the existing Multistream classification method in terms of both accuracy and execution time.

The main contributions of our work are as follows.

1. We present an efficient method for direct density ratio estimation in the Multistream setting. The model used in this method is updated online. The density ratios are used for data shift adaptation between the streams. We provide theoretical convergence rate for the proposed method.

2. We present a technique for detecting asynchronous concept drifts between the source and the target stream data using direct density ratios.

3. We propose an efficient approach for Multistream classification by fusing asynchronous concept drift adaptation into data shift adaptation. We derive the time and space complexity of the proposed approach.

4. We use benchmark real-world and synthetic datasets to evaluate our approach, and compare the performance with the baseline methods. In addition to the only existing method for Multistream classification, we use some of the acclaimed state-of-the-art data stream mining techniques as baseline methods.

4.1 The Proposed Approach

Figure 4.1 and Algorithm 4 illustrate the core components in FUSION. It has four main modules, i.e., *Density Ratio Estimation (DRM)*, *Drift Detection (DDM)*, *Classification*, and

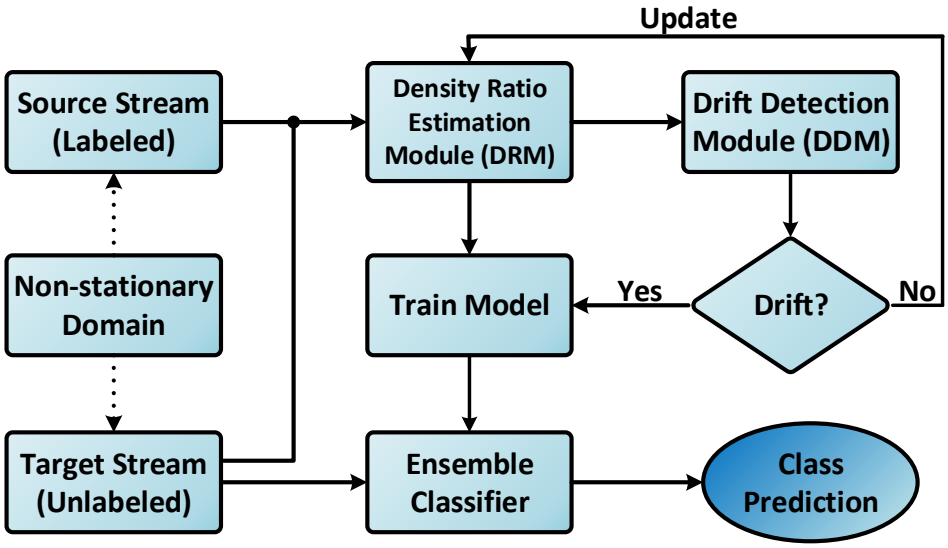


Figure 4.1: Overview of FUSION

Update. As mentioned in the problem statement, both source and target streams are generated from the same non-stationary domain, having asynchronous data drift between them. We use two fixed-size sliding windows to store recent instances from $\mathcal{S}$ and $\mathcal{T}$, referred to as the source and the target sliding window, and denoted as $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$ respectively. The size of the sliding windows is denoted by N_m.

FUSION uses an ensemble classifier for classification. The first model in the ensemble is trained on the initial instances from the source sliding window. However, to correct possible covariate shift between $\mathcal{S}$ and $\mathcal{T}$, each instance from the sliding window is associated with an importance weight. FUSION uses density ratios estimated from the Density Ratio Estimation Module (DRM) as importance weights. Label for any new instance arriving in $\mathcal{T}$ is predicted by taking the majority voting from the ensemble classifier. DRM updates the model for density ratio estimation incrementally with each incoming instance in either sliding windows.

As the system keeps receiving new instances in $\mathcal{S}$ or $\mathcal{T}$, the Drift Detection Module (DDM) detects drift between the distributions represented by data in the sliding windows

Algorithm 4 FUSION: Multistream Classification

Input: Labeled source stream data $\mathcal{S}$, The size of sliding windows N_m.
Output: Labels predicted on $\mathcal{T}$ data.

1: Read first N_m instances from $\mathcal{S}$ and $\mathcal{T}$ into $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$ respectively.
2: Learn $\boldsymbol{\alpha} = \{\alpha_i\}_{i=1}^{N_m}$ using *LearnAlpha* (Algorithm 5).
3: Estimate $\left\{\beta(\boldsymbol{\mathcal{W}}_S^{(i)})\right\}_{i=1}^{N_m}$ (Section 4.1.2).
4: Initialize $\mathcal{M}$ by learning a base model from $\boldsymbol{\mathcal{W}}_T$, $\boldsymbol{\mathcal{W}}_S$, and $\left\{\beta(\boldsymbol{\mathcal{W}}_S^{(i)})\right\}_{i=1}^{N_m}$.
5: **repeat**
6: Receive a new instance $\boldsymbol{x}$.
7: **if** $\boldsymbol{x} \in \mathcal{T}$ **then**
8: Predict label for $\boldsymbol{x}$ by taking the majority voting.
9: **end if**
10: Slide the corresponding window ($\boldsymbol{\mathcal{W}}_S$ or $\boldsymbol{\mathcal{W}}_T$) for including the new instance.
11: Update $\boldsymbol{\alpha}$ using *UpdateAlpha* (Algorithm 6).
12: Check for any drift in data using *DetectDrift* (Algorithm 7).
13: **if** *DetectDrift* returns *True* **then**
14: Recalculate $\boldsymbol{\alpha} = \{\alpha_i\}_{i=1}^{N_m}$ using *LearnAlpha*
15: Update $\mathcal{M}$ using *UpdateClassifier* (Algorithm 8).
16: **end if**
17: **until** $\mathcal{T}$ exists

using density ratios estimated by DRM. If there is a drift, a new model is trained on source sliding window instances along with their updated importance weights. Moreover, the ensemble classifier and the sliding windows are updated.

Table 4.1 lists frequently used symbols in this chapter. Throughout the chapter, typically a bold symbol or letter is used to denote a set of elements, and a superscript is used to indicate the index of an element in the set. A subscript is used to indicate the association of an entity to a type. For example, $\boldsymbol{\mathcal{W}}_S^{(i)}$ denotes the i^{th} data instance in the source sliding window. We present a detailed discussion about different modules of FUSION in rest of this section.

Table 4.1: Frequently used symbols

$\mathcal{D}$: Domain	$\mathcal{M}$: Ensemble Classifier
$\mathbb{P}$: Set of non-stationary processes	P_S: Source stream probability distribution
$\mathcal{S} \in \mathbf{P}$: A labeled source stream	P_T: Target stream probability distribution
$\mathcal{T} \in \mathbf{P}$: An unlabeled target stream	$\beta(\boldsymbol{x})$: Importance weight for $\boldsymbol{x}$
$\boldsymbol{\mathcal{W}}_S$, $\boldsymbol{\mathcal{W}}_T$: Source and Target Sliding window	L: The maximum allowable ensemble size
$\boldsymbol{x}$: d-dimensional features (or covariates)	$\boldsymbol{\alpha}$: The set of parameters of the Gaussian kernel model
$y \in \mathbf{Y}$: Class label of a data instance	N_m: The size of $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$

4.1.1 Density Ratio Estimation Module (DRM)

The Density Ratio Estimation module (DRM) of FUSION uses a Gaussian kernel model for direct density estimation. The model is updated incrementally as new instances appear in $\mathcal{S}$ or $\mathcal{T}$. In this section, we describe the DRM, and its online update procedure.

Gaussian Kernel Model

At a particular time, we define the source distribution P_S, and the target distribution P_T by the distributions represented by data instances in $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$ respectively at that moment. Density ratio for an instance $\boldsymbol{x}$ is defined by $\beta(\boldsymbol{x}) = \frac{P_T(\boldsymbol{x})}{P_S(\boldsymbol{x})}$. If $\boldsymbol{x}$ is an instance from $\mathcal{S}$, $\beta(\boldsymbol{x})$ is used as its importance weight in the learning process. We will discuss more on training later in this section.

Using Gaussian kernel model, $\beta(\boldsymbol{x})$ is modeled as follows:

$$\hat{\beta}(\boldsymbol{x}) = \sum_{i=1}^{N_m} \alpha_i K_\sigma \left(\boldsymbol{x}, \boldsymbol{\mathcal{W}}_T^{(i)} \right) \tag{4.1}$$

where $\boldsymbol{\alpha} = \{\alpha_i\}_{i=1}^{N_m}$ are parameters to be learned from data samples, and $K_\sigma(\cdot, \cdot)$ is a Gaussian kernel with kernel width σ, i.e., $K_\sigma(\boldsymbol{x}, \boldsymbol{x}') = \exp\left\{ -\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|^2}{2\sigma^2} \right\}$. The target sliding window instances, $\boldsymbol{\mathcal{W}}_T$, works as the Gaussian centers. Each parameter α_i is associated to the i^{th} Gaussian kernel, i.e., i^{th} instance in $\boldsymbol{\mathcal{W}}_T$. We choose kernel width σ by *likelihood cross validation* following (Sugiyama et al., 2008). Any other basis functions can also be used in place of Gaussian kernels in Eq. (4.1).

Learning Parameters

The target distribution is estimated by the weighted training distribution, $\hat{P}_T(\boldsymbol{x}) = \hat{\beta}(\boldsymbol{x})P_S(\boldsymbol{x})$. The parameters $\boldsymbol{\alpha} = \{\alpha_i\}_{i=1}^{N_m}$ in model (4.1) are learned so that the Kullback-Liebler divergence from $P_T(\boldsymbol{x})$ to $\hat{P}_T(\boldsymbol{x})$ would be minimized. This leads to the following convex optimization problem-

$$\begin{aligned} &\underset{\{\alpha_i\}_{i=1}^{N_m}}{\text{maximize}} \left[\sum_{j=1}^{N_m} \log\left(\sum_{i=1}^{N_m} \alpha_i K_\sigma\left(\boldsymbol{\mathcal{W}}_T^{(j)}, \boldsymbol{\mathcal{W}}_T^{(i)}\right)\right)\right] \\ &\text{subject to } \frac{1}{N_m}\sum_{j=1}^{N_m}\sum_{i=1}^{N_m} \alpha_i K_\sigma\left(\boldsymbol{\mathcal{W}}_S^{(j)}, \boldsymbol{\mathcal{W}}_T^{(i)}\right) = 1, \\ &\text{and } \alpha_1, \alpha_2, \ldots, \alpha_{N_m} \geq 0. \end{aligned} \tag{4.2}$$

Algorithm 5 LearnAlpha: Learn DRM Parameters

Input: Source instances $\boldsymbol{\mathcal{W}}_S = \left\{\boldsymbol{\mathcal{W}}_S^{(i)}\right\}_{i=1}^{N_m}$, target instances $\boldsymbol{\mathcal{W}}_T = \left\{\boldsymbol{\mathcal{W}}_T^{(i)}\right\}_{i=1}^{N_m}$, the learning rate ϵ, and the kernel width σ.
Output: DRM parameters $\boldsymbol{\alpha} = \{\alpha\}_{i=1}^{N_m}$.

1: $\mathbf{K}^{(i,j)} = K_\sigma(\boldsymbol{\mathcal{W}}_T^{(i)}, \boldsymbol{\mathcal{W}}_T^{(j)}); \quad i, j = 1, \ldots, N_m.$
2: $\mathbf{p}^{(j)} = \frac{1}{N_m}\sum_{i=1}^{N_m} K_\sigma(\boldsymbol{\mathcal{W}}_S^{(i)}, \boldsymbol{\mathcal{W}}_T^{(j)}); \quad j = 1, \ldots, N_m.$
3: Initialize $\boldsymbol{\alpha}$.
4: **repeat**
5: Gradient ascent step:
$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + \epsilon\mathbf{K}(1./\mathbf{K})$.
6: Satisfy constraints:
$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + (1 - \mathbf{p}^T\boldsymbol{\alpha})\mathbf{p}/(\mathbf{p}^T\mathbf{p})$,
$\boldsymbol{\alpha} \leftarrow max(0, \boldsymbol{\alpha})$,
$\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha}/(\mathbf{p}^T\boldsymbol{\alpha})$.
7: **until** convergence
8: **Return** $\boldsymbol{\alpha} = \{\alpha\}_{i=1}^{N_m}$.

Algorithm (5) outlines the steps for learning the parameters $\boldsymbol{\alpha}$ for the model in Eq. (4.1). First, Gaussian kernels are calculated for all $\boldsymbol{\mathcal{W}}_T$ instances at Line 1. Next, gradient ascent is performed until convergence while satisfying the constraints at Lines 5-6. Once the set of

parameters $\boldsymbol{\alpha}$ is learned from data, importance weight for any instance $\boldsymbol{x}$ is calculated using Eq. (4.1).

Updating Parameters Online

Since data continuously arrives in source and target streams, the model in Eq. (4.1) needs to be updated also by updating $\boldsymbol{\alpha}$. Kawahara and Sugiyama have proposed an online update method of $\boldsymbol{\alpha}$ in (Kawahara and Sugiyama, 2012). However, unlike Multistream classification scenario, this method assumes only one stream of data with a sliding window to define the set of reference and test data instances. In FUSION, we adapt this method for Multistream classification scenario.

As mentioned before, $\boldsymbol{\mathcal{W}}_T$ instances act as the Gaussian kernel centers. For each $K_\sigma(\cdot, \boldsymbol{\mathcal{W}}_T^{(i)})$, there is a corresponding parameter α_i in the set $\boldsymbol{\alpha}$, which works as the weight for that Gaussian function. Therefore, if there is a new instance in the target stream, it affects the optimization problem in Eq. (4.2). So $\boldsymbol{\alpha}$ needs to be updated while satisfying constraints.

The online update method is based on the online learning technique for kernel methods proposed in (Kivinen et al., 2004). Assuming that β is searched within a reproducing kernel Hilbert space $\mathcal{H}$, the following reproducing property holds-

$$\langle \beta(\cdot), K(\cdot, \boldsymbol{x}') \rangle = \beta(\boldsymbol{x}') \tag{4.3}$$

Let $E_i(\beta)$ be the empirical error for $\boldsymbol{\mathcal{W}}_T^{(i)}$, $E_i(\beta) = -\log \beta(\boldsymbol{\mathcal{W}}_T^{(i)})$. It can be observed from Eq. (4.2) that estimated density ratio $\hat{\beta}$ is calculated by minimizing $\sum_{i=1}^{N_m} E_i(\beta)$ under the constraints. Let $\tilde{E}_i(\beta)$ be the regularized empirical error, that is-

$$\tilde{E}_i(\beta) = -\log \beta(\boldsymbol{\mathcal{W}}_T^{(i)}) + \frac{\lambda}{2} \|\beta\|_{\mathcal{H}}^2 \tag{4.4}$$

where $\lambda (> 0)$ is the regularization parameter, and $\|\beta\|_{\mathcal{H}}$ denotes the norm in $\mathcal{H}$ space.

Considering the reproducing property in Eq. (4.3), and the regularized empirical error shown in Eq. (4.4), the estimated density ratio ($\hat{\beta}$) can be updated using a new instance in the target stream, denoted by ($\boldsymbol{\mathcal{W}}_T^{(N_m+1)}$) as follows-

$$\hat{\beta}' = \hat{\beta} - \eta \partial_\beta \tilde{E}_{N_m+1}(\hat{\beta}) \tag{4.5}$$

where η is the learning rate, and ∂_β denotes partial derivative with respect to β. Since we consider Gaussian kernel model (Eq. (4.1)), replacing the partial derivative in Eq. (4.5), we get-

$$\hat{\beta}' = \hat{\beta} - \eta \left(-\frac{K_\sigma\left(\cdot, \boldsymbol{\mathcal{W}}_T^{(N_m+1)}\right)}{\hat{\beta}\left(\boldsymbol{\mathcal{W}}_T^{(N_m+1)}\right)} + \lambda\hat{\beta} \right) \tag{4.6}$$

Using the Eq. (4.1), values in $\boldsymbol{\alpha}$ should therefore be updated as follows-

$$\begin{cases} \hat{\alpha}_i' \leftarrow (1-\eta\lambda)\hat{\alpha}_{i+1} & i = 1, \ldots, N_m - 1 \\ \hat{\alpha}_i' \leftarrow \frac{\eta}{\hat{\beta}\left(\boldsymbol{\mathcal{W}}_T^{(N_m+1)}\right)} & i = N_m \end{cases} \tag{4.7}$$

Algorithm 6 outlines the online updating of $\boldsymbol{\alpha}$. As discussed before, a new instance in the target stream changes the optimization problem in Eq. (4.2). Therefore, $\boldsymbol{\alpha}$ needs to be updated along with constraint satisfaction. On the contrary, if the new instance arrives in the source stream, it does not affect the optimization problem directly. However, the constraints may be violated due to the new instance. Therefore, the constraints need to be satisfied again. Subsequently, the corresponding sliding window is updated with the new instance.

4.1.2 Training and Classification

FUSION uses an ensemble classifier, denoted as $\mathcal{M}$. We start by loading the first N_m instances from $\mathcal{S}$ and $\mathcal{T}$ into $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$ respectively, which are referred to as the warm-up period data. FUSION trains the first model in the ensemble using the warm-up period

Algorithm 6 UpdateAlpha: Update DRM Parameters

Input: Source instances $\boldsymbol{\mathcal{W}}_S = \left\{\boldsymbol{\mathcal{W}}_S^{(i)}\right\}_{i=1}^{N_m}$, target instances $\boldsymbol{\mathcal{W}}_T = \left\{\boldsymbol{\mathcal{W}}_T^{(i)}\right\}_{i=1}^{N_m}$, new instance $\boldsymbol{x}$, the kernel width σ, the regularization parameter λ, and the learning rate η.
Output: Updated DRM parameters, $\boldsymbol{\alpha} = \{\alpha\}_{i=1}^{N_m}$.

1: **if** $\boldsymbol{x} \in \mathcal{S}$ **then**
2: $\quad \boldsymbol{\mathcal{W}}_S^{(N_m+1)} \leftarrow \boldsymbol{x}$.
3: $\quad \mathbf{p}^{(j)} = \frac{1}{N_m}\sum_{i=1}^{N_m} K_\sigma(\boldsymbol{\mathcal{W}}_S^{(i+1)}, \boldsymbol{\mathcal{W}}_T^{(j)})$, $j = 1, \ldots, N_m$.
4: $\quad$ Go to Line 10.
5: **end if**
6: $\boldsymbol{\mathcal{W}}_T^{(N_m+1)} \leftarrow \boldsymbol{x}$.
7: $\mathbf{p}^{(j)} = \frac{1}{N_m}\sum_{i=1}^{N_m} K_\sigma(\boldsymbol{\mathcal{W}}_S^{(i)}, \boldsymbol{\mathcal{W}}_T^{(j+1)})$, $j = 1, \ldots, N_m$.
8: $\hat{\beta}(\boldsymbol{\mathcal{W}}_T^{(N_m+1)}) = \sum_{i=1}^{N_m} \alpha_i K_\sigma(\boldsymbol{\mathcal{W}}_T^{(N_m+1)}, \boldsymbol{\mathcal{W}}_T^{(i)})$.
9: Update $\boldsymbol{\alpha}$ using Eq. (4.7).
10: Satisfy constraints:
$\quad \boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} + (1 - \mathbf{p}^T\boldsymbol{\alpha})\mathbf{p}/(\mathbf{p}^T\mathbf{p})$,
$\quad \boldsymbol{\alpha} \leftarrow max(0, \boldsymbol{\alpha})$,
$\quad \boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha}/(\mathbf{p}^T\boldsymbol{\alpha})$.
11: **if** $\boldsymbol{x} \in \mathcal{S}$ **then**
12: $\quad \boldsymbol{\mathcal{W}}_S^{(i)} \leftarrow \boldsymbol{\mathcal{W}}_S^{(i+1)}$, $i = 1, \ldots, N_m$.
13: **else**
14: $\quad \boldsymbol{\mathcal{W}}_T^{(i)} \leftarrow \boldsymbol{\mathcal{W}}_T^{(i+1)}$, $i = 1, \ldots, N_m$.
15: **end if**
16: **Return** $\boldsymbol{\alpha} = \{\alpha\}_{i=1}^{N_m}$.

data. However, due to covariate shift between the source stream ($\mathcal{S}$) and the target stream ($\mathcal{T}$), importance weights for labeled source data should be considered in the learning process. These importance weights are estimated by the Density Ratio Estimation (DRM) module using warm-up period data from $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$ as follows-

$$\hat{\beta}\left(\boldsymbol{\mathcal{W}}_S^{(i)}\right) = \sum_{j=1}^{N_m} \alpha_j K_\sigma\left(\boldsymbol{\mathcal{W}}_S^{(i)}, \boldsymbol{\mathcal{W}}_T^{(j)}\right), i = 1, \ldots, N_m \tag{4.8}$$

Any learning algorithm that incorporates importance weight of training instances can be used in FUSION. As new instances arrive in $\mathcal{S}$ or $\mathcal{T}$, the ensemble classifier $\mathcal{M}$ is updated if there is a drift to ensure that it represents the current concepts. A new base model is trained using data in $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$ at that time. Drift detection and updating method used

by FUSION will be discussed later in this section. FUSION predicts the majority voted class in the ensemble as the class of an incoming test instance from the target stream.

4.1.3 Drift Detection Module (DDM)

As mentioned before, $P_T(\boldsymbol{x})$ is estimated by $\hat{P}_T(\boldsymbol{x}) = \hat{\beta}(\boldsymbol{x})P_S(\boldsymbol{x})$. The classifier is updated following a drift, i.e., a significant difference between $P_T(\boldsymbol{x})$ and $\hat{\beta}(\boldsymbol{x})P_S(\boldsymbol{x})$. Let $\boldsymbol{\alpha}^0$ be the set of initial parameters. These parameters are updated online as new instances arrive in $\mathcal{S}$ or $\mathcal{T}$. Let $\boldsymbol{\alpha}^t$ be the set of parameters at time t. Let $\hat{\beta}_0$ and $\hat{\beta}_t$ are density ratios defined by $\boldsymbol{\alpha}^0$ and $\boldsymbol{\alpha}^t$ respectively. The following likelihood ratio measures the deviation of the weighted training distribution from the test distribution at time t.

$$S = \sum_{i=1}^{N_m} \ln \frac{P_T\left(\boldsymbol{\mathcal{W}}_T^{(i)}\right)}{\hat{\beta}_0 P_S\left(\boldsymbol{\mathcal{W}}_T^{(i)}\right)} = \sum_{i=1}^{N_m} \ln \frac{\hat{\beta}_t\left(\boldsymbol{\mathcal{W}}_T^{(i)}\right)}{\hat{\beta}_0\left(\boldsymbol{\mathcal{W}}_T^{(i)}\right)}$$

A drift is detected if $S > -\ln(\tau)$, where τ is a user-defined parameter. It can be proved that the false alarm rate of the drift detection algorithm is bounded by τ. The efficiency of FUSION stems from the fact that in addition to estimating importance weights, it uses the same Gaussian kernel model for drift detection. Therefore, FUSION detects drift without adding any extra overhead.

Algorithm 7 sketches drift detection of FUSION. A drift is detected if the drift score S, i.e., the sum of log-likelihood ratios is greater than a pre-fixed threshold. As $\boldsymbol{\alpha}$ is updated online with any new instance in $\mathcal{S}$ or $\mathcal{T}$, both importance weight estimation and drift detection of FUSION are efficient.

4.1.4 Classifier Update

Algorithm 8 sketches the update procedure of FUSION. If a significant difference, i.e., a drift between the distributions represented by weighted source and target data is detected,

Algorithm 7 DetectDrift: Drift Detection

Input: Target instances $\boldsymbol{\mathcal{W}}_T = \left\{\boldsymbol{\mathcal{W}}_T^{(i)}\right\}_{i=1}^{N_m}$, Set of initial parameters $\{\alpha_i^0\}_{i=1}^{N_m}$, Set of current parameter $\{\alpha_i^t\}_{i=1}^{N_m}$, The kernel width σ, and The parameter τ.
Output: $True$ if drift is detected, else $False$.

1: $\hat{\beta}_0(\boldsymbol{\mathcal{W}}_T^{(i)}) = \sum_{j=1}^{N_m} \alpha_j^0 K_\sigma(\boldsymbol{\mathcal{W}}_T^{(i)}, \boldsymbol{\mathcal{W}}_T^{(j)})$ for $i = 1, \ldots, N_m$.
2: $\hat{\beta}_t(\boldsymbol{\mathcal{W}}_T^{(i)}) = \sum_{j=1}^{N_m} \alpha_j^t K_\sigma(\boldsymbol{\mathcal{W}}_T^{(i)}, \boldsymbol{\mathcal{W}}_T^{(j)})$ for $i = 1, \ldots, N_m$.
3: $S = \sum_{i=1}^{N_m} \ln \frac{\hat{\beta}_t(\boldsymbol{\mathcal{W}}_T^{(i)})}{\hat{\beta}_0(\boldsymbol{\mathcal{W}}_T^{(i)})}$.
4: **Return** $S > -\ln(\tau)$.

the Gaussian kernel model in (4.1) needs to be updated by re-evaluating $\boldsymbol{\alpha}$. Therefore, if a drift is detected, $\boldsymbol{\alpha}$ is recalculated from $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$ using Algorithm 5. Then, importance weight of each instance in $\boldsymbol{\mathcal{W}}_S$ is evaluated following Eq. (4.8) using the re-evaluated $\boldsymbol{\alpha}$. Next, a new model is trained based on instances from $\boldsymbol{\mathcal{W}}_S$ along with importance weights. Finally, the ensemble classifier $\mathcal{M}$ is updated using the newly trained model along with re-initializing $\boldsymbol{\mathcal{W}}_S$, and $\boldsymbol{\mathcal{W}}_T$. The maximum number of models $\mathcal{M}$ can contain is L. If $\mathcal{M}$ contains less than L models currently, the new model is simply added to $\mathcal{M}$. Otherwise, the least desired model in the ensemble is replaced by the new model.

Algorithm 8 UpdateClassifier: Update the Classifier

Input: Target instances $\boldsymbol{\mathcal{W}}_T = \left\{\boldsymbol{\mathcal{W}}_T^{(i)}\right\}_{i=1}^{N_m}$, DRM parameters $\{\alpha_i\}_{i=1}^{N_m}$, the kernel width σ, and threshold τ.
Output: The updated ensemble.

1: Get $\boldsymbol{\alpha}$ from $\boldsymbol{\mathcal{W}}_S$ and $\boldsymbol{\mathcal{W}}_T$ using Algorithm 5 and 6.
2: Calculate $\left\{\hat{\beta}(\boldsymbol{\mathcal{W}}_S^{(i)})\right\}_{i=1}^{N_m}$ using Eq. (4.8).
3: Train a new classifier M_n using weighted $\boldsymbol{\mathcal{W}}_S$.
4: Find the least desired model, M', in the current ensemble.
5: Update the ensemble by replacing M' with M_n.
6: **Return** the updated ensemble classifier.

As instances in $\mathcal{T}$ are unlabeled, it is not practical to find the least desired model by calculating accuracy. Rather, we calculate the confidence of a classifier on each instance in $\boldsymbol{\mathcal{W}}_T$, and replace the model having the least average confidence. We use SVM as the base

model in our experiments. A method to produce probabilistic output from an SVM model has been proposed in (Platt, 1999). We use the probability associated with each predicted class as its confidence in classification. Confidence for most classifiers can be calculated from classification metadata. For examples, the confidence of Bayesian classifier and clustering based classifiers can be estimated using associated probabilities and techniques proposed in (Haque et al., 2016) respectively.

4.2 Theoretical Analysis

In this section, first we analyze the convergence rate of density ratio generated by the Density Ratio Estimation (DRM) module. Then, we derive the time and space complexity of FUSION.

4.2.1 Convergence Rate

In order to get the convergence rate, we first prove that the error function $\tilde{E}_i(\hat{\beta})$ is a *strictly convex function*, and gradient of $\tilde{E}_i(\hat{\beta})$ is *Lipschitz* continuous and bounded. Next, we find the convergence rate of *UpdateAlpha* (Algorithm 6). Finally, we determine the convergence rate of the *Density Ratio Estimation*(DRM) module.

Lemma 1. *$\tilde{E}_i(\hat{\beta})$ is a strictly convex function, i.e., $\tilde{E}_i(t\hat{\beta}'+(1-t)\hat{\beta}) < t\tilde{E}_i(\hat{\beta}')+(1-t)\tilde{E}_i(\hat{\beta})$.*

Proof. From the definition, $\tilde{E}_i(\hat{\beta}) = -\log \hat{\beta}(\boldsymbol{\mathcal{W}}_T^{(i)}) + \frac{\lambda}{2}\left\|\hat{\beta}\right\|_{\mathcal{H}}^2$.

Since *logarithm* is a concave function, we have-

$$\log(t\hat{\beta}' + (1-t)\hat{\beta}) > t\log\hat{\beta}' + (1-t)\log\hat{\beta} \tag{4.9}$$

$$\frac{\lambda}{2}\|t\hat{\beta}' + (1-t)\hat{\beta}\|^2 < \frac{\lambda t}{2}\|\hat{\beta}'\|^2 + \frac{\lambda(1-t)}{2}\|\hat{\beta}\|^2 \tag{4.10}$$

Then from Eq. (4.9) and Eq. (4.10), we get

$$\tilde{E}_i(t\hat{\beta}' + (1-t)\hat{\beta}) < t\tilde{E}_i(\hat{\beta}') + (1-t)\tilde{E}_i(\hat{\beta}) \tag{4.11}$$

Here, we assume that $\hat{\beta}'$ is not equal to $\hat{\beta}$. □

Lemma 2. *Gradient of* $\tilde{E}_i(\hat{\beta})$ *is* Lipschitz *continuous and bounded, i.e.,* $\left\|\nabla\tilde{E}_i(\hat{\beta}') - \nabla\tilde{E}_i(\hat{\beta})\right\| \leq L\left\|\hat{\beta}' - \hat{\beta}\right\|$, *where* $L > 0$.

Proof. The gradient of $\tilde{E}_i$, $\nabla\tilde{E}_i(\hat{\beta}) = -\frac{K_\tau(\cdot,\boldsymbol{\mathcal{W}}_T^{(i)})}{\hat{\beta}} + \lambda\hat{\beta}$

Therefore-

$$\begin{aligned}\left\|\nabla\tilde{E}_i(\hat{\beta}') - \nabla\tilde{E}_i(\hat{\beta})\right\| &= \left\|-\frac{K_\tau(\cdot,\boldsymbol{\mathcal{W}}_T^{(i)})}{\hat{\beta}'} + \lambda\hat{\beta}' + \frac{K_\tau(\cdot,\boldsymbol{\mathcal{W}}_T^{(i)})}{\hat{\beta}} - \lambda\hat{\beta}\right\| \\ &\leq |\lambda|\left\|\hat{\beta}' - \hat{\beta}\right\| + \left|\frac{K_\tau(\cdot,\boldsymbol{\mathcal{W}}_T^{(i)})}{\hat{\beta}\hat{\beta}'}\right|\left\|\hat{\beta}' - \hat{\beta}\right\| \\ &\leq L\left\|\hat{\beta}' - \hat{\beta}\right\|\end{aligned}$$

Here, $|\lambda| + \left|\frac{K_\tau(\cdot,\boldsymbol{\mathcal{W}}_T^{(i)})}{\hat{\beta}\hat{\beta}'}\right| \leq L$, assuming that $\hat{\beta}, \hat{\beta}' \neq 0$. □

Theorem 1. *Assume there are positive numbers* M, D, *such that* $\left\|\tilde{E}_i(\hat{\beta}')\right\| \leq M$ *and* $\left\|\hat{\beta}' - \hat{\beta}\right\|^2 \leq D$. *Then, for step size* $\eta = \frac{1}{\gamma N}$, $\mathbb{E}[\tilde{E}_i(\hat{\beta}') - \tilde{E}_i(\hat{\beta})] \leq \frac{LQ}{2N}$, *where* $Q = max\left\{\frac{\eta^2M^2}{2\eta c-1}, \left\|\hat{\beta}' - \hat{\beta}\right\|^2\right\}$.

Proof. From Lemma 2, for any $\boldsymbol{\mathcal{W}}_T^{(i)}$, we know that
$\left\|\nabla\tilde{E}_i(\hat{\beta}') - \nabla\tilde{E}_i(\hat{\beta})\right\| \leq L\left\|\hat{\beta}' - \hat{\beta}\right\|$

Therefore, we have-

$$\tilde{E}_i(\hat{\beta}') \leq \tilde{E}_i(\hat{\beta}) + \frac{1}{2}L\left\|\hat{\beta}' - \hat{\beta}\right\| \tag{4.12}$$

$$\mathbb{E}[\tilde{E}_i(\hat{\beta}') - \tilde{E}_i(\hat{\beta})] \leq \frac{1}{2}L * \mathbb{E}\left[\left\|\hat{\beta}' - \hat{\beta}\right\|^2\right] \tag{4.13}$$

Then from Eq. (4.12) and Eq. (4.13), we can get

$$\mathbb{E}[\tilde{E}_i(\hat{\beta}') - \tilde{E}_i(\hat{\beta})] \leq \frac{LQ}{2N} \tag{4.14}$$

□

Therefore, the convergence rate for *UpdateAlpha* (Algorithm 6) is $\mathcal{O}\left(\frac{1}{N}\right)$, where N is the sample size.

The convergence rate of *LearnAlpha* (Algorithm 5) is $\mathcal{O}\left(n^{-\frac{1}{2+\gamma}}\right)$ for arbitrary small $\gamma > 0$, where n is the number of instances (Sugiyama et al., 2008). Assuming that the parameters of the DRM module are estimated initially by *LearnAlpha* algorithm using N_1 number of instances, and thereafter updated online by *UpdateAlpha* algorithm using N_2 instances ($N_2 >> N_1$), the convergence rate of DRM is $\mathcal{O}\left(\frac{1+N_1^{\frac{1+\gamma}{2+\gamma}}}{N_1+N_2}\right)$.

4.2.2 Time and Space Complexity

FUSION has four modules, i.e., *Density Ratio Estimation (DRM)*, *Drift Detection (DDM)*, *Classification*, and *Update*. DRM has two operations, one is to learn $\boldsymbol{\alpha}$ (Algorithm 5), and the other one is to update $\boldsymbol{\alpha}$ online (Algorithm 6). Time complexity to learn $\boldsymbol{\alpha}$ is $\mathcal{O}(N_m^2)$, where N_m is the size of the sliding windows. Time complexity to update $\boldsymbol{\alpha}$ is $\mathcal{O}(N_m)$. As DRM learns $\boldsymbol{\alpha}$ only once at the beginning, and updates it onward, the amortized time complexity of DRM is less than $\mathcal{O}(N_m^2)$. Time complexity of DDM is $\mathcal{O}(N_m)$. Time complexity of classification and update depends on the learning algorithm used as the base model. Therefore, FUSION has total time complexity of $\mathcal{O}(N_m^2) + f(N_m)$, where $f(N_m)$ is the time complexity for training a new model. However, amortized time complexity of FUSION is much less as $\boldsymbol{\alpha}$ is learned from data occasionally only if a data drift is detected at Line 15, or initially at Line 5 of Algorithm 4.

The space complexity of DRM is $\mathcal{O}(N_m^2)$, which dominates space complexities of other modules. Moreover, most learning algorithms have space complexity less than that. Therefore, overall space complexity of FUSION is $\mathcal{O}(N_m^2)$. Both time and space complexity of

FUSION are functions of N_m. In real-world applications, N_m can be tuned to execute FUSION within available resource.

4.3 Evaluation

In this section, we describe the experiment setup, and evaluate the proposed approach using synthetic and benchmark real-world datasets. We compare the performance of the proposed approach with a number of baseline methods.

4.3.1 Datasets

Table 4.2 lists the datasets used in the experiments. The first four datasets are from real-world, all of them are publicly available. The *ForestCover* and *Physical Activity Monitoring (PAMAP)* datasets are collected from the UCI repository as mentioned in Section 3.11.1 of Chapter 3. The *KDD* (Lichman, 2013) dataset contains TCP connection records extracted from LAN network traffic over a period of two weeks. Each record refers either to a normal connection or an attack. Without loss of generality, we use the first $200,000$ data instances from the KDD dataset. The last real-world dataset used in this chapter is Electricity (MOA, 2015), which contains data collected from the Australian New South Wales Electricity Market. In this market, the price is affected by demand and supply. The class label identifies the change of the price relative to a moving average of the last 24 hours.

SynRBF@X are synthetic datasets generated using the *RandomRBFGeneratorDrift* tool from MOA (Bifet et al., 2010) framework as mentioned in Section 3.11.1 of Chapter 3, where X is the Speed of change of centroids in the model. We generate two such datasets using $X = \{0.002, 0.003\}$ to evaluate the approaches on concept drifts having various intensities and frequencies. We generate two versions of *SynRBF@002*, using a different number of cluster centroids and classes. We normalize all the datasets used, and reshuffle the instances from different classes randomly to remove novel classes from them.

We generate a biased source stream from each dataset mentioned above using a method similar to previous studies (Huang et al., 2006; Chandra et al., 2016) as follows. First, we detect concept drifts in the dataset by employing a Naïve Bayes classifier to predict class labels, and monitoring its performance using *ADWIN*, similar to (Bifet and Gavaldà, 2007). A minibatch is constructed from data instances between the points at which *ADWIN* detects a significant change in the performance, i.e., a concept drift. Following (Huang et al., 2006), we first compute the sample mean $\bar{\boldsymbol{x}}$ of a minibatch. Next, we divide the minibatch to form the source and target minibatches. Each instance $\boldsymbol{x}$ is selected to be included in the biased source minibatch according to the probability $P\left(\xi = 1|\boldsymbol{x}\right) = \exp\left(-\frac{\|\boldsymbol{x}-\bar{\boldsymbol{x}}\|^2}{2\sigma^2}\right)$, where σ is the standard deviation of $\|\boldsymbol{x}-\bar{\boldsymbol{x}}\|$, for all $\boldsymbol{x}$ in the minibatch. Finally, we select $n\%$ of the instances in the minibatch to be included in the source minibatch, and the rest of the instances are included in the target minibatch. The source and target minibatches are concatenated together to form the source and the target stream respectively. We vary n in our experiments to introduce different level of sampling bias between the source and target streams.

4.3.2 Baseline Methods

The first baseline method we use is *Multistream Classifier (MSC)* (Chandra et al., 2016), which is the only available method in the literature for Multistream classification. MSC uses *Support Vector Machine (SVM)* as the base classifier. To implement the base classifier, we use weighted LibSVM library (Chang and Lin, 2011) with RBF kernel. Moreover in MSC, Kernel Mean Matching (KMM) (Huang et al., 2006) has been used for data shift adaptation. We evaluate the quadratic program in KMM using the CVXOPT python library (Dahl and Vandenberghe, 2008). To select the parameters of KMM, we use $B_{kmm} = 1000$, $\epsilon_{kmm} = \frac{\sqrt{N_S}-1}{\sqrt{N_S}}$, and γ_{kmm} as the median of pairwise distances in the training set, as suggested in (Chandra et al., 2016).

Although MSC is the only available method for Multistream classification, there are a number of methods available for traditional data stream classification. We use SVM and Adaptive Hoeffding Tree (AHT) (Bifet et al., 2010) on a single stream formed by combining the source and the target stream to examine if these approaches really suffer in presence of covariate shift and asynchronous drift in streaming data.

4.3.3 Setup

We implemented the proposed approach FUSION and one of the baseline methods MSC using *Python* version 2.7.6. To implement SVM and AHT, we used Weka (Hall et al., 2009) and MOA (Bifet et al., 2010) respectively. All the methods have been evaluated using a *Linux* machine with *2.40 GHz* core and *16 GB* of main memory. For a fair comparison, we have used SVM with the RBF kernel as the base classifier in the proposed approach (FUSION). As mentioned in Section 4.1.1, the kernel width (σ) for the Gaussian kernel model in FUSION is selected by likelihood cross-validation. In the experiments, we have used $N_m = 500$, and $L = 2$, and $\tau = 0.0001$. Moreover, we used regularization parameter $\lambda = 0.01$ and learning rate $\eta = 1$ following (Kawahara and Sugiyama, 2012).

4.3.4 Classification Performance

The first set of experiments are designed for comparing classification accuracy and execution time of the approaches considered on all the datasets mentioned in Table 4.2.

Classification Accuracy

Classification accuracy on different datasets have been shown in Table 4.3. The proposed approach (FUSION) clearly outperforms all the other baseline approaches. As stated before, we apply SVM and AHT on the combined stream for examining if simply combining the source and the target streams is useful. For a fair comparison, we consider that true labels

Table 4.2: Characteristics of datasets

Dataset	**# features**	**# classes**	**# instances**
ForestCover	54	7	150,000
KDD	42	23	200,000
PAMAP	53	19	150,000
Electricity	8	2	45,311
SynRBF@002-1	50	5	100,000
SynRBF@002-2	70	7	100,000
SynRBF@003	70	7	100,000

Table 4.3: Comparison of performance

dataset	FUSION		MSC		AHT		SVM	
	Accuracy	Time (Seconds)	Accuracy	Time (Seconds)	Accuracy	Time (Seconds)	Accuracy	Time (Seconds)
ForestCover	**85.10**	469.89	84.4	270.57	61.52	0.05	69.29	8.78
KDD	**97.30**	417.85	96.80	451.54	97.2	0.05	96.29	10.0
PAMAP	**99.80**	471.99	97.40	564.56	94.95	0.08	88.04	7.54
Electricity	**76.50**	238.33	74.60	601.08	75.02	0.02	73.37	0.09
SynRBF@002-1	**98.10**	415.22	93.60	533.33	85.58	0.07	86.29	8.51
SynRBF@002-2	**96.20**	561.86	69.80	232.34	83	0.13	44.13	7.72
SynRBF@003	**93.10**	591.18	58.30	194.49	80.11	0.12	41.28	8.75

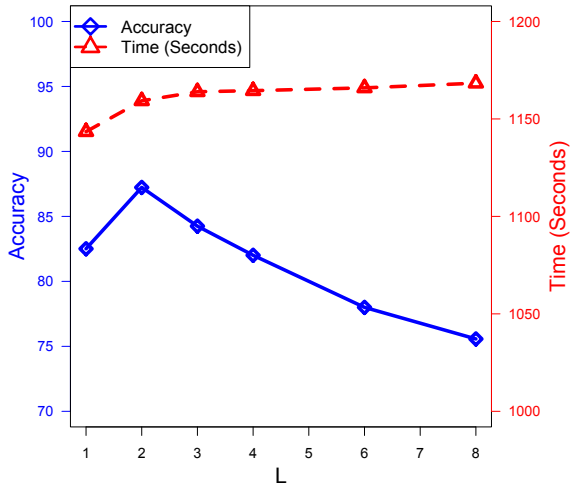
of only the source stream instances are available. Since both SVM and AHT are fully supervised models, we update the model on labeled source stream data instances once a concept drift is detected. We use ADWIN (Bifet and Gavaldà, 2007) for detecting concept drifts. Performance of these baseline methods is evaluated on unlabeled target stream data. We observe that SVM and AHT perform poorly compared to the proposed approach on both real-world and synthetic datasets. It indicates that SVM and AHT suffer on the combined stream due to not handling data shift and asynchronous data drifts as stated in Section 2.2.2. The proposed approach also outperforms MSC by a big margin especially on synthetic datasets, where we introduce frequent concept drifts. This indicates that FUSION adapts to data shift and data drifts more efficiently than MSC.

Execution Time

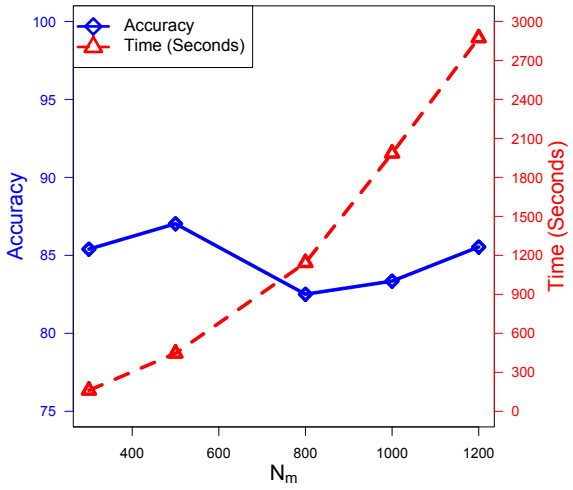
As discussed in Section 4.2.2, amortized time complexity of FUSION is less than $\mathcal{O}(N_m^2) + f(N_m)$, where N_m is the size of the sliding windows, and $f(N_m)$ is the time complexity for learning a new model. On the contrary, time complexity of MSC is $\mathcal{O}(N_m^3)$, where N_m is the maximum size of the sliding windows. Therefore, worst case time complexity of FUSION is better than MSC. Table 4.3 shows average time to process 1000 instances (in seconds) by the approaches on different datasets. We observe that FUSION achieves competitive execution time compared to MSC if not better. Online density ratio estimation, and inherent drift detection contribute to the improved performance of FUSION in terms of execution time. The other two baselines SVM and AHT shows better execution time understandably as they do not counter for data shift and asynchronous data drift adaptation.

4.3.5 Parameter Sensitivity

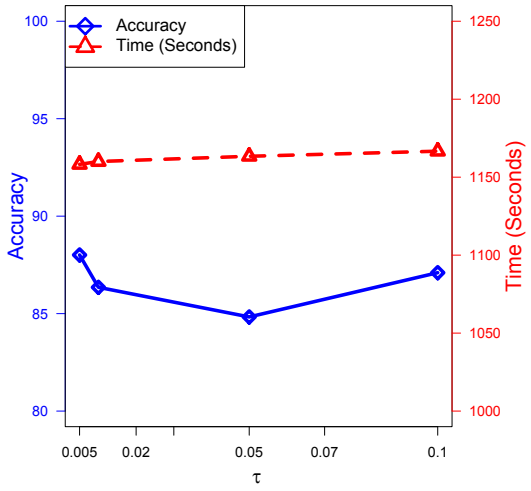
The next set of experiments are designed to examine parameter sensitivity of FUSION. In these experiments, we have used $N_m = 800$, $L = 1$, and $\tau = 0.0001$ as the default setting if not mentioned otherwise.



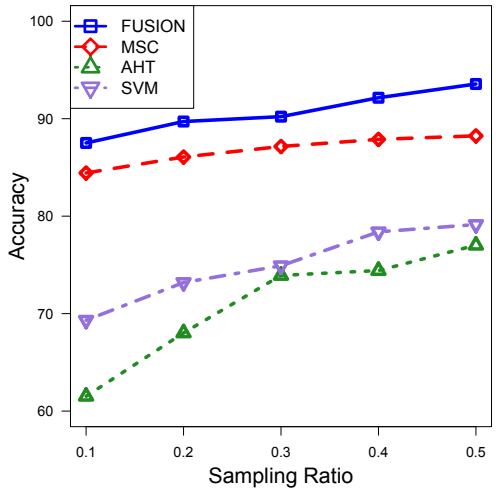
(a) Sensitivity to the ensemble size



(b) Sensitivity to the size of the sliding window



(c) Sensitivity to the drift detection parameter



(d) Sensitivity to Sampling Bias

Figure 4.2: Parameter sensitivity of FUSION on ForestCover dataset

Ensemble Size

First, we vary the ensemble size (L), and observe how it affects FUSION on the ForestCover dataset from Figure 4.2a. We observe that initially with increasing ensemble size, accu-

racy also increases. However, further increasing the ensemble size decreases the accuracy slightly, possibly due to a correlation among individual model errors (Tumer and Ghosh, 1996). The execution time of FUSION increases slightly with increasing ensemble size. As mentioned before, unlike MSC, FUSION uses an ensemble classifier containing only models trained for predicting target stream data. As a consequence, ensemble management is much lightweight in FUSION compared to MSC. Therefore, changing ensemble size does not affect the execution time of FUSION significantly.

Maximum Window Size

Next, we examine effect of window size on FUSION using the ForestCover dataset in Figure 4.2b. We observe that the accuracy remains similar with little fluctuations as the size of the sliding window (N_m) increases. However, the execution time increases with the size of the sliding window. The time complexity of FUSION is quadratic with respect to N_m as analyzed in Section 4.2.2, which is reflected in the experiment result. In real-world applications, N_m can be tuned to execute FUSION within the resource limit.

Drift Detection Parameter

Figure 4.2c shows effect of the drift detection parameter (τ) on FUSION. We mentioned in Section 4.1.3 that the false alarm rate of the drift detection algorithm is bounded by τ. We observe from the figure that as τ increases, the number of false alarms produced by the drift detection increases, and the classifier is updated using wrong data instances. Therefore, the accuracy decreases and the execution time increases with increasing τ as expected.

Sampling Bias

Finally, we observe the performance of FUSION with different sampling bias introduced in ForestCover dataset from Figure 4.2d. As discussed in Section 4.3.1, we vary sampling bias

in the dataset by varying sampling ratio between the source and the target stream. As an example, sampling ratio 0.1 means that we sample only 10% data to be labeled, i.e., included in the source stream data. The rest 90% data are considered unlabeled and included in the target stream data. Therefore, increasing sampling ratio results into decreasing sampling bias and vice versa. We observe that all the methods have better accuracy as the sampling ratio increases. However, the proposed approach FUSION exhibits the best performance. We also observe that performance of AHT and SVM improve rapidly with increasing sampling ratio, as the penalty for not handling sampling bias reduces.

To summarize, the experiments indicate that FUSION is not much sensitive to its parameters. However, it seems that choosing good values for L and N_m is vital for getting better performance. These parameters can be set by doing cross-validation on initial warm-up period data. Furthermore, as the time and space complexity of FUSION depends on N_m, it can be tuned for executing FUSION within the resource limit.

CHAPTER 5

SDKMM: SAMPLING-BASED DISTRIBUTED KERNEL MEAN MATCHING[1]

As the limitation of labeled data instances has become a reality, covariate shift due to various reasons, such as sampling bias, has also become a common problem. Recent studies have identified the sampling bias problem in areas such as natural language processing (Jiang and Zhai, 2007), computer vision (Bergamo and Torresani, 2010), and text mining (Chen et al., 2009). As mentioned in Chapter 5, traditional classifiers based on "stationary distribution assumption" greatly suffer in the presence of sample selection bias (Zadrozny, 2004). Therefore, the problem of covariate shift adaptation has attracted data mining researchers in recent years.

Most algorithms for addressing sampling bias first estimate corresponding distributions, and then apply appropriate corrections based on the estimation (Huang et al., 2006). However, estimating distribution from a multidimensional data itself is known to be a hard problem (Harel et al., 2014). There are a few approaches available in the literature that address sampling bias without estimating the biased probability densities. Kernel Mean Matching (KMM) (Huang et al., 2006) is such a approach, which estimates density ratio for each training instance $\boldsymbol{x}$, denoted by $\beta(\boldsymbol{x}) = \frac{P_{te}(\boldsymbol{x})}{P_{tr}(\boldsymbol{x})}$, directly by minimizing mean discrepancy between the training and test data distributions in a Reproducing Kernel Hilbert Space (RKHS) (Gretton et al., 2009). These density ratios, also referred to as *importance weights*, are then used to adapt the given training data for learning an appropriate model to perform prediction on test data (Chandra et al., 2016).

[1] ©2016 IEEE. Portions Adapted, with permission, from A. Haque, Z. Wang, S. Chandra, Y. Gao, L. Khan, and C. Aggarwal, "Sampling-based Distributed Kernel Mean Matching using Spark," IEEE International Conference on Big Data (Big Data), pp. 462-471, December 2016; ©2016 IEEE. Portions Adapted, with permission, from S. Chandra, A. Haque, L. Khan, and C. Aggarwal, "Efficient Sampling-Based Kernel Mean Matching," IEEE International Conference on Data Mining (ICDM), pp. 811-816, December 2016.

KMM solves a quadratic optimization program (more details in Section 5.1.2) to estimate importance weights for training data instances. This approach has a time complexity cubic in the size of training data, and linear with respect to the test data size. Therefore, despite being very useful in addressing sampling bias, KMM often becomes a bottleneck when employed in data mining operations such as data stream classification, where the classifier needs to be updated regularly (Haque et al., 2016). Moreover, computations in KMM require the whole training and test dataset to be in the memory. In scenarios where the dataset is distributed across multiple systems, one cannot directly employ KMM to perform density ratio estimations for the complete dataset.

One can use a smaller subset from the original dataset to overcome the above challenge. However, density ratio estimation depends on the distribution represented by the dataset used, i.e., data distribution of the subset in this case. Therefore, the subset generated should preserve the original data distribution. Constructing such a subset from a multidimensional data is not trivial. First, this subset may have a completely different data distribution from its original superset. Second, all data patterns in the original dataset may not be captured within this subset, which might limit its relevance in applications such as data classification.

In this chapter, we present a sampling-based approach to address the limited scalability problem in KMM. More specifically, instead of using the whole training and test dataset at once, we first generate the number of bootstrap samples of size m from the training data of size n ($m < n$). It is well established that the distribution represented by bootstrap samples closely follow the original distribution (Bickel and Freedman, 1981). Moreover, the number of training samples is determined in such a way that each instance in the original training set is selected at least once with very high probability. In addition to taking samples from the training set, we split the test dataset into a number of partitions. Then, we consider each possible pair of training sample and test partition as a train-test component. Next, we apply KMM on each of the components separately. Finally, we aggregate instance weights calculated from the components to estimate weight for each training instance.

Importantly, we show that KMM can be applied to different train-test components independently from each other. Therefore, the presented approach can be applied to large datasets in parallel and distributed fashion. It can reduce the overall time of calculating instance weights due to the small size of the components. Moreover, increasing number of training samples and test partitions, i.e., increasing number of components may help in improving quality of density ratio estimation, without increasing the execution time of this approach. Motivated by this, we propose a distributed version of the sampling approach. We implement it using *Apache Spark*, a distributed cluster computation framework. Experiment results show a significant *speed up* achieved by the distributed version while maintaining very competitive estimation accuracy.

The primary contributions of our work are as follows-

1. We present a method to address limited scalability of Kernel Mean Matching (KMM) by dividing training and test data into samples and partitions respectively, and applying KMM on each pair of sample and partition (referred to as components).

2. We show that KMM on different components can be applied independently in parallel and distributed manner. Therefore, we propose a distributed version of the sampling algorithm.

3. We implement the algorithm using *Apache Spark*. We discuss the design challenges of implementing this algorithm in a distributed environment, and propose design choices to address those challenges.

4. We thoroughly evaluate the proposed approach over benchmark datasets. The experiment results show that the proposed approach achieves significant *speed up* over the centralized algorithm. Moreover, experiment data also indicates that using a larger training data only improves the estimation accuracy, with minimal or no effect on execution time.

5.1 Background

As the proposed approach in this chapter is based on the Kernel Mean Matching (KMM) algorithm and implemented using *Apache Spark*, we provide a brief discussion on KMM and Spark, along with a list of frequently used notations in this section.

5.1.1 Notations

Table 5.1: Commonly used symbols and terms

$\mathbf{X}_{tr}$: Training data covariates	m : Size of each training sample
$\mathbf{X}_{te}$: Test data covariates	η : Sampling error tolerance
D : Domain	d : Number of dimensions
$\boldsymbol{x}$: Set of covariates (data instance)	$\beta(\boldsymbol{x})$: Weight of instance $\boldsymbol{x}$
y : Class label	k : Number of test partitions
s : Number of training samples	$\boldsymbol{\mathcal{C}}$: Set of train-test components
n_{tr}, n_{te} : Total number of train and test instances	$\boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{P}}$: Set of training samples and test partitions

Table 5.1 lists frequently used symbols in this chapter. In general, we use a bold letter to indicate a set, and a capital-bold letter to indicate a set of sets. Elements of a set are typically indexed by a subscript integer, if not specified otherwise. For example, $\mathbf{X}_{tr}$ denotes training data covariates, and $\mathbf{X}_{tr}^{(i)}$ denotes the i^{th} data instance in the training data. A hat or tilde over any symbol indicates estimated value.

5.1.2 Kernel Mean Matching

The idea in Kernel Mean Matching (KMM) is to minimize the mean distance between weighted training data distribution $\beta(\boldsymbol{x})P_{tr}(\boldsymbol{x})$ and corresponding test data distribution $P_{te}(\boldsymbol{x})$ in a Reproducing Kernel Hilbert Space (RKHS) $\mathcal{F}$ with feature map $\phi : \mathcal{D} \rightarrow \mathcal{F}$. Mean distance is measured by computing the *Maximum Mean Discrepancy* (MMD)

$$\left\| E_{\boldsymbol{x} \sim P_{tr}(\boldsymbol{x})}[\beta(\boldsymbol{x})\phi(\boldsymbol{x})] - E_{\boldsymbol{x} \sim P_{te}(\boldsymbol{x})}[\phi(\boldsymbol{x})] \right\| \tag{5.1}$$

where $\|\cdot\|$ is the l_2 norm, and $\boldsymbol{x} \in \mathbf{X} \subseteq \mathcal{D}$ is a data instance in dataset $\mathbf{X}$. Here, it is assumed that $P_{te}(\cdot)$ is absolutely continuous with respect to $P_{tr}(\cdot)$, i.e. $P_{te}(\boldsymbol{x}) = 0$ whenever

$P_{tr}(\boldsymbol{x}) = 0$. Additionally, the RKHS kernel h is assumed to be universal in $\mathcal{D}$. It has been shown that under these conditions, minimizing MMD in Equation 5.1 converges to $P_{te}(\boldsymbol{x}) = \beta(\boldsymbol{x})P_{tr}(\boldsymbol{x})$ (Yu and Szepesvári, 2012).

In particular, minimizing MMD to obtain optimal importance weights is equivalent to minimizing the corresponding quadratic program that approximates the population expectation with an empirical expectation. The empirical approximation of MMD (Equation 5.1) to obtain the desired $\hat{\beta}(\boldsymbol{x})$ is given by

$$\hat{\boldsymbol{\beta}} \approx \arg\min_{\beta} \left\| \frac{1}{n_{tr}} \sum_{\boldsymbol{x} \in \mathbf{X}_{tr}} \beta(\boldsymbol{x})\phi(\boldsymbol{x}) - \frac{1}{n_{te}} \sum_{\boldsymbol{x} \in \mathbf{X}_{te}} \phi(\boldsymbol{x}) \right\|^2 \tag{5.2}$$

where $\hat{\beta}(\boldsymbol{x}) \in \hat{\boldsymbol{\beta}}$, and $\mathbf{X}_{tr}$, $\mathbf{X}_{te}$, n_{tr}, n_{te} are training data covariates, test data covariates, size of the training data and size of test datasets respectively. The equivalent quadratic program is as follows.

$$\begin{gathered} \hat{\boldsymbol{\beta}} \approx \underset{\beta}{\text{minimize}} \frac{1}{2}\boldsymbol{\beta}^T \mathbf{K} \boldsymbol{\beta} - \boldsymbol{\kappa}^T \boldsymbol{\beta} \\ \text{subject to } \beta(\boldsymbol{x}) \in [0, B], \forall \boldsymbol{x} \in \mathbf{X}_{tr} \\ \text{and } \left| \frac{1}{n_{tr}} \sum_{\boldsymbol{x} \in \mathbf{X}_{tr}} \beta(\boldsymbol{x}) - 1 \right| \leq \epsilon \end{gathered} \tag{5.3}$$

where $\mathbf{K}$ and $\boldsymbol{\kappa}$ are matrices of a RKHS kernel $h(\cdot)$ with $K_{ij} = h(\mathbf{X}_{tr}^{(i)}, \mathbf{X}_{tr}^{(j)}) \in \mathbf{K}$, and $\kappa_i = \frac{n_{tr}}{n_{te}} \sum_{j=1}^{n_{te}} h(\mathbf{X}_{tr}^{(i)}, \mathbf{X}_{te}^{(j)}) \in \boldsymbol{\kappa}$. $B > 0$ is an upper bound on the solution search space, and ϵ is the normalization error.

5.1.3 Apache Spark

Apache Spark (Zaharia et al., 2010) is an in-memory cluster-computing platform for data analytics. It allows machines to cache data in the memory avoiding disk I/O, and reuses it in multiple MapReduce-like parallel operations. In-memory caching contribute to much faster computation by Spark compared to most MapReduce-based platforms, e.g., *Apache Hadoop*.

The main abstraction of Spark is Resilient Distributed Dataset (RDD), which is a collection of objects partitioned across a set of machines. RDDs play a central role in the fault tolerance mechanism of Spark, which maintains transformation operations on each RDD as a lineage. These lineages are recorded as centralized metadata in the master node. Therefore, if a partition is lost, Spark applies the same transformation operations on the original RDD to rebuild just that partition.

5.2 The Proposed Approach

Kernel Mean Matching (KMM) algorithm (discussed in Section 5.1.2) is sequential in nature. As mentioned in (Miao et al., 2015), it has time complexity of $\mathcal{O}(n_{tr}^3 + n_{tr}^2 d + n_{tr} n_{te} d)$, where d is the number of dimensions. In real-world applications, especially in data streams, large volume of data occur at high speed. In such scenarios, the classifier needs to be updated regularly to cope with any change of class boundaries known as a concept drift (Haque et al., 2013). Often the size of data, using which the classifier needs to be updated, are very large with possible sampling bias between training and test data distributions. KMM can be used in such scenarios for updating the classifier with sampling bias correction. However, it adds a bottleneck in the periodic update process due to its limited scalability.

In this chapter, we use the principles of KMM to present a sampling-based distributed and parallel algorithm for efficient estimation of density ratios. We refer to this algorithm by Sampling-based KMM or SKMM.

5.2.1 Sampling-based KMM (SKMM)

Given an i.i.d. set of training covariates $\mathbf{X}_{tr}$ and an i.i.d. set of test covariates $\mathbf{X}_{te}$, such that $\mathbf{X}_{tr}$ is sufficiently large, the problem is to efficiently estimate density ratio or instance weight $\beta(\boldsymbol{x}) = \frac{P_{te}(\boldsymbol{x})}{P_{tr}(\boldsymbol{x})}$ for each $\boldsymbol{x} \in \mathbf{X}_{tr}$ using the Kernel Mean Matching (KMM) method.

Since the main contributing factor to the high time complexity of KMM comes from n_{tr}, the challenge of limited scalability of KMM can be addressed naively by splitting the training data into smaller subsets, and applying KMM over each subset independently. Union of density ratio estimates from all the samples provide instance weight for each training data instance. However, such a method may not perform well, as a small subset of training data instances (chosen uniformly at random) may exhibit a glaringly different distribution compared to the original training data distribution. This can adversely affect the KMM output (Yu and Szepesvári, 2012).

The estimation of $\beta(\boldsymbol{x}) \in \boldsymbol{\beta}$ is sensitive to the training data distribution, i.e., the estimates may vary depending on the size and choice of instances used as training data. Bootstrap methods (Efron, 1992) have been shown to be extremely useful when estimators are unstable. In this scenario, one can employ a bootstrap sampling process by generating samples with replacement from the given training data. However, a naive bootstrap sample from the training data will consist of n_{tr} instances. This does not aid in improving the computational time efficiency of KMM as desired. Therefore, the *m-out-of-n* bootstrap sampling (or m/n bootstrap) method is more appropriate since $m < n_{tr}$ can be fixed. Here, m is the sample size and $n = n_{tr}$. We utilize this notion to achieve scalability for sampling bias correction.

The time complexity of KMM is linear with respect to the size of the test data, i.e., n_{te}. If n_{te} is small, KMM applied on the whole test data along with each sample from the training dataset, estimates weights for corresponding instances in the training sample. If a sufficiently large number of samples are considered, union of weights from all the samples provide weight for each instance in the training set. On the contrary, if n_{te} is also large, which is a realistic scenario in data streams, complete test data cannot fit into the memory. In such cases, KMM cannot be applied directly using the whole test data. One can address this challenge by sampling from test data also. However, sampling over the test dataset

only approximates its data distribution. Moreover, a method that partitions the test data and applies KMM on each partition independently, has been demonstrated to achieve better performance (Miao et al., 2015). Therefore, instead of sampling, SKMM divides test data into k partitions, where k is specified by the user.

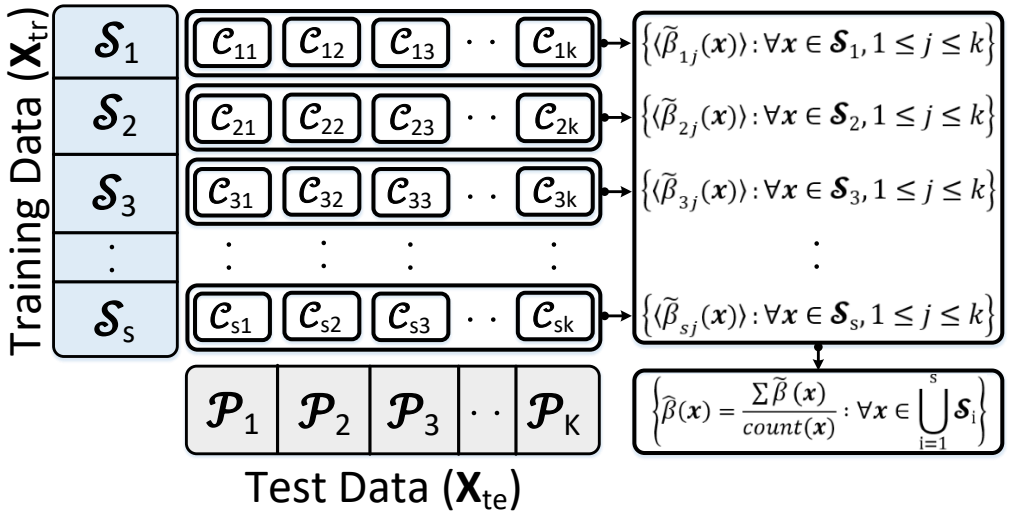


Figure 5.1: Illustration of the *SKMM* process.

Figure 5.1 illustrates this many-to-many computation scheme using training samples and test partitions. Components are formed by taking each possible pair of training sample and test partition. Since SKMM takes s samples from the training data, and creates k partitions in test data, the number of components is $s * k$. Applying KMM to each of the components results in weights estimated for instances in the corresponding training sample. Final weight for an instance is calculated by taking the average of all weights calculated for that instance from different components.

Algorithm 9 sketches the sampling-based approach. First, SKMM takes samples from the training data with replacement. Size of each sample is calculated by $m = \frac{\text{size}(\mathbf{X}_{tr})}{k}$, where k is the number of partitions in test data, which is a user input. Ideally, union of all the samples should contain each instance from the training data at least once. Since sampling is

Algorithm 9 SKMM ($\mathbf{X}_{tr}$, $\mathbf{X}_{te}$, k, η, θ)

Input: $\mathbf{X}_{tr}$: Training data covariates; $\mathbf{X}_{te}$: Test data covariates; k: Number of test data partitions; η: Sampling error tolerance; θ: KMM parameters.
Output: $\hat{\boldsymbol{\beta}}$: Estimated weights for training instances.

$m \leftarrow \frac{\text{size}(\mathbf{X}_{tr})}{k}$
$s \leftarrow \lceil \frac{\ln \eta}{m \ln\left(1-\frac{1}{n_{tr}}\right)} \rceil$ // Number of training samples
$\boldsymbol{\mathcal{S}} \leftarrow \text{genSample}(\mathbf{X}_{tr}, m, s)$
$\boldsymbol{\mathcal{P}} \leftarrow \text{partition}(\mathbf{X}_{te}, k)$
$\boldsymbol{\mathcal{C}} \leftarrow \text{cartesian}(\boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{P}})$ // Formation of Components
$\tilde{\boldsymbol{\beta}} \leftarrow \text{zeros}$
for $\forall \boldsymbol{\mathcal{C}}_{ij} \in \boldsymbol{\mathcal{C}}$, $i \leftarrow 1 \ldots s$ and $j \leftarrow 1 \ldots k$ **do**
    $\tilde{\boldsymbol{\beta}}_{ij} \leftarrow \text{KMM}(\boldsymbol{\mathcal{C}}_{ij}, \theta)$
    $\tilde{\boldsymbol{\beta}} \leftarrow \text{aggregate}(\tilde{\boldsymbol{\beta}}_{ij})$
end for
Return $\hat{\boldsymbol{\beta}} = \left\{ \hat{\beta}(\boldsymbol{x}) = \frac{\tilde{\beta}(\boldsymbol{x})}{count(\boldsymbol{x})} | \forall \boldsymbol{x} \in \bigcup_{i=1}^{s} \boldsymbol{\mathcal{S}}_i \right\}$

done randomly with replacement, inclusion of each training instance in the sampling process cannot be guaranteed. However, if a sufficiently large number of samples are taken, one can be highly confident that each instance from the training data will be selected at least once in the sampling process. We denote this confidence as $(1-\eta)$, where η is the sampling error tolerance. The minimum number of samples s to be generated can be calculated using the following Lemma.

Lemma 3. *Let s be the number of training samples generated from $\mathbf{X}_{tr}$ in SKMM, where each sample $\boldsymbol{\mathcal{S}}_i$, $i \leftarrow 1 \ldots s$, consists of m instances selected randomly with replacement from the training data. The minimum number of samples required to be generated such that an instance $\boldsymbol{x} \in \mathbf{X}_{tr}$ belongs to the set $\bigcup_{i=1}^{s} \boldsymbol{\mathcal{S}}_i$ with probability at least $(1-\eta)$ is given by* $\left\lceil \frac{\ln \eta}{m \ln\left(1-\frac{1}{n_{tr}}\right)} \right\rceil$

Proof. Probability that a data instance $\boldsymbol{x} \in \mathbf{X}_{tr}$ is not selected in any of the s independent samples, each having m independent trials, is $\left(1-\frac{1}{n_{tr}}\right)^{ms}$. Using the definition, $\eta \leq \left(1-\frac{1}{n_{tr}}\right)^{ms}$. Therefore, $s \geq \frac{\ln \eta}{m \ln\left(1-\frac{1}{n_{tr}}\right)}$. □

SKMM also splits the test data into k partitions. Let $\boldsymbol{\mathcal{P}}$ be the set of test data partitions, where $\boldsymbol{\mathcal{P}}_j$, $j \leftarrow 1 \ldots k$, denotes the j^{th} test partition consisting of $\frac{n_{te}}{k}$ test instances. Next, SKMM takes a cartesian product of set $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{P}}$, to pair each sample from the training set with each partition of the test data. We refer to each pair as a train-test component, or simply as a component. Let $\boldsymbol{\mathcal{C}}$ be the set of such components, where $\boldsymbol{\mathcal{C}}_{ij}$ is one of the components in set $\boldsymbol{\mathcal{C}}$ that consists of the training sample $\boldsymbol{\mathcal{S}}_i$ and test partition $\boldsymbol{\mathcal{P}}_j$. SKMM then applies the kernel mean matching algorithm on each of the components $\boldsymbol{\mathcal{C}}_{ij} \in \boldsymbol{\mathcal{C}}$ to calculate weights for each instance $\boldsymbol{x} \in \boldsymbol{\mathcal{S}}_i$. In this process, multiple weights may be estimated for training instances that are selected in multiple samples. Therefore, SKMM aggregates instance-wise weights denoted by $\tilde{\beta}(\boldsymbol{x})$ for all $\boldsymbol{x}$ selected in the sampling process, i.e., $\bigcup_{i=1}^{s} \boldsymbol{\mathcal{S}}_i$. Finally, SKMM outputs $\hat{\beta}(\boldsymbol{x}) \in \hat{\boldsymbol{\beta}}$, density ratio or importance weight for instance $\boldsymbol{x}$, by dividing aggregated $\tilde{\beta}(\boldsymbol{x})$ with the number of times $\boldsymbol{x}$ is selected in $\bigcup_{i=1}^{s} \boldsymbol{\mathcal{S}}_i$.

5.2.2 Sampling-based Distributed KMM (SDKMM)

It is apparent that calculation of instance weights in different components are independent from each other. In other words, KMM can be applied on different components in parallel, which makes SKMM highly parallelizable. As discussed before, time complexity of KMM is cubic in n_{tr} and linear in n_{te}. If computations on different components can be done in parallel and distributed fashion, total computation time can reduce drastically due to small sizes of a training sample and a test partition. Motivated by this, we next propose the distributed version of SKMM, referred to as Sampling-based Distributed KMM (SDKMM).

Figure 5.2 shows the workflow of SDKMM. Following the creation of components from training samples and test partitions, SDKMM distributes the components to different Mappers. Ideally, each Map function should be invoked only on one component $\boldsymbol{\mathcal{C}}_{ij}$ for calculating weights of instances in $\boldsymbol{\mathcal{S}}_i$. All the weights from Mappers are then aggregated instance-wise by a Reducer. Finally, the reducer divides the aggregated weight for each instance by the

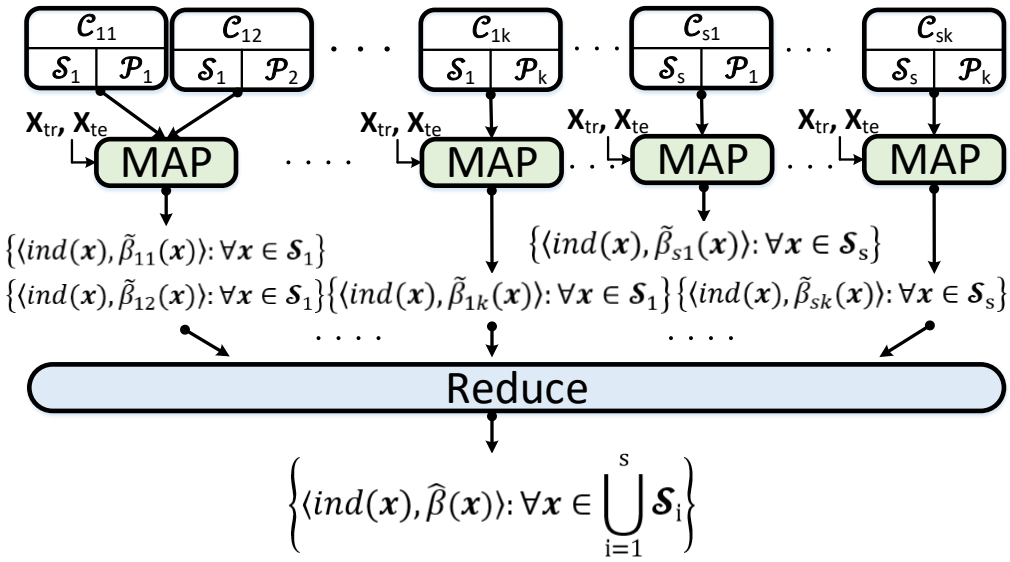


Figure 5.2: Workflow of SDKMM

number of times that instance is selected in the sampling process to calculate the final estimated weight for that instance.

Algorithm 10 details the SDKMM approach. First, it calculates the size of each training sample and the number of such samples at Lines 1-2 according to Lemma 3. Then, it takes s number of samples, each consisting of m instances selected randomly with replacement from the training data at Line 3. It also splits the test data into k partitions, and creates $s * k$ components by considering each possible pair of training sample and test partition at Lines 4-5. Next, the components are distributed over the Mappers (worker nodes) for applying KMM on these in parallel. For each component $\boldsymbol{\mathcal{C}}_{ij}$, Map function applies KMM on training sample $\boldsymbol{\mathcal{S}}_i$ and test partition $\boldsymbol{\mathcal{P}}_j$. The Map function provides output in $\langle key, value\rangle$ pairs for all $\boldsymbol{x} \in \boldsymbol{\mathcal{S}}_i$, where key is the index of $\boldsymbol{x}$ and value is the estimated weight for $\boldsymbol{x}$ from $\boldsymbol{\mathcal{C}}_{ij}$, denoted by $\tilde{\beta}_{ij}(\boldsymbol{x})$.

All these estimated weights for different $\boldsymbol{x} \in \boldsymbol{\mathcal{S}}_i$ are received by the Reducer (worker node), where $\bigcup_{i=1}^{s} \boldsymbol{\mathcal{S}}_i$ denotes the set of all training instances which are selected at least once in the sampling process. Since the instances are selected randomly with replacement, an

Algorithm 10 SDKMM ($\mathbf{X}_{tr}$, $\mathbf{X}_{te}$, k, η, θ)

Input: $\mathbf{X}_{tr}$: Training data covariates; $\mathbf{X}_{te}$: Test data covariates; k: Number of test data partitions; η: Sampling error tolerance; θ: KMM parameters.

Output: $\hat{\boldsymbol{\beta}}$: Estimated weights for training instances.

1: $m \leftarrow \frac{\text{size}(\mathbf{X}_{tr})}{k}$

2: $s \leftarrow \lceil \frac{\ln \eta}{m \ln\left(1-\frac{1}{n_{tr}}\right)} \rceil$ // Number of training samples

3: $\boldsymbol{\mathcal{S}} \leftarrow \text{genSample}(\mathbf{X}_{tr}, m, s)$

4: $\boldsymbol{\mathcal{P}} \leftarrow \text{partition}(\mathbf{X}_{te}, k)$

5: $\boldsymbol{\mathcal{C}} \leftarrow \text{cartesian}(\boldsymbol{\mathcal{S}}, \boldsymbol{\mathcal{P}})$ // Formation of Components

6: Master node distributes components over worker nodes.

7: **MAP:**

8: Each worker node calculates instance weights by applying KMM on each component $\mathcal{C}_{ij}$ (consists of $\boldsymbol{\mathcal{S}}_i$ and $\boldsymbol{\mathcal{P}}_j$) received.

9: Emit $\langle ind(\boldsymbol{x}), \tilde{\beta}_{ij}(\boldsymbol{x})\rangle$, $\forall \boldsymbol{x} \in \boldsymbol{\mathcal{S}}_i$.

10: **REDUCE:**

11: Calculate $\langle (ind(\boldsymbol{x}), \sum \tilde{\beta}(\boldsymbol{x})), count(\boldsymbol{x})\rangle$, $\forall \boldsymbol{x} \in \bigcup_{i=1}^{s} \boldsymbol{\mathcal{S}}_i$.

12: Emit $\langle ind(\boldsymbol{x}), \hat{\beta}(\boldsymbol{x}) = \frac{\sum \tilde{\beta}(\boldsymbol{x})}{count(\boldsymbol{x})}\rangle$, $\forall \boldsymbol{x} \in \bigcup_{i=1}^{s} \boldsymbol{\mathcal{S}}_i$.

13: **Return** $\hat{\boldsymbol{\beta}} = \left\{ \hat{\beta}(\boldsymbol{x}) | \forall \boldsymbol{x} \in \bigcup_{i=1}^{s} \boldsymbol{\mathcal{S}}_i \right\}$

instance can be selected in multiple samples. Therefore, the Reduce function first aggregates all these estimated weights instance-wise, denoted by $\sum \tilde{\beta}(\boldsymbol{x})$. To get the final estimated weight of a training instance $\boldsymbol{x} \in \boldsymbol{\mathcal{S}}_i$, which is denoted by $\hat{\beta}(\boldsymbol{x})$, $\sum \tilde{\beta}(\boldsymbol{x})$ is divided by the number of times $\boldsymbol{x}$ is selected in the sampling process (referred to as $count(\boldsymbol{x})$).

5.2.3 Challenges and Design Choices

In this section, we discuss some design challenges that exist in implementing Algorithm 10, and choices available to address these challenges. First, the main objective of SDKMM is to reduce execution time for addressing sampling bias. As discussed in Section 5.1.3, *Apache Spark* is much faster than other existing cluster computing framework due to in-memory caching ability. Therefore, we use *Spark* to implement SDKMM.

Second, as shown in Algorithm 10, SDKMM creates components in the master node by taking every possible combination of training samples and test partitions, and then distributes these components among the worker nodes for parallel computations. SDKMM is more efficient when large number of small-size samples are taken from the training dataset, because the samples can be processed quickly in parallel due to small size of each sample. At the same time, more samples help to estimate the distribution of the training data more accurately as discussed in Section 5.2.1. However, taking large number of samples from training data also increases number of components to a great extent. Often we observe that total size of components exceeds size of the main memory in the master machine. Therefore, instead of full data instances, we store only the indices of training and test data instances in a component. This greatly reduces the total size of the components.

Third, since only indices of instances are stored in the components, worker machines need access to the original training and test data to form the actual component before applying KMM. We share the original training and test data among the worker nodes using *Spark Broadcast* API before actual *Map* begins.

Finally, it is desired that only one component be processed per invocation of *Map* function. Spark automatically sets the number of *Map* tasks based on the number of slices in RDD (Resilient Distributed Dataset) depending on its size. However, SDKMM greatly reduces the size of RDD containing components by replacing actual data instances by indices. Therefore, using default setting does not satisfy the objective of invoking one *Map* per component. To fully utilize the cluster, we set the minimum between the number of cores and the number of components as number of slices, i.e., number of *Map* tasks.

5.2.4 Complexity Analysis

The sequential KMM approach has a time complexity of $\mathcal{O}(n_{tr}^3 + n_{tr}^2 d + n_{tr} n_{te} d)$ (Miao et al., 2015). The proposed approach (SKMM) creates components by pairing a training sample

of size m, and a test partition of size $\frac{n_{te}}{k}$. Therefore, time complexity of estimating weights for all training instances in a component is $\mathcal{O}(m^3 + m^2d + m\frac{n_{te}}{k}d)$. In case of SDKMM, time complexity to process all the components remains the same as time to process a single component due to parallel processing. Finally, the aggregation of weight estimations requires $\mathcal{O}(m)$. Together, the time complexity of SDKMM is $\mathcal{O}(m^3 + m^2d + m\frac{n_{te}}{k}d) + m$. Clearly, SDKMM achieves quicker execution time as k increases, i.e., m decreases. Similarly, the space complexity of sequential KMM is $\mathcal{O}((n_{tr})^2 + n_{tr}n_{te})$, whereas that of SDKMM is $\mathcal{O}(m^2 + m\frac{n_{te}}{k} + n_{tr})$.

5.3 Evaluation

Table 5.2: Characteristics of datasets

Dataset	**# Features**	**Total Size**
ForestCover	54	50,000
KDD	34	50,000
PAMAP	53	50,000
PowerSupply	2	29,928
SEA	3	50,000
Syn002	70	50,000
Syn003	70	50,000
MNIST	780	50,000

5.3.1 Datasets

Table 5.2 lists the datasets used in the experiments. We use two synthetic datasets, *Syn002* and *Syn003*, which are generated using MOA (Bifet et al., 2010). Others are real-world datasets, and publicly available (Asuncion and Newman, 2007; Fan et al., 2008). Since the execution time of centralized KMM increases greatly with increasing size of the dataset, we consider the first *50,000* instances from each dataset in our experiments. In order to simulate sampling bias between the training and test data, we follow a procedure similar to

a previous study (Huang et al., 2006). For each dataset, we first compute the covariate mean $\bar{\mathbf{X}}$ of all data instances, and select n_{tr} data instances with probability of $P\left(\xi = 1|\mathbf{X}^{(i)}\right) = \exp\left(-\frac{\left\|\mathbf{X}^{(i)}-\bar{\mathbf{X}}\right\|^2}{2\sigma^2}\right)$, where ξ is an indicator variable with 1 indicating selection of $\mathbf{X}^{(i)}$ as a training instance, and σ is the standard deviation of $\left\|\mathbf{X}^{(i)} - \bar{\mathbf{X}}\right\|$, $\forall \mathbf{X}^{(i)} \in \mathbf{X}$. Remaining part of the dataset is considered for testing.

5.3.2 Baseline Methods

We use two baseline methods to compare performance with our proposed approach SDKMM. The first baseline approach is the original centralized Kernel Mean Matching (KMM), which uses the whole training and test data for density ratio estimation. We denote this approach as CenKMM.

The second baseline method is denoted as EnsKMM, which is proposed by Miao et al. (Miao et al., 2015). In this approach, first the test data instances are divided into k partitions. Since $\beta(\boldsymbol{x}) \propto P_{te}(\boldsymbol{x})$, an ensemble of estimators is then obtained in EnsKMM, where each estimator estimates weights for all the training instances based on one of the test partitions and the whole training data. Finally, the estimates from individual estimators are combined to form $\hat{\boldsymbol{\beta}} = \frac{1}{k}\sum_{i=1}^{k}\hat{\boldsymbol{\beta}}_i$, where $\hat{\boldsymbol{\beta}}_i$ is the set of estimated weights from i^{th} estimator. While the study demonstrates improvements in accuracy and execution time, computational efficiently is still limited by requiring the complete training dataset in the memory. Since computations on individual estimators can be done in parallel, we implement EnsKMM also using *Apache Spark* for a fair comparison with the proposed approach.

5.3.3 Setup

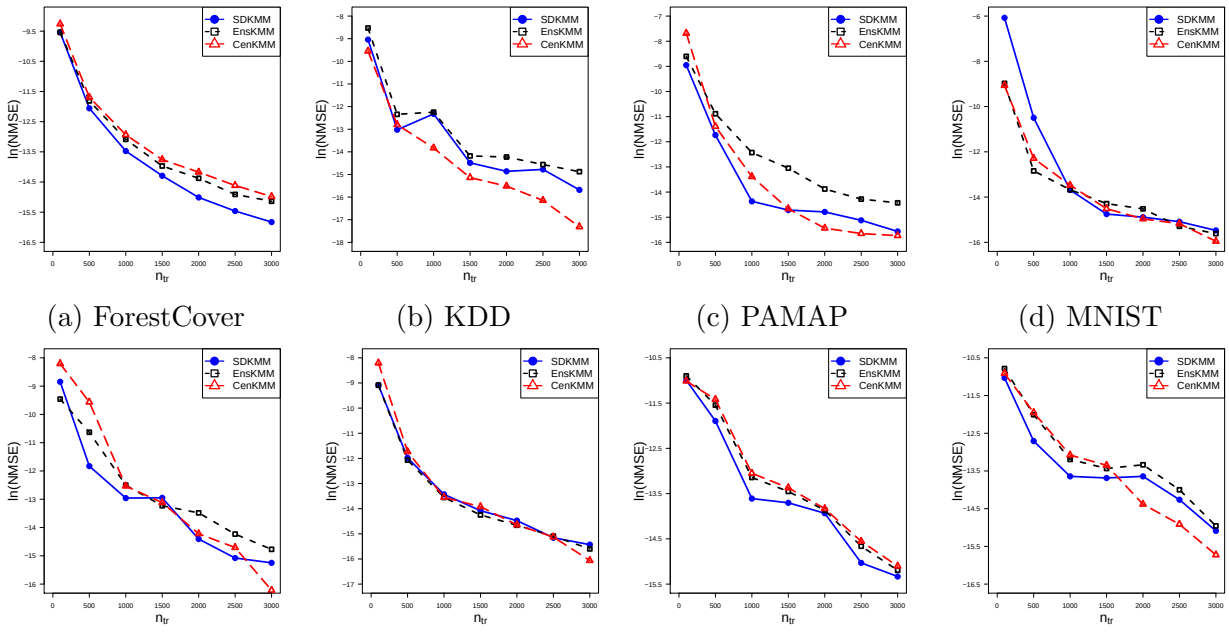
We implemented all the approaches considered in this chapter using *Python* version 2.7.5. We used the QP solver in CVXOPT python library (Dahl and Vandenberghe, 2008) to execute the KMM quadratic program, with $B = 1000$ and $\epsilon = \frac{\sqrt{n_{tr}}-1}{\sqrt{n_{tr}}}$. Following (Huang et al.,

2006), we use a Gaussian kernel with width γ equal to the median of pairwise distances. All the experiments related to SDKMM and EnsKMM were performed on a cluster running *Spark* version 1.5.1. The cluster has 12 nodes, each with eight *2.40 GHz* cores and *16 GB* of main memory. In the experiments, we have used $n_{tr} = 1000$, $k = 10$, and $\eta = 0.01$ as the default setting if not mentioned otherwise.

5.3.4 Normalized Mean Square Error (NMSE)

In the first set of experiments, we compare goodness of estimated importance weights by different approaches mentioned in Section 5.3.2. We measure goodness of estimated weights (denoted as $\hat{\beta}(\boldsymbol{x}) \in \hat{\boldsymbol{\beta}}$) by Normalized Mean Square Error (NMSE) defined as $\frac{1}{n}\sum_{i=1}^{n}\left(\frac{\hat{\beta}(\mathbf{X}^{(i)})}{\sum_{j=1}^{n}\hat{\beta}(\mathbf{X}^{(j)})} - \frac{\beta(\mathbf{X}^{(i)})}{\sum_{j=1}^{n}\beta(\mathbf{X}^{(j)})}\right)$, where $\beta(\mathbf{X}^{(i)}) = \frac{1}{P(\xi=1|\mathbf{X}^{(i)})}$, following (Miao et al., 2015). Figure 5.3 shows NMSE of all the approaches with increasing size of the training set (n_{tr}). In the experiments, we report natural logarithm of NMSE value for ease of interpretation. It can be observed that all the approaches show lower NMSE score with increasing size of the training set as expected. The proposed approach SDKMM shows very competitive performance if not better in terms of weight estimation accuracy compared with CenKMM and EnsKMM.

In the next experiment, we vary the number of partitions in the test data (k), and observe the effect on the quality of weight estimation. As discussed in Section 5.2.1, the size of each training sample m is inversely proportional to the value of k. Moreover, the number of training samples s is inversely proportional to the value of m. In other words, increasing value of k results into decreasing m and increasing s, and vice versa. Therefore, it is expected that NMSE should decrease with increasing k in case of SDKMM, due to a better estimation of the training data distribution by more bootstrap samples from training data. This is evident from Figure 5.4, which shows that SDKMM outperforms the other approaches in most of the cases with increasing value of k.



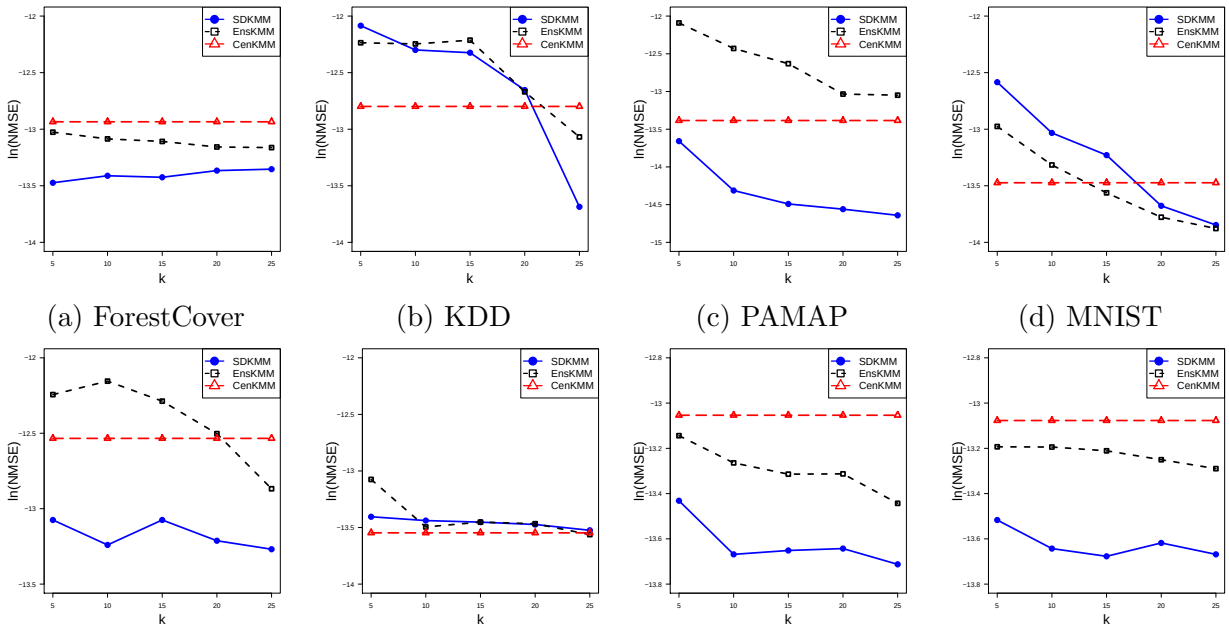
(a) ForestCover (b) KDD (c) PAMAP (d) MNIST

(e) PowerSupply (f) SEA (g) Syn002 (h) Syn003

Figure 5.3: Logarithm of NMSE with increasing size of training set (n_{tr})

5.3.5 Execution Time

Total execution time of SDKMM along with baseline approaches with increasing size of training dataset is shown in Figure 5.5. As discussed in Section 5.2.4, both CenKMM and EnsKMM have cubic time complexity with respect to the size of the training data (n_{tr}). However, in SDKMM, components are formed by taking samples from training data, and by partitioning test data. Moreover, each partition is processed in parallel. Therefore, SDKMM should have the best performance in terms of execution time among all the approaches considered, which is evident from Figure 5.5. We observe that with increasing n_{tr}, time required for estimating instance weights remain almost same in case of SDKMM due to distributed and parallel execution of components. On the contrary, time increases rapidly in case of CenKMM and EnsKMM. This is significant since in data streaming scenario, n_{tr} can be extremely large due to high speed continuous data entering into the system. SDKMM can be employed in these scenarios.



(a) ForestCover (b) KDD (c) PAMAP (d) MNIST

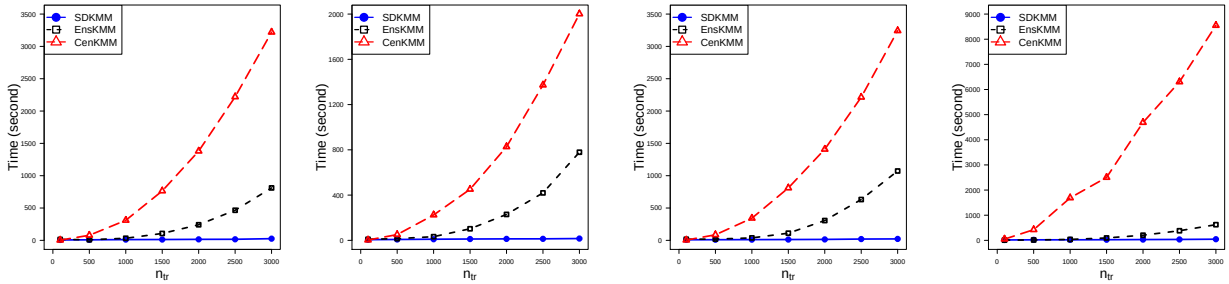
(e) PowerSupply (f) SEA (g) Syn002 (h) Syn003

Figure 5.4: Logarithm of NMSE with increasing the number of test partitions (k)

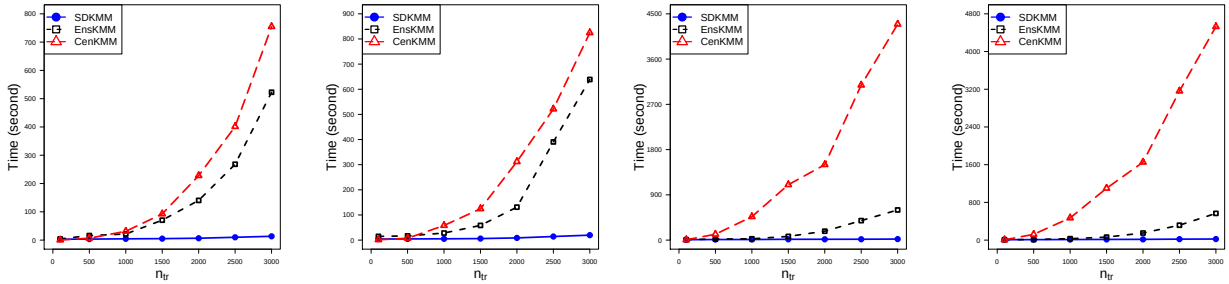
Figure 5.6 shows total time consumed by different approaches with increasing value of k with $n_{tr} = 500$. As discussed before, size of each training sample (m) and the number of training samples (s) are inversely proportional and proportional respectively with respect to k. Consequently, execution time decreases in general with increasing k due to distributed processing of smaller sized samples in case of both SDKMM and EnsKMM. However, SDKMM requires much lower time than EnsKMM due to sampling from the training data besides partitioning the test data. Both EnsKMM and SDKMM require lower time than CenKMM.

5.3.6 Speed up

We compare *speed up* achieved by SDKMM and EnsKMM on different datasets in Figure 5.7. We define speed up by $\frac{T_{sq}}{T_d}$, where T_{sq} and T_d are the execution time of sequential and distributed approach respectively on a given set of training and test data. It is clear from the

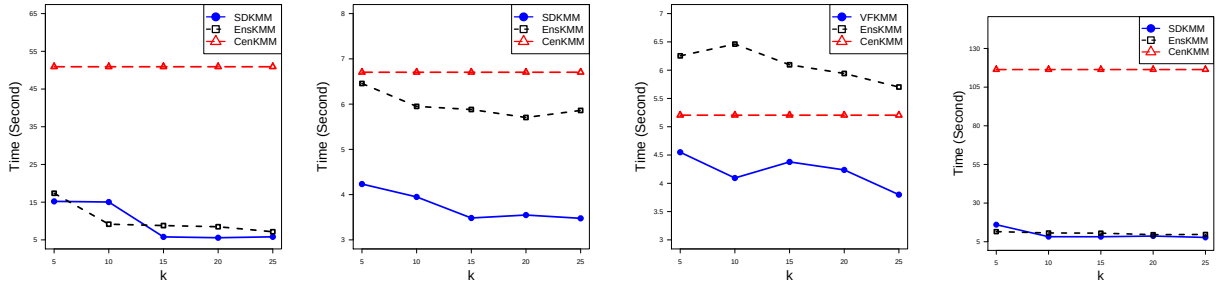


(a) ForestCover (b) KDD (c) PAMAP (d) MNIST



(e) PowerSupply (f) SEA (g) Syn002 (h) Syn003

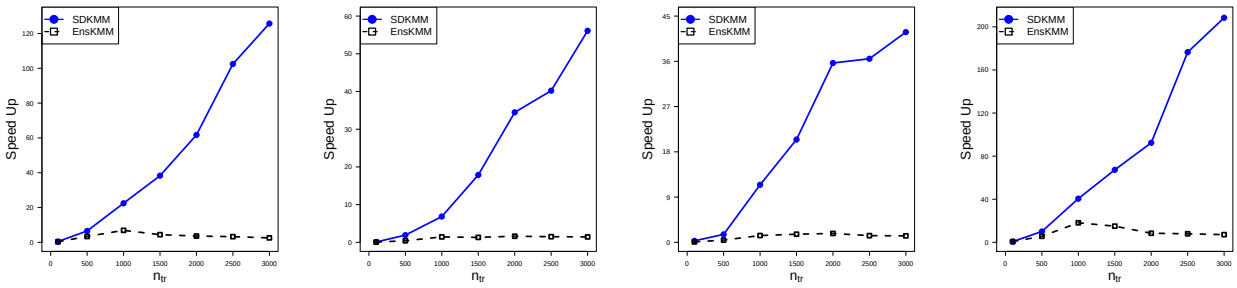
Figure 5.5: Total execution time in seconds with increasing size of training set (n_{tr})



(a) KDD (b) PowerSupply (c) SEA (d) Syn002

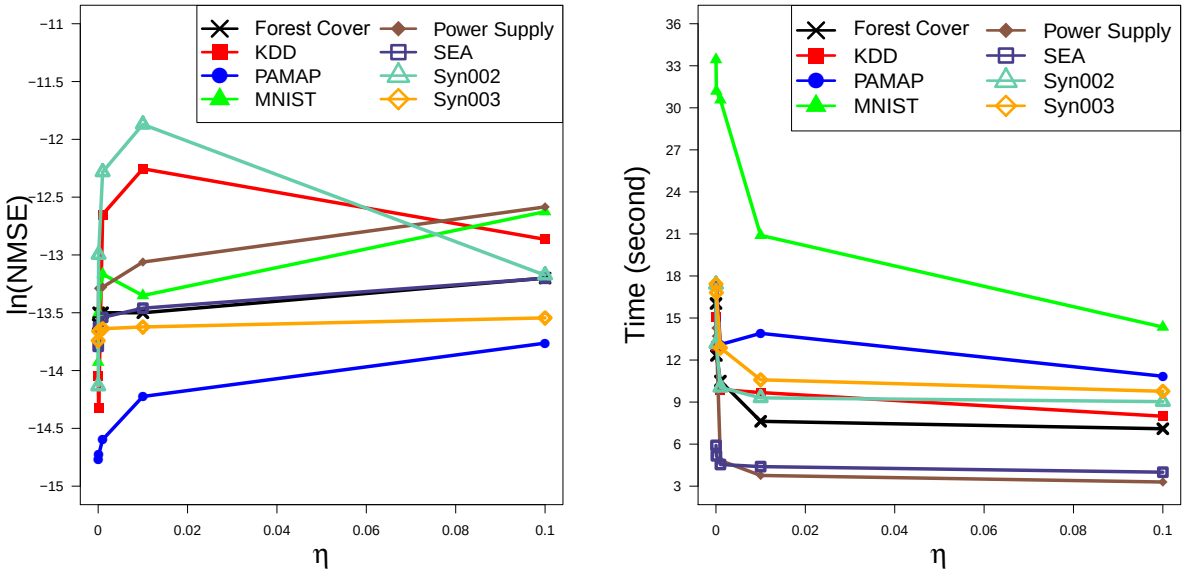
Figure 5.6: Total execution time in Seconds with increasing the number of test partitions (k)

plots that SDKMM achieves much more speed up compared to EnsKMM. More importantly, with increasing n_{tr}, speed up of SDKMM increases rapidly, whereas EnsKMM shows only a limited speed up. As evident from Figure 5.3, larger n_{tr} also results in better estimation accuracy. Therefore, both Figure 5.5 and Figure 5.7 suggest that with more training data, SDKMM provides much better estimates while requiring similar execution time. We skip identical results on some of the datasets in Figure 5.6 and 5.7.



(a) KDD (b) PowerSupply (c) SEA (d) Syn002

Figure 5.7: Speed up with increasing size of training set (n_{tr})



(a) η vs ln($NMSE$) (b) η vs Execution Time

Figure 5.8: Sensitivity to the Sampling error tolerance (η)

5.3.7 Sensitivity

Figure 5.8 shows sensitivity of SDKMM to the parameter η in terms of error and execution time. We observe from Figure 5.8a that NMSE increases in general with increasing η. On the contrary, Figure 5.8b indicates that time in general decreases with increasing η. Increasing η results in lower number of larger size samples. Therefore, increasing η affects inherent approximation of the training data distribution adversely, which results in increasing NMSE. Moreover, less number of samples results in slightly less execution time. The possible reason

is that smaller number of samples results into cheaper aggregation of weights in terms of time. In both cases, we observe that NMSE and time changes slowly with increasing η, indicating that SDKMM is not significantly sensitive to η.

CHAPTER 6

CASTLE: A DISTRIBUTED FRAMEWORK FOR DATA STREAM CLASSIFICATION[1]

With the proliferation of the Internet of Things (IoT) networks, we observe an outburst of streams of data in this modern age. It is estimated that until 2020, the size of the digital universe will double every two years (Yin and Kaynak, 2015). Due to the colossal volume and high speed, often a special term "Big Data" stream is used to specify such streams of data. These are characterized by four V's, i.e., Volume, Velocity, Variety, and Veracity.

The word "Big" in "Big data" itself defines the volume. At present, the data produced per day is in quintillion bytes range, which is expected to increase even more in nearby future (Wang et al., 2014). Systems dealing with Big Data streams receive this high volume data continuously. Moreover, data in a Big Data stream may have different types of features and unstructured data. This is why, usually data needs to undergo preprocessing before applying classification methods, which is an extra overhead. Thus, traditional data mining systems are not capable enough to classify this huge amount of data, which is constantly in motion (Zikopoulos et al., 2011).

Finally, veracity refers to uncertainty due to data inconsistency, bias, noise, feature evolution, concept drift etc. Feature evolution occurs when the features in the stream shift meaning, range, or context, or when new features are added mid-stream. In addition to these new challenges, concept drift, which is a common challenge in data stream mining, is

[1]

also present in analyzing Big Data streams. Therefore, any machine learning technique must be scalable in order to be suitable in this context.

In this chapter, we propose a scalable distributed framework for addressing the challenges in classifying Big Data streams. This framework is based on HSMiner (Parker et al., 2012), which is a multi-tiered ensemble classifier model. HSMiner builds a hierarchy of ensemble classifiers by breaking down the classification problem. At the bottom of the hierarchy, there are two distinct types of base learners for different types of features. Non-numeric features induce Naïve Bayes base classifiers. On the other hand, numeric features induce AdaBoost ensembles of linear classifiers. After receiving each new chunk of data, the hierarchical structure is updated to keep this up to date with the current concept trends.

HSMiner (Parker et al., 2012) has been described briefly in Section 6.1. As in a Big Data stream, high volume data enters continuously into the system, the size of a data chunk that needs to be processed within a unit time is large. Moreover, the Big Data streams typically have a large number of numeric features. So, HSMiner requires building a large number of AdaBoost ensembles for maintaining the hierarchical structure after receiving each new data chunk. This is an expensive process as the system needs to iterate a number of times over all data instances in the data chunk for building each AdaBoost ensemble. Therefore, HSMiner (Parker et al., 2012) may suffer scalability issue in case of Big Data stream.

The process of forming a AdaBoost ensemble for a feature is completely independent of forming the same for any other feature. So, there is scope for parallel training and maintenance of these AdaBoost ensembles. We have used this fact for addressing the limited scalability problem in HSMiner. In this chapter, we propose the scalable and distributed version of HSMiner, referred to as *CASTLE* (Haque et al., 2014). We examine three different MapReduce-based designs to form AdaBoost ensembles for different numeric features in parallel. Each of the strategies executes only one MapReduce job for building all the AdaBoost ensembles needed per data chunk. First two designs build all the feature-based

AdaBoost ensembles under a particular class in the same Map task. The third design does not have this constraint. In the third design, the task of building feature-based AdaBoost ensembles is distributed among different Map tasks regardless of class information.

The primary contributions of this work are as follows:

1. We point out scalability issue of base method HSMiner (Parker et al., 2012). To address this challenge, we identify independent components of HSMiner for applying parallelism.

2. We design three MapReduce-based distributed solutions for achieving significant Speed Up and scalability on HSMiner. We analyze advantages and disadvantages of each of these solutions.

3. We implement our proposed solutions using Apache Hadoop. We improve performance of the proposed solutions by making several design choices.

4. We evaluate the performance of CASTLE with the proposed designs on several established benchmark datasets, and compare with baseline approaches.

6.1 Background

Our proposed framework CASTLE uses three MapReduce-based designs for improving scalability of Hierarchical Stream Miner (HSMiner) (Parker et al., 2012). We briefly discuss HSMiner and MapReduce in this section.

6.1.1 Hierarchical Stream Miner (HSMiner)

The core concept behind HSMiner (Parker et al., 2012) is the use of a multi-tiered ensemble, depicted in Figure 6.1. We break down the multi-class classification problem by creating

a top-tier ensemble of per-class classifiers following a one-against-all paradigm. Each per-class classifier is an ensemble of single class classifiers. Each single class classifier is further decomposed into feature-based classifiers. The feature-based classifiers may take one of two forms. Non-numeric features are processed using the Naïve Bayes algorithm. Each numeric feature, however, is modeled using an AdaBoost ensemble of linear threshold classifiers.

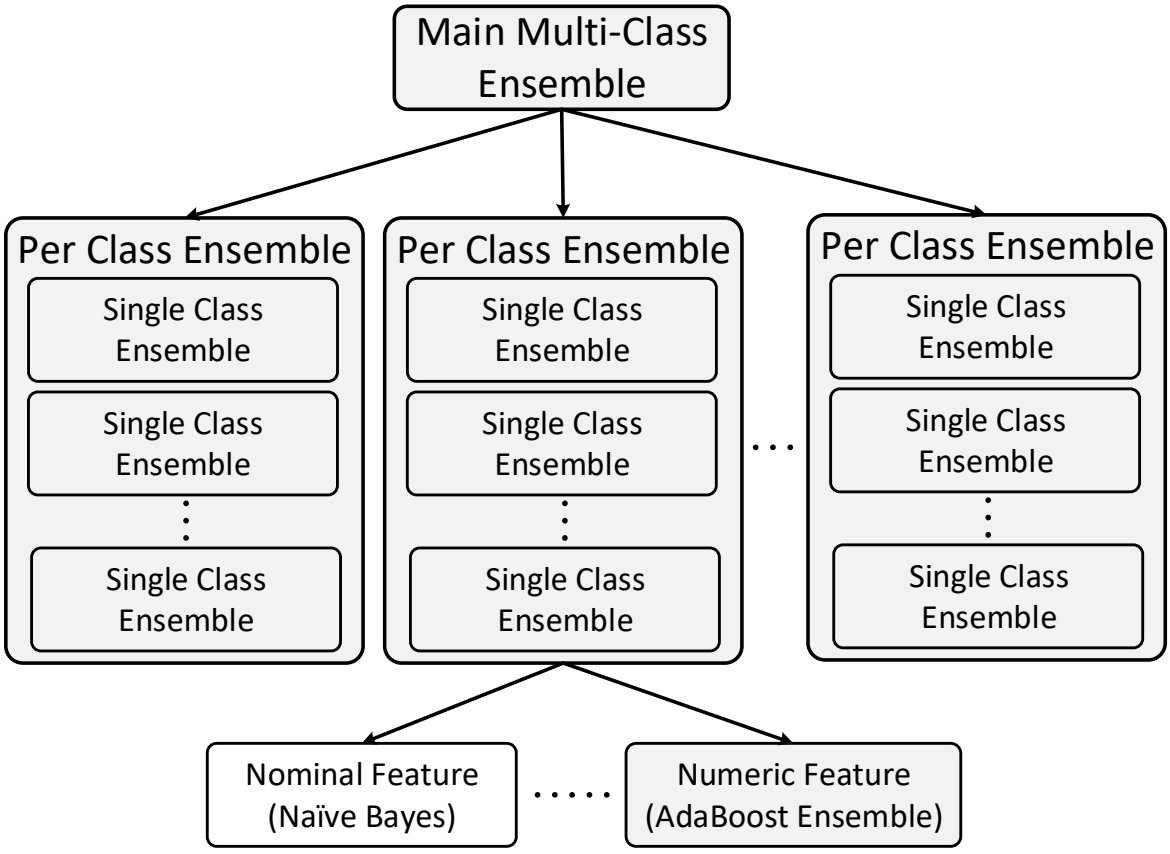


Figure 6.1: Hierarchical structure of HSMiner

As the data stream enters into the system, it is segmented into discrete data chunks for processing. After receiving a new data chunk, this hierarchical model is used to predict label for each data instance in the chunk. For the prediction, each feature-based classifier contributes a vote in a range of -1 to $+1$. Single class ensemble classifiers calculate their vote by summing up votes from contributing feature-based classifiers under it. Each per-

class classifier calculates its vote by summing up votes from all of its member single class ensembles. Finally, the class label with the highest per-class vote is predicted as the label of that data instance. A portion of the data chunk is then used as training data to update the ensembles in order to adapt to changes in the data stream. Upon completion of this routine, the next data chunk enters into the system, and the process repeats.

Training data is used to update the whole hierarchical structure. For each label present in the training data, a new single class ensemble is created under corresponding per-class ensemble. To do this, different feature-based learners under that new single class ensemble are formed as well. Thus, after receiving a new training data chunk, some new single class ensembles are added in the hierarchy under different per-class ensembles. Subsequently, these single class ensembles under different per-class ensembles are evaluated using the newest data chunk. To keep the number of single class ensembles limited, the system then prunes worst ensembles based on the evaluation. This process ensures that the overall classifier is kept up to date with the current concept trends.

HSMiner (Parker et al., 2012) offers several key contributions to address different properties of Big Data stream classification. First, it uses features in their native format without performing any data normalization preprocessing. It uses both numeric and non-numeric features equally. Second, as each class-based classifier maintains its own set of chunk-based classifiers, which in turn maintain their own set of feature-based classifiers, the features can be pruned such that only the necessary features are retained for each class. This ensures that, as features drift in the stream, each class independently adapts to the features best suited for their target label. Finally, HSMiner shows better efficiency in terms of accuracy and execution time compared to other state-of-the-art methods which try to solve the similar problem.

6.1.2 MapReduce Programming Model

MapReduce (Dean and Ghemawat, 2008) is a distributed framework and programming model for processing very large datasets. It is based on master-worker architecture. Typically, one of the nodes in the cluster works as the master node. When a job is submitted, the master node assigns input splits to different worker nodes. A worker node, which is assigned a Map task reads records from assigned splits, transforms each record into a key-value pair, and sends it to the user-defined Map function. Map function receives input as the form of key-value pairs, processes these according to functionality provided by user, and emits the intermediate key-value pairs.

Intermediate key-value pairs from Map functions are then partitioned by the partitioning function. All the key-value pairs of one partition are assigned to the same worker node, which now works as a Reducer. Reducer then arranges these intermediate key-value pairs of the assigned partition into some groups. By default, all values having the same key are grouped together to form a list of values. User-defined Reduce function is called once for each of the grouped key and associated list of values (Dean and Ghemawat, 2008). Finally, the output of the Reducer function is written into Distribute File System (DFS).

6.2 Shortcomings and the proposed Solution

As discussed in Section 6.1, HSMiner builds a number of AdaBoost ensembles for each numeric feature after receiving each new data chunk. In Big Data streams, the number of features can be very large. For example, in case of a textual stream, each distinct keyword is regarded as a feature. So in a large corpus, the number of dimensions, i.e., the number of features can be in the order of tens of thousands.

To build an AdaBoost ensemble learner for a feature, the number of iterations needed to form final ensemble classifier is equal to the number of weak classifiers we need in that

ensemble (Freund and Schapire, 1995). Moreover, at each iteration, it needs to iterate over all the data instances of the current data chunk. So, complexity for forming AdaBoost ensembles under one class-based ensemble is $O(n * w * f)$, where n is the number of data instances in the data chunk, w is the number of weak classifiers needed in the ensemble, and f is the number of numeric features, for which we need to form the ensemble. For each new data chunk, HSMiner needs to build a number of AdaBoost ensembles for each of the numeric features. Since in Big Data streams, big volume data continuously enters into the system, the size of data chunk needs to be processed within an unit time can be of a large size. Thus, our base work (Parker et al., 2012) may face limited scalability while processing test data instances in (near) real-time, especially when the number of features or size of the data chunk is large.

To address this problem, we observe that the whole process of forming AdaBoost ensembles for one feature is totally independent of forming AdaBoost ensembles for any other feature. So, AdaBoost ensembles for different features can be formed in parallel. In our proposed framework, referred to as CASTLE, we propose three MapReduce-based designs, namely Class Level Distribution (CLD), Improved Class Level Distribution (ICLD) and Feature Level Distribution (FLD). Next we elaborate details of each of these designs.

6.2.1 Class Level Distribution (CLD)

Only one MapReduce job is executed in CLD after receiving a new data chunk for building all the feature-based AdaBoost ensembles under different class-based ensembles. In this design, all the weak classifiers for AdaBoost ensembles under a specific class-based ensemble are formed in a single Map procedure.

Pseudocode of Mapper for Class Level Distribution is given in algorithm 11. The Map procedure starts with an initial uniform weight distribution to the data instances in the current data chunk (line 3). Input to the Map procedure is in $<key, value>$ format. In

Algorithm 11 MapperCLD$\{l, (f_1, f_2, .., f_m, T)\}$

1: $//D$ contains data chunk loaded from *Distributed Cache*
2: **for** $i = 1 \rightarrow m$ **do**
3: $w^1[1..n] \leftarrow$ InitializeDataInstanceWeights(n)
4: **for** $t = 1 \rightarrow T$ **do**
5: $(h^{(t)}, \alpha^t, \epsilon) \leftarrow$ LearnWeakClassifier(D, w^t, f_i)
6: $w^{t+1}[1..n] \leftarrow$ UpdateWeights$(D, h^{(t)}, w^t, \epsilon)$
7: **Emit** $(l, (f_i, \alpha^t, h^{(t)}))$
8: **end for**
9: **end for**

this case, key is the name of the class l, under which AdaBoost ensemble needs to be formed. Value is the list of all features that need AdaBoost ensembles under this class-based ensemble. For each of the features, CLD executes T number of iterations to create weak classifiers needed to form AdaBoost ensemble for that feature. At each iteration CLD forms a weak classifier $h^{(t)}$, calculates error (ϵ), and assigns weight (α) to this weak classifier. Finally, Mapper emits output as $<key, value>$ pair (line 7). In this case, the key is only the name of the class under which feature-based AdaBoost ensembles are created. The corresponding value is the combination of feature index, the weak classifier which is created in the current iteration along with its weight. In addition to emitting the output, weight of each data instance is updated for the next iteration at line 6.

Algorithm 12 ReducerCLD$\{l, List(f, \alpha_f^t, h_f^{(t)})\}$

1: $AssociativeArray \leftarrow$ LoadClassifiers()
2: **for** *all* $feature\ f \in AssociativeArray$ **do**
3: $H^{(f)} \leftarrow \sum_{t=1}^{T} \alpha_f^t h_f^{(t)}$
4: $e \leftarrow$ CalculateError$(H^{(f)}, D)$
5: $w \leftarrow$ AssignWeight$(H^{(f)}, e)$
6: **Emit** $(l, (f, H^{(f)}, e, w))$
7: **end for**

Pseudocode of Reducer for CLD is shown in algorithm 12. In Hadoop, Reducer receives key and list of values associated to that key from Mapper as input. Values associated with

the same key are processed by the same Reducer. In CLD, Reducer receives class label l as the key and a list of all the weak classifiers that are formed for different feature-based ensembles under that class label as the value.

As output key from Mapper of CLD is only the name of the class, all weak classifiers, which are part of all feature-based ensembles under that class come to the same Reducer as the list of values. This list of weak classifiers is not feature-wise sorted. This means, weak classifiers for a specific feature can be scattered throughout the list of values. So, Reducer must have to keep track of weak classifiers for different features. That is why, after receiving the list of values, Reducer sorts feature-wise weak classifiers in an associative array (line 1).

Reducer forms the final AdaBoost ensemble by taking weighted sum of all the weak classifiers for that feature at line 3. $H^{(f)}$ denotes AdaBoost ensemble for feature index f. Reducer also calculates error rate e of this new AdaBoost ensemble on current data chunk (line 4) and assigns weight w to it (line 5). Finally at line 6, Reducer emits label l as key and AdaBoost ensemble along with its error rate and weight as the value.

Output from Reducer is written in the HDFS file system, which is eventually copied to the local file system. AdaBoost ensemble formed using MapReduce-based parallelism are then extracted from the file afterwards, and used to update the whole hierarchical structure.

6.2.2 Improved Class Level Distribution (ICLD)

Recall that Reducer of CLD uses an associative array to sort the weak classifiers formed for different feature-based AdaBoost ensembles. However, use of associative array in Map or Reduce is not encouraged as it may cause memory overflow in case of Big Data stream. Moreover, this kind of in-memory sorting in Reducer can be very expensive. In order to address this problem, we propose an improved design for CASTLE, referred to as Improved Class Level Distribution (ICLD).

ICLD uses Hadoop's own secondary sorting mechanism to avoid in-memory sorting. However, Hadoop does secondary sorting only on the key of intermediate key-value pairs. In our

case, we need to sort the list of values on feature index, which is part of value. So, to do secondary sorting on values, ICLD customizes default partitioner, grouping comparator and sort comparator classes.

Algorithm 13 MapperICLD$\{l, (f_1, f_2, .., f_m, T)\}$

1: $//D$ contains data chunk loaded from *Distributed Cache*
2: **for** $i = 1 \rightarrow m$ **do**
3: $w^1[1..n] \leftarrow$ InitializeDataInstanceWeights(n)
4: **for** $t = 1 \rightarrow T$ **do**
5: $(h^{(t)}, \alpha^t, \epsilon) \leftarrow$ LearnWeakClassifier(D, w^t, f_i)
6: $w^{t+1}[1..n] \leftarrow$ UpdateWeights$(D, h^{(t)}, w^t, \epsilon)$
7: **Emit ((**l, f_i**),** $(f_i, \alpha^t, h^{(t)})$**)**
8: **end for**
9: **end for**

Pseudocode for Mapper of ICLD is shown in Algorithm 13. It computes all the weak classifiers for all input features under the input class just like Mapper of CLD. The only difference between these two Mappers is that feature index is included both in key and value of intermediate key-value pairs generated by ICLD Mapper (line 7).

The default partitioner of Hadoop computes hash value only on the key and assigns the partition to the key-value pair based on this result. All the intermediate key-value pairs belonging to the same partition go to the same Reduce function. In case of ICLD, key of intermediate key-value pair contains both label and feature index. To include all the key-value pairs having the same class label in the same partition, partitioner function needs to be applied only on the label. That is why, ICLD uses customized partitioner function that is shown in Algorithm 14. This Partitioner function extracts the label from the key, and computes the hash value based only on the label. It ensures that, all the intermediate key-value pairs having same label in key field are sent to the same Reducer.

All intermediate key-value pairs assigned to a particular Reducer are processed by series of invocation to the user defined Reduce function. GroupingComparator function decides which Map output keys within the assigned key-value pairs to the Reducer will be grouped

Algorithm 14 PartitionerICLD$\{Key, Value, numR\}$

1: //numR is the number of Reducers
2: $Label \leftarrow$ ExtractLabel(Key)
3: $HashValue \leftarrow$ HashOnLabel($Label, numR$)
4: **Return** $HashValue$

together and processed by the same Reduce function call. SortComparator function is used to sort the keys within the same group. ICLD needs to make sure that all the weak classifiers formed for different features under a single label are processed by the same invocation of the Reduce procedure. To do so, ICLD uses customized GroupingComparator to ensure all the keys having the same label are grouped together, and sent to the same invocation of Reduce function. Finally, customized SortComparator is used to sort the keys, first on the label, and then on the feature index of the key. Sorting the keys of intermediate key-value pairs makes sure that associated values are also sorted. Thus, key-value pairs assigned to a specific invocation of Reduce function are secondary sorted on values. Customized GroupingComparator and SortComparator functions are shown in Algorithm 15 and 16 respectively.

Algorithm 15 GroupingComparatorICLD$\{Key1, Key2\}$

1: $L1 \leftarrow$ ExtractLabel($Key1$)
2: $L2 \leftarrow$ ExtractLabel($Key2$)
3: $ComparisonResult \leftarrow$ Compare($L1, L2$)
4: **Return** $ComparisonResult$

Pseudocode for Reducer for ICLD is shown in Algorithm 17. Reducer of ICLD is essentially similar to that of CLD. The only difference is, in case of Reduce function of ICLD, the list of values are already sorted according to the index of features. So, all weak classifiers of a specific feature have contiguous places in the input list of values. That is why, Reduce function of ICLD doesn't need to use an Associative Array to manage weak classifiers for different features, which is very expensive. Thus, intuitively reducer of ICLD should work more efficiently than reducer of CLD.

Algorithm 16 SortComparatorICLD$\{Key1, Key2\}$

1: $L1 \leftarrow$ ExtractLabel$(Key1)$
2: $L2 \leftarrow$ ExtractLabel$(Key2)$
3: $isEqual \leftarrow$ Compare$(L1, L2)$
4: **if** $!isEqual$ **then**
5: **Return** $ComparisonOnLabel$
6: **else**
7: $F1 \leftarrow$ ExtractFeature$(Key1)$
8: $F2 \leftarrow$ ExtractFeature$(Key2)$
9: $isEqual \leftarrow$ Compare$(F1, F2)$
10: **Return** $isEqual$
11: **end if**

Algorithm 17 ReducerICLD$\{(l, f),\ List(f, \alpha_f^t, h_f^{(t)})\}$

1: **for** *all feature* $f \in List$ **do**
2: $\{(\alpha_f^t, h_f^{(t)})\} \leftarrow$ RetrieveWeakClassifiers(f)
3: $H^{(f)} \leftarrow \sum_{t=1}^{T} \alpha_f^t h_f^{(t)}$
4: $e \leftarrow$ CalculateError$(H^{(f)}, D)$
5: $w \leftarrow$ AssignWeight$(H^{(f)}, e)$
6: **Emit** $(l,\ (f, H^{(f)}, e, w))$
7: **end for**

6.2.3 Feature Level Distribution (FLD)

In the previous designs, the list of all features that need AdaBoost ensembles under a specific class is sent to a single Map task. Unlike this, in Feature Level Distribution (FLD), features are distributed over different Map tasks for building AdaBoost ensembles regardless of the class. Therefore, FLD exploits MapReduce-based parallelism more efficiently.

Pseudocode of Mapper for FLD is given in algorithm 18. In this case, Map procedure receives the name of the class l as the input key. It receives combination of a single feature index f and the number of weak classifiers needed T as the value. At each iteration, Map task builds a weak classifier along with its error rate and weight for feature-based AdaBoost ensemble under that feature and class (line 4). Finally, at the end of an iteration, Map emits the weak classifier created in the current iteration using feature f and class l at line 6. This

Algorithm 18 MapperFLD$\{l,\ (f,T)\}$

1: $//D$ contains data chunk loaded from *Distributed Cache*
2: $w^1[1..n] \leftarrow$ InitializeDataInstanceWeights(n)
3: **for** $t = 1 \rightarrow T$ **do**
4: $\quad (h^{(t)}, \alpha^t, \epsilon) \leftarrow$ LearnWeakClassifier(D, w^t, f_i)
5: $\quad w^{t+1}[1..n] \leftarrow$ UpdateWeights$(D, h^{(t)}, w^t, \epsilon)$
6: $\quad$ **Emit** $((l, f),\ (\alpha^t, h^{(t)}))$
7: **end for**

time, the key is composed of class name and the index of feature. On the other hand, the value is composed of the weak classifier itself and weight of that weak classifier.

Pseudocode of Reducer for FLD is shown in algorithm 19. As output key from Mapper has composite format containing both class name and feature index, so unlike previous designs, all the weak classifiers formed for a specific feature and class come to a single Reducer. For this reason, Reducer in FLD does not need to deal with feature-wise sorting the list of values. Moreover, it will have larger number of Reducers and thus better parallelism in Reducer level.

Reducer builds all the final AdaBoost ensembles for specified feature and class (line 1). It

Algorithm 19 ReducerFLD$\{(l, f),\ List(\alpha_f^t, h_f^{(t)})\}$

1: $H^{(f)} \leftarrow \sum_{t=1}^{T} \alpha_f^t h_f^{(t)}$
2: $e_f \leftarrow$ CalculateError$(H^{(f)})$
3: $w_f \leftarrow$ AssignWeight$(H^{(f)})$
4: **Emit** $(l,\ (f, H^{(f)}, e_f, w_f))$

also calculates the error of the AdaBoost ensemble on the current data chunk and assigns a weight to the ensemble. Finally, Reducer emits the feature-based AdaBoost ensemble along with its error and weight at line 4.

6.2.4 Design Choices and Analysis on different aspects of design

We make several choices to implement the above mentioned designs. First, as each Map and Reduce task needs the current data chunk, we put this in the *Distribute Cache* of Hadoop. It

is a facility provided by the MapReduce framework to cache files needed by the applications. The framework copies the necessary files on to the slave nodes before any tasks for the job are executed on that node. Current data chunk is loaded from the *Distributed Cache* into data structure before the actual Map begins. Size of a single data chunk is typically small. So, sharing the data chunk using *Distributed Cache* or loading it into the data structure requires small amount of memory. On the other hand, to form AdaBoost ensemble for a feature, system needs to iterate over the current data chunk for a large number of times. So, loading the dataset in the main memory makes the process considerably efficient.

Second, the number of Map tasks in a Hadoop job depends on the number of input splits. In our case, input to the Map task is the file with lines containing indexes of features, which need feature-based AdaBoost ensembles under different class-based ensembles. Though, the size of this file is not big, still processing each line of this file requires large amount of computation. If the default maximum split size of Hadoop is used, then the number of input splits as well as the number of Mappers becomes very small. As a consequence, almost all the feature-based AdaBoost ensembles are built in a single Mapper and execution time goes high. So, to customize it for our designs, we have modified maximum split size of Hadoop to increase the number of input splits i.e., the number of Mappers. In this way, performance is much improved in terms of execution time.

Input to the Mapper in CLD and ICLD designs is the list of all features that need an AdaBoost ensemble under a single class ensemble. On the other hand, input to the Mapper in the FLD design is combination of class name and index of a single feature. Therefore, all the features that need AdaBoost ensembles under different class-based ensembles are distributed along the available Mappers more efficiently. As a result, the number of Map tasks using FLD should be larger than that of other two designs. So, FLD enjoys more parallelism in Mapper level. Moreover, details of FLD from Section 6.2.3 indicates that, it should have better parallelism than all the other designs in Reducer level too.

From the above discussion, it is clear that, Intuitively FLD should perform better than the other designs in a cluster environment, especially when the number of numeric feature is large. In the next Section, we present experimental data that also supports this analysis in terms of execution time and Speed Up.

6.3 Evaluation

In this section, we present the empirical results to illustrate efficiency and scalability achieved by the proposed framework (CASTLE). First, we present the performance of CASTLE in terms of classification accuracy. Next, we compare the execution time of the proposed designs on different datasets.

Table 6.1: Characteristics of datasets

Dataset	# Features	# Classes	Total Size
ForestCover	54	7	581,000
PAMAP	52	19	3,850,505
Synthetic10	100	10	500,000
Synthetic15	100	15	500,000

6.3.1 Datasets

We have used several benchmark real-world and synthetic datasets for evaluating the proposed framework. Table 6.1 depicts the characteristics of the datasets. We use the *ForestCover* and the *Physical Activity Monitoring (PAMAP)* as real-world datasets for evaluating the proposed approach and the baselines. These datasets have been introduced in Section 3.11.1 of Chapter 3. For the experiments in this chapter, however, we have used all the instances from these datasets.

The other two datasets we use in the evaluation are synthetically generated using the *RandomRBFGeneratorDrift* tool from the *MOA* (Bifet et al., 2010) framework. We generate

two versions of this dataset. In the first version, referred to as *Synthetic10*, we generate total 10 classes, each having 100 numeric features. On the contrary, we increase the number of classes to 15 in the second version of this synthetic dataset, referred to as the *Synthetic15* dataset. We intentionally vary the number of classes between the two versions in order to compare the performance of the approaches in terms of both accuracy and execution time.

6.3.2 Setup

We have used JDK version 1.8.0.77 for implementing the sequential HSMiner method. For MapReduce implementation, we have used Hadoop version 2.7.4. We have evaluated the implementation of the proposed designs using a cluster with 12 nodes, where each node has eight 2.40 *GHz* cores, and 16 *GB* of main memory.

We implement the three MapReduce-based designs proposed in Section 6.2 for the proposed framework, referred to as *CASTLE-CLD*, *CASTLE-ICLD*, and *CASTLE-FLD*. In order to compare the classification result, we have used *DXMiner* (Masud et al., 2011) as the baseline, since it is an efficient framework for classifying instances from an evolving data streams, and focuses on solving many of the problems similar to the proposed framework.

Table 6.2: Comparison of classification performance on different datasets

Classification Method	ForestCover	PAMAP	Synthetic10	Synthetic15
CASTLE	7.9%	**2.36%**	**7.6%**	**8.4%**
DXMiner	**5.2%**	48.3%	32.4%	39.49%

6.3.3 Classification Accuracy

In the first set of experiments, we compare the performance of the proposed approach (CASTLE) with the baseline approach DXMiner. Table 6.2 shows the the classification performance by these approaches. We observe that CASTLE shows significantly better accuracy in case of PAMAP and synthetic datasets. On the other hand, it shows competitive accuracy

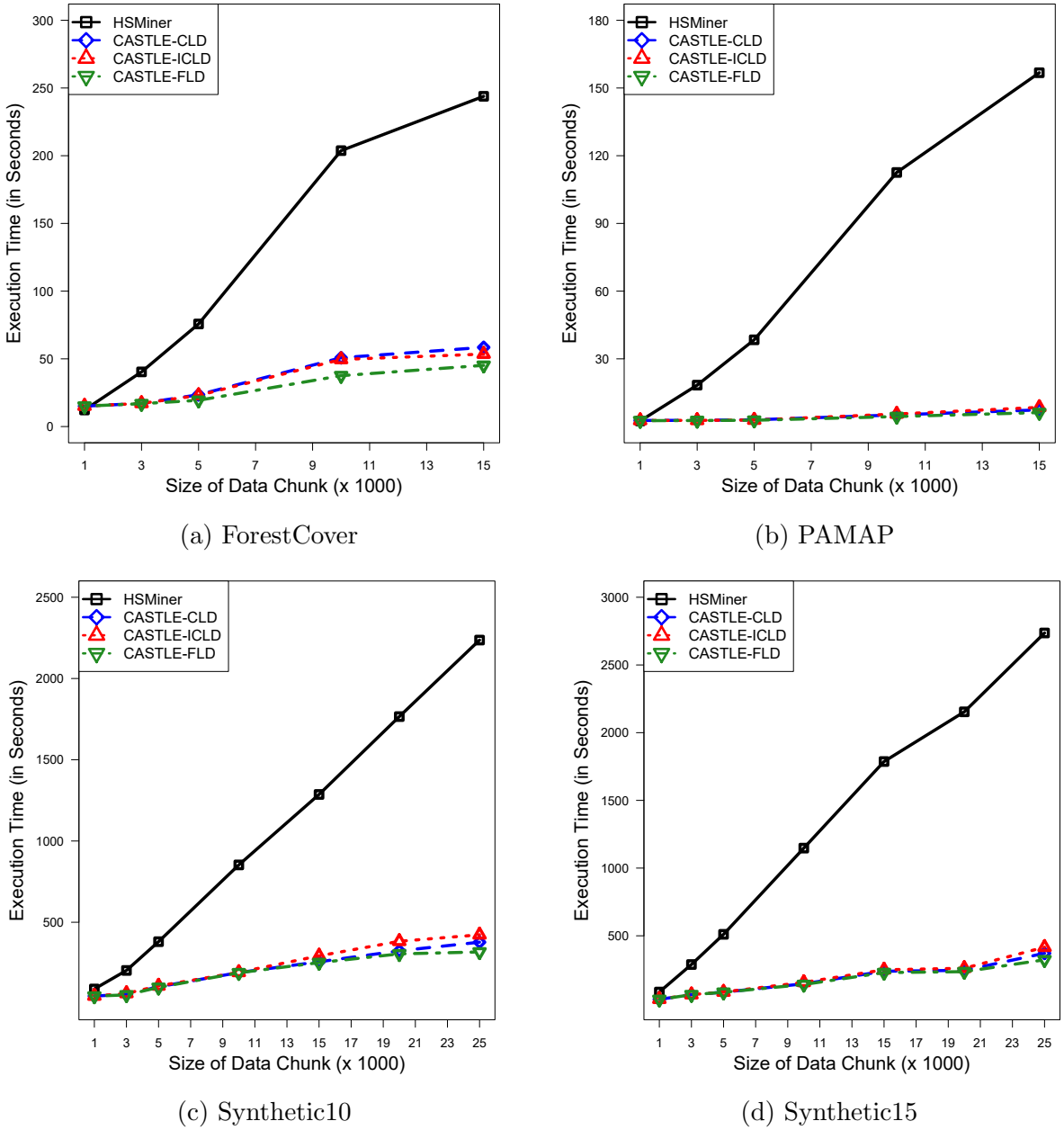
in case of ForestCover dataset. Please recall that CASTLE has the same principles as the base method HSMiner. As the proposed designs for CASTLE do not loose any accuracy, these also show the same accuracy as HSMiner. More comparison between HSMiner and DXMiner in terms of other performance metrics can be found in (Parker et al., 2012).

6.3.4 Execution Time

We compare the execution time of the proposed framework with all the three MapReduce-based designs with the sequential HSMiner in Figure 6.2. In this experiment, we use the cluster in order to evaluate the execution time of the proposed designs. Since small data chunk size often leads to bad model due to underfitting problem, and too large size of data chunks may cause the overfitting problem, we have used different size of data chunk to evaluate performance of the proposed designs.

We observe that the execution time of the sequential HSMiner increases almost linearly with increasing chunk size. Initially for small size of data chunk, basic HSMiner shows better performance. However, with increasing size of chunk, MapReduce-based frameworks show significantly better performance in terms of execution time. The synthetic datasets have a larger number of numeric features and therefore CASTLE requires lot more AdaBoost ensembles to be formed on these datasets. So, the difference in execution time between basic HSMiner and CASTLE with different MapReduce-based designs is more evident in case of the synthetic dataset. This result on execution time is not surprising as Hadoop has its own overhead that contributes to its greater execution time initially when the chunk size is not large. As the chunk size gets larger, this overhead becomes well-paid and MapReduce implementation shows better execution time.

ICLD shows better performance than CLD on all datasets. Please recall that ICLD uses Hadoop's secondary sorting mechanism to avoid feature-wise sorting on the list of values. Therefore, ICLD has less memory requirement and achieves better execution time. On the



(a) ForestCover

(b) PAMAP

(c) Synthetic10

(d) Synthetic15

Figure 6.2: Comparison among Basic HSMiner and CASTLE in terms of execution time per chunk

other hand, CLD shows slightly better performance than ICLD in case of PAMAP dataset. PAMAP has larger number of classes and similar number of numeric features compared to ForestCover. Therefore, in this case, extra computational and network overhead for shuffling

larger number of intermediate key-value pairs in ICLD overcome the cost for feature-wise sorting in CLD. However, ICLD is still preferable than CLD for lower memory requirement.

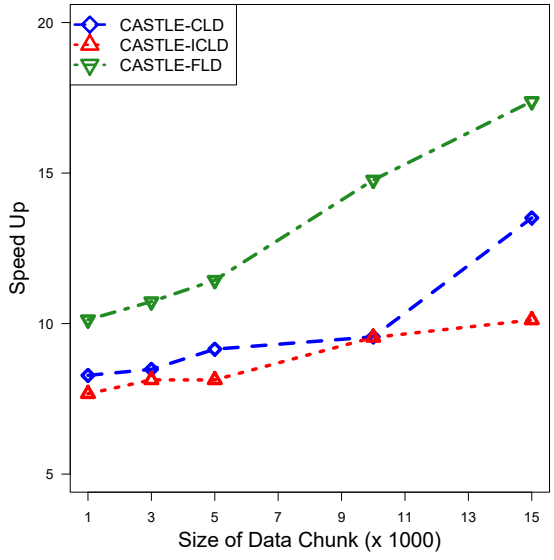
In all cases, CASTLE with FLD outperforms both CASTLE with CLD and ICLD. This supports the analysis given in Section 6.2.4. In FLD, all the features are distributed among available Mappers for building AdaBoost ensembles on those features. So, MapReduce-based parallelism is better exploited in FLD. As a result, FLD shows better performance than the other two designs on all datasets.

6.3.5 Speed Up

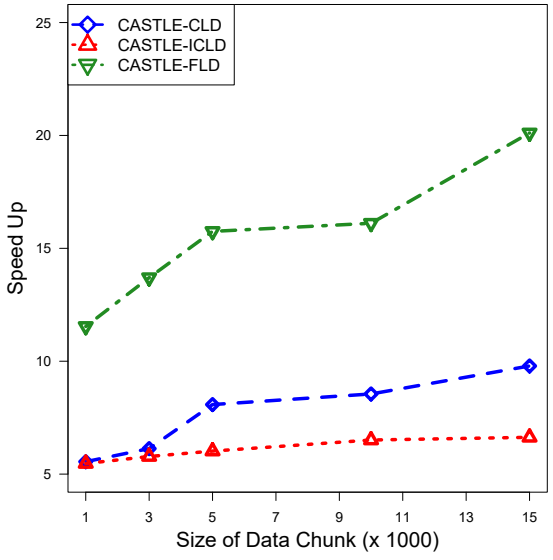
In the next set of experiments, we compare the *Speed Up* achieved by the variants of CASTLE. We define Speed Up by $\frac{T_{sq}}{T_d}$, where T_{sq} and T_d are the time taken by the single machine execution, and the cluster execution respectively. As shown in Figure 6.3, we observe that all the variants of CASTLE achieve significant Speed Up as expected. Moreover, the Speed Up increases with increasing size of chunks.

It is evident that CASTLE-FLD achieves the highest Speed Up among all the variants, which is even more significant on the syntehtic datasets. For example, cluster execution of CASTLE-FLD is around 40 times faster than the single node execution on the Synthetic15 dataset with chunk size 25×10^3. The result is expected as FLD distributes data instances among the worker nodes both horizontally and vertically. Therefore, it capitalizes the parallelism better than other variants of the proposed framework. The difference in performance becomes even more visible with increasing number of features and classes.

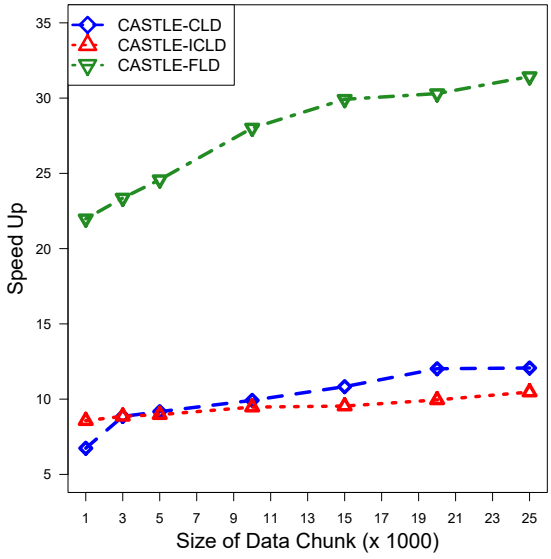
It is apparent from the experiment results that all the variants of CASTLE reduce execution time of HSMiner to a great extent. Moreover, the Speed Up achieved by different variants of CASTLE indicates that it would be useful in addressing the limited scalability problem in data stream mining, especially when the number of features is very large.



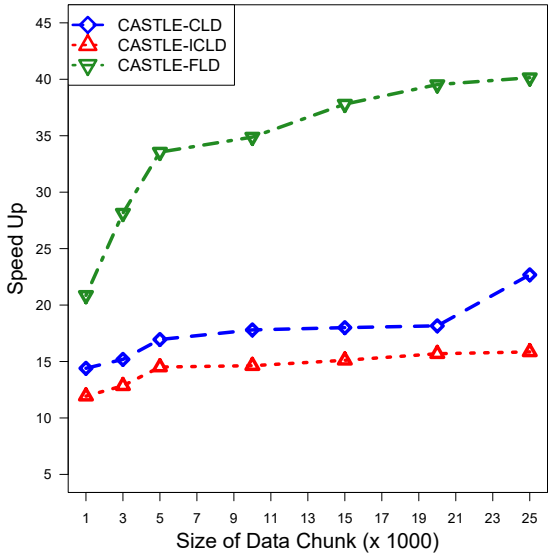
(a) ForestCover



(b) PAMAP



(c) Synthetic10



(d) Synthetic15

Figure 6.3: Speed Up achieved by CASTLE

CHAPTER 7

FUTURE WORK

In this chapter, first we briefly discuss the approaches presented in the dissertation. Subsequently, we provide some future directions on these approaches.

7.1 Discussion

The challenges in data stream mining stem from its own inherent properties, such as infinite length, change in the underlying concepts, the emergence of a novel class, limited labeled data, and sampling bias between the training and the test distribution. Moreover, in this age of big data and Internet of Things (IoT) streams, scalability of an approach is also very important to fit into the requirement. In this dissertation, we have proposed four paradigms for addressing all these challenges. Next, we summarize each of them under respective headings.

7.1.1 ECHO

We have proposed the first approach, referred to as ECHO, for handling infinite length, concept drift and appearance of novel classes over the stream data in Chapter 3. ECHO uses an ensemble classifier, where each model is trained using a semi-supervised clustering method. It employs two estimators for estimating the classifier confidence in addition to classifying each test data instance. ECHO detects any change in the underlying concepts in data by monitoring the classifier confidence. Moreover, it selects important data instances for updating the classifier using the estimated confidences. Once a concept drift is detected, a new model is trained on the recent partially labeled data instances, and the ensemble classifier is updated. It also detects any novel class in the stream by analyzing the outliers. Empirical results indicate that ECHO achieves competitive accuracy, if not better, compared to the baseline fully-supervised approaches using partially labeled data.

7.1.2 FUSION

We have proposed FUSION for efficient Multistream classification in Chapter 4, where unlabeled test data from a target stream needs to be classified using labeled training data from a source stream. The main challenges of Multistream classification are data shift, and asynchronous concept drifts between source and target stream data. To address these challenges, FUSION uses an ensemble classifier, where each model is trained using weighted instances from the source stream. The density ratio between the target and the source distribution, which is estimated by a Gaussian kernel model, is used as the importance weight for the source instances. The same weights are also used for addressing asynchronous concept drifts. Experiment results show the effectiveness of the proposed approach.

7.1.3 SDKMM

In Chapter 5, we have proposed a sampling-based distributed and parallel approach that computes density ratios between training and test data distributions efficiently using the Kernel Mean Matching (KMM) algorithm. These density ratios can be used for correcting sampling bias in data. Empirical results indicate that the proposed approach is effective in addressing the limited scalability problem in the original KMM algorithm. Therefore, it can be very useful for addressing sampling bias problem in real-world data mining applications on large volume of data.

7.1.4 CASTLE

We have proposed a scalable and distributed framework, referred to as CASTLE, for addressing various challenges in Big Data and IoT stream mining in Chapter 6. This framework is based on HSMiner, which is a hierarchical classification model for data stream classification (Parker et al., 2012). The bottleneck of this method is that it needs a large number of

feature-based AdaBoost ensembles for numeric features in data. Therefore, it may face limited scalability issue in case of Big Data stream, where the size of data chunk or the number of features is large. We address this challenge by proposing three MapReduce based designs for CASTLE. We have shown that all of these designs help to achieve significant Speed Up over the base method. We have also analyzed different aspects of the designs. Results from experiments also supports these analyses.

7.2 Future Directions

In the future, the performance of the proposed approaches in this dissertation can further be improved in terms of execution time and memory requirement. Moreover, these approaches can be applied to various domains in order to address challenges that are commonly present in data stream mining. Next, we discuss some possible future extension to the proposed approaches.

7.2.1 Ensemble FUSION

We introduced Multistream classification framework FUSION in Chapter 4. It uses a Gaussian kernel model for estimating the importance weights, i.e., the density ratios. We have analyzed the theoretical convergence rate of the approach. Empirical results show very promising performance by FUSION in terms of classification accuracy. However, FUSION is comparatively slower than the state-of-the-art approaches for analyzing a single stream due to the execution of the density ratio estimation. In other words, the bottleneck in FUSION is the batch learning of the Gaussian kernel model parameters. The parameters are learned using the batch algorithm initially, when a new model is trained following a concept drift, and then updated online. The time complexity of the batch algorithm is $\mathcal{O}(n^2)$, where n is the size of the sliding window.

We presented SDKMM in Chapter 5, in order to address the limited scalability problem in the Kernel Mean Matching (KMM). In SDKMM, first, we split the test data and take bootstrap samples from the training data to form train-test components. Next, we compute instance weights from each component independently and in parallel. Finally, we aggregate individual component outputs for calculating the final result. A similar idea can be applied to improve the time complexity of the batch parameter learning algorithm of the Gaussian kernel model used in FUSION. One can create similar train-test components, maintain an ensemble of Gaussian kernel models on such components, and update the ensembles with each incoming target instance. Since computation on different models in the ensemble can be done in parallel, and the size of the window n would be much smaller for each component model, it would significantly improve the time complexity of the overall process. Moreover, due to aggregating the importance weights calculated from different models, this approach should reduce the variance in estimated density ratio as well, and thereby possibly could improve classification performance.

7.2.2 Multistream Regression

We focus on predicting the class of unseen test data from the target stream using the labeled data from the source stream in the Multistream classification problem. An interesting research direction could be examining how significantly covariate shift, e.g., sampling bias affect a regression scenario, where instead of predicting a category, a real value for the target variable is predicted. The subsequent question to answer in this direction is how to define and handle the covariate shift and the asynchronous concept drift in regression.

7.2.3 Multistream Domain Adaptation

In the Multistream classification problem, we assume two streams of data, i.e., source and target streams, where the source stream generates only labeled training data, and the target

stream generates unlabeled test data. Moreover, although these streams generate data from the same domain, the data distributions are not the same, but related by a covariate shift. This type of scenarios may arise due to difficulty in collecting training instances with true labels.

Although Multistream classification resembles the covariate shift problem in the streaming setting, it does not cover another realm of data adaptation, known as the *Domain Adaptation* or the *Transductive Transfer Learning* problem (Pan and Yang, 2010). In the Multistream version of this problem, instead of producing data from the same domain, the source and the target streams may generate data from different domains. This setting resembles the scenarios, where data instances are very scarce in the target domain, but are abundant in a different but related domain. Despite having a number of researches on domain adaption using fixed size training and test data (Pan and Yang, 2010), application of this in data stream mining has been mostly ignored. An obvious extension of the Multistream classification is relaxing the constraint that the source and the target stream generate data from the same domain.

7.2.4 Zero-day Attack Detection

Intrusion Detection System (IDS) is very important for safeguarding the computer networks from malicious activities or policy violations. In IDS, patterns found in past malicious attacks are analyzed for detecting any future attacks. Like most data streams, a major difficulty in designing a IDS is the scarcity of labeled attack data instances, as it requires time and many often human effort to confirm an attack on the network. In order to overcome this scarcity of attack data problem, a number of systems have been proposed, for example *Honeypots* (Vasilomanolakis et al., 2015), *Honey-Patching* (Araujo et al., 2014). In these systems, the attacker is deceived as if the attack was successful, and more attack data is collected as the attacker continues the attacks. However, based on the services, the location,

or the type of the network, some specific types of attacks may be carried out more often than other types of attacks. It may introduce a sampling bias in the collected training data with respect to the test data.

The above resembles the aspects of the Multistream classification problem. In addition to the regular intrusion detection, another important problem is to detect previously unseen cyber attacks before they reach any unpatched vulnerable computer systems, known as the *zero-day Attack Detection*. The idea presented in Section 3.3 of Chapter 3 along with the principles of FUSION could be used for effectively addressing the regular intrusion detection and the zero-day attack problems, especially when the training data is biased with respect to the test data.

7.2.5 Political Unrest Prediction

Predicting civil and political unrest ahead of time using various sources, such as tweets, news articles, blogs, etc. have become an attractive research field recently (Ramakrishnan et al., 2014). Civil and political unrest, especially the violent ones often claim a lot of property damages, and even human lives. Therefore, the main motivation of this research is to prevent loss of property and instability in the society. Although a lot of information about the world and social affairs can be gathered from the sources mentioned above, collecting labeled training data is still an issue. Furthermore, not every region, country, or ethnicity have a similar presence in these sources. For example, some regions, such as the north-america and middle-east-asia, may have a greater presence in news articles than the others. Moreover, distribution of news may change based on the time period considered for collecting the training data. All these factors may introduce sampling bias in the training data. Therefore, predicting civil and political unrest using sampled labeled data would be an interesting case-study for the techniques proposed in this dissertation.

REFERENCES

Aggarwal, C. C. (2006). On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pp. 607–618. VLDB Endowment.

Aggarwal, C. C. and P. S. Yu (2010). On classification of high-cardinality data streams. In *SDM*, pp. 802–813. SIAM.

Al-Kateb, M., B. S. Lee, and X. S. Wang (2007). Adaptive-size reservoir sampling over data streams. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, SSDBM '07, Washington, DC, USA, pp. 22–. IEEE Computer Society.

Alippi, C., G. Boracchi, and M. Roveri (2013). Just-in-time classifiers for recurrent concepts. *IEEE Trans. Neural Netw. Learning Syst. 24*(4), 620–634.

Araujo, F., K. W. Hamlen, S. Biedermann, and S. Katzenbeisser (2014). From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, New York, NY, USA, pp. 942–953. ACM.

Asuncion, A. and D. Newman (2007). Uci machine learning repository.

Baron, M. (1999). Convergence rates of change-point estimators and tail probabilities of the first-passage-time process. *Canadian J. of Statistics 27*, 183–197.

Bergamo, A. and L. Torresani (2010). Exploiting weakly-labeled web images to improve object classification: a domain adaptation approach. In *Advances in Neural Information Processing Systems*, pp. 181–189.

Bickel, P. J. and D. A. Freedman (1981). Some Asymptotic Theory for the Bootstrap. *The Annals of Statistics 9*(6), 1196–1217.

Bifet, A. and R. Gavaldà (2007). Learning from time-changing data with adaptive windowing. In *SDM*. SIAM.

Bifet, A., G. Holmes, B. Pfahringer, P. Kranen, H. Kremer, T. Jansen, and T. Seidl (2010). Moa: Massive online analysis, a framework for stream classification and clustering. In *Journal of Machine Learning Research*, pp. 44–50.

Brzezinski, D. and J. Stefanowski (2014, May). Combining block-based and online methods in learning ensembles from concept drifting data streams. *Inf. Sci. 265*, 50–67.

Chandra, S., A. Haque, L. Khan, and C. Aggarwal (2016). An adaptive framework for multistream classification. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, pp. 1181–1190. ACM.

Chang, C.-C. and C.-J. Lin (2011). Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST) 2*(3), 27.

Chen, B., W. Lam, I. Tsang, and T.-L. Wong (2009). Extracting discriminative concepts for domain adaptation in text mining. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 179–188.

Cieslak, D. and N. Chawla (2007, Oct). Detecting fractures in classifier performance. In *ICDM 2007*, pp. 123–132.

Dahl, J. and L. Vandenberghe (2008). CVXOPT: A python package for convex optimization.

Dean, J. and S. Ghemawat (2008, January). Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*(1), 107–113.

Dyer, K. B., R. Capo, and R. Polikar (2014). Compose: A semisupervised learning framework for initially labeled nonstationary streaming data. *IEEE Transactions on Neural Networks and Learning Systems 25*(1), 12 – 26.

Efraimidis, P. S. and P. G. Spirakis (2006, March). Weighted random sampling with a reservoir. *Inf. Process. Lett. 97*(5), 181–185.

Efron, B. (1992). *Bootstrap methods: another look at the jackknife*. Springer.

Ester, M., H.-P. Kriegel, J. Sander, and X. Xu (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of 2nd International Conference on Knowledge Discovery and*, pp. 226–231.

Fan, R.-E., K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin (2008). Liblinear: A library for large linear classification. *The Journal of Machine Learning Research 9*, 1871–1874.

Fan, W., Y. an Huang, H. Wang, and P. S. Yu (2004). Active mining of data streams. In *in Proceedings of the Fourth SIAM International Conference on Data Mining*, pp. 457–461.

Faria, E., I. Goncalves, J. Gama, and A. Carvalho (2013, Oct). Evaluation methodology for multiclass novelty detection algorithms. In *Intelligent Systems (BRACIS), 2013 Brazilian Conference on*, pp. 19–25.

Freund, Y. and R. E. Schapire (1995). A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, EuroCOLT '95, London, UK, UK, pp. 23–37. Springer-Verlag.

Gama, J., P. Medas, G. Castillo, and P. Rodrigues (2004). Learning with drift detection. In *Advances in artificial intelligence–SBIA 2004*, pp. 286–295. Springer.

Gama, J., I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia (2014). A survey on concept drift adaptation. *ACM Computing Surveys (CSUR) 46*(4), 44.

Gretton, A., A. Smola, J. Huang, M. Schmittfull, K. Borgwardt, and B. Schölkopf (2009). Covariate shift by kernel mean matching. *Dataset shift in machine learning 3*(4), 5.

Hall, M., E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten (2009, November). The weka data mining software: An update. *SIGKDD Explor. Newsl. 11*(1), 10–18.

Haque, A., L. Khan, and M. Baron (2016, Feb). Sand: Semi-supervised adaptive novel class detection and classification over data stream. In *Thirteenth AAAI Conference on Artificial Intelligence*, pp. 1652–1658.

Haque, A., L. Khan, M. Baron, B. Thuraisingham, and C. Aggarwal (2016, May). Efficient handling of concept drift and concept evolution over stream data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 481–492.

Haque, A., B. Parker, and L. Khan (2013). Labeling instances in evolving data streams with mapreduce. In *2013 IEEE International Congress on Big Data (BigData Congress)*, pp. 387–394.

Haque, A., B. Parker, L. Khan, and B. Thuraisingham (2014, June). Evolving big data stream classification with mapreduce. In *2014 IEEE 7th International Conference on Cloud Computing*, pp. 570–577.

Haque, A., Z. Wang, S. Chandra, B. Dong, L. Khan, and K. W. Hamlen (2017). Fusion: An online method for multistream classification. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, New York, NY, USA, pp. 919–928. ACM.

Haque, A., Z. Wang, S. Chandra, Y. Gao, L. Khan, and C. Aggarwal (2016, Dec). Sampling-based distributed kernel mean matching using spark. In *2016 IEEE International Conference on Big Data (Big Data)*, pp. 462–471.

Harel, M., S. Mannor, R. El-yaniv, and K. Crammer (2014). Concept drift detection through resampling. In *ICML-14*, pp. 1009–1017. JMLR Workshop and Conference Proceedings.

Hayat, M. Z. and M. R. Hashemi (2010). A dct based approach for detecting novelty and concept drift in data streams. In *SoCPaR*, pp. 373–378. IEEE.

Huang, J., A. Gretton, K. M. Borgwardt, B. Schölkopf, and A. J. Smola (2006). Correcting sample selection bias by unlabeled data. In *Advances in neural information processing systems*, pp. 601–608.

Jiang, J. and C. Zhai (2007). Instance weighting for domain adaptation in nlp. In *ACL*, Volume 7, pp. 264–271.

Kanamori, T., S. Hido, and M. Sugiyama (2009). A least-squares approach to direct importance estimation. *The Journal of Machine Learning Research 10*, 1391–1445.

Kawahara, Y. and M. Sugiyama (2012, April). Sequential change-point detection based on direct density-ratio estimation. *Stat. Anal. Data Min. 5*(2), 114–127.

Kivinen, J., A. J. Smola, and R. C. Williamson (2004, August). Online Learning with Kernels. *IEEE Transactions on Signal Processing 52*, 2165–2176.

Klinkenberg, R. (2004, August). Learning drifting concepts: Example selection vs. example weighting. *Intell. Data Anal. 8*(3), 281–300.

Kouloumpis, E., T. Wilson, and J. D. Moore (2011). Twitter sentiment analysis: The good the bad and the omg! *Icwsm 11*, 538–541.

Koychev, I. (2000). Gradual forgetting for adaptation to concept drift. In *In Proceedings of ECAI 2000 Workshop Current Issues in Spatio-Temporal Reasoning*, pp. 101–106.

Koychev, I. (2002). Tracking changing user interests through prior-learning of context. In *Adaptive Hypermedia and Adaptive Web-Based Systems*, Volume 2347 of *Lecture Notes in Computer Science*, pp. 223–232. Springer Berlin Heidelberg.

Kuncheva, L. I. and W. J. Faithfull (2012, Nov). Pca feature extraction for change detection in multidimensional unlabelled streaming data. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pp. 1140–1143.

Lichman, M. (2013). UCI machine learning repository.

Masud, M. M., J. Gao, L. Khan, J. Han, and B. Thuraisingham (2010). Classification and novel class detection in data streams with active mining. In *Proceedings of the 14th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining - Volume Part II*, PAKDD'10, Berlin, Heidelberg, pp. 311–324. Springer-Verlag.

Masud, M. M., J. Gao, L. Khan, J. Han, and B. M. Thuraisingham (2008). A practical approach to classify evolving data streams: Training with limited amount of labeled data. In *ICDM*, pp. 929–934.

Masud, M. M., J. Gao, L. Khan, J. Han, and B. M. Thuraisingham (2011). Classification and novel class detection in concept-drifting data streams under time constraints. *IEEE Trans. Knowl. Data Eng. 23*(6), 859–874.

Miao, Y.-Q., A. K. Farahat, and M. S. Kamel (2015). Ensemble kernel mean matching. In *Data Mining (ICDM), 2015 IEEE International Conference on*, pp. 330–338. IEEE.

MOA (2015). Moa massive online analysis-real time analytics for data streams repository data sets. `http://moa.cms.waikato.ac.nz/datasets/`.

Nishida, K., K. Yamauchi, and T. Omori (2005). Ace: Adaptive classifiers-ensemble system for concept-drifting environments. In *Multiple Classifier Systems*, Volume 3541 of *Lecture Notes in Computer Science*, pp. 176–185. Springer.

Pan, S. J. and Q. Yang (2010). A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on 22*(10), 1345–1359.

Parker, B., A. M. Mustafa, and L. Khan (2012). Novel class detection and feature via a tiered ensemble approach for stream mining. In *ICTAI*, pp. 1171–1178.

Parker, B. S. and L. Khan (2015). Detecting and tracking concept class drift and emergence in non-stationary fast data streams. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pp. 2908–2913.

Platt, J. C. (1999). Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pp. 61–74. MIT Press.

Ramakrishnan, N., P. Butler, S. Muthiah, N. Self, R. Khandpur, P. Saraf, W. Wang, J. Cadena, A. Vullikanti, G. Korkmaz, C. Kuhlman, A. Marathe, L. Zhao, T. Hua, F. Chen, C. T. Lu, B. Huang, A. Srinivasan, K. Trinh, L. Getoor, G. Katz, A. Doyle, C. Ackermann, I. Zavorin, J. Ford, K. Summers, Y. Fayed, J. Arredondo, D. Gupta, and D. Mares (2014). 'beating the news' with embers: Forecasting civil unrest using open source indicators. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, New York, NY, USA, pp. 1799–1808. ACM.

Reiss, A. and D. Stricker (2012). Introducing a new benchmarked dataset for activity monitoring. In *ISWC*, pp. 108–109. IEEE.

Ross, G. J., D. K. Tasoulis, and N. M. Adams (2011). Nonparametric monitoring of data streams for changes in location and scale. *Technometrics 53*(4), 379–389.

Settles, B. (2009). Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison.

Song, X., M. Wu, C. Jermaine, and S. Ranka (2007). Statistical change detection for multi-dimensional data. In *13th ACM SIGKDD*, NY, USA, pp. 667–676. ACM.

Spinosa, E. J., A. e. P. de Leon F. de Carvalho, and J. ao Gama (2008). Cluster-based novel concept detection in data streams applied to intrusion detection in computer networks. In *ACM SAC*, pp. 976–980.

Sugiyama, M., S. Nakajima, H. Kashima, P. V. Buenau, and M. Kawanabe (2008). Direct importance estimation with model selection and its application to covariate shift adaptation. In *Advances in neural information processing systems*, pp. 1433–1440.

Tumer, K. and J. Ghosh (1996). Error correlation and error reduction in ensemble classifiers. *Connection Science 8*(3-4), 385–403.

Vasilomanolakis, E., S. Karuppayah, M. Mühlhäuser, and M. Fischer (2015, May). Taxonomy and survey of collaborative intrusion detection. *ACM Comput. Surv. 47*(4), 55:1–55:33.

Wang, L., J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu (2014, Feb). Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–499.

Widmer, G. and M. Kubat (1996). Learning in the presence of concept drift and hidden contexts. *Machine Learning 23*(1), 69–101.

Yin, S. and O. Kaynak (2015, Feb). Big data for modern industry: Challenges and trends [point of view]. *Proceedings of the IEEE 103*(2), 143–146.

Yu, Y.-l. and C. Szepesvári (2012). Analysis of kernel mean matching under covariate shift. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, pp. 607–614.

Zadrozny, B. Z. (2004). Learning and evaluating classifiers under sample selection bias. In *In International Conference on Machine Learning (ICML)*, pp. 903–910.

Zaharia, M., M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pp. 10.

Zhu, X. (2010). Stream data mining repository. `http://www.cse.fau.edu/~xqzhu/stream.html`.

Zhu, X., P. Zhang, X. Lin, and Y. Shi (2007, Oct). Active learning from data streams. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pp. 757–762.

Zhu, X., P. Zhang, X. Lin, and Y. Shi (2010, Dec). Active learning from stream data using optimal weight classifier ensemble. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 40*(6), 1607–1621.

Zikopoulos, I., C. Eaton, and P. Zikopoulos (2011). *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. Mcgraw-hill.

BIOGRAPHICAL SKETCH

Ahsanul Haque received his B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET) in February 2011. In the quest for higher studies, Ahsanul joined The University of Texas at Dallas (UT Dallas) in 2012 as a graduate student. He received his M.S. in computer science from UT Dallas in May 2016. The Computer Science department recognized his extraordinary academic success by awarding him the *Certificate of Academic Excellence*. Research focus of Ahsanul is on real-time stream data analysis and semi-supervised learning including active learning and transfer learning. His research works have been published in premier computer science conferences, such as AAAI, CIKM, ICDE, ICDM, IEEE Big Data, etc. During his doctoral studies at UT Dallas, Ahsanul received prestigious merit-based *Lars Magnus Ericsson* and *ECS Louis Beecherl Jr.* Graduate Fellowships. Beyond academics, Ahsanul enjoys listening to music, reading autobiographies, and traveling.

CURRICULUM VITAE

Ahsanul Haque

Email: ahsanul.haque@utdallas.edu

Educational History:

Ph.D., Computer Science, The University of Texas at Dallas (UT Dallas), 2017
M.S., Computer Science, The University of Texas at Dallas (UT Dallas), 2016
B.Sc., Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), 2011

Semi-supervised Adaptive Classification over Data Streams
Ph.D. Dissertation
Department of Computer Science, UT Dallas
Advisor: Dr. Latifur Khan

Bandwidth Allocation and Scheduling in WiMAX Technology
Undergraduate Thesis
Department of Computer Science and Engineering, BUET
Advisor: Dr. Md. Humayun Kabir

Professional Experience:

Graduate Research/Teaching Assistant, UT Dallas, September 2012 – present
Research Science Intern, eBay, May 2017 – August 2017
Data Science Intern, JPMorgan Chase & Co., June 2015 – August 2015
Software Engineer, Samsung Electronics, March 2011 – July 2012

Honors and Awards:

Lars Magnus Ericsson Graduate Fellowship, Office of the Dean, Erik Jonsson School of Engineering and Computer Science, UT Dallas, 2016.
Certificate of Academic Excellence, Department of Computer Science, UT Dallas, 2016
ECS Louis Beecherl, Jr. Graduate Fellowship, Office of the Dean, Erik Jonsson School of Engineering and Computer Science, UT Dallas, 2015
Degree with Honors, BUET, 2011
Deans List award, Deans' Office, Faculty of Electrical Engineering, BUET, 2006-2011

Selected Publications:

– Haque, A.; Wang, Z.; Chandra, S., Dong. B., Khan, L., *FUSION - An Online Method for Multistream Classification*, in proceedings of the 26^{th} ACM International Conference on Information and Knowledge Management (CIKM), Singapore, Nov 2017, pp. 919-928.

– Haque, A.; Chandra, S.; Khan, L.; Aggarwal, C., *Efficient Multistream Classification using Direct Density Ratio Estimation*, in proceedings of the 33^{rd} IEEE International Conference on Data Engineering (ICDE), San Diego, CA, 2017, pp. 155-158.
– Haque, A.; Khan, L.; Baron, M., *SAND: Semi-Supervised Adaptive Novel Class Detection and Classification over Data Stream*, in proceedings of 30^{th} AAAI Conference on Artificial Intelligence, Phoenix, AZ, 2016, pp. 1652-1658.
– Haque, A.; Khan, L.; Baron, M.; Thuraisingham, B.; Aggarwal, C., *Efficient Handling of Concept Drift and Concept Evolution over Stream Data*, in proceedings of 32^{nd} IEEE International Conference on Data Engineering (ICDE), Helsinki, Finland, 2016, pp. 481-492.
– Chandra, S.; Haque, A.; Khan, L.; Aggarwal, C., *Efficient Sampling-based Kernel Mean Matching*, in proceedings of the IEEE International Conference on Data Mining (ICDM), Barcelona, Spain, 2016, pp. 811-816.
– Chandra, S.; Haque, A.; Khan, L.; Aggarwal, C., *An Adaptive Framework for Multistream Classification*, in proceedings of the 25^{th} ACM International Conference on Information and Knowledge Management (CIKM), Indianapolis, IN, 2016, pp. 1181-1190.
– Haque, A.; Zhuoyi, W.; Chandra, S.; Gao, Y.; Khan, L.; Aggarwal, C., *Sampling based Distributed Kernel Mean Matching using Spark*, in proceedings of IEEE International Conference on Big Data (IEEE BigData), Washington, DC, 2016, pp. 462-471.
– Haque, A.; Khan, L.; Baron, M., *Semi Supervised Adaptive Framework for Classifying Evolving Data Stream*, in proceedings of the 19^{th} Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), Ho Chi Minh City, Viet Nam, 2015, pp. 383-394.
– Haque, A.; Parker, B.; Khan, L.; Thuraisingham, B., *Evolving Big Data Stream Classification with MapReduce*, in proceedings of the 7^{th} IEEE International Conference on Cloud Computing, Anchorage (IEEE Cloud), AK, 2014, pp. 570-577.
– Haque, A.; Chandra, S.; Khan, L.; Aggarwal, C., *Distributed Adaptive Importance Sampling on graphical models using MapReduce*, in proceedings of IEEE International Conference on Big Data (IEEE BigData), Washington, DC, 2014, pp. 597-602.
– Haque, A.; Parker, B.; Khan, L.; Thuraisingham, B., *Intelligent MapReduce Based Framework for Labeling Instances in Evolving Data Stream*, in proceedings of the 5^{th} IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom), Bristol, UK, 2013, pp. 299-304.
– Dong, B.; Li, Y.; Gao, Y.; Haque, A.; Khan, L., *Multistream Regression with Asynchronous Concept Drift Detection*, accepted for publication in proceedings of IEEE International Conference on Big Data (IEEE BigData), Boston, MA, 2017.
– Haque, A.; Tao, H.; Chandra, S.; Liu, J.; Khan, L., *A Framework for Multistream Regression with Direct Density Ratio Estimation*, accepted for publication in proceedings of 32^{nd} AAAI Conference on Artificial Intelligence, New Orleans, LA, 2018.