FISH: Feature Interpretable Shapelet Histograms

Using shapelets to create interpretable histograms

Emil Dalgaard Moroder
emorod19@student.aau.dk

Fatemeh Shad Bakhsh
fbakhs22@student.aau.dk

Iñigo Sanz Ilundain
isanzi22@student.aau.dk

Jan Mackeprang Damgaard Hansen
jmdh19@student.aau.dk

Kristian Skov Johansen
kjohan19@student.aau.dk

Nicholas Herrmann
nherrm19@student.aau.dk

Computer Science
Aalborg University
Denmark

From Febuary To May - 2023

Abstract

Few-Shot learning (FSL) is a field that has recently seen an increase in attention due to the FSL paradigm having the ability to do regression and classification while only having very few data points. In context, to time series data, the FSL paradigm has shown to be valuable, since many important time series domains contain very few samples. In addition, Shapelets have been shown to be a good representation of differentiating between time series classes. Usually, only a single shapelet is used, however, this paper proposes the FISH method, which combines multiple shapelets into a histogram, while extracting additional information. Moreover, in the domain of time series classification (TSC), it is essential that end-users understand the reasoning behind the classification since it is often used in problems such as heart-disease diagnosis, and predicting heart arrhythmias from ECG data. The proposed method has been shown to perform as well as state-of-the-art (FSL-TSC) models, while also giving interpretable results.

Contents

1 Introduction

The field of machine learning has become more successful and efficient, especially in certain areas such as neural networks [1]. These computational models learn numerous functions that can be used to solve regression or classification problems.

Neural networks can be used with all kinds of data, such as tabular data or images. While such data is rather static, there is also another class of data, being Time-Series (TS) data. This is a type of data where time is a dimension representing a change in data points over time. This type of data has been extensively used in fields such as medicine or agriculture, where the variables are measured in constant periods of time. An issue with this kind of data is that there is usually a lot of it, and most of it is raw data from a sensor, for instance. This brings up the issue of annotating the data since it requires expert knowledge. This is something that is done manually, takes a lot of time and is very tedious. Traditional models usually require a lot of annotated data to train on, however, the question is what to do if there is not such a large amount of annotated data available.

To alleviate this issue of useful training data, an entire field has been created, called Few-Shot Learning (FSL) [2]. FSL is an umbrella term for methods that attempt to generalize data with very few, or zero, annotated data points. There are several ways to do this but there are three general categories being focused on the *Data*, the *Model* or the *Algorithm*[3]. Each of these categories has its strengths and weaknesses, however, they all attempt to contribute to solving the issue of a few data points. The *Data* category will be the general focus of this report.

During the last few years, this FSL field has been extensively investigated and used on TS data. One of the state-of-the-art methods that have been investigated is the principle of finding Shapelets [4]. The general idea of this method is finding a subsequence of the TS data, that best represents the label in that dataset. Another method that has been used a lot is that of Prototypical Networks [5], where a network is trained not to find a specific label of a data point but is trained to map out a latent space that can represent the classes instead. Another method that combines both of these methods is the Dual Prototypical Shapelet Network [6], which tries to combine the inherent interpretability of shapelets with the classification strengths of the Prototypical Networks.

The DPSN model is an important step forward, since it is also interpretable, by using shapelets to reason about classification results. This is critical, especially when it comes to annotating large quantities of raw data since there is a need to verify that the model actually annotated the data correctly.

With all this in mind, this report proposes a novel method called Feature Interpretable Shapelet Histograms (FISH). The idea of this method is to extract as much information, based on shapelets, as possible from a few annotated data points and represent them in a histogram of multiple shapelets. These histograms can then be used to train a machine-learning model to obtain annotations for the raw data.

The structure of the report from here on out will be Section 2 will describe all the concepts

that the FISH method uses. Section 3 will go into detail about how the FISH method works and Section 4 will be experimental results from it. Finally, Section 5 will conclude on the method and bring up ideas for future work.

2 Preliminaries and Related Work

This section will introduce relevant concepts that are required to understand the problem domain and the FISH method. It will explore information theory, time series, interpretability, few-shot learning and state-of-the-art-models. The first four concepts will be used in the FISH method, and the results of the state-of-the-art models will be directly compared with the FISH method.

2.1 Information Gain

Information gain is a metric studied in the field of information theory. Information can informally be described as the level of "surprise" that a given event that occurs from a random variable gives. As an example, take a random variable X whose domain is the following: $dom(X) = \{x_1, x_2, \dots x_n\}$, with associated probabilities $P = \{p_1, p_2, \dots p_n\}$, meaning X takes value x_i with probability p_i. The information found in observing X taking the information of x_i is given in equation 1.

$$I(X = x_i) = \log_2(\frac{1}{p_i}) \tag{1}$$

It can be seen that the information in an observation is inversely correlated with the probability of the value; there is more information associated with observing unusual events. The expected information gain $\mathbb{E}(I(X))$, also called the *entropy* $H(X)$, can be found by the sum of each value's associated information, multiplied by the probability of seeing this value, or equivalently, the probability of receiving this information. This can be seen in equation 2.

$$H(X) = \sum_{p_i \in P} p_i \log_2(\frac{1}{p_i}) \tag{2}$$

Using properties of logarithms, a more regular form of equation 2 can be made as it can be seen in equation 3.

$$H(X) = -\sum_{p_i \in P} p_i \log_2(p_i) \tag{3}$$

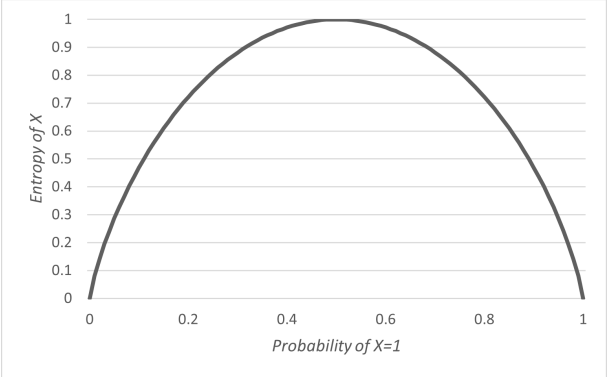


Figure 1: *Illustration of entropy of a random variable $X \in \{0,1\}$, which can take on two values. The figure shows that entropy is greatest when the two outcomes are equally likely.*

Entropy, as defined in equation 3, is a measure of the expected amount of information associated with the observation of a random variable[7]. It can also be thought of as a measure of unpredictability. For example, Figure 1 shows that the entropy of a random variable X, which can be either 1 or 0, is greatest when the two outcomes are equally likely, meaning that X is most unpredictable.

Entropy can be used to gain information on unlabelled data. As an example, take a set of training data $\mathcal{D}_{train} = \{\chi_1, \chi_2 \dots \chi_n\}$ with each data point consisting of random variables $\chi_i = \{X_1, \dots X_n\}$ and having associated label $L_i \in \{l_1, l_2\}$. If an unlabelled datapoint χ_u is then received, consisting of the same set of random variables as the labelled training data, then the label L_u of χ_u is itself a random variable. The task then becomes to learn as much information about L_u, or equivalently, to lower the entropy $H(L_u)$ as much as possible. This is done by observing the random variables $X_1 \dots X_n$ one by one, in an order determined by the amount of information each one reveals. More specifically, the training data are used to calculate the *expected entropy* of a label L after observing a random variable X_j. By observing X_j, the set of training examples can be split into groups; e.g. if X_j is a binary variable, then split the training examples into two groups, and a ternary variable would split them into three groups. If the expected entropy of these groups is lower than the entropy of the complete set of unsorted data points, then information has been gained. The expected entropy of a set of groups is calculated as defined in equation 4.

$$H(L|X_j) = \sum_{i}^{n} q_i H_i \tag{4}$$

where q_i is the quotient found by dividing the number of training examples found in group i by the total number of training examples and H_i is the entropy of group i. The information gain can then be calculated as $\Delta I = H(L) - H(L|X_j)$

2.2 Time Series

Time series data are sequences of data collected over and ordered by time. A common example is a data set consisting of the hourly energy consumption of a household, as shown in table 1. Here, each data point is a measurement of the amount of energy consumed throughout the past hour. As in this example, time series data sets usually consist of data points that are collected at equal time intervals. In the given example, there is only one dependent variable; consumption, which is dependent on time. Time series with this characteristic are called *univariate*. Had there been multiple dependent variables, the time series would be *multivariate* [8].

t	**13:00**	**14:00**	**15:00**	**16:00**	**17:00**	**18:00**	**19:00**	**20:00**	**21:00**	**22:00**
Consumption (kWh)	0.10	0.17	0.15	0.21	0.20	0.25	0.15	0.20	0.28	0.15

Table 1: *Time series of hourly energy consumption of a household, measured in kWh.*

2.2.1 Time Series Analysis

Time series are common in computer science, and therefore, a number of analysis methods have been devised. One pattern which is common in time series analysis is to divide the time series up into subsequences and apply a function to each subsequence in order to extract information from the subsequence. This technique is called *sliding window* since it can be thought of as sliding a window across the sequence and applying the function to what is visible through the window.

A common first step when analyzing time series is to use a sliding window with an averaging function to denoise the data. For instance, using a sliding window of size 3 on the time series shown in table 1, and averaging the data points in the window yields the smoothed time series shown in figure 2. This technique is often called the *rolling average*.

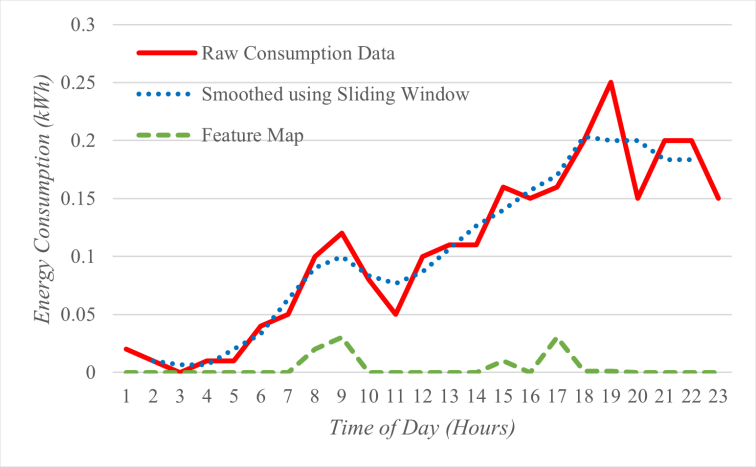


Figure 2: *Effect of using a rolling average with window size 3 to reduce noise in time series data, and feature map found by applying the kernel* $[-0.125 \quad -0.125 \quad 0.5 \quad -0.125 \quad -0.125]$ *to the smoothed data. For the feature map, the y-axis represents how well the given window matches the pattern defined by the kernel.*

Applying a sliding window with an averaging function is a useful preprocessing step, but sliding windows can also be used to extract useful features of the time series. A common operation to apply when using sliding windows for feature extraction is convolution.

2.2.2 Convolutions for time series analysis

In the context of vectors of finite length, the convolution operation is a method for assessing the presence of a pattern in a vector. The pattern is encoded as a weight vector $\boldsymbol{w}$ called the *kernel*, which is then slid across the time series $\boldsymbol{x}$. Formally, the convolution operation is defined as follows[1]:

$$[\boldsymbol{w} \circledast \boldsymbol{x}]_i = \sum_{u=0}^{L-1} \boldsymbol{w}_u \boldsymbol{x}_{i+u} \tag{5}$$

where L is the length of $\boldsymbol{w}$. Equation 5 shows how to calculate the i'th value of the output vector, using the window $[\boldsymbol{x}_i, \boldsymbol{x}_{i+1}, \ldots \boldsymbol{x}_{i+L-1}]$. The convolution is then applied to each window of length L, in order to create the full output of the convolution. Each value i of the output vector shows how well the pattern encoded as $\boldsymbol{w}$ is present in the given window, and for this reason, it is often called the *feature map*. An application of convolution is shown in figure 2. Here, the kernel $[-0.125 \quad -0.125 \quad 0.5 \quad -0.125 \quad -0.125]$ is applied to the smoothed energy consumption data. Intuitively, this kernel finds instances of peaks; it rewards high values in the centre of the window and penalizes high values in the edges. This kernel was able to identify two peaks; one in the morning and one in the afternoon. Of course, convolution can be used to identify much more complex patterns than peaks. Ideally, one would have a

[1]This is actually the cross-correlation operation, but in a machine learning context, the two operations are used interchangeably

machine learning model to learn which patterns are the most useful and the kernels which capture them.

Equation 5 shows how to apply the convolution operation to a one-dimensional vector. This is useful when analyzing univariate time series, but for multivariate time series $\mathbf{X}$, we may want to find patterns across variables. In this case, we can define a two-dimensional kernel $\mathbf{W} \in \mathbb{R}^{W \times H}$, and define the convolution operation as follows:

$$[\mathbf{W} \circledast \mathbf{X}]_{i,j} = \sum_{u=0}^{W} \sum_{v=0}^{H} w_{u,v} x_{i+u,j+v} \tag{6}$$

The principle is the same as the convolution of equation 5. Each entry i, j of the resulting two-dimensional feature map is the result of the elementwise product of the weight matrix $\mathbf{W}$ and the window of $\mathbf{X}$ whose upper left entry is $\mathbf{X}_{i,j}$. The feature map would then show where multivariate patterns occur. An example of a multivariate pattern could be that the first variable, x_1 peaks (as in figure 2), while the second variable, x_2, is increasing.

2.2.3 Shapelets

Another commonly used analysis tool is *shapelets*. Shapelets are time series subsequences that can be used to represent a class [4]. For instance, if we have three classes, each with five time series, we can use a sliding window over all the time series in order to extract shapelets. We can then determine which shapelets best represent each class. This representation could be by appearance frequency; if a particular shapelet appears at least ten times in each time series of class 1, and never appears ten times or more in instances of any other class, this shapelet is representative of class 1.

Ye and Keogh [4] take a more sophisticated approach based on decision trees. First, they define the distance from a shapelet to a time series as:

$$\min_a(d(s, t_a^L)) \tag{7}$$

where t_a^L is the subsequence of time series t which starts at a and has length L, which is also the length of the shapelet s. Here, d is some distance function capable of computing the distance between the shapelet and the time series subsequence. Then, for each potential shapelet in the data set (meaning each subsequence of each time series), the distance from that shapelet to each time series is calculated, and the time series are sorted by this distance, as shown in figure 3. The n classes are then split into two groups, with the split point being chosen based on information gain (see 2.1).

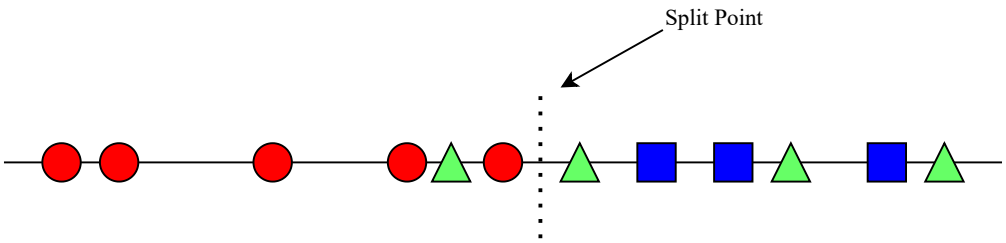


Figure 3: *Illustration of splitting based on information gain. Each shape represents a different class and the line represents the distance from the shapelet. This illustration shows, for one shapelet, the distance of every time series in the data set to that shapelet. The axis represents the distance between the shapelet and the time series of the given class.*

2.3 Interpretability

The following section provides an overview of interpreting machine learning models while focusing on time series classifiers.

To interpret denotes explaining information in simple and understandable terms that a human could easily comprehend, however, a problem arises, which is that interpretability is somewhat subjective depending on the knowledge of the human. It is typical to use images or textual information to explain the information. In machine learning, a model is said to be interpretable, if a human can understand the model's particular predictions [9].

Interpretability is important since it is essential that the end-user *trusts* the model's predictions, e.g. self-driving cars, where safety is essential. Furthermore, given a model that can predict cancer, it could be beneficial to interpret the classification to understand the underlying causes [9]. An example, of interpreting a model could be in the case of object detection in images, where the model can "highlight" the parts that made it predict a particular class, where this could easily be validated by a human. In addition, this explains how the model learns, and would make the user trust the model [9].

Prototype methods, which are closely linked to *KNN's*, are inherently interpretable at a modular level. Interpreting these models is intuitive since it is easy to calculate the distance between the prototypes and the given queries. A prototype method is presented in Section 2.4.1, however, it should be noted that a decoder is needed to embed it into lower dimensional space where the prototypes and queries can be visualized [9].

Deep Neural Networks (DNNs) are inherently non-interpretable due to their complexity, and in addition, their learned features are often not very useful for interpretability. One method for interpreting DNNs is *Feature Importance*, such as *LIME*. It is used to identify which learned features were the most important for the given prediction [9, 10].

The interpretability of *univariate time series classification (UTCS)* can be tackled in numerous ways. The most straightforward approach would be to encode the Univariate Time Series (UTC) in a prototypical network. Afterwards, a decoder can be applied to visualize the prototypes and queries. However, the shapelets presented in Section 2.2.3 could be utilized to interpret a model's prediction. This is exactly what is proposed by Tang, Liu, and Long [6]

which will be explained further in Section 2.5.1. In their proposal, they describe two types of shapelets, *Representative Shapelets* and *Discriminative Shapelets*. The representative shapelet is identified by the prototype, while the discriminative shapelet is learned by combining its prototype and the learned shapelet features. The latter shapelet must exist in every sample of that class, while never being present in any other class. These two shapelets can be used for interpretability by plotting the representative shapelet and the discriminative shapelet together. This allows the user to see which part of the representative shapelet can be used to differentiate between classes [6]

2.4 Few-Shot Learning

Conventional models, such as DNNs, have been shown to be highly accurate, however, they require large datasets. If the data is limited, the model does not learn enough signal and will not be able to generalize on new unseen data. *Few-Shot Learning (FSL)* tries to solve these aforementioned problems by learning to generalize on only a few data points [11].

A widely used definition of FSL is that a model learns from experience E from some tasks T with respect to some performance measurement P if its accuracy can be improved with more E on T, and E is very small [11].

In FSL, one typically defines the *N-way-K-shot* problem, where N denotes the number of classes, and K denotes the number of examples per class. Each FSL problem has a *support set* S representing the small set of training data, and the *query set* Q defines the task the model should predict. This is illustrated in Figure 4, where the numbers should be imagined as handwritten digits. Each class has one example, and there are 5 classes in the support set, consequently, it would be defined as *5-way-1-shot*. The whole task has $N \times K$ samples. In addition, it should be noted, that the query example should never appear in the support set, and therefore, the specific handwritten query is not the same example as the one in the support set, meaning $S \cap Q = \emptyset$ [11].

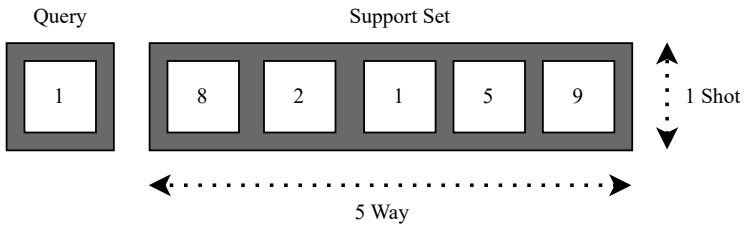


Figure 4: *Few shot learning example*

Two potential approaches for solving the FSL problems are *Learning Metric Algorithm* and *Learning Model Parameters*. Typical methods for the latter are *MAML* and *MAML++*, which will not be described further [11]. The most typical learning metric algorithms are *Siamese Networks*, *Matching Networks* and *Prototypical Networks* [11], where Protonets will be expanded

on in Section 2.4.1, which is particularly interesting due to to its inherent interpretable nature as shown in Section 2.3.

The Siamese Neural Network is a simple network, where the input is negative and positive sample pairs, where the objective is to learn a similarity function, that learns to evaluate the similarity of positive samples highly, and negative sample pairs poorly. To be more precise, the negative sample is a pair of, e.g. two images, where the two images have different classes, and the positive pair contains the same class. The Matching Network is much more complex, however, it can be understood by mapping the data into an embedding space by modelling the sample distance distribution [11].

2.4.1 Protonets

Prototypical networks, or *protonets*, is a framework for few-shot classification proposed by Snell, Swersky, and Zemel [5]. The idea behind protonets is to embed the support set $S = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2) \dots (\boldsymbol{x}_N, y_n)\}$ into a low-dimensional feature space. Assuming the embedding method embeds instances of the same class close to each other, a *prototype* of the class can be created as the mean of the instance embeddings:

$$\boldsymbol{c_k} = \frac{1}{n} \sum_{\boldsymbol{x}_i \in S_k} f_\phi(x_i) \tag{8}$$

where S_k is the set of support set examples with label k, and n is the length of this set. $f_\phi(x_i)$ is the embedding function with learnable parameters ϕ. Unlabeled data points are then classified by embedding them and assigning to them the class label of the prototype to which they are nearest as measured by a distance function d.

Technically, given an input $\boldsymbol{x}$, the protonet will produce a probability distribution over the classes, signifying how probable it is that the input belongs to each class:

$$p(y = k|\boldsymbol{x}) = \frac{exp(-d(f_\phi(\boldsymbol{x}), \boldsymbol{c}_k))}{\sum_{k'} exp(-d(f_\phi(\boldsymbol{x}), \boldsymbol{c}'_k))} \tag{9}$$

Here, d signifies the distance function (e.g. the Euclidian distance), while exp is the exponential function $exp(x) = \mathbf{e}^x$. k' is the set of all classes. In short, the probability that $\boldsymbol{x}$ is of class k is calculated by applying the exponential function to the distance between the embedding $f_\phi(x)$ and the prototype of class k, and normalizing the result.

Learning of the parameters ϕ is done by minimizing the negative log-probability for the true class: $\mathcal{L}(\phi, \boldsymbol{x}) = -\log(p(y = k|\boldsymbol{x})$. Training of the protonet is done over the entire support set, using episodic training. Episodic training means that the model is given batches of training data, and updates its parameters based on the results of each batch. Each iteration of this process is called an episode. In protonets, every episode consists of choosing N random classes, splitting the examples from each chosen class into a support set and a query set, computing the prototypes based on the support examples, and updating loss based on the query set and the prototypes [5] using the process described in equation 9.

Protonets are one of the most successful models for few-shot learning. Their success stems from the fact that they don't have to explicitly learn the parameters necessary for classifying data; as long as it embeds class instances relatively close to each other, they will be able to correctly classify a majority of queries. This allows the embedding model to classify correctly with relatively few parameters, which helps prevent overfitting [5].

2.5 State-of-the-art models

In the following section, some of the most popular state-of-the-art few-shot learning models for time series classification are discussed. These models are discussed since they will be compared against the FISH method. These subsections are just a general overview of the models and do not go into small details about them.

2.5.1 Dual Prototypical Shapelet Networks (DPSN)

DPSN tackles few-shot learning time series classification while adding interpretability to the model. This method consists of three main components: 1) A feature extraction method that converts time series data into SFA and shapelet features. To be more specific, the Symbolic Fourier Approximation (SFA) word histogram is a handcrafted feature extraction technique used in time series classification that has shown promising results; 2) The classification algorithm creates a prototype for each class by merging SFA features using a prototypical network, as discussed in subsection 2.4.1; 3) Dual interpretability is incorporated into this model, by including representative shapelets identified by a prototype during classification, and the discriminative shapelets that are learned by integrating shapelet features and prototypes. To provide more details on SFA, the data for the feature extraction is typically sparse and high-dimensional, so dimension reduction is applied to the histogram by eliminating SFA words with all zero values for each training data. This process transforms the time series into dense SFA features suitable for classification. Two hyperparameters are required for the SFA word histogram: the sampling window size (LS) and a value (w) for noise reduction in Discrete Fourier Transform (DFT). [6, 12]

2.5.2 BOSS

The BOSS (Bag-of-SFA-Symbols) model integrates the extraction of subsequences while maintaining robustness against noisy data. The BOSS model is related explicitly to feature extraction and classification in time series analysis. The BOSS model represents each time series by using SFA words to form a collection of substructures. This approach provides several advantages. Firstly, it is efficient and fast since hashing is used to determine the similarity of substructures. Secondly, it incorporates noise reduction. Lastly, it ensures invariance to the horizontal alignment of each substructure within the time series, a property known as phase shift invariance[13]. As seen in figure 5, in the BOSS model, sliding windows of length w are extracted from a time series and normalized. Furthermore, mean normalization can be

enabled or disabled to obtain offset invariance. The model transforms the time series into an unordered set of SFA words, providing phase shift-invariance. Numerosity reduction, which means that all duplicate words of SFA words are ignored until a new SFA word is identified, is applied to prevent overemphasizing stable sections of a signal. A histogram is built from the SFA words, counting their occurrences while ignoring their order. This ensures phase invariance of substructures and thus eliminates the necessity for preprocessing the samples by an expert to align the substructures [13].

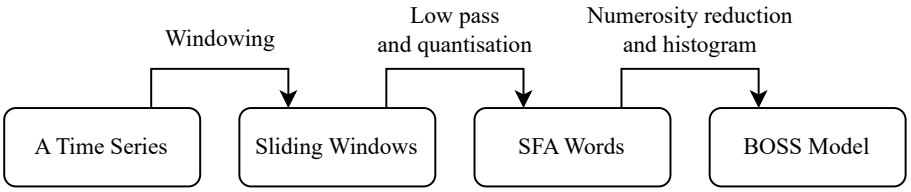


Figure 5: *Overview of the BOSS Framework[13].*

3 Method

The overall flow consists of two main parts: the FISH feature extraction, and the prototypical network. Feature extraction consists of identifying relevant features that can be used to differentiate between classes, where a feature is some aspect of a time series, which is unique for each class. A further explanation of what a feature consists of in FISH will be presented in the following section. In the second part, each time series is analysed in terms of these features, and the result is used in the prototypical network.

The following Figure 6 describing the FISH method, will provide an overview of the pipeline, whereafter each step will be explained. The specific steps will be explained more thoroughly in the later subsections.

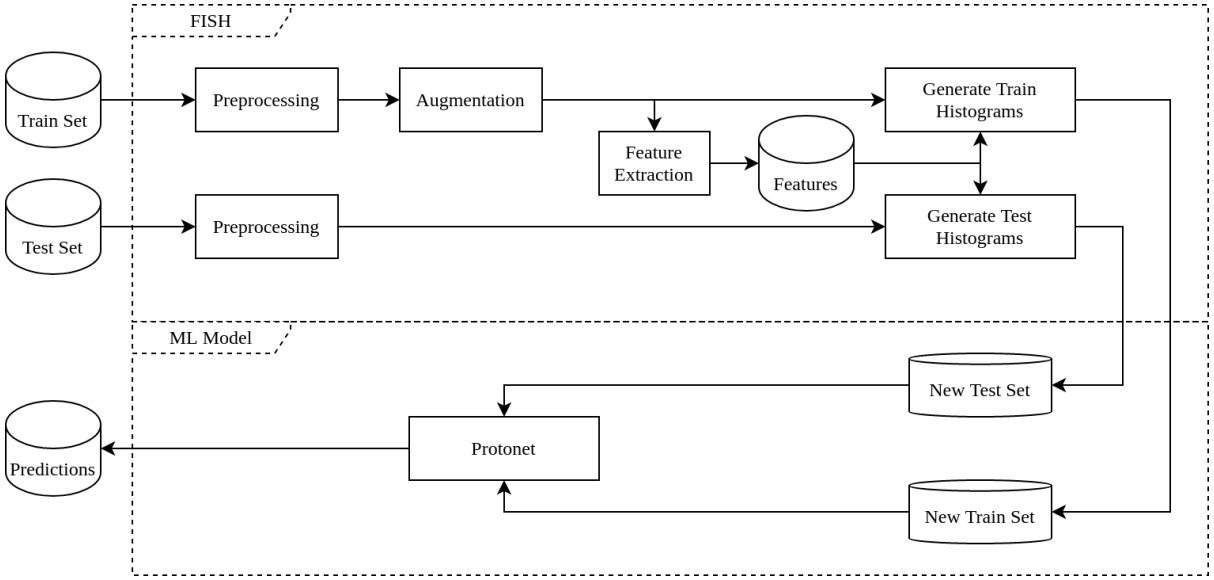


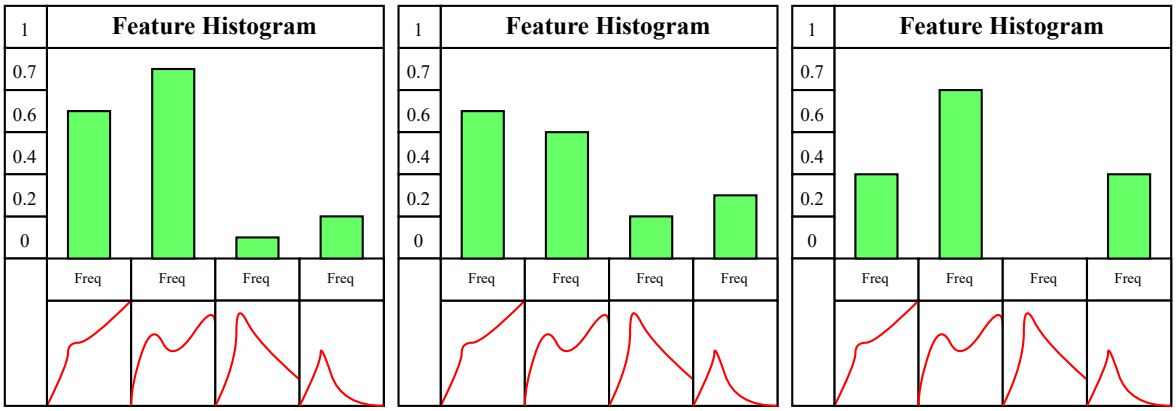
Figure 6: *Flow of the FISH method*

In Figure 6, the training data and the test data, consisting of a few labelled univariate time series are *preprocessed*, which consists of normalizing the data. The preprocessed training data is used for the generation of features through *Feature Extraction*. This results in a set of *features* which, in combination with the original preprocessed data, are used to generate a new training and test set. These new training and test sets consist of *feature histograms*. A feature histogram is a vector consisting of values generated for each time series, in combination with the set of features. A deeper explanation can be found in the following section. The newly generated train histogram is used to train the Protonet, where the goal is to learn the generalisation of the data. Afterwards, the Protonet is used to classify the test set.

3.1 Features and Feature Histograms

In the FISH method, features consist of two parts: a *shapelet* and an *attribute*. As such, a feature is a 2-tuple $F = (shapelet, attribute)$, where the attribute is used to generate a value using the shapelet and a time series. For example, an attribute could be the minimum distance (minDist) between the shapelet and the time series, or the frequency with which the shapelet occurs in the time series. To give a concrete example: if the feature F_1 consists of the shapelet S_1 and the attribute *frequency*, then a time series TS_1 can be analysed using F_1 to determine how many times S_1 occurs in TS_1. This can be useful if S_1 occurs more frequently in one class than another.

As a feature can generate a value for a given time series, a set of features can generate a set of values. In the FISH method such a set, generated for a single time series, is called a *Feature Histogram*. Three of these histograms can be seen in figure 7 below.



(a) *Feature Histogram one* **(b)** *Feature Histogram two* **(c)** *Feature Histogram three*

Figure 7: *Graphs showing the resulting* `Freq` *results from 3 given datapoints in some dataset.*

Each sub-figure represents a single feature histogram, where each column corresponds to the value generated from a single feature. Below each column, the feature used to generate the value can be seen, with the attribute in all these cases being *Freq*, and the shapelet shown.

As can be seen, each histogram is generated from the same four features, with differing generated values. If the features used are good, histograms generated from data points of different classes will be different, while ones generated from the same class will be similar.

This is more clear if the graphs in figure 7 are combined into a shapelet histogram, where each of the values in the extracted data gets put into a boxplot instead. An example of the resulting histogram can be seen in figure 8.

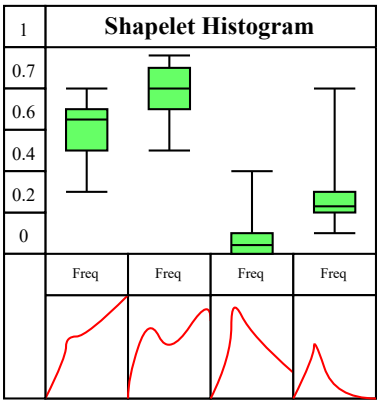


Figure 8: *A full shapelet histogram with all the data points into one graph.*

Here each column, or rather box in the boxplot, shows the combined value for their corresponding columns in the feature histograms used to generate it. A low variance means that the values are similar for the data points, while a high one means it is dissimilar.

If such a histogram is generated for a class, with good features, the variance would be low as the values for each data point would be similar, while histograms generated for differing classes would have a little overlap for each feature.

While this is a useful tool for classification through machine learning, it is also a visual aid for understanding the data. For instance, if we have three different labelled shapelet histograms, where each defines the shapelet properties of its class, a human could identify an unlabelled feature histogram by finding the shapelet histogram to which it is most similar.

3.2 Pre-Processing

The pre-processing consists of normalizing the time series data and zero-indexing the class labels. To be specific, this means that each data point in the time series has a value between $[0, 1]$, and the original labels consisting of any class index value are converted to $\{0, 1, ..., n - 1\}$.[2] The purpose of this preprocessing is to both to make the implementation easier, but also to make the shapelet histograms more interpretable.

3.3 Data Augmentation

A facet of the FISH method that has to be considered is the possibility of features overfitting the training data. This is akin to a classifier overfitting to its training data, by learning noise, particularities of the training data, or other irrelevant information. In the same manner, some features could potentially differentiate the training data, while not differentiating the test data. As such, the FISH method augments the original data with slightly modified versions of the original.

[2]This is done because the datasets used later, being from the UCR archive, have peculiar index-values

The training data consists of a set of labelled series $S = \{(label, series)\}$, where the label is some natural number and the series is a one-dimensional vector of real numbers. A modification is then a modification of the series while maintaining the original label.

One such modification is smoothing, which attempts to reduce the noise while leaving general shapes in the series. This is done through a reduction in the change between points in the series, where points with a value below their neighbours increase in value, while those above decrease.

Each point is, as such, given the value of the average between it and its neighbours. A neighbour for a given point $p_x \in series$ is given by p_{x-1} and p_{x+1}, as in the point before and after a point is its neighbours. However, this is extended to any natural number, such that p_x and its closest i neighbours means $\{p_{x-i}, ..., p_{x-2}, p_{x-1}, p_x, p_{x+1}, p_{x+2}, ..., p_{x+i}\}$. Where in the case of it not having the given number of neighbours, they are simply ignored. An example of this would be the first point in a series p_0, which does not have any lower neighbours, and as such p_0 and its closest neighbours simply means $\{p_0, p_1\}$.

This can also be expressed as a simple algorithm, as can be seen in algorithm 1

Algorithm 1 SeriesSmoothing(series, t)

Require: $series = [p_1, p_2, ..., p_n]$ where $p_{1...n} \in \mathbb{R}$
Require: $t \in \mathbb{N}$
1: $smooth\_series \leftarrow []$
2: **for** $p_i \in s_{series}$ **do**
3: Append average of p_i, and t closest neighbours to *smooth_series* ▷ As described in the text above algorithm
4: **end for**
return *smooth_series*

Another augmentation is also possible, being the *noisification* of the time series data. Where smoothing tries to remove noise, this method tries to add it. The idea follows the lines of the old saying: "If everyone is a hero, no one is.", i.e. if all classes have a series with noise, the noise is not something that differentiates them.

To combat this, a very naive method is used, which simply moves each point in a series a set amount either up or down randomly. Which can be seen in algorithm 2.

Algorithm 2 SeriesNoisification(series, t)

Require: $series = [p_1, p_2, ..., p_n]$ where $p_{1...n} \in \mathbb{R}$
Require: $t \in \mathbb{R}$
1: $noisy\_series \leftarrow []$
2: **for** $p_i \in s_{series}$ **do**
3: Append $p_i + p_i * t$ to $noisy\_series$
4: **end for**
return $noisy\_series$

Using the two methods together would result in a training set tripling in size.

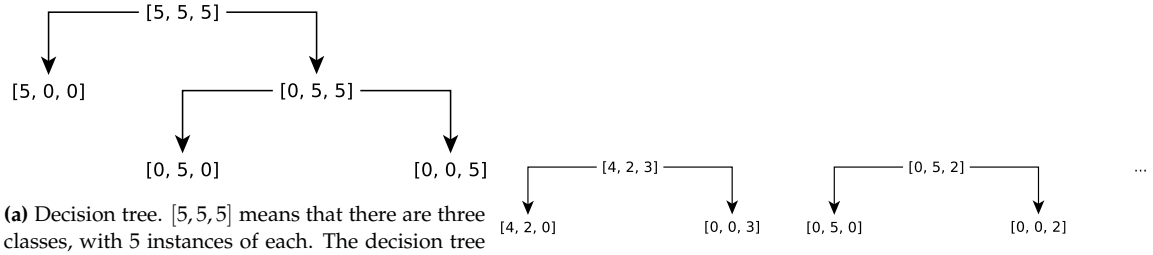
3.4 Feature Extraction

The whole idea of the FISH method builds upon the method described in the paper *Time Series Shapelets: A New Primitive for Data Mining* [4]. It introduces the idea of finding shapelets via the information gained by splitting upon the minimum distance to the given shapelet. It then, through repeatedly splitting the training set, generates a decision tree.

While the FISH method does not use a decision tree, it still uses and expands upon the idea of using information gain from shapelets. However, it attempts to improve what features can be detected, and the quantity which can be found.

In the paper, only a single attribute is considered, which is the minimum distance to a given shapelet. However, this might not be the best in all cases. As such, the FISH method proposes a method which can consider multiple attributes, and automatically use the most appropriate to a given training set.

Moreover, FISH allows a higher feature count to be extracted. A decision tree is inherently limited in the number of splits it generates, and, as such, the number of features it can find. This can be seen in figure 9a, where the training set only allows two splits.



(a) Decision tree. $[5,5,5]$ means that there are three classes, with 5 instances of each. The decision tree separates the classes based on features.

(b) FISH sampling.

Figure 9: *Decision tree sampling vs FISH sampling for a 5-shot-3-way training set.*

The FISH counteracts this by sampling the training data, which takes some subset of the whole training set. An example of this can be seen in figure 9b, where subsets $[4,2,3]$ and $[0,5,2]$ is shown. The numbers correspond to the number of data points in each class. Each

subset can then be used to generate a feature, that best fits that specific situation. Which in turn allows for a potentially higher feature count.

3.4.1 Sampling

The result of each sample is a feature, which is the combination of an attribute and a shapelet. Where an attribute is any given function, that takes a time series and a shapelet, and results in a real number between 0 and 1.[3] An example is *Minimum Distance*, which is the minimum Euclidian distance between the shapelet and time series, divided by the maximum possible distance.

In order to generate a set of such features, multiple approaches are possible. One method is what the paper *Time Series Shapelets: A New Primitive for Data Mining*[4] does. Where a classification tree is generated, with each node being the highest-scoring feature. However, this approach is rather limited when used with a separate classifier. As it generates a limited number of features and doesn't explore all combinations of data points.

The approach this section explores tries to counteract these two limitations. It does so by repeatedly sampling the training data until either some set limit or no new features can be created. This is the goal of the algorithm 3.

Algorithm 3 GenerateFeatures(S, featureLimit)

Require: $S = \{(label, series)\}$
Require: $label \in \mathbb{N}$
Require: $series = [x_1, x_2, ..., x_n]$ where $x_{1...n} \in \mathbb{R}$
Require: $featureLimit \in \mathbb{N}$
1: $features \leftarrow \emptyset$
2: $priorSamples \leftarrow \emptyset$
3: **while** $|features| < featureLimit$ **do**
4: $sample \leftarrow$ some random subset of S
5: **if** $sample \notin priorSamples$ **then**
6: $priorSamples \leftarrow priorSamples \cup sample$
7: $windows \leftarrow GenerateWindows(sample)$
8: $feature \leftarrow FindOptimalFeature(sample, windows)$
9: $features \leftarrow features \cup feature$
10: **end if**
11: **end while**
return $features$

In lines 1 and 2, both *features* and *priorSamples* are initialised to the empty set. Whereafter, given a set of labelled series and a *featureLimit* line 3-6 repeatedly samples the training data,

[3] This method could trivially be altered to allow values outside the 0 to 1 range. However, it is used to create more visually consistent histograms.

until enough features have been created. Notably, line 5 and 9 check whether the sample and feature respectively, has been found before. As such, no duplicate samples and features are used, which, can lead to an infinite loop in the case of no new sample or feature being possible. Line 6 simply adds the novel sample to the set of checked samples in *priorSamples*.

Line 7 is the creation of candidate shapelets. A shapelet is in essence just some subseries from any *series* in S. As an example, *GenerateWindows*($\{[0,1,2],[3,4]\}$) would generate $\{[0,1],[1,2],[0,1,2],[3,4]\}$. However, due to windows disregarding the position they were found in, the first value of any window is zero, and the following are moved correspondingly. As such, the prior example would result in: $\{[0,1],[0,1],[0,1,2],[0,1]\} \Rightarrow \{[0,1],[0,1,2]\}$.

This comes with the advantage that a window can be seen as a set of changes to the value, as opposed to a set of values. This means that an attribute generates the same value for a given shapelet, regardless of where it was found.

The combination of sample and windows is then in line 8 used to find the optimal feature, which will be explained in the following section. While line 9 simply add any novel features to the set of *features*.

3.4.2 Finding the Optimal Feature

This section pertains to evaluating the possible features, through combining the given set of windows and some set of attributes. While the actual evaluation of a feature lies in the next section.

Algorithm 4 FindOptimalFeature(S, W)

Require: $S = \{(label, series)\}$
Require: $label \in \mathbb{N}$
Require: $series = [x_1, x_2, ..., x_n]$ where $x_{1...n} \in \mathbb{R}$
Require: W or windows is a set of series, all shorter than *series* length n
Require: A is a set of attributes, e.g. frequency, min distance...
1: $bestFeature \leftarrow$ **NULL**
2: $bestScore \leftarrow 0$
3: **for** $w \in W$ **do**
4: **for** $a \in A$ **do**
5: **if** $EvaluateWindow(S, a, w) > bestScore$ **then**
6: $bestFeature \leftarrow (a, w)$
7: $bestScore \leftarrow EvaluateWindow(S, a, w)$
8: **end if**
9: **end for**
10: **end for**
 return *bestFeature*

As such, the above algorithm 4 iterates through the two sets W, the set of windows; and A, the set of attributes; on lines 3 and 4. Line 5 to 8 evaluates the feature with an

algorithm described in the next section. Upon an evaluation larger than the current best, it sets *bestFeature* and *bestScore* to the current feature and evaluation respectively.

3.4.3 Feature Evaluation

In order to decide on a feature's importance, a comparable metric must be used. As described in section 3.1, a good feature is one that generates similar values for the same class, while generating different values for different classes. As such, any metric that corresponds to that description could be used.

The FISH method evaluates the features based on information gain from a binary split. This, in essence, translates to: How big a reduction in entropy is gained through splitting the values generated from a feature in two.

As an example, take some feature F and a set of series S, the set of values V generated from $\{F(s)|s \in S\}$ where $F(s)$ means the value generated from feature F on series s. As such, the set $V = \{v_1, v_2, ..., v_n\}$ can be split into two sets: one with all the values lower than some constant k, and one with those higher. Which results in the sets $V_l = \{v|v \in V \wedge v < k\}$ and $V_k = \{v|v \in V \wedge v > k\}$. Assuming an addition of a label to the values, it would be possible to calculate the entropy of both set's labels and compare them with the original set.

With that said, this section contains the generation and evaluation of values, while the next section pertains to the calculation of entropy reduction. Firstly, the evaluation of windows can be seen in algorithm 5.

Algorithm 5 EvaluateWindow(S, a, w)

Require: $S = \{(label, series)\}$
Require: $label \in \mathbb{N}$
Require: $series = [x_1, x_2, ..., x_n]$ where $x_{1...n} \in \mathbb{R}$
Require: a is some attribute, e.g. frequency, min distance...
Require: $w = [x_1, x_2, ..., x_n]$ where $x_{1...n} \in \mathbb{R}$, where $|w| < |s_{series}|$ for all $s \in S$
1: $valueCount \leftarrow \{(value, classCount)\}$ ▷ Where classCount is how many occurrences of each class the given value has
2: **for** $s \in S$ **do**
3: $value \leftarrow GenerateValue(a, s_{series}, w)$
4: $valueCount_{value} \leftarrow$ Adds single occurrence of label s_{label}
5: **end for**
return $CalculateInformationGain(valueCount, CalculateEntropy(S))$

In order to calculate the values, with the addition of labels, a histogram is used. This can be seen in line 1, where *value* is the value generated from the feature, and *classCount* is the number of labels of each class at that value. An example of its value after filling it up for *2-shot-2-way* would be $\{(0.1, [2, 0]), (0.8, [0, 2])\}$. Which consists of two buckets each with two occurrences, of class 0 and 1 respectively. With the prior bucket having a value of 0.1 and the

latter 0.8.

That example would occur in the case of all time series of class 0 generating the value 0.1 in combination with the given feature, while all of the class 1 generating the value of 0.8. A situation such as this, where all elements of a given class are in a single bucket, is, however, rather rare.

As such, a better way to understand this is as a value line, where each time series falls somewhere between 0 and 1. The same example as above can be seen visualized below in figure 10 as a value line.

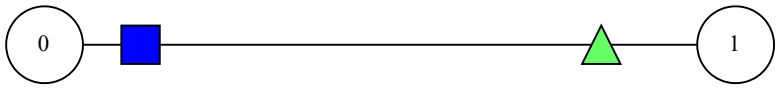


Figure 10: *A value line with two occurrences of class 0 at value 0.1, and two of class 1 at 0.8.*

However, as mentioned this is a very unrealistic scenario. A more likely occurrence can be seen in figure 11 below.

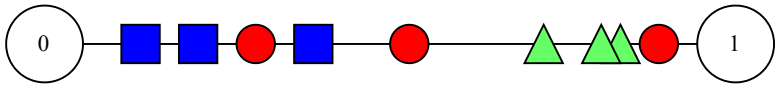


Figure 11: *A value line for 3-shot-3way.*

Figure 11 shows what a potential occurrence distribution might look like in a *3-shot-3-way* scenario. This brings up the next part of algorithm 5, namely lines 2-4. These lines loop through all the time series in the sample and generate a value for each from the feature. With the feature being the combination of a given window w and a given attribute a. The value generated is highly dependent on which attribute is used.

While it was stated that essentially any function can be used as an attribute, this report focused on three different ones. These three are *Minimum Distance*, *Frequency* and *Occurence Position*.

The *Minimum Distance* is the same as from the shapelet paper mentioned earlier. The frequency attribute measures how often a given shapelet occurs in a given series. Where a value of 0 would mean the shapelet never occurs, and a value of 1 would mean it always occurs.

The definition of this occurrence is some sub-series of the time series which is of the same length as the shapelet. As an example, let us say that we want to calculate the frequency of the shapelet $[0, 1]$ in the time series $[0, 1, 2, ..., n]$ the set of candidates is then $\{[0, 1], [1, 2], ..., [n - 1, n]\}$. With the caveat that a shapelet is a set of changes to value as opposed to a set of values. As such, any candidate which changes according to the shapelet, i.e. $[k, k + 1]$, would be an occurrence. This means, that for this example, the frequency would be 1.

However, as the FISH method works for real numbers, some leeway is necessary; a change of 0 to 1 might be functionally equivalent to 0 to 1.01. Therefore, a tolerance is introduced,

which is a constant value that any change might stray from the shapelet.

As such, the attribute *occPos* is the position of the first such occurrence, with tolerance, relative to the maximum possible position.

Algorithm 6 Frequency(series, window)

Require: $series = [x_1, x_2, ..., x_n]$ where $x_{1...n} \in \mathbb{R}$
Require: $window = [y_1, y_2, ..., y_n]$ where $y_{1...n} \in \mathbb{R}$, where $|window| < |s_{series}|$ for all $s \in S$
Require: $tolerance \leftarrow z$ where $z \in \mathbb{R}$ and $0 \leq z \leq 1$
1: $windows \leftarrow GenerateWindows(series, |window|)$
2: $occuranceCount \leftarrow 0$
3: **for** $w \in windows$ **do**
4: **if** $IsMatch(window, w, tolerance)$ **then**
5: $occuranceCount \leftarrow occuranceCount + 1$
6: **end if**
7: **end for**
return $occuranceCount / |windows|$

This can all be seen in algorithm 6, where the set of candidates, i.e. windows, is calculated in line 1. This is done through the same algorithm as the generation of shapelets originally, however, with the caveat that no duplicates are removed. Each window is then checked in lines 3-7, for whether it is an occurrence. Where line 2 and 5 counts the number of such occurrences.

Notably, line 4 uses an algorithm called `IsMatch`, which checks whether the window is equivalent to the shapelet with a given *tolerance*.

Algorithm 7 IsMatch(windowA, windowB, tolerance)

Require: $windowA, windowB = [y_1, y_2, ..., y_n]$ where $y_{1...n} \in \mathbb{R}$, where $|windowA| < |s_{series}|$ for all $s \in S$, and $|windowA| = |windowB|$
Require: $tolerance = z$ where $z \in \mathbb{R}$ and $0 \leq z \leq 1$
1: **for** $i = 0 \leq |windowA|$ **do**
2: $dist = |windowA_i - windowB_i|$
3: **if** $dist > tolerance$ **then**
return $FALSE$
4: **end if**
5: **end for**
return $TRUE$

In the algorithm 7, line 1 is simply an index iterator between 0 and the length of the windows. At each such index, the distance between the two values is calculated in line 2. Absolute distance is used since it always results in positive numbers. If any such distance is higher than the *tolerance*, the algorithm returns *FALSE*, i.e. not a match, in lines 3-4.

However, should no distance be higher than the tolerance, true is returned.

3.4.4 Information Gain

This last section concerns itself with calculating the reduction in entropy gained from splitting the generated value histogram generated in the prior section. This reduction in entropy is also called information gain, which is described in depth in Section 2.1.

In the prior section, it was briefly mentioned how the information gain would be calculated through a binary split, with a set of values below some constant and one for those above. This constant, or rather, split point can in theory be any value that lies above the lowest value of the histogram and below the highest. However, the large majority of the possible split points will lead to the same two sets.

As an example, say that we want to find a split-point for the histogram $H = \{(0, [...]), (0.5, [...]), (1, [...])\}$. For this histogram, there exist two groups of split points, namely $\{splitPoint | v \in R \wedge v > 0 \wedge v < 0.5\}$ and $\{splitPoint | v \in R \wedge v > 0.5 \wedge v < 1\}$. Any value within one of these groups generates the same binary split as any other value from that group, e.g. 0 is both smaller than 0.1 and 0.2, while 0.5 and 1 are both larger.

As such, the FISH method uses the halfway point between values, e.g. for H it would be $\{0.25, 0.75\}$. However, it is not known beforehand which of these points produces optimal information gain, so each is checked. This is done in algorithm 8

Algorithm 8 CalculateInformationGain(valueCount, priorEntropy)

Require: $valueCount = \{(value, classCount)\}$
Require: $value \in \mathbb{R}$, and $0 \leq value \leq 1$
Require: $priorEntropy \in \mathbb{R}$
1: $bestGain \leftarrow 0$
2: $splitPoints \leftarrow GenerateSplitPoints(valueCount)$
3: **for** $sp \in splitPoints$ **do**
4: $\quad gain \leftarrow priorEntropy - CalculateSplitEntropy(valueCount, sp)$
5: $\quad$ **if** $gain > bestGain$ **then**
6: $\quad\quad bestGain \leftarrow gain$
7: $\quad$ **end if**
8: **end for**
return $bestGain$

Line 2 generates all the halfway points between values, an example of which can be found above the algorithm. A halfway point could be calculated as the difference between a pair of values divided by 2. Line 3 then iterates over all *splitPoints*, where line 4 then calculates the information gain as the reduction in entropy. How entropy is calculated can be found in section 2.1. Lines 1 and 5-7 simply maintain the highest found information gain, which is then returned.

3.5 Protonet

The FISH method does not alter the standard Prototypical Network. However, the network should still be described. The network consists of four identical convolutional blocks, structured as:

- 1 Dimensional Convolution layer with a kernel size of 2, stride 1 and padding 1.
- 1 Dimensional batch normalization
- A ReLu layer
- 1 Dimensional Max pooling layer wither kernel size of 2

The channels of each block are as follows:

- Block 1: (in: 01) (out: 64)
- Block 2: (in: 64) (out: 64)
- Block 3: (in: 64) (out: 64)
- Block 4: (in: 64) (out: 64)

No further changes to the Protonet have been made.

4 Experimental Evaluation

To evaluate this novel FISH method some experiments are needed. This section will go into detail on how such experiments are run, with what data, and discuss the results. This section will also compare against the *DPSN*, *BOSS* and *ST* methods. Finally, the interpretability of the FISH method will also be evaluated.

4.1 Datasets

Table 2 presents a list of datasets used in the project, which are derived from the UCR Time Classification Archive [14]. These datasets are widely utilized in time series analysis and were specifically chosen due to their usage in state-of-the-art classification models. The state-of-the-art models that are compared with our proposed method are as follows: DPSN, BOSS, and Shapelet Transform(ST).

Dataset	N. of Classes	Train Samples	Test Samples	Time Series Length
ArrowHead	3	36	175	251
BME	3	30	150	128
CBF	3	30	900	128
Chinatown	2	20	343	24
ECG200	2	100	100	96
GunPoint	2	50	150	150
GunPointAgeSpan	2	135	316	150
GunPointOldVersusYoung	2	136	315	150
ItalyPowerDemand	2	67	1029	24
MoteStrain	2	20	1252	84
Plane	7	105	105	144
SonyAIBORobotSurface1	2	20	601	70
SonyAIBORobotSurface2	2	27	953	63
SyntheticControl	6	300	300	60
ToeSegmentation1	2	40	228	277
TwoLeadECG	2	23	1139	82
UMD	3	36	144	150
Wine	2	57	54	234

Table 2: *List of datasets that will be experimented on.*

4.2 Experiment Setup

There are several different hyperparameters that can be changed in both the feature extraction and the Protonet. However, a larger focus will be put on the feature extraction experiments, since that is the primary focus of this report. The experiments will be run on 6-shot and

8-shot since it allows for direct comparison between our model and the DPSN Tang, Liu, and Long [6] and BOSS Schäfer [13] model.

For all the results, each experiment is run against all the 18 datasets mentioned earlier, a total of 3 times to get a good average. For each of those times, the feature extractor is forced to remake its features, thereby making it select new samples and potentially new features. For each of these times, n samples are taken, either 6 or 8 shots, from the *train* set of the UCR datasets. The rest of the data in the *train* set is discarded. Finally, it will run tests on the full *test* set in the given UCR dataset.

To find a good general setup of these hyperparameters, an iterative approach is used to find good values. The hyperparameters that are experimented on are the following:

1. Feature Count
    - Denoted the number of features the feature extractor should create.
2. Different Attributes
    - A set of different attributes that can be given to the feature extractor to make features from
3. Smoothing Degree
    - Describes how much smoothing is added to all the shots, following the algorithm described earlier in the method section.
4. Noise Degree
    - Describes how much noise is added to all the shots, following the algorithm described earlier in the method section.

The experiments will be run in that order, where the best configuration for a hyperparameter will be carried on to the next experiment.

Each hyperparameter has been given a range or a preset that they can be set to, these can be seen in the following table:

Hyperparameter	**Values**
Feature Count	4, 8, 16, 32, 64, 128, 256
Attributes	MinDist, MaxDist, OccPos [0.1, 0.01, 0.001, 0.0001, 0.00001], OccPos All, OccPoss All + MinDist, Freq [0.1, 0.01, 0.001, 0.0001, 0.00001] Freq All, Freq All + MinDist
Smoothing Degree	1, 2, 4, 8, 16, 32, 64, 128
Noise Degree	1%, 2%, 4%, 8%, 16%, 32%, 64%

Table 3: *Note: The `Freq [...]` means that the Freq attribute is run with each of the values in the list, i.e. `Freq 0.1` is an experiment, `Freq 0.01` is an experiment, etc. Freq All means it's a single experiment where all the Freq ones are combined, i.e. `Freq 0.1, Freq 0.01, ...` in a single experiment*

After a good general configuration has been found, an "optimal" experiment will be run, to directly compare the FISH method against others.

All the experiments are run on a computer with a *Intel i9-9900K* processor with a *RTX 2070* and *16 GB* of ram.

For all the experiments, the ProtoNet was given 5 epochs to train on, with 100 iterations for each epoch.

All the above is mostly focused on accuracy results, however, interpretability is also important. Therefore, after the accuracy experiments, some casework will be done on a few of the datasets, to analyze interpretability.

4.3 Accuracy Results and Discussion

The results for each different parameter will be shown and discussed here. Every subsection here is the isolated results and discussion of the parameters result. After each parameter has been investigated, an optimal configuration run will be made that shows the general performance of the FISH method.

4.3.1 Feature Count

The first hyperparameter that will be experimented on is the *Feature Count*. These experiments were run with *4*, *8*, *16*, *32*, *64*, *128*, *256* feature counts, and `MinDist`[4].

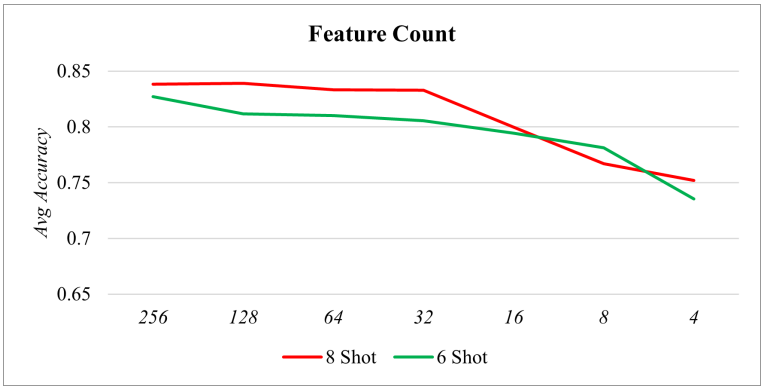
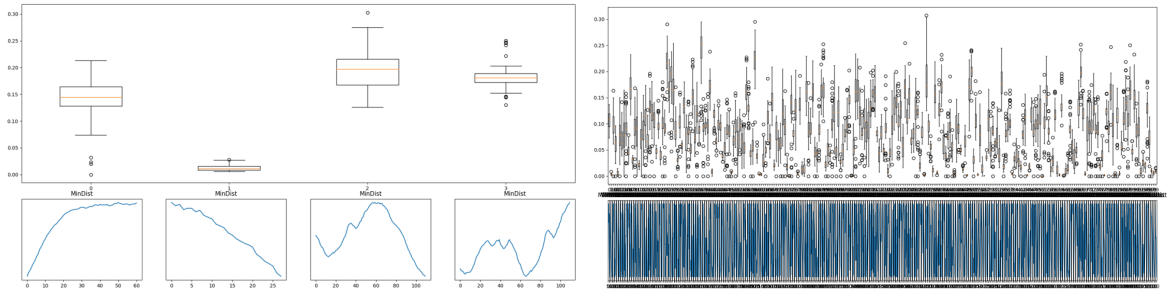


Figure 12: *Accuracy over feature count*

It can generally be seen in Figure 12 that the accuracy keeps increasing with the feature count, but not that drastically above 32 features. While the accuracy increases with the feature count, the interpretability of the histograms becomes worse, since the histograms become enormous.

As an example, a look can be taken at the *ArrowHead* dataset with 4 features and 256 features.

[4] `MinDist` was used here, since it required no parameters.



(a) *4 Features* **(b)** *256 Features*

Figure 13: *Feature Histogram from the ArrowHead datasets class 1 (6 shot)*

It can be clearly observed in Figure 13 that having a 4-shapelet histogram is more interpretable, while the 256-shapelet histogram is too large and complex, which in turn makes it difficult to interpret the results.

With this in mind, it can be seen increasing the feature count has a clear trade-off between accuracy and interpretability. Therefore a compromise has to be made, if it is desired to have higher interpretability on a dataset, lower feature counts can be used at the loss of some accuracy. However if the goal is to only get high accuracy, the feature count can be set as large as one wants.

Another downside to higher feature counts is that it takes a long time for the feature extractor to process it. Therefore, for the sake of simplicity and speed, 32 features will be used in the following experiments, since it was observed in figure 12 that this is where the accuracy flattens out. A feature count of 32 was consequently chosen, and the next hyperparameter that will be experimented on is the *Attributes*.

4.3.2 Attributes

The attribute results can be difficult to interpret since it depends heavily on the chosen dataset. Some datasets do better with some attributes and worse with others. However, an average graph can be made over the different attribute experiments, as can be seen in figure 14:

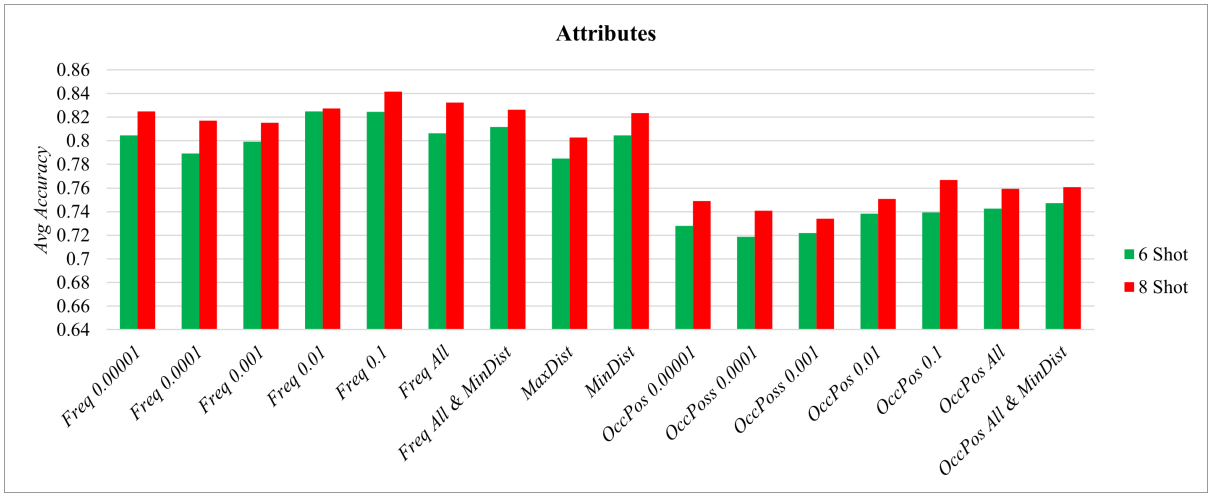


Figure 14: *Average accuracy over different attributes (32 features)*

It can be seen in figure 14 that `Freq 0.1` seems to be the best one overall by a few percentage points. It should also be noted that it is possible that certain combinations of attributes are better than just this single frequency-based one, however, for general purposes the `Freq 0.1` seems to be a reasonable pick. Based on this information the next experiment will be run with 32 features and the `Freq 0.1` attribute.

4.3.3 Smoothing

The following experiment will test the *smoothing* degree, which augments the data. This experiment will test the smoothing degrees of *1, 2, 4, 8, 16, 32, 64, 128*. As described in Section 3.3, this smoothing degree indicates the number of neighbours of a data point to make a rolling average from.

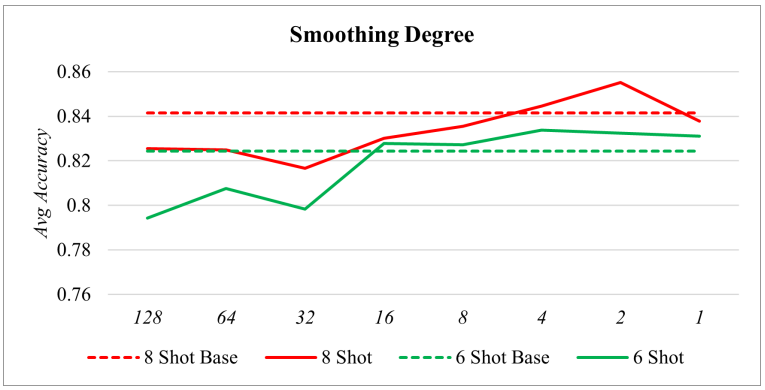


Figure 15: *Accuracy over Smooth degree for 32 features and the attribute* ***`Freq 0.1`****. The "base" lines are the accuracy for 32 features from before.*

As can be observed in Figure 15 large smoothing degrees perform poorly, while low values from 1 to 4 perform reasonably well. As a result, a smoothing degree of 2 was chosen for further experimentation.

4.3.4 Noise

The final hyperparameter to experiment on is the noise augmentation. This will be run with *1%, 2%, 4%, 8%, 16%, 32%, 64%* noise degrees. As described in section 3.3, the noise degree is describing how much each data point is allowed to deviate in the y-axis, based on their original y-value.

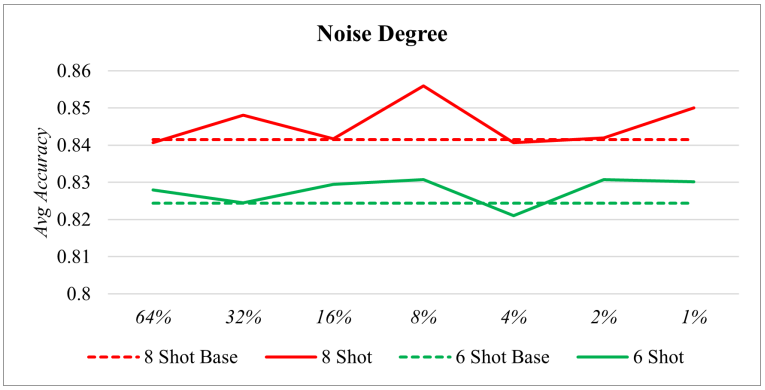


Figure 16: *Accuracy over Noise degree for 32 features, the attribute* `Freq 0.1` *and smoothing degree at 2. The "base" lines are the accuracy for 32 features from before.*

As can be seen in Figure 16 the accuracy over noise degree is somewhat random. It seems to either increase accuracy a bit, or keep it the same as the baseline 32 features accuracy. It can also be said that this hyperparameter is very situational since some datasets benefit more from noise than others. A look can be taken at the dataset *Wine* with 6-shot, where it can be seen that it hits a "sweet spot" with a little bit of noise in it.

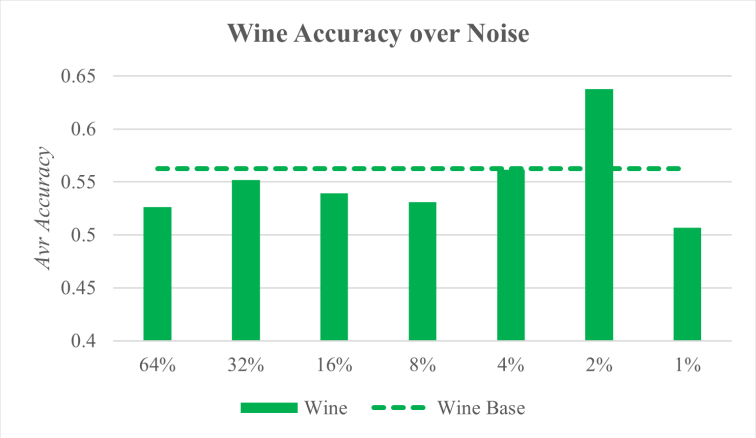
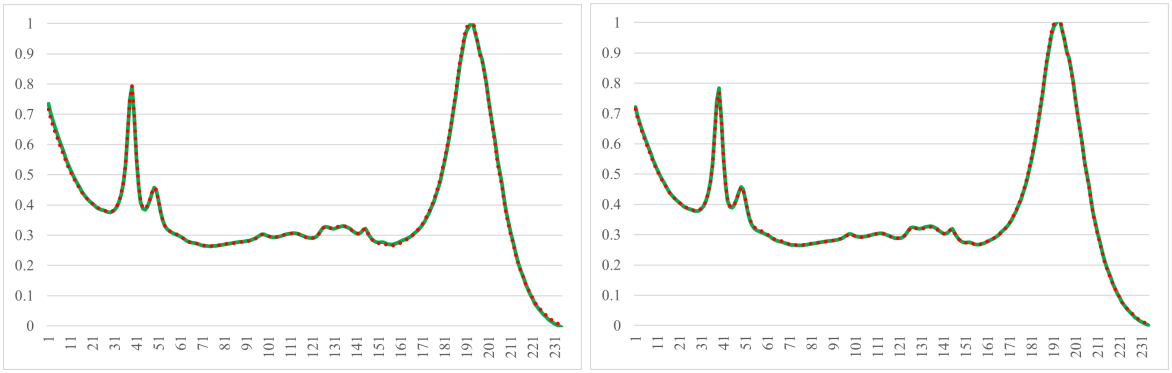


Figure 17: *Accuracy over Noise degree for 32 features in the Wine dataset on 6 shot, the attribute* `Freq 0.1` *and smoothing degree at 2. The "base" lines are the accuracy for 32 features from before.*

It can be seen in figure 17 that the *Wine* accuracy jumps up with a low amount of noise augmentation. This seems to indicate that at least some of the datasets benefit from some level of noise degree, while others do not. It should be mentioned that it makes sense that adding noise to the *Wine* dataset is reasonable since it is a peculiar dataset, where the two classes are practically identical. Figure 18 illustrates these minute differences.



(a) *Class 1* **(b)** *Class 2*

Figure 18: *Example data from the Wine datasets two classes. Both graphs contain two data points (one is marked as dotted to make it clear that they are practically on top of each other)*

So adding a bit of noise to better differentiate these two classes in the *Wine* dataset can be a good idea. However, it is not a universal thing for all the datasets that more noise makes the results better, so it should be used situationally instead. That being said, based on figure 16 it can be concluded that a noise degree of 0-2% is a good pick since it either makes the results better or is equivalent to no noise.

4.3.5 Optimal Configuration

Based on all the results, a general configuration for the feature extractor can be set as:

- Feature Count: 32 or larger
- Attributes: `Freq 0.1`
- Smooth Degree: 1-4
- Noise Degree: 0-2%

This configuration can now be used to make a direct comparison to other methods, being *DPSN*, *BOSS* and *ST*. The data for these other methods will be taken from the *DPSN* paper[6, table 1]. These results are intended to focus on accuracy only, so a high feature count of 256 will be used. The sections after this one will look more into interpretability. Using the range of optimal configurations, a single optimal configuration will be used, being:

- Feature Count: 256
- Attributes: `Freq 0.1`
- Smooth Degree: 2
- Noise Degree: 1%

This configuration is used for all the following optimal results.

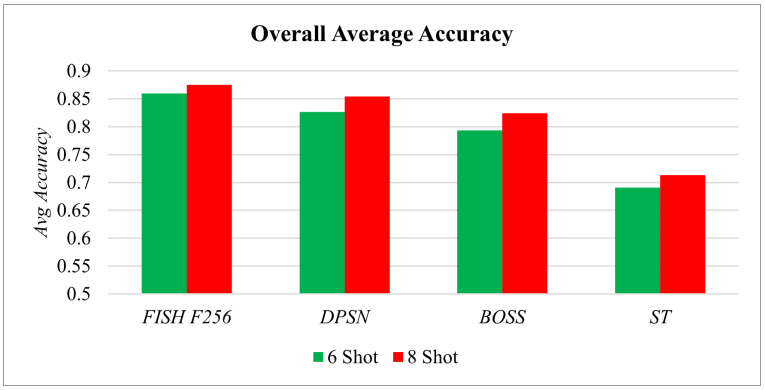


Figure 19: *Average accuracy of the FISH method against DPSN, BOSS and ST.*

As can be seen in figure 19 the FISH method with 256 features beats all the other methods on average. A deeper look can be taken into how the methods performed against each dataset. To do this, a table is created that ranks the accuracy of each dataset. The style of the following tables will take inspiration from DPSN[6, table 1]. The values marked in bold are the method that got the highest accuracy for the given dataset, while those values that are underlined have the second-best accuracy. At the bottom of each of the tables, the total amount of times a method got 1st place, 2nd place, or lost will be given. Finally, a win rate is given which denotes the number of times a given method won compared to lost. The loss rate is defined in a similar way but in reverse. The following two tables show how the FISH method performed in the 18 datasets:

Dataset Name	Classes	FISH 256F	DPSN	*(Diff)*	BOSS	*(Diff)*	ST	*(Diff)*
ArrowHead	*3*	69.2%	**78.7%**	*13.8%*	67.6%	*-2.3%*	51.3%	*-25.9%*
BME	*3*	**88.7%**	72.0%	*-18.8%*	82.7%	*-6.7%*	64.1%	*-27.8%*
CBF	*3*	97.2%	**97.6%**	*0.3%*	96.1%	*-1.2%*	89.8%	*-7.7%*
Chinatown	*2*	**95.2%**	80.4%	*-15.5%*	78.6%	*-17.5%*	71.6%	*-24.8%*
ECG200	*2*	75.6%	**83.2%**	*10.0%*	75.9%	*0.4%*	65.6%	*-13.3%*
GunPoint	*2*	86.7%	**95.4%**	*10.0%*	84.5%	*-2.5%*	79.9%	*-7.9%*
GunPointAgeSpan	*2*	**85.8%**	84.9%	*-1.1%*	80.6%	*-6.1%*	63.6%	*-25.9%*
GunPointOldVersusYoung	*2*	**99.8%**	80.1%	*-19.7%*	77.2%	*-22.6%*	68.8%	*-31.0%*
ItalyPowerDemand	*2*	**90.0%**	80.9%	*-10.1%*	78.8%	*-12.4%*	62.6%	*-30.4%*
MoteStrain	*2*	**89.0%**	82.4%	*-7.4%*	79.0%	*-11.2%*	77.9%	*-12.5%*
Plane	*7*	**100.0%**	99.8%	*-0.2%*	99.2%	*-0.8%*	55.4%	*-44.6%*
SonyAIBORobotSurface1	*2*	**88.3%**	55.1%	*-37.6%*	53.7%	*-39.2%*	58.5%	*-33.7%*
SonyAIBORobotSurface2	*2*	**82.5%**	81.2%	*-1.5%*	79.7%	*-3.3%*	68.9%	*-16.5%*
SyntheticControl	*6*	**93.8%**	76.5%	*-18.5%*	76.6%	*-18.4%*	76.7%	*-18.2%*
ToeSegmentation1	*2*	85.6%	**96.2%**	*12.4%*	85.4%	*-0.2%*	77.5%	*-9.4%*
TwoLeadECG	*2*	**97.3%**	93.5%	*-3.9%*	89.8%	*-7.7%*	75.8%	*-22.0%*
UMD	*2*	68.2%	85.6%	*25.5%*	**85.9%**	*25.9%*	85.6%	*25.4%*
Wine	*2*	54.3%	**64.5%**	*18.9%*	57.2%	*5.4%*	50.0%	*-7.9%*
1st		*11*	*6*		*1*		*0*	
2nd		*4*	*9*		*3*		*2*	
Loss		*7*	*12*		*17*		*18*	
1st Rate (%)		*61%*	*33%*		*6%*		*0%*	
Loss Rate (%)		*39%*	*67%*		*94%*		*100%*	

Figure 20: *6 Shot Accuracy for each of the datasets against DPSN, BOSS and ST. The* `Diff` *column is a percentage difference between the accuracy of the given method and the FISH 256F.*

Dataset Name	Classes	FISH 256F	DPSN	*(Diff)*	BOSS	*(Diff)*	ST	*(Diff)*
ArrowHead	*3*	74.3%	**83.0%**	*11.8%*	74.5%	*0.2%*	47.8%	*-35.7%*
BME	*3*	**87.1%**	74.4%	*-14.7%*	86.1%	*-1.1%*	69.2%	*-20.6%*
CBF	*3*	**98.4%**	97.2%	*-1.1%*	96.0%	*-2.4%*	88.6%	*-9.9%*
Chinatown	*2*	**95.5%**	83.4%	*-12.7%*	81.7%	*-14.5%*	71.7%	*-24.9%*
ECG200	*2*	77.9%	**86.5%**	*11.0%*	78.0%	*0.1%*	67.1%	*-13.9%*
GunPoint	*2*	90.4%	**95.9%**	*6.0%*	88.0%	*-2.7%*	58.7%	*-35.0%*
GunPointAgeSpan	*2*	84.3%	**90.5%**	*7.4%*	87.5%	*3.8%*	83.8%	*-0.6%*
GunPointOldVersusYoung	*2*	**99.8%**	87.6%	*-12.2%*	85.3%	*-14.6%*	72.4%	*-27.5%*
ItalyPowerDemand	*2*	**91.5%**	84.9%	*-7.2%*	83.0%	*-9.4%*	77.9%	*-14.9%*
MoteStrain	*2*	**91.2%**	82.9%	*-9.1%*	80.6%	*-11.6%*	79.7%	*-12.6%*
Plane	*7*	99.3%	**99.9%**	*0.6%*	99.5%	*0.2%*	53.4%	*-46.2%*
SonyAIBORobotSurface1	*2*	**92.2%**	55.9%	*-39.3%*	54.2%	*-41.2%*	58.8%	*-36.2%*
SonyAIBORobotSurface2	*2*	81.4%	**84.1%**	*3.3%*	83.1%	*2.0%*	75.0%	*-7.9%*
SyntheticControl	*6*	**95.3%**	84.0%	*-11.9%*	79.8%	*-16.3%*	87.5%	*-8.2%*
ToeSegmentation1	*2*	87.6%	**96.5%**	*10.1%*	86.0%	*-1.9%*	79.9%	*-8.8%*
TwoLeadECG	*2*	**97.9%**	95.0%	*-2.9%*	91.4%	*-6.6%*	74.8%	*-23.5%*
UMD	*2*	67.1%	89.2%	*33.0%*	**91.0%**	*35.6%*	86.9%	*29.5%*
Wine	*2*	64.1%	**66.4%**	*3.5%*	57.4%	*-10.5%*	50.0%	*-22.0%*
1st		*9*	*8*		*1*		*0*	
2nd		*3*	*7*		*6*		*2*	
Loss		*9*	*10*		*17*		*18*	
1st Rate (%)		*50%*	*44%*		*6%*		*0%*	
Loss Rate (%)		*50%*	*56%*		*94%*		*100%*	

Figure 21: *8 Shot Accuracy for each of the datasets against DPSN, BOSS and ST. The `Diff` column is a percentage difference between the accuracy of the given method and the FISH 256F.*

As can be seen in figure 20 the FISH method gets the best overall accuracy in the 18 datasets, with a win rate of 61% in 6-shot. The second best method is DPSN with a win rate of 44%. An interesting observation is that in figure 21 it can be seen that DPSN seems to catch up to the FISH method.

A reason for this decrease in accuracy from the FISH method could be that while the method gets more data points, its feature count remains the same. This could be because there is less information that is able to be extracted for each shot for each feature. It could therefore be that there is a correlation between feature count and shot count, i.e. increasing shots also means that feature count should be increased.

However, in general, it can be seen that the FISH method is performing quite well against other state-of-the-art methods.

4.4 Intepretability Results and Discussion

While the performance is promising, a deeper look into the FISH method of interpretability is shown in this section. This is done by means of casework, where two of the datasets will

be used, being *GunPoint* and *Wine*.

4.4.1 Case: GunPoint

This casework will use the GunPoint dataset in order to illustrate how the FISH method can be used for interpretability, and as a result, we have chosen to use 4 features instead of a higher number, even though a higher number results in higher accuracy. The reason is, it is easier to reason and understand fewer features, and consequently, it could be argued that the fewer the features the better the interpretability. This is a trade-off, that has to be considered, which was further explained in the methodology section.

The GunPoint dataset has two classes which are named class 0 and class 1, which can be seen in the following two figures 22 and 23. What should be highlighted are the four features of the classes and the boxplot, wherein the boxplot can be used to quickly see how important the shapelet is for the class. This can be seen by the spread of the boxplot. For example, for class 1 it can be seen the last shapelet's boxplot has a huge spread, which means that this shapelet is not that important for the class. On the other hand, shapelet 1, which is the second shapelet, has a tiny spread, which indicates that this shapelet is essential for the class. Class 0 is somewhat similar, however, it can be observed that the last feature is more important for the class than for class 1. This information could be used as an argument for the classification, however, further information is probably needed to convince the reader.

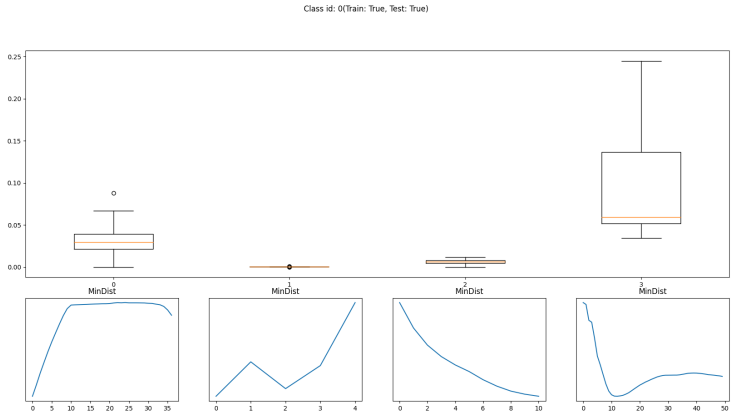


Figure 22: *The feature histogram for class 0 in the GunPoint dataset*

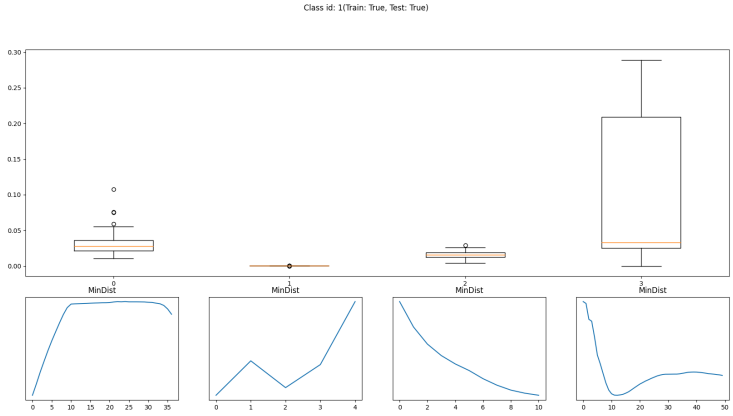


Figure 23: *The feature histogram for class 1 in the GunPoint dataset*

In addition to the earlier graphs, we also create a larger overview of the shapelets which can be seen in 24, and an overview of the original time series data in 25. It can be seen that Shapelets 1 and 2 are smaller than expected, however, when zooming in they have the same shape as in the prior example, and the feature extractor method identified that these small shapelets are very important for the classes. Together all these shapelets (features), can be used to differentiate between the classes, given how frequently they are seen in an example, while also considering how important the given shapelet is for the class.

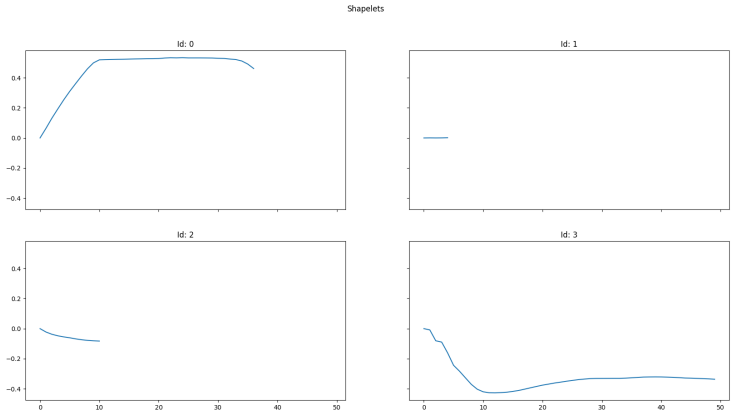


Figure 24: *All the four shapelets found in the GunPoint dataset*

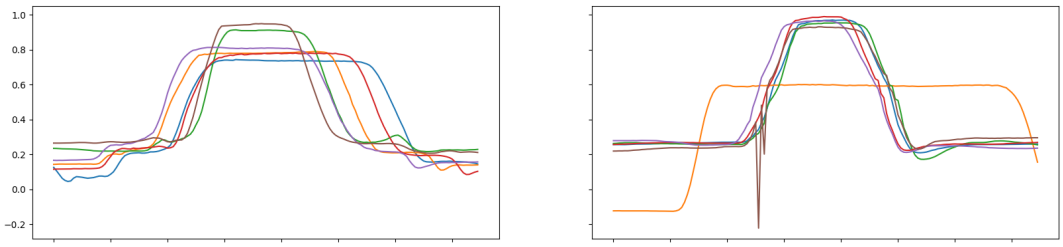


Figure 25: *Visual of how the original data in the GunPoint dataset looked like. The left figure shows 6 samples of class 0 while the right shows 6 samples of class 1.*

In the overview of the original time series data, it can be seen that the two classes are very similar, while the main observation that quickly springs into one's mind is that class 0 seems to be somewhat wider. One could see them as a normal distribution, where class 0 tends to have a somewhat larger standard deviation, however, this is not always the case. Moreover, it can be seen that class 1 seems to have more 'noise', e.g. the random vertical lines and the orange example which seems random, which could confuse the model. These previous points were also the reason that there was looked into outlier removal and smoothing of the data.

4.4.2 Case: Wine

This case study will look into the Wine dataset which shares some similarities with the GunPoint dataset, being that both have two classes and that the identified shapelets seem to have somewhat the same importance to each class, while the differences are only minute, however, they can be used for interpretability if one looks a bit closer.

Before dwelling deeper into the FISH graphs, it is worth exploring the original dataset which can be seen at 26. To the naked eye, the time series of class 0 and class 1 seems to be almost identical, which makes it especially tricky to classify and understand. At first glance, one might see that class 1 has a bit more noise, however, now the interesting part is how the FISH methodology can be used for interpretability.

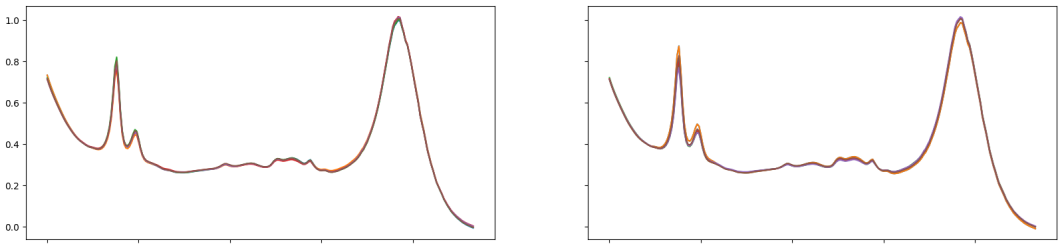


Figure 26: *Visual of how the original data in the Wine dataset looked like.*

The following FISH graph for class 0 and class 1 can be seen at 27 and 28. An overview of the four shapelets can be seen in 29, wherein their length is more clearly illustrated. It can be seen that all 4 extracted features for both classes have a very small spread in the boxplot,

which indicates they are important for the class. This furthermore makes sense, since the two classes look almost identical in the original time series data. However, if one looks closer it can be seen that all the shapelets for class 0 have a little wider boxplot, and when looking at the last shapelet, it can be seen that class 0 has many more outliers where the shapelet is less frequent. These arguments could be used to argue for interpretability, however, it is difficult to argue for since these classes are so similar, and the model only identified these minute differences.

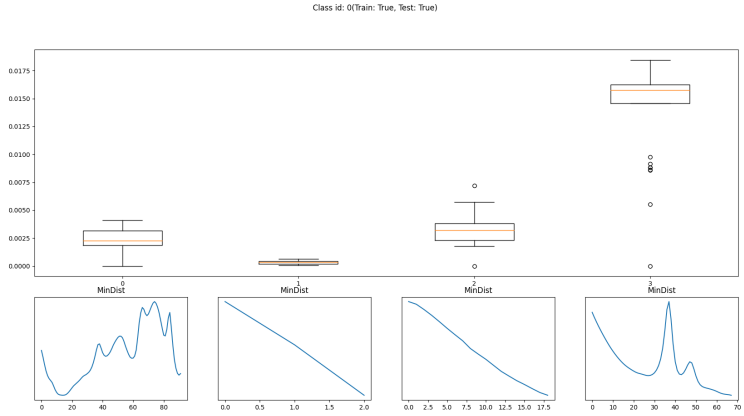


Figure 27: *The feature histogram for class 0 in the Wine dataset*

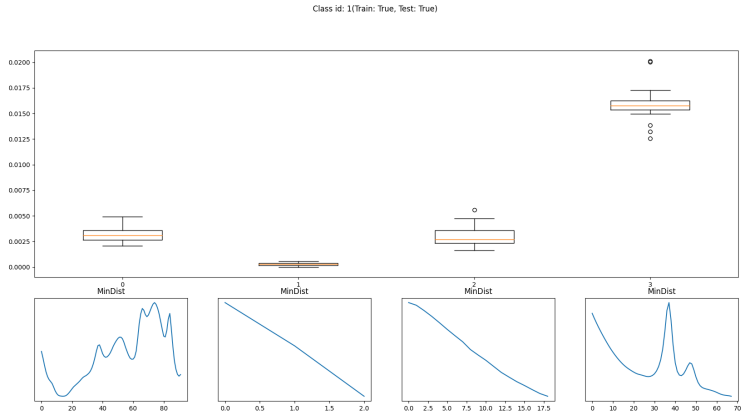


Figure 28: *The feature histogram for class 1 in the Wine dataset*

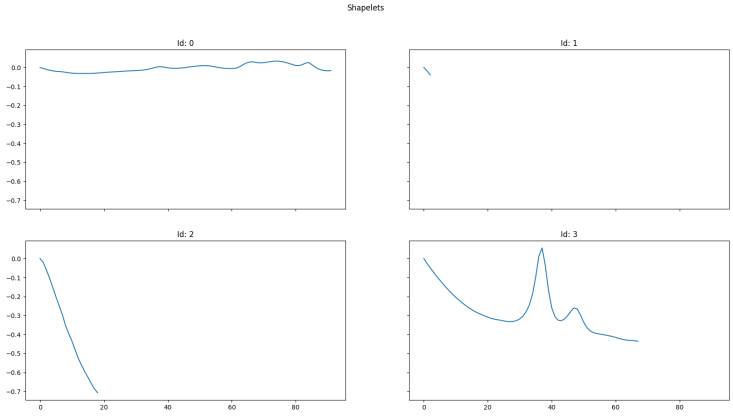


Figure 29: *All the four shapelets found in the Wine dataset*

5 Conclusion and Future Work

5.1 Conclusion

It can be concluded that the FISH method shows potential since it on average outperforms the state-of-the-art models *DPSN* and *BOSS* while giving interpretable results. In addition, in specific datasets such as *GunPoint*, our method outperforms the competition by a wide margin, even getting close to 100%. Another interesting aspect is that even in the datasets where the second best model wins, i.e. *DPSN*, the results from the FISH method are still more or less equivalent to the *BOSS* method.

The FISH method also allows for a trade-off between interpretability and accuracy, consequently making it possible for the user to select the desired outcome. As was seen in the interpretability cases, it is easy to reason about the results of the FISH method, using a combination of the shapelets and the FISH histograms.

However, it should also be stated that it does not seem that the FISH method scales as well with more shots as compared to *DPSN*, but comparable results were still found.

It can then be concluded, that the FISH method is an effective method for increasing accuracy for time series data in the few-shot field without hindering the interpretability of the results.

5.2 Future Work

While it was concluded that the FISH method performs well, there are several aspects that can be improved.

Currently, the FISH method only works with univariate time series. Extending the method to be able to process multivariate data would vastly improve the method's use cases in the field.

The running time of the FISH method is not great, mainly because the method uses naive implementations, and consequently, many aspects of the method could be optimised, where one of the major aspects is the pruning methods.

In addition, the sampling method is currently random, which might not be optimal, since some samples might provide greater information gain than others, thus prioritisation of these samples could be useful.

As stated, the ProtoNet was more or less left unchanged. The reason being the focus was on feature extraction. Most likely better settings and optimisations can be implemented into the ProtoNet to get even better accuracies.

Bibliography

[1] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[2] Yaqing Wang and Quanming Yao. "Few-shot Learning: A Survey". In: *CoRR* abs/1904.05046 (2019). arXiv: 1904.05046. URL: http://arxiv.org/abs/1904.05046.

[3] Yaqing Wang et al. "Generalizing from a Few Examples: A Survey on Few-Shot Learning". In: *ACM Comput. Surv.* 53.3 (July 2020). ISSN: 0360-0300. DOI: 10.1145/3386252. URL: https://doi.org/10.1145/3386252.

[4] 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*. KDD '09. New York, NY, USA: Association for Computing Machinery, 2009, 947–956. ISBN: 9781605584959. DOI: 10.1145/1557019.1557122. URL: https://doi.org/10.1145/1557019.1557122.

[5] Jake Snell, Kevin Swersky, and Richard S. Zemel. "Prototypical Networks for Few-shot Learning". In: *CoRR* abs/1703.05175 (2017). arXiv: 1703.05175. URL: http://arxiv.org/abs/1703.05175.

[6] Wensi Tang, Lu Liu, and Guodong Long. "Interpretable Time-series Classification on Few-shot Samples". In: *CoRR* abs/2006.02031 (2020). arXiv: 2006.02031. URL: https://arxiv.org/abs/2006.02031.

[7] Fazlollah M. Reza. *An introduction to information theory*. New York: McGraw-Hill, 1961.

[8] G. Kirchgässner, J. Wolters, and U. Hassler. *Introduction to Modern Time Series Analysis*. Springer Texts in Business and Economics. Springer Berlin Heidelberg, 2012. ISBN: 9783642334368. URL: https://books.google.dk/books?id=AfBunhJtxqwC.

[9] Yichang Wang. "Interpretable time series classification". Theses. Université Rennes 1, Sept. 2021. URL: https://theses.hal.science/tel-03509607.

[10] Leilani H. Gilpin et al. "Explaining Explanations: An Approach to Evaluating Interpretability of Machine Learning". In: *CoRR* abs/1806.00069 (2018). arXiv: 1806.00069. URL: http://arxiv.org/abs/1806.00069.

[11] Yisheng Song et al. "A Comprehensive Survey of Few-shot Learning: Evolution, Applications, Challenges, and Opportunities". In: (2022). arXiv: 2205.06743 [cs.LG].

[12] 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.

[13] Patrick Schäfer. "The BOSS is concerned with time series classification in the presence of noise". In: *Data Mining and Knowledge Discovery* 29 (2015), pp. 1505–1530.

[14] Yanping Chen et al. *The UCR Time Series Classification Archive*. www.cs.ucr.edu/~eamonn/time_series_data/. July 2015.