Otto-von-Guericke-University Magdeburg
Faculty of Computer Science

Master Thesis

A comparison of Time Series Classification Algorithms based on their ability to learn in an Early Classification Context

Author:
Ismail, Wahba

Matriculation Number:
217526

Examiner and Supervisor:
Prof. Dr. Myra Spiliopoulou
Knowledge Management and Discovery Lab
Institute of Technical and Business Information Systems
Otto-von-Guericke-University Magdeburg

2nd Supervisor:
Name
Institute
University

25.05.2021

Wahba, Ismail:
A comparison of Time Series Classification Algorithms based on their ability to learn in an Early Classification Context
Master Thesis, Otto-von-Guericke-University Magdeburg, 2021.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur eget porta erat. Morbi consectetur est vel gravida pretium. Suspendisse ut dui eu ante cursus gravida non sed sem. Nullam sapien tellus, commodo id velit id, eleifend volutpat quam. Phasellus mauris velit, dapibus finibus elementum vel, pulvinar non tellus. Nunc pellentesque pretium diam, quis maximus dolor faucibus id. Nunc convallis sodales ante, ut ullamcorper est egestas vitae. Nam sit amet enim ultrices, ultrices elit pulvinar, volutpat risus.

Acknowledgement

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur eget porta erat. Morbi consectetur est vel gravida pretium. Suspendisse ut dui eu ante cursus gravida non sed sem. Nullam sapien tellus, commodo id velit id, eleifend volutpat quam. Phasellus mauris velit, dapibus finibus elementum vel, pulvinar non tellus. Nunc pellentesque pretium diam, quis maximus dolor faucibus id. Nunc convallis sodales ante, ut ullamcorper est egestas vitae. Nam sit amet enim ultrices, ultrices elit pulvinar, volutpat risus.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ABR	Auditory Brainstem Response
BOSS	Bag of SFA Symbols
CNN	Convolutional Neural Networks
DFT	Discrete Fourier Transformation
DTW	Dynamic Time Warping
ECTS	Early Classification of Time Series
EE	Elastic Ensemble
ERP	Edit Distance with Real Penalty
eTSCA	Early Time Series Classification Algorithms
FS	Fast Shapelets
HM	Harmonic Mean
IG	Information Gain
i.i.d	Independent and Identically Distributed
KNN	K-Nearest Neighbor Algorithm
KNNED	K-Nearest Neighbor using Euclidean Distance
LCSS	Longest Common Subsequence
LOOCV	Leave One Out Cross Validation
LS	Learned Shapelets
MCB	Multiple Coefficient Binning
MPL	Mimimum Prediction Length
MSM	Move-Split-Merge
MTSC	Multivariate Time Series Classification
PAA	Piecewise Aggregate Approximation
PF	Proximity Forest
SAX	Symbolic Aggregate Approximation
SFA	Symbolic Fourier Approximation
ST	Shapelet Transform
TSC	Time Series Classification
TSCA	Time Series Classification Algorithms
TSF	Time Series Forest
TWE	Time Warp Edit
WDTW	Weighted Dynamic Time Warping

Chapter 1

Introduction

Time Series Classification (TSC) is a research field which has grabbed a lot of attention in the data mining community during the last decade. This goes back to the fact that time series data exists, by nature, in numerous scenarios in our life; medical examination records of patients [GM01; GO12], consumption of electrical devices in smart homes and smart grids [Gao+14; JZ14] and astronomy [Pro+06; RK13] are only few examples of them.

The problem of time series data classification has been tackled in different ways and motivated by vairous objectives. When the main driving objective is to attain high classification accuracy, then this is referred to simply as Time Series Classification (TSC). TSC has also been extended to cover space and time complexity along with accuracy. On the other hand, if time plays a critical role in the problem being tackled beside accuracy; like early detection of malfunctions in industrial processes or diagnosis of a patient with a life threatning disease. In this case the classification problem is referred to as Early Time Series Classification (eTSC).

A lot of algorithms have been introduced to tackle the problem of TSC. According to the [Bag+17], these algorithms can be divided, based on their technique, into six groups. Distance-based time series algorithms compare instances of time series, usually using 1-NN classifiers and an elastic distance measure between the data points of both time series. Phase dependent interval algorithms operate by extracting informative features from intervals of time series, they are more suitable for long and noisy data than distance-based time series algorithms. Phase independent interval algorithms are used when a class can be identified using a single or multiple patterns regardless of when they occur during the time series. Dictionary based algorithms consider the number of repetitions of patterns as a factor of classification and not just simple occurence of one. Ensembles combine the power of different algorithms, either of different or same core technique, then make the final classification decision based on voting. In addition to the previous algorithms, there are also deep learning time series algorithms which build classifiers using generative as well as discriminative neural networks models.

On the other hand, Early Time Series Classification algorithms are designed to deal with less data in order to achieve earliness of prediction, without sacrificing accuracy as much as possible. Since accuracy and earliness are contradicting by nature, eTSC's main goal is to find a tradeoff between both objectives [Gup+20]. Many of the techniques that have been applied in TSC have also been applied for eTSC. Using 1-

NN classifiers is a very popular TSC technique, which has been extended for eTSC by combining it with Mimimum Prediction Length (MPL)[XPP12]. Phase independent intervals[GO12; He+15], generative and discriminative model based classifiers[LLF14; Che+18] as well as deep learning algorithms [HLT18; Ruß+19] have all been extended to tackle eTSC problems.

Both, Time Series Classification Algorithms (TSCA) and Early Time Series Classification Algorithms (eTSCA), have introduced well performing algorithms in terms of their respective performance measures. The existence of comprehensive benchmark data archives [Dau+18; Bag+18] has helped regulate and standardize the process of comparison between classifiers. Algorithms can be benchmarked on a diverse set of datasets with different characteristics to avoid biased results and cherry picking.

Like other fields, the *No free lunch theorem* exists in the TSC domain as well [Bag+17; Bos18]. This means that there is no one specific algorithm that can generalize to all problems and prevail over all other competing algorithms. With that being the case, this leaves an open door for continuous investigation of how the performance of algorithms would change on different data sets and in different contexts.

Inspired by the of the problem of eTSC and its application in diverse areas. We define a situation in which a classifer is put into test to learn on only a part of the available data; to see if it will be able to gracefully degrade. We consider a classifier to be graceful, or fault tolerant, if it is able to maintain statistically significant results by learning on less data in comparison to it's results when learning on full length data. We refer to this situation as the early classification context.

The motivation behind experimenting with such a context is driven by the medical field, where sometimes patients have to go through strenuous tests to be diagnosed. This is specially the case in audiological examinations; as patients are exposed to signals which pushes the limits of their hearing discomfort to be able to examine their problems. A classifier which can attain comparable reasults while only learning on a fraction of the whole time series data would shorten this process and thus relief patients from their discomfort.

Goal of this Thesis

The goals of this master thesis can be addressed as:

- Provide a solution for adapting existing time series workflows for an early classification context. The solution should run experiments on a group of data sets from the UCR/UEA archive and measure the performance of classifiers within the early classification context. By the end of the experiments the solution should collect the results and provide an analysis of the results, the analysis should:
  - Identify if classifiers can sustain good performance while learning on fewer data points in comparison to learning on the full length
  - Identify if there is a winning classifier across the executed erxperiments
  - Identify the aspects of the data set which affect the performance of classifiers within the context. Such as train size, number of classes, number of dimensions, length, etc.
- Identify a suitable evaluation criteria for classifiers in the proposed early classification context, trading classification accuracy against time series length.

- Evaluate the proposed solution for adapting the time series workflows to be used for future recommendations

Chapter 2

Classification Algorithms

This Chapter discusses the definitions and background of the topics mentioned in this thesis. We discuss the nature of time series data, the two problems of time series classification (TSC) and early time series classification (eTSC) then present the different techniques encompassed by them. We will discuss TSCAs in more details as they are the main focus of our framework, but a review of the problem of early classification and a review of it's algorithms is also included; as it motivates the context in which the framework operates.

2.1 Time Series Data

2.1.1 Definitions

A time series is a finite sequence of ordered observations, either based on time or another aspect [AML19; Bag+17]. The existence of the time component makes time series an abundant form of data that covers various domains like; medicine, finance, engineering and biology [LTB18].

A time series dataset is a collection of time series instances.

Definition 1

$$\textit{A set } T \textit{ of } n \textit{ time series instances, } T = \{T_1, T_2, \ldots, T_n\}.$$

Each of the time series instances T_i consists of a sequence of observations collected at consequent time points.

Definition 2

$$\textit{A time series } T_i\textit{, of length } L \textit{ is represented as } T_i = [t_1, t_2, \ldots t_L].$$

An observation t_i represents the value of the collected measurement at some point of time i.

Time series data can come in different forms. It is important to comprehend what different forms the data can take and what implicit assumptions they convey; to be able to choose the suitable algorithms and tools to deal with it.

The first form of time series data is referred to as univariate time series. This form of data exists when the observations of instances capture a singular value across time.

Definition 3

Univariate time series T_i, of length L is represented as $T_i = [t_1, t_2, \dots t_L]$
With t_j as a real valued number.

The other form is when multiple measurments are captured by the observations. According to [Lön+19], it is essential to differentiate between the two ways multiple time series can be generated; panel data and multivariate time series data. If more than one variable is being observed during a single experiment, with each variable representing a different measurement; this is called multivariate time series.

Definition 4

Multivariate time series T_i, of length L is represented as $T_i = [t_1, t_2, \dots t_L]$
With t_j having M dimensions, each one of them is a univariate time series.

While panel data is when the same kind of measurments is collected from independent instances; like different patients or diverse industrial processes. For panel data, it is possible to assume that the different instances are independent and identically distributed (i.i.d), but this assumption doesn't hold for observation of a single instance. The same goes for multivariate time series, individual univariate observations are assumed to be statistically dependant.

2.1.2 Nature of Time Series Data

Having discussed the dependency assumptions in time the different forms of time series data. It is this dependency that makes time series data challenging for conventional machine learning algorithms, which are used for tabular and cross-sectional data. Tabular and cross-sectional data assume observations to be i.i.d [Lön+19].

If we were to tabularize time series data; convert it into a tabular form by considering each observation as an individual feature. Then it would be possible to apply conventional machine learning algorithms, under the implicit modelling assumption that observations are not ordered. This means that if the order of the features was changed, still the model result will not change. This assumption can work for some problems, but it doesn't have to work for all problems.

2.2 Time Series Classification

Time series classification is a subtype of the general classification problem, which considers the the unique property of dependency between adjacent features of instances [BB17b]. The main goal of time series classification is to learn a function f, which given a training dataset $T = \{T_1, T_2, \dots, T_n\}$ of time series instances along with their corresponding class labels $Y = \{y_1, y_2, \dots y_n\}$ where $y_i \in \{1, 2, \dots C\}$, can predict class labels for unseen instances [Den+13].

Time series classification has been studied with different objectives, some papers focused on attaining the highest accuracy of classification as the main goal [Kat16; JJO11; BB17a; LTB18; SL17b; Faw+20], while other papers focused on attaining lower time complexity [RK04; Bag+17; TPW20; Pet+16; SL17a].

In this master thesis, we are more interested in assessing the results in terms of accuracy than time complexity. We define accuracy like [SL20]; as the percentage of

correctly classified instances for a given dataset D, either being a training or testing dataset.

Definition 5

$$Accuracy = \frac{number\ of\ correct\ classifications}{|D|}$$

2.3 Time Series Classification Algorithms

This chapter will introduce different types of TSCAs and discuss the various techniques that each type apply. There are multiple ways to divide TSCAs in order to better understand them. In this thesis we follow the grouping defined by [Bag+17].

2.3.1 Whole Time Series Algorithms

Whole time series similarity algorithms, also called distance-based algorithms, are methods that compare pairs of time series instances. An unlabeled time series instance is given the class label of the nearest instance in a training data set [Kat16]. There are two main techniques for carrying out the comparison; either by comparing vector representations of the time series instances, or by combining a defined distance function with a classifier, KNN being the most common one [LTB18]. Whole time series algorithms are best suited for problems where the unique features can exist anywhere along the whole time series[Bag+17].

One of the simplest forms of whole time series is 1-NN with Euclidean Distance [FRM94], yet it can suprisingly attain high accuracy compared to other distance measures [XPK10]. But Nearest Neighbor with Euclidean Distance (KNNED) is an easy to beat baseline, due to it's sensitivity for distortion and inability to handle time series of unequal lengths [XPK10; Kat16; LTB18]. This lead many of the researchers to focus on defining more advanced and elastic distance measures that can compensate for misalignment between time series [AML19]. The standard and most common baseline classifier utilizing elastic distance measures is 1-NN with Dynamic Time Warping (DTW) [Bag+17]. In contrast to the idea that more powerful machine learning algorithms will be able to defeat the simple KNN and an elastic measure, DTW has proved to be a very tough opponent to other algorithms and other elastic distance measures as well [Kat16; LB15; Wan+13]. But there were also other distance metrics that have been introduced in literature, these inlcude extensions of DTW on one hand like; Weighted Dynamic Time Warping (WDTW) which penalizes large warpings based on distance [JJO11] and Derivative Dynamic Time Warping (DDTW) [KP01; GŁ13] which uses derivatives of sequences as features rather than raw data to avoid singularities. On the other hand, Edit Distance with Real Penalty (ERP) [CN04], Time Warp Edit (TWE) [Mar08], Longest Common Subsequence (LCSS) [DGM97] and Move-Split-Merge (MSM) [SAD12] are all alternatives for distance measures, yet multiple experiments have considered DTW to be relatively unbeatable [Bag+17; AML19; BB17a]. To the extend of our knowledge, the most powerful whole time series classifiers is Proximity Forest (PF) [Luc+19].

Nearest Neighbor with ED

The Euclidean distance is a remarkably simple technique to calculate the distance between time series instances. Given two instances $T_1 = [t1_1, t1_2, ...t1_n]$ and $T_2 =$

[$t2_1$,$t2_2$,...$t2_n$], the euclidean distance between them can be determined as:

$$ED(T_1, T_2) = \sqrt{\sum_{i=1}^{n}(t1_i - t2_i)^2} \tag{2.1}$$

Euclidean distnace has been preferred to other classifiers due to it's space and time efficiency, but it suffers from two main shortcomings [BRT13; JJO11; Kat16]. The first one is that it cannot handle comparisons between time series of different lengths. While the second one is it's sensitivity to minor discrepancies between time series; it would calculate large distance values for small shiftings or misalignments. Although other metrics have been introduced to overcome the drawbacks of euclidean distance, experimental proof showed that this is only the case for small datasets, but for larger datasets the accuracy of other elastic measures converge with euclidean distance [Hil+14; Din+08; Bag+12].

Nearest Neighbor with DTW

Dynamic Time Warping was a very strong baseline for time series classification for a long time [AML19; Bag+17]. It was first introduced as a technique to recognize spoken words that can deal with misallignments between time series that Euclidean Distance couldn't handle [TPW20]. Figure 2.1 shows a comparison between ED and DTW while computing the distance between two time series instances.

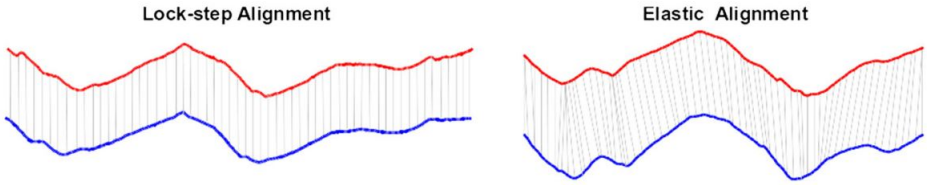


Figure 2.1: Mapping of Euclidean distance (lock-step measure) versus mapping of DTW distance (elastic measure) [AML19].

To calculate the distance between two time series instances $T_1 = [t1_1, t1_2, ...t1_n]$ and $T_2 = [t2_1, t2_2, ...t2_n]$; a distance matrix $M(T_1, T_2)$, of size n × n, is calculated for T_1 and T_2. With $M_{i,j}(t1_i, t2_j)$ representing the distance between $t1_i \in T_1$ and $t2_j \in T_2$. The goal of DTW is to find an optimal path that minimizes the cumulative distance between points of T_1 and T_2.

A candidate path $P = [p_1, p_2, ...p_p]$ is to be found by traversing M. For a path to be valid it must conform to three conditions:

- $p_1 = (t1_1, t2_1)$
- $p_p = (t1_n, t2_n)$
- for all i <m :
  - $0 \leq t1_{i+1} - t1_i \leq 1$
  - $0 \leq t2_{i+1} - t2_i \leq 1$

Finding an optimal path under DTW can be computationally expensive with complexity of $O(n^2)$ for a time series of length n [SL17a; Pet+16]. Consequently it is usual to use a contraint with the path; to prevent comparison of points outside a certain window [TPW20]; like the famous Sakoe-Chiba Band [SC78], Itakura Parallelogram [Ita75] and Ratanamahatana-Keogh Band [RK04]. Typically DTW can make use of it's warping window to handle distortion in time series, but still it is vulnerable to cases where the difference in length between instances length is larger than the warping window [Tan+19].

Nearest Neighbor with MSM Distance

Move-Split-Merge is distance measure that was first introduced in [SAD12]. The main purpose of introducing MSM was to combine certain characteristics within one distance measure. These are; robustness to misalignments between time series instances, being an actual metric unlike other distance measures like DTW and LCSS, assure translation invariance and achieve quadratic run-time [LB15].

The way MSM works is pretty much like other edit distance methods; it determines the similarity between two instances through the usage of a set of operations to transform one of them to the other. These operations, as the name indicates, are; move, split and merge [Bag+17].

Move is the substitution of one single time point of a series with another. Split divides a single time point of a series into two consecutive time points holding the same value as the originial time point. Merge can be seen as the opposite of Split, it combines two consecutive time points holding the same value into one time point.

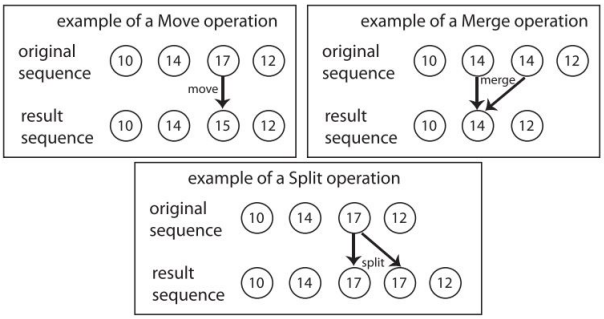


Figure 2.2: Examples of the Move, Split, Merge operations [SAD12].

Each of the previously mentioned operations is associated with a cost. The cost of a move is equal to the absolute difference between the old and the new value of the time point. The costs of split and merge are equal and they are set to a constant to satisfy the symmetry property of metricity [SAD12; TPW20].

Proximity Forest

Proximity forest was developed by [Luc+19]. It was introduced as an addition to scalable time series classification, offering a more scalable and accurate classifier than EE [TPW20]. On one side, EE was an accurate classifier being one the state of the

art algorithms and the best among distnace based algorithms, as it combines 11 NN-algorithms each using a different elastic measure. But on the other hand, EE's training process was very slow as it scales quadratically with the training size of the data set [LB15; Bag+17]. This goes back to the leave-one-out-cross-validation (LOOCV) used to optimize the parameters for each used metric [Shi+20].

Proximity Forest wanted to achieve two main goals. The first was to offer an adaptable algorithm that can scale with huge data sets consisting of millions of time series instances. Beating EE, by orders of maginute, and other state of the art algorithms in terms of training and testing run time complexity. While the other goal was to develop a competitive algorithm on the UCR data sets archive without the need to sacrifice accuracy for scalability as is the case with BOSS-VS [Luc+19].

Capitalizing on the previous research that has been put in developing specialized time series distance measures and inspired by the existing EE [Faw+20; Faw+19b]. Proximity forests combine the the eleven elastic distances from EE along with a tree-based algorithms to form an ensemble of decision trees. The reason behind using tree-based algorithms lies in the divide-and-conquer strategy that they adopt, which makes them scalable for large data sets. Also a stochastic process is used for the selection of distance measures and their hyper-parameters, which usually hinders the performance of other algorithms, like KNN, that need to learn the hyper-parameters of the utilized distance measure for each data set before using it [Luc+19]. Proximity forests can scale sublinearly with training data set size, but quadratically with the length of the time series [Shi+20].

Proximity forests are based on a similar concept as Random Forests [Bre01], another tree-based ensemble, which learns only on a subset of the available features for building tree nodes. This process insinuates in a factor of variability between the trees that form the ensemble but each with a low bias. The collective classification accuracy of the ensemble then tends to provide better results than any if it's single classifiers [Luc+19].

The building unit of a proximity forest is called the proximity tree. A proximity tree and a decision tree are similar on all aspects, but they differ in the tests they apply in internal nodes. A conventional decision tree builds it's nodes using attributes. When an instance is being tested, it is compared to the value of the attribute and then follows the branch to which it conforms.

Unlike conventional decision trees, that use feature values for their nodes, proximity trees build their nodes using randomly selected examplars. When an instance to be tested, an elastic distance measure is calculated and then it follows the branch of the nearest examplar.

An internal node in the tree holds two attributes; *measure* and *branches*. As noted in [Luc+19], a measure is function $object \times object \rightarrow \mathbb{R}$. Proximity Forest uses the same 11 distance measures used by EE; Euclidean distance (ED) Dynamic time warping using the full window (DTW); Dynamic time warping with a restricted warping window (DTW-R); Weighted dynamic time warping (WDTW); Derivative dynamic time warping using the full window (DDTW); Derivative dynamic time warping with a restricted warping window (DDTW-R); Weighted derivative dynamic time warping (WDDTW); Longest common subsequence (LCSS); Edit distance with real penalty (ERP); Time warp edit distance (TWE); and, Move-Split-Merge (MSM). Proximity Forest saves a lot of the computational cost by replacing parameter searches with random sampling [Faw+20; Faw+19a]. While branches is a vector of the possible branches to follow, each branch holding two attributes; *examplar* and *subtree*. *examplar* is a time series instance to which a query instance is compared, and *subtree* refers to the

tree an instance should follow in case it is closest to a specific examplar.

If all time series in a specific node share the same class, then a leaf node is created and the value of the class label is assigned to the *class* attribute of this node. During classification, if a query instance is to reach this node, it is directly labeled with the value of it's *class* attribute.

When a query time series is to be classified, it starts at the root node of a proximity tree. The distance between the query and each of the randomly selected examplars is calculated, using the randomly selected distance measure at the node. Then the query travels down the branch of the nearest examplar. This processes is repeated, passing through the internal nodes of the tree till the query reaches a leaf node, where it is assigned the class label of that node. This whole process is then repeated for all the trees constituting the forest. The final classification of the forest is made by majority voting between it's trees.

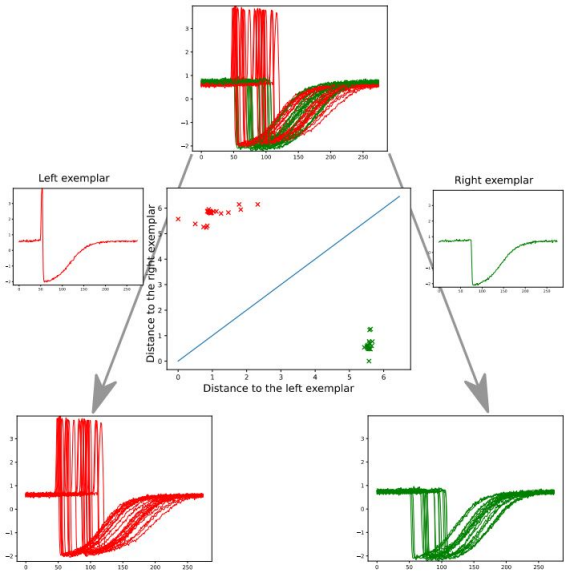


Figure 2.3: Visual depiction of the root node for the 'Trace' dataset (simplified to 2 classes). The top chart represents the data at the root node (one colour per class) while the data at the bottom left and right represent the data once split by the tree. The two time series in the middle left and right are the exemplars on which the tree is splitting. The scatter plot at the center represents the distance of each time series at the root node with respect to the left and right exemplars (resp. x- and y-axes) (Color figure online) [Luc+19].

2.3.2 Phase Dependent intervals Algorithms

Phase dependent algorithms is a group of algorithms that extract temporal features from intervals of time series. These temporal features help with the interpretibility of the model; as they give insights about the temporal characteristics of the data [BR16],

unlike whole time series algorithms that base their decisions solely on the similarities between instances. Another advantage of phase dependent algorithms is that they can also handle distortions and misalignments of time series data [Den+13].

According to [Bag+17], phase dependent algorithms are best used for problems where discrimanatory information from intervals exist, this would be the case with long timer series instances and which might include areas of noise that can easily deceive classifiers. Like the case with the SmallKitchenAppliances dataset, in which the usage of three classes; a kettle, a microwave and toaster is recorded every 2 minutes for one day. Not only the pattern of usage is discrimanatory in such case, but also the time of usage.

Typically using interval features requires a two phase process; first by extracting the temporal features and then training a classifier using the extracted features [Den+13]. There are n(n-1)/2 possible intervals,for a time series of length n[Bag+17]. There is also a wide variety of features, also called literals, to extract for each interval. These cover simple statistical measures as well as local and global temporal features [SK16; RA04; Den+13]. This introduces one of the main challenges for phase dependent algorithms, that is which intervals to consider for the feature extraction step. Which [RA04] proposed a solution for by only considering intervals with lengths equal to powers of two [Bag+17].

Time Series Forest

Time Series Forest (TSF) is an algorithm that was introduced in 2013 by [Den+13]. They motivated their model with two main criteria; contributing to interpretable time series classification through the use of simple statistical temporal features and reaching this goal by creating an efficient and effective classifier.

TSF considers three types of interval features; mean, standard deviation and slope. If we were to consider an interval with starting point t_1 and with ending point t_2. Let v_i be the value at a specific point t_i. Then the three features can be denoted as:

$$mean(t_1, t_2) = \frac{\sum_{i=t_1}^{t_2} v_\mathrm{i}}{t_2 - t_1 + 1} \tag{2.2}$$

$$std(t_1, t_2) = \begin{cases} \frac{\sum_{i=t_1}^{t_2} (v_\mathrm{i} - mean(t_1, t_2))^2}{t_2 - t_1} & \text{if } \mathrm{t}_1 < \mathrm{t}_2 \\ 0 & \text{if } \mathrm{t}_1 = \mathrm{t}_2 \end{cases} \tag{2.3}$$

$$slope(t_1, t_2) = \begin{cases} m & \text{if } \mathrm{t}_1 < \mathrm{t}_2 \\ 0 & \text{if } \mathrm{t}_1 = \mathrm{t}_2 \end{cases} \tag{2.4}$$

Where m denotes the slope of the least squares regression line for the training dataset.

For building the trees, TSF introduced a new splitting criteria at the tree nodes, which they called the Entrace. A combination of Entropy and distance; to break the ties between features of equal entropy gain by preferring splits that have the furthest distance to the nearest instance. They also use a specific number of evaluation points rather than checking all split points for the highest information gain. In their experiment [Bag+17] found these two criteria to have negative effect on accuracy.

As mentioned earlier the feature space for creating interval features is huge, TSF adopts the same random sampling technique that Random Forests use reducing the feature space from $O(M^2)$ to only $O(M)$, by considering only $O(\sqrt{M})$ random interval sizes and $O(\sqrt{M})$ random starting points at each tree node [Den+13]. The final

classification of a testing instance is done using majority voting of all time series trees created.

2.3.3 Phase Independent intervals Algorithms

Phase independent intervals, or just shapelets as less formally known, are subseries which are ultimately distinctive of classes regardless of their place on the time series [SL17a; Bag+17]. They were first introduced in [YK09] as an alternative for KNN approaches; to overcome their shortcomings.

Shapelets reduce the space and time complexity needed by KNN, because they are formed from subsequences which are shorter than the original time series. Needing only one shapelet at classification time, they form a compressed format of the classification problem [BB17a; YK09; MKY11]. While KNN classify based on comparison to other instances, shapelets provide insight about the unique features of classes and thus more interpretable results of how the classification was carried out. Finally, shapelets are best suited for problems where a certain pattern can differentiate instances which is harder to detect when comparing whole series [Bag+17; BB17b]. Figure 2.4 shows how shapelets can separate instances by calculating distances and provide interpretable explanation by projecting the data in a shapelet transform space.

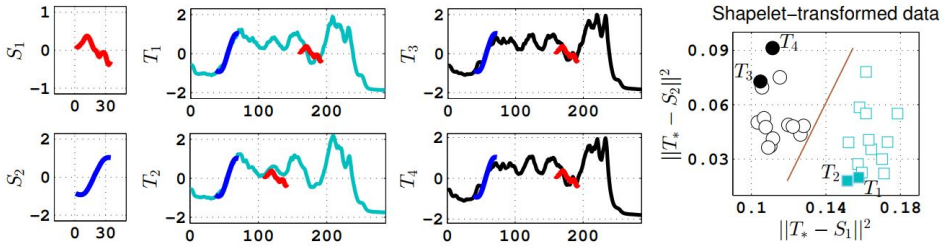


Figure 2.4: An illustration of two shapelets S1, S2 (leftmost plots) learned on the Coffee dataset. Series' distances to shapelets can optimally project the series into a 2-dimensional space, called the shapelet-transformed representation [Lin+12] (rightmost plot). The middle plots show the closest matches of the shapelets on series of two classes having light-blue and black colors [Gra+14].

The original shapelet algorithm enumerated all possible shapelets and embeded the best ones, based on information gain assessment, in a decision tree. Together with a calculated distance threshold, the shapelets and the threshold are used together as splitting criteria [LTB18; Sch15]. There have been many attempts to speed up the process of shapelets discovery, by determining good shapelets in a faster manner. Two of them are; Fast Shapelets (FS) [RK13] and Learned Shapelets (LS) [Gra+14]. FS applied discretization through Symbolic Aggregate Approximation (SAX) to reduce the length of time series, while LS tried to learn the shapelets [Shi+20]. Later on the idea of tranforming time series data to an alternative space was adopted in [Hil+14], the transformed data consists of distances to the best k shapelets, then classification is done using an ensemble of eight classifiers.

Learned Shapelets

Learned Shapelets (LS) was proposed in [Gra+14] as a new prospective for approaching time series shapelets. Instead of searching for shapelets through enumeration of all candidates, LS learns K near-to-optimal shapelets that can linearly separate instances through a stochastic gradient objective [LTB18; Bos18]. The found shapelets need not to be a subsequence of one of the training examples [Bag+17; SL17a].

LS follows a two steps technique. In the begining LS looks for a set of shapelets from the training data set using two parameters; L controls the length of shapelets searched, while R controls the scaling of subsequences. Then these shapelets are clustered using a K-Means algorithm and instances are represented in a new K-dimensional format where the values of the features represent the minimum distance between the instance and one of the shapelets.

For the second step, using the new features representation, LS can start learning class probabilities for instances by considering a logistic regression model for each of the classes and optimizing a regularized logistic loss function. The regularized loss function updates the shapelets and the weights of features. This process keeps iteratively going untill either the model converges or the maximum number of iterations is reached.[Bos18]. In summary, the main objective of the aglorithm is to learn collectively the optimal shapelets and the weights linear hyper-plane that minimizes the objective function [Bag+17; Gra+14].

Shapelet Transform

The first Shapelet Transform (ST) was introduced in [Hil+14]. While the original algorithm embeded shapelets discovery in decision trees and assesed candidates through enumeration and the use Information Gain (IG) at each node, ST proposed a different way that saved repeating the brute force multiple times [Bos18]. ST seggregated the shapelets discovery process from the classifier. This seggregation opened the door for choosing classifiers freely and considering more accurate classifiers than decision trees [Bag+17; LB15]. Also [Hil+14] experimented with other shapelet assessment metrics like Kruskal-Wallis, F-stat and Mood's median to find out that F-stat attained higher accuracies than the other three and than IG [Bos18].

ST follows a three step procedure. In the beginning, a data transformation phase is carried out by utilizing a single-scan algorithm and extracting K best shapelets from the training data set where K represents a cutoff threshold for the maximum number of shapelets to extract without affecting the quality of the shapelets extracted. Then a reduction process is done by clustering the shapelets together untill they reach a user defined number. Finally, The clustered shapelets are then used to transform the original datasest, by representing instances in terms of their disatances to each one of the extracted shapelets. They experimented with different classifiers other than decision trees, these are; C4.5 tree, 1-NN, naive Bayes, Bayesian network, Random Forest, Rotation Forest, and support vector machine, for which decision trees proved to be the worst among all, while support vector machine proved to be the best [Hil+14].

ST was then extended again by [BB17b], the intuition behind it was that the previously used assessment technique couldn't hanlde multi-class problems [BB17b]. Instead of assessing shapelets that discriminate between all classes, they accomodated a one-vs-all technique so that shapelets are assessed on their ability to separate one class to all other classes. They also introduced a balancing technique to represent each of the classes with the same number of shapelets [Bag+17]. For the classification, a

combination of tree based, kernel based and probabilistic classifiers were used in an ensemble on the transformed data set [Shi+20; LTB18]. Each of the classifiers was given a weight based on it's training accuracy and the final classification used weighted voting [BB17b]. Although ST has proved to be a competent accurate classifier, it suffers from high training-time complexity [Shi+20].

ST is one of multiple classifier in the bigger ensemble HIVE-COTE [LTB18]. It also is among the slowest components of the ensemble; due to the exhaustive shapelet enumeration which reaches a complexity of $O(n^2 \cdot l^4)$, where n is the size of the data set and l is the length of the time series [Shi+20]. During a modification that was introduced in [Bag+20] for HIVE-COTE, some enhancements were applied to ST. Some of these enhancements were new, others were already suggested before. The first enhancement was to add a time contract; so that it limits the exhaustive enumeration process to happen only during the contracted time. The value of the time contract is configured using a new added parameter. The second enhancement was to train a Rotation Forest classifier on the transformed data instead of the mixed ensemble.

In the beginning of the training process, ST explores the number of shapelets that exist in the data set during the contract time, then approximates the number of shapelet samples it can acquire from each time series. Based on the results, it modifies the time of the model using a reinforcement learning technique. For the shapelets sampling process a random search is executed. If the time contract is long enough, then all shapelets are considered. Then, an evaluation process starts, where all the acquired samples are compared and filtered; to exclude redundant shapelets. The filtered samples are then evaluated using information gain (IG); to keep the best distinctive shapelets of the classes. For multi-class problems, a one vs all evaluation is carried out as per the suggestion by [BB17b]. After that the weak shapelets are excluded and the remaining shapelets are merged into a pool and the transformation process starts. Finally, the rotation forest classifier is trained. In our framework, we use a contractable version of ST instead of the original. The implemented version allows for specifying a contrat time and uses a random forest classifier.

2.3.4 Dictionary Based Algorithms

As previously discussed, shapelet algorithms are very good at identifying unique subseries which can be related to class labels either by their presence or absence. This technique although very helpful for some problems, but it can deceived if the class labels are defined by the relative frequency of some patterns and not just the a single match [Lar+18; Bag+17]. Dictionary based algorithms is a family of algorithms that uses the frequency of words as the foundation of finding discriminatory features of classes [MVB19]. They can address this type of problems by tracking the frequency counts of repeating patterns, which makes same resilient to noisy data [Shi+20]. To achieve this, dictionary based algorithms transform time series data into words; to reduce their dimensionality and approximate their values. Then based on the distribution of the words they measure similarity between instances [LKL12; Sch15].

In general, the process of dictionary based classifiers involves using a sliding window, of some length w, over the data. Then for each window, an approximation process is carried out to reduce it's length to from w to a shorter length l [Bag+20]. A quantization process then follows; to discretize the real values into words by assigning letters to each value. When all the windows are finished, the number of occurences of words is calcualted and each time series is represented by a histogram [Shi+20]. In the end, similarity between instances is calculated based on similarity between histograms.

Differrent dictionary based algorithms apply different approximation and discretization techniques. For example two of the older algorithms; Bag of Patterns (BoP) [LKL12] and Symbolic Aggregate Approximation-Vector Space Model (SAX-VSM) [SM13] use an approximation method called Symbolic Aggregate Approximation (SAX) [Lin+07]. SAX extends Piecewise Aggregate Approximation (PAA), a method that divides a time series into equal consecutive segments and concatenates their means [Shi+20] Then to transform the series to words, it creates break points using quantiles from the normal distribution. In addition to using SAX, SAX-VSM merges it with the vector space model from the Information Retrieval domain. The major difference between BOP and SAX-VSM, is that SAX-VSM calcualtes it's frequencies on the class level rather than on the series level. Then applies weighting using inverse document frequency [Bag+17].

On the other hand, the newer algorithms; BOSS [Sch15], BOSS-VS [Sch16] and WEASEL [SL17a] use another method which is called Symbolic Fourier Approximation (SFA) which approximates data using coefficients of Discrete Fourier Transformation (DFT). Then apply an adaptive technique called Multiple Coefficient Binning (MCB) to quantize the data. These two techniques; SFA and MCB, will be discussed in more details in the next parts.

BOSS

Bag of SFA Symbols (BOSS) is a dictionary based algorithm that was introduced by [Sch15] in 2015. Like the previous dictionary based classifiers, BOP and SAX-VSM, BOSS also used a windowing technique to extract patterns from the data and transform them into words, but it had other significant differences to them [Bag+17]. BOSS was concerned with the issue of dealing with noisy data which, at that time, received little attention; due to the common practice of either using raw data directly or handling noise in a preprocessing stage. The goal was to introduce an algorithm faster than it's rivals of the same group, robust for existence of noise in the data and competitive with existing TSCA.

The BOSS model is divided into several steps. In the beginning, it passes a window of size w over the time series to extract substructures. Each of the extracted windows is then normalized to obtain amplitude invariance, while obtaining offset invariance by subtracting the mean value for each window is data set specific and can be decided upon based on a parameter. Then the substructures are quantized using SFA transformations which transforms the data into unordered words; this helps further reduce the noise in the data by making it phase shift invariant and makes it possible to apply string matching algorithms on the data. Since some data sets might happen to have long constant signals, this will lead to SFA transformations of their windows to produce the same words multiple times and cause higher weights to be assigned for them. BOSS applies a numerosity reduction technique adopted from [Lin+07; LKL12] that ignores multiple sequential occurences of the same word. Finally time series instances can be compared for differences, using the noise reduced substructures using a customized distance measure inspired by Euclidean distance. We will, briefly, discuss the main components that are used for each of the previously mentioned steps.

To extract the substructures from a given time series instance $T = [t_1, t_2, ...t_n]$, a windowing function is used to split it into fixed sized windows $S_{i;w} = (t_i, , ...t_{i+w-1})$, each of them is of a size w. The total number of windows that can be created for a time series of length n is $n - w + 1$, and each consecutive windows overlap on $w - 1$ points. In order to achieve offset and amplitude invariance, each window is z-normalized by

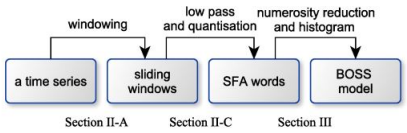


Figure 2.5: The BOSS workflow [Sch15]

subtracting the mean from it's original values and then dividing the difference by the standard deviation.

$$windows(T, w) = \{S_{1;w}, S_{2;w}, ..., S_{n-w+1;w}\} \tag{2.5}$$

After running the windowing function, the real values of the time points inside the windows are transformed into words using SFA [SH12]. SFA is an alternative way of representing time series data, that instead of using real values, uses a sequence of symbols which are referred to as SFA words based on a predefined alphabet of specific length. SFA accomplishes two things; low pass filtering through removal of noisy components from the data and string representation which allows for the use of string matching algorithms.

For SFA to achieve it's goals, two main operations have to be carried out; approximation and quantization. Approximation is the process of representing a specific signal of length n using another signal of lower length l. This is achieved using Discrete Fourier Transformation (DFT); a transformation technique which is applied to a singal represented by a sequence of equally spaces values into a sequence of coefficients of complex sinusoid waves ordered by their frequencies [LLY17]. The higher coefficients in a DFT refer to data with with rapid changes, which can be considered as noise in the signal and thus ignored. So considering only the first $l \ll n$ coefficients acts as a low pass filter for noise producing a smoother signal.

Quantization also helps reduce noise by splitting the frequency domain into equi-depth bins, then maps each of the Fourier real and imaginary coefficients into a bin. BOSS utilizes Multiple Coefficient Binning (MCB), an adaptive technique that minimizes the loss of information which is a side effect of quantization. After the approximation step, a matrix is is built from the Fourier transformations of the training data set using only the first $\frac{l}{2}$ coefficients, including both the real and imaginary values for each coefficient. Then using an alphabet Σ of size c, MCB creates for each columm of the matrix c+1 break points using equi-depth binning. During classification, to acquire the SFA word for a Fourier transformed time series, a lookup is done on the precomputed MCB bins and a word is assigned if the value falls within it's bin's boundaries.

After the transformation of instances to SFA words, BOSS uses a customized distnace measure referred to as BOSS distance; to measure the similarity between the transformed instances. BOSS distance is a variation of Euclidean distance, which compares instances based on the histograms formed from their transformed forms. The appearence of the same SFA words in both instances, is considered to be a notion of similarity. While the absence of SFA words might be caused by one of two reasons; the absence of some substructures from either instances, or due to the presence of noise which disfigures the time series. Instances under BOSS distance are compared

based on shared SFA words only, thus excluding words of frequencies equal to 0. BOSS distance as noted by [Sch15] is:

Given two time series instances T_1 and T_2 and their corresponding BOSS histograms $B_1 : \Sigma^l \to \mathbb{N}$ and $B_2 : \Sigma^l \to \mathbb{N}$, where l is the word length and Σ is an alphabet of size c. Their BOSS distance is:

$$D(T_1, T_2) = dist(B_1, B_2) \tag{2.6}$$

where

$$dist(B_1, B_2) = \sum_{a \epsilon B_1; B_1(a) > 0} [B_1(a) - B_2(a)]^2 \tag{2.7}$$

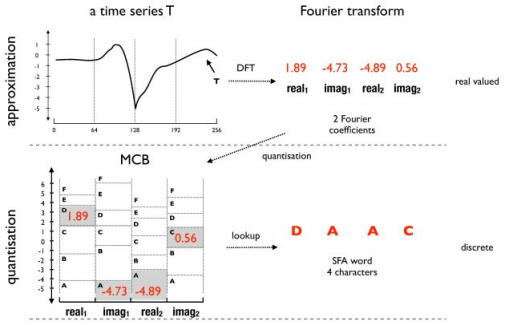


Figure 2.6: A time series is (a) approximated (low pass filtered) using DFT and (b) quantised using MCB resulting in the SFA word DAAC [Sch15]

Each single BOSS classifier utilizes 1-NN approach along with it's BOSS model. The reason behind using 1-NN is that it is a simple technique that doesn't add to the parameters of the model, but rather proved to be a robust one. When classifying a query isntance, BOSS searches for the nearest neighbor in a sample of candidates by chossing the closest instance based on the BOSS distance.

Finally, BOSS Ensemble is introduced as an ensemble of multiple BOSS classifiers. While a fixed window length is used over time series instances for BOSS classifier, BOSS Ensemble considers representing each time series instance by ensembling multiple BOSS classifiers, each of a different window length. When the training data is fitted, BOSS Ensemble acquires a group of scores for each of the different window lengths. To classify a query instance, BOSS Ensemble acquires the best accuracy score from the score sets returned during training. Then considers all window lengths that obtained accuracies within a factor of the best accuracy score. Each of the considered windows predicts a class label for the query instance using 1-NN. Majority voting is applied and the most dominant class label is assigned.

In [MVB19], some enhancements were introduced to BOSS. These included randomising the parameter settings, subsampling training data for each classifier, adding a voting scheme to assign weights to classifiers based on the probability distribution from cross validation on training dataset, fixing the number of classifiers and adding a time contract.

The new enhanced version of BOSS is named the Contractable BOSS or CBoss. In the contractable BOSS, the ensemble is assigned to build as many individual BOSS

classifiers as possible during a specified duration of time (the contracting time) despite the properties of the data set. In cases of small data sets, this might require huge memory resources; as the number of possible classifiers to create would increase. This was overcome by introducing a parameter to limit the number of maximum classifiers to create. On the other side, for huge data sets there is a chance of building problems to happen and the loss of progress. Which was overcome by adding checkpoints to periodically save the current state of the classifier.

Multiple experiments were carried out using different variations of the enhancements. None of the enhancements alone proved competence to the original BOSS performance, but a combination of them together made it possible to achieve significant results. More details about the experiments an be found in [MVB19]. In our framework, we use CBoss as a replacement for the original BOSS; as recommended by [MVB19] and already replaced in the new improved version of the ensemble HIVE-COTE in [Bag+20].

WEASEL

After the success that BOSS achieved, it became one of the baseline classifiers of TSC. Other reasearch has either included it as a building component of their ensembles [LTB18; Bag+15], or have used it, as a baseline for comparison of their algorithm's performance [Faw+20; Shi+20; Luc+19] Later on, the same team that developed BOSS introduced a newer algorithm, Word ExtrAction for time SEries cLassification (WEASEL) [SL17a]. A dictionary based classifier which is very similar to BOSS and can be thought of as an extension to it, with more focus on scalability [MVB19]. WEASEL motivated their work with the absence of scalable classifiers, at the time, that can deal with big data sets; as the existing were either not scalable enough or not accurate enough. Despite being a non-ensemble classifier, WEASEL can compete with powerful ensembles in terms of accuracy without the need for long prediction times, but this comes with the cost of resource expensive training [MVB19].

Like the other dictionary based approaches, WEASEL uses a sliding window over the data instances to extract substructures. These substructures are then discretized per window to extract features and finally a machine learning classifier is learned over the transformed features. But WEASEL differentiates itself from the other algorithms in the way it constructs and filters the features. We will discuss the new approaches that WEASEL uses for extracting discriminative features, building a model to deal with multiple occurences of words and variable windows' lengths and selecting features to reduce runtime and exclude irrelevant features.

In the beginning, WEASEL uses multiple sliding windows of different lengths; to extract substructures. While keeping track of their order; in order to use them later on as bi-gram features. This replaces the process of building multiple models then choosing the best one, with building only one model which can learn from the concatenated high-dimensional feature vector. Then each of the substructures is normalized and a DFT is applied. Instead of filtering out the higher Fourier coefficients, WEASEL applies an ANalysis Of VAriance (ANOVA) F-test to keep real and imaginary Fourier values that best separate instances from different classes. Then the kept coefficients are discretized into words, based on alphabet of size c, using binning boundaries. Instead of using equi-depth binning, WEASEL applies an information gain based binning technique; which further more helps separating instances of classes. In the end, a histogram is built using all the windows lengths and the extracted features, unigrams and bi-grams. Irrelevant features are then filtered out using a Chi-squared test.

The final highly discriminative feature vector produced is then used to learn a logistic regression classifier.

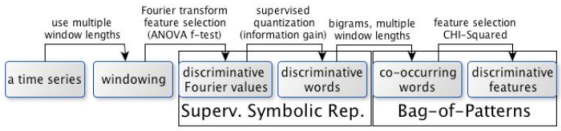


Figure 2.7: WEASEL Pipeline: Feature extraction using a novel supervised symbolic representation, and a novel bag-of-patterns model [SL17a]

WEASEL, like BOSS, transforms extracted windows from a time series into words using an alphabet of size c. They identify two main drawbacks for SFA, and introduce a new supervised symbolic representation technique which is based on SFA but overcomes the drawbacks. The first drawback of SFA is that it acts like a low pass filter and excludes the high frequency components from the Fourier transformation, which might discard important features in some scenarios. The other drawback, is that SFA defines the boundaries of bins during quantization independent of the class labels; which might cause SFA words of equal frequencies to appear unnecessarily in multiple classes. In order to over come the drawbacks of SFA, WEASEL follows two steps; discriminative approximation using ANOVA F-test and discriminative quantization using information gain.

As mentioned earlier, approximation involves representing a time series of length n with a shorter, yet informative, representation of length l by applying a Fourier transformation. WEASEL aims at keeping the best class separating l, real and imaginary, Fourier coefficients by applying a one-way ANOVA F-test. The test verifies the hypothesis that the means of two or more distributions/groups differ from each other [Low14]. This can be tested by comparing two variances; the variance between groups and the variance within the groups. Using the notations in [SL17a], *mean square between* (MS_B) and *mean square within* (MS_W) respectively. The F-values is then calculated as :

$$F = \frac{MS_B}{MS_W} \tag{2.8}$$

Then l coefficients with the highest F-values are kept; as these represent features which have big differences between the classes (high MS_B) and small differences within the same class (low MS_W).

After that, discretization is carried out to set the split thresholds for each of the extracted Fourier value. Discretization involves a binning process, where each Fourier value is divided into a number of bins. Each bin is represented by an upper and a lower boundary, and assigned a letter from an alphabet c. Previous quantization techniques used equi-depth or equi-width binning, but these techniques are solely based on values and ignore the distribution of classes. WEASEL applies a binning technique based on IG; assuring that for each partition the majority of the values would belong to the same class.

The result of the feature extraction phase is a high dimensional feature vector with a dimensionality of $\mathcal{O}(min(Nn^2, c^l)$, where c is an alphabet, l is the length of a word, n is the total number of instances and n is the length of the time series. Since WEASEL uses bi-grams and $\mathcal{O}(n)$ window lengths, the dimensionality of the feature

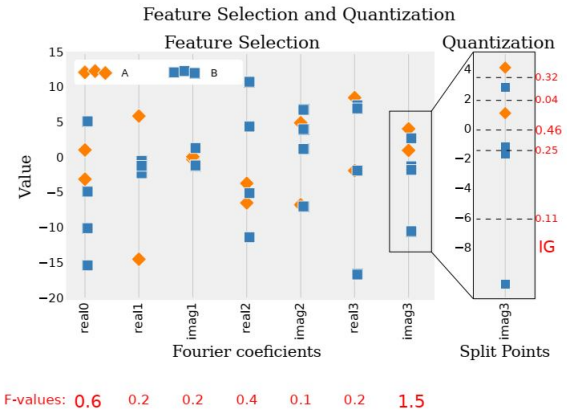


Figure 2.8: On the left: Distribution of Fourier coefficients for a sample dataset. The high F-values on imag3 and real0 (red text at bottom) should be selected to best separate the samples from class labels 'A' and 'B'. On the right: Zoom in on imag3. Information gain splitting is applied to find the best bins for the subsequent quantization. High information gain (IG) indicates pure (good) split points [SL17a]

space increases to $\mathcal{O}(min(Nn^2, c^{2l} \cdot n)$. This enormous feature space is then reduced using a Chi-squared (χ^2) test. Chi-squared tests is a statistical test used to determine if there is a significant difference between the recognised frequencies and the expected frequencies of of a features within a group. This implies that features which have high (χ^2) values are statistically frequent in certain classes. By comparing the (χ^2) values of features to a threshold, all features with scores lower than the threshold can be excluded from the faeture space. This usually reduced the feature space by 30% - 70% and helped train the logistic regresion classifier in a timely manner.

2.3.5 Deep Learning Algorithms

Deep learning is a long established group of machine learning algorithms which has proved competence in many areas [LBH15] and which have encouraged the introduction of deep learning algorithms for time series classification [WYO17]. Deep learning is appealing for investigating time series data; due to the role that the time dimension play as a structure for the data and because deep learning algorithms can scale linearly with training size [Shi+20].

The general framework of deep learning neural networks is described by [Faw+19a] as an architecture of L layers, each one of them is considered an input domain representation. Each of the layers consists of small computing units referred to as neurons. Neurons compute elements for the output of each layer. A layer l_i applies an activation function to it's input then passes the output to the next layer $l_{\mathrm{i}+1}$. The behavior of activation functions in each layer is controlled by a set of parameters θ_i and are referred to by the term weights. The weights are assigned to the links between the inputs and outputs of the network's layers. A neural network carries out a series of computations to predict the class label for a given input x, these calculation are noted

as:

$$f_{\mathrm{L}}(\theta_{\mathrm{L}}, x) = f_{\mathrm{L\text{-}1}}(\theta_{\mathrm{L\text{-}1}}, f_{\mathrm{L\text{-}2}}(\theta_{\mathrm{L\text{-}2}}, \ldots, f_1(\theta_1, x))) \tag{2.9}$$

Where f_{i} represents the activation function applied at layer l_{i}.

In the training phase of a neural network, the weights are randomly initialized. Then a forward pass of the input x is done through the model and an output vector is computed, in which every component of the vector represent a class probability. Using a cost function, the model's predicion loss is calculated and weights of the network are updated, in a backward pass process, using a gradient descent. By continuous interation of forward passes and weight updates by backward passes, the neural network learns the best weights to minimize the cost function. For the prediction of unseen instances, or inference phase, the input data is passed through the network in a forward pass. Then using the class probabilities of the output vector, the class with the highest probability is assigned.

Many of the deep learning research on time series focused on the use of variants of Convolutional Neural Networks (CNN). [Faw+19b] CNN model is based on the idea of learning convolutional filters that can accurately represent the data. Using these filters, CNNs can learn the hierarchical structure of the data while incorporating translation invariance[LMT16]. Multi-Scale Neural Networks (MCNN) [CCC16] and LeNet [LMT16] were among the first experiments of using CNN in time series classification. MCNN was composed of three stages; transformation stage, local convolution stage and full convolution stage. The transformation stage aimed at applying a low pass filter to exclude noise and to capture different temporal patterns. Local convolution is a stage where different scale features are extracted and local features of the time series are learned. In the final stage, the full convolution stage, all the features are concatenated and the final prediction is made. On the other hand, LeNet consists of 2 convolutional layers, after each one a max pooling is carried out to do sub-sampling. For the first layer, 5 filters are applied and the max pooling size is 2. While for the second there are 20 filters and max pooling size is 4. In [WYO17] MultiLayer Perceptrons (MLP), Fully Convolutional Networks (FCN) and the Residual Networks (ResNet) were tested on 44 datasets from the UCR time series archive, where FCN was not significantly different from state of the art and MCNN was not significantly different from COTE and BOSS [SL17a].

A recent more comprehensive review of deep learning algorithms was done by [Faw+19a]. In which an experiment between nine deep learning algorithms was done. These included the classical MCNN and LeNet, in addition to the three algorithms from the previously mentioned experiment; MLP, FCV and ResNet. The other algorithms included were Encoder [SPK18], Multi Channel Deep Convolutional Neural Network (MCDCNN) [Zhe+14; Zhe+16], Time Convolutional Neural Network [Zha+17] and Time Warping Invariant Echo State Network (TWIESN) [TH16]. For more details about the algorithms and their structures, please refer to [Faw+19a].

Finally, the newest deep learning algorithm introduced for time series classification is InceptionTime [Faw+20], which is the newest state of the art and which is an ensemble of of deep convolutional neural networks. It can attain high accuracy scores while maintaining scalability. We will discuss in more details the InceptionTime algorithm in the next section.

InceptionTime

InceptionTime [Faw+20] is a recent deep learning algorithm for TSC problems, which is able to achieve high accuracy score [Rui+20]. Motivated by the increasing interest

in deep learning algorithms in the TSC domain along with the need for scalable algorithms that can deal with large data sets, either in number of instances or in length of the time series, and scale to them. InceptionTime is inspired by AlexNet [KSH12], an algorithm that was considered a breakthrough for deep learning algorithms [Alo+18], and wanted to achieve the same but for the domain of TSC.

InceptionTime is based on the idea that the combination of deep CNN with residual connections, like in ResNet, can attain higher classification performance [Faw+19a] Since CNN has proved competency with image classification, there seemed a potential opportunity to be able to use deeper CNN for time series; since time series data is mainly structured on only one dimension which is time, while images have two spacial dimensions. This opened the door for using more complex models for TSC problems which would be computationally challenging to use for images. The building blocks of an InceptionTime network are called Inception modules, these are were introduced by [Sze+15] and evolved later on to Inceptionv4 [Sze+17]. The InceptionTime classifier is an ensemble of 5 InceptionTime networks each initialized with random weights and are assigned equal weights for the final prediction. We will discuss in more details the structure of an InceptionTime network using the notation mentioned in [Faw+20].

InceptionTime's structure is similar to that of a ResNet, but instead of using three residual blocks, InceptionTime uses only two. Each of the residual blocks in composed of three Inception modules instead of the traditional fully convolutional network. Like residual networks, a linear skip connection exists between the two residual blocks of the network, passing the input from one block to be concatenated with the input of the other; this helps passing information from earlier layers of the network to deeper layers and thus mitigating the vanishing gradient issue [He+16]. After the residual blocks, a Global Average Pooling (GAP) layer exists where the multivariate time series output is averaged over the time dimension. The final component of the network is a conventional fully-connected softmax layer with a count of neurons similar to the count of output classes.

Inside the inception module, there are two main components; the bottleneck layer and the variable length filters. Assuming that the input in a multivariate time series data of M dimensions. The job of the bottleneck layer is to transform the data from having M dimensions, into a multivariate data set having m dimensions, where $m \ll M$. This is done by passing a group of sliding filters m with length 1 and a stride of size 1. Which will substantially reduce the dimensionality of the time series data and consequently will also decrease the model's comlexity making it more robust for overfitting problems on data sets of small sizes. The other benfit of including the bottleneck layer, is that it allows utilizing longer filters on the data than the original ResNet using approximately the same number of parameters; due to the lower number of dimensions that the filters will have to deal with.

The output of the bottleneck layer is then passed for a set of variable length filters, the second component of the network, of length l where $l \epsilon$ 10, 20, 40. In addition to the filters, a parallel MaxPooling operation is carried out followed by a bottleneck layer; to make the model robust to small data noises. The final multivariate output is then formed by concatenating the output from each filter based convolution along with the output of the MaxPooling operation. This whole process is executed for each Inception module in the network.

In the end, an InceptionTime network is able to learn the underlying hierarchical structures of a time series by stacking multiple Inception modules and learning from the different filter sizes, which had been learned during training, inside them.

The final InceptionTime classifier is an ensemble of 5 InceptionTime networks.

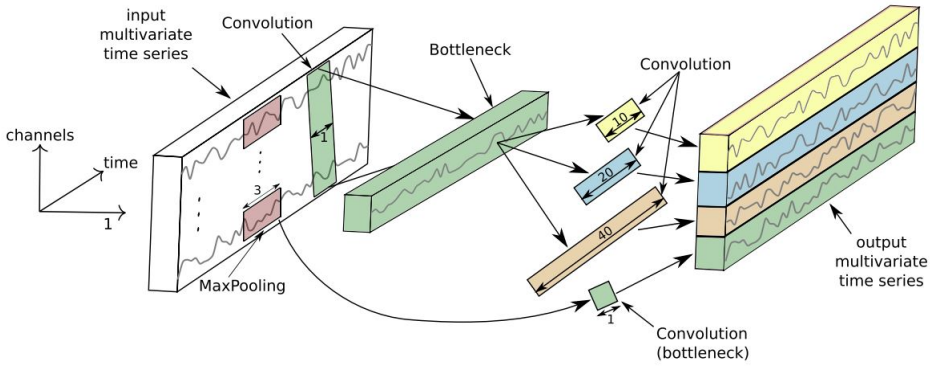


Figure 2.9: Inside Inception module for time series classification. For simplicity a bottleneck layer of size m = 1 is used [Faw+20]

InceptionTime networks accuracy scores showed high variances, a problem which have been discussed by [SW17] and was found in ResNet networks as well [Faw+19b]. This happens due to the random initialization of the networks and due to the stochastic approach used for optimization. InceptionTime classifier follows an ensembling technique for neural networks to handle TSC [Faw+19b] in order to leverage the high variability; adhering to the idea that combining multiple networks would yield better results than one classifier. During the classification of an instance, InceptionTime combines the logistic output of the 5 networks and assigns an equal weight to each of them. This can be denoted by the equation

$$\hat{y}_{\mathrm{i,c}} = \frac{1}{n}\sum_{j=1}^{n} \sigma_{\mathrm{c}}(x_{\mathrm{i}}, \theta_{\mathrm{i}}) | \forall_{\mathrm{c}} \epsilon [1, C] \tag{2.10}$$

Where $\hat{y}_{\mathrm{i,c}}$ is the class probability for instance x_{i} as belonging to the class c, which is the averaged logistic output σ_{c} over the randomly initialized models n.

2.4 Early Time Series Classification

On another side, Early Time Series Classification (eTSC) is also a classification problem which considers the temporal nature of data, but with a focus on slightly different objectives than TSC.

eTSC's main objective is to learn a model which can classify unseen instances as early as possible, while maintaining a competitive accuracy compared to a model that uses full length data or compared to a user defined threshold[XPP09]. Which is a very challenging objective; due to the, naturally, contradicting nature of earliness and accuracy. In general, the more data is made available for the model to learn the better accuracy it can attain [Mor+19; TM16; XPP12; Mor+17b]. This is why eTSC is considered as a problem of optimizing multiple objectives.

We have already defined accuracy for TSC in section 2.2. For earliness, we follow the definition mentioned by [SL20]; as the mean number of data points s after which a label is assigned.

Definition 6

$$Earliness = \frac{\sum_{Ti \in D} \frac{s}{len(T_i)}}{|D|}$$

There are multiple ways to compare algorithms for eTSC based on the two objectives accuracy and earliness. For instance, one could fix the value of earliness and compare accuracies of algorithms at a defined early point of time, fix a certain accuracy value to be achieved and compare how early classifiers can reach it, or combine both accuracy and earliness in one score [SL20]. The $F_\beta - measure$ is an evaluation measure which combines both accuracy and earliness in one equation. It is the calculation of the weighted average between earliness and accuracy. The value of β is the weighting variable; it can be used to give higher importance to either of the aspects over the other. $F_\beta - measure$ definition is:

Definition 7 $F_\beta = (1 + \beta^2) \frac{accuracy\ (1\ -\ earliness)}{\beta^2 accuracy\ +\ (1\ -\ earliness)}$

A special case of the $F_\beta - measure$ is the Harmonic mean (HM) or the $F_1 score$. The HM assigns equal weights for earliness and accuracy by assigning the value of β to 1. It is a popular choice of evaluation for eTSC problems and has been previously used by [GO12] and [SL20]. For the scope of our framework, we will also consider the weighted average $F_\beta - measure$ to evaluate our classifiers.

eTSC is needed in situations in which waiting for more data to arrive can be costly or when making late decisions can cause unfavorable results [Mor+17a; Par+13; Lin+15]. This is why eTSC has been applied in various domains like early medical diagnosis [GM01; GO12], avoiding issues in network traffic flow [Ber+06], human activity recognition [YD19; Gup+20] and early prediction of stock crisis [GRO14].

2.5 Early Time Series Classification Algorithms

2.5.1 ECTS

The idea of eTSC evolved over time. In the beginning, similar concepts to eTSC existed where the main goal was to classify prefixes of data, but these techniques lacked the reasoning of how to provide reliable classification results[Lv+19; SK16]. One of the earliest papers that formally defined eTSC as a tradeoff between earliness and accuracy was the paper by [XPP09], proposing a technique which is named after the problem itself; Early Time Series Classification (ECTS). The main idea was to examine the stability of of Nearest Neighbor approaches, to keep correctly classifying data instances, in the space of full length series and the subspaces created from subsequences of the training data set [Lv+19; Mor+17a]. ECTS utilizes the concept of reverse nearest neighbors (RNN) to learn for each training instance the Minimum Prediction Length (MPL). MPL represents the earliest point in time for which a training instance can be used to classify a query instance without loss of accuracy. The definition mentioned by [XPP09] defines MPL as:

Definition 8 *In a training data set T with full-length L, for a time series $t \epsilon T$, the minimum prediction length (MPL for short) of t, $MPL(t) = k$ if for any $l (k \leq l \leq L)$, (1) $RNN^l(t) = RNN^L(t) \neq \varnothing$ and (2) $RNN^{k-1}(t) \neq RNN^L(t)$. Specifically, if $RNN^L(t) = \varnothing$, then $MPL(t) = L$.*

During classification MPL is calculated for each instance (in the same study a relaxed variant is introduced using clusters of instances). As for the classification, as the data

points of the testing instance start to arrive, the classification method tries to find the nearest instance which has reached it's MPL. If there was no suh instance found, then more data points are needed till a trusted classification can be done. ECTS was criticized [He+15; GRO14; Xin+11] because it is based on nearest neighbor approaches; which only provide classification results based on distance without deriving patterns from the data. This means that data users cannot attain insights about the problem using the results.

2.5.2 Shapelet Based Algorithms

A group of methods [Xin+11; GO12; GRO14; He+15] focused on the interpretability of the results obtained by eTSCAs, motivated by domains where not only correct classification is important, but also getting insights and understanding what contributed to the final result. One of these domains would be the medical field, where undesrtanding the reason for a patient's disease is very important than just linking the patient to previously seen patients [Xin+11]. Shapelets provided a good solution for these methods. Since shapelets, as previously mentioned, are defined as unique subsequences which can uniquely identify classes. Early Distinctive Shapelet Classification (EDSC) was a framework consisting of 2 steps; feature extraction and feature seletion. In the feature extraction step, all local shapelets are extracted and a distance threshold is learned. EDSC developed a technique which they called the best-matching-distance (BMD), which consideres a shapelet to be distinctive, by comparing it to all instances in the training data set and considering target classes distribution. If the majority of instances closer to the shapelet belong to a target class, it is considered to be distinctive of this class. In the feature selection step, EDSC selects a smaller set of features based on three main criteria; earliness, frequency and distinctivness. EDSC was later extended in two different directions.

In [GO12] EDSC was extended in the aspect of data dimensionality; as the original EDSC only convered univariate data sets. The extension covered; adapting the feature extraction to discover multivariate shapelets along with calculating distance thresholds for each dimension and using a distance threshold based on information gain, adapting the feature selection step, by modifying the equations calculating the notions of earliness, frequency and distinctivness to multiple dimensions. In [GRO14] EDSC was extended in the aspect of data uncertainty. The main idea was that shapelets use a distance threshold when being compared to instances, if the distance between the shapelet and the instance is less than the threshold, the instance is given the label of the shapelet. On the other hand, since the threshold represents a range, a shapelet is more certain of the class label given to an instance which has a distance closer to 0 than the class label given to an instance falling on the limits of the threshold. Motivated by the interest of uncertainty in the medical field, the goal was to provide a notion of uncertainty along with the interpretability of shapelets. The proposed Modified EDSC with Uncertainty estimates (MEDSC-U) calculated uncertainty as (1 - confidence), where confidence is a numeric value represting two aspects; confidence that the distance between the shapelet and the instance is less than the threshold and confidence that the shapelet can correctly classify the instance. This confidence notion could then be generalized for classes represented by multiple shapelets. The issue of confidence for eTSC had been discussed in other papers like [Par+13] and [Lv+19].

2.5.3 ECDIRE

The paper by [Mor+17b] propsed a new framework; Early Classification of Time Series based on Discriminating Classes Over Time (ECDIRE). There were two goals that ECDIRE wanted to achieve [Mor+17a]. The main goal of the framework was to track the accuracy evolution for a group of probabilistic classifiers across time; to identify safe time points after which predictions can be made. This would allow for avoiding unnecessary checking at each time point. The other goal was to offer an outliers discarding mechanism using a reliability condition. ECDIRE consisted of three training steps. In the first step the framework analyzes the data set and tries to identify the safe points at which classes can be differentiated from each other. This is done by defining time points as percentages of the full length and training, at each time point, a probabilistic classifier. The early points are then defined as those which maintain an accuracy of a desired percentage from that obtained on the complete data [SK16]. In the second step, a reliability threshold is determined using the class probabilities of the assigned class and the second highest class; this helps avoid uncertain classifications or very soon time points. In the last step, the final probabilistic classifiers are trained using the early time points identified in the first step. As a backup plan, if the last early time point did not coincide with the full length of the time series, ECDIRE acts like other methods and builds classifiers on each time point.

2.5.4 TEASER

One of the latest classifiers introduced is the Two-tier Early and Accurate Series classifiER (TEASER) [SL20]. TEASER's main focus was challenging the logic of previous classifiers on determining the point of time at which a classification can be assigned to a time series instance. Previous methods would try to learn from the training data set a fixed prefix needed to be seen before it can start classification; like MPL in ECTS and EDSC, or to learn a safe point like in ECDIRE. These techniques would require the classifiers to always wait till they reach these fixed points of time before they can assign a label. Whereas TEASER was designed to give a classification once confident of it's decision, and not to depend on any fixed time points nor assume that all data instances were recorded at the same starting point. TEASER models the problem of eTSC with a two tier solution. In the first tier, a slave classifier is regularly checking the incoming data and assigning class probabilities. These probabilities are then assessed in the second tier by a master classifier, which either accepts the classification from the slave, or rejects it and decides to wait for more data to arrive. The classifiers are paired through out the whole process, such that for each slave classifier there is a master classifier that reads it's output and assesses it's results. TEASER starts off by defining the number of classifier pairs that will be neeeded, this is calculated by dividing the length of the longest data instance in the training data set by the window length which is a user defined parameter. Which is noted as $S = [n_{max}/w]$, where n_{max} is the length of the longest instance and w is the window length. During training, slave classifiers are trained on subsequences s_i, or snapshots, of the time series where the length of $s_i = i.w$. These snapshots are z-normalized based on the values existing only in them and not on values from the full length data as recommended by [Mor+17a].

The slave classifiers would then calculate three features; the class label with the highest probability $c(s_i)$, the vector of class probabilities for all available classes $P(s_i)$,

and the difference between the class probability of the highest two classes denoted as Δd. These three features are then passed to the paired master classifier, which trains a model over them using a one-class Support Vector Machine (oc-SVM). A parameter that plays an important role in this process is v, it is the number of consecutive times the same class label is given to an instance. Once the same label is given v times for consecutive snapshots, the master classifier is confident of this class label. Figure 2.10 shows an example of how TEASER components handle a misleading data instance from the traces of microwave and a digital receiver.

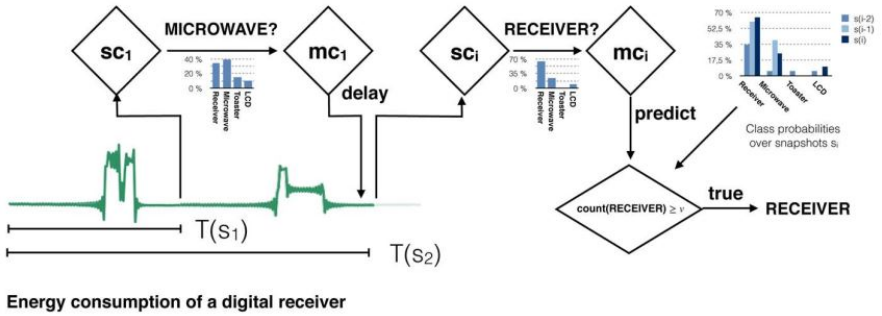


Figure 2.10: TEASER is given a snapshot of an energy consumption time series. After seeing the first s measurements, the first slave classifier sc_1 performs a prediction which the master classifier mc_1 rejects due to low class probabilities. After observing the i-th interval which includes a characteristic energy burst, the slave classifier sc_i (correctly) predicts RECEIVER, and the master classifier mc_i eventually accepts this prediction. When the prediction of RECEIVER has been consistently derived v times, it is output as final prediction [SL20]

Chapter 3

Related Work on Comparing Time Series Classification Algorithms

The main contribution of this thesis is to provide a framework that compares TSCAs in an early classification context. The idea of creating frameworks to standardize the process of comparison between algorithms is not new to the time series data domain. It was actually needed due to the increasing interest in time series data and the vast amount of newly introduced algorithms. In this chapter we will briefly discuss 3 other frameworks that had been developed to compare TSCA on univariate and multivariate data sets.

3.1 The great time series classification bake off

One of the most famous studies in comparing TSCAs is [Bag+17] and which has inspired some of the work done in this thesis. At the time when this paper was published, the TSC domain had already witnessed hundreds of algorithms being proposed, each claiming superior performance. There was no clear structure, at the time, for experimenting and benchmarking algorithms' performance, which pushed for having a more concrete structure to be followed in succeeding research. The primary goal of the experiment was to assess the average accuracy of the different classifiers on a broad set of univariate data sets. The Scalability and Efficiency of the classifiers were considered as secondary goals.

3.1.1 Inclusion Criteria

The experiment collected 18 different classifiers to be compared with 2 baseline classifiers on the UCR data archive (discussed in 5.1.1). Algorithms were selected based on three criteria; an algorithm had to be published in a high impact conference or journal, used some data sets from the UCR archive to assess performance and the availability of the code.

In order to better describe the various techniques applied by the algorithms and what differences characterize each of them, they formulated a taxonomy which groups algorithms based on the features they try to find to discriminate between different classes. These groups were; whole series similarity (also referred to as distance-based algorithms), phase dependent intervals, phase independent intervals (shapelets), dictionary based and ensemble algorithms. We follow the same taxonomy for our experiment as well and describe these groups in more details in section 2.3.

3.1.2 Experimental Design

The framework ran 100 resampling folds for each of the data sets for each classifier, totalling 8500 (85 data sets $\times$ 100 resamples) experiments for each classifier. The first run used the default train/test split provided with the data set, while the remaining resamples were stratified to preserve the class distribution in the default split. Using the same parameters as the relevant published papers, the experiment limited the number of parameter values for each classifier to a maximum of 100. They used different number of cross validation folds based on the type of algorithm. LOOCV was used for algorithms utilizing distance based measures, while for the other classifiers a new model was created for each parameter set. Resulting in a total of 850,000 models per classifier.

3.2 Deep learning for time series classification: a review

Another paper that focused on comparison of deep learning algorithms for TSC was [Faw+19a]. Deep learning algorithms had already proved competency in many application fields including image classification, speech recognition and natural language processing [He+16; SK16; KSH12; Gua+19] and was gaining more popularity in TSC problems [Zhe+14; Zhe+16; Zha+17]. The main goals were to benchmark the performance of deep learning algorithms on TSC problems; as they were less studied than the other algorithms and to provide an open source framework for deep learning on TSC.

3.2.1 Inclusion Criteria

The framework included 9 discriminative model end-to-end deep learning algorithms developed to work on TSC problems. They started by grouping deep learning algorithms into two main groups; generative and discriminative approaches.

Generative models involve an unsupervised step which tries to learn a quality representation of the time series, then this new representation is fed to another classifier in a learning phase. But this family of algorithms was excluded due to it's complexity and incompetence compared to the other group [LGI17; Bag+17].

As for the discriminative models, algorithms that needed preprocessing of features prior to learning were also excluded; to avoid the bias of hand crafted features. The remaining discriminative approaches were; Multi Layer Perceptron (MLP), Fully Convolutional Neural Network (FCN), Residual Network (ResNet), Encoder, Multi-scale Convolutional Neural Network (MCNN), Time Le-Net, Multi Channel Deep Convolutional Neural Network (MCDCNN), Time Convolutional Neural Network (Time-CNN) and Time Warping Invariant Echo State Network (TWIESN).

These 9 algorithms were to be compared to each other and to 7 other classifiers. A group consisting of the best 4, out of the 18, classifiers that were included in the experiment by [Bag+17] which are; Elastic Ensemble (EE), Bag Of SFA Symbols (BOSS), Shapelet Transform (ST), Collective of Transformation-based Ensemble(COTE). They have also included Hierarchical Vote Collective of Transformation-Based Ensembles (HIVE-COTE), which is an extension of COTE using hierarchical voting and two extra classifiers. In addition to Proximity Forest, an ensemble based on the same concept of Random Forests but using class examplars instead of feature split values. Finally, the classic 1-NN classifier utilizing Dynamic Time warping (DTW) elastic distance measure with the warping window set through cross-validation on training data set. Many of these classifiers we cover later on in section 2.3.

3.2.2 Experimental Design

The experiment covered both univariate data sets and multivariate data sets. The univariate data sets were represented by the same 85 z-normalized data sets from the UCR archive as [Bag+17]. As for the multivariate data sets, 12 data sets from the archive by Mustafa Baydogan[1] were used. Due to the existence of instances with unequal lengths in the multivariate archive, they used linear interpolation, suggested by [RK05], such that all instances inside the same data set are adjusted to the length of the longest instance. Z-normalization was not applied to any of the Baydogan archive.

In order to avoid bias from the initial weights assigned to the classifiers, for each of the data sets for each classifier, the framework used 10 runs for training and then considered the mean accuracy of all runs together. The original train/test split was used in all 10 runs, but the initial weights were randomized. The framework applied optimization for the hyperparameters of the deep learning algorithms, but not for the other classifiers. With the exception of TWIESN, the number of epochs used during optimization ranged between 100 and 5000. A model checkpoint procedure was involved. This meant that after training a model for 1000 epochs, the model which attains the least error on the validation data set is chosen for evaluation. All the models were initialized randomly using the methodology from [GB10] and were optimized using variants of stochastic gradient descent; like Adam [KB14] and AdaDelta [Zei12]. For FCN, MLP, and ResNet if the training loss had not improved for 50 consecutive epochs, then learning rate was decreased by 0.5 to a minimum of 0.0001.

3.3 The great multivariate time series classification bake off

A recent study was carried out by [Rui+20] focusing on comparing multivariate TSCAs. With previous research paying more attention to univariate classifiers and univariate problems, multivariate time series classification (MTSC) recieved less attention. Which meant that the MTSC domain is now in the same position that univariate TSC had been in some years ago and there is a need for a benchmarking framework to guide upcoming research on how to compare multivariate TSCAs to already existing ones.

MTSC is simply the classifiation of time series data collected, where multiple features are collected at each time point and a single label is assigned. MTSC is more

[1] http://www.mustafabaydogan.com/multivariate-time-series-discretization-forclassification.html

challenging than univariate TSC because the discriminative features can be in the interaction between the multiple dimensions and not only based on the interdependency of consecutive values in one dimension. The framework inlcuded two techniques that algorithms follow to deal with multivariate data; either dedicated multivariate TSCAs which are, by design, able to handle multiple dimensions, or adapted univariate classifiers which can handle multivariate time series instances. The adapted univariate classifiers technique is based on two pillars; assuming independence between the dimensions of the data and fitting separate univariate classifiers to each dimension then ensemble their results. We used this technique in our experiment for handling multivariate data sets.

3.3.1 Inclusion Criteria

Although univariate and multivariate TSC problems differ in their complexity, but the algorithms for both can be grouped, based on the technique they apply, in the same way. These are; distance based similarity, phase dependent intervals, phase independent intervals (shapelets), dictionary based and deep learning algorithms.

The framework collected 16 classifiers from these different groups and compared them on the 30 data sets from the UEA multivaraite data archive (discussed in 5.1.2). The included classifiers had to meet 2 simple criteria; code availability and runnable code. The classifiers encompassed 3 variations of the classic DTW which represented the baseline; DTW using Independent Warping (DTW_{I}) which calculates DTW distance for each dimension separately then sums all distances, DTW using Dependent Warping (DTW_{D}) which simultaneously calculates the DTW distance across all dimensions for each time point and DTW using Adaptive Warping (DTW_{A}) which adapts to each instance by calculating both DTW_{I} and DTW_{D} and then chooses between them based on a threshold score. The multivariate classifiers were; Generalized random shapelet forest (gRFS), WEASEL+MUSE, Canonical interval forest (CIF), Random Convolutional Kernel Transform (ROCKET), Multiple Representation Sequence Learner (MrSEQL), Residual network (ResNet), InceptionTime, Time series attentional prototype network (TapNet). The adapted univariate classifiers were; the ensemble HIVE-COTE and each of it's single components; Shapelet Transform (ST), Time Series Forest (TSF), Contractable Bag of Symbolic Fourier Approximation Symbols (CBOSS) and Random Interval Spectral Ensemble (RISE). Two specialized toolkits in time series machine learning were used for the experiment. One of them is python based, while the other is java based. Most of the algorithms are implemented in both, but few exist only in one. More information is available in [Rui+20].

3.3.2 Experimental Design

The framework ran 30 resamples for each data set for each classifier to calculate performance metrics, the first run using the default train/test split of the data set, while the remaining 29 runs using stratified sampling; to maintain the class distribution of the original split. In contrast to the previous experiments where only accuracy was measured, this experiment had other measures beside which are; area under the ROC, balanced accuracy, F1, negative log likelihood, Matthew's correlation coefficient, recall/sensitivity, precision and specificity. Still accuracy was the primary performance measure used in comparisons; due to it's easier interpretation.

The classifiers were initialized using the default structure and hyperparameters in their original published papers. Apart from the classifiers that have internal tuning

processes, no other external tunings were applied. Even for the window size of DTW, which is a was proved to have a small but significant improvement of performance for univariate data sets [RK05]. Using a side experiment, where an untuned DTW was compared to a naive implementation of DTW with window sizes between 0% and 100% on 21 of the data sets. The experiment had proved that the untuned DTW was better on 14 data sets, but with no significant difference between both. Which meant that tunning was not needed as far as the experiment is concerned.

The data sets of the UEA archive are not normalized and were presented to the classifiers without any preprocessing. This might had caused a disadvantage for WEASEL+MUSE, which doesn't have an internal normalization process, against other classifiers like, ROCKET; gRSF; TapNet; InceptionTime; ResNet; CBOSS; STC and CIF, which do. The decision of excluding data normalization was based on another side experiment that they carried out. A comparison was done using the three DTW variants, once on normalized data and once on non-normalized data. The experiment had proved that data normalization had no significant effect on the performance of the classifiers, but overall the performance of DTW declined with normalization. The same experiment was repeated for HIVE-COTE and it's components and the same result was observed. This meant that data normalization was also not considered important for the scope of the experiment.

3.4 Evaluation Criteria

All three frameworks follow the same methodology, introduced by [Dem06], for comparing their classifiers over multiple data sets. When using multiple resamples over data sets, the score of a classifier on a given data set is calculated as the average of it's accuracy over all resamples. The methodology recommends using a Friedman ranking test to refute the null hypothesis; that there is no difference between the ranks of the classifiers. After refuting the null hypothesis and applying ranks to each classifier, a pairwise post-hoc analysis is done according to the recommendation of [BCM16]. Which applies a Wilcoxon signed rank test using a Holm's alpha correction of value 5% to form cliques of classifiers. Each clique represents a group of classifiers where there is no significance between their pairwise comparisons. Results are described by a critical difference diagram, where the average rank of each classifier is noted and cliques are expressed by a thick line.

Chapter 4

Framework for Comparing Time Series Classification Algorithms in Early Classification Context

This chapter describes our proposed framework which compares TSCAs in a context that is inspired by the problem of eTSC. We begin by defining what we mean by early classification context and how our framework design simulate such context for TSCAs to operate in. Then we discuss the conceptual structure of the framework, our implementation for its different components and how to evaluate it. In this chapter we will be answering our research questions:

- How to adapt existing TSC workflows for the early TSC context ?
- How to evaluate classifiers in the early TSC context ?
- How to evaluate the proposed solution ?

4.1 Early Classification Context

As motivated earlier, eTSC is a field which is concerned with classification of time series data with earliness and accuracy as the main objectives. Dedicated eTSCAs focus on building models that can learn class labels of the data as early as possible while maintaining accuracy [Mor+17a]. Our framework, on the other hand, investigates the adaptation of the context in which conventional TSCA operate to learn a specific classification problem and the effect it has on the classifiers' performance.

We define our early classification context as the problem in which a conventional TSCA is trained on labeled but incomplete time series data. Then this trained model is used to classify unseen data instances of full length. This context helps investigate if TSCA are able to learn about the problem at hand without having to wait for the full data to be made available. In other words, the benefit acquired from waiting for

more data points to be made available for learning is less than the benefit of having a slightly less accurate result but at an earlier point of time.

4.2 Conceptual Design

To answer our first research question ***How can we adapt existing TS classification workflow for the early TS scenario ?*** We propose a framework that simulates an early classification context by shortening the length of the data before feeding it to TSCAs to learn on. These classifiers are then evaluated by comparing their performance to a baseline represented by their performance on the full length data. In the beginning, the framework divides whole timeseries data instances into smaller subsequences using a chopping algorithm, and then starts multiple learning processes.

For the first learning process, it feeds only one subsequence for the classifier to learn on. Then in each following learning process, the framework adds the next subseuqnce to the training data in addition to the previous ones. These processess simulate an incremental learning scenario, where the classifier is not exposed to all the data points collected from the beginning to the end at once, but rather exposed to the data in a sequential procedure. During each learning process the classifier gets to learn more information about the data instances than the previous process. Finally, to assess how good the performance of the classifiers that were exposed to less data points is, we compare their performance to that of a baseline. The baseline for each classifier is determined by its performance when exposed to full length training data.

Our framework design can divided into two main components; the comparison testbed and the recommender. Each of these components consists of smaller elements which are responsible for specialized tasks which we will discuss in more details. Figure 4.1 shows the conceptual design of the framework and all its costituent components.

4.2.1 The Testbed

The first component is the comparison testbed. It is the main engine of the framework for configuring and executing learning processes. There are multiple units that make up the testbed. These are; the testbed management console, the model preparation agent, the chopper and the experiment executor. Each of these units is responsible for a specific task and communicate information either to the other units in the testbed or to other units in the recommender.

The Testbed Management Console

In order for the testbed to be able to configure its components and experiments, it needs to collect information about the desired shape of experiments. The testbed management console is the interface which the user interacts with to provide all the necessary information. The console allows the user to specify details about the data sets to be used, the configuration of the chopping algorithm and the struture of the experiments to be executed in terms of models specification, training and testing processes design. The testbed management console creates the central configuration which will be followed by all the other elements.

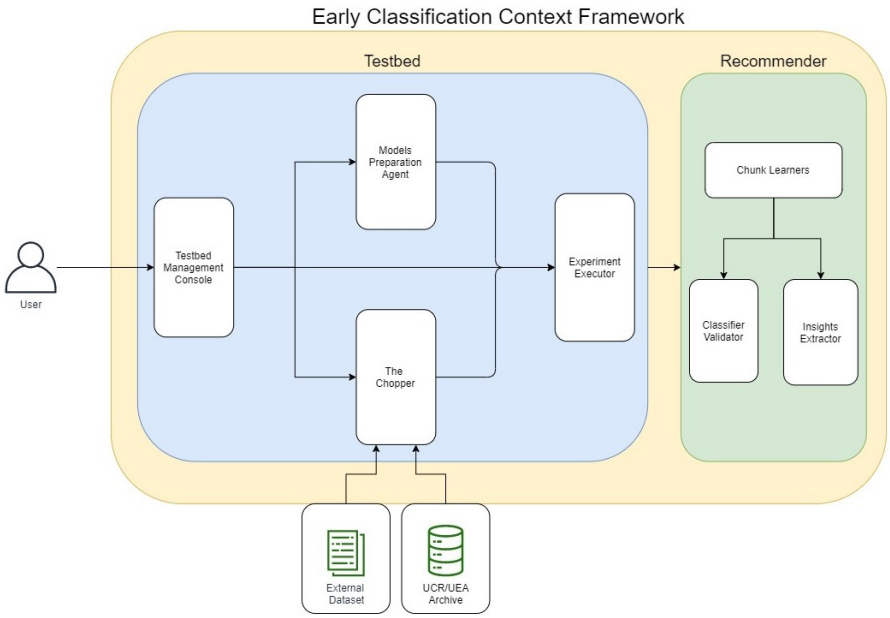


Figure 4.1: Conceptual design of the proposed framework

The Model Preparation Agent

The model preparation agent is responsible for constructing and composing the classification models that will be used in the learning processes. Based on the configuration from the testbed management console, the model preparation agent includes or excludes models from the experiments and supply the hyperparameter space for each model. The configured models are then passed to the experiment executor.

The Chopper

The chopper algorithm is the essence of the early classification context. It is the component which simulates the incremental learning scenario for the classifiers. Based on the configuration from the testbed, the chopper collects information about the different data sets that will be used in the learning processes. Then for each data set, it applies the chopping on its data instances based on their length. The chopped data would then be combined with the algorithm; to start the learning processes.

The Experiment Executor

The last element in the testbed is the experiment executor. It collects the configured models from the model preparation agent along with the chopped data sets from the chopper for the actual execution of the experiments. Based on the central configuration, it configures the environment for the learning proccesses to provide unbiased results, as well as sets the evaluation scheme of the testing process. After the experiments are finished, it provides insights from the results for each process. It also communicates the results and analysis for the recommender to learn on.

4.2.2 The Recommender

The other component of the framework is the recommender. As the name suggests, the recommender is responsible for providing reliable suggestions about the most suitable classifiers to use for unseen data sets. Based on results from the testbed experiments, it learns to predict the performance of different classifiers on data sets in the early classification context. Then when provided with an unseen data set, the recommender would suggest which classifier or multiple classifiers will attain good results if used to learn on incomplete subsequences of data. The recommender consists of three elements; the chunk learners, the classifier validator and the insights extractor.

The Chunk Learners

The chunk learners are a group of classifiers which learn on the results from the experiments of the testbed. The word chunk in the name refers to the chunks (subsequences) that were shown for the classifiers to learn on. Each learner is responsible for learning on the performance of classifiers for a given chunk and consequently predicting the performance of these classifiers for unseen data sets. The prediction results are then passed to the classifier validator and the insights extractor.

The Classifier Validator

The classifier validator is a component which assesses the prediction results from the chunk learners. Its main objective is to decide if the predicted performance results for the classifiers on a data set qualify as good recommendations or not. The classifier validator should exclude classifiers from the recommendations if it evaluates thier performances as weak and not competent. The result from the classifier validator is the actual final recommendations of the framework.

The Insights Extractor

The insights extractor is responsible for analyzing and extracting insights from the results of the chunk learners. It reveals hidden patterns and provides understandable insights about how the performance predictions were made and how recommendations were chosen.

4.3 Implementation

This section describes our implementation for the framework and all its components. In the beginning we discuss our design protocol for building the framework. Then we demonstrate our technical design for each of the components, the selected toolkits for building the components and the flow of information through communication channels that exist between the framework's units.

4.3.1 Design Protocol

Our design protocol covers the main design aspects in the framework like; the selection criteria of the classifiers to be included and the evaluation criteria for assessing the performance of the classifiers.

There is a wide variety of TSCAs that has been introduced in the last decade. Since our goal is to compare TSCAs in the early classification context, we specify an inclusion criteria which would cover this wide spectrum of classifiers. To represent the currently available techniques; we follow the grouping criteria mentioned in [Bag+17] that breaks down classifiers into 5 main groups based on the technique applied. In addition to a 6^{th} group which represents deep learning TSCAs. We represent each of the groups by 2 classifiers; one which is an ensemble and another which is a non-ensemble. The reason we include both ensembles and individual classifiers; is that although for each group there is a classifier which can outperform the others, yet combining classifiers in an ensemble have proven to outperform all of its individual classifiers [Faw+19a].

The other aspect of the framework is to choose an evaluation criteria for assessing the performance of classifiers. Since our framework is inspired by the problem of eTSC; we are interested in evaluating classifiers based on their performance in predicting correct class labels, as well as considering how early they make the prediction. Previous eTSC research have used the harmonic mean of accuracy and earliness as an evaluator for classifiers [SL20]; because it is allows combining both in a single score. The harmonic mean considers both earliness and accuracy to be equally important. In our framework we use F_β; a generalized form of the harmonic mean which allows the assignment of different weights to earliness and accuracy using the parameter β. We prefer using F_β to offer more flexibility in adjusting weights to suit cases where either of the objectives is more important than the other.

4.3.2 Toolkits

We implemented our framework in python language. For the implementation of the non-deep learning algorithms we used the open source libraires *sktime*V.0.5.3 [Lön+19] and *pyts*V.0.11.0 [FJ20], while for the implementation of the deep learning algorithm InceptionTime we used the library *sktimedl*V.0.2.0 which provides an interface for the implementation provided in the study [Faw+20]. Finally, we used the library $Tslearn V$.0.5.0.5 [Tav+20] as a mediator between the different model implementations; due to it's capacity to process raw data into the needed input format by any of the libraries, as well as converting data format between the different libraires.

4.3.3 The Testbed Implementation

The Testbed Management Console Implementation

The Testbed Management Console is the tool that allows the user to interact with the framework and configure the experiments and the learning processes which will be the basis of the final recommendations. At this stage, the user provides the necessary information required by the framework to formulate and start its learning experiments. These information provide the guidelines for the other components in the framework like the model preparation agent and the data chopper. Which are essential for the preparation of data sets and models for the learning processes. Table 4.2 shows a summary of each of the needed information, represented as a parameter for the framework.

We implemented the user input interface using *docopt* [1]. A library which offers command-line interfaces through the use of simple arguments and elements. It also allows setting default values for parameters which can be overriden if a value is passed.

[1] https://github.com/docopt/

Parameter	Data Type	Description
Dataset	String List	The name of the data set(s)
Splits	Integer	The number of splits to apply on the data set
Hyperparam	Boolean	Learn hyperparameters for the model or skip
NumIterations	Integer	Number of hyperparameters combinations to sample
ChunksToKeep	Integer List	The indexes of the chunks to use
FromBeg	Boolean	Reveal data chunks from beginning or end
ScoringFunction	String	The strategy for evaluating the model on test data

Table 4.2: Input parameters required for using the framework

To start running the framework, one provides the necessary parameter values through a simple terminal command. Listing 4.1 shows a sample command for running our master python file *run.py* on the Computers data set using hyperparameters optimization, while applying a split into 10 chunks.

```
python run.py --etsc --dataset=Computers --cv --split=10
```

Listing 4.1: Sample command for providing user input to the framework

The Model Preparation Agent Implementation

The model preparation agent is responsible for preparing the time series classifers and creating their hyperparameter space for each learning process. We design our model preparation agent to adjust the preparation of models to handle univariate and multivariate data sets.

As mentioned in section 2.3, we follow the grouping criteria by [Bag+17] which arranges TSCAs into 5 main categories based on their techniques. We have included classifiers representing each of the 5 groups, in addition to a 6^{th} group which represents the family of deep learning time series algorithms. We tried, as far as the implemented libraries allowed, to represent each group with 2 classifiers; one that is a non-ensebmle and another which is an ensemble. A total of 10 classifiers are included in the framework covering all 6 groups, these can be broken down into the following:

- Distance based : 1NN using MSM distance and PForest
- Phase dependent algorithms : TSF

- Shapelets: LS and ST
- Dictionary based : WEASEL and BOSS
- Deep learning : InceptionTime
- Classical baselines: 1NN using euclidean distance and 1NN using DTW

Table 4.4 shows the configuration and hyperparameters for all the included classifiers. We used the same hyperparameter space like that of the original published papers as closely as possible.

Classifier	**Parameters**
MSM	c = {0.01, 0.1, 1, 10, 100}
PForest	k = 100 r = 5
TSF	r = 500 splitting = {entropy, gini} maxfeatures = {sqrt, log2}
LS	λ_w = {0.01, 0.1, 1} L_{min} = {0.025, 0.075, 0.125, 0.175, 0.2} R = {1, 2, 3} K = {0.05, 0.15, 0.3} η = 0.01 maxIter = {2000, 5000, 10000}
ST	t = 60 mins n={500,100}
CBOSS	t = 60 mins k={50, 100, 250, 500} s=250 p=[0.5 - 1]
WEASEL	ANOVA = {True, False} bigrams ={True, False} binning = {equi-depth, equi-width, information-gain}
Inception	epochs = 1500 batch size = 64 η = 0.001 kernel sizes = {10, 20, 40}
DTW	full warping window

Table 4.4: Parameters and configuration for TSCAs

As mentioned earlier in section 3.3, there are two ways to work on multivariate time series data sets [Rui+20]. The first way is by using bespoke multivariate classifiers

which can deal with the multiple dimensions available in the data. While the other is by using ensembles of univariate classifiers, each fit to a separate dimension under the assumption of independence between the dimensions. Our framework applies the second technique for handling multivariate data sets.

The model preparation agent checks the metadata of the data sets that the management console communicated, then it determines the number of dimensions each data set comprises. If the data set is univariate then the model preparation agent initalizes the default classifer implementations provided by the libraries. In case the data set was found to be multivariate, it initializes a special classifier type which extends the usage of univariate classifiers for multivariate problems. This special classifier is reffered to as $ColumnEnsembleClassifier$ in the *sktime* library implementation and as the $MultivariateClassifier$ in *pyts*.

There are two assumptions that we make at this step in our framework. First, we assume that to extend a specific univariate classifier to a problem, we should fit the same type of the classifer on all dimensions of the data. This is goes back to our goal, that we want to compare classifiers from different groups and not to create an ensemble which makes use of different classifiers applying different techniques, like HIVE-COTE does. Second, we assume that we always have to fit one classifier per dimension, which have proved to be not a feasible solution for high dimensional data sets. We will discuss this in more details in our results.

After the model preparation agent finishes, it passes the initialized models to the experiment executor for the learning processes to actually start.

The Chopper Implementation

The chopper is the data sets handling module of the framework. It is not only designed to apply chopping of the data based on the configuration, but also to load the data sets and preprocess it. The chopper collects information about the data sets to be used, the number of splits (chops) to be applied, the chopping direction and the chops filteration. These information are passed from the management console in the form of the parameters $Dataset$, $Splits$, $FromBeg$ and $ChunksToKeep$ respectively.

The first task of the chopper is to process input data. It is configured to process data sets in either of two ways; automatic download from the UCR/UEA data archives, or manually provided data sets that comply with some specifications. Since the data loading module from *pyts* [FJ20] offers integrated tools with the UCR/UEA data archives for fetching and downloading data, we have configured our chopper to extend *pyts* by transparently checking both data archives when provided with a data set name. The other option would be to provide a data set folder holding the name of the data set. Inside the folder there should exist one training data set file with the name "FolderName_TRAIN" and one testing data set file with the name "FolderName_TEST". Which is the case for most of the data sets from both archives. For multivariate data sets, the chopper assumes that the training data set file, as well as the testing data set file, contains data of all dimensions. For maximum compatibility, the chopper extends the data processing utilities from *sktime* [Lön+19] which can read data sets that comply with the guidelines provided by the UCR and the UEA archives. We consider "arff" format as the primary expected data type.

Based on the assumption that all data instances are of the same length, we implement our early classification context using a time series chopping algorithm. The algorithm is provided with some time series data set T with instances of length L and three user defined parameters. Parameter s decides the number of splits that the

length L should be split to. $FromBeg$ is a flag which indicates whether the data chopping should occur from the beginning or the end of the time series T. $ChunksToKeep$ which provides the user with the option to use only specific chunks of interest and exclude the processing for all other chunks. Algorithm 1 demonstrates how the chopping algorithm works.

The result of the algorithm is then used to create new copies of T; $[T_1, T_2, \ldots, T_s]$, each copy T_i containing all instances from T but revealing a subsequence of L which we call the chunk. A chunk is a sequence of p data points starting from the beginning or the end of the instances in T based on the value of the parameter $FromBeg$. Each data set copy T_{i+1} reveals one more chunk than the previous data set T_i and the data set T_s represents the set with instances of the full length L. It should be noted that our algorithm tries to have an equal value for p used for each chunk, but this is limited by the length of the time series and the provided number of splits. The resulting data sets are then passed to the training process, where each algorithm applies it's own technique to learn on the data set.

Algorithm 1 The Chopping Algorithm

```
1: function GetSplitIndexes(T, s, ChunksToKeep)
2:     L ← ExtractLength(T)
3:     ChunksSizes ← GetChunkSizes(L, s)
4:     SplitIndexes ← CumSum(ChunksSizes)
5:     if KeepChunks is not Null then
6:         FilteredSplitIndexes ← KeepChunks(SplitIndexes, ChunksToKeep)
7:     else
8:         FilteredSplitIndexes ← SplitIndexes
9:     end if
10:    return FilteredSplitIndexes
11: end function
```

To better understand how the chopping algorithm works, consider the next example. Let's assume we have a data set T with instances of length $L = 100$, we set the value of s= 10 and $FromBeg$ = True. The algorithm starts by calculating the value of p for each of the required 10 chunks using the function 2.

The resulting list $ChunksSizes$ from the algorithm will contain the value 10 for each of the chunks ($ChunksSizes$ = [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]); this is because the length of the time series is divisible by the number of splits provided, so it results in perfect splits of the data and thus chunks of equal sizes. If we assume the same scenario again but with a different s value like 8, the algorithm will try to provide as equal values of p as possible giving back a $ChunksSizes$ of values [13, 13, 13, 13, 12, 12, 12, 12].

After calculating the chunk sizes, the chopping algorithm translates $ChunksSizes$ into a list of indexes called $SplitIndexes$. Which is then used to create the different copies T_i by selecting subsequences. The values of $SplitIndexes$ are the indexes of the last time point that a specific chunk should read. We calculate these values by carrying out a cumulative sum over the values of $ChunksSizes$. For example, the list $ChunksSizes$ = [10, 10, 10, 10, 10, 10, 10, 10, 10, 10] is translated into the list $SplitIndexes$ = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]. This means that the corresponding data set T_1 will contain all it's data instances represented by the sub-

Algorithm 2 Function to Get Chunks Sizes

```
1: function GetChunkSizes(L, s)
2:     ChunksSizes ← [ ]
3:     for i ← 0 to s − 1 do
4:         if i < L (mod s) then
5:             p = (L ÷ s) + 1
6:         else
7:             p = L ÷ s
8:         end if
9:         ChunksSizes.append(p)
10:    end for
11:    return ChunksSizes
12: end function
```

sequences from the 1st time point till the 10th time points, and the data set T_3 will contain subsequence from the 1st time point till the 30th time point. When we apply the same translation to the list $ChunksSizes$ = [13, 13, 13, 13, 12, 12, 12, 12], it gets translated into $SplitIndexes$ = [13, 26, 39, 52, 64, 76, 88, 100].

Finally, the algorithm filters out the list $SplitIndexes$, if the user provided a list of specific chunks to keep using the variable $ChunksToKeep$. The values passed through $ChunksToKeep$ corresponds to the indexes of the chunks the user is interested in, all the other chunks are excluded. Note that this process is different from reducing the number of splits; as the number of splits affects the sizes of the chunks being created. While $ChunksToKeep$ is used only to filter out the chunks after their sizes are already determined and translated into $SplitIndexes$. The space of possible values for $ChunksToKeep$ is between [1, s]. The logic of the function is demonstrated in function 3

Algorithm 3 Function to filter out Chunks

```
1: function KeepChunks(SplitIndexes, ChunksToKeep)
2:     FilteredSplitIndexes ← [ ]
3:     for i ϵ ChunksToKeep do
4:         FilteredSplitIndexes.append(SplitIndexes[i])
5:     end for
6:     return FilteredSplitIndexes
7: end function
```

If we consider the list $ChunksToKeep$ = [1, 3, 5] for our example $SplitIndexes$ = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]. The resulting filtered list will be $FilteredSplitIndexes$ = [10, 30, 50], which will result in only 3 copies of T; T_1, T_3 and T_5, each represented by it's respective subsequence length.

If we have a closer look at the parameter s, we can recognise that it contributes to the granularity of the earliness factor that we are calculating. The value of s decides the total number of chunks we will use, which subsequently decides also the ratio represented by each chunk to the total length of the time series T. That means the greater the value of s, the smaller the chunk sizes and consequently the more granular

results we can test for the algorithms. If we consider our previous example, when we set the value of $s = 10$ then the length of each subsequence would be close to 10% of the total length. If we decided to change the value of $s = 20$, then each chunk will roughly represents 5% of the total length and so on. Increasing the number of splits to be used comes with the cost of time and resources, each extra split adds one extra run per algorithm per data set.

The final result of the chopper is a set of copies of the input data sets. Each copy contains the exact number of instances as the original data set, but the length of its time series instances is decided based on the number of splits that were applied. Some of these copies might be excluded from the results based on the chunks filteration process, while the other remaining ones are passed to the experiment executor to be combined with the prepared models for the actual execution of the experiments.

The Experiment Executor Implementation

The experiment executor is where the actual learning and testing happens. It communicates with the management console for information regarding the learning and testing processes, then sets up the learning environment accordingly. This includes the sampling of hyperparameter space, the number of cross validation folds and the evaluation criteria on which the testing data sets will be evaluated. After the preparation of the chunks for the data sets by the chopper and the initialization of the classifiers in done by the model preparation agent, the experiment executor starts multiple learning processes.

A learning process involves running one classifier on one data set for a specific chunk. During training, the number of hyperparameter sampling is decided by the framework parameter $NumIterations$. Each of these training processes is then followed by a testing process, where performance scores are calculated using the function provided by the parameter $ScoringFunction$ from the management console. After the testing processes are finished, it extracts patterns from the testing results and provides analysis on the performance across and within classifiers in the early classification context.

For the training processes, it is possible to choose whether to optimize hyperparameters for the classifiers through sampling and cross validation or to use the default paramters. Our experiments executor uses a randomized search over the hyperparameters space of classifiers. The framework parameter $NumIterations$ represents the number of random samples to consider when optimizing the hyperparameters. If the space of the hyperparameters is less than $NumIterations$, then the space is exhausted by applying a grid search. Finally, if the experiment is configured to do hyperparameter optimization, then the classifier version that attained the highest validation score is used for calculating the performance score on the testing data set.

For the testing processes, the experiment executor allows the usage of any chosen performance metric which is supported by *sklearn* library [Ped+11]. The chosen performance metric is utilized for the evaluation during both; the cross validation processes and for calculating performance on the test data set. For our implementation, we considered "balanced accuracy" as our default performance metric; to account for problems where data sets have imbalanced classes distribution.

Our evaluation is not only dependent on the performance of classifiers on testing data sets, but also on considers the amount of data that was used for learning. To answer our research question ***how to evaluate the classifiers in the early classification context ?*** We explain why we choose using the $F_\beta - measure$ as an evaluation

mesure for the classifiers and how do we calculate its value.

We previously defined the $F_\beta - measure$ in chapter 2.4, it has been used, in the form of $F_1 - Score$, by [SL20] as the objective function for evaluating their algorithm $TEASER$. The $F_\beta - measure$ is a popular choice for calculating a score that combines 2 objectives at the same time, which are earliness and performance for eTSC problems. Likewise, we use $F_\beta - measure$ as the means of evaluating classifiers in our experiments; due to it's capacity to incorporate both objectives in one score. Which allows us to compare the trained classifiers in terms of a one score value without worrying about fixing the value of either the earliness or the performance, as well as changing assigning different weights for the objectives if needed.

On the other hand, the $F_\beta - measure$ equation that was discussed in chapter 2.4 doesn't fit directly into our evaluation framework. If we try to substitute the results for a classifier trained on the full length data into this equation, we would always get a value of 0. The reason is that the equation is designed to do reverse scoring to the value or earliness, by re-coding its value so that the less seen data points, the higher the contribution of earliness in the equation. This is simply achieved by using (1 - earliness) instead of using the value of earliness directly. In case of calculating scores at full length, this would yield a value of 0 regardless of the performance of the classifier; since the earliness value at full length is 1, then the equation (1 - earliness) would always give a 0 score causing the numerator to be 0.

Our modification to the $F_\beta - measure$ is a two fold procedure. First, we reverse its scale, so that the lower the value the better. Then we reverse score the result again so that it returns to the original scale. We represent our 2 steps modification in equations 4.1 and 4.2.

$$F_\beta - reversed = \begin{cases} 1 & \text{if earliness} = 1 \wedge \text{performance} = 0 \\ 1 & \text{if earliness} = 0 \wedge \text{performance} = 0 \\ (1+\beta^2)\frac{\text{(1 - performance) earliness}}{\beta^2\text{(1 - performance) + earliness}} & \text{otherwise} \end{cases} \tag{4.1}$$

$$F_\beta = 1 - F_\beta - reversed \tag{4.2}$$

In the first step, we reverse the scale of the original $F_\beta - measure$ equation by changing its equation. Instead of reverse scoring earliness and using the actual performance score value, we use the actual earliness value and reverse the performance metric score. We name this measure the $F_\beta - reversed$, to clarify that it represents a reversed value of the F_β used in other papers. Using $F_\beta - reversed$, if we consider the case of classifiers learning at full length, it would solve the problem of attaining a score of 0 and replace it with a real value.

There are 2 special cases to be handled during this step. First, if the classifier doesn't predict anything correct but at the earliest point of time. Second, if the classifier doesn't predict anything correct while seeing the full length. These two cases are then assigned the lowest value, which is 1. In the second step, we reverse score the value from the first step again; to math the expected input for generating critical difference diagrams. We feed these equation with the values of the features $Revealed\%$ and $Testscore$, for the attibutes earliness and performance respectively.

After the successful running of the experiments and calculating performance scores using the provided scoring function, we create a single report for each run experiment. We produce one report per classifier per data set per chunk. These reports collect the essential data and statistics that form the basis of our results, they also form

Item	Description
Classifier	The classifier name
Train time	The total training time (CPU Time)
Test time	The total testing time (CPU Time)
Test score	Performance score on testing data set
Params	List of hyperparameters used and their values
Revealed%	Percent of data points used for training from the total length
F-score	$F_\beta - measure$ combining test score and revealed%
Data set	The name of the data set

Table 4.6: Analysis collected for each experiment

the input for the recommender. Table 4.6 shows the structure of the output for the analysis report.

Most of these information are already available before the learning process starts; because they are metadata about the process which is already known like the $Classifier$, $Dataset$ and $Revealed\%$. While other information are available once the testing process finishes. For example $Traintime$ and $Testtime$ represent the total duration of the learning and testing processes. The $Testscore$ represents performance of the classifier on testing data set. The last measure to calculate is the $F - score$; that's because it needs the testing process to be completely finished and the $Testscore$ to be calculated. In the end, all of these information are used to compare the classifiers in the early classification context and used as an input for the recommender.

4.3.4 The Recommender Implementation

One of the goals of our framework is to be able to recommend good performing algorithms to use on unseen data sets. The Recommender is the component which makes the final decision regarding which algorithms are best to use. It actually is responsible for answering our research question ***How to evaluate the proposed solution?***. A recommender which is able to learn how the different classifiers perform on data sets given chopped training data, would be able to suggest which classifier to use for unseen data sets. Our recommender is not a recommender system in the strict sense, but rather a group of learners that use the performance results of the classifiers from the testbed and predict their performance for new data sets. For the recommender learning, we use the results from all data sets, whether they were finished by all classifiers or not.

The Chunk Learners Implementation

The chunk learners are a group of regression algorithms that learn for each data set for each chunk for each classifier the $F - score$ value that was attained. We learn for each chunk a separate regressor, this means that we learn one regressor for each of the 10%, 20%, 30% and 100% chunks. The regressor that learns on the 10% chunk results, will be used for predicting scores on the new data sets assuming that only 10% of the

training data length was provided. The regressor that learns on the 20% chunk results predicts for the 2^{nd} chunks and so on. When predicting for a new data set, each of the chunk learners predicts the $F - score$ for each of the 6 TSCA, including the dummy on this data set. Then these results are passed on to the classifier validator for the final prediction and for the insights extractor for getting insights.

For the implementation of the chunk learners, we have chosen random forests as our regression models. A Random Forest can be either a classifier or a regressor. It was first introduced in [Bre01]. A random forest is an ensemble of k tree predictors, in which every tree is grown using a randomly sampled input vector independent of the other trees. This technique is called Bagging, in which the training data is generated by randomly selecting N examples with replacement where N is the original training data set size. During the building of the tree, at each split, only a sample of the features are selected as candidates for the splitting [CPB18]. Since the output is numerical in case of regressor random forests, the mean squared generalization error for each tree is calculated and then the final prediction of the forest is the average error over all k trees [SSS17]. The first reason for choosing random forests, is using a classifier which can offer insights about the how the predictions were made. Although random forests are considered as black-box algorithms; due to the large number of trees which makes it hard to get insights about the prediction rule. However, they offer interpretable insights about the selection of features in the prediction and an importance measure for each feature [CPB18]. Also random forests are fast learning algorithms, due to the independence between the individual trees, which allows for parallel training. Finally, random forests have been applied in many scientific fields and have proved to be a competent and accurate algorithm [Jog+17; CPB18].

The Classifier Validator Implementation

The classifier validator is the module that produces the final recommendations. It processes the results from the chunk learners to compare classifiers performances and give the final decision. It applies a logic for deciding which are the good performing classifiers on the data sets for each of the early classification chunks, as well as the full length chunk. We define a good classifier based on its performance in comparison to the dummy classifer on a specific data set. After the caluclation of all the results for all data sets is finished, a ranking procedure is carried out for each data set separately. The classifier with the highest $F - score$ is given the lowest (best) rank on the data set, while the classifier with the lowest $F - score$ is given the highest (worst) rank. If two, or more, classifiers have equal scores, they are given the minimum rank. For example if the second and third classifiers are equal they are both given the rank 2. Then the ranking of each of the classifiers is compared to that of the dummy classifier. All the classifiers with ranks equal to or more (worse) than the dummy classifier are flagged as bad performing classifiers, while the others are considered good performing classifiers. In the end, the bad performing classifiers are filtered for each data set and the final recommendation is presented.

The Insights Extractor Implementation

The insights extractor is responsible for providing analysis results and insights about the results of the chunk learners processes. If more than one learning process is started by the chunk learners, the insights extractor reports insights for each process in a separate folder. Then it aggregates the performance metrics scores across all the

run experiments and provide a summary report on the overall performance of the recommender. Since we are using random forest regressors, it provides insights about the feature importance for each of the chunks. A specific feature which might be important for learning on the 10% chunk data sets, might be less (more) important for the other chunks.

Chapter 5

Data Sets and Experimental Design

In order to address our research questions, this thesis considered a group of data sets from the UCR data archive [Dau+18] and the UEA data archive [Bag+18]. This Chapter will describe the data sets that were used to conduct our experiments and their general properties. Then we will describe our experimental setup.

5.1 UCR/UEA Archives

The University of California Riverside (UCR) and the University of East Anglia (UEA) data archives have been used as references in literature for many experiments on time series problems [AML19; Faw+20; Bag+17; YD19; Rui+20; Faw+19a] and specially for benchmarking the performance of algorithms.

5.1.1 UCR Archive

The UCR archive previously had 85 univariate data sets but in 2018 was then expanded to reach 128 data sets. Despite sharing the property of being univariate, the data sets cover a variety of values for other aspects like; training data size [16, 8926]; testing data size [20, 16800]; number of classes [2, 60]; length of time series [15, 2844] and for some data sets length of instances vary within the data set; and the nature of the data being collected. The archive offers a single predefined train/test split for each of the data sets to facilitate reproducability of results. Finally, the data is z-normalized; to remove offset and scaling. Z-normalization is the transformation of the data to have a mean of zero and one unit of standard deviation.

5.1.2 UEA Archive

The other archive we used is the UEA archive for multivariate data sets. Like the UCR archive, was developed and expanded over time. It started as a small archive of 12 data sets collected by Mustafa Baydogan, then later on, in 2018, was expanded to reach 30 data sets as a collaboration between researchers of UEA and UCR. The data

sets vary in their characteristics; training data size [12, 30000]; testing data size [15, 20000]; number of classes [2, 39]; length of time series [8, 17984]; number of dimensions [2, 1345] and six different different groups based on their application area. The archive was processed so that the data instances have equal lengths, instances of missing data points were excluded and a single predefined train/test split is provided.

5.1.3 Used Data Sets

For the scope of our experiment, a total of 77 data sets were selected from both archives to be used within the framework. We have based our selection criteria for data sets on:

- The data belongs to a real time series structure behind it and not converted from another data format.
- All instances in the data sets has the same length of time series.
- The data data sets should not contain any missing values.

Based on the previously mentioned criteria, we excluded data sets that belonged to either of the types; Simulated, Image or Motion type; to comply with our first selection criteria. For the other two criteria; we avoided any further preprocessing than what is offered by the data archives. As mentioned before, the univariate data sets from the UCR are already Z-normalized. Z-normalization is a common and recommended practice for time series data as it has shown to have a positive impact on the performance of classifiers [Bag+17; Faw+19a] This form of preprocessing is not T for the multivariate data sets from the UEA archive.

We excluded data sets with missing data and avoided filling these missing values; as we believe that data imputation is a whole another topic. We also avoided any interpolation of data. A technique which has been previously used in [RK05; Faw+19a] to make all instances of the data set of the same length as the longest instance. Our goal in this experiment is to focus on comparing the performance of the classifiers in our proposed context based while ensuring, as much as possible, that the main contributing factor is the technique behind the classifiers. This doesn't mean that preprocessing of time series data is of a less importance, but we simply excluded it because we believe that it doesn't fit into our current scope.

The remaining collection of data sets still represented a wide variety in terms of data criteria. We present the statistics about the remaining data sets all together in figure 5.1. Since the smallest data set (StandWalkJump) and the largest data set (InsectWingbeat) are the same as the UEA archive; training and testing data sizes are the same as previously mentioned ranges in the original archive, [12, 30000] and [15, 20000] respectively. The data sets covers covers from binary classification problems up to problems with 52 classes, the range [2, 52]. The range of the time series length is [24, 3000], with Chinatown being the shortest and MotorImagery the longest data set. For the 22 multivariate data sets, the number of dimensions still covered the whole range [2, 1345] like the UEA archive. Finally after our exclusion criteria, the remaining 14 problem types are AUDIO, DEVICE, ECG, EEG, EOG, EPG, HAR, HEMO, MISC, OTHER, SENSOR, SOUND, SPECTRO and Traffic.

Although the data sets that we used covered a variety of properties, but within some properties there were huge imbalances in the distribution of values. Within grouping by training data set size, the group of data sets with sizes [501, 1000] was the smallest, with only 4 data sets which is less than half of the next smallest group. Grouping by series length was the best, among the different groupings, in terms of

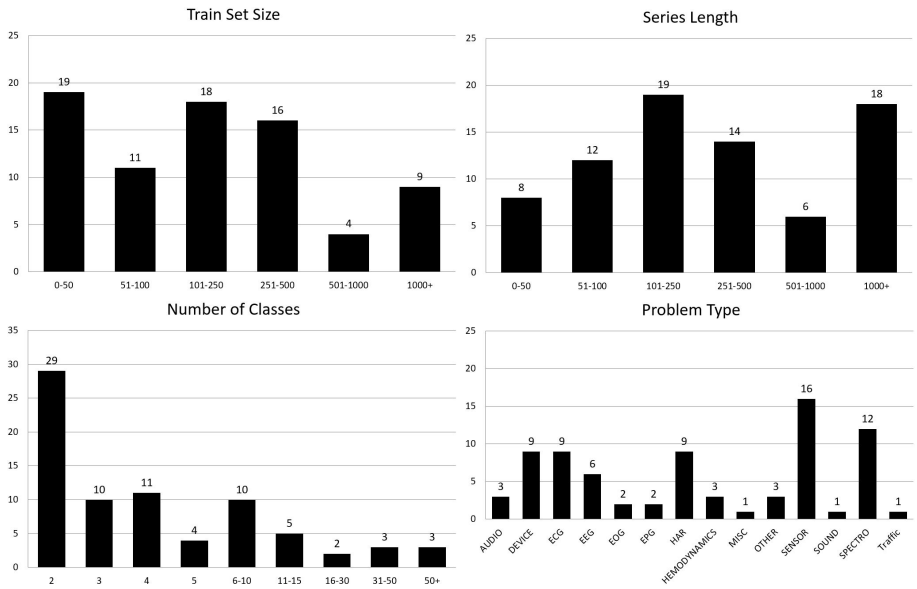


Figure 5.1: Metadata of the used data sets

imbalance. Data sets of lengths [501, 1000] were the smallest group with a count of 6 compared to the largest group, length [101, 250], of 19 data sets. Number of classes was highly imbalanced; the binary classification group contained 29 data sets, which was more than double the second largest group [6, 10] of 10 data sets. This is expected because all the data sets from the UCR archive fall into the same bin, while the multivariate data sets, which are already less in number, are distriuted among multiple bins. Finally, there was an over representation of type SENSOR than any other type with a total of 16 data sets. While multiple other groups were represented by very few data sets like MISC, SOUND and TRAFFIC which only had 1 data set; or like EOG and EPG which had 2 data sets.

Since the final goal of our work is to learn a recommender which will be able to suggest the best performing classifier to unseen data sets based on their properties. It is important to include all necessary and useful metadata about the data sets, which will help the recommender correctly give its recommendations. Imblanced data sets can easily produce misleading results in terms of performance, specially if using a simple performance metrics like accuracy. In cases of extreme imbalance, one can attain very high accuracy scores, simply by always predicting the majority class. Handling the data imbalance is a two fold solution. The first step to help learn about the skewness is to have a definition for imbalance. Then based on this definition flag the data sets as either having balanced classes or not. The second step is to use a suitable performance metric which can give actual performance results and not be mislead by the imbalance.

We discuss the first step here as it is a feature of the data set, the other step we consider in the design of our experiments in section 5.2. To be able to inform our recommender about the skewness property of a data set, we create a new feature, in addition to the previously discussed ones ; to indicate whether a data set is balanced in terms of class distribution or skewed. We define a data set to be skewed if there

exists a class in this data set which is, at least, as twice as likely as any other class.

To better exaplain how this property works, consider the two data sets; DS_1 and DS_2 each of which contains 5 classes. DS_1 has a class distribution: $P(C_1)$ =20%, $P(C_2) = 15\%$, $P(C_3) = 15\%$, $P(C_4) = 25\%$ and $P(C_5) = 25\%$. While DS_2 has a class distribution: $P(C_1) = 20\%$, $P(C_2) = 15\%$, $P(C_3) = 15\%$, $P(C_4) = 30\%$ and $P(C_5) = 20\%$. In this case we would flag DS_1 as a balanced data set; because there is no prior class probability which is double any other. On the other hand, DS_2 would be flaged as skewed/imbalanced because class C_4 has a probability of 30% which is double the value of C_2 and C_3. We denote this feature with the parameter *Balanced*, which is a boolean flag.

Table 5.1: Summary of the 77 data sets used in the experiments

Dataset	**Train Size**	**Test Size**	**Length**	**#Classes**	**Type**	**#Dim**	**Balanced**
ACSF1	100	100	1460	10	DEVICE	1	T
AtrialFibrillation	15	15	640	3	ECG	2	T
BasicMotions	40	40	100	4	HAR	6	T
Beef	30	30	470	5	SPECTRO	1	T
Car	60	60	577	4	SENSOR	1	T
Chinatown	20	345	24	2	Traffic	1	T
CinCECGTorso	40	1380	1639	4	ECG	1	F
Coffee	28	28	286	2	SPECTRO	1	T
Computers	250	250	720	2	DEVICE	1	T
Cricket	108	72	1197	12	HAR	6	T
DuckDuckGeese	60	40	270	5	AUDIO	1345	T
Earthquakes	322	139	512	2	SENSOR	1	F
ECG200	100	100	96	2	ECG	1	F
ECG5000	500	4500	140	5	ECG	1	F
ECGFiveDays	23	861	136	2	ECG	1	T
ElectricDevices	8926	7711	96	7	DEVICE	1	F
EOGHorizontalSignal	362	362	1250	12	EOG	1	T
EOGVerticalSignal	362	362	1250	12	EOG	1	T
Epilepsy	137	138	207	4	HAR	3	T
ERing	30	270	65	6	HAR	4	T

Dataset	**Train Size**	**Test Size**	**Length**	**#Classes**	**Type**	**#Dim**	**Balanced**
EthanolConcentration	261	263	1751	4	OTHER	3	T
EthanolLevel	504	500	1751	4	SPECTRO	1	T
FaceDetection	5890	3524	62	2	EEG	144	T
FingerMovements	316	100	50	2	EEG	28	T
FordA	3601	1320	500	2	SENSOR	1	T
FordB	3636	810	500	2	SENSOR	1	T
FreezerRegularTrain	150	2850	301	2	SENSOR	1	T
FreezerSmallTrain	28	2850	301	2	SENSOR	1	T
Fungi	18	186	201	18	OTHER	1	T
Ham	109	105	431	2	SPECTRO	1	T
HandMovementDirection	160	74	400	4	EEG	10	T
Handwriting	150	850	152	26	HAR	3	F
Heartbeat	204	205	405	2	AUDIO	61	F
HouseTwenty	34	101	3000	2	DEVICE	1	T
InsectEPGRegularTrain	62	249	601	3	EPG	1	F
InsectEPGSmallTrain	17	249	601	3	EPG	1	F
InsectWingbeatSound	25000	25000	600	10	AUDIO	1	T
ItalyPowerDemand	67	1029	24	2	SENSOR	1	T
LargeKitchenAppliances	375	375	720	3	DEVICE	1	T
Libras	180	180	45	15	HAR	2	T
Lightning2	60	61	637	2	SENSOR	1	F

Dataset	**Train Size**	**Test Size**	**Length**	**#Classes**	**Type**	**#Dim**	**Balanced**
Lightning7	70	73	319	7	SENSOR	1	F
LSST	2459	2466	36	14	OTHER	6	F
Meat	60	60	448	3	SPECTRO	1	T
MoteStrain	20	1252	84	2	SENSOR	1	T
MotorImagery	278	100	3000	2	EEG	64	T
NATOPS	180	180	51	6	HAR	24	T
NonInvasiveFetalECGThorax1	1800	1965	750	42	ECG	1	T
NonInvasiveFetalECGThorax2	1800	1965	750	42	ECG	1	T
OliveOil	30	30	570	4	SPECTRO	1	F
PEMS-SF	267	173	144	7	MISC	963	T
Phoneme	214	1896	1024	39	SOUND	1	F
PigAirwayPressure	104	208	2000	52	HEMO	1	T
PigArtPressure	104	208	2000	52	HEMO	1	T
PigCVP	104	208	2000	52	HEMO	1	T
Plane	105	105	144	7	SENSOR	1	F
PowerCons	180	180	144	2	DEVICE	1	T
RacketSports	151	152	30	4	HAR	6	T
RefrigerationDevices	375	375	720	3	DEVICE	1	T
Rock	20	50	2844	4	SPECTRO	1	T
ScreenType	375	375	720	3	DEVICE	1	T
SelfRegulationSCP1	268	293	896	2	EEG	6	T

Dataset	**Train Size**	**Test Size**	**Length**	**#Classes**	**Type**	**#Dim**	**Balanced**
SelfRegulationSCP2	200	180	1152	2	EEG	7	T
SemgHandGenderCh2	300	600	1500	2	SPECTRO	1	T
SemgHandMovementCh2	450	450	1500	6	SPECTRO	1	T
SemgHandSubjectCh2	450	450	1500	5	SPECTRO	1	T
SmallKitchenAppliances	375	375	720	3	DEVICE	1	T
SonyAIBORobotSurface1	20	601	70	2	SENSOR	1	F
SonyAIBORobotSurface2	27	953	65	2	SENSOR	1	T
StandWalkJump	12	15	2500	3	ECG	4	T
StarlightCurves	1000	8236	1024	3	SENSOR	1	F
Strawberry	613	370	235	2	SPECTRO	1	T
Trace	100	100	275	4	SENSOR	1	T
TwoLeadECG	23	1139	82	2	ECG	1	T
UWaveGestureLibrary	2238	2241	315	8	HAR	3	T
Wafer	1000	6164	152	2	SENSOR	1	F
Wine	57	54	234	2	SPECTRO	1	T

5.2 Experimental Design

In this section we discuss the design of the experiments for the 2 components of our framework; the testbed and the recommender. The experiments for the testbed include running the implemented TSCAs on the data archives using different data chunks and calculating their performance using our implementation for the $F_\beta - measure$ previously discussed in subsection 4.3.3. While the experiments for the recommender include learning classifiers on the results of the testbed to predict the best performing classifiers for unseen data sets.

5.2.1 The Testbed Experimental Design

Our experiments were conducted on univariate data sets from the UCR archive as well as multivariate data sets from the UEA archive. The data sets were chosen based on the criteria discussed in the subsection 5.1.3. Each of the data archives offers a train and test split of the data which we have used unchanged for all the classifiers. All our experiments were run on a LINUX server with an AMD Ryzen 7 1700 Processor and 32GB RAM using Python version 3.7.10.

We used the same testbed configuration through out all our experiments, which we will discuss as follows. We fixed the value of the parameter $Splits = 10$, that is for every data set the data would be revealed for the classifiers incrementally in batches of 10% from the total length. The chopping algorithm was set to always reveal data from the beginning of the time series.

Due to time limitations, we imposed a constraint on the number of chunks to be used and a time constraint for each experiment. We restricted our experiments to run only on the 1^{st}, 2^{nd}, 3^{rd} and 10^{th} chunks of the data sets by setting $ChunksToKeep = \{1,2,3,10\}$. The 10^{th} chunk was included to represent the baseline performance of each classifier if shown the full length data. While the first 3 chunks were used to represent the classifiers in the early context scenarios. We defined a time constraint of 2 days for each experiment. Experiments that didn't finish within the time constraint were stopped to make space for other experiments to run.

Hyperparameters optimization was always carried out on all data sets with a $NumIterations$ maximum sampling value of 50 iterations, unless proven unfeasible for any of the classifiers due to memory shortage or time constraint. In this case the experiment is repeated for all classifiers without hyperparameters optimization. We fixed the number of cross validation folds to 5. The selected performance metric for our experiments was *BalancedAccuracy*. Balanced accuracy is a performance metric which can handle data sets with skewed class distributions by avoiding inflated performance estimates. To calculate balanced accuracy, the *Recall* value is computed for each class then averaged over the total number of classes.

Classifiers that failed the experiments

Some of the included classifiers in the framework were excluded from our experiments; either because they couldn't operate in the early classification context or because they couldn't attain comparable results to the published performances by previously published frameworks.

For example, KNNED and InceptionTime were excluded due to the nature of their techniques which couldn't handle our created context. Both algorithms are clearly able to learn on chopped training data sets. However once the testing phase is reached,

they would fail to classify instances of the testing data, showing errors related to mismatches between the expeccted length of instances and the provided length. Yet they would finish the last chunk where the data is provided in it's original full length. The reason why KNNED fails such scenario, is that it uses ED which is a point-wise comparison distance measure that cannot compare time series of unequal lengths [Tan+19]. On the other hand, InceptionTime is a deep learning model whose input layer architecture, represented by number of nodes, depends on the length of the input time series [Faw+19a]. Since we use full length instances for testing, this caused an overflow of input data than what the model structure was expecting. There have been literature discussing adapting TSCAs to unequal time series, but this is out of the scope of our experiments. For more details refer to [CCP09; Tan+19; Faw+19a]

Although KNNDTW has been a competent time series classifier for decades; thanks to the exploitation of the elastic distance measure DTW. We couldn't get either implementation of KNNDTW from *sktime* and *pyts* to work on the chopped data; due to errors in the data representation needed by lower level internal libraries. KNNDTW would have been able to operate in the early classification context if used with a full warping window; which would have been successful in handling extreme classification cases like classifying full length data even when learning on the 10% chunk data set.

Two classifiers were excluded because they attained significantly inferior results compared to the published scores; these are KNNMSM and LS. Specially on the InsectWingbeatSound data set, for which we attained a difference in performance of -45.71% for KNNMSM and -20.71% for LS than the results published by [Bag+17]

Adjustments applied to remaining classifiers

Our experiments proceeded with the remaining 5 classifiers; PForest, TSF, ST, CBOSS and WEASEL. To focus only on the comparison of performance between classifiers and not preprocessing, we consider only data sets where all instances have the same length and no attributes have missing data. We also focus our interest on data which is originally a timeseries data. This means that it is a collection of specific measurements across a span of time. Which lead us to exclude some data set types from taking part in the experiment, more details are presented in subsection 5.1.3. We cover a total of 77 data sets from both the UR and the UEA archives, out of which 55 data sets are univariate and 22 data sets are multivariate.

Time and resources limitations played a big role shaping our experiments. Running classifiers is computationally expensive [SL20] and takes up to hundreds of processing days. This lead us to modify some algorithms so that they can finish promptly; like in the case of PForest, or to use an enhanced version of the original classifier with an option to control the learning phase time; like BOSS and ST.

Due to time limitations; We used the contractable implementations of BOSS and ST offered by *sktime*. These implementations allows passing a time parameter which controls the sampling space for BOSS and the shapelets searching time for ST, beside other performance enhancements. We set the value of the time contract for both to 60 minutes. We have discussed the literature details of these enhancements for ST in subsection 2.3.3 and for BOSS in subsection 2.3.4.

We applied 2 adjustments to the PForest classifier. First, we excluded ED from the pool of distance measures that PForest selects from; since the ED has previously proven not to be compatible with our early classification context when used with the KNN classifier. Second, we excluded TWE from the pool of distance measures when the size of the training data set exceeded 150 instances, which is the median of all

data sets training sizes. TWE is a very slow distance measure. It has been reported by [Bag+17] to be the slowest among all elastic distance measures. When measured on classifing 10 test instances from the StarlightCurves data set, it performed 10,000 times slower than ED.

Also we relaxed the time constraint for PForest; due to its quadratic complexity with length [Luc+19] which made it very slow as we used longer chunks. Since the other algorithms already had a bit of an advantage either for using simple tranformations or for using time contracts, we felt that it would be unfair for PForest to abide by the time constraint for the 10^{th} chunk. Our exception for PForest was if the first 3 chunks were able to finish within the time constraint, we would leave the 10^{th} chunk to run even if it exceeded it. More details about runtime durations is provided in our results.

Learning weights for the $F_\beta - measure$

We use the $F_\beta - measure$ as an objective score for the performance of the classifiers. The $F_\beta - measure$ has been a popular choice for evlauating eTSC algorithms [SL20] because of its capacity to combine both earliness and accuracy. Previous literature has always considered using a special case of the scoring function in which both earliness and accuracy contribute with the same weights. This case is referred to as the Harmonic mean or the F-1 score and can be simply achieved by assigning the value of β to 1.

For our experiments we conducted some experiments to decide if the use of equal weights for accuracy and earliness best suits our newly created context. We experimented with the harmonic mean (i.e. $\beta = 1$) like the previous papers and plotted the distribution of values for $F_\beta - measure$. Figures 5.2 show a comparison between the distribution of values of F_1 and the distribution values of $BalancedAccuracy$ for CBoss. The boxplots from the accuracy image shows that CBoss on the full length of the data has a median value of **0.829**, while on the 10% data it shows a median value of **0.44**. The two boxes do not overlap which means that there is a significant difference in the scores of the two versions. What we would like for the $F_\beta - measure$ in this scenario, is to give a privilege for the 10% version of the classifier as it learns on less training data, such that it would have a high score if it can score a balanced accuracy close to that of the 100% version.

By checking the boxplots for the F_1, we find that the median value for the 10% version is **0.83**, while the median value of the 100% version is **0.708**. Also the distribution of the values for the 10% version is very close to the median and the minimum value is higher than the median of the 100% version. Which means that even for the data sets on which the 10% version of CBoss scores very low accuracy scores, it will still get a high F_1 relative to the 100% version. The harmonic mean over-compensates for the lower accuracy scores with very high values for earliness causing misleading values.

To decide which β value is best to use, we carried out a sequence of experiments using $\beta = [0.1, 0.9]$. The value that gave the most reasonable results was 0.5. Figure 5.3 shows the same comparison between the boxplots of $BalancedAccuracy$ and $F_\beta - measure$ for CBoss but with $\beta = 0.5$. Still F_1 allows for compensation between accuracy and earliness for the 10% version, but it refines the values. The median value for the 10% version became **0.705** while that of the 100% is **0.795**. The distribution of values for the 10% is spread more and the maximum value is falling near the median value of the 100%.

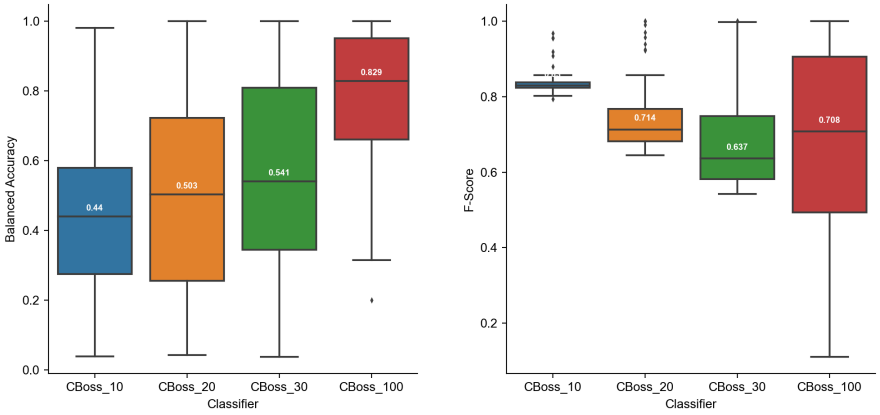


Figure 5.2: Comparison between Balanced Accuracy (left) and F_1 (right) for CBoss

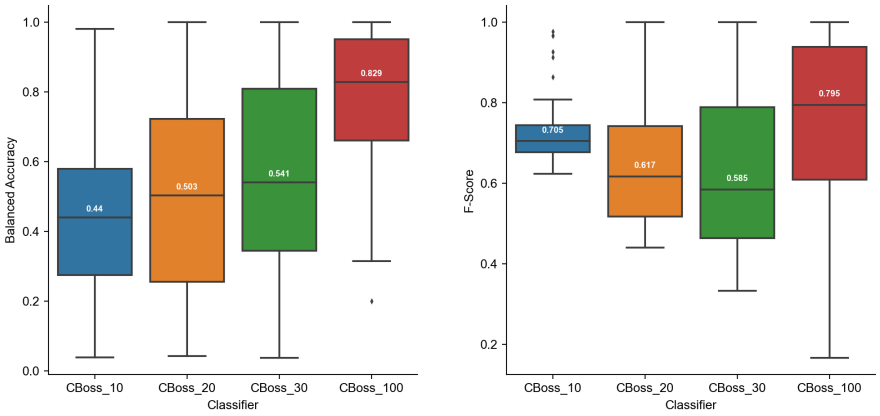


Figure 5.3: Comparison between Balanced Accuracy (left) and $F_{0.5}$ (right) for CBoss

We believe that selecting the best value of β is not trivial. It is problem dependant and not one specific value can fit for all problems. For our experiments we decided to give earliness less priority than performance, but in other situations it might be the case that both of them contribute with the same importance or even earliness is more critical.

5.2.2 The Recommender Experimental Design

As previously discussed in the recommender design, we train one random forest regressor per chunk, which we call the chunk learner. Each chunk learner is trained on the results for a specific chunk and is responsible for predicting the performance score, $F_\beta - measure$, for all classifiers on unseen data sets for that specific chunk.

To train the chunk learners, we use the results of experiments from the testbed combined with metadata about the data set. Each training instance consists of metadata features from the table 5.1 in addition to 3 features from the testbed results in table 4.6; the classifier name, the revealed% and the $F_\beta - measure$ value. The results of all data sets are included in the learning process, regardless of whether they were finished by all the classifiers and for all the chunks or not. We only exclude the three data sets PEMS-SF, DuckDuckGeese and FaceDetection; because they were only finished by one classifier. Which leaves us with 74 data sets.

In the beginning, we randomly split the data into training and testing data sets. Then for each chunk learner, we start a grid search process, on the training data, for learning hyperparameters and model selection. In the end, the chunk learners are evaluated using the best selected model on the unseen testing data sets for the final evaluation.

For the creation of training and testing data sets, we use an 80/20 split with random selection. A total of 59 data sets are used for the learning of chunk learners, while 15 data sets are left out for the final evaluation.

For the training of the chunk learners we use a grid search; to select the best combination of hyperparameters for each chunk learner model for its designated chunk. During the grid search operation, a 10-fold cross validation process is carried out in which each training model is trained and evaluated 10 times. The final score for each model is represented by its average across all 10-folds. We define the hyperparameter space for 2 parameters of the random forests; the maximum depth of trees = {3, 5, 10, ∞} and the number of trees per forest = {2, 3, 4, 5, 10, 50, 100, 250, 500}. After the grid search is finished, the best performing model is selected to be used as the final chunk learner on the unseen testing data sets. To avoid biases in the results of the chunk learners from the selection of training and testing data sets, this whole process is repeated 50 times using random selection of the data sets for training and testing. We use the seeds = [0, 49] for the random selection of data sets and for the bootstrapping and sampling of features in the building for trees; to allow for reproducability of the results.

Since we are using regression random forests, one mandatory preprocessing step has to be carried out; to transform the cateogrical features into numerical features. We do *one-hot-encoding* on the the two features $classifier$ and *type*, which represent the name of the classifier that was used and the type of the data set respectively.

Chapter 6

Results

This chapter analyzes the results of our experiments. We will divide our analysis as follows; analysis for the results of the same classifier across different revealed percentages of data (chunks), analysis for the results of different classifiers across the same revealed percentages of data (chunks) and analysis for different classifiers and different percentages all together using the $F_\beta - measure$.

There were two classifiers that we excluded because they attained significantly inferior results compared to their published scores; these are KNNMSM and LS. Specially on the InsectWingbeatSound data set, for which we attained a difference in performance of -45.71% for KNNMSM and -20.71% for LS than the results published by [Bag+17].

For the remaining classifiers, not all of them were able to finish on all data sets within the time constraint. Specially the multivariate archive, which was challenging for most of the algorithms in terms of memory and time complexity. Our choice of adapting univariate classifiers for multivariate problems using column ensembling might have made things even more challenging; because this technique fits for each dimension in the data sets a classifier, which is a memory and space extensive solution. This was clearly evident for the runs as the number of dimensions of the data sets increased.

TSF was the most successful algorithm in finishing the biggest number of datasets. It was able to finish all the data sets from both archives, with the exception of FaceDetection only. WEASEL was the second best, finishing all data sets from both archives, except for the largest three multivariate data sets; DuckDuckGeese, FaceDetection and PEMS-SF. WEASEL is known for its quality and speed but not for scalability [Luc+19], it failed these data sets due to out of memory errors. CBoss failed to run on DuckDuckGeese, FaceDetection, PEMS-SF and MotorImagery within our time constraint. ST is known to be one of the slowest algorithms, but we use an enhanced version to accelerate its performance. However, ST was not able to finish within our time limit on 15 multivariate datasets; due to its linear time complexity with the number of dimensions. It hit the limit of the time constraint once the number dimensions reached 6. It has also failed on the Fungi dataset, because it couldn't find high quality shapelets during the contract time that we cofigured it to use. The slowest of all and which achieved the least progress on the data sets was PForest. It finished a total of 7 multivariate data Sets and 35 univariate data sets. Although PForest can attain quasi linear training times with the size of training data, but it has a training complexity,

which can be quadratic with length based on its randomly selected distance measure. This is why PForest hit the time contraint easily as the length of the time series increased. All 5 classifiers completed 36 common data sets. TSF, WEASEL and CBoss finished more than 72 data sets, ST finished 61 data sets and PForest finished 42 data sets.

Table 6.1 presents the results of our experiments for the used 5 classifiers on all 77 data sets. We display values for balanced accuracy, $F_\beta - measure$ value and revealed% for each classifier. The default train/test split of the data sets was used for all data runs.

Table 6.1: F_β scores for all 5 classifiers on 77 data sets for all chunks

Data set	CBoss				PForest				ST				TSF				WEASEL			
	10%	20%	30%	100%	10%	20%	30%	100%	10%	20%	30%	100%	10%	20%	30%	100%	10%	20%	30%	100%
ACSF1	0.7778	0.7477	0.7081	0.8780	0.6527	0.4819	0.3780		0.8030	0.8163	0.8433	0.7847	0.7740	0.6838	0.6613	0.5977	0.7899	0.7619	0.6842	0.8900
AtrialFibrillation	0.6667	0.4800	0.4310	0.1667	0.7000	0.6000	0.4000	0.2857	0.7143	0.4800	0.5385	0.3478	0.6667	0.5455	0.3438	0.2254	0.7143	0.5455	0.4310	0.2254
BasicMotions	0.7105	0.8421	1.0000	1.0000	0.6739	0.6154	0.5588	0.7059					0.8200	0.9697	1.0000	1.0000	0.7500	1.0000	1.0000	1.0000
Beef	0.6765	0.5000	0.5800	0.7619	0.6714	0.5106	0.3710	0.2553	0.6667	0.5581	0.5385	0.7619	0.7857	0.7273	0.7273	0.6512	0.6579	0.5106	0.5000	0.7619
Car	0.6846	0.5393	0.5189	0.8000	0.7143	0.5333	0.4474	0.4280	0.6875	0.5926	0.6134	0.7809	0.7500	0.7164	0.6613	0.6693	0.6846	0.5393	0.4558	0.8387
Chinatown	0.6388	0.6526	0.7110	0.9388	0.6548	0.5484	0.9748	0.9674	0.6630	0.7311	0.7293	0.9638	0.9561	0.9820	0.9784	0.9566	0.6388	0.4742	0.3912	0.9353
CinCECGTorso	0.6854	0.5433	0.4528	0.7578					0.6752	0.5562	0.5606	0.8988	0.7235	0.6748	0.7358	0.8876	0.7616	0.8160	0.6891	0.9516
Coffee	0.7314	0.9573	0.9566	1.0000	0.7642	0.6512	0.7115	1.0000	0.7917	0.8485	0.9566	0.9123	0.8944	0.8485	0.8057	1.0000	0.7314	0.9180	0.6560	1.0000
Computers	0.7717	0.7168	0.6811	0.6425	0.7427	0.6289	0.5714	0.5537	0.7525	0.6780	0.6628	0.6382	0.7841	0.7299	0.7097	0.6642	0.7193	0.7042	0.7032	0.6729
Cricket	0.6569	0.7423	0.8171	1.0000									0.8445	0.8521	0.9179	0.8980	0.7156	0.6575	0.7781	0.9827
Earthquakes	0.8068	0.7606	0.7398	0.7039	0.8068	0.7606	0.7398	0.7119	0.7536	0.7606	0.7398	0.5950	0.8068	0.7606	0.7398	0.7039	0.8068	0.7554	0.7337	0.7039
ECG200	0.6942	0.5674	0.4778	0.7962	0.7667	0.7767	0.5483	0.8780	0.7632	0.8081	0.7575	0.8309	0.8448	0.8511	0.8632	0.8193	0.7439	0.5926	0.5119	0.8193
ECG5000	0.6912	0.6739	0.8134	0.9283	0.7469	0.6585	0.6137	0.5533	0.8147	0.8715	0.8973	0.9248	0.9219	0.9123	0.9026	0.9218	0.7410	0.8070	0.7452	0.9264
ECGFiveDays	0.7441	0.6168	0.5606	1.0000	0.7286	0.6543	0.6006	0.8208	0.7229	0.6734	0.6521	0.9985	0.7516	0.6728	0.6213	0.9032	0.7320	0.6315	0.5882	0.9985
ElectricDevices	0.7024	0.5731	0.5578	0.6558					0.6961	0.5575	0.4304	0.3292	0.7315	0.6354	0.6122	0.6355	0.7513	0.6637	0.6086	0.7262
EOGHorizontalSignal	0.6506	0.4682	0.4058	0.4200	0.6509	0.4614	0.3393		0.6522	0.4659	0.4058	0.4173	0.6544	0.4659	0.3958	0.4581	0.6496	0.4743	0.3672	0.4227
EOGVerticalSignal	0.6538	0.4704	0.3777	0.3693					0.6506	0.4674	0.4095	0.3353	0.6525	0.4682	0.4045	0.3985	0.6490	0.4674	0.3513	0.2714
Epilepsy	0.9662	1.0000	1.0000	1.0000					0.7993	0.7920	0.8678	0.9819	0.9169	0.9735	0.9733	0.9640	0.9825	1.0000	1.0000	1.0000
ERing	0.6800	0.5340	0.5135	0.8736	0.6617	0.5015	0.4061	0.4518					0.7547	0.7689	0.8266	0.9405	0.6856	0.5820	0.5876	0.9405
EthanolConcentration	0.6741	0.5127	0.4235	0.3579					0.6954	0.5590	0.4729	0.5541	0.7045	0.5838	0.4996	0.3688	0.6770	0.5177	0.4649	0.5073
EthanolLevel	0.6779	0.5181	0.4346	0.3937					0.6866	0.5618	0.4719	0.3085	0.6942	0.5464	0.4554	0.3937	0.6742	0.5168	0.4240	0.4385
FingerMovements	0.7105	0.6202	0.5526	0.5147									0.7062	0.6250	0.5588	0.4444	0.7247	0.6202	0.5588	0.5351
FordA	0.8003	0.8341	0.8237	0.6833					0.7087	0.6150	0.5944	0.4967	0.7940	0.7570	0.7585	0.7774	0.8329	0.7970	0.9160	0.9577
FordB	0.7765	0.7465	0.7232	0.5672					0.7285	0.6017	0.6037	0.4813	0.7488	0.6722	0.6400	0.6352	0.7630	0.7696	0.6417	0.7830
FreezerRegularTrain	0.7222	0.9715	0.9987	0.9956	0.7890	0.6250	0.8963	0.9049	0.7237	0.9433	0.9987	0.9987	0.9645	0.9982	0.9982	0.9891	0.7222	0.9287	0.7329	0.9751
FreezerSmallTrain	0.7226	0.9457	0.9974	0.9656	0.7518	0.6360	0.7200	0.6611	0.7191	0.9127	0.9982	0.9996	0.8893	0.9798	0.9978	0.8085	0.7222	0.8955	0.6456	0.9505
Fungi	0.6516	0.4553	0.3324		0.6492	0.4757	0.4373	0.8947					0.6554	0.5280	0.5489	0.9337	0.6486	0.4595	0.3429	0.9468
Ham	0.7188	0.6131	0.5500	0.6154	0.7211	0.6269	0.5559	0.6563	0.7188	0.6087	0.5800	0.7191	0.7471	0.7636	0.7273	0.7404	0.7258	0.6222	0.5500	0.5952
HandMovementDirection	0.6814	0.5008	0.4328	0.3149	0.6670	0.5184	0.4737	0.3275					0.6770	0.5421	0.4394	0.4985	0.6729	0.5008	0.3892	0.3149
Handwriting	0.6469	0.4449	0.3753	0.4110					0.6469	0.4465	0.3596	0.3114	0.6570	0.4952	0.4341	0.2649	0.6471	0.4416	0.3338	0.2639
HouseTwenty	0.7172	0.7855	0.8489	0.9687	0.8190	0.7544	0.7184	0.8770	0.7627	0.7791	0.8239	0.8081	0.8628	0.9225	0.8927	0.7791	0.8864	0.9407	0.9204	0.9583
InsectEPGRegularTrain	0.9672	0.9432	1.0000	1.0000	1.0000	0.9950	1.0000	1.0000	0.6918	0.5592	0.4855	0.9850	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
InsectEPGSmallTrain	0.9129	1.0000	0.9467	1.0000	0.9950	0.9005	0.9851	1.0000	0.6918	0.5577	0.4833	0.9651	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
InsectWingbeatSound	0.6522	0.4898	0.4517	0.4964					0.6521	0.4851	0.3914	0.5654	0.7115	0.6489	0.6383	0.5633	0.6566	0.5333	0.4813	0.5839
ItalyPowerDemand	0.6943	0.5766	0.7891	0.8641	0.7071	0.7106	0.5802	0.9495	0.6866	0.5852	0.6913	0.9447	0.7592	0.7142	0.7609	0.9531	0.6943	0.5841	0.5566	0.9117
LargeKitchenAppliances	0.7273	0.6186	0.5714	0.7289	0.6979	0.5607	0.5045	0.3783	0.6904	0.5495	0.4726	0.6154	0.7212	0.6061	0.5417	0.5154	0.6816	0.5405	0.4629	0.6382

Data set	CBoss				PForest				ST				TSF				WEASEL			
	10%	20%	30%	100%	10%	20%	30%	100%	10%	20%	30%	100%	10%	20%	30%	100%	10%	20%	30%	100%
Libras	0.6241	0.4404	0.4136	0.7936	0.6269	0.4274	0.3238	0.0541	0.6299	0.4524	0.3966	0.8387	0.6790	0.6465	0.6853	0.7682	0.6352	0.4898	0.5078	0.8387
Lightning2	0.7328	0.6438	0.5850	0.6923	0.7089	0.6612	0.5850	0.8031	0.7328	0.5966	0.5537	0.6565	0.7328	0.6893	0.7039	0.6565	0.7328	0.6612	0.6414	0.6212
Lightning7	0.6941	0.5541	0.5218	0.6945	0.6619	0.4974	0.3963	0.2566	0.6584	0.4771	0.4280	0.6497	0.6755	0.6186	0.5896	0.6945	0.6867	0.5242	0.4150	0.6795
LSST	0.6241	0.5174	0.4533	0.4529									0.6885	0.5777	0.5268	0.4489	0.6823	0.5844	0.5108	0.5318
Meat	0.8636	0.7059	0.8500	0.8583	0.7353	0.5714	0.4643	0.5977	0.8077	0.9231	0.8012	0.9587	0.8333	0.8276	0.8333	0.9180	0.7609	0.6154	0.4643	0.8193
MoteStrain	0.7139	0.7458	0.6324	0.9320	0.7060	0.7817	0.7693	0.8688	0.7341	0.5966	0.7172	0.8992	0.8789	0.8666	0.8698	0.8452	0.7716	0.7817	0.7946	0.9418
NATOPS	0.6469	0.5186	0.4640	0.8000	0.6376	0.4753	0.3795	0.1860					0.6443	0.5207	0.5171	0.7682	0.6426	0.5269	0.5407	0.8000
NonInvasiveFetalECGThorax1	0.7116	0.5647	0.6373	0.7851					0.7298	0.7056	0.6204	0.2848	0.8362	0.7959	0.7829	0.8453	0.6454	0.4942	0.4060	
NonInvasiveFetalECGThorax2	0.6838	0.7068	0.5754	0.8227					0.6795	0.7408	0.7543	0.5136	0.8697	0.8353	0.8052	0.8784	0.6453	0.4637	0.3614	0.9045
OliveOil	0.6935	0.8571	0.9211	0.8387	0.6667	0.5000	0.5000	0.5113	0.7727	0.9231	0.8846	0.8780	0.8000	0.8000	0.8846	0.8387	0.6579	0.4898	0.4474	0.8780
Phoneme	0.6627	0.5061	0.4238	0.2683	0.6443	0.4479	0.3233	0.0135	0.6647	0.5018	0.4130	0.2345	0.6576	0.4936	0.3978	0.1652	0.6684	0.5135	0.4360	0.2578
PigAirwayPressure	0.6605	0.5082	0.4185	0.9115					0.6581	0.5006	0.5130	0.7543	0.6534	0.4722	0.3517	0.1851	0.6807	0.5730	0.4968	0.5865
PigArtPressure	0.9771	0.9940	0.9822	0.9761					0.8881	0.9073	0.9534	0.9524	0.6624	0.5067	0.4369	0.3515	0.9370	0.9652	0.9591	1.0000
PigCVP	0.8077	0.8786	0.9257	0.9406	0.6443	0.4529			0.6881	0.5892	0.7251	0.7219	0.6599	0.4903	0.3880	0.2363	0.7142	0.6809	0.8155	0.9173
Plane	0.6721	0.4884	0.5223	1.0000	0.6909	0.5316	0.5637	1.0000	0.6736	0.6316	0.6357	1.0000	0.9118	0.9767	0.9538	0.9881	0.6615	0.4884	0.8692	1.0000
PowerCons	0.7432	0.7164	0.7595	0.9046	0.7647	0.6792	0.6847	0.9931	0.7368	0.6180	0.6673	0.9248	0.8103	0.7742	0.7796	0.9931	0.7323	0.6457	0.6259	0.9519
RacketSports	0.6749	0.5096	0.4389	0.7724	0.6914	0.5319	0.4654	0.3804					0.7517	0.7255	0.7821	0.8796	0.6914	0.5588	0.5232	0.8485
RefrigerationDevices	0.7280	0.6000	0.5647	0.4883	0.7000	0.5848	0.5530	0.4066	0.7246	0.6036	0.5714	0.5045	0.7294	0.6329	0.5800	0.5317	0.6969	0.5792	0.5290	0.4484
Rock	0.7083	0.5882	0.6111	0.8780	0.6818	0.5333	0.4278	0.4053	0.6754	0.5128	0.4474	0.8544	0.7174	0.6452	0.6392	0.6948	0.6724	0.5797	0.3911	0.7619
ScreenType	0.7016	0.5859	0.5212	0.4379	0.6933	0.5405			0.6908	0.5682	0.5045	0.4300	0.6989	0.5780	0.5258	0.3808	0.7101	0.5758	0.5227	0.4776
SelfRegulationSCP1	0.7614	0.7052	0.6862	0.7449									0.8097	0.8389	0.8265	0.7604	0.7673	0.8637	0.7845	0.7604
SelfRegulationSCP2	0.7209	0.6400	0.6211	0.4720	0.7081								0.7293	0.6316	0.5836	0.4226	0.7209	0.6316	0.6172	0.5512
SemgHandGenderCh2	0.7810	0.6957	0.7602	0.8563					0.6911	0.7090	0.8235	0.7752	0.8150	0.8905	0.8758	0.9120	0.7667	0.7007	0.6625	0.7487
SemgHandMovementCh2	0.6633	0.4898	0.3921	0.5918					0.6689	0.5573	0.4747	0.4183	0.7356	0.6792	0.7126	0.8128	0.6633	0.4898	0.3960	0.2513
SemgHandSubjectCh2	0.6789	0.5165	0.4914	0.7096					0.6837	0.6218	0.6172	0.5272	0.7586	0.7826	0.7696	0.8940	0.6667	0.5000	0.4529	0.5872
SmallKitchenAppliances	0.7964	0.7353	0.6842	0.7378	0.6938	0.5837	0.4854		0.7113	0.5535	0.5321	0.7408	0.7727	0.7059	0.6638	0.7954	0.7435	0.7026	0.5870	0.6729
SonyAIBORobotSurface1	0.7228	0.8016	0.5272	0.4126	0.8168	0.8307	0.8485	0.7908	0.7060	0.7104	0.7017	0.9202	0.7873	0.7720	0.7460	0.6445	0.7212	0.8841	0.8621	0.8003
SonyAIBORobotSurface2	0.7000	0.6669	0.8502	0.8947	0.7547	0.6644	0.7324	0.8548	0.7554	0.6712	0.6401	0.8947	0.8088	0.7926	0.7785	0.7413	0.7973	0.7704	0.8981	0.9340
StandWalkJump	0.7000	0.5000	0.4643	0.2857	0.6875	0.5217	0.4643		0.6875	0.5714	0.4643	0.2857	0.7143	0.5455	0.5385	0.3478	0.7143	0.5714	0.5800	0.2857
StarlightCurves	0.6805	0.5271	0.4523	0.9688					0.7431	0.6986	0.7900	0.9151	0.9250	0.9413	0.9521	0.9528	0.6911	0.5294	0.4376	0.9375
Strawberry	0.7643	0.6916	0.9573	0.9698	0.7643	0.6916	0.6563	0.9039	0.8033	0.7769	0.7686	0.9268	0.8957	0.9024	0.9478	0.9366	0.7643	0.6916	0.6563	0.9565
Trace	0.6835	0.5755	0.7000	1.0000	0.7273	0.6154	0.5465	0.4743	0.6818	0.5063	0.3955	1.0000	0.7740	0.7339	0.6842	0.9875	0.6724	0.5442	0.5000	1.0000
TwoLeadECG	0.7236	0.6084	0.6552	0.9956	0.7581	0.7010	0.6714	0.9307	0.7223	0.6176	0.6357	0.9978	0.8011	0.8276	0.9348	0.7018	0.7223	0.6230	0.6493	0.9978
UWaveGestureLibrary	0.6546	0.4839	0.3930	0.8157					0.6569	0.4776	0.3688	0.8595	0.7006	0.6772	0.7101	0.8193	0.6524	0.4971	0.4290	0.7761
Wafer	0.9271	0.9807	0.9847	0.9961	0.9319	0.9500	0.8955	0.9935	0.9450	1.0000	0.9949	1.0000	0.9948	0.9906	0.9931	0.9931	0.9932	0.8810	0.8763	0.9996
Wine	0.7596	0.6334	0.7180	0.6957	0.7177	0.6429	0.7030	0.5378	0.7477	0.6526	0.6599	0.7162	0.7727	0.6626	0.6198	0.6753	0.7222	0.6154	0.6071	0.7577

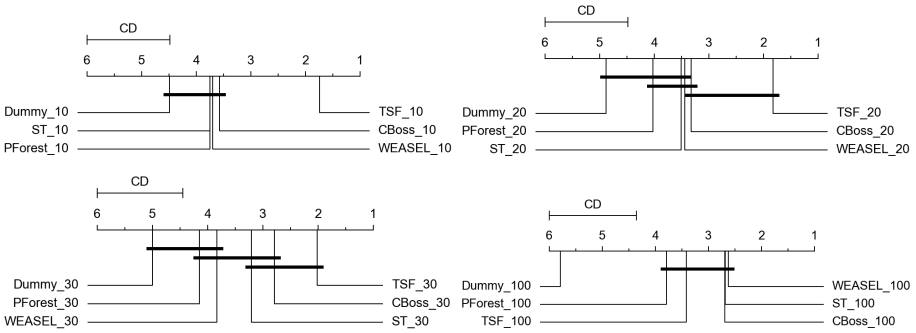


Figure 6.1: Critical Difference diagrams per percentage across all classifiers

We divide our analysis into three parts: a comparison of the effect of data chopping within the same classifier, comparison on the effect of data chopping across different classifiers and unified comparison between all classifiers across all data chops. Since some of the analysis results we present include comparisons between the same classifier trained on different percentages of data, we use a special naming convention. We name the classifiers using the format $NameOfClassifier\_Revealed\%$, where the first part of the name refer to the name of the classifier and the second part refers to the percentage of data used to train the classifier. For example, TSF_10 refers to classifier TSF which was trained on 10% of the data length. This naming convention helps clarify the type of the classifier and the context of comparison regardless of the type of analysis being discussed.

6.1 Results Across Classifiers

An aspect to investigate within the early classification context is the relative performance of classifiers to each other while more data becomes available to train on. We fix the percentage of data revealed, revealed%, and compare the performance of classifiers, which is represented by their ranking based on the value of $F_\beta - measure$. We will dig deeper into the properties of the data sets, to explore the relation between the characteristics of the data and the best performing algorithm.

To better understand the actual performance of the classifiers on the data sets and assess the quality of their results, we introduce a new baseline classifier to the comparison. The baseline classifier always predicts the majority class of a given data set. We name the classifier *dummy*; as an indication of its role in the comparisons. To ensure the integrity of our comparisons, we calculate balanced accuracy and $F_\beta - measure$ for the dummy classifier on all data sets and for the same 4 chunks. Although the dummy classifier will always predict the majority class and thus score the same balanced accuracy, but its $F_\beta - measure$ will differ based on the revealed%.

Figure 6.1 shows the critical difference diagram across all the classifiers and for a fixed revealed percentage of data. In the beginning when only 10% of the data is revevaled, TSF is significantly better than all the other classifiers. The remaining four classifiers form a clique together and are not significantly different from the dummy classifier. Inside that clique, CBoss is the best classifier, WEASEL is the second best and both PForest and ST are almost identical in their ranking.

When the revealed percentage is increased to 20%, TSF still prevails as the first classifier but with no significant different to CBoss and WEASEL. There is still no significant difference between the PForest, ST and WEASEL and between the dummy classifier. ST is able to learn better shapelets on the data and beats PForest. On the third chunk, CBoss and ST are not significantly different from TSF and join it to form the first clique. While WEASEL and PForest are not significantly different from the dummy classifier. The middle clique shows no significant difference between CBoss, ST, WEASEL and PForest. However WEASEL and PForest are both significantly worse than TSF. On the full length of data, the 5 classifiers are all not significantly different from each other. WEASEL attains the best ranking followed by ST and CBoss which are identical, while TSF loses its edge on full length data. PForest ranks the last among the 5 classifiers. All the classifiers form one clique which is significantly better than the dummy classifier.

The most apparent conclusion from the graphs is that TSF is able to do better on data sets during the earlier stages of the early classification context. We believe the reason why TSF is always better on the shorter chunks of data; is because it uses three simple aggregations as features and doesn't depend on advanced features like the other classifiers do. On the other side, the BOP technique applied by WEASEL and CBoss is challenged by the chopping, which notably decreases the chances of repetition of distinctive patterns in the data within such a small length. PForest is also expected to perform badly as it needs to stretch the extent of its elastic distance measures; by measuring distances between testing instances which are $10x$ the length of the training instances. ST gets the lowest scores because in cases like our experiments where majority of the data is chopped, it is very hard to find shapelets that can uniquely indentify the different classes by learning on only the first 10% of the data. An obvious change is visible when more data is made available for training, the techniques of; WEASEL; CBoss and ST are able to make use of the more available information to find better and more distinctive features in the data; which helps them be more competent. Also PForest is able to score get better performances; because its elastic distance measure do not have to compensate for the big difference in length.

6.2 Data Set characteristics and performance in early classification context

The goal of our reasearch is to learn a recommender which can suggest the best classifiers to unseen data sets. The recommender analyzes the characteristics of the data set and then based on the performance from previous experiments, it suggests the most suitable classifiers. Thus it could be very convenient if the characteristics of the data sets we used; like length of series, number of classes and train size are able to provide insights about the best algorithm for a data set. These comparison aspects were first defined in [Bag+17] and have also been adopted in [Faw+19a]. The group of data sets we use for this analysis is very small and for some analysis we break them down by certain criteria. We believe that our results should be interpreted carefully and not generalized for all data sets.

6.2.1 Length of Series

The first feature of the data sets we investigate is the length of the time series. Like the findings of [Bag+17] on univariate data sets and the results of [Faw+19a] on multivariate data sets, our results in the early classification context, didn't show information about the performance of the classifiers in accordance with the length of time series. We show the results of the 5 classifiers on data sets grouped by their time series length in tables 123124 for the 10% length, 123224234 for the 20% length, 1242354235 for the 30% length and 1242335 for the 100% length. The tables display for each classifier the total number of times it was able to score higher $F_\beta - measure$ than the baseline classifier. In the case where multiple classifiers are able to score higher than the baseline on one data set, then the score of this data set is counted for each one of them. We couldn't infer a strong relation between the length of time series and the performance of classifiers except for TSF. Our findings for TSF shows that it is consistently better than the baseline on all length groups for all data chunks. It was able to score good results on all the data sets in the groups (51-100), (101-250) and (1001 +) even by training on only the 10% chunk. Unlike TSF, none of the other classifiers was able to beat the basline for the data set group with the least length (1-50). There is a clear enhancement in the performance of all the classifiers as they learn on more data chunks as we have mentioned before, but no clear relation between the length of the time series and the performance across the different chunks.

6.2.2 Training Data Set Size

The second feature of the data sets we investigate is the training size, and how the number of instances affects the performance of classifiers in our introduced context. We display our results for the train size feature in tables; 123124 for the 10% length, 123224234 for the 20% length, 1242354235 for the 30% length and 1242335 for the 100% length. The table displays for each classifier the total number of times a it was able to score higher than the baseline classifier. TSF consistently scores better than, or at least the same as, the other classifiers on all train size groups for the first 3 chunks. The scores for TSF on all the train size groups are better than the baseline across the different chunks except for the second group (51-100) on the 10% chunk. TSF levels with the baseline classifier on the two data sets Lightning2 and Lightning7 on thee 10% chunk, but beats it once more data is available. During the 20% and the 30% chunks, WEASEL cannot beat the baseline classifier on the data sets in the largest train size group. However on the 100% chunk it catches up and beats it on both data sets. Our results don't show any significance from comparing algorithms based on the training size across the different early classification runs. There is also no significance of patterns of the performance of the same classifiers.

6.2.3 Type of Problem

The third feature of the data sets we investigate is the type of problem. The intention is to try to find an evidence if there is a classifier which is suitable for certain types of problems. We display our results for the data set type feature in tables; 123124 for the 10% length, 123224234 for the 20% length, 1242354235 for the 30% length and 1242335 for the 100% length. Our results show that TSF is able to achieve better results than the baseline classifier on all data sets of the type SPECTRO. This performance is consistently kept across all the different chops, till the other classifiers catch up on

the full length data chop. PForest is not able to beat the baseline on types HAR and SOUND during any of the experiments. However, since both types consist only of one data set, we cannot conclude that PForest will not be able to learn on such types. During the full length experiment, all the classifiers are able to beat the baseline classifier on all the data sets for the groups; SPECTRO, Traffic, DEVICE and EPG.

6.2.4 Number of classes

The fourth feature of the data sets we investigate is the number of classes. The goal is to try to find a connection between the number of classes of data sets and if it has an impact on the performance of certain classifiers. We display our results for the number of classes feature in tables; 123124 for the 10% length, 123224234 for the 20% length, 1242354235 for the 30% length and 1242335 for the 100% length. Like our previous observations for the other faetures, TSF is constantly better than the baseline classifier on all groups across all the chunks. It starts with a good performance on all groups in the 10% experiment and maintains a slight improvement till it reaches the 100%. PForest and ST generally perform better on data sets with small number of classes. As they move towards the 100% chunk they tend to enhance their performance gradually. WEASEL and CBoss on the contrary learn better on the data sets with larger number of classes at the earlier chunks. PForest cannot beat the baseline classifier on the group of (11+) across all chunks. Since the group consists of only two data sets, we cannot generalize this conclusion.

6.3 Results Within Classifier

In our first experiment, we carry out a comparison between copies of the same classifier but trained using different chunks of data; to explore how their performances evolve in the early classification context. We combine results that are based on both $F_\beta - measure$ and balanced accuracy. We exclude the baseline classifier from this comparison.

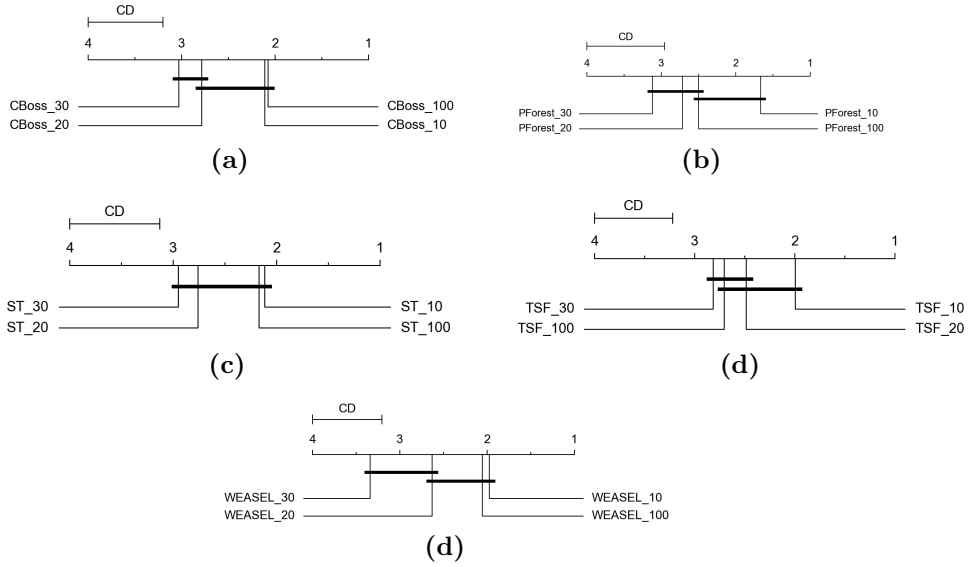


Figure 6.2: **(a)** CBoss **(b)** PForest **(c)** ST **(d)** TSF **(e)** WEASEL

Figure 6.2 shows the critical difference diagram for each classifier separately. We compare each classifier on all the data sets that it finished across all the different chunks, regardless of the other classifiers. The best performing algorithms lie on the right side of the diagram and the worse performing algorithms on the left side. A bar is used to connect a clique of classifiers which are not significantly different based on their ranking.

All classifiers show no significant difference between their 100% classifier version and the 10% version based on their $F_\beta - measure$ scores. For CBoss, CBoss_100 comes in the first place and CBoss_10 comes in the second place. While for PForest, ST and WEASEL; their respective 10% versions; ST_10, PForest_10 and WEASEL_10 rank first. Although TSF shows no significant difference as well, but its 100% version TSF_100 comes in the third place. The 20% and 30% versions for all the classifiers always show no difference in ranking.

If we switch to the critical difference diagrams of accuracies, we find rather different results. For all the classifiers, the 100% and the 10% version are always significantly different in their ranking. WEASEL_100, CBoss_100 and PForest_100 are all significantly better than any of their other versions. TSF_100 and ST_100 are ranked better than their 30% versions TSF_30 and ST_30, but they are not significantly different. This conforms with the concept of eTSC that as more data becomes available the better a classifier performs and also demonstrates the role that earliness plays in the scores of $F_\beta - measure$.

6.4 Results on Runtime Duration

Another aspect we investigate for our experimentts is the runtime of the classifiers. It is not possible to carry out a comparison between the different algorithms we have

used; because of the different adaptations we have applied to them. For some classifiers, have used an enhanced version which allows setting a contract time for feature extraction phases; like in the case of CBoss and ST. This parameter allows us to restrict the amount of time given for the classifier. For PForest, we have excluded two distance measures; euclidean distance because it cannot handle the early classifcation context and TWE distance because it is the slowest elastic distance measure. These enhancements make our results not representative for the original classifiers runtime performance. Instead we provide information about the CPU time that was needed by each classifier to complete training on the data sets. Table 1232346263458 shows the CPU time, in seconds, for all the classifiers on all the used data sets. The values in the table represent the training time of the best model selected from the 5 cross validation folds. If no cross validation was carried out, then the value represents the CPU time for the training process directly. Failed runs and runs that exceeded the time constraint are excluded from the table.

Here should be a table

TSF and WEASEL have short training durations. TSF doesn't use any complex features and instead depends on simple summary statistics on intervals. The stochastic selection of intervals speeds its learning process even faster. WEASEL on the other hand is designed for speed and quality, but this comes on the expense of memory. Although WEASEL takes short training times, it has failed to complete on some data sets due to its large memory print. ST is known to be a strong but rather a significantly slow classifier. The use of the contractable version of ST clearly pays off, it stabilizes the training time across the different chunks by limiting the time during which ST looks for shapelets. The slowest classifier among all experiments was PForest.

Please see Appendix (should mention appendix number or letter) for more information the breakdown of duration by length

Appendix (should mention appendix number or letter) for the breakdown of duration by train size

and Appendix (should mention appendix number or letter) for the breakdown of duration by number of dimensions

6.5 Recommender Results

We divide our results for the recommender into two main parts. The firs part presents the results for the chunk learners, while the second part presents the results for the final recommendation of the whole recommender. In the chunks learners results part, we discuss the performance of each chunk learner and the important features for building the random forests. In the final recommendation part, we discuss the overall performance of the recommender based on its ability to correctly suggest good performing classifiers.

6.5.1 Are the chunk learners able to learn F-scores for the classifiers ?

Table 6.3 shows the overall performance of the chunk learners across all 50 runs. In general, the chunk learners are able to learn about the performance of classifiers on the different chunks, but their mean errors are relatively high. As evident from the results, there is an inverse relation between the performance of the chunk learners and the revealed%. The chunk learner that trains on the 10% results is able to score

the best scores across all the chunk learners. The more revealed data, the harder it becomes for the chunk learners to accurately predict performance of classifiers.

Revealed%	Avg MAE	Avg RMSE
10	0.048274974	0.067191131
20	0.095708094	0.124290807
30	0.120365517	0.154616709
100	0.155801376	0.195245175

Table 6.3: Overall Performance of chunk learners over 50 runs

Figure 6.3 gives a more detailed view of the distribution of RMSE for the chunk learners. It is clear that not only the curves of the higher chunk learner shift towards higher error values, but also the spread of the values is wider. This indicates that few of the predictions made by the lower chunk learner are significantly worse than their actual values. While the higher chunk learner is generally worse and makes larger mistakes by predicting values that are clearly worse than the actual performance of the classifiers.

Figures 6.4 and 6.5 visualize the MEA and RMSE of each chunk learner per classifier. It is evident from the results that for CBoss, ST and WEASEL; the chunk learners are able to score better results as more data is revealed. It is the opposite scenario for PForest. The first 3 chunk learners score the least errors for PForest, but on the 100% dara the errors significantly increase to be the worst among all. This explains the the long tail that appears in the 100% chunk learner RMSE histogram due to the large differences between the actual and predicted F-scores for PForest on the 100% data.

6.5.2 Which features contribute the most to the F-Score Prediction ?

One of the main reasons why we chose to use random forests for the chunk learners, is that they give a notion of importance for features that were used to build the trees. This helps understand what features play role in the prediction of scores for the classifiers for each chunk.

Figure 6.6 shows the average importance of the features across all the 50 runs. The feature importance is calculated using the default method from the *sklearn* library; the mean decrease of impurity. For each selected feature in the internal nodes of the regression trees, the impurity of the split is calculated using variance reduction. After the whole forest is built, the mean variance reduction for each feature is calculated over all the trees.

For the lower chunk learners, the main metadata of the data sets; number of classes, length, train size and test size have the greatest importances among all the features. The number of classes persistently has the highest feature importance over all other features. It is also visible that TSF has high importance on the 10% chunk results, which conforms with our CD diagram on the 10% data; due to its significantly better performance. This importance of TSF fades away as more data is revealed for the classifiers, while on the other hand the imoprtance of the dummy classifier increases.

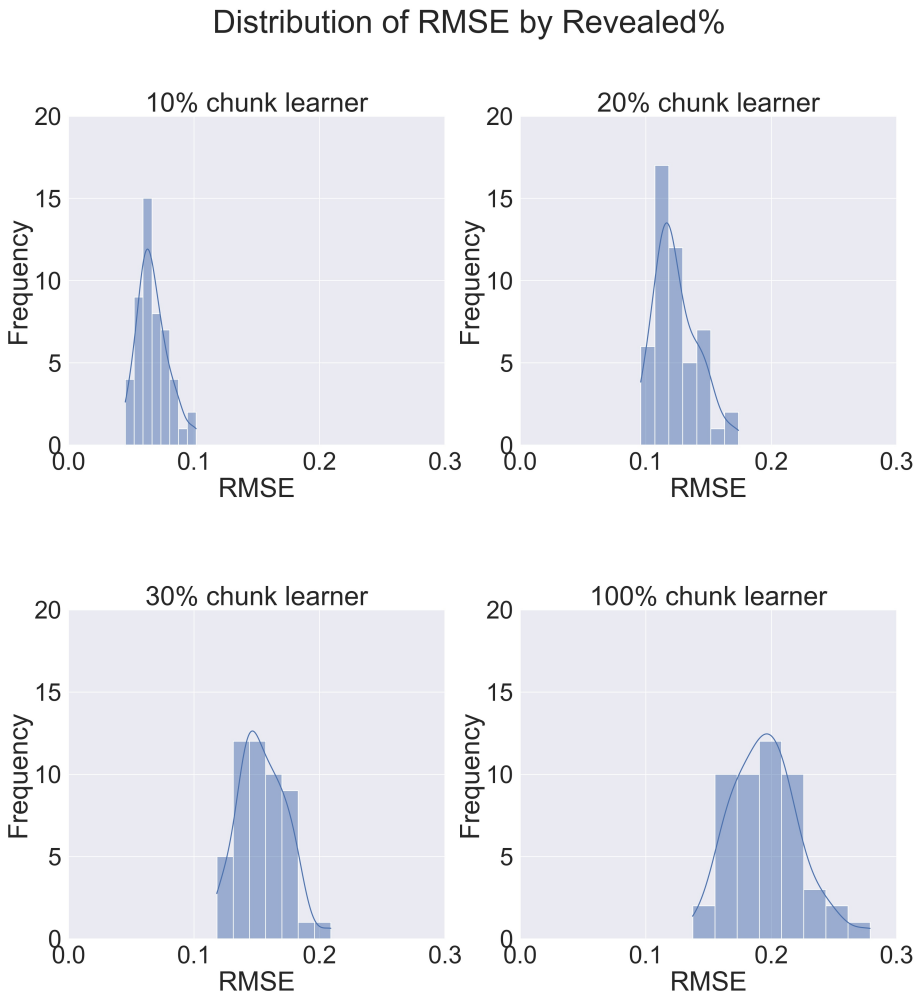
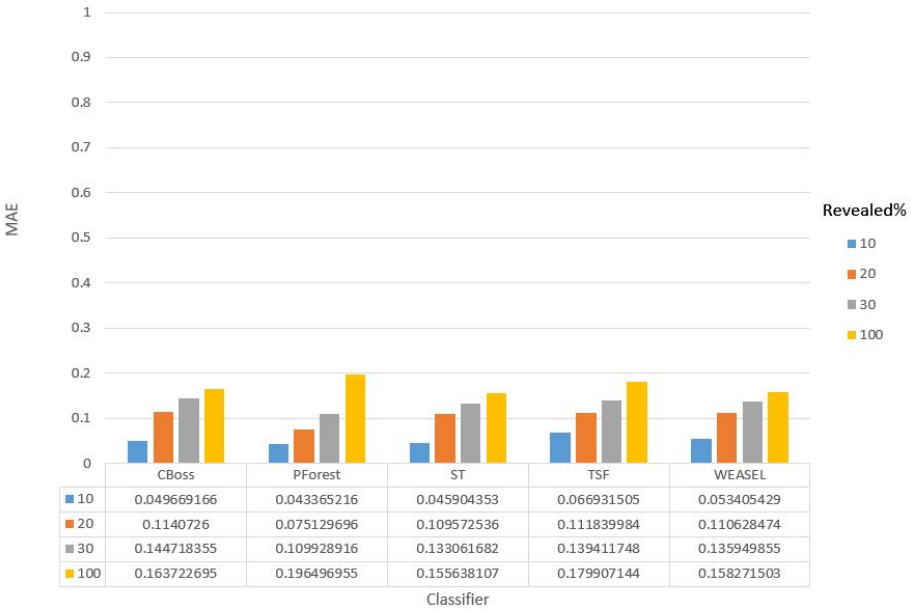
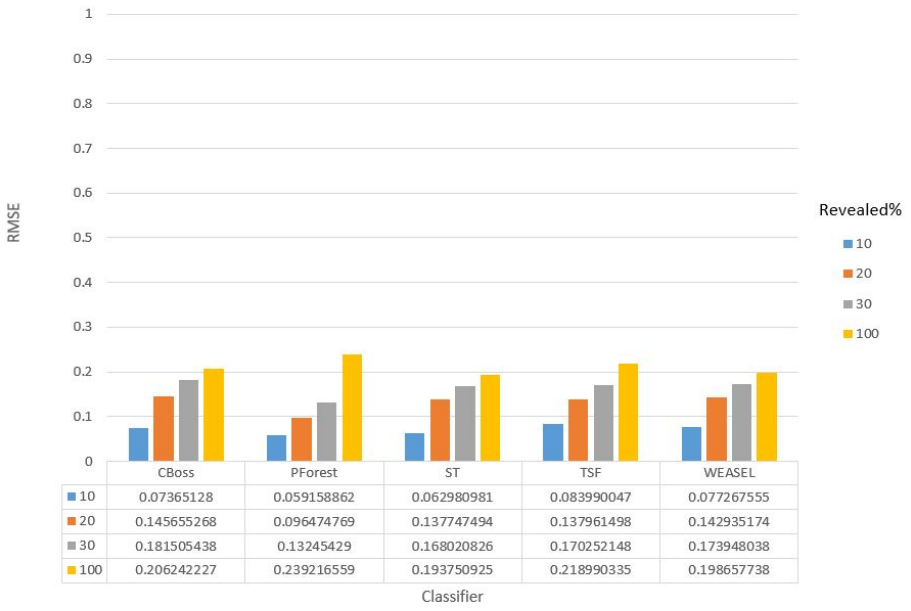


Figure 6.3: Histograms for distribution of RMSE per chunk over 50 runs



	CBoss	PForest	ST	TSF	WEASEL
10	0.049669166	0.043365216	0.045904353	0.066931505	0.053405429
20	0.1140726	0.075129696	0.109572536	0.111839984	0.110628474
30	0.144718355	0.109928916	0.133061682	0.139411748	0.135949855
100	0.163722695	0.196496955	0.155638107	0.179907144	0.158271503

Figure 6.4: Avg MAE for classifiers per chunk over 50 runs



	CBoss	PForest	ST	TSF	WEASEL
10	0.07365128	0.059158862	0.062980981	0.083990047	0.077267555
20	0.145655268	0.096474769	0.137747494	0.137961498	0.142935174
30	0.181505438	0.13245429	0.168020826	0.170252148	0.173948038
100	0.206242227	0.239216559	0.193750925	0.218990335	0.198657738

Figure 6.5: Avg RMSE for classifiers per chunk over 50 runs

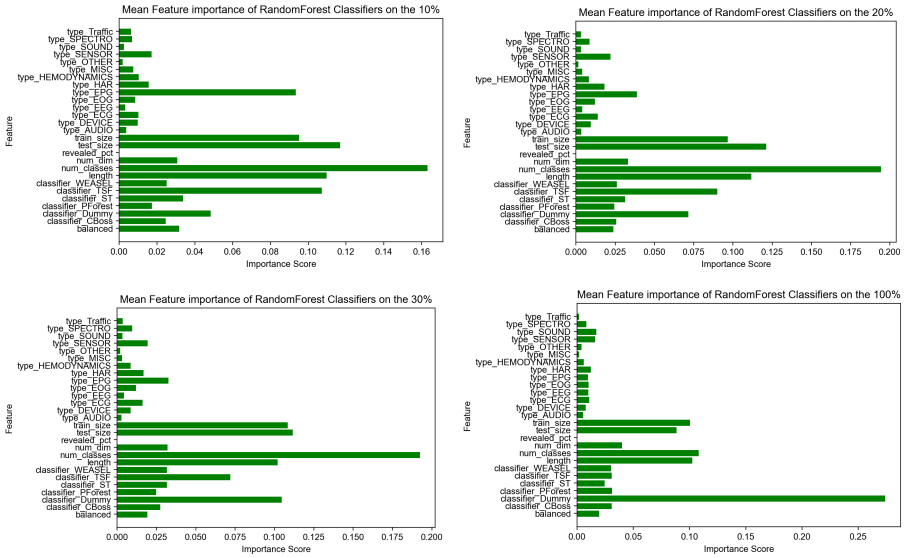


Figure 6.6: Mean Feature Importance per chunk across 50 runs

For the 100% chunk learner, the Dummy classifier has the highest importance. This conforms with our results from the CD diagram from Figure 6.1, where all the classifiers are much better than the Dummy classifier; which makes it easy to purely separate from all other results. This then leads the importance of all the other classifiers to become closer to each other.

6.5.3 Are the final recommendations of the framework consistent with actual results ?

In the end the final result of our framework is the recommendations of the recommender.

It achieves a recall of 1 on the 100% data. This indicates that on the full length data, the recommender never suggests a classifier which performs badly. Although this is a quite impressive result, but we cannot contribute the score to the ability of the classifier to learn on the data. From our CD diagrams results we know that the dummy classifier is significantly worse than all the classifiers on the 100% chunk. We also know that from the F-Score results that the chunk learner on the 100% data has the highest average RMSE values across all the runs and it makes predictions which are Thus, we believe that the reason such a high score is achieved; is because the dummy classifier is the baseline that decides whether a certain classifier is good or bad. The dummy classifier is a very weak classifier and allows for a wide span of error without affecting the results of the recommendation.

Chapter 7

Findings Discussion

This chapter should discuss the Findings

Chapter 8

Summary and Outlook

This chapter should discuss the Summary

Appendix

Bibliography

[Alo+18] Md Zahangir Alom et al. "The history began from alexnet: A comprehensive survey on deep learning approaches". In: *arXiv preprint arXiv:1803.01164* (2018).

[AML19] Amaia Abanda, Usue Mori, and Jose A Lozano. "A review on distance based time series classification". In: *Data Mining and Knowledge Discovery* 33.2 (2019), pp. 378–412.

[Bag+12] Anthony Bagnall et al. "Transformation based ensembles for time series classification". In: *Proceedings of the 2012 SIAM international conference on data mining*. SIAM. 2012, pp. 307–318.

[Bag+15] Anthony Bagnall et al. "Time-series classification with COTE: the collective of transformation-based ensembles". In: *IEEE Transactions on Knowledge and Data Engineering* 27.9 (2015), pp. 2522–2535.

[Bag+17] Anthony Bagnall et al. "The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances". In: *Data Mining and Knowledge Discovery* 31.3 (2017), pp. 606–660.

[Bag+18] Anthony Bagnall et al. "The UEA multivariate time series classification archive, 2018". In: *arXiv preprint arXiv:1811.00075* (2018).

[Bag+20] Anthony Bagnall et al. "A tale of two toolkits, report the third: on the usage and performance of HIVE-COTE v1. 0". In: *arXiv e-prints* (2020), arXiv–2004.

[BB17a] Aaron Bostrom and Anthony Bagnall. "A shapelet transform for multivariate time series classification". In: *arXiv preprint arXiv:1712.06428* (2017).

[BB17b] Aaron Bostrom and Anthony Bagnall. "Binary Shapelet Transform for Multiclass Time Series Classification". In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXXII: Special Issue on Big Data Analytics and Knowledge Discovery*. Ed. by Abdelkader Hameurlain et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 24–46. ISBN: 978-3-662-55608-5. DOI:

10.1007/978-3-662-55608-5_2. URL: https://doi.org/10.1007/978-3-662-55608-5_2.

[BCM16] Alessio Benavoli, Giorgio Corani, and Francesca Mangili. "Should we really use post-hoc tests based on mean-ranks?" In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 152–161.

[Ber+06] Laurent Bernaille et al. "Traffic classification on the fly". In: *ACM SIGCOMM Computer Communication Review* 36.2 (2006), pp. 23–26.

[Bos18] Aaron Bostrom. "Shapelet Transforms for Univariate and Multivariate Time Series Classification". PhD thesis. University of East Anglia, 2018.

[BR16] Mustafa Gokce Baydogan and George Runger. "Time series representation and similarity based on local autopatterns". In: *Data Mining and Knowledge Discovery* 30.2 (2016), pp. 476–509.

[Bre01] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[BRT13] Mustafa Gokce Baydogan, George Runger, and Eugene Tuv. "A bag-of-features framework to classify time series". In: *IEEE transactions on pattern analysis and machine intelligence* 35.11 (2013), pp. 2796–2802.

[CCC16] Zhicheng Cui, Wenlin Chen, and Yixin Chen. "Multi-scale convolutional neural networks for time series classification". In: *arXiv preprint arXiv:1603.06995* (2016).

[CCP09] Jorge Caiado, Nuno Crato, and Daniel Peña. "Comparison of times series with unequal length in the frequency domain". In: *Communications in Statistics—Simulation and Computation®* 38.3 (2009), pp. 527–540.

[Che+18] Weihao Cheng et al. "Predicting Complex Activities from Ongoing Multivariate Time Series." In: *IJCAI*. 2018, pp. 3322–3328.

[CN04] Lei Chen and Raymond Ng. "On the marriage of lp-norms and edit distance". In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 2004, pp. 792–803.

[CPB18] Raphael Couronné, Philipp Probst, and Anne-Laure Boulesteix. "Random forest versus logistic regression: a large-scale benchmark experiment". In: *BMC bioinformatics* 19.1 (2018), pp. 1–14.

[Dau+18] Hoang Anh Dau et al. *The UCR Time Series Classification Archive*. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/. Oct. 2018.

[Dem06] Janez Demšar. "Statistical comparisons of classifiers over multiple data sets". In: *The Journal of Machine Learning Research* 7 (2006), pp. 1–30.

[Den+13] Houtao Deng et al. "A time series forest for classification and feature extraction". In: *Information Sciences* 239 (2013), pp. 142–153.

[DGM97] Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. "Finding similar time series". In: *European Symposium on Principles of Data Mining and Knowledge Discovery*. Springer. 1997, pp. 88–100.

[Din+08] Hui Ding et al. "Querying and mining of time series data: experimental comparison of representations and distance measures". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1542–1552.

[Faw+19a] Hassan Ismail Fawaz et al. "Deep learning for time series classification: a review". In: *Data Mining and Knowledge Discovery* 33.4 (2019), pp. 917–963.

[Faw+19b] Hassan Ismail Fawaz et al. "Deep neural network ensembles for time series classification". In: *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2019, pp. 1–6.

[Faw+20] Hassan Ismail Fawaz et al. "Inceptiontime: Finding alexnet for time series classification". In: *Data Mining and Knowledge Discovery* 34.6 (2020), pp. 1936–1962.

[FJ20] Johann Faouzi and Hicham Janati. "pyts: A Python Package for Time Series Classification". In: *Journal of Machine Learning Research* 21.46 (2020), pp. 1–6. URL: http://jmlr.org/papers/v21/19-763.html.

[FRM94] Christos Faloutsos, Mudumbai Ranganathan, and Yannis Manolopoulos. "Fast subsequence matching in time-series databases". In: *Acm Sigmod Record* 23.2 (1994), pp. 419–429.

[Gao+14] Jingkun Gao et al. "Plaid: a public dataset of high-resoultion electrical appliance measurements for load identification research: demo abstract". In: *proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*. 2014, pp. 198–199.

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[GŁ13] Tomasz Górecki and Maciej Łuczak. "Using derivatives in time series classification". In: *Data Mining and Knowledge Discovery* 26.2 (2013), pp. 310–331.

[GM01] M Pamela Griffin and J Randall Moorman. "Toward the early diagnosis of neonatal sepsis and sepsis-like illness using novel heart rate analysis". In: *Pediatrics* 107.1 (2001), pp. 97–104.

[GO12] Mohamed F Ghalwash and Zoran Obradovic. "Early classification of multivariate temporal observations by extraction of interpretable shapelets". In: *BMC bioinformatics* 13.1 (2012), p. 195.

[Gra+14] Josif Grabocka et al. "Learning time-series shapelets". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 392–401.

[GRO14] Mohamed F Ghalwash, Vladan Radosavljevic, and Zoran Obradovic. "Utilizing temporal patterns for estimating uncertainty in interpretable early decision making". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 402–411.

[Gua+19] Chaoyu Guan et al. "Towards a deep and unified understanding of deep neural models in nlp". In: *International conference on machine learning*. PMLR. 2019, pp. 2454–2463.

[Gup+20] Ashish Gupta et al. "A Fault-Tolerant Early Classification Approach for Human Activities using Multivariate Time Series". In: *IEEE Transactions on Mobile Computing* (2020).

[He+15] Guoliang He et al. "Early classification on multivariate time series". In: *Neurocomputing* 149 (2015), pp. 777–787.

[He+16] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[Hil+14] Jon Hills et al. "Classification of time series by shapelet transformation". In: *Data Mining and Knowledge Discovery* 28.4 (2014), pp. 851–881.

[HLT18] Huai-Shuo Huang, Chien-Liang Liu, and Vincent S Tseng. "Multivariate time series early classification using multi-domain deep neural network". In: *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE. 2018, pp. 90–98.

[Ita75] Fumitada Itakura. "Minimum prediction residual principle applied to speech recognition". In: *IEEE Transactions on acoustics, speech, and signal processing* 23.1 (1975), pp. 67–72.

[JJO11] Young-Seon Jeong, Myong K Jeong, and Olufemi A Omitaomu. "Weighted dynamic time warping for time series classification". In: *Pattern recognition* 44.9 (2011), pp. 2231–2240.

[Jog+17] Amod Jog et al. "Random forest regression for magnetic resonance image synthesis". In: *Medical image analysis* 35 (2017), pp. 475–488.

[JZ14] Zbigniew Jerzak and Holger Ziekow. "The DEBS 2014 Grand Challenge". In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. Mumbai, India: Association for Computing Machinery, 2014, pp. 266–269. ISBN: 9781450327374. DOI: 10.1145/2611286.2611333. URL: https://doi.org/10.1145/2611286.2611333.

[Kat16] Rohit J Kate. "Using dynamic time warping distances as features for improved time series classification". In: *Data Mining and Knowledge Discovery* 30.2 (2016), pp. 283–312.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[KP01] Eamonn J Keogh and Michael J Pazzani. "Derivative dynamic time warping". In: *Proceedings of the 2001 SIAM international conference on data mining*. SIAM. 2001, pp. 1–11.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[Lar+18] James Large et al. *From BOP to BOSS and Beyond: Time Series Classification with Dictionary Based Classifiers*. 2018. arXiv: 1809.06751 [cs.LG].

[LB15] Jason Lines and Anthony Bagnall. "Time series classification with ensembles of elastic distance measures". In: *Data Mining and Knowledge Discovery* 29.3 (2015), pp. 565–592.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[LGI17] Thach Le Nguyen, Severin Gsponer, and Georgiana Ifrim. "Time series classification by sequence learning in all-subsequence space". In: *2017 IEEE 33rd international conference on data engineering (ICDE)*. IEEE. 2017, pp. 947–958.

[Lin+07] Jessica Lin et al. "Experiencing SAX: a novel symbolic representation of time series". In: *Data Mining and knowledge discovery* 15.2 (2007), pp. 107–144.

[Lin+12] Jason Lines et al. "A shapelet transform for time series classification". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2012, pp. 289–297.

[Lin+15] Yu-Feng Lin et al. "Reliable early classification on multivariate time series with numerical and categorical attributes". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2015, pp. 199–211.

[LKL12] Jessica Lin, Rohan Khade, and Yuan Li. "Rotation-invariant similarity in time series using bag-of-patterns representation". In: *Journal of Intelligent Information Systems* 39.2 (2012), pp. 287–315.

[LLF14] Kang Li, Sheng Li, and Yun Fu. "Early classification of ongoing observation". In: *2014 IEEE International Conference on Data Mining*. IEEE. 2014, pp. 310–319.

[LLY17] Xin Liao, Kaide Li, and Jiaojiao Yin. "Separable data hiding in encrypted image based on compressive sensing and discrete fourier transform". In: *Multimedia Tools and Applications* 76.20 (2017), pp. 20739–20753.

[LMT16] Arthur Le Guennec, Simon Malinowski, and Romain Tavenard. "Data augmentation for time series classification using convolutional neural networks". In: *ECML/PKDD workshop on advanced analytics and learning on temporal data*. 2016.

[Lön+19] Markus Löning et al. "sktime: A unified interface for machine learning with time series". In: *arXiv preprint arXiv:1909.07872* (2019).

[Low14] Richard Lowry. "Concepts and applications of inferential statistics". In: (2014).

[LTB18] Jason Lines, Sarah Taylor, and Anthony Bagnall. "Time series classification with HIVE-COTE: The hierarchical vote collective of transformation-based ensembles". In: *ACM Transactions on Knowledge Discovery from Data* 12.5 (2018).

[Luc+19] Benjamin Lucas et al. "Proximity forest: an effective and scalable distance-based classifier for time series". In: *Data Mining and Knowledge Discovery* 33.3 (2019), pp. 607–635.

[Lv+19] Junwei Lv et al. "An effective confidence-based early classification of time series". In: *IEEE Access* 7 (2019), pp. 96113–96124.

[Mar08] Pierre-François Marteau. "Time warp edit distance with stiffness adjustment for time series matching". In: *IEEE transactions on pattern analysis and machine intelligence* 31.2 (2008), pp. 306–318.

[MKY11] Abdullah Mueen, Eamonn Keogh, and Neal Young. "Logical-shapelets: an expressive primitive for time series classification". In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2011, pp. 1154–1162.

[Mor+17a] Usue Mori et al. "Early classification of time series by simultaneously optimizing the accuracy and earliness". In: *IEEE transactions on neural networks and learning systems* 29.10 (2017), pp. 4569–4578.

[Mor+17b] Usue Mori et al. "Reliable early classification of time series based on discriminating the classes over time". In: *Data mining and knowledge discovery* 31.1 (2017), pp. 233–263.

[Mor+19] Usue Mori et al. "Early classification of time series using multiobjective optimization techniques". In: *Information Sciences* 492 (2019), pp. 204–218.

[MVB19] Matthew Middlehurst, William Vickers, and Anthony Bagnall. "Scalable dictionary classifiers for time series classification". In: *International Conference on Intelligent Data Engineering and Automated Learning*. Springer. 2019, pp. 11–19.

[Par+13] Nathan Parrish et al. "Classifying with confidence from incomplete information". In: *The Journal of Machine Learning Research* 14.1 (2013), pp. 3561–3589.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Pet+16] François Petitjean et al. "Faster and more accurate classification of time series by exploiting a novel dynamic time warping averaging algorithm". In: *Knowledge and Information Systems* 47.1 (2016), pp. 1–26.

[Pro+06] Pavlos Protopapas et al. "Finding outlier light curves in catalogues of periodic variable stars". In: *Monthly Notices of the Royal Astronomical Society* 369.2 (2006), pp. 677–696.

[RA04] Juan Jose Rodriguez and Carlos J Alonso. "Support vector machines of interval-based features for time series classification". In: *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Springer. 2004, pp. 244–257.

[RK04] Chotirat Ann Ratanamahatana and Eamonn Keogh. "Making time-series classification more accurate using learned constraints". In: *Proceedings of the 2004 SIAM international conference on data mining*. SIAM. 2004, pp. 11–22.

[RK05] Chotirat Ann Ratanamahatana and Eamonn Keogh. "Three myths about dynamic time warping data mining". In: *Proceedings of the 2005 SIAM international conference on data mining*. SIAM. 2005, pp. 506–510.

[RK13] Thanawin Rakthanmanon and Eamonn Keogh. "Fast shapelets: A scalable algorithm for discovering time series shapelets". In: *proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM. 2013, pp. 668–676.

[Rui+20] Alejandro Pasos Ruiz et al. "The great multivariate time series classification bake off: a review and experimental evaluation of recent algorithmic advances". In: *Data Mining and Knowledge Discovery* (2020), pp. 1–49.

[Ruß+19] Marc Rußwurm et al. "End-to-end learning for early classification of time series". In: *arXiv preprint arXiv:1901.10681* (2019).

[SAD12] Alexandra Stefan, Vassilis Athitsos, and Gautam Das. "The move-split-merge metric for time series". In: *IEEE transactions on Knowledge and Data Engineering* 25.6 (2012), pp. 1425–1438.

[SC78] Hiroaki Sakoe and Seibi Chiba. "Dynamic programming algorithm optimization for spoken word recognition". In: *IEEE transactions on acoustics, speech, and signal processing* 26.1 (1978), pp. 43–49.

[Sch15] Patrick Schäfer. "The BOSS is concerned with time series classification in the presence of noise". In: *Data Mining and Knowledge Discovery* 29.6 (2015), pp. 1505–1530.

[Sch16] Patrick Schäfer. "Scalable time series classification". In: *Data Mining and Knowledge Discovery* 30.5 (2016), pp. 1273–1298.

[SH12] Patrick Schäfer and Mikael Högqvist. "SFA: a symbolic fourier approximation and index for similarity search in high dimensional datasets". In: *Proceedings of the 15th international conference on extending database technology*. 2012, pp. 516–527.

[Shi+20] Ahmed Shifaz et al. "Ts-chief: A scalable and accurate forest algorithm for time series classification". In: *Data Mining and Knowledge Discovery* (2020), pp. 1–34.

[SK16] Tiago Santos and Roman Kern. "A Literature Survey of Early Time Series Classification and Deep Learning." In: *Sami@ iknow*. 2016.

[SL17a] Patrick Schäfer and Ulf Leser. "Fast and accurate time series classification with weasel". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2017, pp. 637–646.

[SL17b] Patrick Schäfer and Ulf Leser. "Multivariate time series classification with WEASEL+ MUSE". In: *arXiv preprint arXiv:1711.11343* (2017).

[SL20] Patrick Schäfer and Ulf Leser. "TEASER: early and accurate time series classification". In: *Data Mining and Knowledge Discovery* 34.5 (2020), pp. 1336–1362.

[SM13] Pavel Senin and Sergey Malinchik. "Sax-vsm: Interpretable time series classification using sax and vector space model". In: *2013 IEEE 13th international conference on data mining*. IEEE. 2013, pp. 1175–1180.

[SPK18] Joan Serrà, Santiago Pascual, and Alexandros Karatzoglou. "Towards a Universal Neural Network Encoder for Time Series." In: *CCIA*. 2018, pp. 120–129.

[SSS17] Balraj Singh, Parveen Sihag, and Karan Singh. "Modelling of impact of water quality on infiltration rate of soil by random forest regression". In: *Modeling Earth Systems and Environment* 3.3 (2017), pp. 999–1004.

[SW17] Simone Scardapane and Dianhui Wang. "Randomness in neural networks: an overview". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7.2 (2017), e1200.

[Sze+15] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[Sze+17] Christian Szegedy et al. "Inception-v4, inception-resnet and the impact of residual connections on learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 31. 1. 2017.

[Tan+19] Chang Wei Tan et al. "Time series classification for varying length series". In: *arXiv preprint arXiv:1910.04341* (2019).

[Tav+20] Romain Tavenard et al. "Tslearn, A Machine Learning Toolkit for Time Series Data". In: *Journal of Machine Learning Research* 21.118 (2020), pp. 1–6. URL: http://jmlr.org/papers/v21/20-091.html.

[TH16] Pattreeya Tanisaro and Gunther Heidemann. "Time series classification using time warping invariant echo state networks". In: *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2016, pp. 831–836.

[TM16] Romain Tavenard and Simon Malinowski. "Cost-aware early classification of time series". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2016, pp. 632–647.

[TPW20] Chang Wei Tan, François Petitjean, and Geoffrey I Webb. "FastEE: Fast Ensembles of Elastic Distances for time series classification". In: *Data Mining and Knowledge Discovery* 34.1 (2020), pp. 231–272.

[Wan+13] Xiaoyue Wang et al. "Experimental comparison of representation methods and distance measures for time series data". In: *Data Mining and Knowledge Discovery* 26.2 (2013), pp. 275–309.

[WYO17] Zhiguang Wang, Weizhong Yan, and Tim Oates. "Time series classification from scratch with deep neural networks: A strong baseline". In: *2017 International joint conference on neural networks (IJCNN)*. IEEE. 2017, pp. 1578–1585.

[Xin+11] Zhengzheng Xing et al. "Extracting interpretable features for early classification on time series". In: *Proceedings of the 2011 SIAM International Conference on Data Mining*. SIAM. 2011, pp. 247–258.

[XPK10] Zhengzheng Xing, Jian Pei, and Eamonn Keogh. "A brief survey on sequence classification". In: *ACM Sigkdd Explorations Newsletter* 12.1 (2010), pp. 40–48.

[XPP09] Zhengzheng Xing, Jian Pei, and S Yu Philip. "Early prediction on time series: a nearest neighbor approach". In: *Twenty-First International Joint Conference on Artificial Intelligence*. Citeseer. 2009.

[XPP12] Zhengzheng Xing, Jian Pei, and S Yu Philip. "Early classification on time series". In: *Knowledge and information systems* 31.1 (2012), pp. 105–127.

[YD19] Omolbanin Yazdanbakhsh and Scott Dick. "Multivariate Time Series Classification using Dilated Convolutional Neural Network". In: *arXiv preprint arXiv:1905.01697* (2019).

[YK09] Lexiang Ye and Eamonn Keogh. "Time series shapelets: a new primitive for data mining". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2009, pp. 947–956.

[Zei12] Matthew D Zeiler. "Adadelta: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012).

[Zha+17] Bendong Zhao et al. "Convolutional neural networks for time series classification". In: *Journal of Systems Engineering and Electronics* 28.1 (2017), pp. 162–169.

[Zhe+14] Yi Zheng et al. "Time series classification using multi-channels deep convolutional neural networks". In: *International conference on web-age information management*. Springer. 2014, pp. 298–310.

[Zhe+16] Yi Zheng et al. "Exploiting multi-channels deep convolutional neural networks for multivariate time series classification". In: *Frontiers of Computer Science* 10.1 (2016), pp. 96–112.

8.1 Declaration of Authorship

I hereby declare that I have written this thesis "A comparison of Time Series Classification Algorithms based on their ability to learn in an Early Classification Context" without any help from others and without the use of documents and aids other than those stated above. Furthermore, I have mentioned all used sources and have cited them correctly according to the citation rules. Moreover, I confirm that the paper at hand was not submitted in this or similar form at another examination office, nor has it been published before.
Magdeburg, 25.05.2021