Optimizing the Computation of the Fourier Spectrum from Decision Trees

Johan Jonathan Nugraha

A thesis submitted to
Auckland University of Technology
in partial fulfillment of the requirement for the degree of
Master of Computer and Information Sciences (MCIS)

April, 2015

School of Computing & Mathematical Sciences

Supervisor: Dr. Russel Pears

Contents

List of Figures

List of Tables

Attestation of Authorship

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Auckland, April 2015

[illegible]

Johan J. Nugraha

Acknowledgements

This thesis would not exist without the tremendous support from my main (and only) supervisor, Russel Pears. Russel is an excellent leader and has always welcomed me to his office, even during his busiest times. Ever keen and sincere to share his deep knowledge, he asked probing questions and patiently guides me to materialize my thoughts and brings me closer to the solution. Being my only supervisor, Russel had to endure some of my less satisfactory attempts at academic writings. His constant encouragements also inspired me to always strive for greater quality. I am especially grateful as my thesis is so much better thanks to his supports.

It is of great pleasure to work along Sripirakas Sakthithasan, who has helped me greatly by recommending reading articles, providing datasets and preprocesses. Without his helps, it would be very hard for me to perform experiments and analyse the results.

Last but not least, my final thanks are reserved to my family and friends who supported me outside of the university life. Although they may not completely understood my works, they faithfully encouraged me just as well. Especially my parents, for raising me and supporting me in ways that can never be repaid.

Ultimately, I would like to thank my fiancée, Ariella Listyani, for being the greatest support which enable me to go through this journey which is difficult to tread at times.

A.M.D.G

Abstract

In the field of mining data streams, processing time is one of the most important factors because data instances continuously arrive at high-speed. Thus, it is crucial to process each instance in a timely manner. This concern is further emphasized when Discrete Fourier Transform (DFT) is applied to Decision Tree systems. DFT is highly beneficial in data stream mining as it allows us to represent a Decision Tree with less memory usage while preserving the accuracy of the prediction model. However, DFT is notorious for being computationally expensive to perform. An existing solution (Kargupta & Park, 2004) to mitigate this problem is by computing coefficients for smaller order. This approach, however, is highly inefficient for deep trees as the underlying problem space grows exponentially.

We seek to improve the efficiency of computing Fourier coefficients of Decision Trees in this study. We propose a divide and conquer solution by decomposing a Decision Tree into small sub-trees. Local Fourier coefficient table are then built for each sub-tree. The final Fourier coefficient table is then compiled by tabulating all the local Fourier coefficient tables. This effectively allows us to derive a coefficient table from an existing coefficient table. The method is particularly useful for two purposes. First is to track and synchronize a Decision tree and its Fourier spectrum on the fly. Second is to handle recurring concepts by reusing previously learned concepts. This thesis serves as a groundwork for future works which will be focused for the latter purpose.

In our experiments, we compare the runtime of our solution to that of the normal DFT process with various order cutoff on both real and artificial datasets. The results show that our solution generally performed better than normal DFT process with various degrees of success from moderate to significant improvements. We also measured the runtime performances of our solution in eager synchronization setup for comparison.

CHAPTER 1

Introduction

Every day, enormous amount of data are produced by systems running continuously. This continuous flow of data creates challenge to efficiently process huge amount of data, potentially infinite, in a timely manner. For example, Google handles more than 5 billion searches per day in 2012 [1]. Another example of continuous system is Yahoo finance keeping track of stock prices listed in every stock market in the world at near real-time, if not real-time.

The Knowledge Discovery in Databases (KDD), recently known as Knowledge Discovery and Data mining, is a multi-discipline field mainly concerned with making sense out of data (Fayyad, Piatetsky-Shapiro, & Smyth, 1996). It involves a wide spectrum of processes; from pre-processing and post-processing of data to pattern recognition and statistical analysis and interpretation of mined results. The extracted knowledge (or pattern) is potentially useful to assist human to make decisions, often to gain economic edge from business competitors.

However, traditional KDD techniques were only suitable for static datasets, typically stored in a data warehouse. When applied to data with continuous flow, KDD has to make immediate decision e.g. raising alarm and perform other security measures when detecting early onset of a possible denial-of-service attack. In static mining, data can be pre-analysed which helps to create the best possible learning model for future predictions. Patterns can then be relearned once it no longer relevant to new batches of data. In stream

[1] http://www.statisticbrain.com/google-searches/

mining, models expired far too fast that discard-rebuild approach is no longer possible.

Because of its nature, extracting patterns from streaming input poses serious challenges. First, streams are open-ended, making the amount of data to process enormously large (potentially an infinite amount). It is simply infeasible to store all arriving data because memory space is limited. Second, data arrives at high-speed. Unlike batch learning setting, a learning algorithm in stream mining context only has one chance to read arriving data instances and thus needs to process such instances on the fly. Third, streams evolve over time. Hinted at above, statistical properties of input instances may change dynamically. It is necessary for a learner to quickly adapt to changes in the stream as obsolete data may reduce the effectiveness of the learner.

This motivated the development of learning techniques with the capability to build models incrementally as data arrives. Incremental learning is crucial as it allows accumulation of information continuously, thus allowing for processing of open-ended input such as streams. Examples of incremental learners are Hoeffding tree (Domingos & Hulten, 2000) and AOG (Gaber, Krishnaswamy, & Zaslavsky, 2005) which are able to dynamically make changes to the learning model as new data instances arrive.

Data stream mining is a maturing research field. Extensive works have been done to extend existing batch learning algorithms into incremental learners, such as SVM (Domeniconi & Gunopulos, 2001), Decision Tree (Domingos & Hulten, 2000; Hulten, Spencer, & Domingos, 2001) and the Bayesian Network (Hulten & Domingos, 2002). A number of works also focused on determining when a concept change occurs, such as the CUSUM (Basseville, Nikiforov, et al., 1993), DDM (Gama, Medas, Castillo, & Rodrigues, 2004) and ADWIN (Bifet & Gavalda, 2007). Such a concept change detection algorithm is typically paired with a base incremental learner to further boost the learners' adaptability to the evolving stream.

However, there are plenty real-world scenarios such as military-use sensor networks, aircraft navigation systems, autonomous mass transport systems, earthquake early warning systems and high-frequency trading systems where previously-known patterns, or concepts, are likely to reoccur. For example, a condition that drives an increase of a commodity price may reappear in the future. Another example is specific tremor patterns may reoccur prior to a massive quake. Such systems have to accurately make predictions and take immediate action in order to avoid disastrous, potentially irrecoverable, damages or losses.

Extracting knowledge from streams with reoccurring concepts is a relatively young field of research. Mining recurrent concepts is a very challenging task as we now have to face additional issues, while considering all the challenges posed by stream mining. The main issue is to figure out how to capture a concept in anticipation to future occurrences. A traditional incremental learner would be highly inefficient as it is not aware of concept recurrence and will treat concept reoccurrences as "just another concept change", resulting in time wasted to relearn previously-known concepts.

There are several works in the field of mining stream with concept recurrences, such as (Gama & Kosina, 2011; Katakis, Tsoumakas, & Vlahavas, 2008; Ramamurthy & Bhatnagar, 2007; Sripirakas & Pears, 2014). A promising strategy in mining recurrent concepts is to train an ensemble of learners and then identify which one best capture the current concept. The model is then stored for future prediction, in the hope that the learned concept will reappear in the stream.

Although simple and effective, this approach is very costly in terms of memory consumption as models representing past concepts are accumulated and thus may be prohibitive for production systems with restricted memory. The memory requirements can be reduced by employing a compression method to models that has learned concepts. Kargupta and Park (2004) showed that it is possible to transform a decision tree structure into a highly condensed form using the Discrete Fourier Transform (DFT). In the study they showed that the DFT of a tree obtained by computing relatively low order coefficients was able to capture virtually all of the classification accuracy. Sripirakas and Pears (2014) showed that the DFT can be particularly useful in mining streams with concept recurrences as it greatly reduces the memory required to store past concepts, while preserving the prediction performance of uncompressed trees.

Despite all the benefits offered by the DFT, it is inherently an expensive computation with time complexity of $O(n^2)$. Unfortunately, the well-known Fast Fourier Transform (FFT) is unsuitable in the case of DFT of decision trees due to the dimensionality issue; FFT only works in univariate transform while the DFT of decision trees is a multivariate problem. Existing solutions proposed (Park, 2001; Sripirakas, 2015) to mitigate the costly DFT computation are: 1) exploiting the properties of Fourier transform and 2) reducing the computational overhead of the DFT. However, the solutions do not attempt to reduce the underlying exponential problem space. This motivates us to find a way to reduce time complexity of computing the DFT of decision trees.

This research focuses on addressing the issue caused by the costly nature of the DFT in its role to support mining recurrent concepts. Inspired by the FFT, our solution uses a divide and conquer strategy to decompose a decision tree into sub-trees and compute Fourier spectra in the localized problem space of sub-trees. Later in this thesis, we will show that our solution in general improves the runtime needed to compute Fourier spectra on both synthetic and real-world datasets.

1.1 Research Questions

The Discrete Fourier Transform (DFT) by its very nature is an expensive computation, where the number of coefficients to compute grows exponentially on the number of distinct features in a decision tree. In practice on univariate data, it is usually implemented as the Fast Fourier Transform (FFT), which uses a divide and conquer strategy to break the performance barrier of the DFT.

Unfortunately, the FFT is not applicable to transform decision trees into Fourier spectra. The main obstacle is that the FFT is a univariate transform while Fourier spectrum of a decision tree is a multivariate problem. Being aware of this problem, Kargupta and Park (2004) proposed a solution exploiting the properties of decision trees in Fourier domain. Another optimization was proposed by Sripirakas which reduces the computational overhead of the DFT (Sripirakas, 2015). These approaches, however, do not attempt to reduce the size of the underlying feature space.

The main objective of this thesis is to explore ways to further improve the DFT computation of decision trees, which raises several questions:

1. Is it possible to reduce the size of problem space by decomposing a decision tree into sub-trees and compute their Fourier spectra separately? Will this approach incur information loss?

2. Assuming that Question 1 leads to a positive answer, is computation of Fourier spectra of sub-trees always less costly than computing the Fourier spectrum of the original tree? Under what conditions is the decomposition approach beneficial?

3. Because Fourier spectrum is highly dependent on the tree structure, is it possible to infer the Fourier spectrum of a decision tree, or a sub-tree from previously-computed Fourier spectra of similar trees?

1.2 Scope

This thesis is limited to the application of DFT on Hoeffding Trees in the stream mining setting with recurrent concepts. Hoeffding Tree is a top-down tree induction algorithm that is specifically designed to learn incrementally which is suited to a stream mining environment. The DFT can be particularly useful in the recurrent concept setting as it allows us to store decision trees in a highly compressed form while preserving the predictive performance of the underlying model (Sripirakas & Pears, 2014).

The application of DFT in this thesis is limited to the binary feature domain for illustration purposes but the DFT of decision trees can be generalized to arbitrary-size feature domain as shown by Park (Park, 2001). However, the implementation of our algorithm is designed for decision trees in binary feature domain to serve as a proof of concept of our solution. Extension of our solution into n-ary feature space will be discussed in the future work section of this thesis.

1.3 Overview of Research Strategy

We briefly present the overall research strategy in this section. We make use of an existing general framework to mine streams with recurrent concepts which is proposed in (Sripirakas, 2015). The framework consists of three main components: a decision forest containing tree classifiers that are learning concepts dynamically, a concept change detector, and a repository pool to collect learned concept classifiers. Each component is designed to tackle a specific problem in recurrent concept mining. The forest manages a collection of decision tree learners, each of which is trained to capture different concepts. This can be achieved in numerous ways, such as training each tree on different partition of feature space, or growing each tree on a distinct root attribute.

Concept drift detectors raise alerts to the system when they detect concept change in the stream. The pool serves as a repository of captured concepts which will be used for future predictions.

The flow of the general framework is as follows. Data instances are routed to the forest and are used to train each tree member separately. Each decision tree in the forest is paired with a concept detector. Whenever a concept change is detected, the tree with the highest accuracy in the forest is selected as the winner tree. The winner tree, having captured a concept, will then be stored in a repository for possible future use in a highly condensed form. In our implementation, the compression used is the DFT. At the beginning of the stream, output prediction will be mainly performed by the forest. When old concepts reoccur in the stream, the system will dynamically switch to the concept classifier with the highest accuracy in the repository. The framework will be described in further detail in Chapter 3.

Later in this thesis we will introduce our solution to efficiently compute the DFT of decision trees, dubbed memoFT. For the experiment runs, we implemented three types of DFT computations, normal DFT and memoFT in both lazy and eager mode. The implementation of the system is configured in such way because we want to compare the runtimes of the three processes. Since our algorithm uses memory-optimization technique, it is necessary to select which winner tree should be compressed using memoFT. In this research, we use a simple statistical count strategy to select on which winner tree memoFT should perform a DFT.

We then run experiments using synthetic and real-world datasets. The Rotating Hyperplane and Radial Basis Function generators are specifically chosen because they generates datasets with polar opposite characteristics. From the real world datasets, we chose the Forest Covertype, NSW Electricity and Flight datasets. The entire experimental setting will be fully explained in Chapter 5.

1.4 Thesis Structure

The rest of this thesis is organized as follows. In Chapter 2, we outline the data stream mining primer, as well as the progress of the research work done in the field, up to the current state. We will then focus on the mining framework serving as the foundation of this thesis in Chapter 3 and discuss the key elements used in the framework, giving special attention to DFT application to decision trees where we will identify the main issue we attempt to solve. In Chapter 4, we present our research design and propose an algorithm that uses a divide and conquer strategy to address the issue raised in Chapter 3. We then proceed to assess our algorithm in terms of runtime and memory usage in Chapter 5. The research achievements, limitations, and future works are discussed in Chapter 6.

Chapter 2

Literature Review

Throughout this chapter, we will describe the progress of research in data stream mining to recurrent concept mining. We start the chapter with defining data mining in a stream environment, describing the challenges and problems it attempt to solve. In the last section, we address mining of recurring concepts which is the basic premise of our research.

2.1 Data Stream Mining

Data mining is a field of study mainly concerned with extraction of meaningful patterns from data. The need to automate this process is ever increasing because we generate data much faster than the human capacity of understanding such data. Hidden in this formidable amount of data is information that is potentially useful, waiting to be taken advantage of.

Data stream mining is a specialized subfield of data mining; it attempts to apply data mining techniques to be suitable for mining open-ended data streams. For this purpose, there is a need to define what a data stream is. Data stream is defined as a flow, or stream, of data instances being transmitted at a high rate of speed.

Several key issues in mining data stream are restrictions on resources: sample size, memory space and time. Classical data mining draws its training instances from static datasets, where learning algorithms are allowed to read instances multiple times, effectively giving the chance for the algorithms to further fine-tune their generated models. Such a learning style, however, is technically infeasible with open-ended input such as data streams.

Enlisted below are the additional constraints that apply in data stream mining:

1. The volume of data is enormously large and potentially grows infinitely. It is impossible to store all arriving data. Instead, only a small amount of instances can be processed and stored at a given time.

2. The learning algorithm is not allowed to read input data in multiple passes and has to make decision or computation, and then discard obsolete instances in a timely manner. This is because the rate of arrival of input is large.

3. Data evolves over time. Data may evolve because of the changes in statistical properties of distribution in the input. It means that mining algorithm has to adapt to evolving streaming data because outdated data may reduce accuracy of the prediction model.

In this section, we will describe research work done addressing Constraint 1 and 2, while work focused to tackle Constraint 3 is described in Section 2.2.

2.1.1 Classification in Data Stream

Batch learning algorithms such as CART (Breiman, Friedman, Stone, & Olshen, 1984), ID3 (Quinlan, 1986), C4.5 (Quinlan, 1993), k-Nearest Neighbor and Neural Networks are not designed to mine data streams where the volume of data is potentially unlimited. This motivates research into development of incremental learning algorithms. Building models incrementally is an important capability as it supports a continuous learning process and accumulation of information over time. An incremental learner effectively builds a model once and continuously updates the model as new instances arrive, preserving previously acquired knowledge (Ade & Deshmukh, 2013). This incremental update of model is favored because discarding and rebuilding models can be very costly to perform in terms of processing time.

Domingos and Hulten (2000) developed the Hoeffding tree to perform incremental induction of decision trees by making use of Hoeffding bound. The usage of the Hoeffding bound eliminates the need for the tree learner to read instances multiple times that was previously required by traditional top-down tree induction algorithms. In the study, Hoeffding tree was implemented as VFDT (Very Fast Decision Tree) which was shown to run much faster and be more tolerant to noise compared to C4.5 (Quinlan, 1993).

Domingos and Hulten then extended the idea in a later study (2002), applying the Hoeffding bound to other classification learners. In the study, Domingos and Hulten used Bayesian Network as the base learner. They showed that Bayesian Network fitted with Hoeffding bound to be superior compared to Dirichlet distribution-based Bayesian learner developed by Heckerman et al. (1995)

Gaber et al. (2005) developed AOG (Algorithm Output Granularity) as an alternative approach to incremental learning that is resource-aware. AOG is designed to handle fluctuating data rates commonly occurring with systems involving networks of sensors. AOG consists of three stages: Mining, adaptation and knowledge merging. The first two stages are dynamically adjusted to cope with variable input rate. Knowledge merging is performed when allocated memory space is full. In the study, it is shown that AOG is also suitable for clustering and frequent pattern tasks.

Domeniconi and Gunopulos (2001) introduced an approach to perform incremental learning using the Support Vector Machine (SVM) as the base learner. In the study, the base learner process instances in batches. Domeniconi and Gunopulos identified four possible techniques to train SVM incrementally. 1) Error Driven, where wrongly classified instances are preserved for fine tuning the model. 2) Fixed Partition, where models are built separately on fixed-size batches and aggregated. 3) Exceeding Margin where instances located in the margin of a SVM model are preserved to update the model. 4) Combining Error Driven and Exceeding Margin.

2.2 Mining Evolving Data Streams

As discussed in Section 2.1, data streams are open-ended and as such, data may change overtime. As a consequence, a pattern, or *concept*, previously learned at one point of time becomes obsolete as new instances represents different concepts. The change of concept, popularly known as *concept drift*, is caused by the shift of statistical information of the instances.

Several research works focused to tackle concept drift typically makes use of sliding window mechanism (Bifet & Gavalda, 2006, 2007; Hulten et al., 2001). A sliding window is particularly useful in stream mining for detecting concept changes occurring in the stream, and preserving instances for updating models (Hoeglinger & Pears, 2007).

Basseville and Nikiforov conducted one of the earliest surveys of change detection algorithms (1993), where they compiled change detection methods in four categories: control charts, filtered derivative algorithms, cumulative sum (CUSUM) tests, and Bayes-type algorithms. All four categories involved usage of sliding windows of which statistical information is checked against adaptive threshold to detect changes.

Domingos et al. showed that VFDT can be extended to handle concept drift with a drift detection mechanism (2001). The resulting algorithm is called CVFDT (Concept-adapting Very Fast Decision Tree). CVFDT grows a sub-tree when it detects that a split node is outdated. The split is then replaced by the new sub-tree when the sub-tree grows more accurate than the old one, resulting in a smooth transition of adapting to new concepts.

Gama et al. introduced Drift Detection Method (DDM) to detect occurrence of abrupt change (2004). The algorithm also maintains a sliding window while collects two sets of statistical information; that of all instances and information of instances from beginning to the point where classification errors start increasing. Additional feature introduced by DDM is two level *state* triggers, *Warning* and *Change*. As a warning is raised, instances are collected anticipating when change is triggered. When change is triggered, the learned model is discarded and a new model is built from the collected instances from the point at which the warning was raised.

Bifet and Gavalda introduced an adaptive sliding window approach called ADWIN (2007). ADWIN uses the Hoeffding bound to determine whether the statistical information of two window partitions are significantly different from each other. Therefore, ADWIN is appealing because it has theoretical guarantee of performance.

2.2.1 Time-based Windowing versus Concept-based Windowing

All of the works mentioned in the previous section falls under the time-based windowing scheme. Generally, there are two types of windowing schemes: time-based and concept-based windowing. Both schemes share the same goal; to enable learning algorithms to cope with changing concepts. In a typical time-based windowing scheme, a sliding window is partitioned into two and constantly monitored as shown in Figure 2.1. If the statistical information of both partitions is "significantly" different (or the difference is above certain threshold), it implies that concept drift has occurred and model needs to be updated (Hoeglinger, Pears, & Koh, 2009).

Concept-based windowing scheme is radically different from its time-based counterpart. The main motivation of this scheme is that older instances are

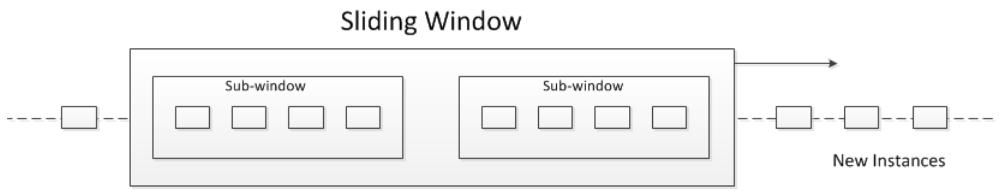


Figure 2.1: Typical sliding window structure for detecting concept change

not always outdated. Concept-based windowing scheme utilizes techniques to preserve some information of previously learned concepts that are still relevant when concept drift occurred.

Figure 2.2 illustrates the difference between time-based and concept-based windowing schemes in recognising concept drifts. Concept drift is illustrated at the top, while time-based and concept-based windowing scheme are located at the middle and bottom, respectively. The concept drift shown consisted of large drift which is superimposed with smaller drift. Time-based scheme would forget the smaller drift which has to be relearned, whereas concept-based scheme can memorize the smaller patterns.

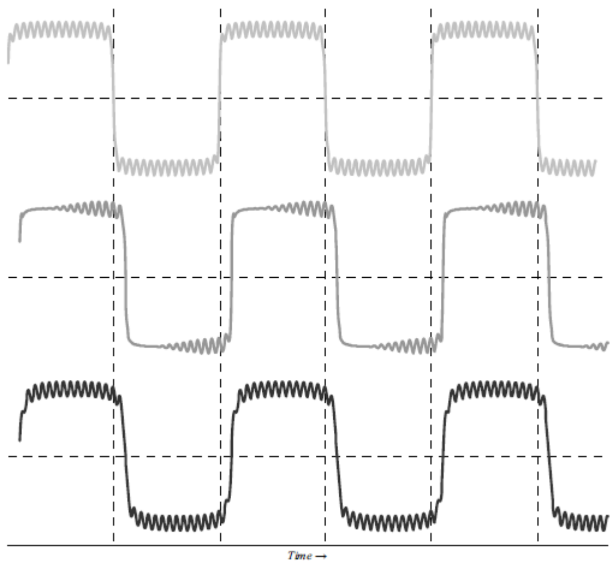


Figure 2.2: Concept drift recognition; time-based windowing vs. concept-based windowing

Wang et al. introduced a concept-based approach for mining data streams to tackle data expiration problem using reduction error analysis (2003). In this approach, a weighted ensemble is built on a set of data batches. Weight of each model in the ensemble is derived based on the expected classification error of each model.

Hoeglinger and Pears presented another concept-based approach using ensemble learning called CBDT (Concept-based Decision Tree) (2009). CBDT maintains an ensemble of decision trees grown separately on the same instances. As new instances arrive, trees with the lowest accuracy are discarded to provide additional memory space for those with better accuracy. Although internally it is a forest, class prediction is performed by individual tree with the highest accuracy.

2.3 Mining Streams with Recurring Concepts

A survey on concept-adaptation was conducted by Gama et al (2014). In the study, they identified two types of concept drift: *Real drift* and *virtual drift*. Real drift occurs when posterior probability of classes is changed, caused by changes in data distribution. Whereas, virtual drift occurs when changes in data distribution do not affect target concept. Figure 2.3 illustrates the difference between the two types of drift. In the figure, instances with different classes are represented as circles with different colors.

Gama et al. (2014) further noted that changes of concept can manifest

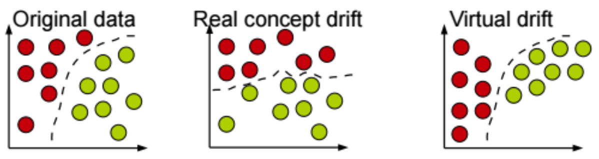


Figure 2.3: Types of concept drifts

in several forms. Drift may occur *abruptly*, where the new concept immediately replace the old concept, or *incrementally*, where there are intermediary concepts between the old and new concept. Drift may also occur *gradually*, where new concept progressively replaces the old concept. Drift can also occur multiple times where the old concept is *reoccurring.*

REDDLA was developed by Li, Wu and Hu to handle streams with recurring concepts and unlabelled instances (2012). The algorithm makes use of a clustering strategy on unlabelled instances, of which information will be used to make refinements to a decision tree model. In the study, concept drifts are detected by monitoring the distance of concept clusters which determines whether an old concept reoccurs, or a real drift is occurring. Several known issues with the approach are: how to accurately predict appearance of novel concepts, and how to correctly set the interval of reoccurring concepts and the high memory usage issue.

An ensemble approach to mine recurring patterns was proposed by Ramamurthy and Bhatnagar (2007). In their approach, all generated models are stored in a *global set*, and each model is trained on a batch of instances. Not all models in the global set are used to classify new instances. The system classifies using only models of which output error is within user-defined threshold. A new model is built whenever the ensemble accuracy is lower than user-defined acceptance threshold τ. The key issues with their approach is the need to fine-tune the acceptance threshold and specify the appropriate size of batches for different datasets.

Katakis et al. also used ensemble learning to tackle recurring concepts called CCP (Conceptual Clustering and Prediction) framework (2008). CCP process instances in batches, where instances are transformed into *conceptual vectors* using a mapping function. These conceptual vectors are then grouped using a clustering technique, forming a set of clusters, from which classification learners are trained separately on each cluster. In the study, they experimented using the Usenet data and showed that the ensemble method proposed has better accuracy than an incremental Naïve Bayes classifier. The approach necessitates the additional process to map input data into conceptual vectors which can be expensive in streams with large dimensionality. Additionally, the mapping function may need to be customized for use in different type of streams.

A two-layer approach is proposed by Gama and Kosina to handle recurring concepts and delayed labelled instances (2011). In the first layer, a base learner is trained using labelled instances in the stream. The model is paired with the second layer learner, called *referee*, which is a meta-classifier grown in parallel with the base model. When concept change is detected, the pair is stored for possible future use. Referees are trained on the region of feature space of their paired base classifiers, hence providing some confidence level of the output of the corresponding base learners. The referees make the decision to keep a classifier with the highest confidence score, provided that the predicted applicability of the classifier is above a user-defined threshold.

One of the most recent researches done in the field is by Sripirakas and Pears (2014) using the Discrete Fourier Transform (DFT) as a compression method for decision trees. The system also used ensemble learner (to be more precise a decision forest) where it maintains a set of trees trained separately on the same instances. Trees with highest accuracy are encoded into condensed representations using the DFT and stored in a *pool* for possible future use. Therefore, the pool effectively contains Fourier spectra of concepts. Class prediction is performed by the best performing classifier in the pool. When a novel concept appears in the stream, the system reverts back to the decision forest classification.

Summary

In this chapter, we have discussed the progression of research from data stream mining to recurrent concept mining, which is the main setting of our research. Most of the works done in recurrent concept mining involves the usage of ensemble learning with a memory structure to preserve previously learned models. However, ensemble learner uses up significant memory space. In the next section, we will describe our mining framework to capture recurring concepts which is able to preserve old concepts in compressed form to address the issue.

CHAPTER 3

Research Background

This chapter describes in further detail background information on important elements used in our research. We start with describing the general framework to mine recurrent concepts. We then describe chosen algorithms which serve as building blocks for experiments to be conducted in Chapter 5. The last part of this chapter is dedicated to describing the challenges of converting decision trees into their Fourier spectra representation, of which a solution will be presented in Chapter 4.

3.1 A General Framework for Data Stream Mining with Recurrent Concepts

Briefly covered in Chapter 2, there are two general approaches to handling recurring concepts in stream mining: The first approach is to map instances to conceptual vectors, which will then be used to train classifiers (Katakis et al., 2008). The second approach is to store classifiers having learned patterns for possible future use (Gama & Kosina, 2011; Li et al., 2012; Ramamurthy & Bhatnagar, 2007; Sripirakas & Pears, 2014). The latter approach is preferred by a number of researchers because of its simplicity and the absent of additional computations required to generate conceptual vectors. Furthermore, it is difficult to produce accurate conceptual vector representation of instances.

Our study also falls into the latter category, where classifiers are archived in anticipation of appearance of previously-known concepts. Sripirakas in-

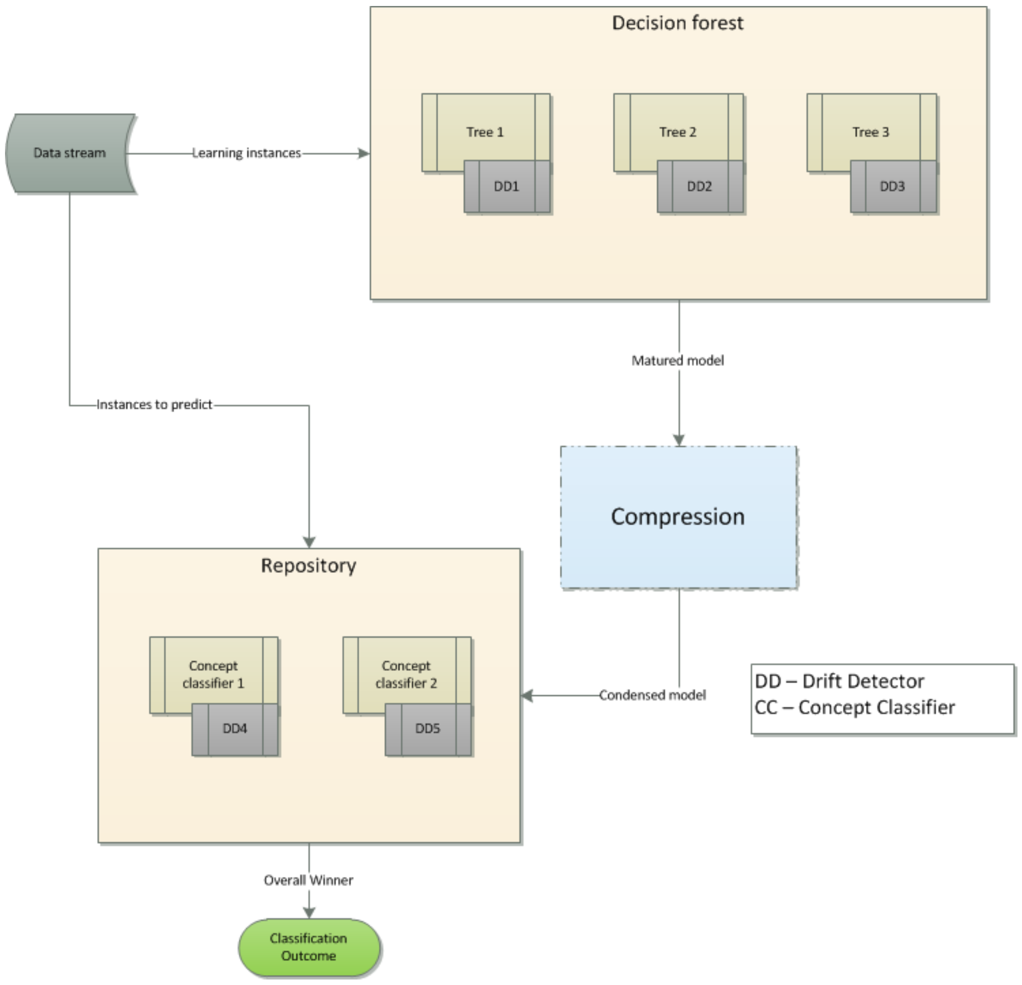


Figure 3.1: General framework to capture recurring concepts in streams, adapted from Sripirakas (2015)

troduced a general framework specialised in capturing recurrent concepts in data streams that requires several components, each of which tackles a specific problem (Sripirakas, 2015). Figure 3.1 shows a general framework for capturing recurrent concepts in streams.

The framework consists of three main components: 1) A decision forest containing classifiers which dynamically learn concepts, 2) a concept change detector and 3) a pool collecting learned concept classifiers.

The decision forest maintains a set of incremental tree learners, each is trained independently from arriving instances. There are several strategies to avoid creating identical trees, such as growing trees in distinct partitions of feature space, or growing trees on different features as the root attribute. We use the latter strategy in our implementation. We use Hoeffding tree as the base incremental tree learner and CBDT algorithm as the chosen forest system, which will be explained in some detail in Section 3.2 and Section 3.4 respectively.

Each tree learner in the forest is paired with a concept drift detector. When concept change is detected, the tree with the best accuracy in the forest is transformed into a condensed representation (Fourier spectrum) and stored in a *repository pool* for future predictions. Therefore, the repository pool contains Fourier spectra representing past concepts. In our implementation, ADWIN is chosen as the concept detection mechanism which will be further described in Section 3.3.

Stream instances are routed to both forest learner and repository pool. However, prediction output is not averaged, but taken from the classifier with the highest output accuracy from either the forest or the pool. Initially, class predictions would mainly be performed by the forest. However, once previously-captured concept reappears in the stream, the system can readily use a classifier from the repository pool.

In the following sections, we will describe the algorithms chosen as the building blocks of our study. Special attention is given to the DFT as the goal of this thesis is to improve the computational runtime complexity of transforming decision trees into Fourier spectra.

3.2 Hoeffding Tree

Hoeffding tree was developed by Domingos and Hulten (2000) as an incremental decision tree induction algorithm specifically designed to handle massive streams of data. As this algorithm is used as the base tree learner in our study, we are going to describe Hoeffding Tree in some detail.

The key idea behind Hoeffding tree is the use of the *Hoeffding bound.* Domingos and Hulten further expands this idea and shows that it can be generalized to other classification learner algorithms (2002). In the study they asserted that Hoeffding bounds allow any learner built on discrete search to process data streams.

In batch learning setting, a split attribute is determined by picking the attribute with highest information gain for ID3 (Quinlan, 1986) and C4.5 (Quinlan, 1993), or Gini index for CART (Breiman et al., 1984). In order to achieve a similar decision mechanism in stream mining without the need for revisiting past instances, Domingos and Hulten employed the Hoeffding bound, which is also known as the additive Chernoff bound.

The Hoeffding bound postulates that with probability $(1 - \delta)$, the true mean of a random variable of range R will not differ from the estimated mean after n independent observations by more than

$$\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{3.1}$$

Here is an example: Suppose there are two attributes, A and B with their respective information gains, $G(A)$ and $G(B), G(A) > G(B), \varepsilon = 0.1$. This would mean that the difference between the two information gains has to exceed 0.1 in order to confidently split on attribute A.

A very attractive feature of Hoeffding tree is its theoretical guarantee of performance. Using the concept of intentional disagreement it is shown that the output of Hoeffding tree is nearly identical to that of tree generated by non-incremental learner for infinitely many examples (Bifet, 2010). The intentional disagreement Δ_i between two decision trees is the probability that a path of an example through the first tree differs from the path going through the other tree. The theoretical guarantee is stated below.

Theorem: *If $HT\delta$ is the tree produced by Hoeffding tree algorithm with desired probability δ given infinite examples, DT is the batch learned decision tree having similar output performance, and p is the leaf probability, then $E[\Delta_i(HT\delta, DT)] \leq \delta/p$.*

3.2.1 VFDT

In the study (Domingos & Hulten, 2000), Hoeffding tree is the theoretical algorithm while the practical implementation is called VFDT, an acronym for **V**ery **F**ast **D**ecision **T**ree. In this thesis, we refer to Hoeffding tree as an umbrella term for any variant of the basic principles of the algorithm, including VFDT. The VFDT algorithm is as shown in Algorithm 3.1. VFDT employed additional heuristics in addition to the Hoeffding bound such as

- Sufficient statistics in each leaf nodes to compute information gain.
- Grace period. ε is computed every n_{min} instances arriving.
- Tie breaking τ is used when the difference between the two information gains are very small.

The Hoeffding tree is chosen as a base classifier for several reasons. First, decision tree models are fairly easy to comprehend. The ease of model interpretation greatly helps data analysts to gain better understanding of a problem (Witten & Frank, 2005). Second, the Hoeffding tree has a sound guarantee that it will be "very close" to the tree induced by batch learning algorithms (Kirkby, 2007; Bifet, 2010). This would mean that Hoeffding tree learners are capable of constructing trees of equivalent quality to those of batch learners without the need to revisit past instances. This is significant because decision trees are one of the best performing learning models in static data mining.

Algorithm 3.1 The VFDT algorithm

```
1: Let $HT$ be a tree containing a single node (the root)
2: Let $l$ be the root node of $HT$
3: Let $\mathbf{X}$ be the set of attributes of the stream
4: for all training instances do
5:     Sort instance into leaf $l$ using $HT$
6:     Update sufficient statistics in $l$
7:     Increment $n_l$, the number of examples seen at $l$
8:     if $n_l \mod n_{min} = 0$ and examples seen at $l$ not all of same class
       then
9:         Compute $\overline{G_l}$ for each attribute
10:        Let $X_a$ be attribute with highest $\overline{G_l}$
11:        Let $X_b$ be attribute with second-highest $\overline{G_l}$
12:        Compute Hoeffding bound $\varepsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}}$
13:        if $X_a \neq X_\emptyset$ and $(\overline{G_l}(X_a) - \overline{G_l}(X_b)) > \varepsilon$ or $\varepsilon < \tau$ then
14:            Replace $l$ with an internal node that splits on $X_a$
15:            for all branches of the split do
16:                Add a new leaf with initialized sufficient statistics
17:            end for
18:        end if
19:    end if
20: end for
```

CART (Breiman et al., 1984) and C4.5 (Quinlan, 1993) are widely recognized and have become *de facto* benchmark standards for research studies.

3.3 Concept Drift Mechanisms

One of the core concerns in mining data streams is dealing with concept drift. Concept drift occurs when there are unforeseen changes in the statistical information of the target of prediction. There are at least three tasks necessary to tackle this issue (Bifet & Gavalda, 2006): detecting changes occurring in the stream, maintaining sufficient statistics up-to-date and updating learning model(s) to keep up with the change occurring.

One of the most common approaches is to make use of a *sliding window* and its variants, where a window storing the most recent examples is being maintained following a set of rules (Bifet, 2010). The contents of such a window are necessary for the three tasks mentioned above.

The simplest form of this approach is to keep a fixed size window where the oldest instance is discarded for every new instance arrival, effectively following a LIFO (last in, first out) policy. Fixed size windowing, however, is a rigid mechanism. It causes a dilemma for users in a trade-off; "big" window is suitable when the concept is stable, whereas a "small" window reflects the most recent distribution better (Bifet & Gavalda, 2007).

This motivated the development of the Adaptive sliding window algorithm, also known as ADWIN, by Bifet and Gavalda (2006) addressing the issues where fixed windowing fall short. It has the capability to dynamically change the size of the window depending on the current state of the stream, where window size decreased when change is occurring and increased during time of stability (Bifet & Gavalda, 2007).

The first version of ADWIN is called ADWIN0 which maintains a window W that is partitioned in two adjacent sub-windows W_0 and W_1, W_0 partition contains older examples and W_1 contains recent ones. ADWIN0 contantly monitors W_0 and W_1. When the difference in sample means between the two partitions is greater than a statistically computed threshold, we can say that the older window, W_0, contains a stale concept and ADWIN0 can safely discard examples contained within it. Similar to Hoeffding tree, ADWIN0 also uses the Hoeffding bound to compute the mean difference threshold, ε_{cut}, allowing ADWIN0 to have sound, rigorous performance guarantee of false positive and false negative rates.

Bifet and Gavalda further refined ADWIN0 and introduced ADWIN as its implementation algorithm (Bifet & Gavalda, 2007). They showed empirically that ADWIN outperforms other well-known concept detection methods, such as (Gama et al., 2004) and (Kifer, Ben-David, & Gehrke, 2004) in terms of false positive rate and false negative rate. Several notable improvements made are as follows:

- Usage of Bernstein bound instead of Hoeffding bound. Hoeffding bound offers sound estimation regardless of statistical distributions but in exchange, it overestimates the probability of large deviations occurring. Bernstein bound can establish a tighter estimation, when variance is provided.
- Reducing the number of hypothesis tests. ADWIN0 is computationally expensive because it needs to perform $(n-1)$ tests on a sub-window containing n elements. ADWIN uses a variant of exponential histogram developed by Datar et al. to maintain sufficient statistics (2002).

3.4 Concept-based Decision Tree (CBDT)

As mentioned previously, research in data stream mining typically focuses on incremental learning in which a model is continuously revised as new examples arrived. The motivation of the approach is to avoid the necessity of reconstructing models as new batches of examples arrive.

Incremental learning algorithm, such as VFDT (Domingos & Hulten, 2000) and its derivatives, typically makes use of a *time-based* sliding window to help maintain sufficient statistical information necessary for determining when to update the model. Such windowing schemes allow models to quickly adapt to new patterns. The disadvantage of this scheme, however, is that old patterns are forgotten once examples associated with them slide out of the window (or in other words, expired) (Wang et al., 2003). Hence, time-based sliding window does not efficiently handle re-emergence of old patterns as every concept change is treated as new pattern arriving.

Hoeglinger and Pears (2007) took a radical approach to tackle this issue, proposing the idea of concept-based approach. It is similar to time-based windowing in terms of the need to selectively discard "stale" information due to memory constraints imposed by stream mining. The difference lies in the criteria of information to discard; the decision to discard in time-based approach is purely based on the age of examples, whereas in a concept-based approach it is based on usage-statistics. The general idea of concept-based approach is that highly relevant features will be frequently used. Additionally, the more information collected about particular features, we can learn more details about the feature (Hoeglinger & Pears, 2007).

In a later study, Hoeglinger and Pears present a concept-based mining system called CBDT (Concept-based Decision Tree) (2009) by maintaining an ensemble of decision trees, or *forest* where every individual tree is grown separately to represent a concept sharing the same examples. This approach differs from traditional with ensemble learning. In a traditional forest of trees scheme, trees are grown on different partitions of training examples, and prediction is done by averaging the results of individual tree (Breiman, 1996).

CBDT adapts to changing concepts by maintaining a forest of trees, each grown separately on distinct attributes. When a concept drift occurs, CBDT simply switches to a better tree in the forest. This mechanism preserves both "strong" and "weak" patterns of older concepts and is shown to adapt very quickly to recurring patterns (Hoeglinger et al., 2009).

The main disadvantage of CBDT is the memory size required to maintain the forest. CBDT creates trees as much as the number of attributes in the stream. Feature selection in stream mining is very hard to do because of its dynamic nature; an attribute that is previously irrelevant might turn out to be highly important at later points in the stream.

3.5 Discrete Fourier Transform (DFT)

Fourier transform started from a simple insight, that any signal can be represented in terms of periodic functions (Arfken & Weber, 2001). The essence of Fourier transform is deconstruction of an arbitrary waveform into separate sinusoids of differing frequencies and amplitudes. If these sinusoids sum up to the original waveform, then we have determined the Fourier transform of the waveform. Mathematically, this relationship can be defined as follows:

$$\mathcal{F}(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-j2\pi\omega t}\, dt \tag{3.2}$$

where $f(t)$ is the waveform to be deconstructed into a set of sinusoids, $\mathcal{F}(\omega)$ is the Fourier transform of $f(t)$ and j is the imaginary number $\sqrt{-1}$.

The definition given above is for continuous Fourier Transform, which is unsuitable for digital machine calculations, such as a CPU. In practice, such machines do not receive a function as input, but rather *samples* of the function which is captured at a constant rate. Furthermore, it would be prohibitive to deconstruct a waveform into an infinite number of sinusoids in terms of memory usage. These factors drove the development of Discrete Fourier Transform. Now let us consider Fourier Transform in the case of a discrete function $f(t) \rightarrow f(t_n)$, where $t_n \equiv n\,\Delta$ with $n = 0, 1, \ldots, N-1$, where N is the total number of partitions (or sinusoids) to be considered. We can then write out the Discrete Fourier transform $\mathcal{F}(k)$ as

$$\mathcal{F}(k) \equiv \sum_{n=0}^{N-1} f(n)\, e^{-j2\pi nk/N} \tag{3.3}$$

where $k = 0, 1, \ldots, N-1$. Note that $e^{-j2\pi/N}$ is a constant and thus can be calculated and stored in advance for multiple values of N. It is usually known as phase factor (Gentleman & Sande, 1966), $W_N = e^{-j2\pi/N}$. Hence the formula can now be succinctly written as

$$\mathcal{F}(k) \equiv \sum_{n=0}^{N-1} f(n)\, W_N^{nk} \tag{3.4}$$

The ability of the Discrete Fourier Transform to deconstruct any waveform into sum of sinusoids is useful as it is easier to analyze a signal in frequency spectrum, as opposed to the time domain. For instance, a transmission wave can be decomposed into a message and noise frequency spectra. Another feature of DFT is that a waveform can be approximated with a finite number of partitions N, albeit with some loss of information. This kind of representation has much smaller storage requirement compared to storing the samples of a waveform itself.

Despite all of the benefits mentioned, DFT comes with a caveat. Note that $e^{-j2\pi nk/N}$ is a complex number in polar form, which means that each component of sum in the formula involves N complex multiplications. Counting the total number of the multiplications needed to be performed gives us the cost of calculating a DFT.

$$\text{Cost of DFT} = \text{number of } k \,.\, N = N^2$$

As N is increased, the DFT approximation is closer to the origin signal, but the cost of calculation also grows prohibitively. The exponential growth of the cost makes it extremely difficult to perform DFT in "higher" resolution. DFT calculation can be performed much efficiently using an algorithm called Fast Fourier Transform, which will be further described in the next section.

3.5.1 Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) is an algorithm to rapidly compute Discrete Fourier Transform. The introduction of FFT algorithm by Cooley and Tukey enables DFT computations which were previously prohibitive (1965). Since then, FFT is widely used in many applications in science, engineering and mathematics. FFT reduced Fourier analysis to a practical procedure that can be applied effectively for wide range of seemingly unrelated fields of application from music synthesizing to biomedical engineering to radar and communications (Brigham, 1988).

FFT (or Cooley-Tukey algorithm) uses divide-and-conquer approach to recursively break down a DFT of size N into two sets of smaller DFTs, even and odd sets. Thus, $f(n)$ is an interleaving of even $f(2m)$ and odd $f(2m+1)$ sequences.

$$\begin{aligned}\mathcal{F}(k) \equiv\ & \text{DFT of even sequence of} f(n) + \text{DFT of odd sequence of } f(n) \\ =\ & \sum_{m=0}^{N/2-1} f(2m)\, W_N^{(2m)k} + \sum_{m=0}^{N/2-1} f(2m+1)\, W_N^{(2m+1)k}\end{aligned}$$

The trick to FFT is to convert the phase factor for N partitions into that of $N/2$ partitions $W_N \to W_{N/2}$, which implies $W_N^{2mk} = W_{N/2}^{mk}$. Hence the equation above can be written as

$$\mathcal{F}(k) \equiv \sum_{m=0}^{N/2-1} f(2m)\, W_{N/2}^{mk} + W_N^k \sum_{m=0}^{N/2-1} f(2m+1)\, W_{N/2}^{mk}, \tag{3.5}$$

FFT recursively breaks down each sequence further into even and odd sequences, which dramatically improves the cost of calculation. It reduces the number of complex multiplications from what was previously $O(N^2)$ into $O(NlogN)$. This improvement is so significant that it propels the proliferation of FFT usage in a wide range of applications since its inception in 1965 (Brigham, 1988).

3.6 Converting a Decision Tree into Fourier Spectrum

A decision tree can be transformed into Fourier form by applying the DFT to paths of the tree. We are using binary decision trees for the purpose of illustrating the process; however, the process is also applicable to arbitrary *n-ary* domain (Kargupta, Park, & Dutta, 2006). Therefore, let us define the j-th Fourier Coefficient $\omega_{\mathbf{j}}$ for d number of binary features.

$$\begin{aligned} \omega_{\mathbf{j}} &= \tfrac{1}{2^d} \sum_{\mathbf{x}} f(\mathbf{x}) \psi_{\mathbf{j}}(\mathbf{x}) \\ \psi_{\mathbf{j}}(\mathbf{x}) &= (-1)^{(\mathbf{j}.\mathbf{x})} \end{aligned} \tag{3.6}$$

$f(\mathbf{x})$ is the classification outcome (the value of leaf node) for path $\mathbf{x}$. $\psi_{\mathbf{j}}(\mathbf{x})$ is the Fourier Basis function; $(\mathbf{j}.\mathbf{x})$ is the inner product operation between vectors $\mathbf{j}$ and $\mathbf{x}$ with $\mathbf{j}, \mathbf{x} \in \{0,1\}^d$.

Each partition $\mathbf{j}$ uniquely corresponds to a certain subset of features in the problem space. The *order* of a coefficient is the same as the order of a partition; that is the number of non-zero values in vector $\mathbf{j}$. For example, $\mathbf{j} = (000)$ has an order of 0, whereas $\mathbf{j} = (001, 010, 100)$ has an order of 1.

We present a simple example of boolean decision tree in 3-attribute setting

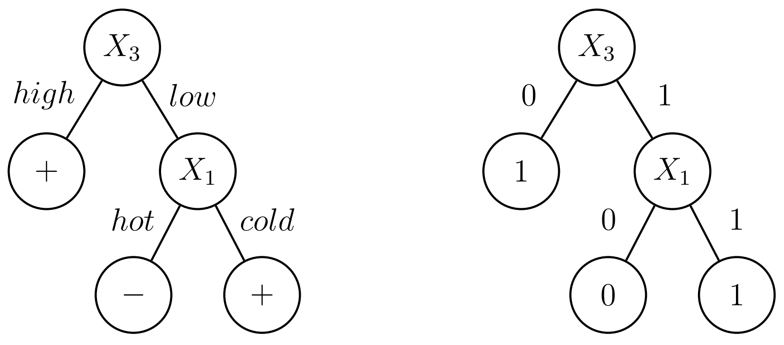


Figure 3.2: A binary Decision Tree in symbolic form (Left) and its numerical representation (Right)

(x_1, x_2, x_3) as shown in Figure 3.2. Note that the Fourier transform is based on numerical (algebraic) representation rather than symbolical. Thus for the remainder of this thesis, we assign numerical forms to illustrate tree examples.

x_1	x_2	x_3	$f(\mathbf{x})$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$\mathbf{j}[x_1, x_2, x_3]$	$\boldsymbol{\omega_j}$
000	3/4
001	1/4
010	0
011	0
100	-1/4
101	1/4
110	0
111	0

Table 3.1: Decision Tree from Figure 3.2 as truth table (Left) and its respective Fourier Coefficient table (Right)

The tree shown in Figure 3.2 contains 3 schemata $(* * 0,\ 0 * 1,\ 1 * 1)$. A schema represents a path from root node to a particular leaf node, e.g. schema $0 * 1$ corresponds to $x_3 \rightarrow x_1 \rightarrow 0$. The wildcard character denotes a collection of vectors. Thus schema $0 * 1$ covers vectors 001 and 011, both having $f(\mathbf{x}) = 0$. Because there are 3 features, there are 2^3 coefficients where $\mathbf{j} \in \{000, 001, 010, \ldots 111\}$.

Fourier coefficients ω_{000} and ω_{011} can then be calculated as follows:

$$\begin{aligned}
\omega_{000} &= \tfrac{1}{2^d}\sum_{\mathbf{x}} f(\mathbf{x})\psi^{000.\mathbf{x}} \\
&= \tfrac{1}{8}f(000)(-1)^{000.000} + \tfrac{1}{8}f(001)(-1)^{000.001} + \\
&\quad \tfrac{1}{8}f(010)(-1)^{000.010} + \tfrac{1}{8}f(011)(-1)^{000.011} + \\
&\quad \tfrac{1}{8}f(100)(-1)^{000.100} + \tfrac{1}{8}f(101)(-1)^{000.101} + \\
&\quad \tfrac{1}{8}f(110)(-1)^{000.110} + \tfrac{1}{8}f(111)(-1)^{000.111} \\
&= \tfrac{(1+0+1+0+1+1+1+1)}{8} = \tfrac{3}{4}
\end{aligned}$$

$$\begin{aligned}
\omega_{011} &= \tfrac{1}{2^d}\sum_{\mathbf{x}} f(\mathbf{x})\psi^{011.\mathbf{x}} \\
&= \tfrac{1}{8}f(000)(-1)^{011.000} + \tfrac{1}{8}f(001)(-1)^{011.001} + \\
&\quad \tfrac{1}{8}f(010)(-1)^{011.010} + \tfrac{1}{8}f(011)(-1)^{011.011} + \\
&\quad \tfrac{1}{8}f(100)(-1)^{011.100} + \tfrac{1}{8}f(101)(-1)^{011.101} + \\
&\quad \tfrac{1}{8}f(110)(-1)^{011.110} + \tfrac{1}{8}f(111)(-1)^{011.111} \\
&= \tfrac{(1+0-1+0+1-1-1+1)}{8} = 0
\end{aligned}$$

After calculating Fourier Coefficients for all possible $\mathbf{j}$ vectors, we can construct a Fourier Coefficient table as shown in Table 3.1. Note that every coefficient where x_2 is selected is 0, or in other words $\omega_{*1*} = 0$. This is consistent with a Lemma proven by Park (2001, p. 56) which guarantees that ω_j will be 0, whenever an attribute not appearing in the tree has value 1 for that particular attribute in its corresponding schema (Park, 2001; Kargupta et al., 2006). Based on this, we can selectively compute only the coefficients for attributes that actually appears in the tree since the rest of the coefficients would be zero. Hence we can build a compacted table as shown in Table 3.2

3.7 Problems with DFT

Unlike traditional data mining from a static dataset, patterns of data classification can change dynamically in stream mining due to shifting distribution of instances, popularly known as concept drift. Therefore, it is possible for

x_1	x_3	$f(\mathbf{x})$
0	0	1
0	1	0
1	0	1
1	1	1

$\mathbf{j}[x_1, x_3]$	$\boldsymbol{\omega_j}$
00	3/4
01	1/4
10	-1/4
11	1/4

Table 3.2: Compacted truth table and Fourier Coefficient table after removing attribute not appearing in the tree, x_2

a stream learner to grow large decision trees as adaptation to new concepts. This raises several concerns in applying DFT to decision trees in real application.

Size of Problem Space

The cost of calculating the entire Fourier Coefficient table is dependent on the size of the tree being converted. To be precise, the size of problem space is dependent on the number of distinct attributes appearing in the decision tree. As demonstrated in previous section, that a 3-binary feature problem space (Table 3.1) can be reduced to a 2-binary feature problem space (Table 3.2) because only 2 attributes actually appear in the tree.

Let P be a set of attributes appearing in the tree, $P \subseteq \{x_1, x_2, \ldots x_d\}$. The size of problem space in binary domain can be defined as:

$$S(_P) = 2^{|P|} \tag{3.7}$$

The equation shows that the size of problem space grows exponentially with respect to the number of distinct attributes appearing in a tree. The size of a tree affects the number of distinct attributes appearing in that tree. Thus, the size of problem space to be computed may turn out to be prohibitive as we convert a decision tree grown over stream instances.

Multivariate DFT

In Digital Signal Processing, the DFT is used to convert signal values from the time domain to the frequency domain. This process is a one-dimensional transformation, thus making it possible for the FFT to work. The FFT algorithm exploits the fact that Fourier Basis functions ψ_n can be pre-computed and stored in a table to be reused.

This is not the case with the DFT of decision trees where transformation is from the $\mathbf{x}$ vector domain to the $\mathbf{j}$ vector domain, where $\mathbf{x}$ and $\mathbf{j}$ are of variable-length, depending on the number of appearing attributes in the decision tree. This explains why our calculation of Fourier Basis functions involves inner product operation, as shown in Equation 3.6. This multivariate nature of decision trees renders it infeasible to pre-compute and store Fourier Basis functions as it is hard to determine when they are reusable or not.

Based on these insights, we can define the cost of calculating the Fourier Coefficient table of a decision tree in the binary domain as the total number of Fourier Basis functions generated over the entire problem space:

$$G(_P) = S(_P)V(_P) = S(_P)(2^{|P|}) = 2^{2|P|} \tag{3.8}$$

$G(_P)$ is the total number of $\psi_j(\mathbf{x})$ generated to calculate a complete Fourier Coefficient table considering only the attributes appearing in the tree $V(_P)$ is the number of Fourier Basis function to be calculated.

3.8 Existing Solutions

In the previous section, we have discussed the costly nature of DFT. In order to mitigate this cost, there are several solutions proposed (Kargupta & Park, 2004; Park, 2001):

1. Exploiting the properties of Decision Trees in Fourier space.

2. Using schema-based Fourier Coefficient methods.

3.8.1 Properties of Decision Trees in Fourier Space

Suppose we wish to compute in a 3-attribute space, given the tree example in Figure 3.2. We can then compute the amount of *energy* contained by a coefficient ω_j as follows:

$$E_{\omega_j} = |\omega_j^2| \tag{3.9}$$

The total *energy* of order n, E_n, is the sum of energy for all partition $\mathbf{j}$ that is of order n. e.g. for $n = 2$,

$$\begin{aligned} E_2 = & \; E_{\omega_{011}} + E_{\omega_{101}} + E_{\omega_{110}} \\ = & \; 0.0625 \end{aligned}$$

In similar way, we can calculate energy of other orders: $E_0 = 0.5627, E_1 = 0.125, E_3 = 0$. Note that energy diminishes as order is increased. In fact, energy decays exponentially with respect to increase of order (Linial, Mansour, & Nisan, 1993; Mansour, 1994; Park, 2001; Kargupta & Park, 2004). Due to this property, Park suggested that DFT should only be performed on "low" orders (2001). This allows us to effectively reduce the number of partitions $\mathbf{j}$ to be calculated by specifying the maximum threshold of order t. Hence the size of problem space from Equation 3.7 can then be redefined as:

$$S(_P) = \sum_{n=0}^{t} \binom{|P|}{n}, \qquad \text{if } t \leq |P| \tag{3.10}$$

3.8.2 Schema-based Fourier Coefficient Calculations

Kargupta and Park (2004) discovered that Fourier Coefficient calculation can be performed on schema-based formula by exploiting the knowledge that $f(\mathbf{x})$ for every $\mathbf{x}$ vector that is member of a schema $\mathbf{h}_{(i)}$ is constant. Therefore, Fourier Coefficient formula can be rewritten where $f(\mathbf{h}_{(i)})$ is the class outcome of schema $\mathbf{h}_{(i)}$ as:

$$\omega_j = \frac{|\mathbf{h}_{(1)}|}{2^d} f(\mathbf{h}_{(1)})\psi_{\mathbf{j}}(\mathbf{h}_{(1)}) + \ldots + \frac{|\mathbf{h}_{(n)}|}{2^d} f(\mathbf{h}_{(n)})\psi_{\mathbf{j}}(\mathbf{h}_{(n)}), \tag{3.11}$$

Note that Equation 3.11 is in binary domain. Using this simplified formula, we can calculate ω_{000} and ω_{011} respectively:

$$\begin{aligned} \omega_{000} = & \tfrac{4}{8} f(**0)\psi_{000}(**0) + \tfrac{2}{8} f(0*1)\psi_{000}(0*1) + \tfrac{2}{8} f(1*1)\psi_{000}(1*1) \\ = & \tfrac{4}{8} + 0 + \tfrac{2}{8} = \tfrac{3}{4} \end{aligned}$$

$$\begin{aligned} \omega_{011} = & \tfrac{4}{8} f(**0)\psi_{011}(**0) + \tfrac{2}{8} f(0*1)\psi_{011}(0*1) + \tfrac{2}{8} f(1*1)\psi_{011}(1*1) \\ = & \tfrac{4}{8}(1-1+1-1) + 0 + \tfrac{2}{8}(1-1) = 0 \end{aligned}$$

Note that schema $0*1$ has classification outcome 0. This particular schema will always contribute 0 when calculating coefficients regardless of the result of inner products associated to that schema. Hence in practice, we only need to consider schemata with class outcome 1 when using schema-based Fourier Coefficient calculation.

Another schema-based computation is proposed by Sripirakas et al (2015) showing that the Fourier basis function of a schema can be computed directly. The proposed theorem is for n-ary feature space but can be simplified for binary feature space. The Fourier base function of partition $\mathbf{j}$ and schema S in the binary feature space can be computed as follows

Case 1: If there exists at least one $(1,*)$ combination of corresponding features in partition $\mathbf{j}$ and schema S, then $\psi_{\mathbf{j}}(s) = 0$.

Case 2: If there exists k combinations of $(0,*)$ pairing of corresponding features in $\mathbf{j}$ and S, then

$$\psi_{\mathbf{j}}(S) = 2^k(-1)^l, \tag{3.12}$$

where l is the number of $(1,1)$ combinations. This particular method completely bypasses the vector inner product operations and effectively reduces computational overheads when computing the entire Fourier spectrum. We can compute ω_{000} and ω_{011} of the tree in the previous example using Sripirakas' optimisation as follows:

$$\begin{aligned}\omega_{000} =&\ \tfrac{1}{2^d}(f(**0)\psi_{000}(**0) + f(0*1)\psi_{000}(0*1) + f(1*1)\psi_{000}(1*1))\\ =&\ \tfrac{1}{8}(2^2+0+2^1) = \tfrac{3}{4}\end{aligned}$$

$$\begin{aligned}\omega_{011} =&\ \tfrac{1}{2^d}(f(**0)\psi_{011}(**0) + f(0*1)\psi_{011}(0*1) + f(1*1)\psi_{011}(1*1))\\ =&\ \tfrac{1}{8}(0+0+0) = 0\end{aligned}$$

By exploiting the properties of Fourier coefficients and using schema-based Fourier Coefficient calculation, the total number of Fourier Basis functions generated can be reduced. We can then revise Equation 3.8 as:

$$G(_P) = S(_P)V(_P) = \begin{cases} 2^{|P|}\tau & if|P| \leq t \\ \left(\sum_{n=0}^{t} \binom{|P|}{n}\right)\tau & if|P| > t \end{cases} \tag{3.13}$$

where τ is a constant representing the number of schemata with class outcome 1, t is the maximum number of order to calculate. Therefore, the cost of calculating Fourier coefficients now depends on the value of maximum order threshold.

These techniques, however, do not reduce the underlying exponential growth of the complete problem space. Furthermore, the number of coefficients to compute is still prohibitive because the growth rate is not linear. To get a better picture, let us compare two hypothetical trees, both sharing the same τ, containing distinct attributes of 10 and 20 ($|P_1|$, $|P_2|$) respectively, and maximum order $t = 5$.

$$G(_{P1}) = \left(\sum_{n=0}^{5} \binom{5}{n} \right) \tau = 638\tau$$

$$G(_{P2}) = \left(\sum_{n=0}^{5} \binom{10}{n} \right) \tau = 21700\tau$$

The DFT for the first tree involves computing 638 Fourier Basis functions per schema, whereas the cost rate on applying the DFT to the second tree jumped to 21700 per schema. As shown clearly, when we merely double the number of distinct attributes appearing in the trees, the effort per schema to compute the Fourier coefficients has grown by more than 30 times.

In practice, the cost of DFT will be managed by reducing the maximum order threshold. This will improve the runtime needed to compute the DFT, but would also mean more information loss. As a consequence, accuracy of resulting coefficient table may drop. This motivates us to seek an alternative way to calculate Fourier coefficients which will be presented in Chapter 4.

Summary

We have described the theoretical fundamentals behind each component used in our experimental study. As discussed in this chapter, the DFT is a very useful tool for spectral analysis but it is an inherently expensive transformation. FFT was developed to bring down the computation cost of DFT to a more manageable growth.

From the perspective of stream mining with recurring concepts, DFT is a valuable tool because it enables us to represent a decision tree in a very compact form, with negligible difference in output predictions. FFT exploits the fact that phase factors for univariate transform are constant. However, decision trees in Fourier domain are multivariate in nature and current approaches to mitigate the problems of applying the DFT to decision trees does not address the underlying exponential growth. Hence, it is necessary to find alternative way to reduce the cost of transforming decision trees into the Fourier domain, which is the main theme of this thesis. We will address this issue and propose a solution in the next chapter.

CHAPTER 4

Research design

In this chapter we present the core methods used in this research. The main research contribution that we make in this thesis is to reduce the computational complexity of the Fourier spectrum derivation from a given decision tree. As discussed before in this thesis spectrum derivation has inherently an exponential time complexity in the number of attributes that appear in the Decision tree. The methods that we present in this chapter are designed to reduce this time complexity by applying a decomposition scheme that is similar in spirit to the well known Fast Fourier Transform (FFT). However, we note that the FFT cannot be used directly in this research as the FFT applies to a univariate case, whereas the problem that we address in this research is inherently a multivariate case.

4.1 Strategies to Improve Time Performance

In order to increase the speed of computation of Fourier coefficients, we adopt two strategies; 1) Aggressively reduce problem space by a divide and conquer strategy and 2) Infer a Fourier coefficient table from an existing Fourier coefficient table.

Divide and conquer is a very well-known and established strategy in mathematical and computer science fields as a technique to reduce running time (Kleinberg & Tardos, 2006, p. 210). A couple of notable examples of algorithm using this strategy are merge sort and Fast Fourier Transform (Kingston &

Kingston, 1990, p. 65). A divide and conquer strategy works as follows. Divide an instance of problem into two partitions (or more) of smaller instances of same type of problem and solve each instances recursively. The solution to the original problem instance resulted from merging of the solutions of the small problem instances.

Partially inspired by the FFT algorithm, our approach to compute the Fourier spectrum of a decision tree follows the same principle mentioned above. Divide a tree into two parts, left and right sub-trees, construct Fourier coefficients for each sub-tree recursively, and assemble the complete Fourier Coefficients for the original tree.

For this approach to work there are two crucial pieces of information. First, the relationship between Fourier coefficients of sub-trees and Fourier coefficients of original tree. It turns out that Fourier coefficients of tree is simply the sum of Fourier coefficients of each sub-tree. Second is whether this strategy will ultimately reduce the size of our problem space or not. We will discuss this further in Section 4.2.

Another strategy to improve performance is to infer a Fourier coefficient table from an existing Fourier coefficient table. The idea behind this approach is really simple. Two trees with similar structure should also be similar in Fourier space. Thus, the effort to calculate Fourier coefficients from similar tree should also be small. show that it is indeed possible to calculate Fourier coefficients from previously-calculated ones. This will be described further in Section 4.3.

4.2 Fourier Spectrum of Sub-trees

As mentioned in Section 4.1, the Fourier spectrum of an arbitrary tree is the same as the sum of Fourier spectra of each sub-tree. We show that there is no information loss when the coefficients are calculated this way. The reason that this holds is because when a tree is decomposed into sub-trees, schemata are also being split. As a consequence, the class outcome $f(x)$ at each sub-tree is partitioned without interfering each other. Therefore, we define our first Theorem as follows:

Theorem 1: *Let L and R be the left and right sub-trees of a decision tree T in binary domain. The Fourier coefficients of T are the sums of Fourier coefficients of L and R*

Proof: When a tree is split, $f(x)$ would be partitioned into $f_L(x)$ and $f_R(x)$ in a way that they do not interfere (or overlap) with each other. Thus, $\forall \mathbf{x}$, $f(\mathbf{x}) = f_L(\mathbf{x}) + f_R(\mathbf{x})$

$$\begin{aligned}
\omega_{\mathbf{j}} &= \tfrac{1}{2^d} \sum_{\mathbf{x}} f(\mathbf{x}) \psi_{\mathbf{j}}(\mathbf{x}) \\
&= \tfrac{1}{2^d} \sum_{\mathbf{x}} (f_L(\mathbf{x}) + f_R(\mathbf{x})) \psi_j(\mathbf{x}) \\
&= \tfrac{1}{2^d} \sum_{\mathbf{x}} f_L(\mathbf{x}) \psi_{\mathbf{j}}(\mathbf{x}) + \frac{1}{2^d} \sum_{\mathbf{x}} f_R(\mathbf{x}) \psi_{\mathbf{j}}(\mathbf{x}) \\
&= \omega_{\mathbf{j}}^L + \omega_{\mathbf{j}}^R,
\end{aligned}$$

where d is the number of distinct attributes appearing in tree T.

To illustrate this, we split the tree shown in Figure 3.2 into Left and Right sub-trees as shown in Figure 4.1. The trees can then be represented in a table shown in Table 4.1. Notice that the vectors are rearranged to emphasize the partitioning of $f(\mathbf{x})$ space when a tree is decomposed into sub-trees.

We also observed that the properties of Fourier spectrum of decision trees described in Section 3.8.1 still holds for these sub-trees. For the left sub-tree, x_3 is the only appearing attribute. Therefore, Fourier coefficients for when either x_1 or x_2 is selected are zero. i.e. $\omega_{\mathbf{j}}^L = 0$, where $\mathbf{j} \in \{1**\} \cup \{*1*\}$. Whereas only x_2 does not appear in the right sub-tree, resulting in $\omega_{\mathbf{j}}^R = 0$, where $\mathbf{j} \in \{*1*\}$. This allows us to calculate the Fourier coefficients of an

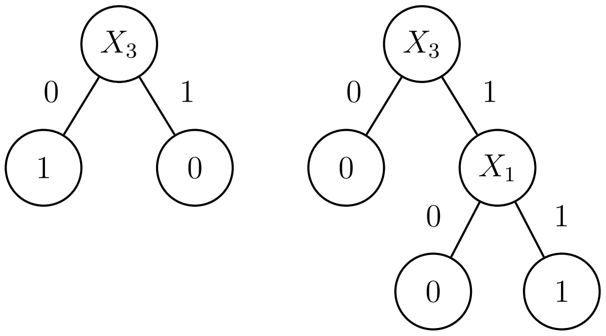


Figure 4.1: Left and Right sub-trees of sample tree shown in Figure 3.2

x_1	x_2	x_3	$f(\mathbf{x})$	$f_L(\mathbf{x})$	$f_R(\mathbf{x})$
0	0	0	1	1	0
0	1	0	1	1	0
1	0	0	1	1	0
1	1	0	1	1	0
0	0	1	0	0	0
0	1	1	0	0	0
1	0	1	1	0	1
1	1	1	1	0	1

$\mathbf{j}[x_1, x_2, x_3]$	$\omega_{\mathbf{j}}$	$\omega_{\mathbf{j}}^L$	$\omega_{\mathbf{j}}^R$
000	3/4	1/2	1/4
010	0	0	0
100	-1/4	0	-1/4
110	0	0	0
001	1/4	1/2	-1/4
011	0	0	0
101	1/4	0	1/4
111	0	0	0

Table 4.1: Truth table (Left) and Fourier Coefficients (Right) of Left and Right sub-trees as shown in Figure 4.1 with vectors rearranged

individual sub-tree in the attribute space appearing only in that particular sub-tree, potentially reducing the total number of Fourier Basis functions needed to be computed to form the complete Fourier coefficients.

The interesting part of this Theorem is that decomposing a tree into sub-trees can be performed recursively until there are no more sub-trees that can be decomposed. As pointed out above, decomposing a tree allows us to compute in localized problem space than the complete problem space. The next logical question is whether the total computation in localized problem space of each sub-tree is lower than calculating in the complete problem space. This is where our second Theorem comes into picture.

Theorem 2: *Given a decision tree in binary domain T, The total number of Fourier coefficients to be calculated on each sub-tree is less than the total number coefficients calculated on T.*

Proof: Let $D \in x_1, x_2, ..x_n$ be the set of distinct attributes appearing in T where $d = |D|$. D_L and D_R are sets of distinct attributes appearing in left and right sub-trees respectively, satisfying the conditions $D_L \subseteq D, D_R \subseteq D$.

Therefore, $d_L = |D_L|$ and $d_R = |D_R|$. d, d_L and d_R are related such that $d_L = d - k, d_R = d - m, k < m$, where k and m are integers. This implies $d - k > d - m$.

The number of coefficients generated from calculating T and its sub-trees are: $S(D) = 2^d, S(D_L) = 2^{(d-k)}, S(D_R) = 2^{(d-m)}$. Therefore, the total number of coefficients calculated on both sub-trees is:

$$S(D_L) + S(D_R) = 2^{(d-k)} + 2^{(d-m)},$$

Thus, we have two possible cases:

Case 1: $k >= 1$,

$$\begin{aligned} S(D_L) + S(D_R) &< 2^{(d-k)} + 2^{(d-k)} \\ &< 2^{(d-k+1)} \\ &< 2^d \end{aligned}$$

Case 2: k = 0

$$\begin{aligned} S(D_L) + S(D_R) &= 2^d + 2^{(d-m)} \\ &> 2^d \end{aligned}$$

hence the theorem holds for Case 1.

Case 1 implies a non-balanced tree structure, therefore distinct split attributes are distributed unevenly between left and right sub-trees. It also implies that the bigger the intersection between D_L and D_R, coefficients to be computed also increased.

Case 2 tells us that if one of the sub-trees contains the same set of distinct attribute as that of origin tree, the total number of coefficient computations on sub-trees will be bigger than total number of computations when performing the DFT on origin tree T the normal way.

Using the Left and Right sub-trees example shown in Figure 4.1, let us compare the number of coefficients to be calculated on each sub-trees as well as the origin tree (Figure 3.2. The The number of coefficients for Left tree, Right tree, and origin tree are as follows:

$$S(D_L) = 2^1 = 2,$$
$$S(D_R) = 2^2 = 4,$$
$$S(D) = 2^2 = 4 < S(D_L) + S(D_R)$$

This is consistent with case 2 of our Theorem. Note that we exclude x_2 as it does not appear in the origin tree. For this particular example, it would be faster to just perform DFT on the origin tree instead of decomposing. However, in our implementation we did not include a decision making mechanism to determine whether to decompose or not. This is because we are more interested to observe the general runtime performance of DFT by aggressively splitting a tree in comparison with normal DFT. As such, this heuristics mechanism would be part of future works.

Let us consider another example of sub-trees as shown in Figure 4.2. For the given example, the number of coefficients at Left sub-tree is $2^2 = 4$, for Right sub-tree, it is $2^3 = 8$. When the two sub-trees are merged, the origin tree has 4 distinct attributes, therefore $2^4 = 16$ coefficients are going to be calculated. So this is an example where calculating localized coefficients is more beneficial.

Unfortunately, the size of problem space is still exponential. However, splitting the calculation task as calculating sub-trees still has its merits, al-

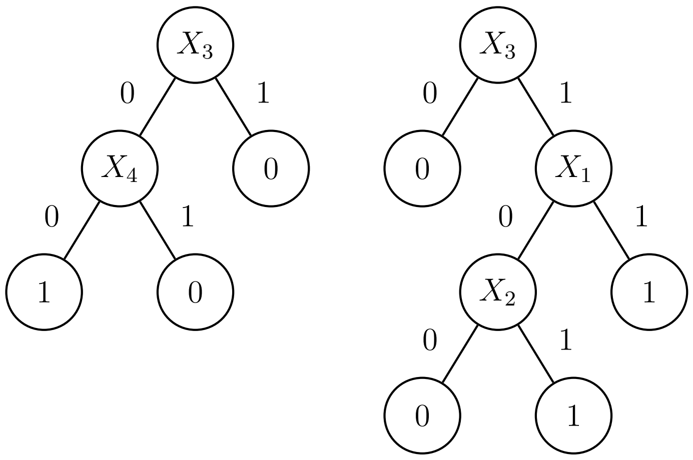


Figure 4.2: Left and Right sub-trees example where decomposing is beneficial

though the exact extent of the benefit is very much dependent on the structure of the tree built by the learner algorithm.

In practice, there is also an additional cost introduced by the necessity to combine the Fourier coefficients of sub-trees. This cost is hard to determine because of numerous factors. First, Fourier coefficient tables generated for sub-trees are of various sizes, so access time is not uniform. Second, there is an overhead prior to combining the Fourier coefficient tables. This is because Fourier coefficients to be combined are of different attribute sets. For example, one coefficient may have $\mathbf{j}$ vector which is consisted of x_1, x_2, x_3 attributes while another coefficient set may consist of only x_2 and x_3. The environment where the system is running will also affect the process. Thus, k must be much bigger than 1 for us to benefit from this approach.

The last important piece to complete this strategy that we have not mentioned so far is the splitting strategy. We can technically decompose a tree at every single split node, but this would have detrimental effect because the number of coefficient merging processes will also grow. It is also possible to use a heuristic-based splitting strategy to avoid the worst case condition highlighted in Theorem 3. In our implementation, however, a simple splitting strategy is used, that is to decompose from a split node of which both child nodes are split nodes. This strategy is much more suitable to our interest in studying the performance of a system that aggressively split a tree to compute the Fourier spectrum.

4.3 Adjusting Fourier Spectrum from an Existing Spectrum

We also show that it is indeed possible to compute certain Fourier coefficients, from previously calculated Fourier coefficients. The intuition to this approach is that two trees with similar structure should also be similar in Fourier space. Therefore the cost to infer Fourier coefficients from previously calculated Fourier coefficients of a similar tree should be lower than calculating those Fourier coefficients from scratch. This brings us to Theorem 3 as follows:

Theorem 3: *Given a tree in binary domain with class outcome $f(\mathbf{x})$, P is a set of changed vectors, $P \subseteq (\mathbf{x})$. If there are some changes in the paths such that the class outcome also changed to $f'(\mathbf{x})$, then*

$$\omega'_{\mathbf{j}} = \begin{cases} \omega_{\mathbf{j}} + \frac{1}{2^d}\sum\limits_{P} \psi_{\mathbf{j}}(P) & \text{if changed vectors from 0 to 1} \\ \omega_{\mathbf{j}} - \frac{1}{2^d}\sum\limits_{P} \psi_{\mathbf{j}}(P) & \text{if changed vectors from 1 to 0} \end{cases} \tag{4.1}$$

Proof: Let $\mathbf{x}$ be the union problem space of both trees, $Q \subseteq \mathbf{x}$, Q is a set of unchanged vectors. P does not intersect Q, therefore $f(P)$ and $f(Q)$ do not interfere with each other. The DFT of a decision tree in binary domain can be written in terms of changed and unchanged vector components as such:

$$\begin{aligned}
\omega_{\mathbf{j}} &= \tfrac{1}{2^d} \sum_{\mathbf{x}} f(\mathbf{x})\, \psi_{\mathbf{j}}(\mathbf{x}) \\
&= \tfrac{1}{2^d} \sum_{P} f(P)\, \psi_{\mathbf{j}}(P) + \frac{1}{2^d} \sum_{Q} f(Q) \psi_{\mathbf{j}}(Q)
\end{aligned}$$

We introduce a function $f_{delta}(\mathbf{x})$, such that $f'(\mathbf{x}) = f(\mathbf{x}) + f_{delta}(\mathbf{x})$. For unchanged vectors, $f_{delta}(Q)$ must be zero, implying $f'(Q) = f(Q)$. For changed vectors, $f'(P) \neq f(P)$ must hold. Because class outcome can only be either 0 or 1, we have two possibilities. First is when change of class outcome from 0 to 1. Second is when change of class outcome from 1 to 0. Fourier coefficients of tree after change can then be expressed as:

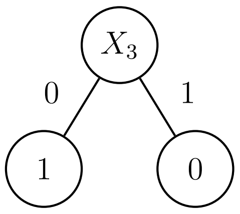
$$\begin{aligned}
\omega'_{\mathbf{j}} &= \tfrac{1}{2^d} \sum_{P} f'(P)\, \psi_{\mathbf{j}}(P) + \frac{1}{2^d} \sum_{Q} f'(Q)\, \psi_{\mathbf{j}}(Q) \\
&= \tfrac{1}{2^d} \sum_{P} (f(P) + f_{delta}(P))\, \psi_{\mathbf{j}}(P) + \frac{1}{2^d} \sum_{Q} f(Q)\, \psi_{\mathbf{j}}(Q) \\
&= \tfrac{1}{2^d} \sum_{P} f(P)\, \psi_{\mathbf{j}}(P) + \frac{1}{2^d} \sum_{Q} f(Q)\, \psi_{\mathbf{j}}(Q) + \frac{1}{2^d} \sum_{P} f_{delta}(P)\, \psi_{\mathbf{j}}(P) \\
&= \omega_{\mathbf{j}} + \tfrac{1}{2^d} \sum_{P} f_{delta}(P)\, \psi_{\mathbf{j}}(P)
\end{aligned}$$

Recall that we have 2 cases. First is when $f(P) = 0, f'(P) = 1$ holds true, implying $f_{delta}(P) = 1$. Second is when $f(P) = 1, f'(P) = 0$ holds true, implying $f_{delta}(P) = -1$. Substituting $f_{delta}(P)$ for both cases gives us

$$\omega'_{\mathbf{j}} = \begin{cases} \omega_{\mathbf{j}} + \tfrac{1}{2^d} \sum_{P} \psi_{\mathbf{j}}(P) & \text{if changed vector from 0 to 1} \\ \omega_{\mathbf{j}} - \tfrac{1}{2^d} \sum_{P} \psi_{\mathbf{j}}(P) & \text{if changed vector from 1 to 0} \end{cases}$$

Hence Theorem 3 holds.

Now let us work on an example to get a better picture of how this theorem works. Suppose we have trees in binary domain of which coefficients have been calculated as shown in Figure 4.3. A split then occurred on the right leaf node, creating a tree shown in Figure 4.4. The dimension of the tree



x_3	$f(\mathbf{x})$	$\omega_{\mathbf{j}}(\mathbf{x})$
0	1	1/2
1	0	1/2

Figure 4.3: Sample tree prior to split (Left) and its corresponding truth table and Fourier coefficients in compact form (Right)

prior to and after the split are different, therefore we need to first expand the **j** vector from $\{x_3\}$ feature space to $\{x_1, x_3\}$. Remember that Fourier coefficients for partitions which any attribute not appearing in the tree is selected is zero, meaning $\omega_{1*} = 0$. ω_0 is mapped to ω_{00} and ω_1 is mapped to ω_{01} respectively. There are two ways to deduce the changed vector P and the direction of the change. First is to compare the truth table directly and find where $f(\mathbf{x}) \neq f'(\mathbf{x})$. Second is to take note the class outcome of the leaf node where split takes place. In this case, leaf node outcome was 0 and split takes place where the right child path leads to a different class outcome, i.e. where $x_3 = 1$ and $x_1 = 1$. Thus $P \in \{11\}$, where P in x_1, x_3 dimension space and the direction of the change $f_{delta}(P) = 1$.

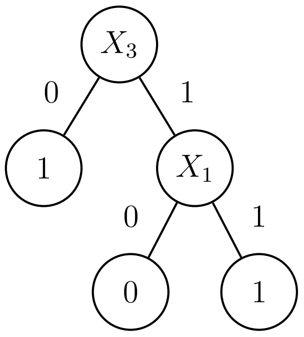
The Fourier coefficients of the tree after split can then be calculated using Equation 4.1. For example, let us compute $\omega'_{\mathbf{00}}$ and $\omega'_{\mathbf{10}}$.

$$\begin{aligned} \omega'_{00} = & \ \omega_{00} + \tfrac{1}{2^d}\sum_P \psi_{00}(P) \\ = & \ 1/2 + \tfrac{1}{2^2}\psi_{00}(11) \\ = & \ 1/2 + \tfrac{1}{4}(-1)^{00.11} = 3/4 \end{aligned}$$

$$\begin{aligned} \omega'_{10} = & \ \omega_{10} + \tfrac{1}{2^d}\sum_P \psi_{10}(P) \\ = & \ 0 + \tfrac{1}{2^2}\psi_{10}(11) \\ = & \ \tfrac{1}{4}(-1)^{10.11} = -1/4 \end{aligned}$$

x_1	x_3	$f(\mathbf{x})$	$\omega_{\mathbf{j}}(\mathbf{x})$
0	0	1	1/2
0	1	0	1/2
1	0	1	0
1	1	0	0

Table 4.2: Expanded truth table and Fourier coefficients of Figure 4.3



x_1	x_3	$f(\mathbf{x})$	$f'(\mathbf{x})$	$\omega_{\mathbf{j}}(\mathbf{x})$	$\psi_{\mathbf{j}}(P)$	$\omega'_{\mathbf{j}}(\mathbf{x})$
0	0	1	1	1/2	1/4	3/4
0	1	0	0	1/2	−1/4	1/4
1	0	1	1	0	−1/4	−1/4
1	1	0	1	0	1/4	1/4

Figure 4.4: State of tree after split (Left) and its truth table and Fourier coefficients prior and after adjustment (Right)

There are several consequences to this theorem. Firstly, it allows us to actively synchronize Fourier spectrum of a tree as it grows while processing input instances, as shown by our example earlier. This eager synchronization can be useful in real-time systems where instances arrive at extremely high speed and avoiding spikes in processor usage is desirable. Secondly, inference of Fourier coefficients will be extremely beneficial for scenarios involving recurring concepts. This is due to the fact that a concept can be represented by a tree structure (or a sub-tree structure) and may reoccur numerous times during the streaming process, especially for periodical data. However recurring concepts are not necessarily identical. A concept may reappear with slightly different values and this will be recaptured by the tree learner as a tree (or sub-tree) structure sharing some similarities with the previous occurrence (Sripirakas & Pears, 2014). Thus, the approach would pave the way for a possibility to build a "smart" caching scheme that is shareable between trees in a forest.

In this study, we only experimentally explore the former consequence of this theorem, while the latter will be left for future research as it requires a caching system in place. The next section describes the algorithm we designed to make advantage of the strategies mentioned.

4.4 memoFT

We present our solution, called memoFT (Memory-Optimization Fourier Transform), to compute the Fourier spectrum of a decision tree by implementing the theorems we have described. As the name implies, memoFT uses a structure in the memory to store Fourier coefficients. The tenet behind memoFT is: avoid computing coefficient table from scratch as much as possible. This is why memoFT needs to maintain memory storage for lookup purposes.

The core module of memoFT is a mechanism that can either build a coefficient table, or perform adjustment to a coefficient table based on its last-known state. Hence, it is heavily based on our third theorem. This core module is interfaced with two modules: *tree analyser* and *split analyser*. Figure 4.5 shows how all modules interact with each other. The split analyser is a module tasked with monitoring the occurrences of splits in a tree. Based on a split information, it generates a traversal plan, a changed vector and the direction of change for the core module to process. The tree analyser module scans a tree generates a collection of traversal plans for leaf nodes with class outcome 1.

The incremental nature of the core module allows memoFT to be used as an eager or lazy evaluation. In eager evaluation, memoFT track the splits in the tree and make adjustment to Fourier coefficient table as necessary with the help of the split analyser.

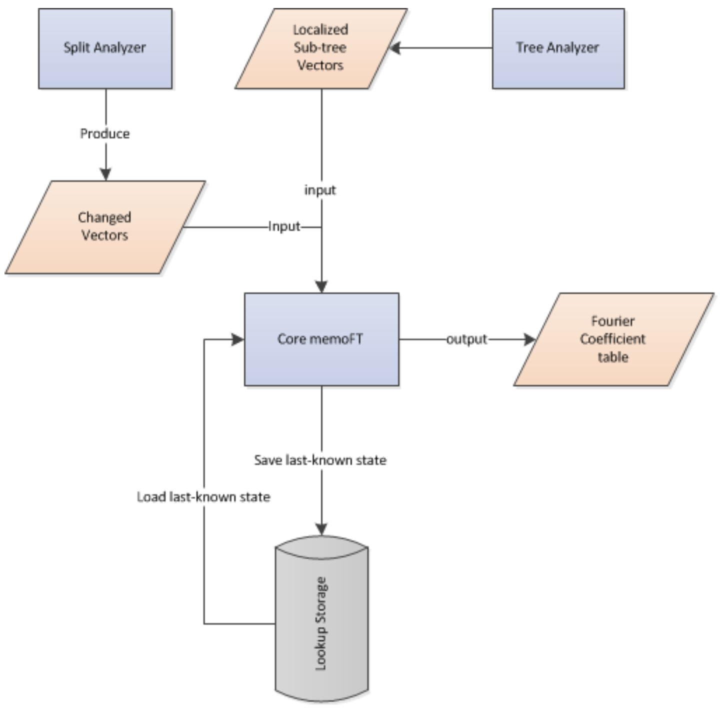


Figure 4.5: memoFT modules and their interactions

In the lazy evaluation mode, memoFT is extended to directly evaluate and build Fourier coefficient table with the help of the tree analyser module. memoFT is designed as an incremental mechanism to serve as a groundwork for future work which will be focused on a caching mechanism. This will be discussed in more detail in the Conclusion chapter of this Thesis.

4.4.1 Partial Encoding of Sub-tree Structures

memoFT maintains a structure that maps a sub-tree to its Fourier coefficients. This can be achieved by representing sub-tree as a series of string which are then associated to their respective Fourier coefficient table. To minimize the memory size, the string encoding format is adapted from a variant of succinct binary tree format (Jacobson, 1989), where a tree structure is represented by a string of bits in which split nodes are represented by 1s and leaf nodes by 0s. The only difference in our format is the sequence of the traversal, where we use a pre-order traversal (a type of depth first search), instead of a breadth-first traversal.

Suppose we have a tree as shown in Figure 4.6; split nodes are circle-shaped and leaf nodes are in rectangles for visual cues. A pre-order traversal will visit the nodes in the order of the numbers written in the figure. Hence, the structure of the example tree can be encoded as 11001100100.

The encoding strategy is limited to static trees, whereas our need is to retrieve coefficient tables for sub-trees with similar structure, rather than being exact. However, we noted that encoded strings of a pair of left and right sub-trees share the same encoding bits up until the split node where they branch out. We call such split nodes as *decomposition points*. Therefore, instead of encoding full sub-trees, we can encode sub-trees partially until the decomposition point, where we can assign storage bins for Fourier coefficients of left and right sub-trees of that decomposition point.

Figure 4.7 shows one of the possible pair of left and right sub-trees from a tree structure shown in Figure 4.6. The structure of left and right sub-trees will be encoded as 1011000 and 1010100 respectively. Both strings share the same series of bits, 101, until the decomposition point. Therefore 101 will be used as a key for storage or retrieval of coefficients of those left and right sub-trees.

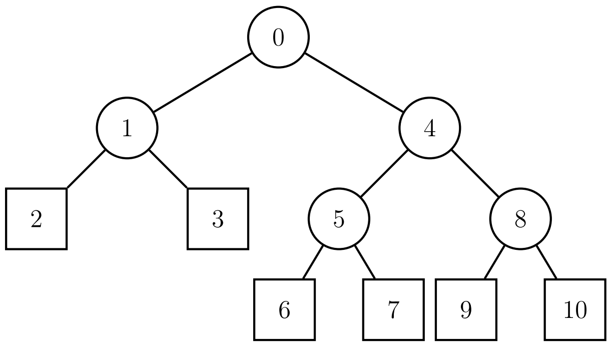


Figure 4.6: Tree structure example and its numbered sequence of traversal.

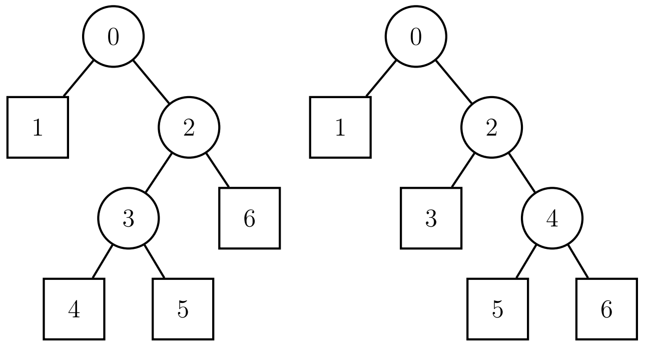


Figure 4.7: Left and Right sub-tree pairing example and their respective traversal sequences.

4.4.2 Core memoFT

The core module of memoFT is the heart of our solution where a tree is broken down into sub-trees, and Fourier coefficients are then calculated (or adjusted) in the local sub-tree feature space. To avoid recalculating Fourier coefficients of sub-trees that are unchanged, it requires a traversal plan to help it determine which sub-trees are affected by the Fourier coefficient calculation. This is necessary, because any changes in a sub-tree will affect the coefficients of that sub-tree at its higher levels, requiring additional work to update the stored coefficients of sub-trees in the memory. The pseudo-code of the core module of memoFT is shown by Algorithm 4.1 and Algorithm 4.2 below.

To summarize, the core module involves a three-step process: 1) Traverse the tree to identify the sub-trees affected by changes (or updates). 2) Calculate a new coefficient table, or perform adjustment to an existing one for the deepest decomposition point affected by change(s). 3) Update all coefficient tables of sub-trees identified by the first step. Using this approach, the algorithm also yields memory storage that is synchronized to the latest state of sub-trees of the origin tree, in addition to producing the final Fourier coefficients. Note that we always decompose the tree at the root node. It is to ensure that we have a static top-level decomposition point. It is harder to

Algorithm 4.1 Core memoFT: Tree Decomposition and Evaluate change

```
Input: origin tree $O$, string traversal plan, coefficient table repository $M$
Output: Fourier coefficients of $O$
1: Let $S$ be a stack of decomposition points.
2: for all visited node $N$ in $O$ as guided by traversal plan do
3:     if $N$ is decomposition node or $N$ is root node then
4:         push $N$ to $S$
5:     end if
6: end for
7: invoke $AdjustAndTabulate$ (Algorithm 4.2) to update $M$ to the current
   state of tree
8: return merged coefficient table at top-level decomposition point stored
   in $M$
```

determine which entry in the memory storage is the top-level decomposition point and without this static reference additional work will be required to scan the memory storage.

4.4.3 Extended memoFT

The extended module of memoFT is a wrapper to allow memoFT to calculate the Fourier coefficients of an entire tree, rather than continuously update Fourier coefficients at every occurring split in the tree. We call this mode of operation as *lazy evaluation.*

Eager evaluation has limited uses, where it is useful only when CPU usage spikes is not desirable. Otherwise, lazy evaluation would be preferable in most cases. Note that the DFT is used in tandem with a tree learner algorithm like follows: The learner grows a tree from data instances and the DFT is later used to compress the grown tree once conditions to stop growing are fulfilled. Hence, it would be more efficient to wait until a tree has "matured" as it eliminates the need to maintain synchronized states of Fourier coefficients of that tree.

The extension module of memoFT is shown in Algorithm 4.3. A decision tree is fully scanned and Fourier coefficients are then calculated incrementally by processing only the paths leading to class label 1 i.e. where $f(\mathbf{x}) = 1$. In this approach, $f(\mathbf{x})$ is effectively $f_{delta}(\mathbf{x})$ because it starts from empty state $f^0(\mathbf{x}) = 0$.

Suppose we have a matured decision tree as shown in Figure 4.8. The first step of lazy evaluation is to scan the tree and collect all the paths to leaf nodes. Paths leading to class label 0 are excluded because they do not contribute to the calculated coefficients. Paths are then processed starting from the shortest to the longest. In this case, $x_3 \rightarrow x_4$ is the first to be processed and $x_3 \rightarrow x_1 \rightarrow x_4 \rightarrow x_2$ is the last.

When the first, second and third paths are processed, there were no previously-calculated coefficients for the sub-trees where they were located (shown in Figure 4.9, Figure 4.10 and Figure 4.11 respectively). Hence coefficient tables are created and then stored in memory for all three sub-trees. When processing the last path, however, it shares the same sub-tree as the

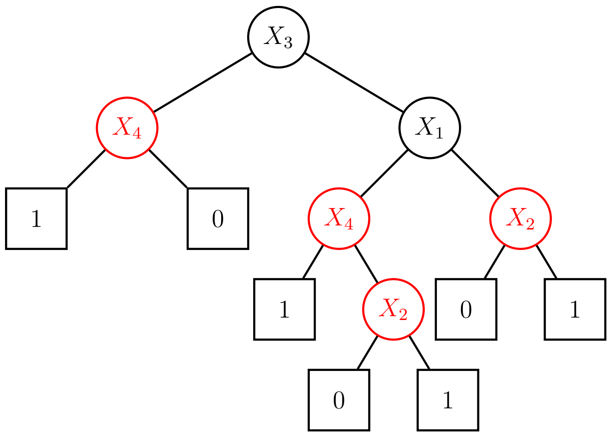
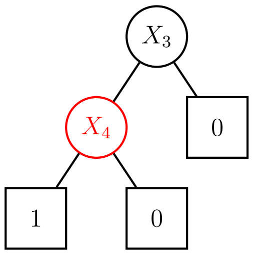
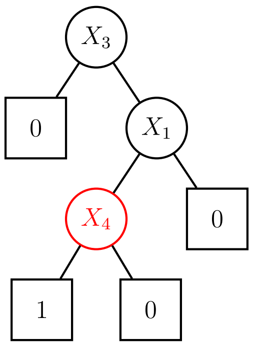


Figure 4.8: Example of a decision tree in binary domain to be evaluated by memoFT.



x_3	x_4	$f^0(\mathbf{x})$	$f_{delta}(\mathbf{x})$	$f(\mathbf{x})$	$\omega_\mathbf{j}$
0	0	0	1	1	0.25
0	1	0	0	0	0.25
1	0	0	0	0	0.25
1	1	0	0	0	0.25

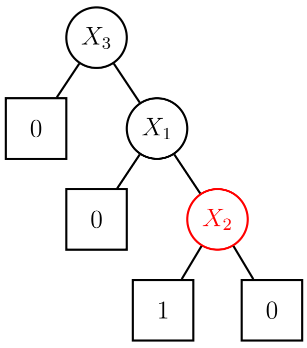
Figure 4.9: Processing first path of example tree in Figure 4.8



x_1	x_3	x_4	$f^0(\mathbf{x})$	$f_{delta}(\mathbf{x})$	$f(\mathbf{x})$	$\omega_\mathbf{j}$
0	0	0	0	0	0	0.125
0	0	1	0	0	0	0.125
1	0	0	0	0	0	0.125
1	0	1	0	0	0	0.125
0	1	0	0	1	1	-0.125
0	1	1	0	0	0	-0.125
1	1	0	0	0	0	-0.125
1	1	1	0	0	0	-0.125

Figure 4.10: Processing second path of example tree in Figure 4.8. Reorganized to show partition by X_3

second path. In this case, coefficient table for the sub-tree is accessed from memory and adjustment is made as shown in Figure 4.12.



x_1	x_2	x_3	$f^0(\mathbf{x})$	$f_{delta}(\mathbf{x})$	$f(\mathbf{x})$	$\omega_{\mathbf{j}}$
0	0	0	0	0	0	0.125
0	1	0	0	0	0	0.125
1	0	0	0	0	0	-0.125
1	1	0	0	0	0	-0.125
0	0	1	0	0	0	-0.125
0	1	1	0	0	0	-0.125
1	0	1	0	1	1	0.125
1	1	1	0	0	0	0.125

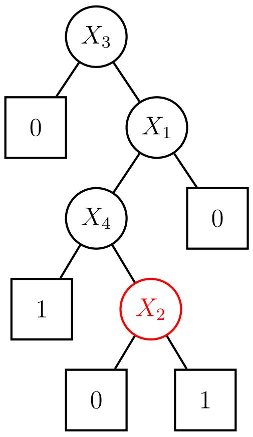
Figure 4.11: Processing third path of example tree in Figure 4.8. Reorganized to show partition by X_3

Algorithm 4.2 Adjust and Tabulate changes

```
Input: tabulation stack $S$, changed vector $\mathbf{x}$, change direction $f_{delta}(\mathbf{x})$
Output: updated state of repository $M$
 1: for all decomposition point $P$ in $S$ do
 2:     if $P$ is first element in $S$ then
 3:         if no prior record of $P$ in $M$ then
 4:             Let $T_1$ be the coefficient table of direct parent sub-tree of
                current decomposition point stored in $M$
 5:             Calculate new coefficient table $T_1$ based on $\mathbf{x}$ and $f_{delta}(\mathbf{x})$
 6:             Assign $T_2$ to the sub-tree where change occurred
 7:             Assign $T_1$ to the sub-tree not affected by change
 8:         else
 9:             Grab coefficient table $T$ of changed sub-tree from $M$ and
                adjust the coefficient values based on $\mathbf{x}$ and $f_{delta}(\mathbf{x})$
10:         end if
11:     else
12:         Let $P'$ be the decomposition point processed in a previous iter-
            ation,
            $T'$ be the combined coefficient table of sub-trees linked to $P'$
13:         Grab record entry for $P$ stored in $M$ and replace the coefficient
            table of the sub-tree where change occurred with $T'$
14:     end if
15: end for
```

Algorithm 4.3 memoFT - Lazy Evaluation

```
Input: decision tree $T$
Output: Fourier coefficient representation of $T$, and a memory storage containing coefficient tables of sub-trees
1: Let $M$ be an empty memory storage
2: Scan $T$ layer-by-layer and collect a list $L$ of split nodes closest to class label 1, i.e. $f(\mathbf{x}) = 1$ //tree analyser
3: for all Split node $N$ in $L$ do
4:     Determine change vector $\mathbf{x}$ and traversal plan caused by $N$. Direction of change is static, $f_{delta}(\mathbf{x}) = 1$
5:     Invoke core memoFT (Algorithm 4.1)
6: end for
7: return coefficient table created at the last invocation of core memoFT
```



x_1	x_2	x_3	x_4	$f(\mathbf{x})$	$f_{delta}(\mathbf{x})$	$\omega'_{\mathbf{j}}$
0	0	0	0	0	0	0.1875
0	0	0	1	0	0	0.0625
0	1	0	0	0	0	-0.0625
0	1	0	1	0	0	0.0625
1	0	0	0	0	0	0.1875
1	0	0	1	0	0	0.0625
1	1	0	0	0	0	-0.0625
1	1	0	1	0	0	0.0625
0	0	1	0	1	0	-0.1875
0	0	1	1	0	0	-0.0625
0	1	1	0	1	0	0.0625
0	1	1	1	0	1	-0.0625
1	0	1	0	0	0	-0.1875
1	0	1	1	0	0	-0.0625
1	1	1	0	0	0	0.0625
1	1	1	1	0	0	-0.0625

Figure 4.12: Processing fourth path of example tree in Figure 4.8

Summary

In this chapter we have presented the design aspect of our research, starting from identifying the problems of application of DFT to decision trees where the size of attribute space increases exponentially. We also presented our solution to reduce the size of attribute space by combining a divide and conquer strategy with usage of memory storage from theoretical perspective as well from an implementation (memoFT) perspective. Our next step is to evaluate our implementation using experiments in various stream conditions which will be described further in Chapter 5

Chapter 5

Experiments and Results

In the previous chapter we identified our research problems, proposed a solution from a theoretical perspective and presented memoFT as the implementation algorithm designed to compute the Fourier spectrum efficiently by applying a divide and conquer strategy to reduce the size of problem space. In this chapter we will describe the experiments designed to evaluate our algorithm. We will then present its performance, as well as discussing other findings observed throughout course of the experimentation.

5.1 Data Sources for Stream Mining

We empirically validate our proposed solution by running a series of experiments using both synthetic and real-world datasets. Synthetic datasets are commonly used in stream mining studies for several reasons (Kirkby, 2007; Bifet, 2010). A synthetic data generator can easily replicate instances and thus costs of storage and transmission are relatively small. Another advantage is it enables us to specify the levels and characteristics of concept drifts. Certain data generator(s) may allow its user to adjust dimensional size of data streams, which is extremely useful for assessing algorithms in terms of scalability.

For the purpose of this research, we use data generators provided by MOA (Massive Online Analysis) stream mining suite. MOA is an open-source framework targeted for mining data streams with concept drift. MOA is conveniently bundled with a collection of data generators frequently used in data stream mining research (Schlimmer & Granger Jr, 1986; Domingos & Hulten, 2000; Street & Kim, 2001; Hulten et al., 2001). In this study, we are using the Rotating Hyperplane and Radial Basis Function (RBF) generators in particular.

Rotating Hyperplane This dataset was firstly introduced by Domingos and Hulten to be used by VFDT (2000), and later used in a comparative study between CVFDT and VFDT (Hulten et al., 2001). The Rotating Hyperplane generator produces data streams containing gradual concept drifts. A hyperplane of dimension d is a set of points x satisfying the condition:

$$\sum_{i=1}^{d} w_i x_i = w_0, \tag{5.1}$$

where x_i is the i-th coordinate of x. Instances for which $\sum_{i=1}^{d} w_i x_i \geq w_0$ are labelled positive and instances for which $\sum_{i=1}^{d} w_i x_i < w_0$ are labelled negative. Rotating hyperplanes are helpful to simulate drifting concepts because position and orientation of the hyperplane can be changed smoothly by simply altering the relative ratio of weights. We fixed the number of drifting attributes to 4, and experimented with drift magnitude 0.01 and 0.05 for this dataset.

RBF This dataset produces concepts that are not so easily approximated by decision tree learners (Bifet, Holmes, Pfahringer, Kirkby, & Gavaldà, 2009), thus introducing a completely different drift characteristics than hyperplane generator. The RBF (Radial Basis Function) generates n number of centroids at random position, each assigned with a random class label and weight. Instances are then generated by randomly picking a centroid where the higher the weight the more likely a centroid will be selected. Attribute values are then displaced by a length that is randomly drawn. Number of centroids is defaulted to 40, unless mentioned otherwise.

Both dataset mentioned above generate numerical attributes. Because our implementation is intended for decision trees in the binary domain, all numerical attributes are firstly normalized and then discretized into two bins prior to being used as stream input.

There are several other data generators bundled with MOA, such as STAGGER (Schlimmer & Granger Jr, 1986) and SEA dataset (Street & Kim, 2001) which we do not use as we need dataset generators that can produce datasets with an arbitrary number of attributes.

5.1.1 Real World Data Sources

Finding publicly-available real world datasets with a large number of instances is not an easy task, especially those containing considerable concept changes. The UC Irvine machine learning repository [1] has one of the largest collections of publicly-available real-world datasets, if not the largest. We consider three datasets for our experiment: Electricity, Forest Covertype and the Flight dataset.

Electricity dataset Is a dataset drawn from the New South Wales (NSW) electricity market in Australia [2]. In Australia, prices of electricity are driven by demand and supply, as opposed to fixed pricing. The price is determined every 5 minutes with classes labelled as UP and DOWN to signify change of prices with respect to average of prices in the last 24 hours. The dataset contains 45312 instances with 7 attributes. The dataset was firstly introduced by Harries and Wales (1999) and is also commonly used in data stream researches (Gama et al., 2004; Sripirakas & Pears, 2014).

[1]http://archive.ics.uci.edu/ml/

[2]http://moa.cms.waikato.ac.nz/datasets/

Forest Covertype dataset This dataset is used to predict cover type of forests from cartographic information (elevation, degrees of slope, etc.) (Blackard & Dean, 1999). The instances are determined from observation samples of size 30 x 30cm per cell acquired from US Forest Service (USFS) Region 2 Resource Information Service (RIS) data. The dataset has 581012 instances with 54 attributes and is also frequently used in studies on stream classification (Oza & Russell, 2001; Gama, Rocha, & Medas, 2003; Bifet, 2010; Sripirakas & Pears, 2014).

Flight dataset This dataset is provided by NASA and is publicly available [3] which is drawn from NASA's FLTz flight simulator. FLTz is a medium-fidelity flight simulator and is useful to develop adaptive flight control and planning emergency maneuvers, among other uses (Chu, Gorinevsky, & Boyd, 2010). The Flight dataset is given as 20 different files, each representing the complete flight data from takeoff to landing. There are four scenarios in total: takeoff, climb, cruise and landing. A data instance is captured every second. Velocity is designated as classification target where it is discretized as UP and DOWN to indicate velocity change with respect to the moving average velocity for the last 10 seconds. All files are merged and irrelevant attributes are excluded to form a single file containing 25043 instances and 30 attributes (Sripirakas, 2015).

5.2 Experiment Settings

DFT process (and memoFT, by extension) discussed extensively in Chapter 4 is a process to transform a decision tree into its equivalent algebraic form. As such, it needs a tree learner algorithm to generate a tree for the DFT process

[3] https://c3.nasa.gov/dashlink/static/media/dataset/FLTz_2.zip

to take place. We select CBDT (Concept-Based Decision Trees) (Hoeglinger et al., 2009) as the base learner to generate trees that will serve as input to both DFT and memoFT.

The framework of our experiment is as shown in Figure 5.1, which is a modified version of the general framework shown in Figure 3.1. CBDT maintains a forest of Hoeffding Trees and each stream instance is routed to every tree being managed. Each tree managed by CBDT has an instance of drift detector (ADWIN) embedded to it. Whenever concept drift is detected,

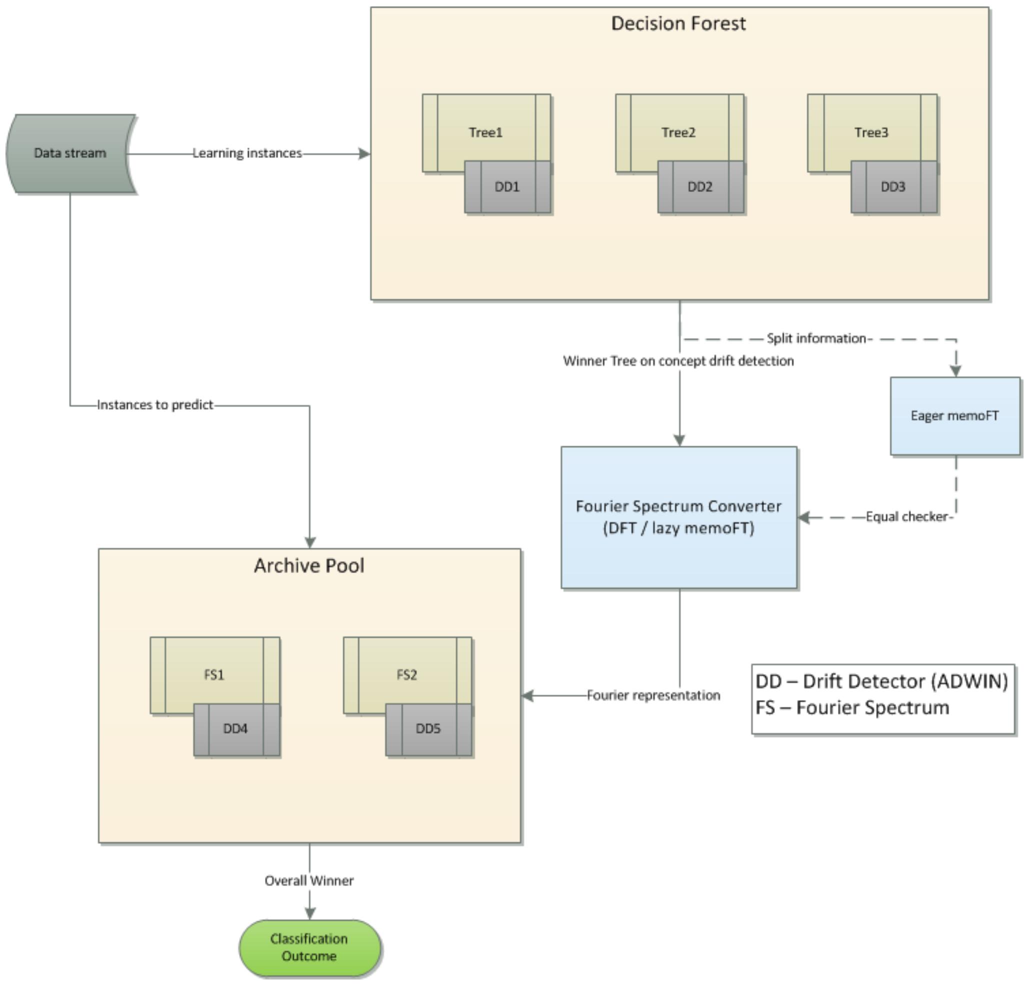


Figure 5.1: Experiment framework to assess memoFT, adapted from Figure 3.1

a model with the highest accuracy is selected as a winner. This winner model may emerge either from the forest in the form of a one of the trees present in the forest or from one of the spectra present in the repository. If the winner emerges from the forest then that tree will then be re-grown from its stump.

As mentioned in Chapter 4, we are interested in the performance of memoFT in both lazy and eager scenarios in comparison to the normal DFT process. Performing eager synchronization of Fourier spectrum for every single tree being managed by CBDT is undesirable because CBDT generates as many trees as the number of attributes in the stream. To reduce the memory usage to a more manageable level, memoFT processes (both lazy and eager) only runs on trees which have been previously declared as winners. Note that this policy reduces computational overhead on the system as a whole and enables experimentation to be less time consuming. It does not have a direct bearing on the research hypothesis that the divide and conquer strategy of applying the DFT to a decision tree improves computational efficiency.

To ensure correctness of the memoFT implementation, Fourier spectra generated by lazy and eager implementations of memoFT are checked for equality in comparison to that resulting from the normal DFT process before registering a new Fourier spectrum entry in the repository.

5.2.1 Parameter Values

Unless mentioned otherwise, all parameters are set at the following default values:

- **Hoeffding Tree** $\delta = 0.00175, \tau = 0.01, n = 100$, where $(1 - \delta)$ the desired probability of correctly selecting a split attribute. τ is the tie tolerance of Hoeffding bound. n is the interval of growth check.

- **ADWIN** drift detection confidence parameter $= 0.01$

- **Decision Forest** maximum number of managed trees = 40, maximum number of nodes in forest = 5000.
- **Repository** maximum number of Fourier spectra = 50, accuracy tie threshold = 0.01 which is the minimum difference in accuracy between a tree with highest accuracy in decision tree forest and spectrum in repository with the highest accuracy for DFT to be activated on most accurate tree in forest.
- **DFT process (memoFT included)** order cutoff = 5, winner count threshold = 2, which is the minimum number of times a tree in the decision tree forest must be declared as a winner before memoFT is used to evaluate the Fourier coefficients of the tree (applies to memoFT only).

5.2.2 Controlled Variables

In addition to the parameters mentioned above, we recognized several factors affecting the performance of memoFT directly and indirectly. The size and dimensionality of a tree directly affects the time needed to convert the tree to its Fourier spectrum representation. This factor is hard to control because structure of a decision tree is highly dependent on the characteristics of the examples used to train it. Indirectly, this variable can be controlled by manipulating the dimensionality of data streams. This is where synthetic datasets may prove useful to assess memoFT, as reducing the dimensionality of data streams in real world data risks exclusion of potentially important features. Hence in our experiments, we only control the dimension of stream for synthetic datasets, with Forest Covertype as exception, where attributes are filtered using WEKA[4] feature selection as ranked by information gain.

[4]http://www.cs.waikato.ac.nz/ml/weka/

Another factor directly affecting time performance of our solution is the value of order cutoff as shown by Equation 3.13 in Chapter 4 where greater order cutoff will increase the cost of calculating coefficients.

The controlled variable values in our experiments are as follows:

- Dimensionality of streams
  - RBF and Rotating Hyperplane - 10, 20 (default), 40
  - Forest Covertype - 30, 40 (default)
- Order cutoff: 2,3,4,5 (default)

5.2.3 System Hardware / Software Setup

The system used in our experiment is developed in C# running on .NET Framework 4.5 environment. All experiments done in this study were conducted on a system running Windows 7 Service Pack 1 operating system with Intel i5 CPU and 8GB RAM. To get more accurate and fair performance measurements, each experiment run was repeated 10 times with .NET runtime memory cleared in between runs.

5.3 Experiment Results

We focus our study on time profiles of three different DFT processes: normal DFT, lazy memoFT and eager memoFT. In this section we will present the general time performance of these processes. We also study the relative effectiveness of memoFT with respect to dimensionality of streams and order cutoff.

5.3.1 Runtime Comparison

We ran experiments on all datasets with default parameter values and measured the runtime between the three DFT processes, which are compiled in Appendix C. The results are then plotted as shown in Figure 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7. For the RBF dataset, we have to reduce the winner count threshold to 1 because there is not detections of concept drift for analysis. We observe several interesting patterns from the runtime comparison with each of the datasets.

- As expected, the runtime of memoFT evaluating trees lazily is faster than normal DFT process. The degree of improvements varies across datasets, where runtime is improved between 40-64% and 46-59% on synthetic and real world datasets respectively.

- Another trend observed is the average runtime of eager memoFT per split is much smaller than the average runtime of lazy memoFT, which also falls within our expectation. However, when these incremental improvements in runtime are aggregated on a per drift basis, eager memoFT performs even worse than normal DFT. A prime example of this is shown in Figure 5.8 (note that eager memoFT is cumulative in this Figure).

- There is a couple of contrasting improvement results from RBF and Rotating Hyperplane datasets. While lazy memoFT works well on RBF, its effectiveness appears to be greatly reduced when processing several trees trained on the Rotating Hyperplane dataset. For example, there is an instance where runtime improvement is merely 18% on Rotating Hyperplane, whereas runtime improvement can be as high as 90% on RBF. Upon closer inspection, we find that the particular tree on Rotating Hyperplane has a high amount of reappearing split attributes. It has

12 split attribute instances, out of which only 5 are distinct. Such tree structure is possible because of the mechanism of Rotating Hyperplane: gradually shifting the weights of drifting attributes. This phenomenon is consistent with *Case 2* of *Theorem 2* from Chapter 4 which implies that reoccurring split attributes on different sub-trees reduce the effectiveness of our solution.

Another interesting finding emerges from runtime measurements of the DFT processes on the Electricity dataset, as shown in Figure 5.9. On this particular dataset, both eager and lazy memoFT are faster than normal DFT. We discovered that the Electricity dataset produces many small trees; most of the trees generated only contain either 2 or 3 split attributes. It seems that when the dimensionality of streams is low, the cost of merging Fourier coefficients is negligible, such that incrementally building Fourier coefficients is faster than computing the DFT normally. The rest of figures showing memory profile on drift detection are shown in Appendix D.

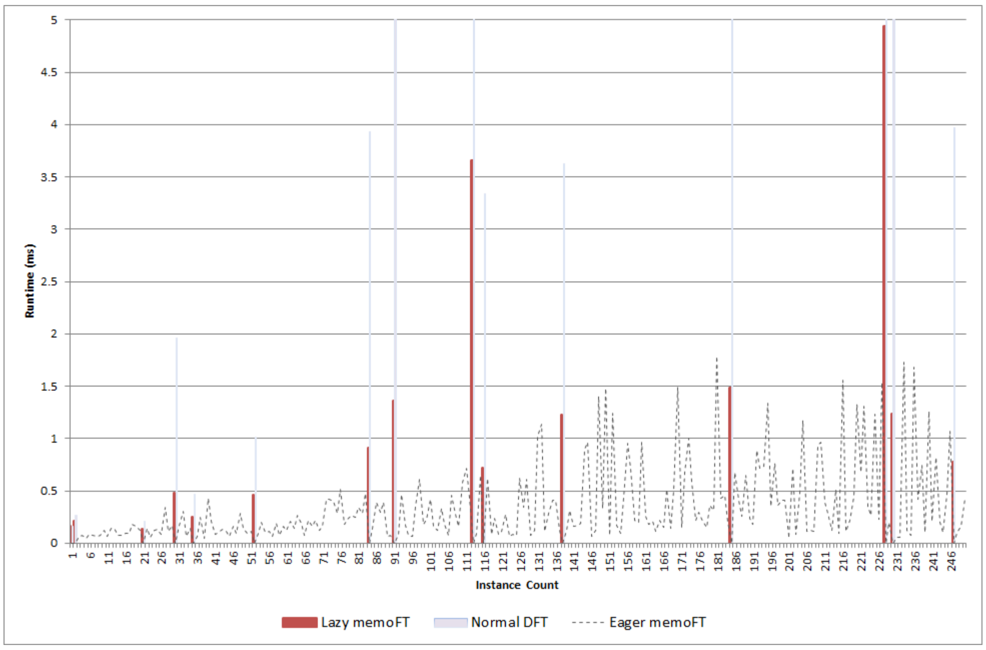


Figure 5.2: Runtime profile of DFT processes on RBF dataset (20 attributes, minimum winner count = 1)

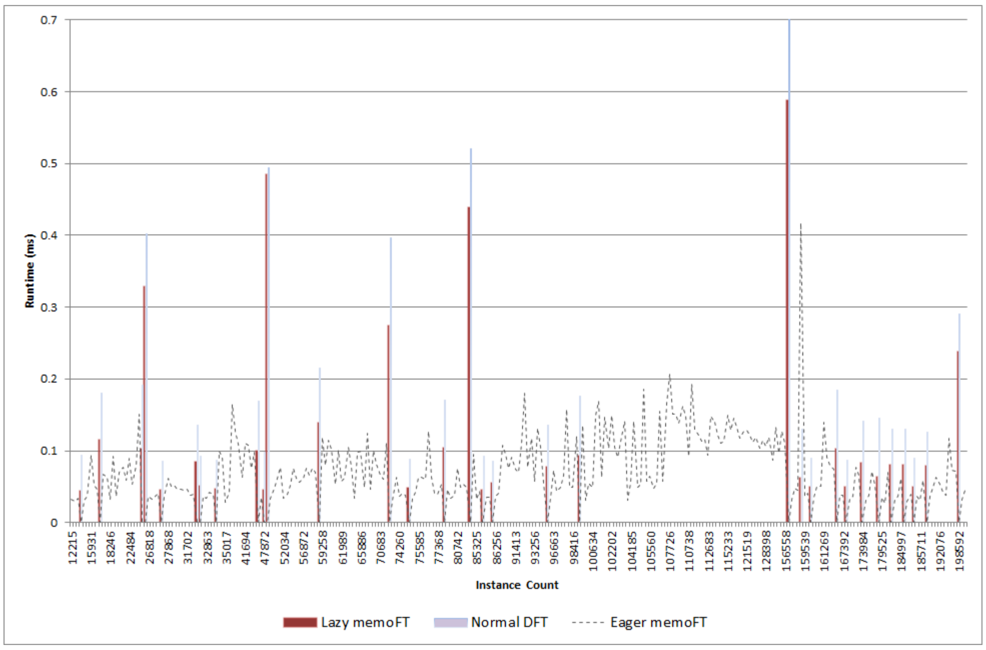


Figure 5.3: Runtime profile of DFT processes on Rotating Hyperplane dataset (20 attributes, drift magnitude = 0.01)

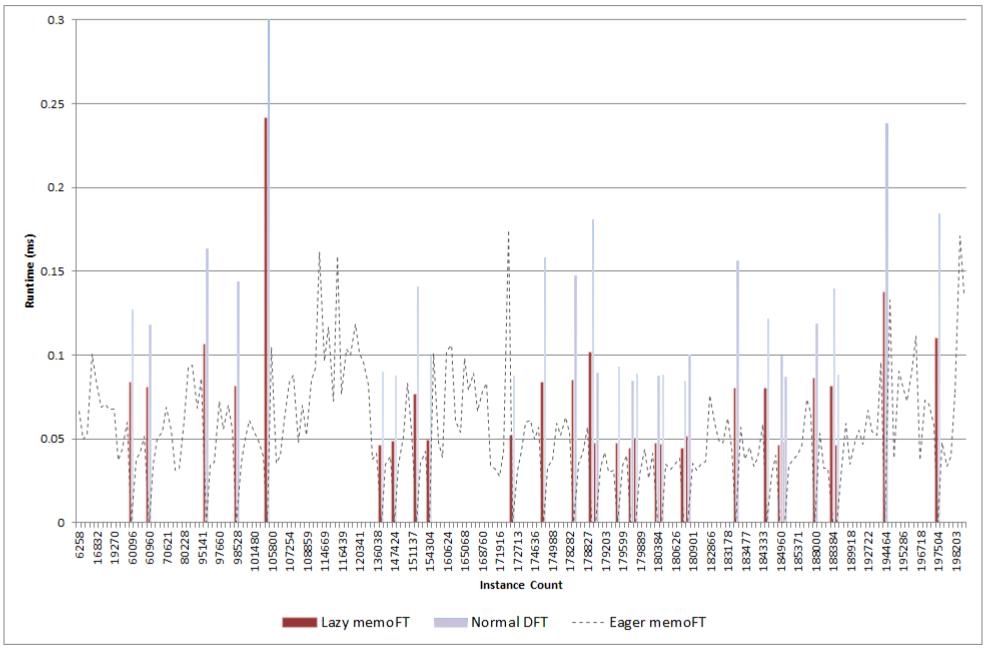


Figure 5.4: Runtime profile of DFT processes on Rotating Hyperplane dataset (20 attributes, drift magnitude = 0.05)

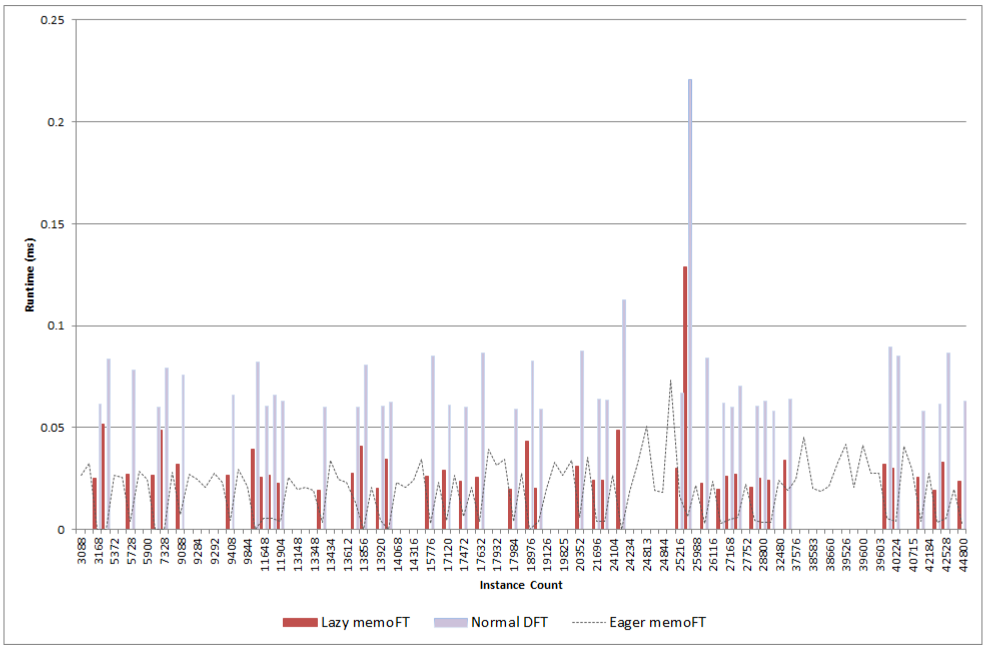


Figure 5.5: Runtime profile of DFT processes on Electricity dataset

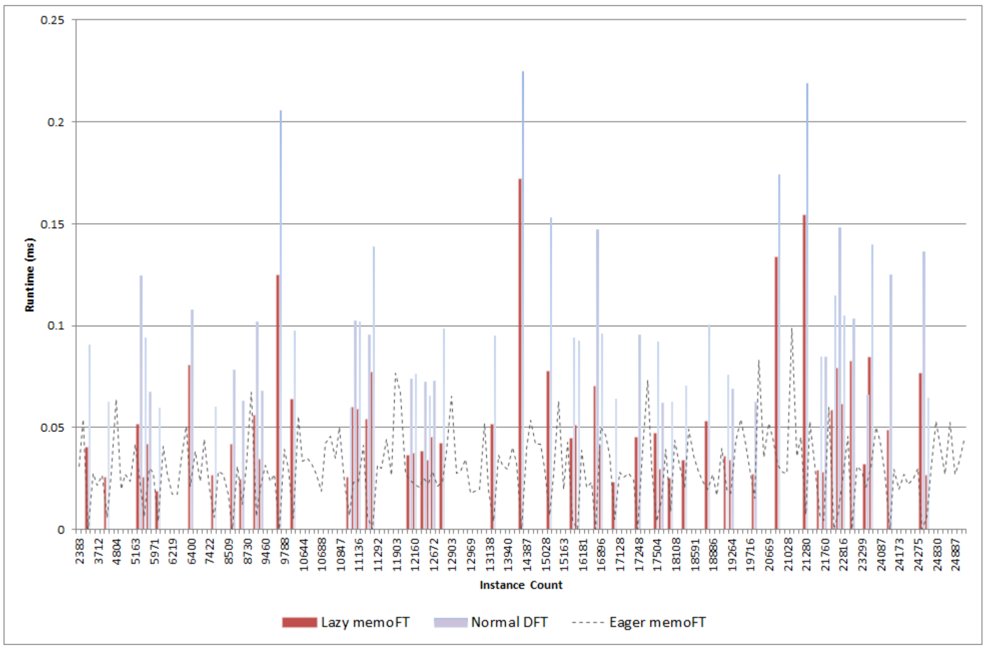


Figure 5.6: Runtime profile of DFT processes on Flight dataset

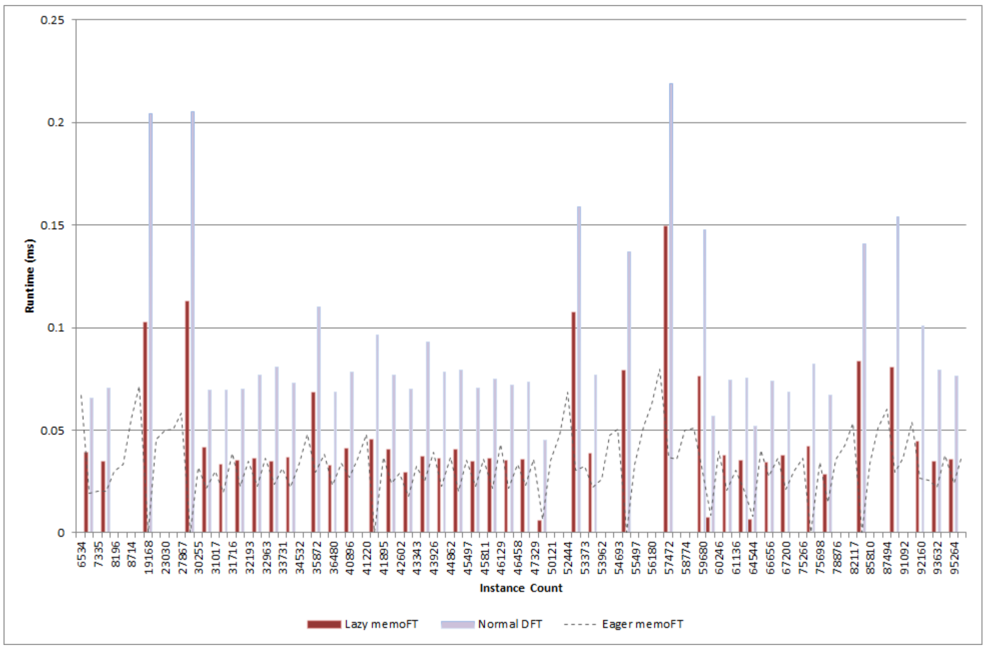


Figure 5.7: Runtime profile of DFT processes on Forest Covertype dataset (40 attributes)

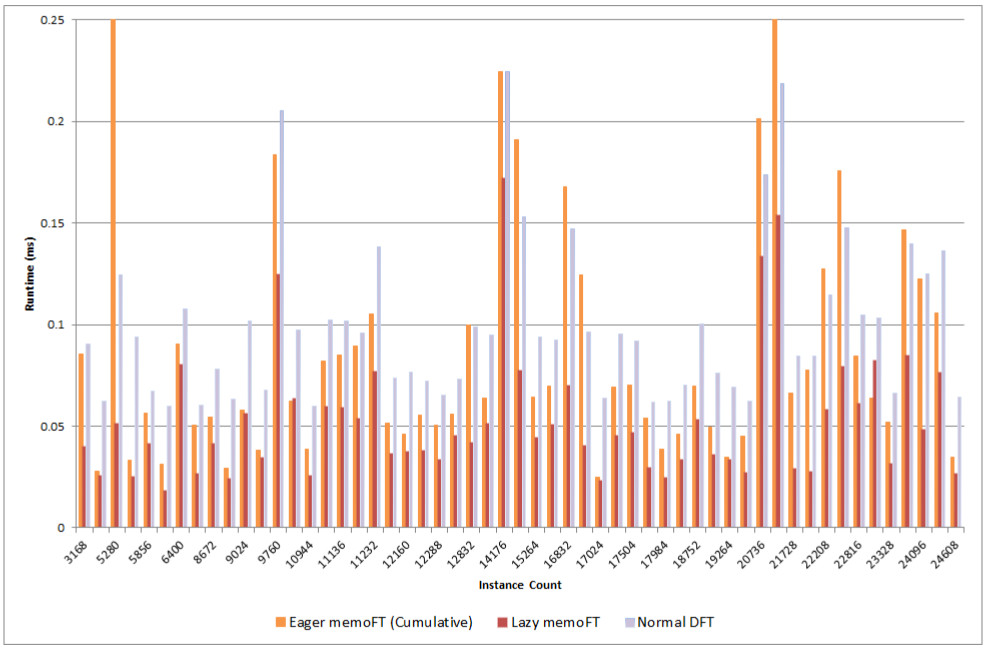


Figure 5.8: Runtime of DFT processes on Flight dataset measured on drift detection

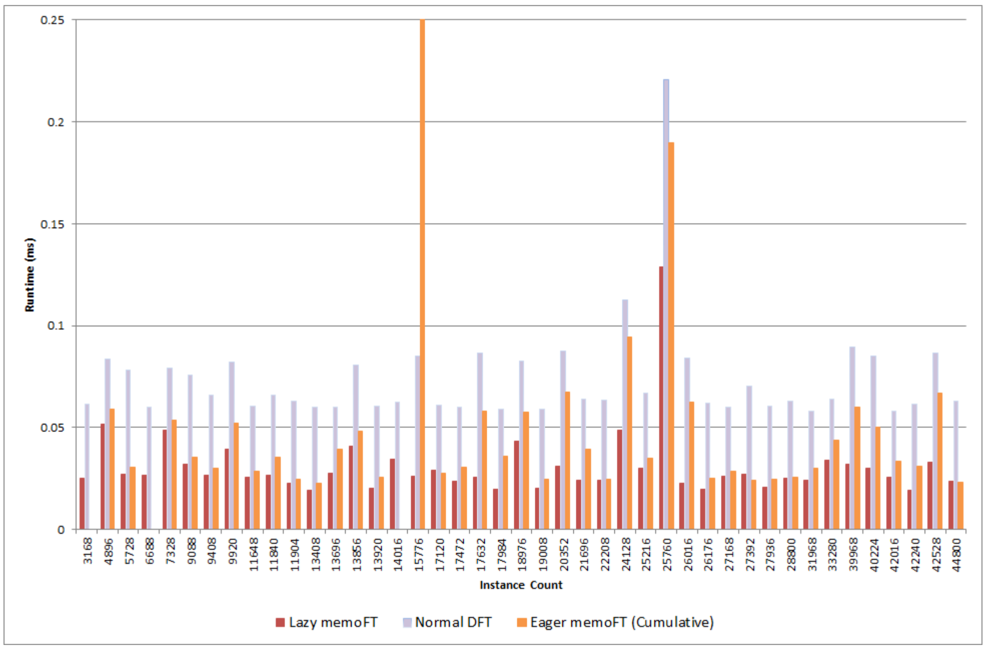


Figure 5.9: Runtime of DFT processes on Electricity dataset measured on drift detection

5.3.1.1 Effects of Dimensionality to Runtime Performance

We then run more experiments to inspect the effects of dimensionality of streams and order cutoff to runtime performance of the DFT processes. We specifically focused our attention on comparing lazy memoFT and normal DFT in these experiments. Because order cutoff also affects accuracy in addition to the cost of DFT calculation, we also measure accuracy for these experiments to show the potential trade-offs between speed and accuracy. We witness similar patterns of general runtime performance reoccurring as mentioned above. Additionally we also notice patterns when dimensionality and order cutoff are altered.

For easier referencing, from here onwards we relabel the datasets using a naming scheme listed in Table 5.1 below, where wildcard character $*$ denotes the number of attributes in the data stream. The result of these experiments are compiled in Appendix A and Appendix D. We then plotted average accuracy and runtime improvement for each dataset as shown in Figure 5.10, 5.11, 5.12, 5.13, 5.14 and 5.15. Runtime improvement shown in the tables is derived from the runtimes of lazy memoFT and normal DFT processes, t_0 and t_1 respectively, using the following formula:

$$\text{Runtime improvement} = \frac{(t_1 - t_0)}{t_1}.100\%$$

We notice a couple of patterns emerging:

- As the dimension of data stream is increased from 10 to 40 (RBF and Rotating Hyperplane), we expected lazy memoFT to scale better in terms of runtime. This is strongly shown in the RBF dataset where improvement of runtime is boosted significantly e.g. for order cutoff $= 4$ runtime improvement grows from 44% to 64% to 73% for 10, 20 and 40 attributes respectively. In contrast, runtime improvement on Rotating Hyperplane datasets only increased by a smaller amount in comparison

Dataset Name	Description
RBF-*	RBF with x attributes
RH1-*	Rotating Hyperplane, drift magnitude = 0.01
RH2-*	Rotating Hyperplane, drift magnitude = 0.05
Covertype30	Forest covertype, 30 most significant attributes
Covertype40	Forest covertype, 40 most significant attributes

Table 5.1: Relabelled dataset names and description

e.g. on RH1 dataset with corresponding order cutoff and dimension sizes, runtime improvements are 43%, 40% and 46%.

- It is interesting to note that at times runtime improvement of lazy memoFT on higher dimensionality can be lower than that of lower dimensionality. We suspect that this is caused by several factors. The first factor has been discussed previously, which is the nature of Rotating Hyperplane to generate trees with multiple attributes reappearing in trees. The second factor is the cost of the effort to combine Fourier coefficients of sub-trees (discussed in Section 4.3). The fact that runtime improvement first increases and then drops suggests two things: 1) that the merging cost is negligible on lower dimensionality. 2) merging cost grows as dimensionality increased.

The cost to merge Fourier coefficients of sub-trees is affected by the structure of the tree i.e. the total number of split and distinct attributes in the tree, how many sub-trees are there in the tree etc. It might be possible to predict the efficacy of memoFT using meta-information contained within the tree. This can be useful to build a decision making module for when to use memoFT. A study on this aspect is identified as part of future work arising from the present research.

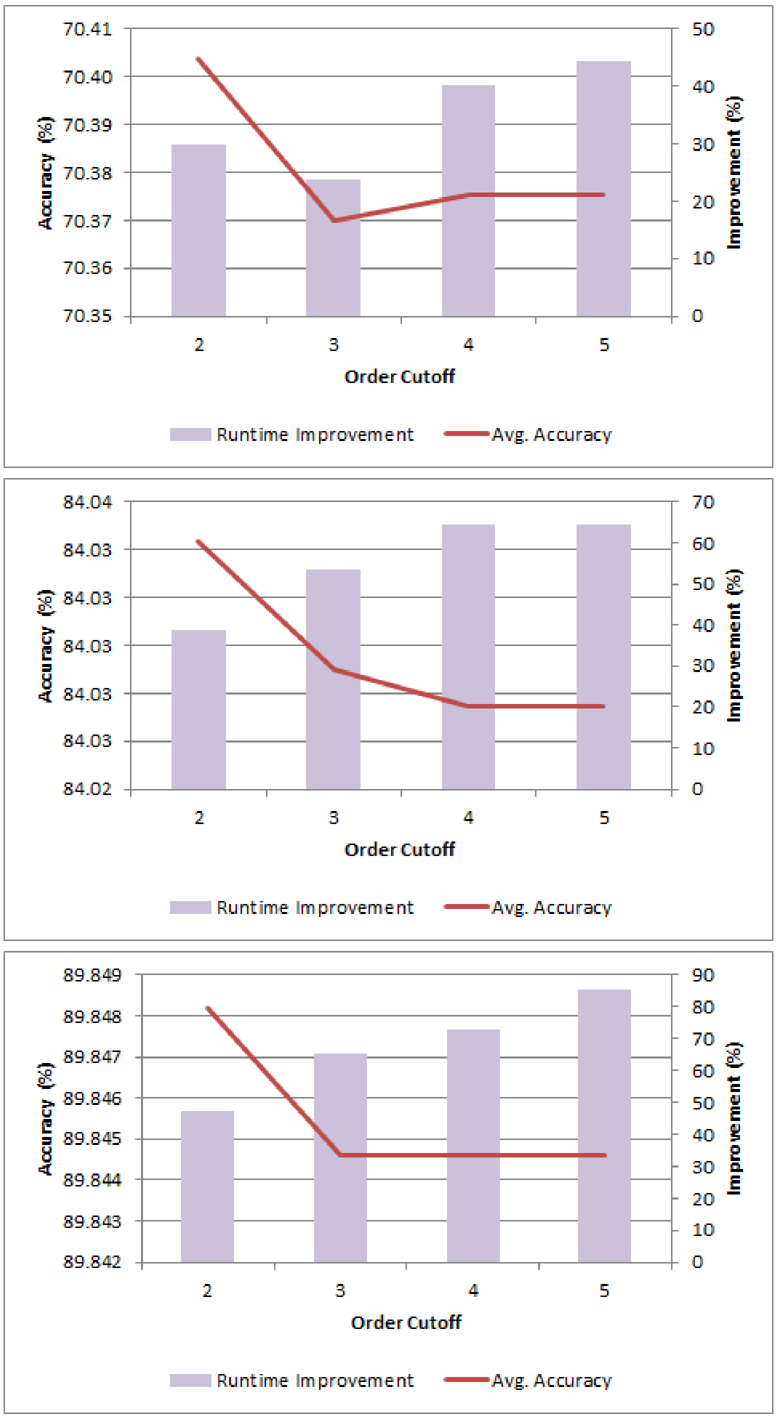


Figure 5.10: Average Accuracy and Runtime Improvement for various order cutoffs on RBF-10 (Top), RBF-20 (Middle) and RBF-40 (Bottom)

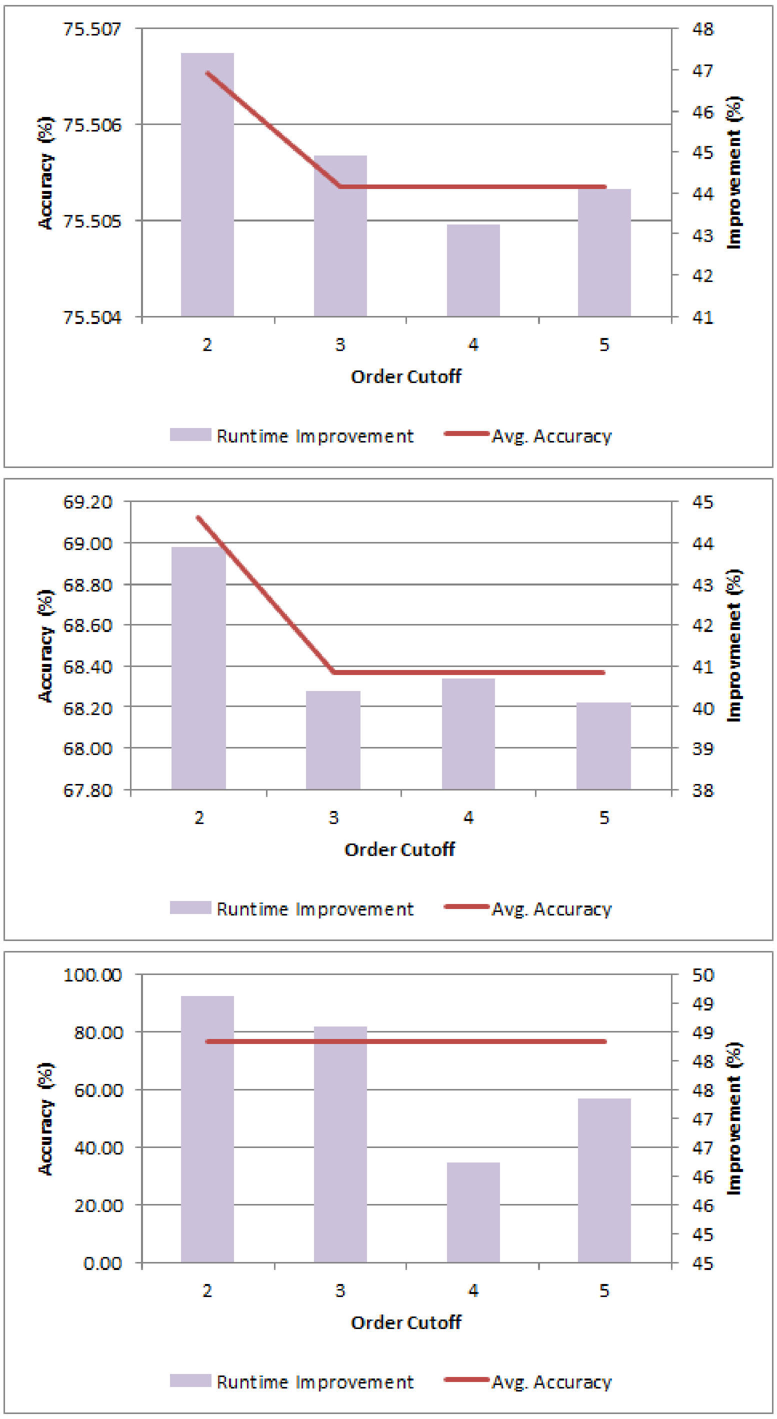


Figure 5.11: Average Accuracy and Runtime Improvement for various order cutoffs on RH1-10 (Top), RH1-20 (Middle) and RH1-40 (Bottom) datasets

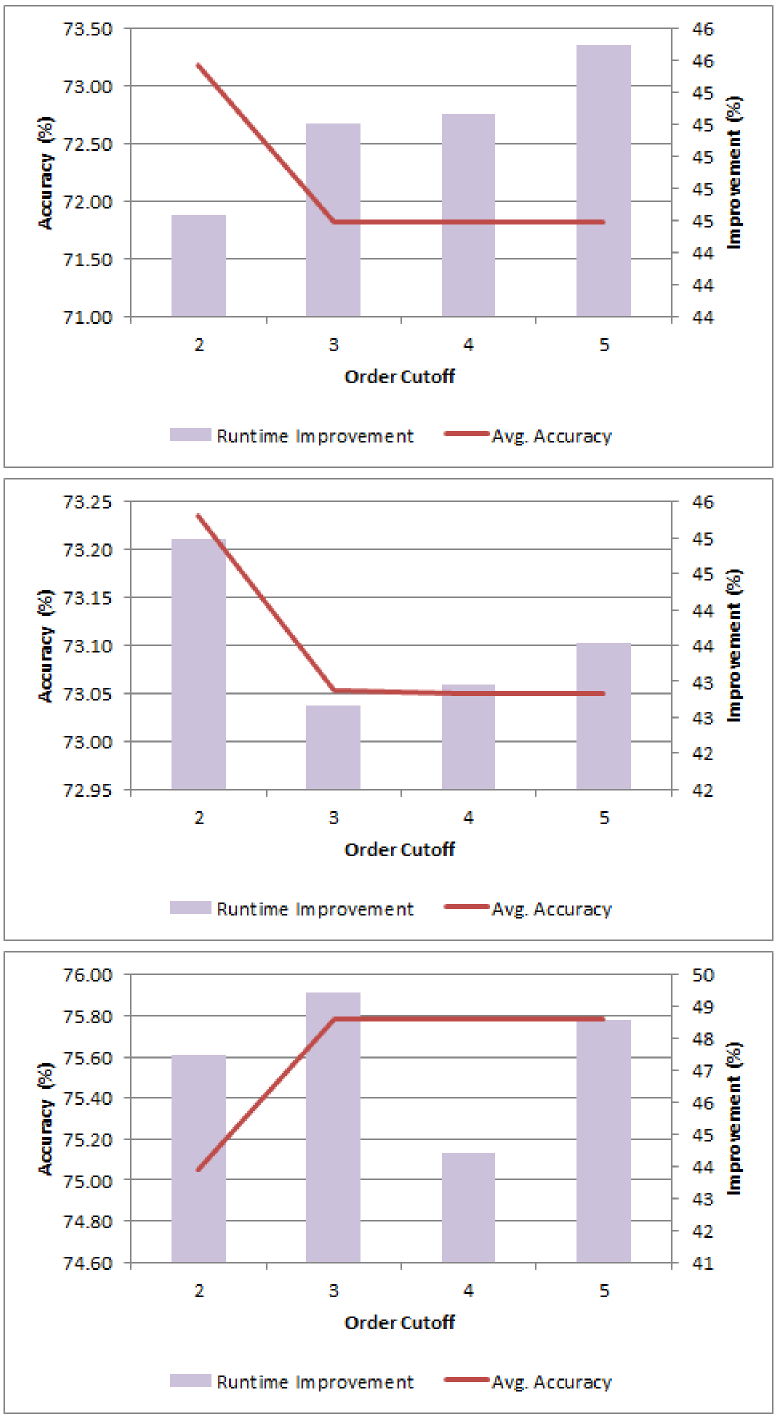


Figure 5.12: Average Accuracy and Runtime Improvement for various order cutoffs on RH2-10 (Top), RH2-20 (Middle) and RH2-40 (Bottom) datasets

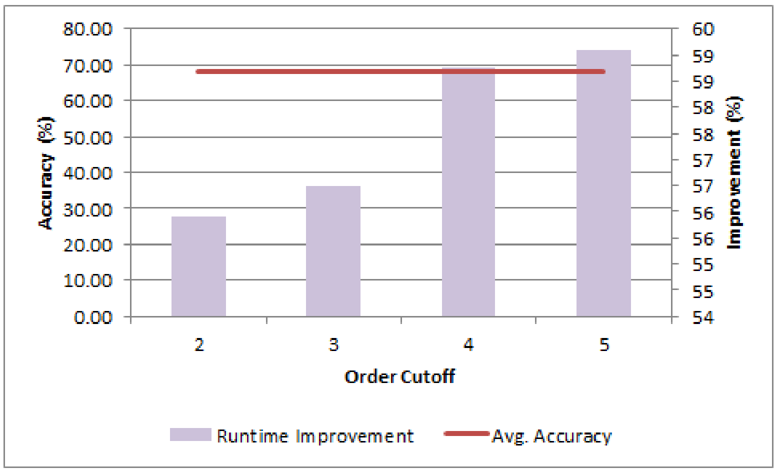


Figure 5.13: Average Accuracy and Runtime Improvement for various order cutoffs on Electricity dataset

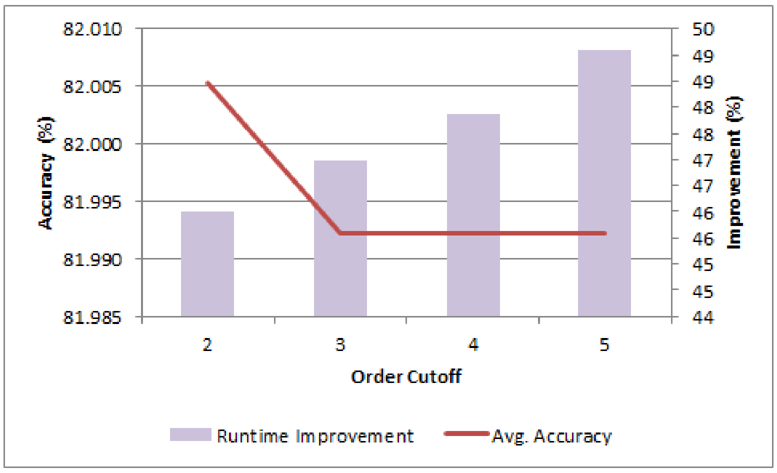
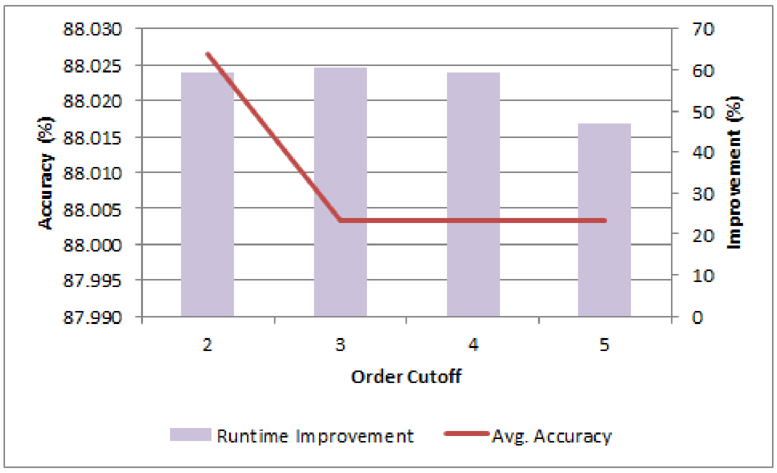


Figure 5.14: Average Accuracy and Runtime Improvement for various order cutoffs on Flight dataset



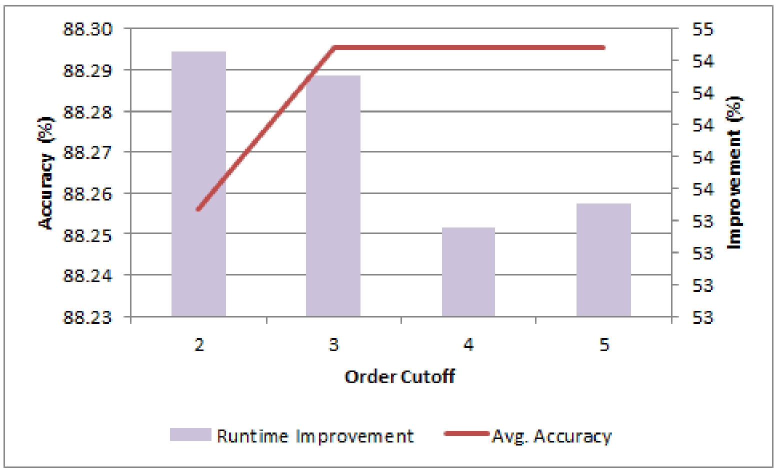


Figure 5.15: Average Accuracy and Runtime Improvement for various order cutoffs on Covertype30 (Top) and Covertype40 (Bottom) datasets

5.3.1.2 Effects of Order Cutoff to Runtime and Accuracy Performance

We also noticed an additional pattern emerging when order cutoff is increased from 2 to 5:

- memoFT process performs much better than normal DFT as order cutoff is increased. This pattern occurs across many datasets: RBF (Figure 5.10), RH2-10 (Figure 5.12), Electricity (Figure 5.13)and Flight (Figure 5.14) datasets. This boost of performance, however, is relatively small compared to the increase of performance when dimension size is increased, which is within our expectations. Recall that the problem space of DFT grows exponentially with respect to dimension size of decision tree; while the growth of significant coefficients is combinatorial by order (refer Equation 3.13).
- We also found another interesting trend, that most experiments running on higher order cutoff yielded lower average accuracy. The only exception was experiments running on RH2-40 and Covertype40 datasets where accuracy increased as order cutoff was raised. This seems to be in opposition to properties of decision trees in Fourier domain that has been discussed in Section 3.8.1. However, it is not actually so. When a tree is converted in DFT form, order cutoff determines how much *information* is preserved from the original tree. It is very likely that setting order cutoff to a certain value might actually filter out *noise* information from the original tree, thus enabling to generalize better to future data. In essence, this is very similar to what tree pruning does, improving generality by collapsing leaf nodes to avoid overfitting.

In practice, it is very hard to determine the value of order cutoff that is optimal in general. We suggest to firstly experiment with a small sized sample starting from order cutoff 1 and incrementally increase the cutoff value until desired performance is achieved whenever a DFT process is involved. Another factor to consider when tuning order cutoff is the dimensionality of trees generated as it learns from the instances. This is because the number of coefficients to be calculated grows in combinatorial fashion with respect to dimensionality. Thus, the increase of order cutoff can be counter-productive in terms of processing speed.

5.3.2 Memory Profile

memoFT is an algorithm that improves runtime performance by making use of memory storage optimisation. Hence it is important to assess it from the perspective of memory consumption. During experiment runs, we also monitor the memory storage maintained by memoFT on each dataset. Because lazy memoFT and eager memoFT ultimately generates the same Fourier coefficient sets, it is quite likely that both processes generate a near identical sized memory structure, if not identical. The average of measured memory consumption of memoFT per dataset are then tabulated as shown by Table B.1, B.2 and B.3.

Once again, we observe that the memory profiles of memoFT on RBF and Rotating Hyperplane datasets are located at opposite extremes. In the previous section, RBF datasets benefit greatly from memoFT while the improvements enjoyed by Rotating Hyperplane datasets are much lower in comparison. From memory perspective, memoFT uses up much more memory storage on RBF as opposed to Rotating Hyperplane datasets.

Quite curiously, memory usage on real-world datasets are closer to that of Rotating Hyperplane. In Rotating Hyperplane datasets, decision trees generated tend to be balanced but containing "high" number of reappearing split attributes, thus lowering the actual dimensionality of the tree. Trees generated by real-world datasets are more likely to contain not as much reoccurring split attributes for a given tree size. However, they are also more likely to be less balanced, which translates to less sub-trees to be maintained by memoFT process.

We also observe the effects of dimensionality and order cutoff to memory consumption of memoFT:

- As dimensionality increases, memoFT consumes more memory. This is very much expected because an increase in dimensionality means more split attributes in the trees, which increases the likelihood of more sub-trees appearing in the tree.
- Consumption of memory of memoFT also increases when order cutoff grows. Order cutoff strongly affects how many coefficients to be calculated (and thus, stored) when memoFT performs DFT on sub-trees. The increase of memory consumption caused by order cutoff is smaller than the increase caused by dimensionality due to the same reason mentioned in previous section.

5.3.3 Recurring Concepts in Real World

We propose that future work on DFT use a shared memory structure to exploit commonality between Fourier spectra that occurs as a result of concept recurrence. This will accelerate the DFT application to different trees that emerge as winners over time by re-using already generated spectra that correspond to sub-trees that manifested at previous points in the stream progression.

In many real world scenarios, concepts may reoccur frequently (Sripirakas & Pears, 2014). Take the Flight dataset for example; it is a collection of flight information taken from 20 flight runs starting from takeoff to landing. In total there are only 7 concepts being repeated 20 times over. However, a concept does not always reoccur in exact forms; they are bound to reoccur with differences due to changes in distribution of information in streams.

As discussed in the experimental settings, our experiment framework involves a forest of Hoeffding trees that is grown in parallel as stream instances are processed. When concept change is detected, a snapshot of the tree with highest accuracy in the forest will be captured and the tree will be regrown from stump. This mechanism allows us to capture concepts appearing in streams (Sripirakas & Pears, 2014).

As we run the experiments, we notice that concepts do indeed reoccur in real-world datasets. In particular, Flight dataset has a very high frequency of concept reoccurrences, which is well expected. An example of reoccurrences are as shown in Figure 5.16 below.

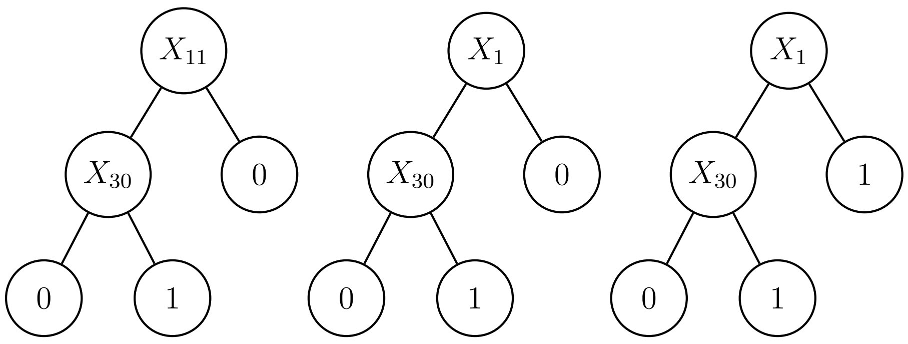


Figure 5.16: Examples of frequently reoccurring concepts in Flight dataset

Concept not only reoccur in "small" tree sizes (like in previous example), but also reoccur in trees with "higher" dimension size. We notice this phenomenon especially In *Covertype*30 dataset, where an example of reoccurring concept in higher dimensionality is shown in Figures 5.17 and 5.18. Both concepts share very similar structure with three identical sub-trees.

Based on these findings, we predict that it would be highly beneficial to implement a caching system of coefficients that is shareable between trees in the forest, as the Fourier spectrum is highly dependent on structure of the origin tree. This is beyond the scope of our research as we focused on the divide and conquer approach of DFT application on decision trees in this study. The potential benefit of exploiting reoccurring concepts is large; hence our future works needs to be focused in this area.

Summary

In this chapter, we presented our experimental framework which allowed us to assess our proposed solution on datasets, both synthetics and real-world, with varying characteristics of stream and distribution of information. We have shown that aggressively decomposing decision trees into sub-trees ultimately leads to better runtime performance than normal DFT process across all datasets used. This approach is also proven to scale better as dimensionality and order is increased compared to normal DFT.

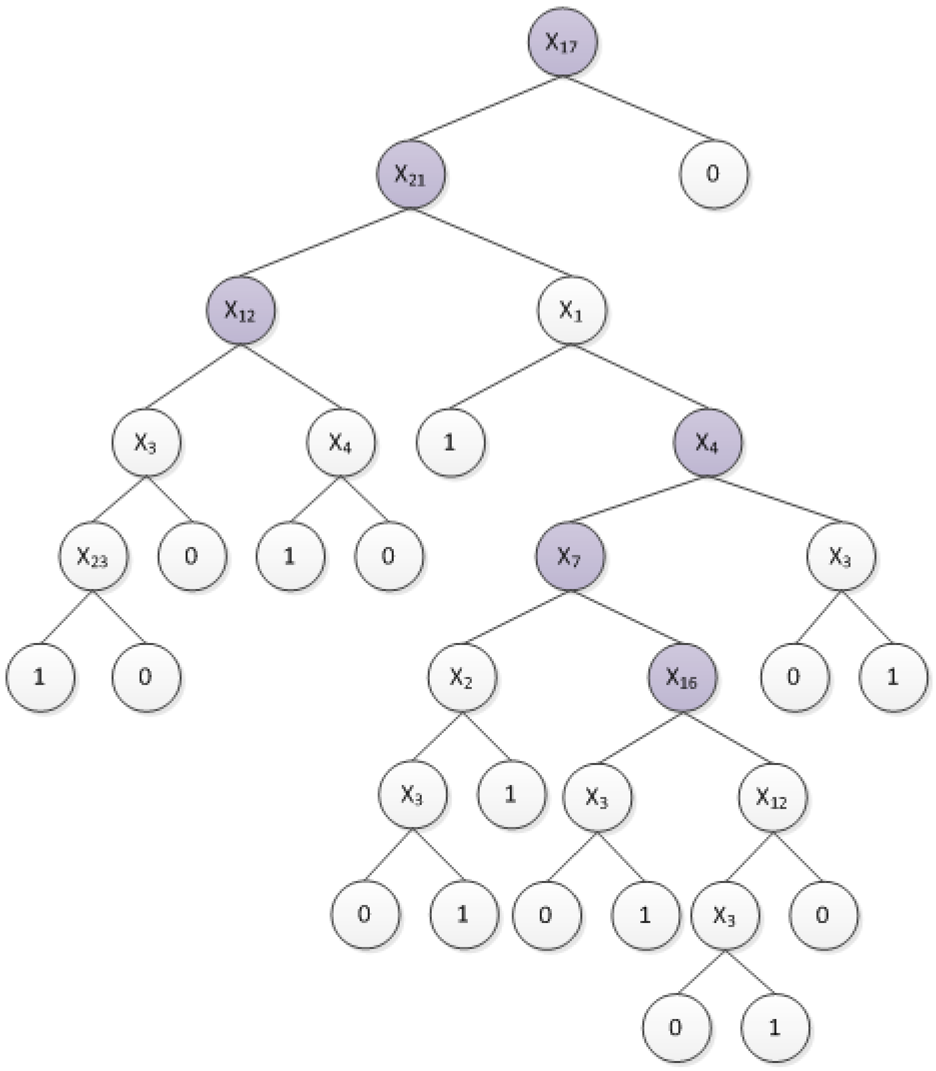


Figure 5.17: Concept captured on Covertype30 dataset

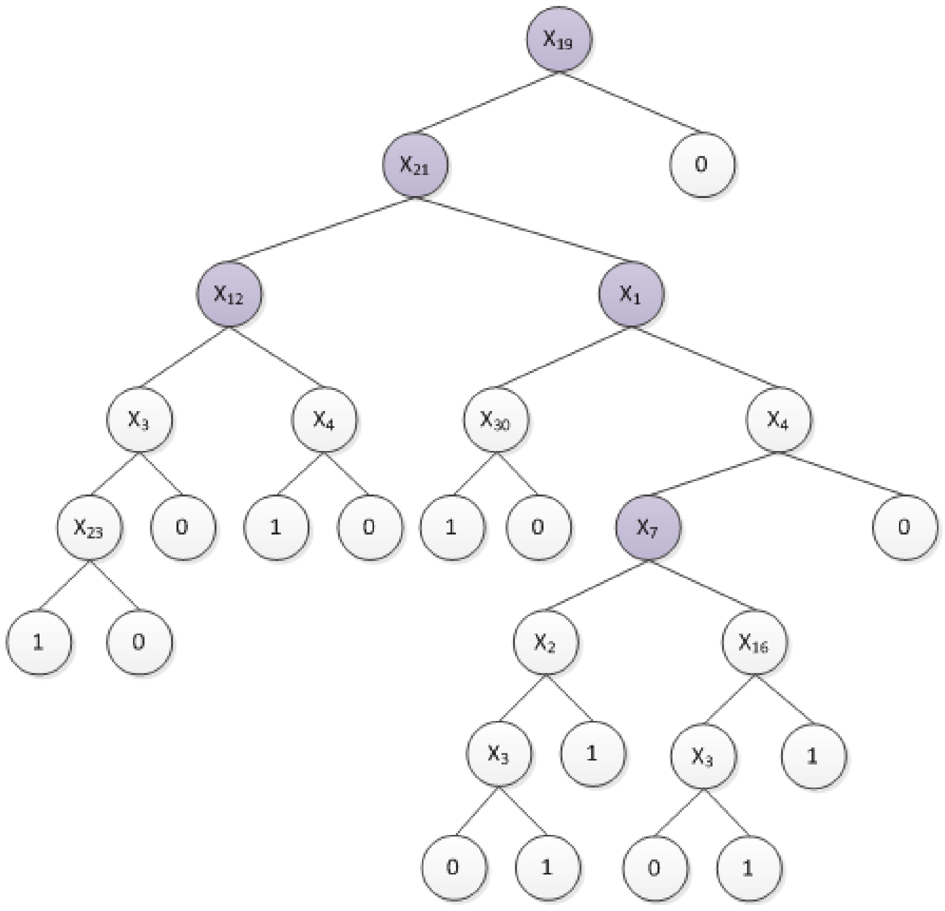


Figure 5.18: Another concept captured on Covertype30 dataset

Chapter 6

Conclusion and Future works

6.1 Research Achievements

In previous chapters, We framed an important issue raised by application of the DFT to support mining recurrent concepts on data streams. Representing decision trees as Fourier spectra is highly beneficial to recurrent concept mining for numerous reasons. First, it reduces the memory requirement of storing a tree that has captured a concept while preserving its accuracy. Second, Fourier spectra classifiers may generalize better than their uncompressed counterparts, resulting in higher accuracy. However, the DFT computation is an expensive operation, where the number of coefficients to compute grows exponentially. Unfortunately, the well-known FFT algorithm cannot be used to address this challenge for the reasons outlined in Chapter 3. This creates a challenge we have to deal with because processing speed is extremely important in stream mining, whereby data instances arrive at high speed.

Several solutions were proposed to mitigate the exponential feature space problem (Kargupta & Park, 2004; Sripirakas, 2015), however none of the solutions mentioned attempted to reduce the size of underlying feature space. We focused our study to address this issue and presented a solution to reduce the size of the problem space.

We presented a novel approach to compute the DFT of a decision tree, called memoFT, which employs a divide and conquer strategy to decompose a decision tree into sub-trees and compute their Fourier spectra in their own localized feature space. The implementation algorithm is similar in spirit to the FFT algorithm. The difference is that the FFT breaks down the problem space directly by exploiting the properties of the univariate transform.

Our experimental study of memoFT shows that memoFT in lazy evaluation mode is superior to normal computation of DFT on all tested datasets with average improvement of about 40-64% on synthetic datasets and 46-59% on real world datasets. The extent of runtime improvement of memoFT is highly dependent on dataset type, whereby the runtime improvement can be as high as 85% on the RBF dataset and as low as 40% on the Rotating Hyperplane dataset.

Another contribution of this research is the theoretical formulation that the Fourier spectrum of a decision tree can be inferred from the Fourier spectrum of another tree. This is significant as the cost of inferring the Fourier spectrum of a decision tree from previously-computed spectra can be lower than computing it from scratch. As described in Chapter 4, this theorem has several implications in recurrent concept mining: 1) it allows us to eagerly synchronize decision trees to their respective Fourier spectra and 2) it opens up the possibility of employing a smart caching mechanism where a spectrum that has captured a concept can be reused and adapted should the concept reoccurs with small differences in the stream.

In this study, we inspected the former implication of this theorem and incorporated it into the core module of memoFT. MemoFT incrementally builds Fourier coefficients of sub-trees, allowing us to track in eager mode and synchronize the Fourier spectrum of a growing tree. From our empirical study, we observed that synchronizing the Fourier spectrum in eager mode performs worse in general than computing Fourier spectrum lazily with our proposed divide and conquer strategy. However, eager evaluation has the lowest average runtime per evaluation run, making it suitable for real-time systems where bursts of CPU usage are undesirable.

6.2 Limitations

We identified several limitations of our research which are listed below:

- The proposed solution is highly specific to improving the computation efficiency of converting decision trees into Fourier spectra. The divide and conquer strategy proposed can only run on a decision tree structure. This necessitates the usage of decision tree classifier. While decision trees are amongst the best classifiers to use in a data stream environment it is desirable to extend the scope of application of the DFT to other types of classifiers.

- Streams with very high dimensionality still pose a serious challenge. As discussed in Chapter 3, the number of coefficients to compute grows exponentially on dimensionality. Our solution attempts to reduce this by computing local Fourier spectra of sub-trees and merge them. Streams with higher dimensionality increases the chance of a higher number of sub-trees as well as the size of coefficient table of each sub-tree. Streams with very high dimensionality would have impact not only on runtime performance, but also on memory consumption because our algorithm is a memory-based optimization. We suggest limiting the growth of decision trees to a certain depth to reduce the severity of this problem when the DFT is applied on very high dimensionality.

- DFT application to decision tree is limited to discrete features. It is not optimized to handle data streams containing numerical (or continuous) features. Continuous valued features can be discretized as part of pre-process. However, such procedure may result in drastic increase of dimensionality, which leads to the same situation described in the previous point.

- memoFT would not be suitable for devices with restricted memory resource. As discussed in Chapter 4, memoFT used memory optimization technique. The algorithm requires additional memory requirements as a tradeoff to gain improvement in runtime performance.

- memoFT is designed for a general-purpose evaluation by incrementally building Fourier coefficient tables in both eager and lazy mode. The algorithm can be optimized in lazy evaluation mode because it processes coefficient adjustments to sub-trees on a per schema basis. If a sub-tree at a lower level contains multiple schemas, the algorithm will perform multiple memory lookup to apply coefficient adjustments. A more efficient approach in lazy evaluation mode is to collect schemas contained in that sub-tree and apply coefficient adjustment in a single lookup operation.

- The lack of an average case computational time guarantee. The FFT algorithm directly divides the problem space into two equal parts and recursively process each part, resulting in a reduction of time complexity from $O(n^2)$ to $O(n \log n)$ (Brigham, 1988). However, our solution indirectly reduces the problem space by dividing a decision tree into sub-trees. This recursive process does not necessarily decompose a decision tree into equal-sized sub-trees, which makes it very hard to derive time performance guarantee.

- Our experiment framework has a dependency on concept change detection. The performance of change detection affects the overall system accuracy. When change detection mechanism falsely identifies concept change, the error will propagate throughout the entire system. This creates a need to carefully select the best performing change detector in terms of false negative and false positive rates.

6.3 Future Work

The time frame given to complete a Master's Thesis restricts the amount of work that can be done in this research. In this section, we will discuss several possible future work in this field.

- Extending the work done in this thesis beyond the binary domain. Our implementation algorithm is designed to compute Fourier spectra in the binary feature space as a proof of concept to test the validity of our theorems. To extend it to work in n-ary feature domain would be a logical step since most real world datasets involves non-binary features.

- There are cases when computing local Fourier spectra can be more expensive than the normal DFT computation under a specific tree structure. An exact decision making mechanism to switch between the two types of computation can be expensive as it may require multiple scans of the decision tree. A plausible alternative is to use a meta-learner that is trained to predict the expected conversion runtime and chooses the best form of computation accordingly.

- Exploit commonality between Fourier spectra that occur as a result of concept reoccurrence. In this research, we have yet to unlock the full potential of Theorem 3 which enables us to infer Fourier spectrum from an existing one (see Section 4.3). Recall that in recurrent concept mining, concept can reoccur in slightly different form instead of exact form, which manifested as winner trees sharing similar structures. Using this Theorem, we can reduce the memory space required to store different versions of reoccurring concepts in the repository by keeping only one version of the concept and making adjustments as needed. Another benefit offered by this approach is the huge boost in runtime to infer Fourier spectra of reoccurring concepts instead of computing them from scratch.

- Another possible future direction is to extend the setting of our research into semi-supervised learning domain. In real world situation, incoming instances may not always be labelled and the effort to label an instance can be expensive, resulting in a delay in labelling. Similar to REDDLA (Li et al., 2012), we can incorporate a clustering technique to our experiment framework to tackle both recurring concept and unlabelled instances.

Appendix A

Average Accuracy and Runtime Improvements

In this appendix we display the table containing general runtime performance of DFT processes on all datasets being tested. Please refer to Table 5.1 for dataset naming convention.

A.1 RBF

			Runtime Improvement (%)	
Dataset	**Order**	**Avg. Accuracy**	**Average**	**St Deviation**
RBF-10	2	70.404	29.926	26.620
	3	70.370	23.715	38.179
	4	70.375	40.254	26.954
	5	70.375	44.364	15.839
RBF-20	2	84.034	38.770	20.553
	3	84.029	53.423	12.225
	4	84.027	64.546	14.951
	5	84.027	64.470	27.886
RBF-40	2	89.848	47.429	14.766
	3	89.845	65.417	7.235
	4	89.845	73.035	20.978
	5	89.845	85.142	15.992

A.2 Rotating Hyperplane

			Runtime Improvement (%)	
Dataset	**Order**	**Avg. Accuracy**	**Average**	**St Deviation**
RH1-10	2	75.507	47.404	10.416
	3	75.505	44.902	12.244
	4	75.505	43.216	19.828
	5	75.505	44.090	11.422
RH1-20	2	69.123	43.888	8.060
	3	68.364	40.400	10.363
	4	68.364	40.690	11.059
	5	68.364	40.108	10.981
RH1-40	2	76.701	49.134	16.438
	3	76.701	48.596	16.783
	4	76.701	46.247	20.073
	5	76.701	47.344	19.607
RH2-10	2	73.186	44.638	10.292
	3	71.811	45.209	11.179
	4	71.811	45.271	12.458
	5	71.811	45.701	12.068
RH2-20	2	73.235	44.986	7.128
	3	73.053	42.672	7.230
	4	73.049	42.960	6.910
	5	73.049	43.536	7.565
RH2-40	2	75.049	47.483	9.863
	3	75.785	49.450	7.786
	4	75.785	44.442	44.889
	5	75.785	48.604	8.948

A.3 Real world datasets

			Runtime Improvement (%)	
Dataset	**Order**	**Avg. Accuracy**	**Average**	**St Deviation**
Electricity	2	68.264	55.914	8.841
	3	68.264	56.477	7.354
	4	68.264	58.770	8.272
	5	68.264	59.084	8.339
Flight	2	82.005	46.006	9.246
	3	81.992	46.973	10.049
	4	81.992	47.878	9.671
	5	81.992	49.109	11.302
Covertype30	2	88.026	59.326	18.283
	3	88.003	60.747	22.770
	4	88.003	59.441	23.054
	5	88.003	46.746	13.700
Covertype40	2	88.256	54.460	22.265
	3	88.295	54.308	22.733
	4	88.295	53.356	25.779
	5	88.295	53.506	24.307

Appendix B

Memory Consumption of memoFT

Dataset	**Order**	**Avg. Memory Usage (bytes)**
RBF-10	2	12027
	3	14925
	4	17138
	5	18205
RBF-20	2	16444
	3	21802
	4	25821
	5	27621
RBF-40	2	19440
	3	26301
	4	30992
	5	32783

Table B.1: Memory usage of memoFT on RBF datasets

Dataset	Order	Avg. Memory Usage (bytes)
RH1-10	2	7277
	3	7414
	4	7444
	5	7444
RH1-20	2	8168
	3	8418
	4	8547
	5	8567
RH1-40	2	7914
	3	8325
	4	8490
	5	8517
RH2-10	2	8196
	3	8860
	4	9046
	5	9079
RH2-20	2	7472
	3	7531
	4	7552
	5	7552
RH2-40	2	8559
	3	7641
	4	7737
	5	7753

Table B.2: Memory usage of memoFT on Rotating Hyperplane datasets

Dataset	Order	Avg. Memory Usage (bytes)
Electricity	2	5748
	3	5755
	4	5757
	5	5757
Flight	2	5889
	3	5985
	4	6009
	5	6014
Covertype30	2	6411
	3	6696
	4	6918
	5	7054
Covertype40	2	6702
	3	7450
	4	8018
	5	8427

Table B.3: Memory usage of memoFT on Real-world datasets

Appendix C

Overall Runtime Measurements

This appendix enlists the tables containing averaged runtime and standard deviations of normal DFT and memoFT runtime. The experiment and its parameters is described in Chapter 5. The total number of Fourier Basis (FB) calculations performed throughout the stream is also recorded to provide additional information for analysis.

Table C.1: RBF - 10 Attributes

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.41005	0.30040	0.35603	0.42201	0.15573	0.10505	1.57576	1.55279	1,742	602
3	0.51733	0.43746	0.55163	0.85798	0.23708	0.21354	2.15304	2.57727	4,577	947
4	0.78681	0.80336	0.59390	0.95307	0.29545	0.34113	2.54330	3.27948	9,100	1,190
5	0.99527	1.21674	0.70325	1.16987	0.36171	0.41904	2.90809	4.07532	13,443	1,268

Table C.2: RBF - 20 Attributes

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.82000	0.58079	0.52550	0.43036	0.16893	0.18895	1.19009	0.37051	8,679	1,337
3	1.69965	1.59551	0.80288	0.82697	0.26835	0.24859	1.59838	0.56082	37,568	2,131
4	4.10510	5.10923	0.96385	0.97153	0.36882	0.40813	1.80621	0.64944	120,242	2,575
5	9.49190	14.92804	1.20162	1.31109	0.51219	0.55390	2.59975	0.92228	302,570	2,796

Table C.3: RBF - 40 Attributes

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	1.42500	0.89657	0.82402	0.83545	0.20494	0.53110	3.76075	5.18292	14,263	1,537
3	3.45329	3.02763	1.16915	1.24863	0.25564	0.53198	3.93806	5.23231	75,925	2,486
4	11.19494	13.55899	1.89780	1.82063	0.36555	0.45652	3.97299	3.34227	300,063	3,095
5	34.16064	48.58551	1.95970	2.06071	0.39569	0.51145	4.20637	3.35017	926,317	3,276

Table C.4: Rotating Hyperplane - 10 Attributes, magnitude 0.01

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.12277	0.06876	0.06900	0.05221	0.04428	0.03567	0.14006	0.14896	1,894	1,629
3	0.12387	0.07075	0.07393	0.05903	0.04702	0.04206	0.14674	0.16119	2,335	1,917
4	0.12703	0.07374	0.07522	0.05849	0.04853	0.03838	0.15326	0.16445	2,422	1,968
5	0.12536	0.07165	0.07541	0.05896	0.04809	0.03811	0.15172	0.16415	2,422	1,968

Table C.5: Rotating Hyperplane - 10 Attributes, magnitude 0.05

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.14572	0.12299	0.08635	0.09348	0.05581	0.04325	0.20416	0.33473	1,591	1,391
3	0.14367	0.12918	0.08767	0.10841	0.07250	0.07666	0.21601	0.40344	1,240	1,099
4	0.14315	0.13021	0.08939	0.12270	0.07499	0.07895	0.21728	0.42829	1,333	1,183
5	0.14640	0.13381	0.09090	0.12445	0.07214	0.05655	0.21704	0.42692	1,346	1,196

Table C.6: Rotating Hyperplane - 20 Attributes, magnitude 0.01

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.16200	0.11262	0.09575	0.08029	0.06488	0.06524	0.33224	0.70015	1,617	1,335
3	0.16703	0.12394	0.10838	0.10533	0.07646	0.08643	0.38803	0.81726	1,859	1,448
4	0.16776	0.13125	0.11045	0.11780	0.07587	0.07436	0.38063	0.74226	2,048	1,556
5	0.16850	0.12974	0.11159	0.11802	0.07627	0.07857	0.38078	0.75214	2,080	1,572

Table C.7: Rotating Hyperplane - 20 Attributes, magnitude 0.05

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.12038	0.03021	0.06637	0.01885	0.06138	0.06236	0.14468	0.06332	389	333
3	0.12375	0.04844	0.07269	0.03543	0.06626	0.08178	0.15583	0.11966	708	576
4	0.12462	0.05141	0.07244	0.03555	0.06370	0.06093	0.15610	0.11760	724	584
5	0.12462	0.04941	0.07225	0.03788	0.06352	0.06120	0.15569	0.12246	724	584

Table C.8: Rotating Hyperplane - 40 Attributes, magnitude 0.01

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.20252	0.17508	0.11501	0.14893	0.08958	0.08969	0.35810	0.54319	645	548
3	0.18748	0.16640	0.11955	0.19160	0.10798	0.11416	0.39173	0.62409	909	748
4	0.18235	0.16649	0.12707	0.22193	0.10557	0.11395	0.38857	0.63238	945	812
5	0.18326	0.16732	0.12406	0.22308	0.10087	0.09217	0.37711	0.62574	1,018	832

Table C.9: Rotating Hyperplane - 40 Attributes, magnitude 0.05

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.16984	0.10733	0.09804	0.09575	0.07482	0.02830	0.30048	0.34714	1,530	1,283
3	0.12947	0.08188	0.07073	0.07010	0.07302	0.04322	0.19365	0.27845	2,619	2,250
4	0.13140	0.08420	0.07966	0.09902	0.07914	0.07014	0.20985	0.33731	2,838	2,390
5	0.12954	0.08392	0.07281	0.07779	0.07473	0.04753	0.19426	0.28682	2,876	2,412

Table C.10: Electricity dataset

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.07391	0.01992	0.03263	0.01179	0.07537	0.41639	0.17821	0.64263	275	247
3	0.07394	0.02172	0.03265	0.01481	0.08096	0.42553	0.19190	0.66160	304	269
4	0.07232	0.02115	0.03039	0.01520	0.07484	0.40932	0.17734	0.63433	314	275
5	0.07450	0.02571	0.03111	0.01693	0.06722	0.40784	0.15859	0.63163	316	276

Table C.11: Flight dataset

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.09978	0.03534	0.05529	0.02729	0.05265	0.20067	0.15316	0.35979	870	522
3	0.10444	0.03879	0.05775	0.03306	0.04390	0.08166	0.12499	0.21592	1,083	586
4	0.10338	0.03992	0.05604	0.03150	0.03773	0.04889	0.10511	0.10969	1,145	596
5	0.09927	0.03926	0.05295	0.03172	0.03475	0.04962	0.09684	0.10935	1,152	596

Table C.12: Covertype - 30 Attributes

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.09091	0.07456	0.04561	0.06985	0.03883	0.03923	0.08016	0.14930	5,074	2,604
3	0.09713	0.13410	0.05938	0.20528	0.04672	0.05999	0.09179	0.27475	11,156	3,638
4	0.11442	0.22013	0.07692	0.30662	0.05858	0.09205	0.11866	0.39552	20,677	4,645
5	0.17652	0.39756	0.12234	0.39568	0.06710	0.10488	0.18454	0.55750	36,583	5,737

Table C.13: Covertype - 40 Attributes

Order	Normal DTF Runtime (ms)		Lazy memoFT Runtime (ms)		Eager memoFT Runtime (ms)		Cumulative Eager Runtime (ms)		Total FB Calculation	
	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Average	St. Deviation	Normal DFT	memoFT
2	0.11241	0.10857	0.06924	0.17402	0.06084	0.07249	0.12104	0.22201	6,932	3,265
3	0.14109	0.25929	0.10526	0.35292	0.07297	0.09095	0.16989	0.49721	19,834	5,399
4	0.17609	0.46732	0.15569	0.60410	0.08446	0.13539	0.21877	0.73068	47,075	8,321
5	0.22392	0.81566	0.18009	0.74081	0.09119	0.16580	0.24551	0.87707	82,764	10,262

Appendix D

Runtime Profile on Drift Detection

The following figures are the runtime of the three DFT processes measured whenever concept drift is detected. The eager evaluation in between drifts are cumulated. Note that due to lack of drifts detected on the RBF dataset, we also show the runtime measured when eager evaluation has not started.

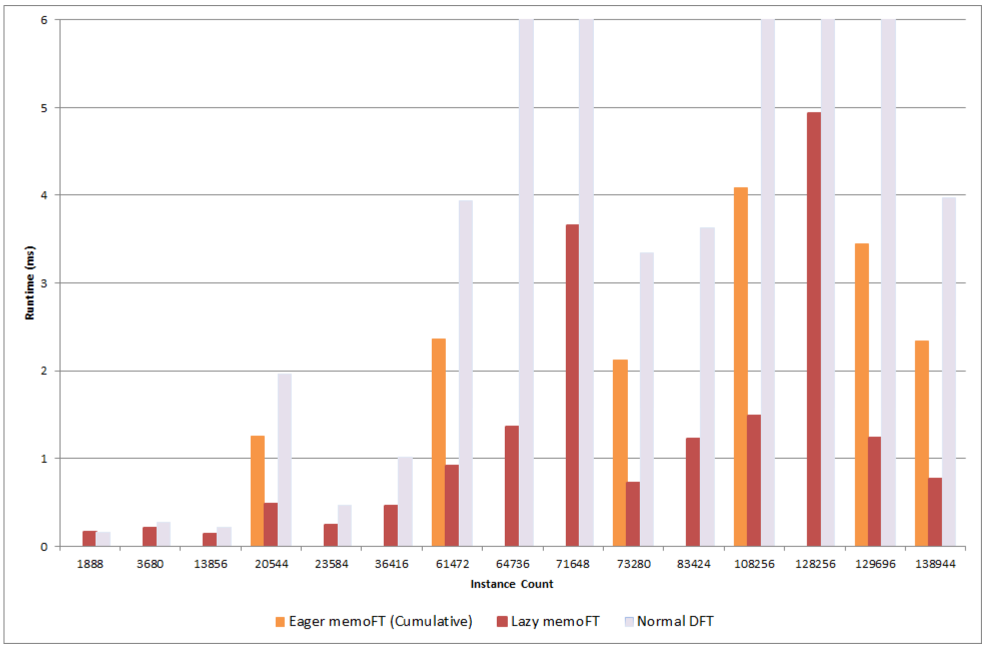


Figure D.1: Runtime of DFT processes on RBF dataset (20 attributes, minimum winner count = 1) measured on drift detection

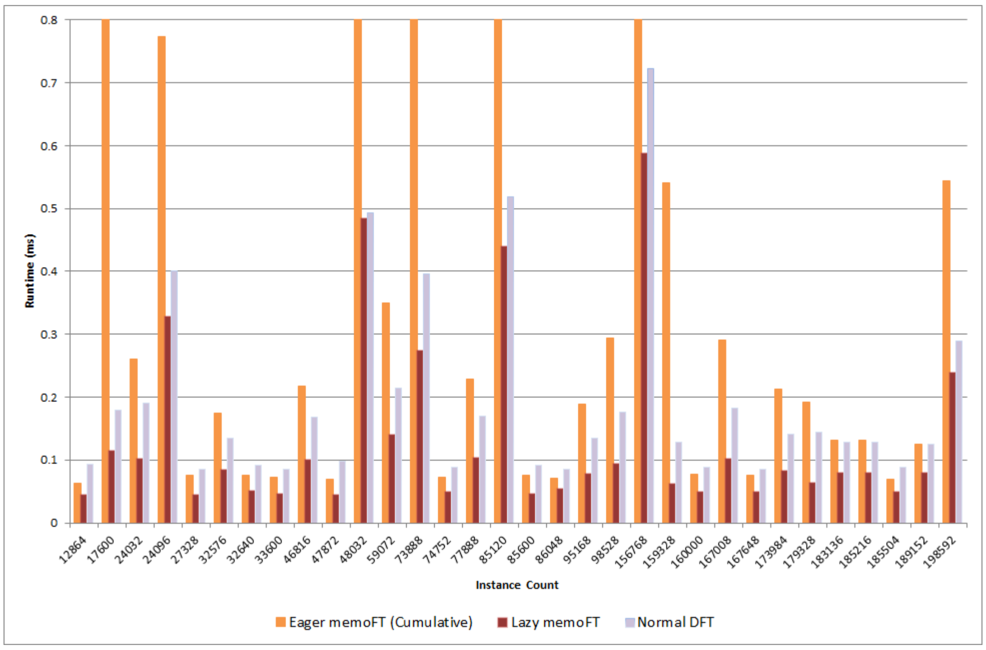


Figure D.2: Runtime of DFT processes on Rotating Hyperplane dataset (20 attributes, drift magnitude = 0.01) measured on drift detection

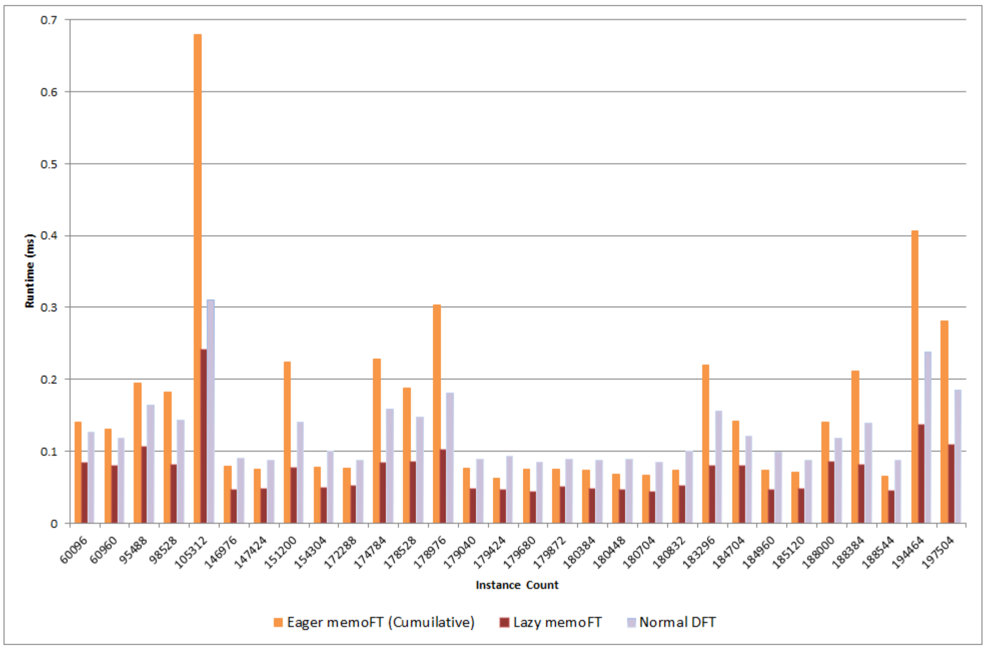


Figure D.3: Runtime of DFT processes on Rotating Hyperplane dataset (20 attributes, drift magnitude = 0.05) measured on drift detection

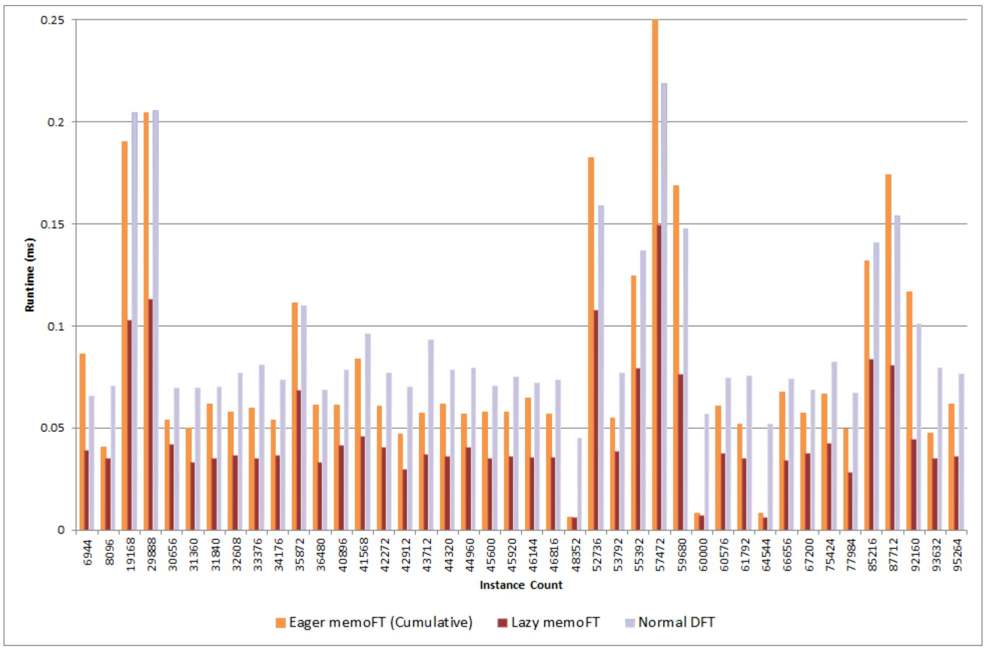


Figure D.4: Runtime of DFT processes on Covertype40 dataset measured on drift detection

References

Ade, M., & Deshmukh, P. (2013). Methods for incremental learning: a survey. *International Journal of Data Mining & Knowledge Management Process*, *3*(4), 119–125.

Arfken, G. B., & Weber, H.-J. (2001). *Mathematical methods for physicists* (5th ed.). Harcourt/Academic Press.

Basseville, M., Nikiforov, I. V., et al. (1993). *Detection of abrupt changes: theory and application* (Vol. 104). Prentice Hall Englewood Cliffs.

Bifet, A. (2010). Adaptive stream mining: Pattern learning and mining from evolving data streams. In *Proceedings of the 2010 conference on adaptive stream mining: Pattern learning and mining from evolving data streams* (pp. 1–212).

Bifet, A., & Gavalda, R. (2006). Kalman filters and adaptive windows for learning in data streams. In *Discovery science* (pp. 29–40).

Bifet, A., & Gavalda, R. (2007). Learning from time-changing data with adaptive windowing. In *Sdm* (Vol. 7, p. 2007).

Bifet, A., Holmes, G., Pfahringer, B., Kirkby, R., & Gavaldà, R. (2009). New ensemble methods for evolving data streams. In *Proceedings of the 15th acm sigkdd international conference on knowledge discovery and data mining* (pp. 139–148).

Blackard, J. A., & Dean, D. J. (1999). Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and electronics in agriculture*, *24*(3), 131–151.

Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). *Classification and regression trees*. CRC press.

Brigham, E. O. (1988). *The fast fourier transform and its applications* (Vol. 1). Prentice Hall Englewood Cliffs, NJ.

Chapelle, O., Schölkopf, B., Zien, A., et al. (2006). *Semi-supervised learning.* MIT press Cambridge.

Chu, E., Gorinevsky, D., & Boyd, S. (2010). Detecting aircraft performance anomalies from cruise flight data. In *Proceedings of the 2010 AIAA infotech aerospace conference.*

Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, *19*(90), 297–301.

Datar, M., Gionis, A., Indyk, P., & Motwani, R. (2002). Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, *31*(6), 1794–1813.

Domeniconi, C., & Gunopulos, D. (2001). Incremental support vector machine construction. In *Data mining, 2001. icdm 2001, proceedings ieee international conference on* (pp. 589–592).

Domingos, P., & Hulten, G. (2000). Mining high-speed data streams. In *Proceedings of the sixth acm sigkdd international conference on knowledge discovery and data mining* (pp. 71–80).

Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI magazine*, *17*(3), 37.

Gaber, M., Krishnaswamy, S., & Zaslavsky, A. (2005). On-board mining of data streams in sensor networks. In *Advanced methods for knowledge discovery from complex data* (p. 307-335). Springer London. Retrieved from `http://dx.doi.org/10.1007/1-84628-284-5_12` doi: 10.1007/1-84628-284-5_12

Gama, J., & Kosina, P. (2011). Learning about the learning process. In *Advances in intelligent data analysis x* (Vol. 7014, p. 162-172). Springer.

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

Gama, J., Rocha, R., & Medas, P. (2003). Accurate decision trees for mining high-speed data streams. In *Kdd* (p. 523).

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

Gentleman, W. M., & Sande, G. (1966). Fast fourier transforms: for fun and profit. In *Proceedings of the november 7-10, 1966, fall joint computer conference* (pp. 563–578).

Harries, M., & Wales, N. S. (1999). Splice-2 comparative evaluation: Electricity pricing.

Heckerman, D., Geiger, D., & Chickering, D. M. (1995). Learning bayesian networks: The combination of knowledge and statistical data. *Machine learning*, *20*(3), 197–243.

Hoeglinger, S., & Pears, R. (2007). Use of hoeffding trees in concept based data stream mining. In *Information and automation for sustainability, 2007. iciafs 2007. third international conference on* (pp. 57–62).

Hoeglinger, S., Pears, R., & Koh, Y. S. (2009). Cbdt: A concept based approach to data stream mining. In *Advances in knowledge discovery and data mining* (pp. 1006–1012). Springer.

Hulten, G., & Domingos, P. (2002). Mining complex models from arbitrarily large databases in constant time. In *Proceedings of the eighth acm sigkdd international conference on knowledge discovery and data mining* (pp. 525–531).

Hulten, G., Spencer, L., & Domingos, P. (2001). Mining time-changing data streams. In *Proceedings of the seventh acm sigkdd international conference on knowledge discovery and data mining* (pp. 97–106).

Jacobson, G. (1989). Space-efficient static trees and graphs. In *Foundations* *of computer science, 1989., 30th annual symposium on* (pp. 549–554).

Kargupta, H., & Park, B.-H. (2004). A fourier spectrum-based approach to represent decision trees for mining data streams in mobile environments. *Knowledge and Data Engineering, IEEE Transactions on*, *16*(2), 216–229.

Kargupta, H., Park, B.-H., & Dutta, H. (2006). Orthogonal decision trees. *Knowledge and Data Engineering, IEEE Transactions on*, *18*(8), 1028–1042.

Katakis, I., Tsoumakas, G., & Vlahavas, I. P. (2008). An ensemble of classifiers for coping with recurring contexts in data streams. In *Ecai* (pp. 763–764).

Kifer, D., Ben-David, S., & Gehrke, J. (2004). Detecting change in data streams. In *Proceedings of the thirtieth international conference on very large data bases-volume 30* (pp. 180–191).

Kingston, J. H., & Kingston, J. H. (1990). *Algorithms and data structures: design, correctness, analysis.* Addison-Wesley Sydney.

Kirkby, R. B. (2007). *Improving hoeffding trees* (Unpublished doctoral dissertation). The University of Waikato.

Kleinberg, J., & Tardos, É. (2006). *Algorithm design.* Pearson Education, Inc.

Li, P., Wu, X., & Hu, X. (2012). Mining recurring concept drifts with limited labeled streaming data. *ACM Transactions on Intelligent Systems and Technology (TIST)*, *3*(2), 29.

Linial, N., Mansour, Y., & Nisan, N. (1993). Constant depth circuits, fourier transform, and learnability. *Journal of the ACM (JACM)*, *40*(3), 607–620.

Mansour, Y. (1994). Learning boolean functions via the fourier transform. In *Theoretical advances in neural computation and learning* (pp. 391–424). Springer.

Oza, N. C., & Russell, S. (2001). Experimental comparisons of online and batch versions of bagging and boosting. In *Proceedings of the seventh acm sigkdd international conference on knowledge discovery and data mining* (pp. 359–364).

Park, B.-H. (2001). *Knowledge discovery from heterogeneous data streams using fourier spectrum of decision trees* (Unpublished doctoral dissertation). Washington State University, Pullman, WA, USA. (AAI3051936)

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, *1*(1), 81–106.

Quinlan, J. R. (1993). C4. 5: Programming for machine learning. *Morgan Kauffmann.*

Ramamurthy, S., & Bhatnagar, R. (2007). Tracking recurrent concept drift in streaming data using ensemble classifiers. In *Machine learning and applications, 2007. icmla 2007. sixth international conference on* (pp. 404–409).

Schlimmer, J. C., & Granger Jr, R. H. (1986). Incremental learning from noisy data. *Machine learning*, *1*(3), 317–354.

Sripirakas, S. (2015). *Capturing recurring concepts in high speed data streams* (Unpublished doctoral dissertation). Auckland University of Technology.

Sripirakas, S., & Pears, R. (2014). Mining recurrent concepts in data streams using the discrete fourier transform. In *Data warehousing and knowledge discovery* (Vol. 8646, pp. 439–451). Springer International Publishing.

Street, W. N., & Kim, Y. (2001). A streaming ensemble algorithm (sea) for large-scale classification. In *Proceedings of the seventh acm sigkdd international conference on knowledge discovery and data mining* (pp. 377–382). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/502512.502568` doi: 10.1145/502512.502568

Wang, H., Fan, W., Yu, P. S., & Han, J. (2003). Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the ninth acm sigkdd international conference on knowledge discovery and data mining* (pp. 226–235).

Witten, I. H., & Frank, E. (2005). *Data mining: Practical machine learning tools and techniques.* Morgan Kaufmann.