Martin Siddons

Registration number 100225776

2021

A Contribution to the Time Series Machine Learning Open Source Java Toolkit: TSML

Supervised by Dr Anthony Bagnall

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

Over the previous decade, the field of Machine Learning research has produced a considerable number of classification algorithms which can be used to make predictions on given data. Comparing the effectiveness of these algorithms can be a difficult task when most are produced as stand-alone code for a research project. In the area of Time Series Classification, the Time Series Machine Learning (TSML) toolkit aims to solve this issue by implementing as many of these algorithms as possible in one place, to allow for easy evaluation and comparison. This report details the implementation of one such algorithm into the toolkit, and evaluates its use against similar algorithms previously implemented.

Acknowledgements

I would like to thank Dr. Anthony Bagnall for his excellent in-depth tutelage in the field of Machine Learning, without which this project would not have been possible.

Contents

1 Introduction 7
1.1 Machine Learning and the Time Series Domain . . . 7
1.2 Core Project Research . . . 8
1.2.1 Transformations . . . 8
1.2.2 Experimental Evaluation . . . 10
1.2.3 Classifiers . . . 11
1.2.4 Clustering . . . 13
1.2.5 Visualisation . . . 14
1.2.6 Regression . . . 15
1.2.7 Core Project Decision . . . 15

2 Design 17
2.1 Classifier Details and Requirements . . . 17
2.1.1 J48 . . . 18
2.1.2 VGEN . . . 18
2.1.3 NSGA-II . . . 19
2.2 UML . . . 20
2.3 Pseudocode of the Classifier . . . 20
2.4 MoSCoW Analysis . . . 24
2.5 Development of Initial Classifier Components . . . 25
2.5.1 VGEN . . . 25
2.5.2 NSGA-II . . . 26
2.6 Classifier Implementation . . . 26

3 Analysis 28
3.1 Code Functionality . . . 28
3.2 Data Description . . . 31
3.3 Classifier Description . . . 32
3.4 Testing Methodology . . . 32

4 Evaluation 33
4.1 Analysis of Results . . . 33
4.1.1 Tuning . . . 33

4.1.2 Classifying 33
4.2 Project Analysis 35
4.3 Further Changes and Additions 36

5 Conclusion **37**

References **38**

List of Figures

1 An example segment of time series data with time on the x axis. A shapelet of the series has been highlighted in green. 44
2 A simplified class diagram of the decision tree segment of the classifier. The root class is J48SS, called by the user. 44
3 A simplified class diagram of the jMetal segment of the classifier, containing the NSGA-II algorithm. Code enters from the decision tree at C45Split and BinC45Split, depending on if binary splits are used. . . . 45
4 A simplified class diagram of the SPMF segment of the classifier, containing the VGEN algorithm. Code enters from the decision tree at C45Split and BinC45Split, depending on if binary splits are used. . . . 46
5 A Screenshot from Windows Task Manager showing CPU usage on an 8 core Intel CPU with HyperThreading running the FastShapelets classifier. 46

List of Tables

1 MoSCoW Analysis of project requirements 47
2 Data characteristics of the datasets used 48
3 Parameters for tuning J48SS 49
4 Results from tuning the J48SS classifier 49
5 Review of objectives listed in the MoSCoW Analysis 50

1 Introduction

1.1 Machine Learning and the Time Series Domain

In the field of Machine Learning, classification is the act of building an algorithm with a given set of data. Data unseen by the algorithm can then be processed by it in order to be classified into a result, which is a prediction of one of several possible outcomes that depend on the features of said data. For example, if we wished to predict whether there might currently be a traffic jam on a nearby road, we would gather relevant data such as the time of day, the weather, the day of the week and month of the year, how many cars have passed a certain spot in the road in a given time-frame, if there are roadworks in the area, etc., and the outcome - whether there was a traffic jam at the given time. This data can then be entered into a classification algorithm to produce a model on traffic, which is called training the model. With the model, the user can enter in data collected at the current time and be given either a true or false statement on whether the model believes there is currently a traffic jam, or the likelihood as a percentage that the model thinks there is a traffic jam. The former is known as *classification*, which is useful for predicting discrete values such as True/False, Male/Female, Large/Medium/Small etc., the latter is *regression*, which is useful for predicting continuous values such as Height, Age, Price etc.

Time Series Machine Learning concerns the classification of data where at least one variable is related to time. For example, this could be analysing the output of a heart monitor to classify whether a patient is at risk of a heart attack (Jin et al., 2009), analysing the sound of an animal to identify which species the animal belongs to (de Camargo et al., 2017), or even estimating how long it might take a team of developers to write a piece of software (Oliveira et al., 2010).

There are a number of algorithms available to researchers today designed to provide insight into different collections of data, but there is little effort being made to consolidate these algorithms into one place and evaluate them against each other to understand which algorithms are best used in which situations. This was the motivation behind putting together the Time Series Machine Learning Open Source Java Toolkit (TSML)[1]. TSML is a Java toolkit compatible with the popular Weka[2] machine learning

[1] *https://github.com/uea-machine-learning/tsml/*
[2] *https://www.cs.waikato.ac.nz/ml/weka/*

open source Java toolkit, with a focus on time series classification. It was put together following the production of Bagnall et al. (2017), where the authors compared eighteen popular Machine Learning algorithms against each other on 85 datasets to understand how well they matched up when presented with the same data in a controlled setting.

The core of this project is to contribute to this open source toolkit by implementing something that can be used by researchers to carry out further study. In the initial project description, this was described as one of the following: Transformations, Experimental evaluation, Classifiers, Clustering, Visualisation or Regression. Since what I will design will be used by others without me being available for questions, it is important that the work is clear and well documented.

1.2 Core Project Research

My research focussed on understanding the core elements that I was expected to choose from to implement in the toolkit. In the previous section I outlined these five elements, and in this section we will explore them in more detail.

1.2.1 Transformations

Transformation is a step that can be taken between gathering data and producing a model, where the data are prepared in a way that makes any model built from it more accurate. Transformation is also known as Feature Engineering, Feature Processing or Feature Extraction where a feature is a variable in the gathered dataset (Google, 2020). Normalisation transforms a continuous variable so that the mean value is 0 and the standard deviation is 1, which brings all the values within that variable to a common scale. Rescaling is a transformation where a vector of values is added or subtracted by a constant, then multiplied or divided by a constant to transform the data from one scale to another, such as from Celsius to Fahrenheit (Lakshmanan, 2019). By combining these methods we can standardise a variable, a common method of which is to subtract the mean and divide by the standard deviation for each value in the variable to obtain the Standard Normal. There is also the min-max scaler, which transforms all values in a variable to between 0 and 1 while maintaining the distance between them (Huilgol, 2020). There are also mandatory transformations that must be used in certain cases where data must be made compatible with the model being used. These include conversion, where non-numeric features are converted into numeric ones such as true and false

into 1 and 0, and input resizing, where the number of inputs is scaled to a fixed size such as with resizing images in image recognition tasks. Bucketing is a form of conversion transformation which can take multiple values, such as a range of continuous values, and map them to a single categorical value, such as all the values from 1 to 1.99 in a variable being categorised as 1. With time series data, transformations generally act on variances in the time domain.

There are a number of transformations available to use on data in the above categories. For normalisation, there is Linear Scaling or Min-Max (for uniform distributions where you don't want to change the shape of the data), Log Scaling (where the majority of the values are concentrated in a small area of the distribution and you want to even this out), Clipping (where the variable has a long tail of values that don't give much insight into the data, so they are bundled into one single value) and Z-Score (where the variable has a large range but few outliers). There are a number of different transformations that can be used to removing time series features, too. Shapelet transforms work on data elements called shapelets, which are subsets of the data set made of a consecutive collection of data points, and example of which can be found in Figure 1. By selecting a shapelet from the data and moving through the data set, we can use a shapelet transform to locate other parts of the data that match that shape, which should indicate the two areas share a commonality. Fourier transforms are transformation algorithms that can take a waveform within space or time and separate it into its component spacial or temporal frequencies. For example, a Fourier transform can take a musical chord and separate it into its individual note frequencies and volumes. Principal Component Analysis is a transformation used to reduce the dimensionality of data by separating out the most important parts, known as the Principle Components, which can be more easily compared (Abdi and Williams, 2010).

Transformations are a critical component of the the Hierarchical Vote Collective of Transformation-based Ensembles (HIVE-COTE), which is currently shown to be one of the top algorithms in the field of time series machine learning (Middlehurst et al., 2021).

All of the previously mentioned transformers have, in some way, previously been added to TSML. Although new transformer algorithms are being created all the time, there is, unfortunately, a machine learning model primarily used for Natural Language Processing called a transformer model that has been popular since 2017, which makes searching for new transformation algorithms quite difficult. All of the transformers I could find while researching this topic were already implemented in some way within

TSML, making this a difficult element to implement in the toolkit.

1.2.2 Experimental Evaluation

Experimental Evaluation is a process of comparing different classifiers across different datasets to discover the strengths and weakness in models and learning how these models can be combined to produce ensembles that utilise the individual model's strengths in combination with other models. This is the core of what TSML was designed for and a number of papers have been written by the TSML team at UEA which details the type of evaluation that can be carried out using the toolkit. Aside from the previously mentioned *Bake Off* paper (Bagnall et al., 2017), the *A Tale of Two Toolkits* series is centred around experimental evaluation. The first paper from the series (Bagnall et al., 2019a) compares the accuracies of algorithms from sktime[3], a Python time series machine learning toolkit compatible with scikit-learn, with their TSML equivalents and advises on improvements that can be made to sktime. The second paper (Bagnall et al., 2019b) explores four dictionary-based classification algorithms (BOSS, cBOSS, S-BOSS, WEASEL) inside TSML across a range of 107 data sets and goes into much detail about the strengths and weaknesses of said algorithms. Paper three (Bagnall et al., 2020) details the use of the HIVE-COTE meta-ensemble, which is an evolution of the COTE classifier detailed in *Bake Off* and formed of a collection of multiple different collections of classifiers working together. More recently, a follow-up paper to this has been written which details changes to HIVE-COTE and how it compares to new algorithms that have more recently been added to TSML (Middlehurst et al., 2021).

The papers explored above give a great overview of the way classifiers within TSML are evaluated. Critical Difference (CD) diagrams are used to easily tell how much better some algorithms are compared to others. First introduced by Demšar (2006), these diagrams display the average of the rank averages of each classifier tested as a value from 1 to n, where n is the number of classifiers tested. Classifiers are then grouped in cliques with a horizontal bar to indicate where two or more classifiers are not significantly different from each other. This significance is determined through a pairwise Wilcoxon signed-rank test, which is a statistical test to assess whether the difference in mean ranks between two classifiers is statistically significant, and is given by the formula $W = \sum_{i=1}^{N_r} [sgn(x_{2,i} - x_{1,i}) \cdot R_i]$, where X_r is the sample size, sgn is the sign func-

[3] *https://www.sktime.org/en/latest/*

tion (whether the value is positive or negative), and R is the Rank. If this value is less than the 5% level of significance for the two-tailed test (1.96), then the difference between the two classifiers is considered not statistically significant and will form a clique (Statistics-Solutions, 2020). This test is then performed against all classifiers tested to produce the CD diagram.

Another method used in evaluation is Cross-validation(CV), which is where a given dataset is randomised and divided into k subsets (often ten) where $k-1$ subsets are used to train the classifier while the remainder is used to evaluate its accuracy. This is then repeated k times, where the evaluation subset is changed each time until all subsets are used as the evaluator (Brownlee, 2020b). The average results from these tests are then taken as representative of the overall quality of the classifier on that data set.

Although Experimental Evaluation doesn't appear to be a large addition to the toolkit by itself, I feel that it could be pared with another element in order to emphasise the impact that element may have on the toolkit.

1.2.3 Classifiers

A classifier, as has previously been covered, is an algorithm used to predict the class of a set of data. Classes are strictly defined in the dataset and the full range of classes known to the model from the start of training . An example of a class can be the numbers 0-9 in the case of recognising handwritten digits, or whether a patient requires further monitoring or not when presenting certain symptoms. Providing the targets with the input data in this way defines classification as a form of supervised learning. Le (2020) details a number of different types of classification algorithms which all have their various strengths and weaknesses.

Linear classifiers can be used on two-class problems by bisecting the input space with a hyperplane - that is, for a problem with only two inputs classifying to just two classes, the two inputs can be mapped to dimensions on a grid and a line can be used to separate them into their respective classes. For three inputs, a three dimensional space is constructed and a plane is used to classify them, and $n > 3$ inputs can be separated with a hyperplane (who's dimension is always n-1). This approach is not particularly accurate in general but is one of the fastest available, and with a large enough number of inputs it can rival more complex solutions (Yuan et al., 2012). Linear classifiers implemented within TSML include LibLinear (Fan et al., 2008), LinearRegression, Winnow

(Littlestone, 1988) and multiple Bayesian classifiers.

A Support Vector Machine (SVM) in its most basic form is a linear classifier which attempts to create the largest gap between two categories by having only the support vectors (values closest to the separation boundary) affect the position of the boundary. However, by utilising a kernel function, extra dimensions can be added to the data in order to allow it to be separated in more ways, allowing a hyperplane to be constructed that will cleanly separate complex distributions. In addition, a technique known as the kernel trick can be utilised to create a kernel function with much less processing, bringing time complexity from $O(n^2)$ to $O(n)$. SVM algorithms within TSML include SMO & SMOreg (Platt, 1998), TunedSVM and LibSVM (Chang and Lin, 2011).

K-Nearest Neighbour (KNN) Classifiers map all inputs to a search space model. Unseen data are then mapped into the search space and are classified by the majority class from the nearest k elements to that point, which are measured using a distance measure such as Euclidean or Manhattan distancing. They are considered a simple class of classifier which require a large amount of storage space (as they must hold all the values from the dataset within the classifier model rather than describing the positions algorithmically), but this means that data can easily be added to the model over time to potentially improve its effectiveness. KNN algorithms previously added to TSML include IBK (Aha and Kibler, 1991), kNN and DTW_kNN.

Boosting is a type of meta-algorithm which utilises multiple classifiers in an ensemble to reduce bias and variance (Breiman, 1996). Generally, these boosting algorithms will be simple so-called 'weak learners' which individually would be poor performing but quick to compute a prediction, but when used together will produce a result far greater than their individual contributions. This is achieved by weighting each classifier according to whether it correctly predicted a class, and then re-weighting on the addition of every instance to the dataset. Some boosting algorithms currently included in TSML are AdaBoost (Freund et al., 1999), MultiBoostAB (Webb, 2000), SimpleLogistic (Sumner et al., 2005) and TunedXGBoost (Chen and Guestrin, 2016).

Classification Decision Trees are one of the most common types of classification algorithms (Wu et al., 2008). They map each variable to a branch on a tree which splits down to leaves, each of which contain a predicted class for the instances with those variable values. The branches are calculated through a splitting criterion, which aims to maximise the 'quality' of a split by choosing to branch on the variable which most cleanly splits the class. Decision trees previously added to TSML include SimpleCart

Breiman et al. (1984), RandomSubspace / ProximityForest / RandomForest (Breiman, 2001), BFTree (Shi, 2007), ADTree (Freund and Mason, 1999), ID3 (Quinlan, 1986), J48 (Quinlan, 1993) and many more.

Artificial Neural Networks (ANN) have been gaining prominence in Machine Learning over the past decade. Modelled after the design of an animal brain, they pass variables into a layer of neurons which combine each variable with a weight, passing it through a threshold and sending the new value on to the another layer of neurons. Values from each neuron in the previous layer are brought together and the same action is applied again until the output layer is reached, which in this case will be a layer of neurons the same size as the number of classes. TSML includes MultilayerPerceptron (Rumelhart et al., 1985) as an ANN suitable for classification.

1.2.4 Clustering

Clustering is a form of unsupervised learning, which relies on the algorithm grouping data with similar variables together (Roman, 2019). They are used when there is no class that can be predicted but the data can still be formed into groups that causes some instances to bear a stronger resemblance to each other than they do to the remaining instances (Witten et al., 2011). Generally speaking, these clusters will often be mapped with a centre point and boundary in order to understand which cluster unseen data could fall into. It is generally used as a data analysis tool to understand more about a given dataset, such as if there are outliers or anomalies, before being processed further (Brownlee, 2020a). Priy (2020) breaks clustering down into four methods:

- Density-Based Methods, which work on the assumption that denser regions have a commonality which isn't shared by the lower-dense regions.
- Hierarchical Based Methods, where clusters form a tree structure where new clusters are formed off of previously created ones. Divisive methods start with a complete dataset and separate out the clusters going down the tree, whereas Agglomerative methods start with each cluster and decide how they come together to form the dataset.
- Partitioning Methods separate the data into k partitions where each partition forms a cluster.

- Grid-based Methods can quickly cluster data by separating it into a grid and clustering data in each cell irrespective of the other cells.

TSML already contains a number of clustering algorithms, including DictClusterer / TTC (Aghabozorgi et al., 2014), KShape (Paparrizos and Gravano, 2015), KMeans (MacQueen et al., 1967), CAST (Ben-Dor et al., 1999) and many more. The field of clustering algorithms is quite diverse and there is a high likelihood there exists is a good algorithm that has yet to be added to the toolkit.

1.2.5 Visualisation

Visualisation is where the output from an algorithm is transformed from raw numbers and labels into another form which makes it easier to understand the data being presented. This is usually in the form of plots / graphs, though depending on the algorithm you might want to visualise how it makes decisions, such as displaying a tree structure in the case of Decision Tree Classifiers or Hierarchical Clusterers, or displaying the accuracies and weights of individual algorithms inside a boosting ensemble.

The scikit-learn[4] toolkit in particular is known for impressive data visualisations due to it being able to hook into and leverage the matplotlib Python library for drawing plots and graphs of data. Fortunately, TSML also includes the ability to hook into matplotlib using python scripts included in the toolkit, however there doesn't appear to be many options available at present for utilising this.

Evaluation of classifiers appears to require many different types of visual aids which have previously been partly covered. These include CD diagrams, bar charts summarising the features in a collection of datasets (such as grouping counts of classes or dataset sizes), plots of time taken to train an algorithm against another feature such as the size of the training set or the accuracy of the algorithm during testing, plotting expected accuracy against actual accuracies (known as a Texas Sharp Shooter plot), to name a few. It would seem TSML does not natively support creation of these plots and it would seem, from their use in previous papers, that this could be a useful addition to the toolkit - especially if this could be developed in a way that didn't require Python or MATLAB to be installed on the user's machine.

[4] *https://scikit-learn.org/stable/*

1.2.6 Regression

Regression is a group of algorithms that are a counterpart to classification. Where classification uses inputs to find a specific class (or 'box') for fitting unseen data into, regression attempts to map those inputs to a continuous variable output. An example of this is predicting the average height of a plant from certain features, or predicting the price of a house from data about it, such as number of rooms, its location or its floorspace. Regression has many similar counterparts to those previously explored under classification, though it's worth pointing out the differences.

Linear Regression involves using a line (or hyperplane) to find the best fit for the given data, where unseen data can be mapped to a point on this line that matches closest with its inputs. (Multiple) Linear Regression generally will only work when there is a clear relationship between the input features and its output, such as the height of a human by age and sex. These are most effective with continuous inputs, whereas classifiers must find a way to split the data and lose information. Regression algorithms in TSML include LinearModel, MultiLinearRegression, LeastMedSq and SimpleLinearRegression.

Logistic Regression is a technique used when the inputs are continuous but the output is binary (either 0 or 1, true or false). The relationship between the input variables is found using a logit function (Tyagi, 2020). Within TSML, LogitBoost (Friedman et al., 1998)uses this technique, and 'FT' (Gama, 2004) is a tree algorithm which uses logistic regression.

Support Vector Machines can also work on regression problems and operate in a similar way to their classification counterparts, but rather than classifying to a number of classes, a Support Vector Regression algorithm finds function that can map to continuous variables. An example in TSML of SVM regression is TunedSVM.

1.2.7 Core Project Decision

After researching the field of Machine Learning and the choices available for the core of my year project, I have decided to focus on adding a new classifier algorithm to TSML, as it appears it could be a substantial addition that I feel can be of use to the team for comparison with other algorithms, or inclusion in ensembles - clusters of algorithms that work together to produce a more accurate prediction. To this end, I intend to also perform my own experimentation with the algorithm I implement in order to under-

stand its strengths and to compare its results to those given in previously written papers utilising said algorithm.

I decided to focus my search here on Decision Tree and Boosting algorithms which are yet to be included within the TSML toolkit. This is due to the fact that Decision Trees are taught earlier in the second semester of the Machine Learning module which will hopefully allow me more time to understand how these algorithms are implemented in TSML once I have adjusted myself to using the features of the toolkit. Boosting algorithms appear to be a focus of research in TSML, where ensemble algorithms in general are often constructed, evaluated and tweaked, such as with BOSS and TSF - as well as collectives of ensembles, such as HIVE-COTE, TS-CHIEF and ROCKET.

He et al. (2020) proposes a framework called ORCL-U, which combines a confidence-based labelling strategy (CLS) and an online rule-based classifier learning approach (ORBCL) to form a classifier which handles unlabelled multivariate time series data and has been proposed to solve the issue of such data taking a large amount of time to process with conventional methods. While the results of evaluation against two other classifiers (PLKNN and SAMULT) proved promising, the complexity of implementing both the labelling strategy and the classifier may be too much considering the amount of time I'll have to implement it.

Malhotra et al. (2017) proposes a Recurrent Neural Network (RNN) technique called TimeNet which can be pre-trained on time series datasets, which it extracts features from. TimeNet can then use the features previously found to classify unseen datasets. Although it is impressive that a classifier can be built to somewhat classify data it's never been trained on, and TSML is currently lacking in RNN classifiers, the results presented didn't seem too impressive when compared with the results currently being produced by meta-ensembles in TSML, and I don't feel there is a need for the toolkit to have a model that can classify data it's not trained on at this time.

Gou et al. (2019) proposes a KNN variant they call Generalized Mean Distance-based K-Nearest Neighbour classifier (GMDKNN). This algorithm proposes to reduce the sensitivity of the model to k by calculating the local mean vectors to the nearest k neighbours, then extending the harmonic mean distance to it's general form, which the authors refer to as the generalized mean distance. The results in this paper initially appear to be impressive, where GMDKNN appears to have the greatest accuracy of all the evaluated classifiers. However, a lack of a CD diagram or any mention of a Wilcoxon Signed-Rank test was a hint that these results may not be as impressive as it

seems. Indeed, the margins between GMDKNN and the second-place classifier appear to be tiny at times, and the accuracies are generally in the 80s and 90s, which indicates to me that the data sets were specifically chosen to make the classifier appear better than it might actually be. This finding, in addition to the fact that KNN classifiers in general are vastly outperformed when compared to many other classifiers available today, have me believe that this would not be a reasonable addition to the TSML toolkit.

Brunello et al. (2019) proposes a variant on the J48 Decision Tree Classifier called J48SS. This classifier adapts the Weka implementation of J48 to handle Sequential and Time Series data in addition to the numerical and continuous variables it can already handle. It is claimed that the classifier can even handle all four attribute types within the same dataset, during one single execution cycle. To handle sequential data, the classifier uses an existing algorithm called VGEN (Fournier-Viger et al., 2014), which the authors claim is the state-of-the-art algorithm for extraction of general sequential generator patterns. To handle time series data, the classifier relies on another existing algorithm called NSGA-II (Deb et al., 2002) to extract time series shapelets. The experiments in this paper focussed on two areas, analysing speech from phone conversations and classification of time series datasets from the UCR archive. The results of the first area were a little disappointing to the researchers, performing noticeably worse than the same result gathered through the Google Speech API, though it was noted that the error rate was still low enough that the company involved with the experiment was happy with the results. The second area produced average accuracies which were impressive when compared to the RandomShapelet algorithm mentioned, though I feel it would be nice to see comparisons to other algorithms. With this in mind, I feel that the TSML toolkit may be able to benefit from the addition of this classifier and I will focus on it's implementation for the remainder of this report.

2 Design

2.1 Classifier Details and Requirements

As noted in the previous chapter, the J48SS classifier contains components to handle the Numeric and Continuous data types that J48 handles, and extends this to handle Time Series and Sequential data too.

2.1.1 J48

The mechanism for handling Numeric and Continuous data types is the same for J48SS as it is for its predecessor J48, which is included in TSML. J48 is the Weka implementation of the C4.5 algorithm introduced by Salzberg (1994), one of the most widely used classification algorithms in machine learning. J48 builds a decision tree by starting with the entire dataset of instances provided by the user, and finds the variable (referred to as an attribute in Weka) which best splits the classes. This is found through calculating the Information Gain (IG) of each attribute at that point in the tree, where the attribute with the largest gain value is chosen. IG is calculated by the formula $Gain(X,A) = H(X) - \sum_{Y \in V} \frac{|Y|}{|X|} H(Y)$, where $H(X)$ and $H(Y)$ represents the entropy of the parent nodes and current potential nodes, $|Y|$ is the number of instances in the parent node and $|X|$ is the number of instances in the currently assessed potential node (Lutes, 2021). The entropy H is calculated with the formula $(X) = -\sum_{i=1}^{c} p(X = i) \log_2(p(X = i))$, where p represents the probability of a given event occurring. When a node is produced which results in all instances having the same class, that node is called a leaf, and will return the value of that class when an unseen instance matches down to that node. If the algorithm forms a node where all attributes have been exhausted but instances of multiple classes remain, a distribution function is created weighted by the remaining classes, by which unseen instances will use.

J48 is tunable across many parameters, including pruning, setting the minimum number of instances per leaf, setting binary splits, using Laplace smoothing, using MDL correction and setting a seed. J48SS, being developed from the same initial code, will also handle these tuning parameters through setters in the class as well as through command line options. In addition, the classifier will handle additional tuning parameters for the additional attribute types below.

2.1.2 VGEN

VGEN is cited by Brunello et al. (2019) (hereafter referred to as 'the J48SS paper') as being implemented in an Open-Source Data Mining library written in Java called SPMF[5] (Fournier-Viger et al., 2016). It's sequential pattern matching operation is detailed in both the J48SS paper, as well as by Fournier-Viger et al. (2014), which describes the process as discovering interesting patterns in sequences - specifically, fre-

[5] *https://www.philippe-fournier-viger.com/spmf/*

quently occurring patterns from within a dataset. The issue highlighted in the J48SS paper is that algorithms presented in the past generate large numbers of patterns, which can be very memory and CPU intensive and gives outputs that can be overwhelming and hard to understand. Due to these issues, a series of algorithms were created to extract 'sequential generator patterns' instead, which can hold information on all the frequent patterns that can be generated, and lead to the development of VGEN.

2.1.3 NSGA-II

The NSGA-II algorithm responsible for extracting shapelets is implemented in a framework for multi-objective optimisation and metaheuristics called jMetal[6] (Durillo and Nebro, 2011). It's operation is detailed in both the J48SS paper, as well as by Deb et al. (2002). The jMetal implementation is threadable, which hopefully will allow it to process shapelets faster than existing classifiers within the TSML toolkit while keeping accuracy high. NSGA-II includes parameters for tuning that I will allow to be configurable from J48SS. These include maxGap, maxPatternLength, maxTime, minSupport, patternWeight and useIGPruning.

A downside with the design of J48SS is that time series and sequential data must be specially labelled in order for the classifier to process it correctly. This involves editing ARFF files to place all attributes (except the class attribute) into a single string of comma-separated values for each instance. All but one attribute tag must be removed from the file header and that tag must be edited to identify the data as either a time series string or sequential string, which is picked up by the classifier once the data is in an Instances object.

There are two major design goals that I wish to accomplish from this project. The first is to be able to tie together the three different components described above into a working system. The second is to ensure maintainability of the code through Javadoc comments, code commenting and removal of all unnecessary files. Further goals are explained in the MoSCoW analysis, below.

[6] *http://jmetal.sourceforge.net/*

2.2 UML

Throughout development, the UML for J48SS has changed a large amount. My initial plan was simple, based off of the diagrams given in the J48SS paper. As I worked further on development, I was able to better define how classes would link together. Although this is not usually how UML would be used, I did find it useful to be able to iterate on the UML as I went, and refer to my current UML diagram to remember how some classes interacted with others. Below you will find the UML diagrams as they exist at project submission, though as this is what I would consider a 'living project', I imagine some further changes to methods to occur in the weeks following submission. Figure 2 shows the classes responsible for creating the decision tree of J48SS. Note that classes `pNSGAII_bestShapelet` and `AlgoVGENModified` are shown here as single classes, though these are only the entry points to their respective algorithms. Figure 3 shows the jMetal portion of the classifier which houses the NSGA-II algorithm for time series shapelet classification. I've done my best to untangle the web of classes, removing approximately half of all classes drawn in my original class diagram, but some issues of confusing composition and dependency still remain. 4 details the SPMF portion of the classifier, which houses the VGEN algorithm for classification of sequential data. This diagram has also changed significantly throughout development, being reduced from nine classes to the five shown in the diagram, leaving relationships between classes here easier to understand than the rest of the project.

2.3 Pseudocode of the Classifier

The core of the J48 algorithm can be described in the below Pseudocode.

Algorithm 1 J48(**D, A**) **return T**

Require: Dataset **D**, formed of instances each containing a set of attribute values and one class attribute, and list **A** of attributes to test.

Ensure: **T**, The fully-formed decision tree of nodes.

```
1:  T ← null
2:  if all D instances have the same class value or A := empty then
3:      branch ← D                                   ▷ the value of the leaf node
4:      T.add(branch)
5:      return T
6:  maxGain ← 0
7:  aSplit ← null                                    ▷ attribute to split on
8:  e ← FindRootEntropy(D)
9:  for attribute a ∈ D do
10:     g ← FindInfoGain(a, e)
11:     if g > maxGain then
12:         maxGain ← g
13:         a ← aSplit
14: A ← A.remove(a)
15: for each value v ∈ a do
16:     S ← all instance of D with a.value
17:     branches ← J48(S, A)
18:     T.add(branches)
19: return T
```

Brunello et al. (2019) has defined a modification to the J48 algorithm defining how J48SS approaches the task of splitting the tree, as can be found below.

Algorithm 2 nodeSplit(**N**) **return T**

Require: Node **N**, containing data to be split.
Ensure: **T**, The fully-formed decision tree of nodes.

```
1:  if N instances have the same class value or other stopping criteria met then
2:      make N a leaf node
3:  else
4:      best_attr ← null
5:      best_ng ← 0
6:      for each attribute a ∈ N do
7:          if a.type := numeric or categorical attribute then
8:              a_ng ← measureInformationGain(a)
9:              if a_ng > best_ng then
10:                 best_ng ← a_ng
11:                 best_attr ← a
12:         if a.type := sequential string attribute then
13:             pat, pat_ng ← VGEN(a)
14:             if pat_ng > best_ng then
15:                 best_ng ← pat_ng
16:                 best_attr ← pat
17:         if a.type := time series string attribute then
18:             shap, shap_ng ← NSGA-II(a)
19:             if shap_ng > best_ng then
20:                 best_ng ← shap_ng
21:                 best_attr ← shap
22:     children_nodes ← splitData(N.instances, best_attr)      ▷ split on best_attr
23:     for each child_node in children_nodes do
24:         R ← nodeSplit(child_node)
25:         N.add(R)
26: return N
```

The core of VGEN requires understanding of Generator Patterns. Fortunately, Fournier-Viger et al. (2014) explains with the below pseudocode how the core of the algorithm works. Note, that for use within J48SS, we are only interested in the most useful (highly

discriminative) pattern rather than the most frequent pattern, so prior to the steps below we split the data to find the information gain, then filter these results further.

Algorithm 3 patternEnumeration(**D**, *minsup*) **return P**

Require: Collection **D**, the sequence database to be searched and *minsup*, the minimum support that the extracted sequential patterns must have.

Ensure: **P**, a collection of the most frequent patterns from **D**, determined by *minsup*.

```
1: F ← findFrequentItems(D)                    ▷ get the list of frequent items
2: for each item s ∈ F do
3:     L ← findLexicallyLarger(F, s)            ▷ get all values of F greater than s
4:     P ← search(s, F, L, minsup)              ▷ see below algorithm
5: return P
```

Algorithm 4 search(*pat*, **F**, **L**, *minsup*) **return P**

Require: *pat*, the patterns to be filtered; **F**, the list of frequent items; **L**, the set of items from **F** that are lexically larger than *pat* and *minsup*, the minimum support that the extracted sequential patterns must have.

Ensure: **P**, a collection of the most frequent patterns from **D**, determined by *minsup*.

```
1: sTemp ← 0
2: iTemp ← 0
3: for each item j ∈ F do
4:     if the s-extention of pat is frequent then
5:         sTemp ← sTemp ∪ j
6: for each item j ∈ sTemp do
7:     L ← findLexicallyLarger(sTemp, j)    ▷ get all values of sTemp greater than j
8:     P ← search(the s-extension of pat with j, sTemp, L, minsup)
9: for each item j ∈ L do
10:    if the i-extention of P is frequent then
11:        iTemp ← iTemp ∪ j
12: for each item j ∈ iTemp do
13:    L ← findLexicallyLarger(iTemp, j)    ▷ get all values of iTemp greater than j
14:    P ← search(the i-extension of P with j, sTemp, L, minsup)
15: return P
```

NSGA-II must be modified to handle data coming in and going back out to the classifier. One of these modifications is to the Crossover Operator, which is defined in Brunello et al. (2019) as taking two lists of binary arrays and swapping the 'tails' of the arrays based on a random index. I believe this will be accomplished with the following operation.

Algorithm 5 shapeletCrossover(**P1**, **P2**) **return C1**, **C2**

Require: **P1** and **P2**, binary array 'parent solutions' which represent shapelets to be crossed.

Ensure: **C1** and **C2**, child solutions of the inputs that represent the shapelets after being crossed.

```
1:  C1, C2 ← null
2:  maxIndex ← 0
3:  if P1 > P2 then
4:      maxIndex ← P2.length - 1
5:  else
6:      maxIndex ← P1.length - 1
7:  p1Index ← randomNumber(0, 1) * maxIndex          ▷ get random points to cross
8:  p2Index ← randomNumber(0, 1) * maxIndex
9:  c1Tail ← P2.copyArray(p1Index, P2.length)
10: c2Tail ← P1.copyArray(p2Index, P1.length)
11: c1Head ← P2.copyArray(0, p1Index)
12: c2Head ← P1.copyArray(0, p1Index)
13: C1 ← JoinArray(c1Head, c1Tail)  ▷ form new shapelets from the crossover points
14: C2 ← JoinArray(c2Head, c2Tail)
15: return C1, C2
```

2.4 MoSCoW Analysis

This proved to be an impossible section to write before I narrowed down my choice of core project to implementing J48SS, as the different options available to me would result is wildly different lists of items to implement. Due to the timescales involved, my MoSCoW analysis was only really completed when I was already developing the features listed, which I agree is not conventional or best practice, though I did feel it

still allowed me to keep track of what I felt I needed to accomplish given the time I had available. The final version of my MoSCoW Analysis can be found in Table 1.

It is the highest importance for me to ensure that I submit a working classifier that can process all four types of attribute to the TSML toolkit GitHub repository, as this work could be of use in further research in time series classification. To this end I also feel it's important to ensure the code is easy to maintain in the future, with comments in the code and Javadoc comments used throughout. I wish to perform tests with sequential and time series specific data to ensure the classifier is working as expected, comparing the time series results to those listed in the original J48SS paper to ensure the classifier is working as expected. It would be nice to test the classifier in an ensemble however I will leave that as an extra feature should I complete general testing quickly. I don't think I will produce a set of tests on the quality of results for numeric and continuous data as I don't feel it will be very insightful as data in these formats is processed in the same way as J48, offering no benefit over that classifier. As the strength of J48SS is in the ability to test on multiple attribute types within the same execution, it would be interesting to see how it performs in this task, though I feel it may be difficult to find suitable test data. From looking at the TSML codebase, I can see that a feature to read .ts time series data files has recently been added, within the `TimeSeriesInstances` class, I would like to utilise this class so that it would not be necessary for users to edit their data files before use.

2.5 Development of Initial Classifier Components

Due to there being little time to spare between learning how to implement Decision Trees and implementing J48SS into the TSML toolkit, it was important that sufficient planning was in place as early as possible. Since I felt it would be difficult to actually implement the tree beforehand without the necessary knowledge, I felt I could best focus my efforts on understanding the VGEN and NSGA-II algorithms in their current implementations.

2.5.1 VGEN

SPMF, which contains VGEN, is licensed under GNU GPL v3 (FSF, 2007), which fortunately allows for free sharing, modification and redistribution of the codebase. I proceeded to download this library and explore how VGEN works, feeding in data and

stepping through the program using a debugger. The code used eight classes to process the patterns and the classes were quite tightly coupled together. I felt this was an issue as I wanted to avoid adding as many classes to TSML as possible to avoid bloat, but wanted to ensure the program would still give the correct results when given the same data. To this end, I decided to start a new project and import just the files that tie into generating the sequential generator patterns, then slowly remove methods and classes that were used in doing so. By separating this into a new project I was able to test changes to the classes through trial-and-error.

2.5.2 NSGA-II

jMetal, which contains the NSGA-II algorithm, is licensed on GitHub[7] under the MIT License for free sharing, modification and redistribution. As with VGEN, the first week with this algorithm was spent looking through the codebase and understanding how VGEN works. However, this proved to be far more difficult than I expected. NSGA-II has multiple variants within the jMetal framework and even the standard form interfaces with at least thirty other classes in multiple ways. In addition, NSGA-II is a parallel algorithm which was incredibly difficult to understand, as I had been exposed to very little threaded Java code in the past.

I cloned the codebase and began by removing classes that I could see were not being used by the algorithm. In some cases I could see inheritance structures where only one inherited class was being used by the algorithm, so I could move the implemented methods from the implemented class into the implementing class before removing the inheritance and deleting the original implemented class. This was integral to my goal of ensuring the code is maintainable going into the future, As I don't intend for the jMetal code to be expanded upon at all in the future, only that it processes shapelets as defined in the J48SS paper.

2.6 Classifier Implementation

I felt fortunate that the Machine Learning labs had a focus on classification trees and featured the J48 algorithm, which gave me practical experience with modifying these algorithms. Following completion of first five Machine Learning module lab sheets, I

[7] *https://github.com/jMetal/jMetal*

began to adapt the J48 classifier code to work with the above components. From reading the J48SS paper I expected that it would be a matter of simply implementing the features as laid out in the paper, however this turned out to be a much more difficult procedure than initially estimated. Data entering the two algorithms required some significant modification and transformation before it could be used by J48SS, and it appeared that the process was not as clearly documented in the paper as I was initially led to believe, at least for a reader with no prior knowledge of extracting sequential generator patterns or shapelet extraction. Development was further stalled due to coursework requirements from other projects so I decided to concentrate on trying to get NSGA-II working correctly at least, since to me that was the more important part of the classifier. Even so, several weeks passed with frustratingly slow progress being made.

It was fortunate, then, that in the final few weeks of the project that I came upon the stand-alone modified Weka classifier published by the paper's lead author[8]. This gave me two useful leads in completing the project. Firstly, I was able to run the datasets mentioned in the J48SS paper on the exact algorithm implemented for the paper, meaning I could verify the results I had matched those in the paper meaning my edits to the datasets were correct. Secondly, it meant I could build my own, simple toy datasets and run them in both the original algorithm and my copy to start to nail down what my algorithm was doing incorrectly. This allowed me some further progress, but swift progress was made when I was able to decompile the key class files within this build of Weka to see the exact implementation of the classifier. The biggest realisation I had was that this implementation of J48SS was using an older version of jMetal (v4.5), which had a completely different layout to the version I was using in this project (v5.10). I began from scratch again with the older version of jMetal, applying some of the same changes I had done previously to this new code and patching in parts of the decompiled Weka code where I was still uncertain until finally, I had a working classifier. Similar patching was done on the remaining areas of code over the following week and once I verified that my results were matching the ones coming from the stand-alone program, I moved on to analysing the data. The downside here is that some of the decompiled code remains untouched in my submitted version, which is poorly constructed and lacking in comments. This was a conscious decision made in order to begin evaluation of the working classifier as soon as possible.

[8] *https://github.com/dslab-uniud/J48SS*

3 Analysis

3.1 Code Functionality

The codebase for the J48SS classifier is included with this project within the tsml.jar Java Archive file. Also included is the testing scripts used for generating the evaluation, as well as the data used in those tests. In addition, the classifier will be available on my branch of the TSML Java toolkit[9] upon submission, with the aim of having a pull request certified in the near future as of this writing. Examples of part of the testing code can be found below.

Listing 1: Testing code used to perform a grid search tuning on J48SS, utilising the TunedClassifier and ParameterSpace classes of TSML.

```
private static void performGridSearch(ParameterSpace ps,
 AbstractClassifier c, String[] datasetnames) throws
 Exception {
  DateTimeFormatter dtf = DateTimeFormatter
   .ofPattern("yyyy/MM/dd HH:mm:ss");

  String tuningLocation = DatasetTools.datasetLocation +
   "/_results/tuning/";
  DatasetTools.createDirIfNotExists(tuningLocation);

  for (String problem : datasetnames) {
    LocalDateTime now = LocalDateTime.now();
    System.out.println(dtf.format(now) + ": Starting J48SS" +
     "tuning on " + problem + " dataset");
    TunedClassifier tcGrid = new TunedClassifier(c, ps);
    tcGrid.setCloneClassifierForEachParameterEval(false);

    Instances data = DatasetTools.loadARFFData
     (DatasetTools.datasetLocation + "/" + problem +
     "/" + problem + "TS_ALL.arff");
    tcGrid.buildClassifier(data);
```

[9] https://github.com/uea-machine-learning/tsml/tree/martin-dev

```
    String[] bestOptions  = tcGrid.getBestOptions();
    ClassifierResults res = tcGrid.getTrainResults();

    String filePath = tuningLocation + "/" + problem;
    DatasetTools.writeOptionsToFile(filePath +
     "bestoptions.txt", bestOptions);
    DatasetTools.writeTestResultsToFile(filePath +
     "results.csv", res);
  }
}
```

Listing 2: Testing code to perform cross validation using the CrossValidationEvaluator and ClassifierResults classes of TSML.

```
private static void performCrossValidation(
 CrossValidationEvaluator cv, String[] datasetNames)
 throws Exception {
  DateTimeFormatter dtf = DateTimeFormatter
   .ofPattern("yyyy/MM/dd␣HH:mm:ss");

  for (String problem : datasetNames) {
    Instances data = DatasetTools.loadAndConcatARFFData
     (DatasetTools.datasetLocation + "/" + problem +
     "/" + problem);
    Classifier[] classifiers = new Classifier[]
     {new LearnShapelets()}; // do more if time allows

    String experimentsDir = DatasetTools.datasetLocation +
     "_collated/Predictions/" + problem + "/";
    DatasetTools.createDirIfNotExists(experimentsDir);

    for (Classifier c : classifiers) {
      LocalDateTime now = LocalDateTime.now();
      System.out.println(dtf.format(now) + ":␣Starting␣" +
       c + "␣on␣" + problem + "␣dataset.");
```

```
        ClassifierResults res =
         cv.crossValidateWithStats(c, data);
        String cName = res.getClassifierName();
        DatasetTools.writeTestResultsToFile(experimentsDir +
         cName + "_results.csv", res);
    }

    // repeat for the TS_ data to test j48ss classes
    data = DatasetTools.loadARFFData(DatasetTools
     .datasetLocation + "/" + problem + "/" + problem +
     "TS_ALL.arff");

    J48SS tunedJ48ss = DatasetTools.bestJ48ssSettings
     (problem); // load settings from tuning
    classifiers = new Classifier[] {new J48SS(),
     tunedJ48ss};

    int cNum = 0; // Since these classifiers are both
     "J48SS", we need a way to tell them apart
    for (Classifier c : classifiers) {
        LocalDateTime now = LocalDateTime.now();
        System.out.println(dtf.format(now) + ":␣Starting␣" +
        "J48SS_" + cNum + "␣on␣" + problem + "␣dataset.");
        ClassifierResults res =
         cv.crossValidateWithStats(c, data);
        String cName = res.getClassifierName();
        DatasetTools.writeTestResultsToFile(experimentsDir +
         cNum + "_" + cName + "_results.csv", res);
        cNum++;
    }
}
LocalDateTime now = LocalDateTime.now();
System.out.println(dtf.format(now) +
":␣Processing␣complete.");
```

}

Documentation of the project was achieved largely through Javadoc comments where necessary, which have been archived into HTML format and included with this distribution. I have done my best to document as much of the project as possible knowing that it will need to be maintained by others in the future. Legal notices on each class were also added to ensure each page remains consistent with the rest of TSML.

Although the codebase being submitted is not perfect regarding commentary, it's readable for the most part and I have the full intention of ensuring the version being submitted to the TSML codebase to be fully updated with commentary added in the weeks following project submission.

An additional feature that I added during development was to hide messages from NSGA-II telling the user about shapelet details found and number of instances being scanned as they could sometimes print at several per second, making them useless outside of debugging. These messages can be toggled back on by calling `j48ss.setVerbosity(true)` before building the classifier.

3.2 Data Description

The second part of my project involves testing my implementation of J48SS to check it's classification efficacy. The data used for this task was a set of twenty randomly selected datasets from the "tscProblems85"[10] collection in the UCR UEA archive, which includes some sets used in J48SS paper. Data comes pre-separated into train/test sets but to avoid over-fitting, I have chosen to combine them and perform a 10-fold cross validation per data set in all experiments. The names and characteristics of these data sets can be found in Table 2. All sets were specifically adapted for J48SS by transforming all attributes into a single string, aside from the class attribute, and prepending TS_ to the attribute name. This enables J48SS to pick up that the attribute is of a time series type and correctly perform shapelet transformations on it. These files can be found with others the zip file with this report.

[10] *https://github.com/uea-machine-learning/tsml/blob/master/src/main/java/experiments/data/DatasetLists.java*

3.3 Classifier Description

The J48SS classifier with default parameters was tested alongside a version of J48SS tuned for each dataset and a FastShapelets classifier with default settings. It was my intention to also test LearnShapelets and ShapeletTree classifiers with default settings here too, but sadly initial tests showed these would take too long to process on my selected data sets and results would not ready by the time this report was due. I would have liked to have tested ShapeletTransformClassifier against an ensemble of J48SS classifiers but again, a simple test showed this classifier would not process all twenty datasets in time to allow for a thorough analysis.

3.4 Testing Methodology

The first test carried out was to tune the J48SS on each time series dataset to find the best parameters for later tests. I also looked to compare my tuning to those conducted in the original paper to see if our results match and if not, to perhaps explore why. Initially the plan was to outdo the J48SS paper by tuning my J48SS on a grid search of four of the five custom parameters for shapelet extraction: crossover probability, mutation probability, number of evaluations and weight used for extraction of the final shapelet. I quickly found, however, that this would be too many settings to allow tuning to be done in a reasonable timeframe, so I decided to pare this back to just the two parameters featured in the original paper, as shown in Table 3. Each grid search result was trained once on a ten fold cross validation of each dataset.

The second test is a evaluation of J48SS, both tuned and untuned, against Fast-Shapelets. This is to evaluate how effective tuning is as well as how effective J48SS is at classification against the original implementation, as well as against another Shapelet classifier. Tests will also be timed to see if any classifier has an advantage there. My null hypothesis here is that the tuned J48SS should be more accurate compared to its untuned variant by a statistically significant margin. I also expect to see J48SS be more accurate than FastShapelets by a statistically significant margin, as well as perform significantly faster due to the parallel capability of the NSGA-II implementation. I utilised `TunedClassifier` for tuning and wrote `ClassifierResults` to file for later comparison.

4 Evaluation

4.1 Analysis of Results

4.1.1 Tuning

Table 4 presents the results from tuning J48 with the parameters listed in Table 3. If we look a table 5 of the original paper by Brunello et al. (2019), we can see that their tuning resulted in different values for patternWeight and numEvals. Adiac was 0.4 and 800, respectively, whereas I found 800 and 0.9 to give the greatest accuracy. With Coffee, the paper found 800 and 0.0 to be the best fit whereas I found 800 and 0.9 the most accurate. Gunpoint was 800 and 0.8 on the paper but my testing found 400 and 0.8 to be the best. On Lightning7, both our best tuning parameters matched. MedicalImages was a close match, with the paper finding 400 and 0.2 as the best and I found 400 and 0.4 the most accurate. With Symbols, we were close again at 800 and 1.0 for the paper and 800 and 0.9 for me. Finally, on the Trace dataset, there was a difference in results as the paper found 400 and 0.5 to be the best but I found 800 and 0.7 delivered the highest accuracy. Most of these results are within a margin of error, but it is unusual that some parameter values are quite different. One explanation of this is that variations introduced by randomness in the shapelet crossover calculation drove some results to be slightly better or worse in certain cases. There is also the possibility that my implementation is flawed in a way I have yet to pick up on, which we may find from evaluating the second test results.

4.1.2 Classifying

It was as I write this last section on evaluating my classifier in the final two days of the project that I realised I made three fundamental errors with my testing strategy.

Firstly, I made the decision to use a class I wasn't familiar with called CrossValidationEvaluator in order to produce my cross validation results, rather than use code I'd written previously and confirmed was correct. Briefly browsing the CrossValidationEvaluator code, I assumed that it would save the results of all ten cross validations to a ClassifierResults object that I could save to disk using code I've written in the past. This turned out to be wrong, and in fact only the final fold was saved to disk. Whether the data was stored in the object and I failed to recall it correctly or whether the object

only stored one copy, I'm unsure of, but either way the test results were incorrectly saved.

Secondly, I made a mistake in the folder structure used to save the results. Instead of saving each classifier within a collated folder and saving the results of each dataset in individual folders within a predictions folder inside the classifier folder, I accidentality saved my dataset folders inside the collated folder and each classifier as a folder within them. At the time I thought it wouldn't be an issue, but even if I had not made my first mistake, this one would have ensured using the CollateResults.collate() tool would have been much more difficult. With this mistake, I was unable to automatically retrieve results for Balanced Accuracy, NLL, AUROC and so on. Although it's possible to retrieve this data manually from the saved ClassifierResults data, in my case it would not be useful due to the first mistake.

Thirdly, I recognise that I did not leave enough time during the end of my project to rerun these experiments, should I come across issues like the above. Much of the final two weeks was spent scrambling to ensure the classifier was fit for purpose and predicting correctly that I used up the time I had put aside for any issues. On top of this I had already made mistakes not previously mentioned in this report, such as running incorrect classifiers or incorrectly setting certain parts of classifiers or code, leading to crashes that went undiscovered for many hours. These things combined burned into the small amount of leeway I had remaining, leading to the above two mistakes to not leave me enough time to re-evaluate.

Should my experiments have been successful, this section would have explored the accuracy, balanced accuracy, Negative Log Likelihood and AUROC statistics, explaining what each of the figures represents and using plots to explore the differences in the classifiers tested. A Critical Difference Diagram would be used to show if there was a difference between the classifiers in the mentioned statistics, and by using a Friedman test I would explore if the differences were statistically significant enough to break out of critical distance cliques and either accept or reject my null hypotheses stated in the testing methodology section.

The results files generated, as well as the flawed testing code, have been included with this report for your perusal.

4.2 Project Analysis

The brief given when choosing a potential project mentioned described contributing to an open source toolkit to assist the Time Series Machine Learning group at UEA. With the submission of my classifier, I hope I have achieved this goal. However without the data to understand how useful the classifier is at this time, the usefulness of this contribution is questionable in my mind. This was by far the most challenging project I have worked on, both in terms of difficulty of completing the work as well as trying to balance the workload with other ongoing commitments. Until now I did not touch on the fact that early tasks I wished to compete were missed, such as making a basic change to the toolkit and updating documentation. These were tasks that always seemed to be pushed back each week as I focussed on other work, until I felt it was too late to begin doing them. Submissions of the Literature Review and progress report were both missed due to issues I was having at the time, and not making contact with my advisor to explain this made it difficult to get back on track with the first half of this project. What work was completed on these things, however, was later expanded upon and added to this report.

Table 5 details how much of the MoSCoW table was achieved during the second semester. As you can see, seven of the twelve objectives were completed fully or close to it, which include submitting the classifier to the repository, adding the ability to process sequential and time series data to the classifier, adding parameters to tune those additions and tuning it. In addition, the code was commented and easy-to-read, though I feel this may be an ongoing process within the toolkit. One of the twelve objectives was attempted, to test the classifier against time series data, but was not successful as detailed above. The remaining four objectives that were not completed were to test with sequential data, to test as part of an ensemble, to test on multiple attributes in one dataset and to implement the ability to handle .ts data files using the TimeSeriesAttributes class. I would be interested in attempting to integrate TimeSeriesAttributes objects into the classifier in the future should I have the time.

My preliminary testing showed an increase in accuracy using the time series formatted data compared to the numerical version of the same data. It was an early intention to be able to run the classifiers on both versions of each data set for comparison but sadly these experiments were not completed in time for inclusion in this report. I also wished I had the time to test against more classifiers, especially the shapelet classifiers

`LearnShapelets`, `ShapeletTransformClassifier` and `ShapeletTree`, though sadly the issues during development left me only a week to run experiments including model tuning, which was not enough time to get a full twenty datasets evaluated with these much slower algorithms. This was the case even before I realised the error in testing.

I realised after starting the evaluation testing that was probably a mistake to use different data from the J48SS paper. If the same datasets were used it would have allowed a much clearer comparison between my implementation and the original - it would have been interesting to see if my results compared to the original within an acceptable margin of error.

My other modules ended up taking up far more of my time that I wished they had done, and I feel I put in more effort to projects in those modules than what they warranted. With the time I could've saved from not worrying about those projects as much, it's likely I could have had this one working sooner leaving me more time for testing and possibly avoiding the issues I had at the last minute.

4.3 Further Changes and Additions

Figure 3 shows an issue I have remaining with the system. I still feel there is more that can be done to reduce complexity, especially when it comes to the instance where BestShapeletProblem creates a BestShapeletSolutionType object, but the BestShapeletSolutionType class contains a method which takes a BestShapeletProblem object as a parameter, leading to confusing areas of composition and dependencies. This is something I would be interested in working on in the toolkit in the future.

In the time since the publication of the original J48SS paper, an update to the NSGA-II algorithm has been published in jMetal called NSGA-III (). Although my knowledge and research on this algorithm is limited, I can see that it includes improvements that could be of use to this implementation of J48SS. If I had time during the project, I would have been interested in implementing this algorithm and running experiments to compare the use of this revision to the implemented version, but sadly time is always the largest restriction to any project. Nevertheless, this finding does leave the door open to further development in the future.

After completing this project and my Machine Learning module labs and coursework project, I regret that I didn't pick to create a visualisation tool for TSML. As it stands, I

feel the toolkit could benefit greatly from a native Java system for generating visual data quickly and easily from `ClassifierResults` objects or CSV results files, as this is a necessity for allowing authors of papers and reports to easily present results, and allowing readers to interpret raw data. I've had an interest in expanding my knowledge of Java to the creation of imagery (graphics, posters and general image design) but have yet to give it a try. With TSML being an Open Source project, I feel motivated to learn how to create visualisations in my own time knowing that if my code is good enough for use, it could benefit other people in the future.

An addition that could be useful to TSML in a future project is the ability to run experiments on multiple datasets concurrently. Figure 5 shows how the processing of a dataset using the `FastShapelets` classifier is restricted to the use of a single core. The ability to leverage the remaining cores to process other single-thread classifiers would allow research such as grid search tuning, for example, to be carried out on more parameters or more datasets in the same amount of time, allowing for more refined models to be used. The downside of this is that you could not guarantee the accuracy of timing experiments using this technique as each core will generally process data at different speeds due to natural variances in the CPU architecture. In addition, this technique will require far more memory to be required. Experiments with `FastShapelets` showed most datasets requiring 2.5GB to 3.5GB of memory each, so running seven or eight of these experiments in tandem, even with 32GB of memory available, could be enough to force the swap file to move data onto the hard drive and massively slow operations. Care would need to be taken to ensure that a balance is maintained, which could be difficult to design around.

5 Conclusion

Although there were many challenges to overcome with this project, I feel proud of the project I've documented in this report and I wish I had the entire year to work on this project without work from the other modules taking my time away from it, as I feel there was so much more I could have contributed to the TSML toolkit. I'm hoping that I will have such time, at least in the near future, to contribute further enhancements and play a small role in furthering the research into time series machine learning.

References

Abdi, H. and Williams, L. J. (2010). Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459.

Aghabozorgi, S., Ying Wah, T., Herawan, T., Jalab, H. A., Shaygan, M. A., and Jalali, A. (2014). A hybrid algorithm for clustering of time series data based on affinity search technique. *The Scientific World Journal*, 2014.

Aha, D. and Kibler, D. (1991). Instance-based learning algorithms. *Machine Learning*, 6:37–66.

Bagnall, A., Flynn, M., Large, J., Lines, J., and Middlehurst, M. (2020). A tale of two toolkits, report the third: on the usage and performance of hive-cote v1. 0. *arXiv e-prints*, pages arXiv–2004.

Bagnall, A., Király, F., Löning, M., Middlehurst, M., and Oastler, G. (2019a). A tale of two toolkits, report the first: benchmarking time series classification algorithms for correctness and efficiency. *arXiv preprint arXiv:1909.05738*.

Bagnall, A., Large, J., and Middlehurst, M. (2019b). A tale of two toolkits, report the second: bake off redux. chapter 1. dictionary based classifiers. *arXiv preprint arXiv:1911.12008*.

Bagnall, A., Lines, J., Bostrom, A., Large, J., and Keogh, E. (2017). The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances. *Data mining and knowledge discovery*, 31(3):606–660.

Ben-Dor, A., Shamir, R., and Yakhini, Z. (1999). Clustering gene expression patterns. *Journal of computational biology*, 6(3-4):281–297.

Breiman, L. (1996). Bias, variance, and arcing classifiers. Technical report, Tech. Rep. 460, Statistics Department, University of California, Berkeley ....

Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group, Belmont, California.

Brownlee, J. (2020a). 10 clustering algorithms with python. **URL:** *https://machinelearningmastery.com/clustering-algorithms-with-python/*.

Brownlee, J. (2020b). A gentle introduction to k-fold cross-validation. **URL:** *https://machinelearningmastery.com/k-fold-cross-validation/*.

Brunello, A., Marzano, E., Montanari, A., and Sciavicco, G. (2019). J48ss: A novel decision tree approach for the handling of sequential and time series data. *Computers*, 8(1):21.

Chang, C.-C. and Lin, C.-J. (2011). Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM.

de Camargo, U. M., Somervuo, P., and Ovaskainen, O. (2017). Protax-sound: A probabilistic framework for automated animal sound identification. *PLOS ONE*, 12(9):1–15.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197.

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research*, 7:1–30.

Durillo, J. J. and Nebro, A. J. (2011). jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771.

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

Fournier-Viger, P., Gomariz, A., Šebek, M., and Hlosta, M. (2014). Vgen: fast vertical mining of sequential generator patterns. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 476–488. Springer.

Fournier-Viger, P., Lin, J. C.-W., Gomariz, A., Gueniche, T., Soltani, A., Deng, Z., and Lam, H. T. (2016). The spmf open-source data mining library version 2. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 36–40. Springer.

Freund, Y. and Mason, L. (1999). The alternating decision tree learning algorithm. In *Proceeding of the Sixteenth International Conference on Machine Learning*, pages 124–133, Bled, Slovenia.

Freund, Y., Schapire, R., and Abe, N. (1999). A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612.

Friedman, J., Hastie, T., and Tibshirani, R. (1998). Additive logistic regression: a statistical view of boosting. Technical report, Stanford University.

FSF (2007). The gnu general public license v3.0. **URL:** *https://www.gnu.org/licenses/gpl-3.0.en.html*.

Gama, J. (2004). Functional trees. 55(3):219–250.

Google (2020). Introduction to transforming data. **URL:** *https://developers.google.com/machine-learning/data-prep/transform/introduction*.

Gou, J., Ma, H., Ou, W., Zeng, S., Rao, Y., and Yang, H. (2019). A generalized mean distance-based k-nearest neighbor classifier. *Expert Systems with Applications*, 115:356–372.

He, G., Xin, X., Peng, R., Han, M., Wang, J., and Wu, X. (2020). Online rule-based classifier learning on dynamic unlabeled multivariate time series data. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*.

Huilgol, P. (2020). 9 feature transformation & scaling techniques: Boost model performance. **URL:** *https://www.analyticsvidhya.com/blog/2020/07/types-of-feature-transformation-and-scaling/*.

Jin, Z., Sun, Y., and Cheng, A. C. (2009). Predicting cardiovascular disease from real-time electrocardiographic monitoring: An adaptive machine learning approach on a cell phone. In *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 6889–6892.

Lakshmanan, S. (2019). How, when, and why should you normalize / standardize / rescale your data? **URL:** *https://towardsai.net/p/data-science/how-when-and-why-should-you-normalize-standardize-rescale-your-data-3f083def38ff*.

Le, J. (2020). The top 10 machine learning algorithms every beginner should know. **URL:** *https://builtin.com/data-science/tour-top-10-algorithms-machine-learning-newbies*.

Littlestone, N. (1988). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine learning*, 2(4):285–318.

Lutes, J. (2021). Entropy and information gain in decision trees. **URL:** *https://towardsdatascience.com/entropy-and-information-gain-in-decision-trees-c7db67a3a293*.

MacQueen, J. et al. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA.

Malhotra, P., TV, V., Vig, L., Agarwal, P., and Shroff, G. (2017). Timenet: Pretrained deep recurrent neural network for time series classification. *arXiv preprint arXiv:1706.08838*.

Middlehurst, M., Large, J., Flynn, M., Lines, J., Bostrom, A., and Bagnall, A. (2021). Hive-cote 2.0: a new meta ensemble for time series classification. *arXiv preprint arXiv:2104.07551*.

Oliveira, A. L., Braga, P. L., Lima, R. M., and Cornélio, M. L. (2010). Ga-based method for feature selection and parameters optimization for machine learning regression applied to software effort estimation. *Information and Software Technology*, 52(11):1155–1166. Special Section on Best Papers PROMISE 2009.

Paparrizos, J. and Gravano, L. (2015). k-shape: Efficient and accurate clustering of time series. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1855–1870.

Platt, J. (1998). Fast training of support vector machines using sequential minimal optimization. In Schoelkopf, B., Burges, C., and Smola, A., editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press.

Priy, S. (2020). Clustering in machine learning. **URL:** *https://www.geeksforgeeks.org/clustering-in-machine-learning/*.

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

Quinlan, R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA.

Roman, V. (2019). Unsupervised machine learning: Clustering analysis. **URL:** *https://towardsdatascience.com/unsupervised-machine-learning-clustering-analysis-d40f2b34ae7e*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Salzberg, S. L. (1994). C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993.

Shi, H. (2007). Best-first decision tree learning. Master's thesis, University of Waikato, Hamilton, NZ. COMP594.

Statistics-Solutions (2020). How to conduct the wilcoxon sign test. **URL:***https://www.statisticssolutions.com/how-to-conduct-the-wilcox-sign-test/*.

Sumner, M., Frank, E., and Hall, M. (2005). Speeding up logistic model tree induction. In *European conference on principles of data mining and knowledge discovery*, pages 675–683. Springer.

Tyagi, N. (2020). 7 types of regression techniques you should know in machine learning. **URL:** *https://www.analyticssteps.com/blogs/7-types-regression-technique-you-should-know-machine-learning*.

Webb, G. I. (2000). Multiboosting: A technique for combining boosting and wagging. *Machine learning*, 40(2):159–196.

Witten, I. H., Frank, E., and Hall, M. A. (2011). *Data mining: practical machine learning tools and techniques*. Elsevier.

Wu, X., Kumar, V., Quinlan, J. R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G. J., Ng, A., Liu, B., Philip, S. Y., et al. (2008). Top 10 algorithms in data mining. *Knowledge and information systems*, 14(1):1–37.

Yuan, G.-X., Ho, C.-H., and Lin, C.-J. (2012). Recent advances of large-scale linear classification. *Proceedings of the IEEE*, 100(9):2584–2603.

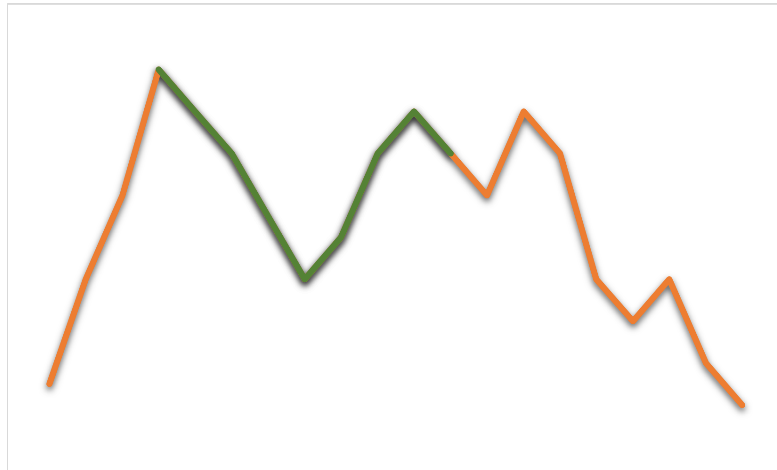

Figure 1: An example segment of time series data with time on the x axis. A shapelet of the series has been highlighted in green.

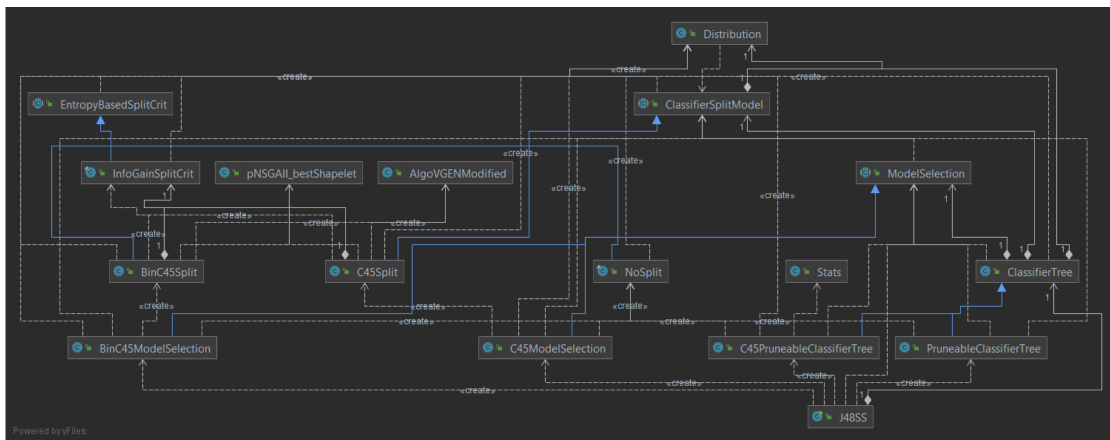

Figure 2: A simplified class diagram of the decision tree segment of the classifier. The root class is J48SS, called by the user.

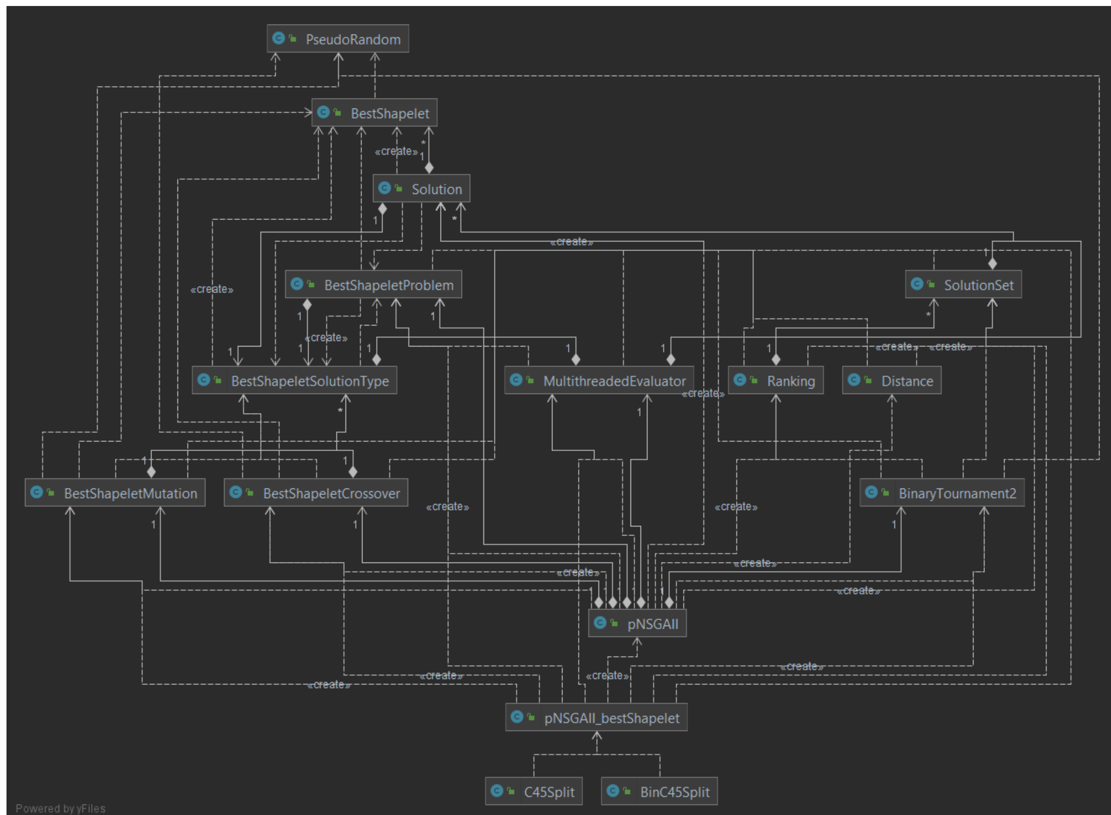

Figure 3: A simplified class diagram of the jMetal segment of the classifier, containing the NSGA-II algorithm. Code enters from the decision tree at C45Split and BinC45Split, depending on if binary splits are used.

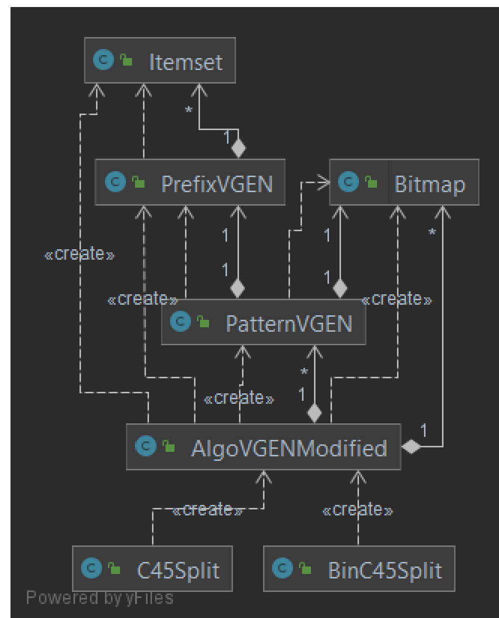

Figure 4: A simplified class diagram of the SPMF segment of the classifier, containing the VGEN algorithm. Code enters from the decision tree at C45Split and BinC45Split, depending on if binary splits are used.

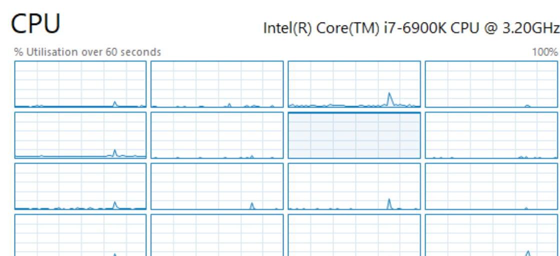

Figure 5: A Screenshot from Windows Task Manager showing CPU usage on an 8 core Intel CPU with HyperThreading running the FastShapelets classifier.

Table 1: MoSCoW Analysis of project requirements

	Requirement	**Priority**
1.	Open a Pull Request on the TSML toolkit GitHub repository submitting a working classifier	Must Have
2.	Integrate the ability to process time series data into the classifier	Must Have
3.	Integrate the ability to process sequential data into the classifier	Must Have
4.	Test the accuracy of the classifier against time series data	Must Have
5.	Test the accuracy of the classifier against sequential data	Must Have
6.	Add the ability to tune parameters for time series and sequential data	Should Have
7.	Ensure the code is correctly commented throughout, including Javadoc commenting	Should Have
8.	Ensure the code imported from other libraries is as easy to read as possible	Should Have
9.	Perform tuning experiments on the classifier against multiple sets of data	Should Have
10.	Perform tests with multiple J48SS in an ensemble against other ensembles on time series data	Should Have
11.	Test the accuracy of the classifier on multiple attribute types in one dataset	Could Have
12.	Implement a way for the classifier to handle .ts files	Could Have
13.	Test the accuracy of the classifier against similar classifiers numeric data	Won't Have
14.	Test the accuracy of the classifier against similar classifiers on continuous data	Won't Have
15.	Perform ensemble tests using numeric or continuous data	Won't Have

Table 2: Data characteristics of the datasets used

Name	Instances	Attributes	Classes	Class Distribution
Adiac	781	176	37	20 / 23 / 20 / 20 / 20 / 20 / 20 / 25 / 20 / 20 / 20 / 26 / 20 / 20 / 21 / 20 / 20 / 22 / 26 / 21 / 20 / 20 / 20 / 23 / 29 / 20 / 20 / 20 / 20 / 20 / 20 / 20 / 20 / 20 / 20 / 25 / 20
Car	120	577	4	30 / 30 / 30 / 30
CBF	930	128	3	310 / 310 / 310
Coffee	56	286	2	29 / 27
FaceAll	2250	131	14	112 / 178 / 176 / 171 / 176 / 187 / 149 / 273 / 140 / 135 / 48 / 106 / 327 / 72
FiftyWords	905	270	50	109 / 91 / 61 / 54 / 38 / 34 / 32 / 24 / 24 / 22 / 22 / 22 / 21 / 18 / 16 / 14 / 14 / 13 / 13 / 13 / 12 / 11 / 10 / 10 / 10 / 10 / 10 / 9 / 9 / 9 / 9 / 9 / 8 / 8 / 8 / 8 / 8 / 8 / 8 / 7 / 7 / 7 / 7 / 7 / 7 / 7 / 7 / 6 / 6
Fish	350	463	7	50 / 50 / 50 / 50 / 50 / 50 / 50
GunPoint	200	150	2	100 / 100
Haptics	463	1092	5	78 / 92 / 93 / 100 / 100
Herring	128	512	2	77 / 51
Lightning7	143	319	7	18 / 17 / 14 / 19 / 15 / 38 / 22
Meat	120	448	3	40 / 40 / 40
Medical Images	1141	99	10	112 / 65 / 75 / 48 / 45 / 23 / 49 / 24 / 106 / 594
OSULeaf	442	427	6	66 / 84 / 75 / 97 / 82 / 38
ScreenType	750	720	3	250 / 250 / 250
SwedishLeaf	1125	128	15	75 / 75 / 75 / 75 / 75 / 75 / 75 / 75 / 75 / 75 / 75 / 75 / 75 / 75 / 75
Symbols	1020	398	6	181 / 162 / 167 / 181 / 162 / 167
Toe Segmentation1	268	227	2	140 / 128
Trace	200	275	4	50 / 50 / 50 / 50
Worms	258	900	5	109 / 44 / 35 / 45 / 25

Table 3: Parameters for tuning J48SS

Name	**Values**	**Default Value**
numEvals	200, 400, 800, 1200	800
patternWeight	0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0	0.75

Table 4: Results from tuning the J48SS classifier

Name	**numEvals**	**patternWeight**
Adiac	800	0.9
Car	1200	0.7
CBF	1200	0.8
Coffee	800	0.9
FaceAll	200	0.8
FiftyWords	400	0.7
Fish	800	0.8
GunPoint	400	0.8
Haptics	200	0.3
Herring	400	1.0
Lightning7	800	0.9
Meat	400	0.5
Medical Images	400	0.5
OSULeaf	400	0.7
ScreenType	200	0.9
SwedishLeaf	1200	0.9
Symbols	800	0.9
Toe Segmentation1	1200	0.6
Trace	800	0.7
Worms	400	0.9

Table 5: Review of objectives listed in the MoSCoW Analysis

	Requirement	**Priority**	**Complete?**
1.	Open a Pull Request on the TSML toolkit GitHub repository submitting a working classifier	Must Have	Yes
2.	Integrate the ability to process time series data into the classifier	Must Have	Yes
3.	Integrate the ability to process sequential data into the classifier	Must Have	Yes
4.	Test the accuracy of the classifier against time series data	Must Have	Attempted*
5.	Test the accuracy of the classifier against sequential data	Must Have	No
6.	Add the ability to tune parameters for time series and sequential data	Should Have	Yes
7.	Ensure the code is correctly commented throughout, including Javadoc commenting	Should Have	Yes
8.	Ensure the code imported from other libraries is as easy to read as possible	Should Have	Yes
9.	Perform tuning experiments on the classifier against multiple sets of data	Should Have	Yes
10.	Perform tests with multiple J48SS in an ensemble against other ensembles on time series data	Should Have	No
11.	Test the accuracy of the classifier on multiple attribute types in one dataset	Could Have	No
12.	Implement a way for the classifier to handle .ts files	Could Have	No
13.	Test the accuracy of the classifier against similar classifiers numeric data	Won't Have	N/A
14.	Test the accuracy of the classifier against similar classifiers on continuous data	Won't Have	N/A
15.	Perform ensemble tests using numeric or continuous data	Won't Have	N/A