Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electric Power Engineering

Skreien, Tarjei Nesbø

Application of signal processing and machine learning tools in fault detection of synchronous generators

An applied approach

Master's thesis in Energy and Environmental Engineering
Supervisor: Arne Nysveen
June 2020

NTNU
Kunnskap for en bedre verden

Preface and acknowledgements

This Master's thesis is the conclusion of my five-year Master's degree in Energy and Environmental Engineering with the Department of Electric Power Engineering at the Norwegian University of Science and Technology. The work was carried out in the spring semester of 2020 under the supervision of Arne Nysveen and Hossein Ehya.

The thesis examines the application of machine learning to measurements taken of a salient pole synchronous generator for the purpose of detecting and diagnosing inter-turn short-circuits. It is a continuation of my specialisation project with the same title that focused on signal processing techniques applied to magnetic measurements of synchronous generators. The findings from that project were applied in feature extraction in this thesis. Where that was a signal processing application, this work is squarely focused on machine learning and its use. Since this is a new initiative by NTNU's Department of Electrical Engineering that strays from previous works, I am excited to be the department's first in what is hopefully a long succession of students working with machine learning. This thesis is therefore intentionally broad, and with an emphasis on the reasoning and justifications made throughout the process, so that it may serve as a beachhead into machine learning for later students.

The limitations imposed upon this project by the COVID-19 pandemic have severely reduced the scope and results. It is my hope that another Master's student can make use of the tools and insights developed during this thesis to investigate what I could not. Delving into these fields has nevertheless been rewarding, challenging, and elucidated the impact these techniques will have in the industry.

I would like to express my gratitude to my supervisor Arne Nysveen for giving me the opportunity to work with such a rewarding and novel project for my Master's thesis. The motivation and inspiration offered by being allowed to dive into something previously not attempted has been invaluable. To my co-supervisor Hossein Ehya, I would direct my appreciation to his cool professionalism and genuine interest in my work and progress. He exudes an image of the consummate professional, one that I can only hope to one day emulate. I would like to thank my friends, family and partner for their support and caring throughout my student years and before.

Abstract

The detection of faults in salient pole synchronous generators is crucial to ensure reliable production in hydroelectric power plants where these machines are located. A single inter-turn short-circuit (ITSC) in the rotor winding can create hot spots that induce insulation failure its neighbours, eventually causing the entire rotor pole winding to fail. This can also cause further mechanical faults caused by vibrations from the resulting uneven magnetic field. This can be avoided if ITSCs are detected early. This thesis examines if machine learning and signal processing can be used for on-line condition monitoring to reveal ITSC in salient pole hydropower generators. This was done by creating several machine learning classifiers to detect ITSC faults, utilising data sets that were constructed using signal processing tools.

A data set for machine learning was built using signal processing techniques to extract features from measurements of a salient pole synchronous generator operated under several different severities of ITSC fault. The features extracted were the power spectral density of integer multiples of the generator's mechanical frequency extracted by fast Fourier transform (FFT), discrete wavelet transform energies, and time series feature extraction based on scalable hypothesis tests (TSFRESH). Using this data set, a wide range of classifiers were trained to detect the presence of ITSC faults. The classifiers evaluated were logistic regression, K-nearest neighbours, radial basis function support vector machine (SVM), linear SVM, XGBoost decision tree forest, multi-layer perceptron (MLP), and a stacking ensemble classifier including all of the aforementioned. The classifiers were optimised using hyper-parameter grid searches. In addition, some feature selection and reduction algorithms were assessed such as random forest feature selection, TSFRESH feature selection, and principal component analysis.

Out of 475 features investigated, high decomposition level relative wavelet energy features, aggregate linear trend features, approximate entropy features, and change quantile features were the most useful features. FFT derived features performed poorly. Correlation to the target value was a strong indication that features will be useful in classification and could thus be used to screen a large number of potential features at the risk of missing features with non-linear relationships.

A general trend during optimisation was that linear machine learning models performed well and that the performance of non-ensemble classifiers *increased* as the complexity *decreased*. The best performance was yielded by a stacking classifier using the optimised Logistic Regression, SVM, MLP, and XGBoost classifiers as base-classifiers, and logistic regression as the meta-classifier. It correctly classified 84.48 % of samples in the hold-out data set, and 84.56 % of the faulty samples present were correctly classified as such. Of the samples that were classified as faulty, 92.74 % were correctly classified. The worst performance was exhibited by the K-nearest neighbours classifier, performing worse than random chance. This demonstrates that ITSC faults are suited to be detected using machine learning, however, these results should be confirmed on larger data sets that include other incipient faults.

Sammendrag

Deteksjon av feil i synkrongeneratorar med utprega polar er avgjerande for å sikre påliteleg produksjon i vasskraftverka der desse maskinene er lokalisert. Ei enkelt kortslutning mellom vindingar (ITSC på engelsk) i rotorfeltviklinga kan forårsake varmeutvikling som induserer isolasjonssvikt hos nabovindingane, og til slutt får heile rotorpolens vikling til å svikte. Det kan òg forårsake ytterlegare mekaniske feil grunna vibrasjonar som stammar frå det resulterande ujamne magnetfeltet. Dette kan unngåast om ein oppdagar ITSC tidleg. Denne oppgåva undersøker om maskinlæring og signalbehandling kan brukast til on-line overvaking av maskintilstand for å avsløre ITSC i vasskraftgeneratorar. Det vart gjort ved å trene fleire maskinlæringsmodellar til å oppdage ITSC-feil, med utgangspunkt i datasett som vart konstruert ved hjelp av signalbehandlingsverktøy.

Eit datasett for maskinlæring vart laga ved bruk av signalbehandlingsteknikkar for å trekke ut trekk frå målingar av ein synkrongenerator med utprega polar som drivast under forskjellige grader av ITSC-feil. Trekka som vart trekt ut var frekvenskomponenten til heiltalmultiplar av generatorens mekaniske frekvens ekstrahert med fast Fourier transform (FFT), diskrete wavelet-transformasjonsenergiar, og tidsserietrekkekstraksjon basert på skalerbare hypotetestar (TSFRESH). Ved hjelp av dette datasettet vart fleire maskinlæringsmodellar trent opp til å oppdage ITSC-feil. Maskinlæringsmodellane som vart evaluert var logistisk regresjon, K-nearest neighbours (KNN), radial basisfunksjon support vector machine (SVM), lineær SVM, XGBoost-beslutningstre-skog, fleirkappa perceptron (MLP) og ein stabelmodell av alle dei nemnde modellane. Modellane vart optimalisert ved hjelp av hyperparameterrutenettsøk. I tillegg vart tre trekkval og -reduksjonsalgoritmar evaluert.

Av 475 trekk som vart undersøkt, var relative wavelet-energiar (RWE) for høge nedbrytingsnivå, aggregerte lineærregresjonstrekk, omtrentlege entropitrekk og endringskvantile funksjoner dei mest nyttige funksjonane. FFT-avleidde trekk presterte dårleg. Korrelasjon med målverdien var ein sterk indikasjon på at trekk vil vere nyttige i klassifiseringa og kunne dermed brukast til å saumfare eit stort tal potensielle trekk, med fare for å gå glipp av trekk med ulineære forhold til målverdien.

Ei generell trend under optimaliseringa var at lineære maskinlæringsmodellar presterte bra, og at ytinga til modellane auka etter kvert som kompleksiteten gjekk ned. Den beste ytinga vart oppnådd av ein stabel av dei optimaliserte logistiske regresjon-, SVM-, MLP- og XGBoostmodellane som grunnlærarar og logistisk regresjon som metalærar. Stabelen klassifiserte 84,48% av prøvene i hold-out datasettet riktig, og 84,56% av dei prøvane med feil i vart riktig klassifisert. Av prøvane som vart klassifisert som feil, var 92,74% korrekt klassifisert. KNN hadde den dårlegaste ytinga. Dette viser at ITSC-feil kan bli oppdaga ved bruk av maskinlæring, men desse resultata bør bekreftast på større datasett som inkluderer andre typar feil.

Contents

List of Figures

List of Tables

Acronyms

ALTL Aggregated Linear Trend Line. 57

ANN Artificial Neural Network. 24, 27

CV Cross-Validation. xxx, 48, 49, 61, 63

CWT Continuous Wavelet Transform. 9, 10

DFT Discrete Fourier Transform. 7

DWT Discrete Wavelet Transform. ix, 9–11, 37, 39–41, 67

EDA Exploratory Data Analysis. xvii, 39, 45, 48, 62

FFT Fast Fourier Transform. 7, 9, 37–39, 41, 64, 67

HWE Hierarchical Wavelet Energy. 10, 11

ITSC Inter-Turn Short-Circuit. 2, 10, 29, 39, 40, 47, 67

IWE Instantaneous Wavelet Energy. 10, 11

KNN K-Nearest Neighbours. 19, 20, 39, 41, 47, 49, 53, 54, 59, 67

MLP Multi-Layer Perceptron. 24, 26, 27, 47, 53, 54, 59, 62, 63, 67

OSS Original Sample Series. v, 35, 36, 45, 49, 59, 61

PCA Principal Component Analysis. 39, 41, 47–49, 51

RSS Reduced Sample Series. v, 35–38, 49, 59, 61, 62

RWE Relative Wavelet Energy. 10, 11, 62, 67

SVM Support Vector Machine. 20, 39, 41, 47, 49, 53, 54, 61, 62, 67

TSFRESH Time Series Feature Extraction Based on Scalable Hypothesis Tests. xii, 67

TWE Teager Wavelet Energy. 10, 11

Chapter 1

Introduction

Salient pole synchronous machines are the machines most commonly used in hydroelectric plants [1], and so are ubiquitous throughout the Norwegian power system. In fact, hydroelectric generation accounted for 95 % of the total electric energy produced in Norway in 2018 [2]. Failure of the synchronous generators that generate the electricity the Norwegian society is run on can incur not only a great expense in restoring the plants, but also a large cost to society. These machines are under ever-increasing operational demands as intermittent power sources enter the power system. The proper running and maintenance of synchronous machines, and by extension the timely detection and diagnosis of their faults, is very important. Hydroelectric generators can suffer failure as a result of undetected incipient faults that induce larger faults. The state-of-the art in on-line fault detection in salient pole synchronous generators is still lacking in this respect.

Machine learning and its associated techniques have quickly matured in recent years. New techniques, coupled with ever increasing computational resources, have made possible new approaches to asset management and monitoring. In the transition from reactive to predictive maintenance, it is vital with accurate estimations of the machine states. This involves integrating sensors, signal analysis, and decision-making algorithms. The potential benefits to society are immense, estimated by McKinsey Digital to reach a total potential economic value of 11 trillion USD in 2025 [3], and the power generation sector is no exception.

By applying on-line condition monitoring, incipient machine faults can be detected in real-time and faults can be detected before they cause unscheduled stops and further damage to the machine.

1.1 Project description and scope

The aim of this thesis is to judge the applicability of machine learning in an on-line condition monitoring system for detection of inter-turn short-circuits in salient pole synchronous machine rotor field windings. To this end, measurement series of the air-gap magnetic field in a synchronous machine operating under varying known fault conditions are analysed using signal analysis tools to extract features. These features are then used to train a classifier to identify the fault conditions of the machine. The approach taken is exploratory, testing a wide range of methods for every part of the process. The research questions are specifically:

Which features of the ones generated are most useful?

Which machine learning models perform best?

Is a single air-gap magnetic field sensor sufficient for reliable ITSC fault detection or are more sensors required?

To answer these questions, a fault classification system builder has been created that includes:

- Automatic sample processing and segmentation from longer sample series
- A feature extraction process capable of processing and organising an arbitrary number of samples
- Exploratory data analysis that gives insight into data distribution and feature redundancy
- A feature selection process that employs several feature selection methods
- A process to assess the usefulness of feature selection, select the best machine learning model among several, and assess the performance of the final model
- A final classifier to detect ITSC faults

1.2 Limitations

The COVID-19 pandemic limited the scope of this thesis. The outbreak of disease, and subsequent closing of the university, hindered the planned gathering of experimental data and turned time spent preparing for experiments into wasted time. It was initially planned to gather a large data set including measurements of several incipient faults of differing severity and different combinations of faults. This would be done using a large sensor suite that integrated concurrent readings of air-gap magnetic field, voltage over and current through stator and rotor windings, and stator vibration measurements. The machine was also to be run in several distinct load conditions for each fault condition.

In place of that data, an inter-turn short-circuit data set gathered from the same machine in 2019 was used [4]. This limits what can be investigated. The data set contains measurements done only for two load conditions, no-load and full-load, and one fault type, inter-turn short-circuits. Only one sensor type, Hall-effect sensors, was used. Machine learning is fuelled by data, and its lack is severe impediment.

The limitations this imposes include, but are not limited to, the following:

- Only one fault condition, inter-turn short-circuits, can be investigated.
- How the results are affected by other incipient faults cannot be investigated.
- Only one type of sensor, air gap mounted Hall-effect sensor, can be evaluated.

- A comparison of the usefulness of different sensors cannot be done.
- The robustness of the classifier in the face of differing load conditions cannot be tested.
- The performance of the classifiers, as well as the ability to accurately assess their performance, suffers for lack of data.

1.3 Structure of the report

Chapter 1, this chapter, introduces the area of study and motivates the need for this thesis work. Furthermore, it describes the objective, scope, limitations and research questions of the work. Directly relevant previous work at NTNU is also summarised.

Chapter 2, Theoretical background, is a review of signal processing and machine learning techniques. The signal processing techniques explained are fast Fourier transform, continuous wavelet transform, and discrete wavelet transform. In addition some core concepts of supervised machine learning are explained along with logistic regression, k-nearest neighbours, support vector machines, decision tree learning, and artificial neural networks.

Chapter 3, Method and results, describes the construction of a classification system. This includes data acquisition and management, feature extraction and selection, deployment of contending machine learning models, and a final model selection.

Chapter 4, Discussion, is a discussion of the applicability of the machine learning techniques in light of the theory and models developed in Chapter 3.

Chapter 5, Conclusion, concludes the report, and makes recommendations for further research.

Appendix A lists the data available.

Appendix B contains the code for the implementations realised in Chapter 3.

Appendix C is an overview of the features calculated by the TSFRESH feature extraction algorithm.

1.4 Previous work

A Master's thesis investigating the use of magnetic flux monitoring for the purpose of detecting inter-turn short-circuits, eccentricity and broken damper bars was performed in the spring of 2019 at the Department of Electrical Engineering, NTNU. In the thesis, hall-effect sensors were placed onto stator teeth in opposite ends of the airgap to measure the magnetic field during operation of a 14-pole 100 kVA machine. The frequency spectra of the measurements taken with and without induced faults were compared to identify differences. A finite element method simulation of the same faults indicated that they

are possible to identify based on the sensor measurement series frequency spectra, but imperfections in the generator obfuscated the fine changes induced by low severity faults. It was found that the harmonics $f_k = k\frac{f_{synch}}{p}Hz$ in the interval 0 to 200 Hz increased with greater number of turns short-circuited. The measurements also indicated that both dynamic and static eccentricity could be detected using the method, while damper bar breakage is not discernible at synchronous operation. The author suggested the signals be investigated further with signal processing tools capable of distinguishing non-stationary frequency components and artificial intelligence techniques [4].

In a specialisation project conducted in the fall of 2019, the signals gathered in [4] were investigated further using signal processing tools. The signal processing techniques investigated were fast Fourier transform, short-time Fourier transform, continuous wavelet transform, discrete wavelet transform and Hilbert-Huang transform. Furthermore, a review of was conducted of support vector machines, decision tree learning, k-nearest neighbours, and artificial neural networks to evaluate their suitability for the task. To determine if the signal analysis tools could be used to detect a rotor field winding inter-turn short-circuit fault, they were applied to air gap magnetic field measurements of a healthy machine and a faulty machine with 10 rotor winding turns short-circuited operating at full load. The fault could be detected in the short-time Fourier and continuous wavelet transforms as a weakening of the 50-100 Hz frequency band. The continuous wavelet transform additionally showed characteristic arching below 25 Hz in the faulty signal. The instantaneous, Teager and hierarchical wavelet energies of the discrete wavelet transform were elevated in the faulty case [5].

Chapter 2

Theoretical background

In this chapter, a theoretical background is lain for the thesis work. This is done in three parts. First is a short section about incipient faults and condition monitoring, second is an examination of the signal processing techniques used for feature generation and lastly is a review of machine learning techniques and considerations. As this master's thesis concerns the same subject matter as a specialisation project by the same author, some theory is adapted from the aforementioned specialisation project report written autumn 2019 [5].

2.1 Incipient faults

The fault in focus in this thesis is what is termed an incipient fault. Incipient faults are the faults that do not themselves significantly compromise the performance of the machine, but that could lead to larger faults and eventual machine failure. Specifically, we will look at methods to detect inter-turn short-circuits in the field winding.

2.1.1 Rotor field winding inter-turn short-circuits

The rotor winding inter-turn short-circuit (ITSC) is the failure of isolation between turns in the rotor winding coil so that the number of turns in the coil is effectively reduced [1]. This can be due to overheating causing damage to the isolation, thermal deformation or mechanical stresses [6], [7]. The fault can then propagate to cause the rotor winding to be further short-circuited and eventually a short to ground [7]. Another issue that could arise from the uneven magnetic field is uneven mechanical stresses that further compromise other machine components [7]. The pole-drop test is the most commonly applied off-line test to detect short-circuited turns in the field winding [7]. It is done by applying low voltage AC to terminals of the field winding and measuring the voltage across each pole. A faulty pole will have a lower voltage across it compared to the other poles [6], [7]. The disadvantage of this test is that it requires the machine to be taken off-line. Off-line tests require shut-down of the machine and are therefore expensive. They also occur while the machine is at a standstill and therefore faults that are induced due to rotational forces can become invisible during the tests [7]. To find the faults present during operation, it is necessary to conduct on-line monitoring and tests [7]. On-line condition monitoring for diagnosing rotor winding ITSC is often done using flux

probe measurements, where the magnetic field registered by a flux probe placed on a stator tooth in the air gap of the machine is analysed by comparison to a healthy case [6], [8].

2.1.2 Condition monitoring

To predict the need for maintenance in machines and to avoid breakdowns, *condition monitoring* systems are used. The machines are monitored so that one can detect faults in the machine by recognising shifts in trends among the monitored characteristics. The field concerns itself with the modelling of the machines, application of measurement equipment, and the analysis of that data to predict trends [9].

Condition monitoring systems include sensors, data acquisition, fault detection and diagnosis. *Sensors* are the hardware that is placed onto the machine to measure some physical characteristic. *Data acquisition* is the collection of techniques that pre-process the sensor outputs to ensure that the data produced can be of use. *Fault detection* is the comparison of data series to what is expected. This can be done by comparing the data to a model of the machine and/or by employing feature extraction methods, i.e. signal processing, to create a *signature* that is examined for fault indications. *Diagnosis* is the post-processing of the abnormal signals to determine the type and severity of the fault [9].

Condition monitoring techniques should be non-invasive, making the least possible intervention in the machines they monitor.

2.2 Signal processing tools

The field of digital signal analysis is ever broadening as new techniques mature and computational resources enable these tools to be applied closer and closer to real time. Signal processing is the act of taking raw data and applying some mathematical operations upon them to thereby gain insight into its components. There are several techniques that do this and their applications depend upon their capacity to handle certain signal properties.

2.2.1 Fast Fourier transform

The Fourier transform represents a target function as its constituent harmonic components, its Fourier series. The Fourier transform is a convolution operation as shown in (2.1). A Fourier series is a periodic function and thus a true representation of the target function requires that the function also be periodic. Many functions are not periodic and are defined only within a range. This constraint is met by all measurement series, as a measurement series is necessarily undefined outside of the experiment. The Fourier transform is then applied with the assumption that the defined function range is one period of said function. The requirements that a signal must fulfil to be Fourier transformed is that it is absolutely integrable and that within any finite time interval it has a finite number of minima, maxima and discontinuities [10]. These conditions are fulfilled for all real signals.

$$\hat{x}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} x(t) e^{-j\omega t} dt = \frac{1}{2} a_0 + \sum_{k=1}^{\infty} (a_k cos(2\pi kt) + b_k sin(2\pi kt)) \tag{2.1}$$

To apply the Fourier transform to discrete signals, the discrete Fourier transform (DFT) was formulated. It involves converting a discrete time signal recorded at fixed sampling intervals into a discrete representation of the signal in the frequency domain, and the fast Fourier transform (FFT) is a commonly applied approach to achieve the DFT [11]. FFT is computationally efficient, reducing the complexity of DFT from $O(N^2)$ to $O(N \cdot log(N))$, where N is the total number of samples [11]. The resulting spectral representation of the time signal is a collection of periodic components in the frequency domain, each with a specific frequency, amplitude and phase angle. The DFT of a sequence of samples $\{x(k)\}$ is calculated using equation 2.2 [10].

$$X(s) = \sum_{n=0}^{N-1} x(k) \cdot e^{-i2\pi \frac{sk}{N}} \tag{2.2}$$

The FFT is used by itself, and often as an initial analysis to gain an overview of the frequencies present in the signal. Alternating current machines are systems that lend themselves well to analysis by Fourier transform due to their stationarity. The Fourier transform analyses the entire input signal at once and thus the time information is lost [10]. Hence, it is not suited for analysing non-stationary signals if the frequencies' temporal location is of interest.

2.2.2 Continuous wavelet transform

The continuous wavelet transform is a technique that extracts frequency components from a signal by convolution. The transform convolutes the signal with a *wavelet* instead of running the Fourier transform [12]. The wavelet has compact support, this means that it is a signal that starts and ends in zero and the integral along its axis is zero [12]. The wavelet is therefore effectively also the windowing function of the operation. The wavelets can be stretched and compressed by changing the scaling factor, a, which enables the convolution integral to pick out different frequencies. The convolution computation is as given in (2.3), adapted from [12]. The notation presented is for continuous wavelet transform of a continuous signal, the discrete signal case is similar. The convolution is applied along the signal for several values of a and the result is combined into a scalogram that depicts the signal components.

$$X(a,b) = \int_{-\infty}^{\infty} x(t)\Psi_{a,b}^{*}dt \tag{2.3}$$

Here the signal to be analysed is denoted by $x(t)$, and the wavelet is given by $\Psi_{a,b}^{*}$ which is dependent upon the coefficients a and b that adjust the scale of the wavelet and its temporal centre, respectively. The wavelet equation is shown in (2.4).

$$\Psi_{a,b}(t) = \frac{1}{\sqrt{a}}\Psi\left(\frac{t-b}{a}\right) \tag{2.4}$$

The mother wavelet, Ψ, is the shape of the wavelet, and there are several different of mother wavelets available. The choice of mother wavelet depends on the characteristics of the signal one is investigating and the properties of interest. For example, the Morlet wavelet is used to pick out smooth variations while the Haar wavelet is more suited to pick out sudden transitions [13]. The Morlet and Haar wavelets are shown respectively in figures 2.1 and 2.2. A rule of thumb is that one looks for a mother wavelet that is similar in shape to the signal being analysed.

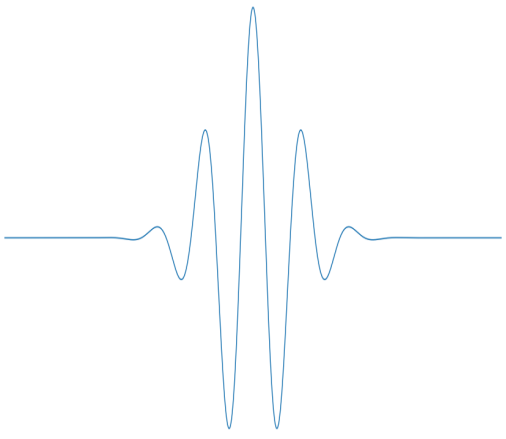


Figure 2.1: The Morlet wavelet.

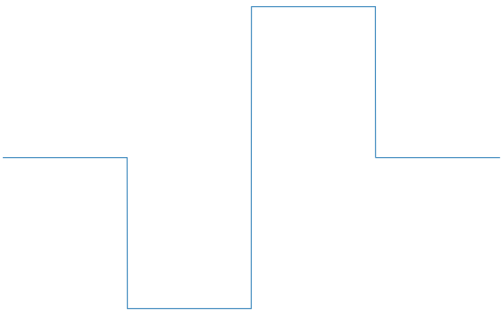


Figure 2.2: The Haar wavelet.

2.2.3 Discrete wavelet transform

The discrete wavelet transformation (DWT) is built upon the same principle as the continuous wavelet transform, that convolution by mother wavelet can extract interesting features from the signal, but its implementation is very different. A very common implementation of DWT is the filter bank implementation. The algorithm functions as a cascade of filters, where each filter corresponds to a level or scale. Each level is composed of a high- and low-pass filter in parallel followed by a downsampling by 2, see figure 2.3. The signal is run through both branches and the result from the high-pass filtering plus downsample is stored as detail coefficients of that level while the results from the low-pass filtering plus downsample, known as the approximation coefficients, are passed to the next level as its input signal. The DWT is several filters set in succession as shown in figure 2.4.

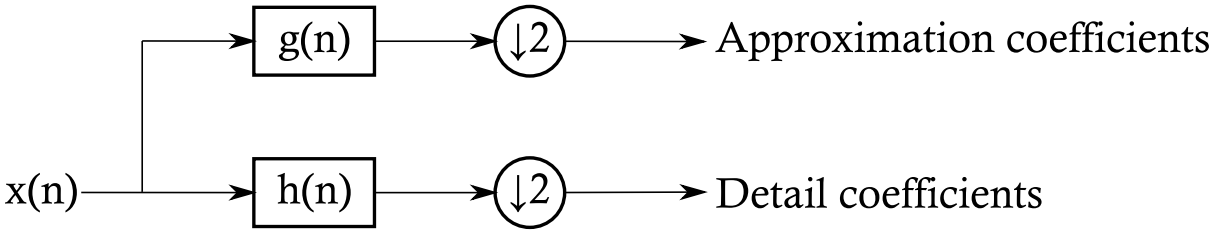


Figure 2.3: One level of the DWT.

This continues until the desired number of decompositions are made. The last low-pass filtering is returned along with the rest of the decompositions. The filters are derived from the chosen mother wavelet. The filters can be kept the same length, 2 for Haar wavelet, since the signal is downsampled in each level. The operations necessary are therefore halved in each level. The filter is shifted by a whole filter length for each application, ensuring no overlap or redundancy.

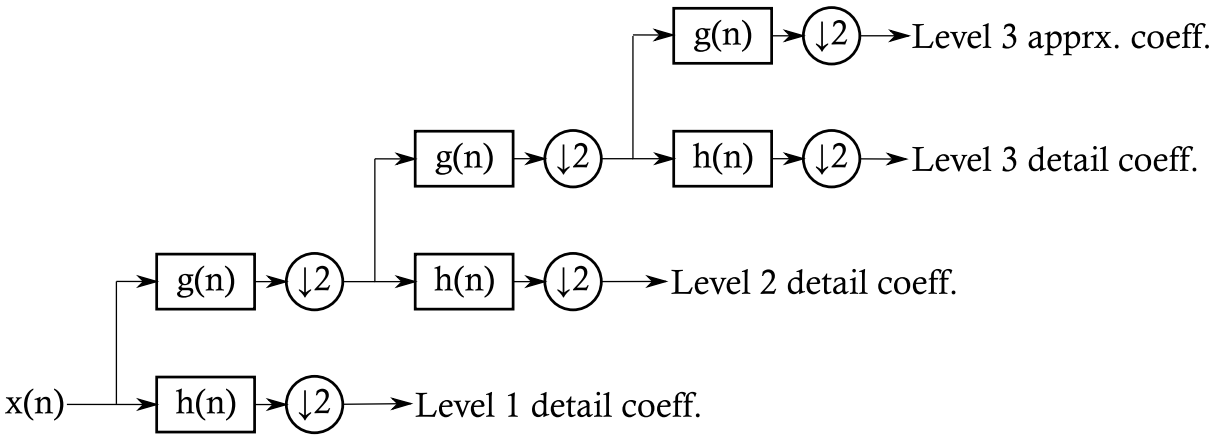


Figure 2.4: A filter bank of cascading filters, equivalent to a 3-level DWT.

The frequencies contained within each decomposition level in a 3-level DWT of a signal sampled with sample rate f_{samp} is given in Table 2.1.

DWT is much faster than CWT due to the downsampling in each stage, and can be as quick as $O(n)$. This is faster than even the FFT with $O(N \cdot log(N))$. CWT gives better

Table 2.1: The frequencies contained within each DWT decomposition level of a signal of length N and sample rate f_{samp}.

Level	Frequencies	Number of coefficients
3	0 to $\frac{f_{samp}}{8}$	$\frac{N}{8}$
	$\frac{f_{samp}}{8}$ to $\frac{f_{samp}}{4}$	$\frac{N}{8}$
2	$\frac{f_{samp}}{4}$ to $\frac{f_{samp}}{2}$	$\frac{N}{4}$
1	$\frac{f_{samp}}{2}$ to f_{samp}	$\frac{N}{2}$

temporal resolution since it can shift the filter only by one sample, but DWT results in much lower storage requirements. When processing many samples in applications such as generating training data for machine learning, this can be important. Features such as mean, median, standard deviation, variance, skewness, kurtosis, entropy and various energies can be extracted from the DWT decompositions to be used in machine learning applications. Energy contents such as the Instantaneous Wavelet Energy (IWE), Teager Wavelet Energy (TWE), Hierarchical Wavelet Energy (HWE) and Relative Wavelet Energy (RWE) give an indication of the energy in each frequency band, and can serve to differentiate faulty from healthy conditions in asynchronous machines [14]. Results from [5], indicated that Inter-Turn Short-Circuit (ITSC)s are accompanied by consistently higher IWE, TWE and HWE of the frequency bands 6-12 Hz and 12-24 Hz than the healthy signal.

A note about nomenclature: Even though the technique is called *discrete* wavelet transform, in signal processing applications both CWT and DWT are implemented discretely. The difference is that CWT is defined continuously and ideally performs an infinite number of shifts of infinitesimal length, while DWT is expressly a discrete algorithm and shifts the length of the wavelet.

Wavelet energies

Given a K-level DWT of a signal where each decomposition level contains N_j coefficients $w_j(r), r = 1..N_j$, several energies can be extracted that reflect some property of the frequency bands.

Instantaneous wavelet energy is a good indicator of the amplitude in each decomposition level. It applies a conventional signal energy calculation, and is computed as shown in (2.5).

$$IWE_j = log_{10}\left(\frac{1}{N_j}\sum_{r=1}^{N_j}(w_j(r))^2\right) \tag{2.5}$$

Teager wavelet energy is more noise robust and can be computed as shown in (2.6).

$$TWE_j = log_{10}\left(\frac{1}{N_j}\sum_{r=1}^{N_{j-1}} |(w_j(r))^2 - w_j(r-1)\cdot w_j(r+1)|\right) \tag{2.6}$$

Hierarchical wavelet energy analyses the centre of each decomposition level. Since DWT downsamples by 2 in each level, the signal may have had to be padded to fit into an integer number of filter applications. This can cause end-effects that affect the energies of IWE and TWE, HWE avoids this effect by ignoring the first and last portions of the coefficients. HWE is given by (2.7), where N_J is the number of coefficients in the level over the current level.

$$HWE_j = log_{10}\left(\frac{1}{N_j}\sum_{r=\frac{N_j-N_J}{2}}^{\frac{N_j+N_J}{2}} (w_j(r))^2\right) \tag{2.7}$$

To compare the energy distribution among the frequency bands, the relative wavelet energy can be taken of the wavelet energies. The RWE of each decomposition level is as shown in (2.8) [15]. E_j (2.9) is the energy of each level and E_{total} (2.10) is the sum of energies across all K levels. RWE has been used successfully with artificial neural networks [15].

$$RWE_j = \frac{E_j}{E_{total}} \tag{2.8}$$

$$E_j = \sum_{r=1}^{N_j}(w_j(r))^2 \tag{2.9}$$

$$E_{total} = \sum_{j=1}^{K} E_j \tag{2.10}$$

2.3 Machine learning

Machine learning is a part of the field of artificial intelligence and concerns itself with enabling machines to learn to solve problems without being explicitly programmed [16]. There are three main types of machine learning: supervised, unsupervised, and reinforcement learning [17]. Supervised learning trains a model to classify a sample into one of several pre-defined groups or approximate some unknown value from a sample. Unsupervised learning takes in unstructured data and looks for patterns in the data. Reinforcement learning is the training of a model to take the correct action in order to maximise some reward. For the purpose of detecting and diagnosing faults, supervised learning is most suited. The strength of machine learning is that one only pre-processes the data and selects the model type and parameters, and then the model is trained based on the training data. Machine learning also offers advantages over many traditional fault detection techniques in that the fault detection can be automated once the model is trained and that the models will be tailored to each application, predicated upon the availability of data.

2.3.1 Supervised learning

Classification models predict to which class among those labelled in the training set a sample belongs to, while *regression* models seek to predict a value. An electrical engineering use case of this that illustrates the difference between classification and regression could be to predict the power consumption in a neighbourhood. If the task is predict if the load will be above a certain threshold, it is a classification task. If the task is to estimate how many kilowatts will be consumed, it is a regression task. To do be able to do this, the models are trained using labelled training data. Labelled training data are samples with known classes or target values associated with them. Supervised learning models are often simply referred to as predictors or classifiers. Some examples of supervised learning models are support vector machines, decision tree machines, artificial neural networks, and K-nearest neighbours. When training a predictor, it naturally becomes well suited to classify the training data but may not classify new samples very well. The goal of any predictor is to be as general as possible, this means that it maintains its predictive power across a range of inputs. The phases of creating a classifier can be divided into feature selection, balancing, training and validation, and testing.

2.3.2 Feature generation and selection

The data is the basis of any machine learning model. To avoid having a too complex model it is desirable to limit the size of each sample, and an additional non-informative feature can actually degrade the performance of the model [18]. To generate these features, the signal processing methods presented in Section 2.2 can be used in concert with discipline knowledge. From a frequency spectrum generated by a signal processing method, one would select the frequencies of the signal that are most informative and generate some features from that. This could be the energy spectrum of a certain decomposition level in discrete wavelet transform, the intensity of some side-band frequencies relative to a harmonic frequency, or any other property of the signal or its transforms. Methods also exist to generate features automatically from a time series, a notable example that is also

capable of feature selection is the TSFRESH feature generation package shown below. With all the features generated, selecting the best among them can make the data more amenable to visual representation, reduce the storage requirements, and reduce training times to improve prediction performance [18]. If the features that are generated initially are good, this step will not impact the data meaningfully. Another feature selection technique is Random Forest feature selection, described closer in section 2.3.10 about decision tree learning.

Time series feature Extraction based on scalable hypothesis tests (TSFRESH)

An algorithm to extract features from time series, called FeatuRe Extraction based on Scalable Hypothesis tests (FRESH) is proposed in [19]. Its intent is to automate time series feature extraction while implementing feature selection. The process is highly parallelised, enabling fast high-volume feature extraction while selecting the most relevant features for the prediction task. A feature is assessed to be relevant if the feature is not statistically independent from the target predictions. This was done by using the statistical inference technique of hypothesis testing which computes a p-value between each feature and the target that quantifies the probability that the feature is not relevant for predicting the target. The features are then selected amongst by rejecting all features with a p-value above a threshold. The algorithm is recommended to be used in concert with principal component analysis to further reduce the number of features.

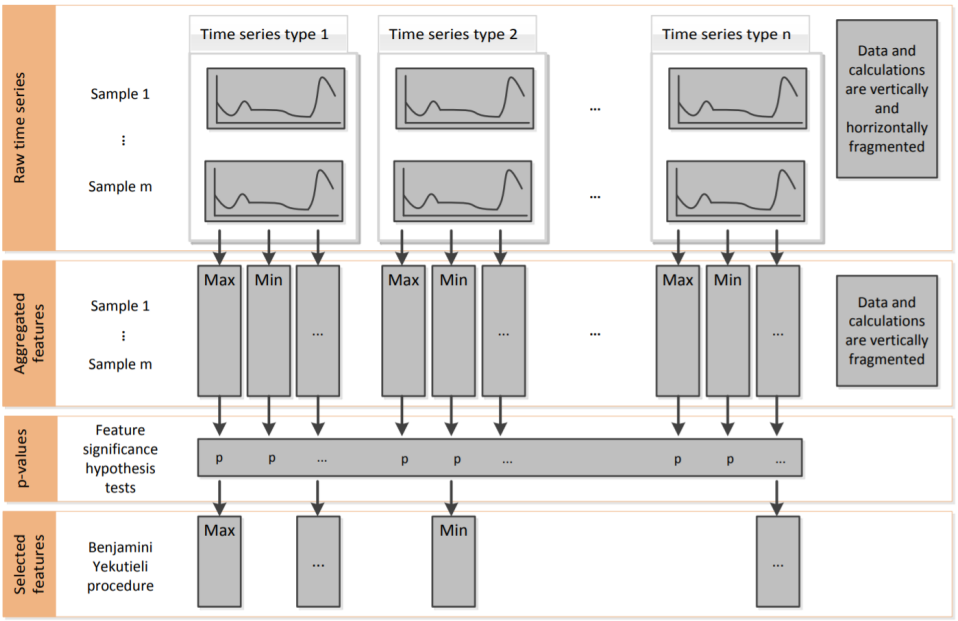


Figure 2.5: The feature extraction and selection process [19].

The FRESH algorithm was integrated into a algorithmic feature generation package, called Time Series FeatuRe Extraction based on Scalable Hypothesis tests (TSFRESH) [20]. TSFRESH is able to generate a total of 794 time series features, using 63 time series characterisation methods as well as apply feature selection methods. TSFRESH run time scales linearly with the number of features extracted, the number of samples, and the number of different time series. It does not scale linearly with respect to the length of the

time series for some more advanced features such as calculation of the sample entropy. Adjusting the calculated features can drastically impact the run time of the algorithm. The researchers showed that TSFRESH worked well to extract relevant features from ensembles of torque sensors in a robot to determine failure of execution of a task as well as an industrial data set from steel production.

2.3.3 Balancing the data set

Depending on the training data set available and the design of the classifiers, it may be necessary to balance the data set if one of the classes are over-represented [21]. This can be due to one class being more frequently observed. In the context of electrical machines, it can reasonably be expected to be able to generate more sample series of machines running without fault than with a fault if the data is drawn from industry. Another reason that data sets can be imbalanced is due to the design of the classifiers. It may be due the use of "one-versus-all" classifiers, where the classifiers attempt to separate each class from the rest. If five classes occur with similar frequency in the data set, i.e. the data set is balanced at the outset, the "one-versus-all" method would make the training data imbalanced for each of the classifiers. To rectify imbalanced classes, some approaches commonly taken are: collecting more data, weighting the classes according to their frequency, using evaluation metrics that correct for imbalance, and resampling [21].

2.3.4 Training and testing

The general process of making a predictor is to choose the model/algorithm, initialise it with random learning variables and define a cost function. The variables are what are trained in a model and the cost function defines how each variable affects the output. Before starting the training procedure, the data set is split into a training and test set as shown in 2.6. The models are then trained by applying gradient descent to minimise the cost function. This is done by introducing labelled samples from the training set to the predictor and comparing the model output to the sample label and adjusting the weights to nudge to output in the correct direction. This continues until either there are no more samples in the training set or until some early stopping criteria is met. The reason early stopping can be desirable is that the model can be over-fitted to the data and give worse predictions for samples outside of the training set if allowed to continue. When the model is trained, its performance is evaluated using a test set of samples not used in the training of the model. How well it predicts the labels of the test set, decides the performance of the model.

Since data sets are not entirely uniform, the results of the train/test procedure are affected by the way the data is split. One split may by chance give very good test results, while another does the opposite. This could result in selecting a model that generalises poorly even though it performs well on the test set. To counter this, k-fold cross-validation, as seen in figure 2.7, can be used [22]. k-fold cross-validation takes in a data set and makes several splits, or folds. Each fold is composed of a training set and a validation set. For each fold, the model is trained on the training set and its performance measured on the validation set. The models performance is then the average performance

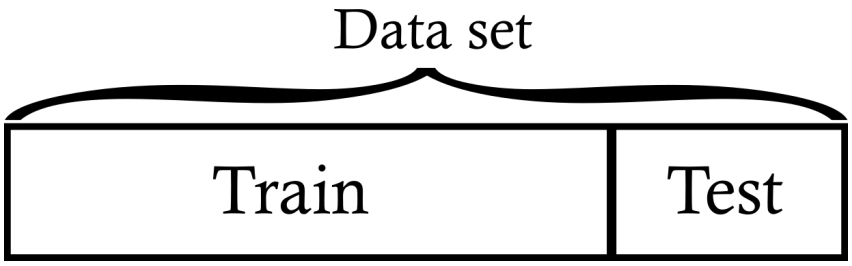


Figure 2.6: Train/test split of a data set.

across all the folds, and the performance is more likely to reflect the true performance of the model on unseen data.

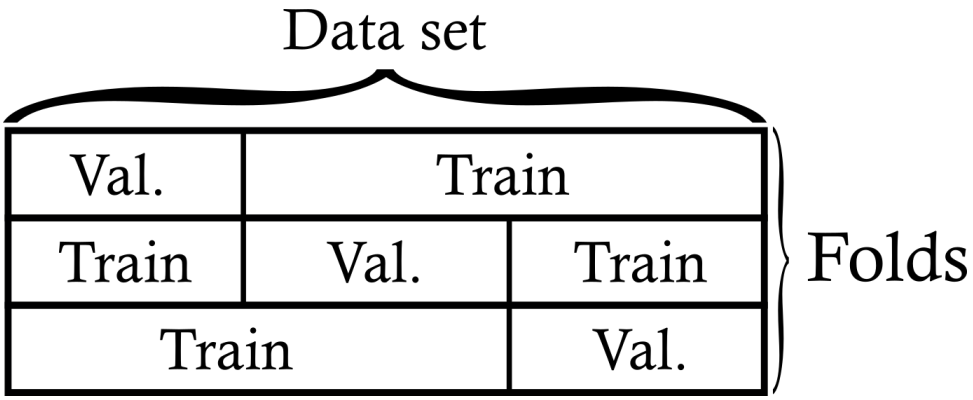


Figure 2.7: Three-fold cross-validation. Each fold is composed of a training and validation set.

If there are several candidate models or model configurations to choose from, the train/test or cross-validation procedure can be repeated for each one and the best one can be selected. However this presents an issue: In selecting the model based on its performance on the test set, the test set is effectively included in the model. The performance estimate of the model is therefore likely to be optimistic. This is known as *The Optimism Principle*, which states that selecting the model on the data that gave it birth will likely work better for these data than for almost any other data that will arise in practice [22]. Since what is of interest when testing a new model is its performance on *new and unseen data*, a part of the data set should be set aside to be used only to assess the performance of the model. This is known as a *hold-out dataset*, as shown in figure 2.8. The entire model selection and tuning process is then done without the hold-out data set, which is only used to evaluate the performance of the final model(s).

2.3.5 Evaluation metrics

There are several ways to evaluate the performance of classifiers, and they give differing results. Perhaps the simplest method is to count the number of correct classifications and divide by the total number of samples. This is what is called the *accuracy* of the classifier, shown in (2.11). It says something about the performance of the classifier, but has trouble with unbalanced data sets. Given an unbalanced electric machine measurement data set containing 99% of samples of healthy machines and 1% of samples of faulty machines, a classifier that always classifies a sample to be healthy would have a 99% accuracy. This

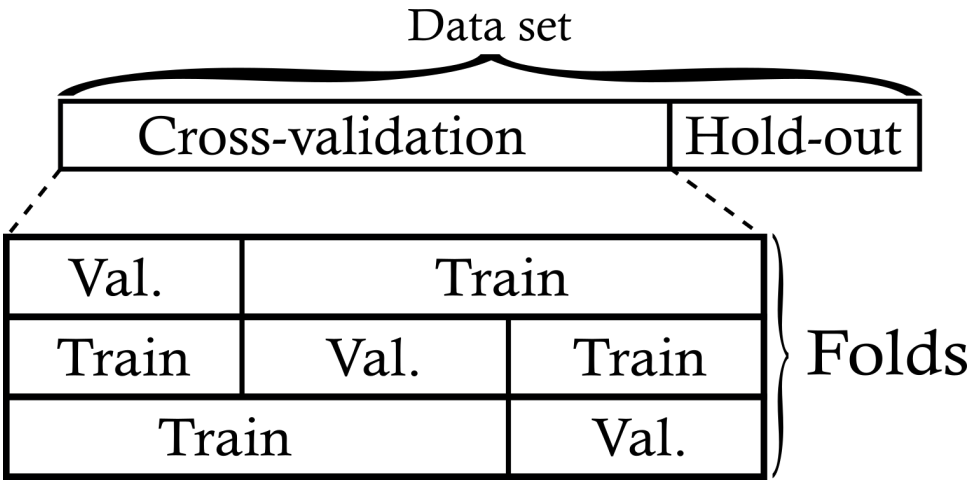


Figure 2.8: Cross-validation with a hold-out data set.

Table 2.2: The confusion matrix.

		Actual	
		True	False
Predicted	True	True positive (TP)	False positive (FP)
	False	False negative (FN)	True negative (TN)

is obviously a poor classifier as it would never correctly classify a single faulty machine. This is addressed by including other measurements that also emphasise the misclassified samples. Some popular measures that do this are the *F-score* and *Receiver Operating Characteristic Area Under the Curve (ROC AUC)*. They work by combining *sensitivity, specificity, and precision*.

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.11}$$

A useful tool to talk about these measures is the confusion matrix for a binary classifier that classifies samples as belonging to the class, true, or not belonging to the class, false. It is shown in table 2.2. The confusion matrix contains the number of samples that are: correctly classified as belonging to the class, true positive (TP); incorrectly classified as belonging to the class, false positive (FP); incorrectly classified as not belonging to the class, false negative (FN); and correctly classified as not belonging to the class, true negative (TN).

Sensitivity, shown in (2.12), is a measure of how well the model picks up on the class, essentially the probability that the class is detected. It is the number of correctly classified samples belonging to the class, divided by all occurrences of the class.

$$sensitivity = \frac{TP}{TP + FN} \tag{2.12}$$

Specificity, shown in (2.13), gives an impression of the model's capacity to correctly classify false samples. It is the number of true negatives divided by the total number of

actual false samples.

$$specificity = \frac{TN}{TN + FP} \tag{2.13}$$

Precision, shown in (2.14), is the ratio of true positives divided by the total number of samples classified as true. A high precision gives confidence that the classifier has made a correct prediction when it returns true.

$$precision = \frac{TP}{TP + FP} \tag{2.14}$$

Each of these has pit-falls when faced with unbalanced data sets and classifiers that classify all samples as either true or false. To balance the possible pitfalls, the F-score is especially good for unbalanced classes and the ROC AUC is a better metric for more balanced data sets. The F-score is defined as the harmonic mean of precision and sensitivity, it weighs the reliability of a classification together with its chance of detecting the class [23]. The F-score is shown in equation (2.15).

$$F_\beta - score = \frac{(\beta^2 + 1) \cdot precision \cdot sensitivity}{\beta^2 precision + sensitivity} \tag{2.15}$$

If $\beta = 1$, it is what is referred to as the F1-score or simply F1 as shown in (2.16) [23].

$$F_1 - score = 2 \cdot \frac{precision \cdot sensitivity}{precision + sensitivity} \tag{2.16}$$

The weighted F1 score calculates an F1 score for each class, faulty and healthy, multiplies each score with the prevalence of each class, adds them together, and divides the sum by the total number of samples. The weighted F1 function of two classes, a and b, of n_a and n_b samples each is shown in (2.17).

$$F_{1,weighted} = \frac{F_{1,a} \cdot n_a + F_{1,b} \cdot n_b}{n_a + n_b} \tag{2.17}$$

A classifier will often not return a 1 for true and a 0 for false, it will return some number in the interval between 0 and 1. How the sample is classified according to that is given by the threshold set. If the threshold is 0.5, any value above or equal to 0.5 will classify the sample as true and vice versa. The Reciever Operating Characteristic curve (ROC) is the sensitivity plotted against (1-specificity) for every threshold between 0 and 1, the ROC AUC is the total area under the ROC as shown in figure 2.9.

2.3.6 Ensemble learners

Ensemble learners are learners that combine several weak learners that may have poor performance to create a strong learner with good performance. There are a few methods of accomplishing this, mainly *bagging, boosting, and stacking.*

Bagging is short for bootstrap aggregating. It can be done with any learning algorithm, but is most common with decision tree algorithms. It is done by creating several

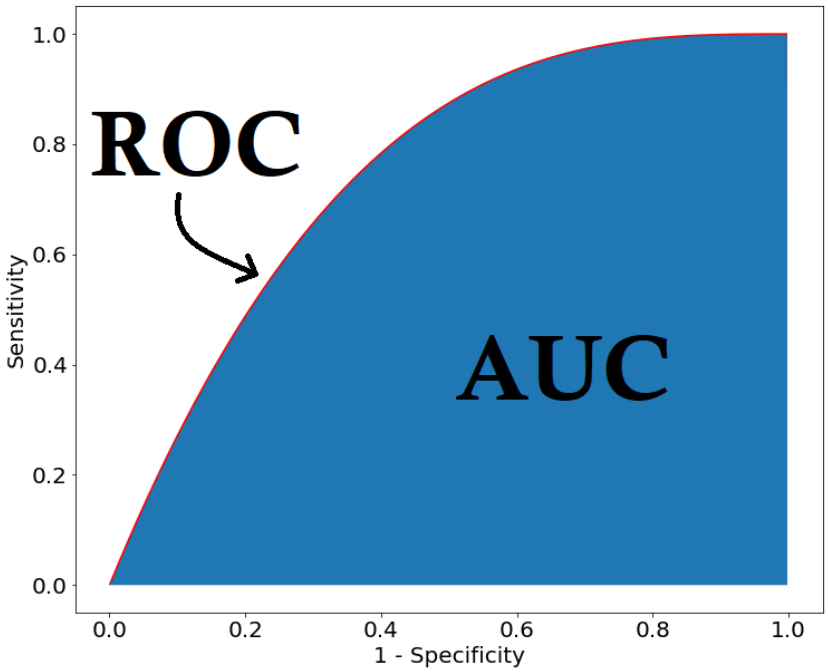


Figure 2.9: The ROC AUC is the area shaded blue.

bootstrap data sets from the training set and training a model using each bootstrapped data set. To create a bootstrapped data set of size N, N samples are drawn with replacement from the original data set. All the models are then combined and their classifications are aggregated. This means that a sample is classified by each model within the ensemble and the ensemble's classification is the mean or majority vote of its constituent models.

Boosting is similar to bagging in that it returns majority vote or mean predictions of several weak learners, but the way the models are generated is different. Where bagging uses a completely random process, boosting generates models consecutively to improve upon the predictions of the last model trained. This is done by first training a single model on the data set. The first model is placed into the ensemble that now consists of one model. The ensemble then makes predictions on the training data set. The samples that the ensemble classified poorly are given additional emphasis in the training of the next model to go into the ensemble, thereby improving the ensemble where it performs worst. This is continued until the ensemble has reached the desired size. The weighting of samples can be done by adding a weight to each sample denoting its importance in the cost function or by oversampling them into a bootstrapped set. This makes the ensemble focus on the hard-to-classify samples. Often the ensemble weights its constituent models according to their performance when aggregating the prediction.

Stacking is to train a meta-learner, which is a model that is trained to interpret the outputs of several other models, to make a prediction based on the predictions of several other learners. The learners that provide predictions to the meta-learner are termed base-learners. It usually outperforms the base-learners it is trained upon. Each of the

base-learners are first fitted to the training set, and their predictions upon the training set are used as the training set for the meta-learner. The base-learners can be any machine learning model that returns predictions. This provides a benefit in that by including different models as base-learners, the weaknesses of one model can be remedied by another.

2.3.7 Logistic regression

Logistic regression estimates the probability that a sample belongs to a class [17]. It does this by fitting a logistic function to samples in a two-class training set $X = \{\boldsymbol{x}_n, d_n\}_{n=1}^{N}$ of N samples. Each sample $\boldsymbol{x}_n = (x_n^1, ..., x_n^p)$ is a vector composed of p features with a class $d = 0, 1$. The logistic function is defined as in (2.18), and the output range of the logistic function is between 0 and 1 for all inputs.

$$p(\boldsymbol{x}_{N+1}) = \frac{e^{\beta_0+\beta_1\cdot x_{N+1}^1+...+\beta_p\cdot x_{N+1}^p}}{1+e^{\beta_0+\beta_1\cdot x_{N+1}^1+...+\beta_p\cdot x_{N+1}^p}} \tag{2.18}$$

To estimate the regression coefficients $\beta_0, \beta_1, ..., \beta_p$, the maximum likelihood method is generally used. This finds the most likely regression coefficients based on the training set. When an unknown sample $\boldsymbol{x}_{N+1}$ is introduced into the function, it returns a value on the interval of 0 to 1. This is the probability that the sample belongs to the class $d = 1$. Since a logistic function only approaches 0 and 1 asymptotically, a decision threshold θ is introduced. It is the threshold above which a sample is classified as belonging to class 1.

$$\begin{aligned} p(\boldsymbol{x}_{N+1}) > \theta &\implies d_{N+1} = 1 \\ p(\boldsymbol{x}_{N+1}) < \theta &\implies d_{N+1} = 0 \end{aligned} \tag{2.19}$$

By raising or lowering θ, the classifier can be made more or less conservative. In the case of fault detection in electric machines, the classifier may return the probability that there exists some fault condition in the machine. If there is a high cost associated with conducting maintenance on the equipment, a high θ could be justified as it will only classify the samples with a very high probability of having a fault as faulty. On the other hand, if there are dire consequences if a fault goes undetected, θ could be lowered.

2.3.8 K-nearest neighbours

K-Nearest Neighbours (KNN) is a supervised learning algorithm that compares a sample to a labelled data set to predict its class or value. It does this by calculating the distance from the sample to be classified to the samples in the training set. The class of the sample is then determined to be the most frequent class of its k nearest neighbours. When using KNN for regression, the value of the sample is set to the average of its k nearest neighbours. It is a non-generalising model since it is not in any real sense trained, it only compares samples to the training set. An illustration of KNN is showed in figure 2.10. The sample $\boldsymbol{x}$ is to be classified into one of the three classes in the training set with $k = 5$. Its distance to every sample in the training set is computed. Among the 5 nearest neighbours, there were 3 "O"'s, and 2 "-"'s. The sample $\boldsymbol{x}$ then classified as belonging to the category "O". The KNN suffers from noise as local topography can

disturb classification, this is seen in the illustration that the classification would change if $k = 3$. This is both a strength and weakness of the KNN as it works well with uneven class borders, but can misclassify samples easily due to outliers in the training set. To remediate this and simultaneously increase the speed of the algorithm, Condensed Nearest Neighbours (CNN) which selects prototypes from the data that best represent each class in the training set is put forward.

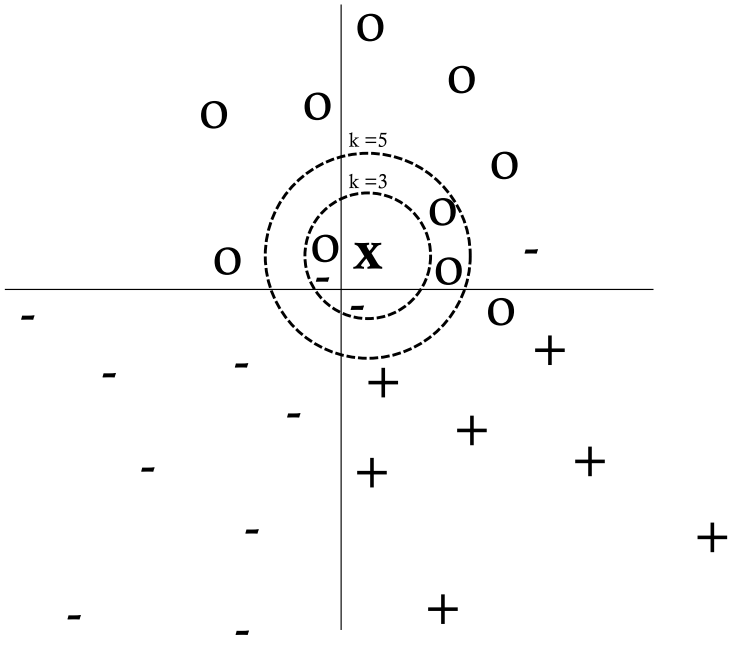


Figure 2.10: An illustration of KNN.

2.3.9 Support vector machine

The Support Vector Machine (SVM) is a supervised learning algorithm intended to classify samples by placing them into a Euclidean space subdivided by hyper-planes. Each sub-space corresponds to a specific class, and the sample is classified according to its position in the sample space. SVMs do this by finding an optimal hyper-plane in the data space that best separates between the different classes that maximises the margin between classes. Given a two-class sample set $X = \{\boldsymbol{x}_n, d_n\}_{n=1}^{N}$ of N linearly separable samples and classes $d = \pm 1$, we would like to find the hyper plane with maximum margin separating the two classes for the purpose of correctly classifying an unknown sample, $\boldsymbol{x}_{N+1}$, as shown in figure 2.11. The hyper-plane is defined by (2.20), the parameters $\boldsymbol{w}$ and b define it completely and are what the SVM seeks to optimise. It does this by finding the support vectors, $\boldsymbol{x}_{\pm}^{s}$, at the frontiers between the two classes that give the widest margins. New samples are classified by evaluating them in (2.21) and assigning them a class.

$$\boldsymbol{w}^T\boldsymbol{x} + b = 0 \tag{2.20}$$

$$\begin{aligned} f(\boldsymbol{x}_{N+1}) &= \boldsymbol{w}^T \boldsymbol{x}_{N+1} + b \\ f(\boldsymbol{x}_{N+1}) > 0 &\implies d_{N+1} = 1 \\ f(\boldsymbol{x}_{N+1}) < 0 &\implies d_{N+1} = -1 \end{aligned} \tag{2.21}$$

The machine described is so far a linear classifier and can only correctly classify linearly separable sets. To handle non-linearity, kernels are introduced. A kernel is a processing trick that "adds" a dimension by performing a non-linear operation on the samples. Examples of kernels are the polynomial and hyperbolic tangent kernels, shown in (2.22) and (2.23) respectively. The tuneable parameters in these kernels are p, β_0, and β_1. To save computation and memory, the kernels are used to evaluate the distance between the samples in the new dimension without actually mapping the samples into it.

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (\boldsymbol{x}_i^T \boldsymbol{x}_j + 1)^p \tag{2.22}$$

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = tanh(\beta_0 \boldsymbol{x}_i^T \boldsymbol{x}_j + \beta_0) \tag{2.23}$$

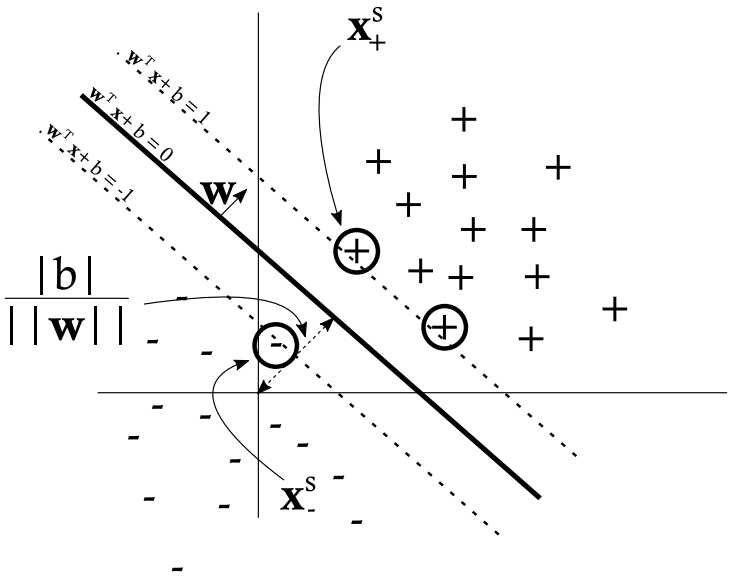


Figure 2.11: An illustration of an SVM distinguishing between two classes. The hyper-plane is the bold black line and the margins are illustrated by the dotted lines. The support vectors are circled.

2.3.10 Decision tree learning

A decision tree is a hierarchical structure composed of several nodes branching out from a root node and ending in leaves. In a binary decision tree, which is most common in machine learning applications, each node has two branches. Each node is an evaluation of some information into true or false, and a branch is followed according to the result of the evaluation. The new node is then evaluated, this continues until a leaf is reached. The leaves contain the decisions of the decision tree. The maximum number of decisions, i.e. the number of nodes in the longest branch, is the depth of the tree. A decision tree

is shown in figure 2.12. As is apparent in the decision tree shown, the features evaluated can be both continuous (temperature) and Boolean (presence of rain).

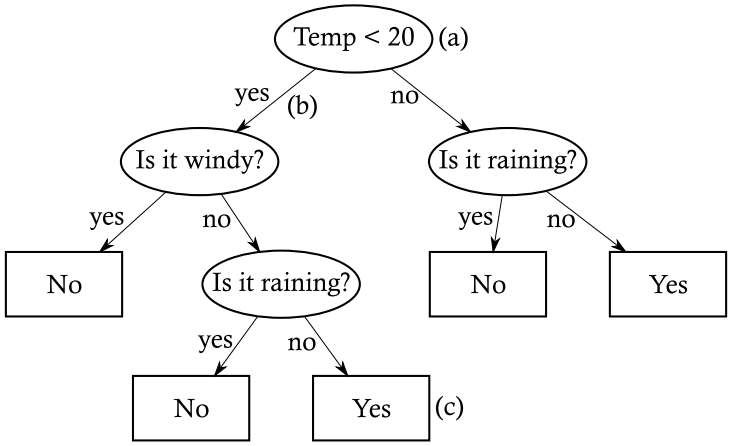


Figure 2.12: An illustration of a decision tree deciding if a person should go outside. Shown in the figure are the root node (a), branches (b), and leaves (c).

In decision tree learning, an optimal decision tree is fitted to the training data. This is done by evaluating the predictive power of each feature using an impurity measure and selecting the feature with lowest impurity as a node. There are several impurity measures (information gain, gain ratio, gini-index, variance reduction, etc.). If the decision tree is allowed to be very large, it can grow to make perfect predictions on the training data as there is effectively a branch for every sample. This creates a decision tree that generalises poorly as it struggles to classify new samples, it is over-fitted. To combat over-fitting random forest are introduced.

Random forest

A random tree is grown like a normal decision tree, only it uses bagging to combine several trees generated from bootstrapped data sets into a forest. To further improve the generalisation capability of the forest, each tree is trained on a randomly selected subset of the features. This creates a forest of full-size trees that is less prone to over-fitting. Each tree has an equal vote in the final classification. To further improve performance, boosting is introduced.

Random forests can also be used for feature selection. Each tree, t, in the random forest assigns a feature importance to each of the features in its feature subset. It does this assessing the misclassification rate on its out-of-bag (OOB) samples, OOB_t. These samples are the ones that were not included in the bootstrapped data set and hence has not been used to construct the tree. This is the baseline for the tree's performance, denoted by $errOOB_t$. It then takes the feature column of feature j, designated X^j, in the OOB samples and randomly permutes the values in the column. The misclassification rate is then assessed anew on this randomly permuted OOB sample set, OOB_t^j. The new misclassification rate is denoted $errOOB_t^j$. If $errOOB_t^j$ is unchanged after the feature was scrambled, feature j is deemed to be less important. If it deteriorates, on the other

hand, the feature is important. This is repeated for every feature, X_j, for every tree, t, in the forest of T trees. The feature importance of each feature, $VI(X^j)$, is the average importance across the entire forest as shown in (2.24) [24].

$$VI(X^j) = \frac{1}{T}\sum_{t=1}^{T}(errOOB_t^j - errOOB_t) \tag{2.24}$$

Boosted trees

Boosted decision tree models are very strong classifiers. The first practical boosting algorithm, AdaBoost, was created in 1995 and is still useful today. AdaBoost, short for adaptive boosting, is a forest of stumps [25]. A stump is a tree with just a root node and two leaves. The stumps are made sequentially and weighted according to their predictive performance. The samples are also given a weight, initially equal, that is increased if the samples are misclassified by the last stump that was generated. The weights of the stumps are calculated as given in (2.25), where the total error is the sum of the weights of the misclassified samples.

$$StumpWeight = \frac{1}{2}\cdot log\left(\frac{1 - TotalError}{TotalError}\right) \tag{2.25}$$

The weights of each sample are then adjusted as shown in (2.26). The scalar a is either 1 or -1 depending on if the sample was misclassified or correctly classified by the stump, respectively. When every sample has been adjusted, all the sample weights are normalised.

$$NewSampleWeight = OldSampleWeight \cdot e^{a \cdot StumpWeight} \tag{2.26}$$

A bootstrapped data set of equal size to the original is made, where the probability that each sample is drawn is proportional to its weight. The weights in the new data set are set to be equal. This new data set is used to train a new stump. The hard-to-classify samples are then given extra emphasis by being more numerous in the data set. The process is continued until the forest is complete. AdaBoost has now been superseded by modern alternatives as boosted trees outperform boosted stumps [26]. Popular modern boosting algorithms are XGBoost, CatBoost, and LightGBM. They use what is known as gradient boosting.

Gradient boost decision trees

XGBoost is extremely popular and does very well in Kaggle competitions as over half the winning implementations in 2015 made use of XGBoost [27] [28]. Kaggle is an international on-line machine learning platform where machine learning experts compete to implement the best machine learning algorithms with a given data set. XGBoost provides the option to increase tree depth and is heavily optimised by use of parallel processing. It is a quick and powerful algorithm [28]. CatBoost and LightGBM are still considered frontier algorithms, but are set to outperform XGBoost in some respects. LightGBM is an extremely optimised boosting algorithm that is especially suited to extremely large

data sets as it can achieve a 20 times speed increase with nearly the same accuracy [29]. CatBoost is an algorithm that seeks to reduce target leakage and therefore increase generalisation in the models, it was shown that CatBoost could outperform XGBoost and LightGBM on several popular machine learning tasks [28].

2.3.11 Artificial neural network

An Artificial Neural Network (ANN) is an approach to machine learning inspired by biology. It is composed of individual *neurons* that are organised into *layers*. Each layer connects to the next, from the input to the output layer. A neuron receives several inputs, x_k, that are weighted with weights, w_k, and one fixed input, x_0. The fixed input is weighted by the weight, b, called the bias. The sum, v, of contributions of the weighted inputs and the bias are passed to the activation function, ϕ. The result is the output $y = \phi(v)$. A neuron is shown in figure 2.13. The number of layers, the number of neurons in each layer and the activation function are the hyperparameters that are selected by the researcher before the learning algorithm is initiated. The weights and biases of each neuron are fitted in the training process.

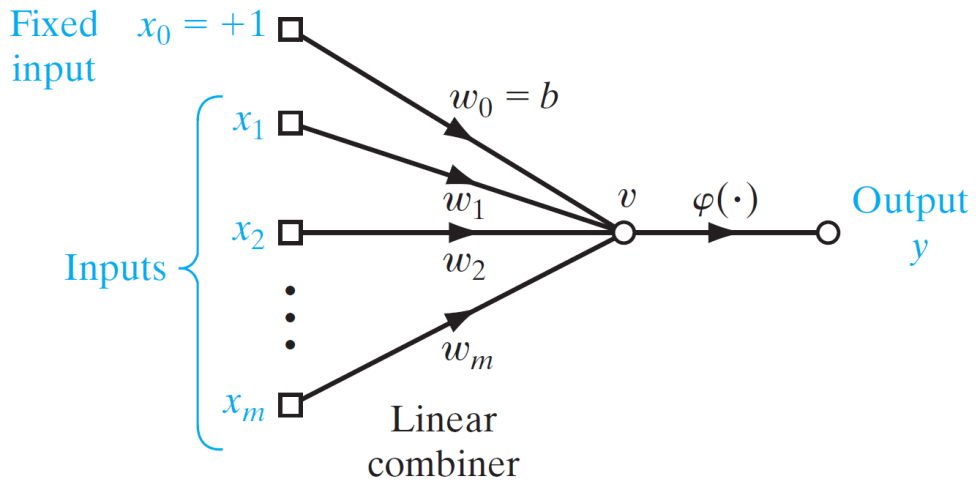


Figure 2.13: An artificial neuron.

Perceptron

The perceptron, the simplest ANN model, was first proposed in by Rosenblatt in 1958 [30]. It is a single layer ANN and so only contains the output layer. A single layer perceptron is shown in figure 2.14. It could classify some problems, but it was shown that it is only capable of correctly classifying linearly separable classes.

Multi-layer perceptron

A Multi-Layer Perceptron (MLP) is able to classify nonlinear problems. It introduced greater complexity by adding one or more *hidden layers* between the input and output. This results in a universal approximator, meaning it can approximate any continuous

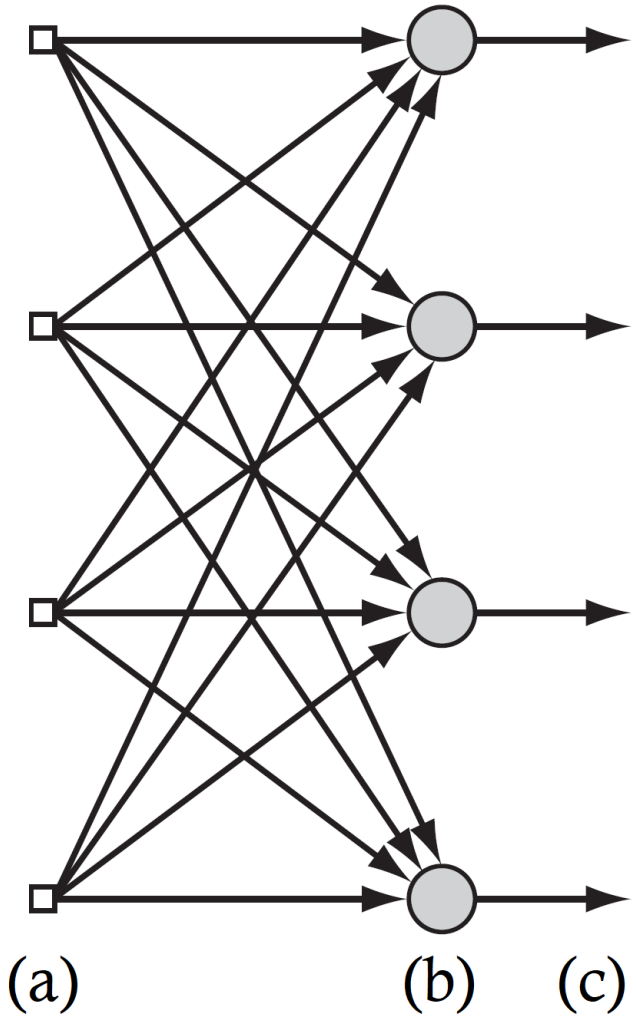


Figure 2.14: A single layer perceptron consisting of inputs (a), neurons (b), and outputs (c).

function. While the output layer is constrained by the problem, the hidden layers can have as many or few neurons per layer as the problem requires. Figure 2.15 shows a 3-layer MLP.

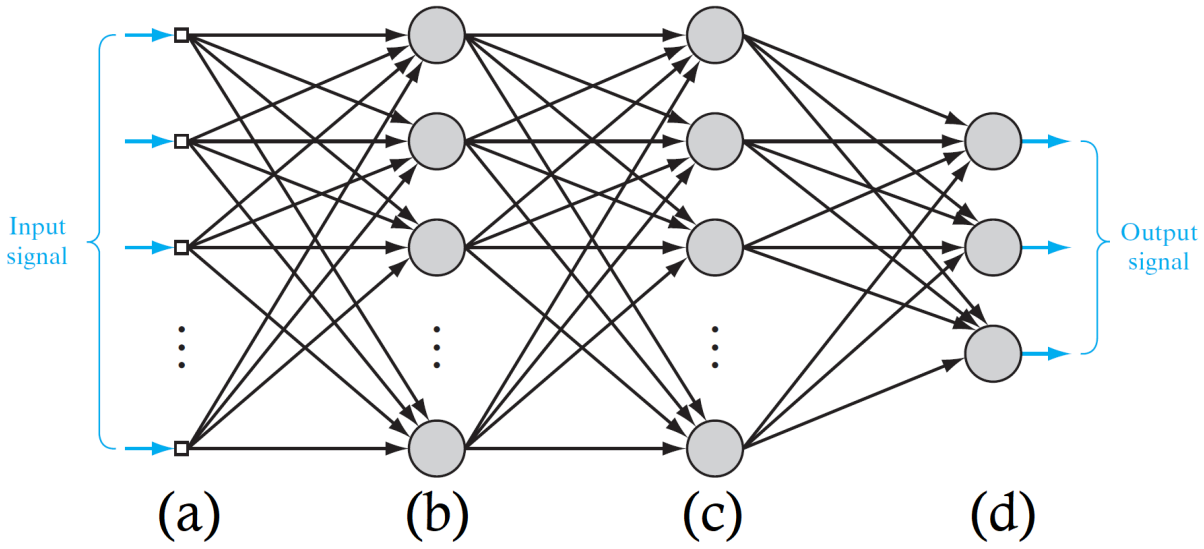


Figure 2.15: A fully connected 3-layer perceptron consisting of inputs (a), the first hidden layer (b), the second hidden layer (c), and outputs (d).

Activation function

The activation function used in the neuron impacts the performance of the model and the computational load of training. Some examples are the Heaviside (2.27), ReLu (2.28) and hyperbolic tangent (2.29) activation functions.

$$\phi(v) = max\left(0, \frac{v}{|v|}\right) \tag{2.27}$$

$$\phi(v) = max(0, v) \tag{2.28}$$

$$\phi(v) = tanh(v) \tag{2.29}$$

The utility of an activation function is that it adds non-linearity to aid in decision making and it is necessary that the activation function is at least piece-wise differentiable to be able to train the model by error back-propagation. The most popular activation function is the ReLU as it adds non-linearity and eases the numerical burden imposed by continuous functions such as the hyperbolic tangent.

Training

MLPs are usually trained using error back-propagation. An output, o_j, is dependent on neurons in every previous layer and the input as shown in (2.30). L is the number of

layers in the MLP and w_{ji} is the weight on the connection from neuron i to neuron j.

$$o_j = y_j^{(L)} = \phi_j\left(\sum_i w_{ji}\phi_i\left(\sum_k w_{ik}\phi_k\left(...\phi_r\left(\sum_m w_{rm}x_m\right)\right)\right)\right) \tag{2.30}$$

The error, e_j, of the output compared to the target value, d_j, is given as (2.31).

$$e_j = d_j - y_j^{(L)} \tag{2.31}$$

The error energy, $\mathcal{E}$, summed over the entire output is given by (2.32).

$$\mathcal{E} = \frac{1}{2}\sum_j e_j^2 \tag{2.32}$$

Since $\mathcal{E}$ is a linear combination of differentiable functions, there exists for every weight w_{ji} a derivative of $\mathcal{E}$. The weights $w_{ji}[k]$ of epoch k can then be adjusted for epoch $k+1$ by way of gradient descent as shown in (2.33). The learning rate, η, is set for each layer and usually decreases nearer to the output. This continues until $\mathcal{E}[k]$ approaches a constant value.

$$w_{ji}[k+1] = w_{ji}[k] - \eta\frac{\delta\mathcal{E}[k]}{\delta w_{ji}[k]} \tag{2.33}$$

Other artificial neural networks

There are many other ANNs, some examples are convolutional neural networks that are suited to classify images, auto-encoders that can be used for compression, anomaly detection and generative models, and radial base function network that substitute the weights of ANNs for vector coordinates.

Chapter 3

Method and results

In this chapter the procedure of data acquisition, data management, feature extraction, and constructing the classifier is described along with its results. Since application of machine learning is a somewhat exploratory process, the considerations that justify choices made throughout the process are included in amongst the method and results.

3.1 Laboratory measurements

The data set is composed of two concurrent Hall-effect sensor readings taken of a salient pole synchronous generator running at synchronous speed with several different ITSC-fault severities induced. The machine, the sensors attached and the measurements are described in this section.

The measurements were taken as a part of the master's thesis work of Ingrid Linnea Groth. It was initially planned to conduct original experiments with more sensors and greater variation of operating conditions, but this proved impossible due to COVID-19-related restrictions on laboratory work. The experiments from which the data was gathered are only briefly recapitulated here for ease of reading, the reader is referred to I. L. Groth's master's thesis for a more detailed description [4].

The experimental generator

The machine, *Brutus*, is a 100 kVA experimental generator with 14 salient poles, constructed to resemble generators commonly situated in Norwegian hydroelectric power plants. It is shown in Figure 3.1.

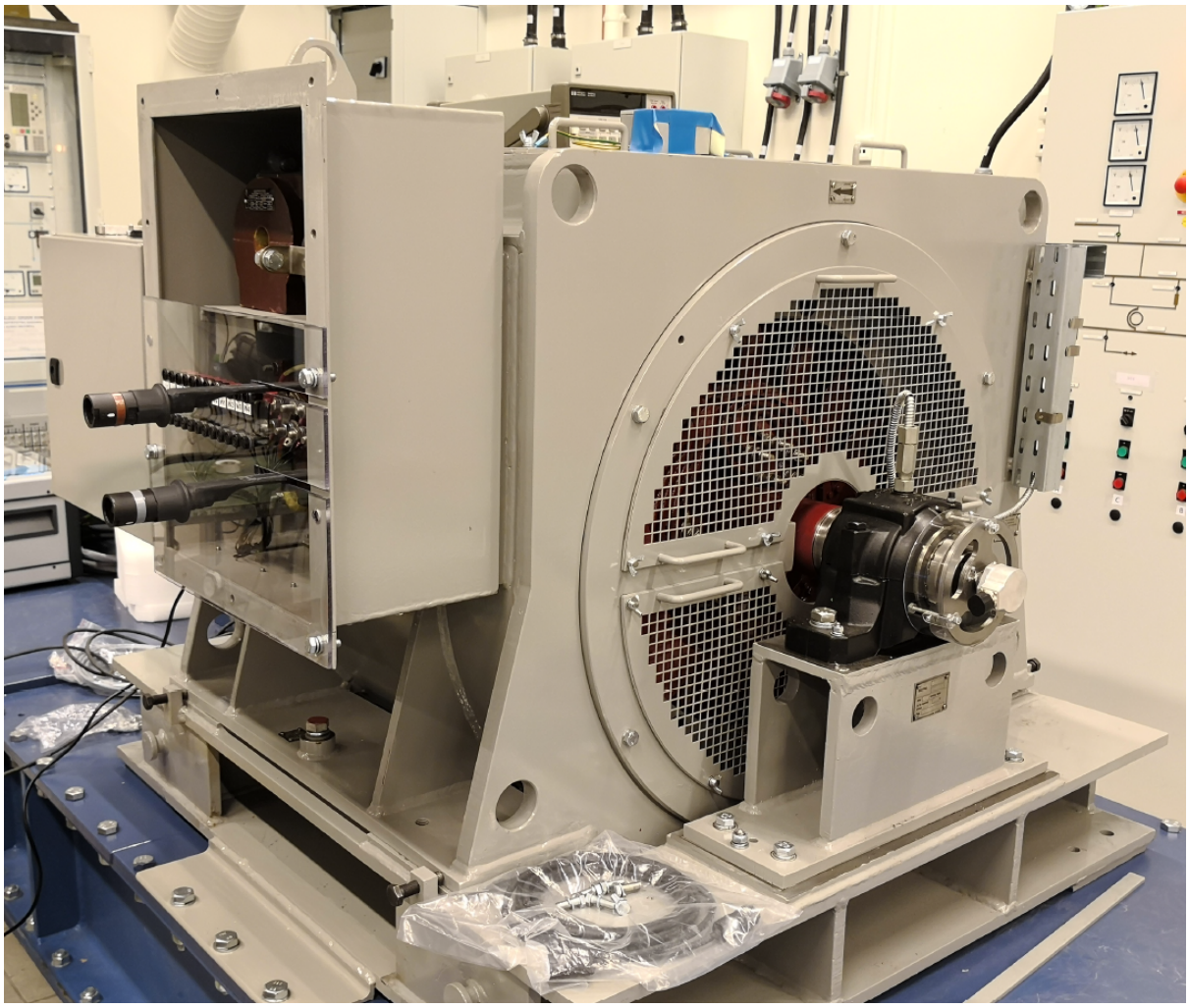


Figure 3.1: Brutus, the laboratory generator in the NTNU Smart Grid laboratory [4].

Its nameplate values are given in Table 3.1 and some defining features of its topography are given in Table 3.2.

Table 3.1: The rated values of Brutus. [4]

Nameplate values	
Nominal power	100 kVA
Nominal voltage	400 V
Nominal current	144.3 A
Nominal speed	428 rpm
Nominal frequency	50 Hz
Nominal power factor	0.90
Nominal exc. current	103 A
No-load exc. current	53.2 A
X_d	2.040 Ω
X_q	2.075 Ω

Table 3.2: The topography of Brutus. [4]

Topography	
Number of poles	14
Number of slots	114
Damper bars per pole	7
Winding connection	Wye
Winding layout	Double-layer
Turns per field winding pole	35
Outer stator diameter	780 mm
Outer rotor diameter	646.5 mm
Nominal air gap length	1.75 mm

The generator is purpose-made to be run with any of several incipient faults induced. The faults that are possible to induce are:

- Inter-turn short-circuits by short-circuiting 1, 2, 3, 7 or 10 turns in the field windings of two opposing poles.
- Broken damper bars by removing damper bars from their slots.
- Static eccentricity by offsetting the stator frame from the centre of the rotor's rotational centre.

Several load conditions can be tested by applying resistive and inductive loads in parallel or series. This enables the generator to be run in several distinct load conditions with several types of faults or combinations of faults. The rotor is shown in Figure 3.2.

Experiments

The generator was driven through a gear box by a 90 kW, 400 V induction motor with four poles and rated speed of 1482 rpm supplied by a three-phase 60 kVA laboratory converter. The prime mover, gear box and Brutus are shown in Figure 3.3. The speed of the induction motor during all tests was set so that the frequency of the generator's electrical output was 50.004 Hz, nearly exactly 50 Hz.

Two Hall-effect sensors were placed into the air gap, glued onto stator teeth, at diametrically opposing ends of the stator. The sensor wires were shielded and the sampling was done using a 16-bit Tektronix MSO 3014 oscilloscope with 10 and 50 kHz sampling rate. Using this set-up, the generator was run in one of two different load conditions with

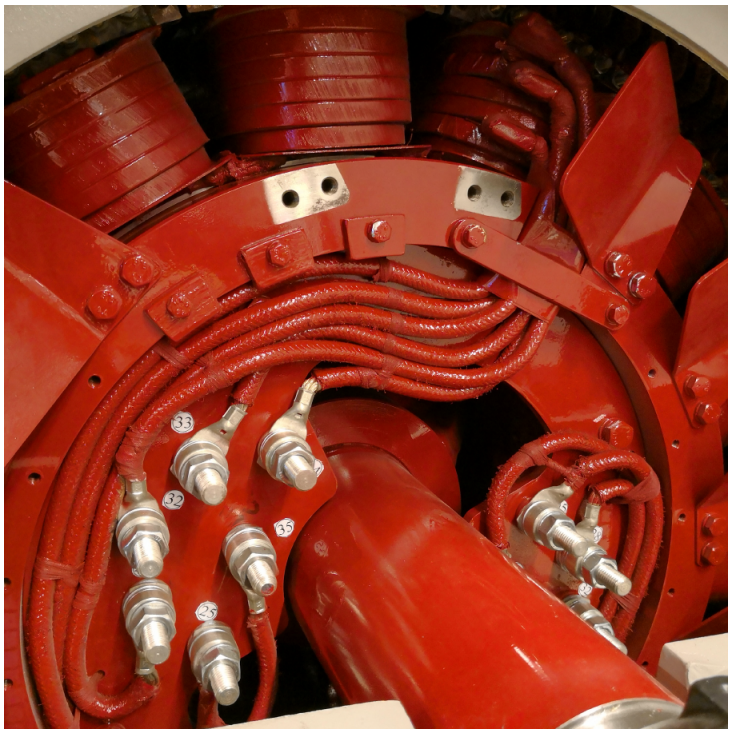


Figure 3.2: The rotor of Brutus. Terminals used to short-circuit rotor winding turns are visible exposed on either side of the shaft. [4]

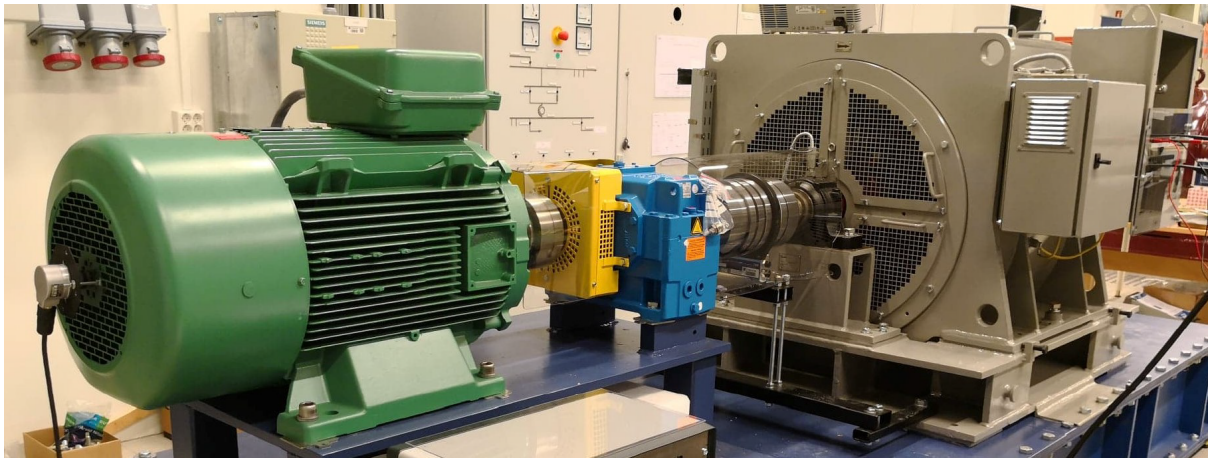


Figure 3.3: Brutus connected to the induction motor through a gear box. [4]

differing degrees of inter-turn short-circuits applied. The conditions termed full and no-load are shown in Table 3.3. Please note that the full-load condition is not the maximum load of the generator, but rather the maximum power that the laboratory converter was able to provide.

	No-load	**Full-load**
Output power	None	65 kVA
Power factor	None	0.93
Exc. current	56 A	84 A
Exc. voltage	10.5 V	14.7 V

Table 3.3: Full and no-load test conditions. [4]

The measurements series available are summarised in Table A.1 in Appendix A. Each measurement series is 100 000 samples long, irrespective of sampling rate. If each sensor is considered as an independent measurement series of the same machine state and the two sampling rates of the same machine state are equal, a total of 48 measurement series were available from the 24 experiments performed. Each one of these 2 and 10 second measurement series encompass many mechanical periods of the machine, a prerequisite to be certain that any fault is captured in the measurement series.

3.2 Data pre-processing

The data was processed to appear similar to something one would sample in a production environment. The code written for loading the data into Python can be seen in Appendix B.1.

In a production deployment of the fault detection system, the measurement series would need to be windowed with the classification run on a sliding window on the last electrical periods to be able to detect faults in near real-time. Since incipient faults are not critical, a long window length of several mechanical periods is possible. The minimum viable window length is 1 mechanical period, as this is the window length necessary to ensure that any fault will pass the sensors. An excessively long window length is prohibitive since it will add little new information and slow down feature extraction. However, the window length should be long enough to remediate end effects in signal processing tools that suffer from them. End effects can be alleviated by analysing a concatenated series if the signal is assumed to be periodic. Since the machine has 7 pole pairs, 7 electrical periods will capture 1 complete mechanical period.

The measurement series before windowing will be called original sample series (OSS), while the measurement series after windowing will be called reduced sample series (RSS). Each RSS will have features extracted from it to be used as a sample in the data set used to train the classifiers. To reduce non-fault-related variations in the RSS, the OSS are cut at rising zero-crossing to have integer electrical periods in each RSS.

Due to the scarcity of data, it is desirable to create several RSS from each OSS. The system is stationary in its steady state and RSS cut from the same OSS will thus be similar. By skipping one electrical period after each captured RSS length of integer mechanical periods, the faults will pass each sensor an electrical period earlier in each contiguous RSS. This is done to provide training samples with faults in every position possible as shown in Figure 3.4. An RSS length of 7 electrical periods, or 1 mechanical period, was chosen to produce the maximum number of RSS possible.

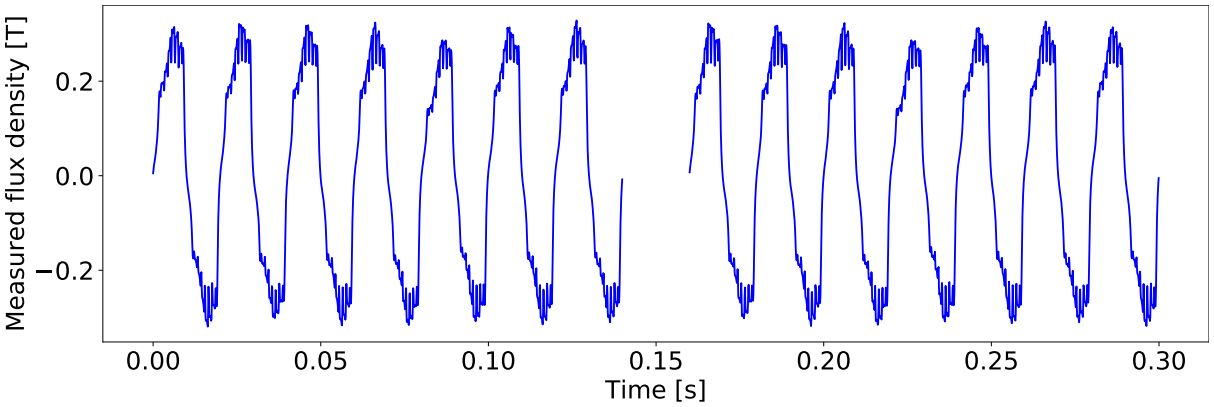


Figure 3.4: Two consecutive RSS cut from the same OSS. They are each of 7 electrical periods, with a 1 electrical period between the two. Note the smaller negative peak occurring in periods 4 and 3 occurring of the first and second RSS respectively. The one period shift between each RSS makes the fault indication appear one position earlier.

A visual inspection of all of the OSS showed continuous measurement series without sudden jumps in values that would have indicated sensor malfunction or data corruption. Furthermore, there were no missing values in the data sets. The data quality is excellent. The data set is, however, slightly imbalanced, with 65.9 % of samples of faulty case. In total 24 experiments were conducted, each of which sampled with two sensors and two sampling frequencies. The samples will later be split into train/test sets according to their OSS. Measurements taken of the same machine state with the same sensor are assigned to the same OSS, irrespective of their sampling frequency. If measurements are from different machine states or different sensors, they are assigned to different OSS. This is done because the OSS that the data belongs to will be used later when splitting the data set into training/validation/hold-out data sets. The code written to partition OSS into RSS is shown in Appendix B.2.

3.3 Feature extraction

Raw time series are very sensitive to small perturbations and thus not suited to be used directly as tabular training data. Features are therefore extracted from each RSS. These features are then used as a basis for feature selection and, finally, as training data. The feature extraction methods used were fast Fourier transform, discrete wavelet transform energies and TSFRESH feature extraction. In total 475 distinct features were extracted. The code written for the feature extraction process can be seen in Appendix B.3.

3.3.1 Fast Fourier transform

The frequency content of each RSS was extracted by FFT. Previous comparisons done in the specialisation project of FFTs of healthy and faulty signals showed that the faulty signal had a marked increase in harmonic frequency components at intervals of $f_m = \frac{50}{7}$ Hz, the mechanical frequency of the generator, outside of the odd multiples of fundamental frequency compared to the healthy case [5]. The frequency components of integer multiples of f_m up to 500 Hz was extracted as features, see (3.1). The FFT implementation is shown in Appendix B.3.1.

$$f_{k,extracted} = k \cdot f_m = k \cdot \frac{2f_{sync}}{p}, k = 0, 1, 3, ... \tag{3.1}$$

3.3.2 DWT wavelet energies

Wavelet energies were good indicators of inter-turn short-circuits in the specialisation project and were decided to be included as features [5]. An algorithm was written to automatically extract and store the Haar wavelet energies of an indefinite number of RSSes. A 12-level-decomposition, Haar wavelet DWT was taken of each RSS and instantaneous, Teager, hierarchical, and relative wavelet energies were computed for each decomposition level. The implementation is shown in Appendix B.3.2.

An issue with DWT is its end effects, which are worsened substantially in each decomposition level as the length of the data series that is transformed is effectively halved in each decomposition level with the Haar wavelet. The adverse effects are diminished as the length of the data series increases since the portions affected by end effects are proportionally smaller. Therefore, each RSS was concatenated to 4 times its length before the discrete wavelet transform was taken. This exploits the assumption that the generator behaviour is stationary and relieves the issue of end effects. In addition, every 10 kHz RSS was upsampled to 50 kHz before running the DWT. Upsampling ensures that each DWT decomposition level contains the same frequencies for every RSS, even if it was originally sampled with different sampling rates. The frequencies within each of the 12 decomposition levels are shown in Table 3.4.

Table 3.4: The frequencies contained within each decomposition level of a 12 level DWT.

Level	Frequencies [Hz]
A12	0 - 6
D12	6 - 12
D11	12 - 24
D10	24 - 48
D9	48 - 97
D8	97 - 195
D7	195 - 390
D6	390 - 781
D5	781 - 1562
D4	1562 - 3125
D3	3125 - 6250
D2	6250 - 12500
D1	12500 - 25000

3.3.3 TSFRESH

A comprehensive feature extraction was done using TSFRESH. Every feature as detailed in Appendix C was extracted, except for the FFT features. TSFRESH's FFT features were not included because FFTs with informative frequency bins were already computed as detailed above and TSFRESH did not offer the ability to select frequencies of interest. Since many of the features generated by TSFRESH are sensitive to the length of the sample series analysed, the 50 kHz measurement series were downsampled by a factor of 5 to 10 kHz before feature extraction. This also saved on computation time, reducing it to 4.7 seconds to generate all the features for a single RSS. The implementation is shown in Appendix B.3.3.

3.4 Exploratory data analysis

Features' correlation to the target value and their variance are indicators of how useful they may be to make classifications. In addition, it is likely that some features are redundant if the features are strongly correlated with each other. These are some of the things one looks for in an Exploratory Data Analysis (EDA) as performed in this section. Before the EDA, the output from the feature extraction in Section 3.3 was formatted using the implementation shown in Appendix B.4. The EDA itself was performed using the implementation shown in Appendix B.5. Any invariant features were removed before the EDA, reducing the number of features from 475 to 417.

A qualitative inspection of the features show that most features have a mean close to zero, with some features' mean far in excess of this. The same is true for the standard deviation, some features have much greater variability than the norm. Plots of the features' means and standard deviations can be seen in Figures 3.5 and 3.6. This indicates that the features need to be standardised to work with some methods. Standardisation is a requirement for many techniques and learners, among them PCA, KNN, and SVM with radial bias [17]. Exactly which specific features' mean and standard deviation deviate from the rest is not of interest in this regard, since the existence of any in the set necessitates standardisation.

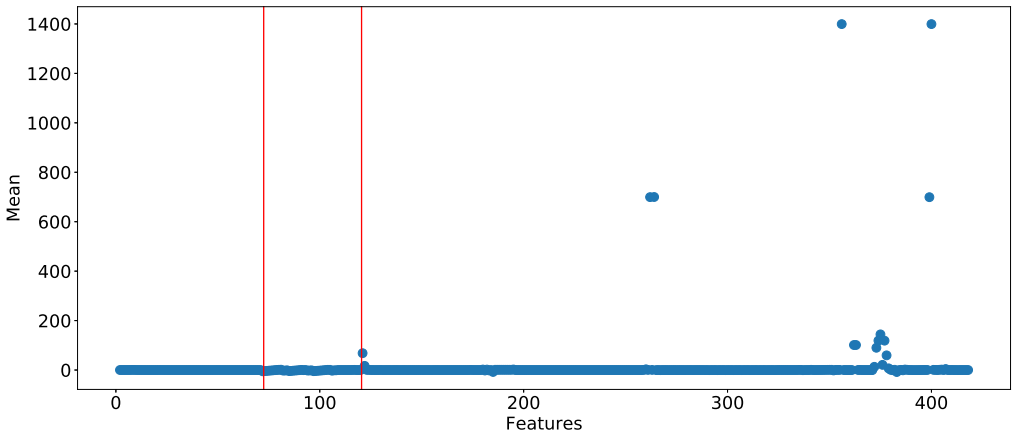


Figure 3.5: Calculated mean values across all samples for each feature. A few features have far larger means than the others. The plot is divided by red lines into three portions. The first portion from the left is the FFT-derived features, the second from the left is the DWT energy feature portion and the last is the collection of TSFRESH generated features.

The Pearson correlation of each feature to target values, that is how correlated the feature is to the number of ITSCs applied to the poles, showed largely uncorrelated features with a few exceptions. DWT energy features had many correlated features, and some TSFRESH generated features were strongly correlated. The FFT-generated features were largely uncorrelated. An overview of features' correlation to number of ITSCs is shown in Figure 3.7. Both negative and positive correlations are useful for classification, but correlation only shows *linear* relationships. There may be nonlinear relationships that

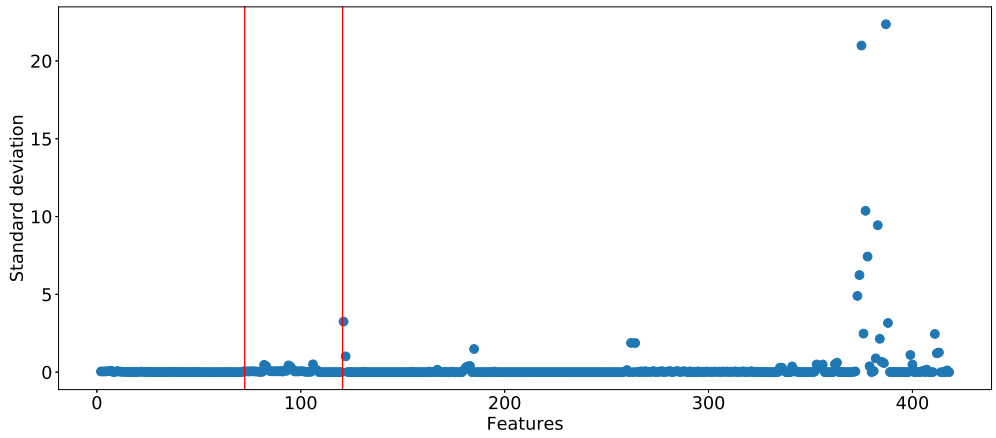


Figure 3.6: Standard deviation across all samples for each feature. The plot is divided by red lines into three portions. The first portion from the left is the FFT-derived features, the second from the left is the DWT energy feature portion and the last is the collection of TSFRESH generated features.

are not revealed by this test. The features with an absolute Pearson's correlation above 0.3 are shown in Table 3.5. This result is in agreement with the conclusions reached in [5], where an increase in DWT energy was a strong indication of ITSCs.

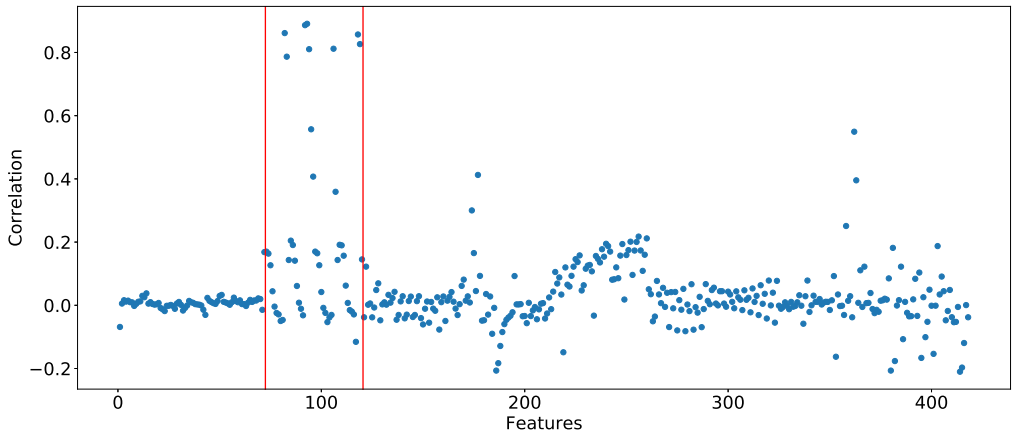


Figure 3.7: An overview of feature correlations. The plot is divided by red lines into three portions. The first portion from the left is the FFT-derived features, the second from the left is the DWT energy feature portion and the last is the collection of TSFRESH generated features.

The correlations above indicates that there are several *relevant* features. However, many of these may be *redundant* if these relevant features are strongly correlated amongst themselves. An effective visual method to investigate this is to construct a correlation matrix, in which each feature's correlation to every other is shown. Figure 3.8 shows the correlation matrix of the features. The centre diagonal line is the features' correlation to themselves, which is necessarily 1. On either side of the diagonal are mirror images of the

Table 3.5: The features most correlated to number of ITSCs.

Feature	**Correlation**
TWE, decomposition level 9	0.890734
TWE, decomposition level 8	0.886363
IWE, decomposition level 10	0.861306
RWE, decomposition level 10	0.856935
RWE, decomposition level 11	0.826443
HWE, decomposition level 10	0.811841
TWE, decomposition level 10	0.810356
IWE, decomposition level 11	0.786659
TWE, decomposition level 11	0.556887
TSFRESH, longest strike above mean	0.549258
TSFRESH, Approximate entropy, (m=2, r=0.7)	0.412482
TWE, decomposition level 12	0.407188
TSFRESH, Longest strike below mean	0.395404
HWE, decomposition level 11	0.359168
TSFRESH, Approximate entropy, (m=2, r=0.1)	0.300201

inter-feature correlations. Here we see that both FFT and DWT features are strongly correlated amongst themselves, while TSFRESH exhibits this to a lesser degree. With this large a correlation between the features, investigating feature selection and reduction methods is merited.

Since there is a high degree of correlation between the features, a Principal Component Analysis (PCA) would give an indication of how variable the samples are. Fewer principal components necessary to capture a given percentage of original variance indicates that the data set contains many features of low variance or high inter-feature correlation. A PCA of the data set was made to to span 95 % of the variance within the data set, resulting in 31 principal components . Of the 31 principal components, 85 % of the variance was contained within the 10 first components. This indicates that many of the features are uninformative or strongly correlated to each other, coinciding with the results from the correlation analysis.

High-dimensional data sets are unsuited to plot directly. To visualise the data set and gain some intuition about its distribution, the high-variance principal components of the PCA can be plotted. A plot of the data set along the two first principal components is shown in Figure 3.9. The plot shows 16 distinct clusters, 24 if healthy and faulty clusters are counted separately, where faulty and healthy sample distributions overlap in most. There does not appear to exist any clear decision boundary along which faulty can be discriminated from healthy. A consequence of this may be that SVM and KNN classifiers perform poorly on the data set.

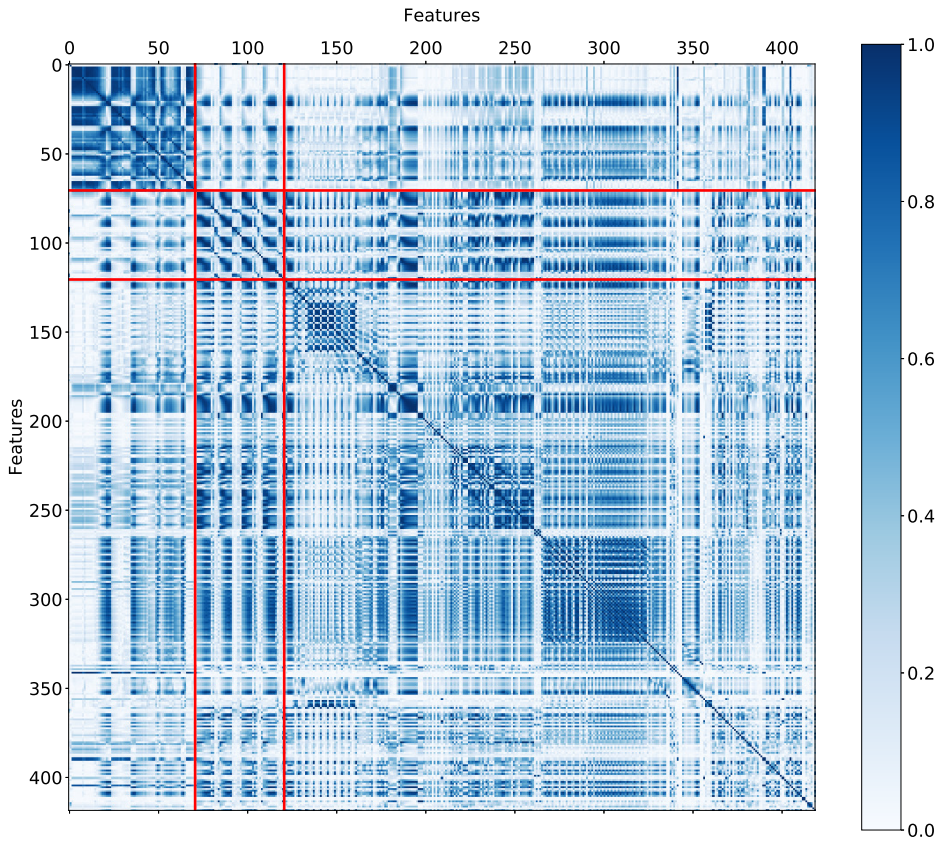


Figure 3.8: The feature correlation matrix. Darker colour indicates a higher correlation between the features. The red lines separate FFT features (left-/top), wavelet energy features (middle), and TSFRESH features (right/bottom).

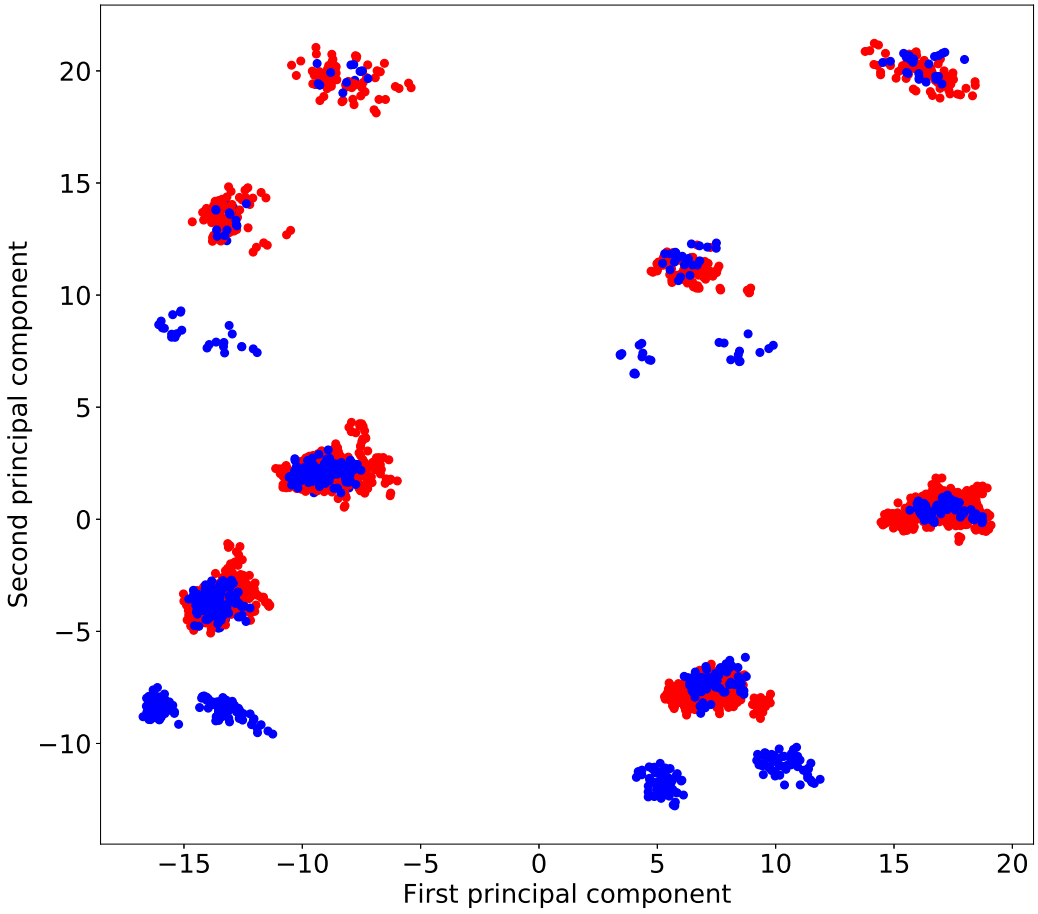


Figure 3.9: Samples plotted along the first and second principal component. Each point represents one sample, with red samples representing faulty machine condition samples and blue samples representing healthy machine condition samples.

3.5 Feature selection

Two feature selection methods, random forest feature selection and TSFRESH, were applied to the feature data set. Please note that before any feature selection was undertaken, a hold-out data set was extracted from it. This was to prevent any target leakage that could result from selecting features based on the entire data set and inadvertently providing the classifiers with features selected for their hold-out data set relevance. Since samples originating from the same OSSes are so similar, the split was also done in a way so as to ensure that no samples originating from the same OSS were split among the hold-out and remainder data sets. The hold-out set contains 15 % of the total samples, and will be used to assess the performance of the final classifier. The feature selection implementation can be seen in Appendix B.6.

3.5.1 Random forest feature selection

The random forest feature selection was done using a forest of 1000 decision tree estimators, which was trained on the training set using Gini impurity as the splitting criterion. During training, every feature is assigned an importance based on its impurity. All features of greater than mean importance were selected, the remainder discarded. This resulted in a feature reduction from 417 to 81 features.

3.5.2 Time series feature extraction based on scalable hypothesis tests (TSFRESH)

Using the feature extraction module included in TSFRESH, a subset of features deemed relevant was extracted. Taking into account the correlations discovered during the EDA, TSFRESH was configured to assume dependent features. False discovery rates in the interval 0.001, 0.01, 0.05, and 0.1 were tried, this resulted in a similar amount of features. The false discovery rate settled upon was 0.05, the rate used in [31]. This resulted in a feature reduction from 417 to 301 features.

3.5.3 Summary

The three versions of the feature data set, hereby termed feature data sets A, B and C, are summarised in Table 3.6. By comparing the performance of classifiers trained upon the different collections of features, some insight into which features are most useful for classifying the fault can be gleaned and which feature selection algorithms are most useful with this data. In a final version of the fault detection system, this knowledge could be used to selectively compute only the most useful features.

Table 3.6: The three data sets taken into machine learning.

Set	Selection method	Num. features
A	None	417
B	Random forest	81
C	TSFRESH	301

3.6 Fault detection

The following section details the development of a classifier intended to detect the presence of ITSCs using the data sets previously created. With the exception of XGBoost, all classifiers were implemented from Scikit-Learn version 0.23.1. The implementation can be seen in Appendix B.7.

This is done in four phases:

1. Selection of the feature data set
2. Hyperparameter optimisation of single machine learning models
3. Evaluation of stacking classifiers
4. Final classifier selection and evaluation on hold-out data set

The first objective, selection of the feature data set, was accomplished by evaluating the results of training a host of different classifiers on each data set. The classifiers chosen were:

- Logistic Regression
- Logistic Regression with PCA
- KNN
- KNN with PCA
- Radial basis function SVM
- Radial basis function SVM with PCA
- Linear SVM
- Linear SVM with PCA
- XGBoost
- Multi-Layer Perceptron
- Stacking classifier

By implementing logistic regression, SVM and KNN with and without a Principal Component Analysis, the effectiveness of PCA in this application can be gauged as well. PCA was not combined with XGBoost because PCA reduces the interpretability of the model, a key strength of decision trees. The stacking classifier combines the other models, with the exception of KNN and KNN with PCA. The KNN models were excluded because of their long prediction time and poor performance relative to the other classifiers. The architecture of the Stacking classifier is shown in Figure 3.10.

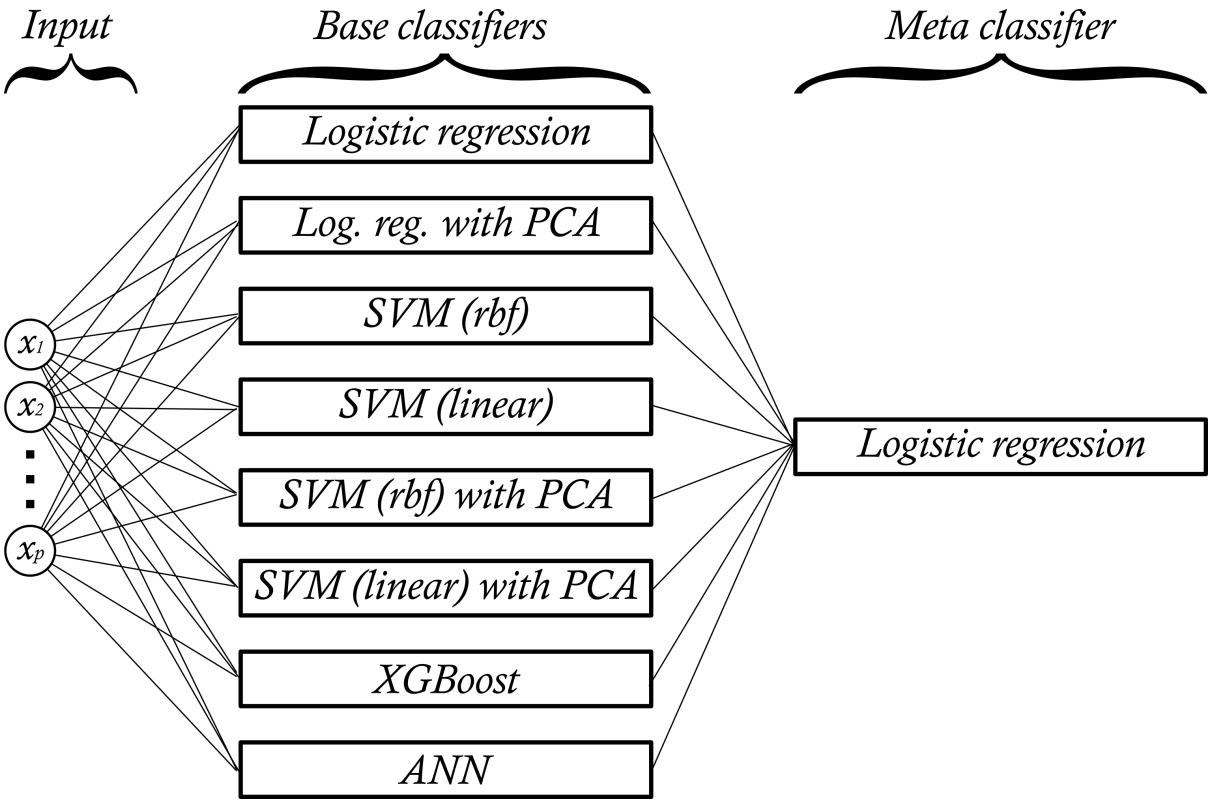


Figure 3.10: The stacking classifier as implemented. It combines the outputs of all the base classifiers via a logistic regression model to make the final classification.

Initial hyperparameter choices

The hyperparameter settings of each classifier are shown in Table 3.7. The classifiers not shown in Table 3.7 use default parameters. None of these hyperparameters have been attempted to be optimised, only given reasonable rule-of-thumb values. Note that the PCA was identically executed in all four applications with the same setting as used in the EDA.

Metrics

The metrics selected are sensitivity and precision. These metrics were chosen due to their lower susceptibility to imbalanced classes, and because they are useful metrics to not only gauge the probability that a fault would be detected but also the confidence that the detection is correct. The reasoning being that it is as important to a power plant operator to avoid false alarms as it is to be alerted of every possible fault.

Cross-validation

This heuristic selection method is vulnerable to random chance. To address this vulnerability, cross-validation is used. The data sets were previously, during the feature selection process, split into a hold-out test data set and a remainder data set. Since the results of a single train/test cycle can be very dependant upon the split of the samples, the classifiers were evaluated by their average performance across a 5-fold Cross-Validation (CV). This

Table 3.7: A summary of the hyperparameters used to compare feature data sets and classifiers. The table is not exhaustive, but includes the most important hyperparameters. The hyperparameters not included were kept as the default for their respective software libraries.

Classifier	Hyperparameter	Setting
KNN	K	20
	Weight	Uniform
SVM (rfb)	Kernel	Radial basis function
	Gamma	$\frac{1}{\text{Number of features}}$
PCA	Spanned variance	95%
XGBoost	Eta	0.3
	Max depth	6
Neural net	Number of hidden layers	2
	Num. neurons 1st layer	200
	Num neurons 2nd layer	100
	Num neurons 3rd layer	14
	Activation function	ReLU

produces 5 folds of CV-train and CV-validation sets drawn from the remainder data set of the initial split. The folds are identical across all classifiers and feature data sets.

The cross-validation split the samples in the remainder data set according to the OSS they belonged to, ensuring that there are no samples from the same OSS in both training and validation sets as RSSes from the same OSS were deemed to be too similar. The objective is to train a classifier to identify faults, not to identify from which OSS the samples are drawn. To check this assertion, the classifiers were evaluated once using cross-validation with random splitting. This resulted in classifiers with near perfect accuracy, an indication that the OSS-dependent split was necessary.

Standardisation

Logistic Regression, KNN and SVM are sensitive to the variance of the samples, this is addressed by applying standardisation. Each cross-validation split was standardised to zero-mean and unity variance. The mean and variance of every feature was calculated from the CV-train set. Both CV-test and CV-validation sets were standardised using the CV-train means and variances.

Results

This procedure was repeated for every classifier on every feature data set and performance metrics were gathered. The results are presented in Table 3.8. This method of model fitting was used for every classifier evaluation at later stages of the classifier development.

Table 3.8: A summary of the results of Logistic Regression, KNN, SVM (radial base function kernel), SVM (linear), Logistic Regression with PCA, KNN with PCA, SVM (radial base function kernel) with PCA, SVM (linear) with PCA, XGBoost, MLP and stack classifiers trained on each data set. Average scores across all classifiers for each data set is also included.

Data set	Classifier	Sensitivity	Precision	ROC AUC
A	Logistic Regression	0.8853	0.7722	0.6774
	Logistic Reg. with PCA	0.8622	0.8131	0.7179
	KNN	0.8269	0.6747	0.5288
	KNN with PCA	0.8201	0.6775	0.5335
	SVM (rbf)	0.8492	0.7050	0.5834
	SVM (rbf) with PCA	0.8538	0.6312	0.4453
	SVM (linear)	0.8859	0.7612	0.6705
	SVM (linear) with PCA	0.8576	0.8176	0.7175
	XGBoost	0.8518	0.7766	0.6788
	Multi-layer Perceptron	0.8833	0.7390	0.6420
	Stack	0.8652	0.8191	0.7179
	Average classifier score	0.8583	0.7443	0.6285
B	Logistic Regression	0.8675	0.7772	0.6721
	Logistic Reg. with PCA	0.8140	0.7394	0.6279
	KNN	0.8074	0.7237	0.6053
	KNN with PCA	0.8392	0.7207	0.6075
	SVM (rbf)	0.8117	0.7029	0.5728
	SVM (rbf) with PCA	0.8149	0.6453	0.4761
	SVM (linear)	0.8790	0.7925	0.6905
	SVM (linear) with PCA	0.7878	0.7392	0.6155
	XGBoost	0.8407	0.7193	0.5938
	Multi-layer Perceptron	0.8702	0.7322	0.6233
	Stack	0.8712	0.7981	0.6955
	Average classifier score	0.8367	0.7355	0.6164
C	Logistic Regression	0.8966	0.7998	0.7092
	Logistic Reg. with PCA	0.8663	0.8082	0.7271
	KNN	0.8282	0.6743	0.5287
	KNN with PCA	0.8222	0.6756	0.5306
	SVM (rbf)	0.8531	0.7226	0.6062
	SVM (rbf) with PCA	0.8492	0.6327	0.4477
	SVM (linear)	0.8972	0.8106	0.7389
	SVM (linear) with PCA	0.8615	0.8162	0.7289
	XGBoost	0.8313	0.7816	0.6747
	Multi-layer Perceptron	0.8859	0.7643	0.6869
	Stack	0.8714	0.8485	0.7627
	Average classifier score	0.8603	0.7577	0.6492

3.6.1 Feature selection and reduction performance

For selection of the best feature data set, three things are taken into account. Firstly, the feature data set with the best average performance across the different classifiers is superior. Any aid to classification performance should be pursued. Secondly, the one with more consistent scores, i.e. smaller variance in the results from cross-validation, is preferred. Thirdly, the feature data set with fewest features is superior if the performance is similar among all the sets. This is due to two reasons: a smaller number of features reduces training and prediction time, and fewer features reduce the risk of overfitting to the data.

Figure 3.11 shows a rough summary of the performances of the classifiers on each feature data set. It appears that the choice of data set does not greatly affect the performance of the classifiers, and the variance of the results is large. However, feature data set C, the TSFRESH feature selection data set, slightly outperforms the rest on every averaged metric. Data set C is thus preferred, and will be utilised from this point onward.

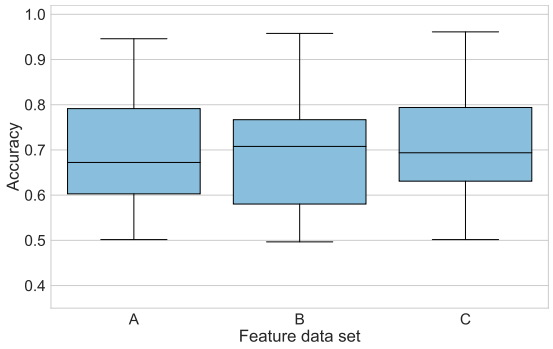
As for feature reduction, i.e. application of PCA, every classifier suffered a drop in performance in nearly every metric when PCA was applied. Of special note is that radial basis function SVM with PCA had an ROC AUC consistently lower than 0.5, which indicates that it performed worse than chance. Due to this, PCA was abandoned. It might still have been justified on grounds of reducing training and prediction time if there were more features or an extremely large number of samples, but no such considerations were necessary.

3.6.2 Hyperparameter optimisation and selection

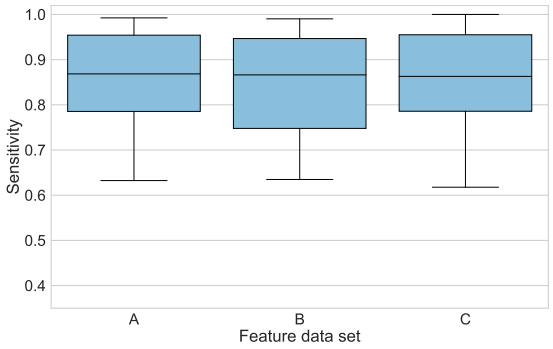
Since a classifier's performance is heavily dependent upon its hyperparameters, all the candidate classifiers were optimised before selecting among them. The optimisation procedure was a 5-fold cross-validating grid search. In this procedure, a hyperparameter grid is defined that contains a range of values for each of the hyperparameters to be optimised. The grid search algorithm then executes a cross-validation of the classifier for every possible combination of these hyperparameters. The mean cross-validation performance is calculated for each hyperparameter combination, and the hyperparameter combination that yields the best performance on the chosen performance metric is selected.

Depending on the complexity of the classifier and the number of hyperparameters to be optimised, the grid search can span thousands of hyperparameter combinations. Since the data set contains so few samples, the training time of each hyperparameter combination was short and a large hyperparameter grid could be investigated. The performance metric used was F1-score because it combines sensitivity and precision.

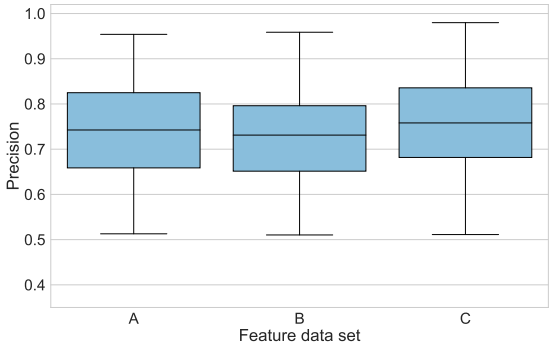
A broad search of the scientific literature yielded few recommendations on hyperparameter grids for grid searches. In lieu of scientific literature, industry expertise was sought out by consulting Kaggle. Kaggle is a site where companies submit data sets with accompanying machine learning problems and post rewards for the best solution, and the winning implementations are open to the public. The hyperparameter grids chosen are therefore interpolations between hyperparameters chosen in winning implementations of



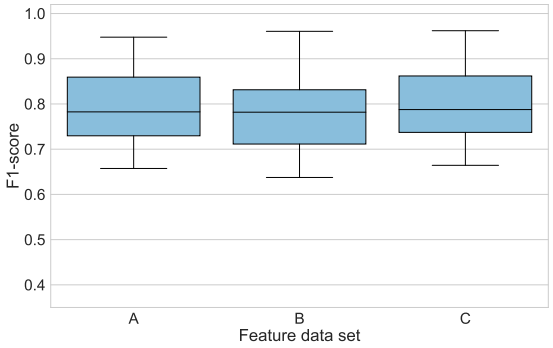
(a) Accuracy.



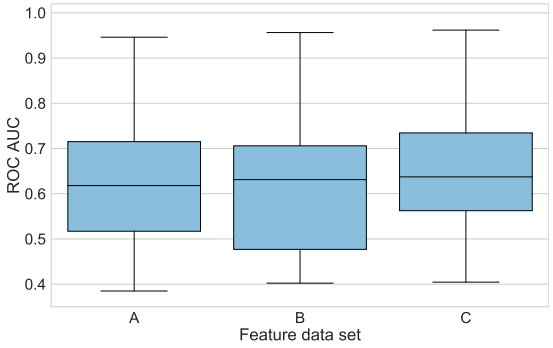
(b) Sensitivity.



(c) Precision.



(d) F1-score.



(e) Receiver operating curve area under curve.

Figure 3.11: The performances across all classifiers on each feature data set are shown in box-and-whisker plots. The boxes extend from the upper to the lower quartile of the distribution, the centre line in each box denotes the median score, and the whiskers envelope the greatest and lowest scores.

the respective algorithms on similar problems from Kaggle. Similar problems were defined as tabular data sets with between 50 and 800 feature columns and between 50 and 5 000 samples, preferably with high cardinality.

The hyperparameter grids that were tested are shown in Table 3.9. The search grid sizes increase with the complexity of the classifiers. KNN, SVM, and Logistic Regression have 175, 50, and 175 different hyperparameter combinations each, while XGBoost boasts 2304 different combinations. This is because XGBoost, being an ensemble classifier, required a larger set of hyperparameter variations to do a thorough grid search. Finally, 256 different combinations of hyperparameters were tried for the MLP.

Table 3.9: Hyperparameter search grids for Logistic Regression, KNN, SVM, and XGBoost classifiers. Note that l1 and l2 are Lasso and ridge regression, respectively. *rbf* and *linear* kernels correspond to radial basis function and linear SVMs. Regarding hidden_layer_sizes: In a configuration of (a,b,c), the depth of the MLP is determined how may numbers there are, in this case three hidden layers deep. Each of these layers have a, b, and c neurons each in order of increasing distance from the input layer.

Classifier	Hyperparameter	Values	Description
Log. Reg.	C	10^k, $k = -10, -9.5, ..., 10$	Inverse of regularisation strength
	penalty	"l1", "l2"	Penalisation norm
KNN	n_neighbors	$1, 3, 5, ..., 351$	Number of nearest neighbours
SVM	C	10^k, $k = -1, 0, 1, 2, 3$	Inverse of regularisation strength
	gamma	10^k, $k = 0, -1, -2, -3, -4$	Inverse of regularisation strength
	kernel	"rbf", "linear"	Kernel type
XGBoost	learning_rate	$0.01, 0.2, 0.3, 0.5$	Learning rate
	n_estimators	$100, 400, 700, 1000$	Number of trees in ensemble
	max_depth	$3, 10, 15, 25$	Maximum tree depth
	col_sample_bytree	$0.8, 1$	Per tree column subsampling ratio
	subsample	$0.6, 0.8, 1$	Sample subsampling ratio
	reg_alpha	$0.7, 1, 1.3$	L1 regularisation term on weights
	reg_lambda	$0, 0.5, 1$	L2 regularisation term on weights
MLP	activation	'identity', 'logistic', 'tanh', 'relu'	The activation function
	batch_size	200, 133, 66, 32	Size of minibatches
	max_iter	200, 500, 1000, 1200	The maximum number of epochs
	hidden_layer_sizes	(50,25,3), (100,50,7), (200,100,14), (300,150,21)	Size and number of hidden layers

The hyperparameter sets with the greatest mean performance across 5-fold cross-validation for each classifier are presented in Table 3.10. Table 3.11 shows the scores of these classifiers across several metrics. Of the optimised classifiers, the XGBoost and KNN are outperformed by the others. KNN's accuracy was 64.0% in an imbalanced data set of 65.9% majority class. This performance is worse than that of a dummy classifier that classifies randomly or always classifying samples as the majority class. Furthermore, KNN is entirely non-generalising with a $k = 1$, implying that the algorithm is not well suited for this problem at all since this was the best result from a grid search of k-values

from 1 to 351.

Table 3.10: The best hyperparameters found from the grid search.

Classifier	Hyperparameter	Value
Logistic Regression	C	$10^{8.5}$
	penalty	l2
KNN	n_neighbors	1
SVM	C	10
	gamma	1
	kernel	linear
XGBoost	colsample_bytree	0.800
	learning_rate	0.500
	max_depth	3
	n_estimators	100
	reg_alpha	1.300
	reg_lambda	0
	subsample	1
Multi-layer Perceptron	activation	identity
	batch_size	200
	hidden_layer_sizes	(50, 25, 3)
	max_iter	200

Table 3.11: The accuracy, sensitivity, precision, F1-score and ROC AUC of the best models found in the hyperparameter grid search.

Classifier	Accuracy	Sensitivity	Precision	F1-score	ROC AUC
Logistic Regression	0.7986	0.8740	0.8376	0.8506	0.7606
KNN	0.6395	0.8350	0.6990	0.7501	0.5723
SVM	0.7940	0.8854	0.8247	0.8501	0.7500
XGBoost	0.7438	0.8576	0.7846	0.8142	0.6927
Multi-layer Perceptron	0.7958	0.9022	0.8170	0.8542	0.7340

3.6.3 Stacking classifiers

Since a stacking classifier improved the performance during the feature data set selection, the same approach is made again using the optimised classifiers. Four stacking classifiers were made with different meta-classifiers, Logistic Regression, MLP, gradient boosting forest, and a random forest classifier. The gradient boosting forest classifier was chosen over XGBoost as a meta-classifier due to greater compatibility with Sci-kit Learn's stacking framework. Since XGBoost is also a variant of gradient boosting forest, it should return similar results at the expense of computing power. The stacks all include the optimised Logistic Regression, SVM, MLP, and XGBoost classifiers as base classifiers. KNN was again excluded due to its poor performance and slow prediction time. Results

are shown in Table 3.12.

Table 3.12: The results from the stacking classifier comparison.

Meta-classifier	Accuracy	Sensitivity	Precision	F1-score	ROC AUC
Logistic Regression	0.7840	0.8701	0.8260	0.8432	0.7472
Multi-layer Perceptron	0.7479	0.8057	0.8276	0.8107	0.7267
Gradient boosting forest	0.7663	0.8268	0.8304	0.8255	0.7314
Random Forest	0.7704	0.8216	0.8388	0.8265	0.7440

Of the stacking classifiers, the Logistic Regression stacking classifier outperformed the others by a large margin. The best stacking classifier is shown in Figure 3.12.

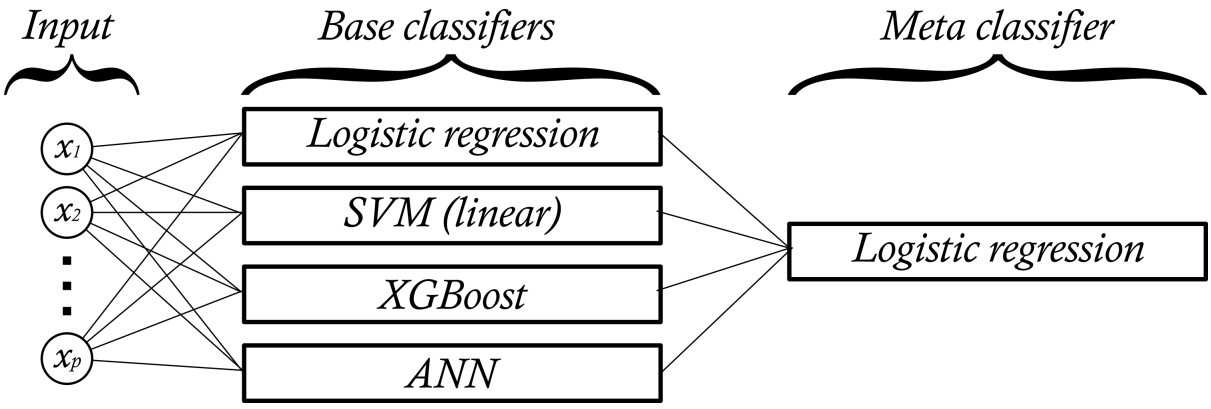


Figure 3.12: A stacking classifier with Logistic Regression as its meta-classifier.

Comparing the performance of the best stacking classifier with that of the best non-ensemble classifier, a somewhat surprising result surfaces. The logistic regression classifier alone on average slightly outperforms the stacking classifier of which it is a part of across the cross-validation folds.

3.6.4 Final classifier

An advantage of stacking classifiers is that they often generalise better than single classifiers, and they usually outperform their base classifiers. However, the hyperparameters of the meta-classifier have not been optimised on the training set as is the case with the simple logistic regression classifier. To gauge their performance on unseen samples, both are trained on the entire training set and tested on the hold-out data set. The results are presented in Table 3.13. On the hold-out set, the stacking classifier outperforms the simple logistic regression classifier. The stacking classifier could likely be further improved by running a grid-search for the optimal hyper parameters of the logistic regression meta-classifier, but this was too computationally expensive to complete within a reasonable time frame without a large, time-consuming refactoring of the code base.

The logistic regression stacking classifier has coefficients that weigh each of its base-classifiers. Since all the base-classifiers return predictions in the same interval, from 0 to

Table 3.13: The results of the best of the single and stacking classifiers on the hold-out data samples.

Classifier	Accuracy	Sensitivity	Precision	F1-score	ROC AUC
Logistic Regression	0.7569	0.6961	0.9435	0.8011	0.7986
Logistic Reg. stack	0.8448	0.8456	0.9274	0.8846	0.8443

Table 3.14: The base-classifier coefficients of the logistic regression classifier used as meta-classifier in the stacking classifier. The models are ranked in order of importance to the final prediction.

Rank	Base-classifier	Coefficient
1	SVM	3.464
2	XGBoost	1.365
3	Logistic regression	-1.042
4	Multi-layer Perceptron	1.036

1, the absolute magnitude of the coefficients is correlated with how large an emphasis is placed on each particular base-classifier. The coefficients of each base-classifier are shown in Table 3.14.

3.6.5 Feature usefulness

From both logistic regression and XGBoost classifiers it is possible to extract feature importances. In the logistic regression classifier, they correspond to the weights associated with the features as described in Section 2.3.7, and in XGBoost they are the average gain across all splits the feature was used in. The most important features for the XGBoost and logistic regression classifiers are shown in Tables 3.15 and 3.16.

Table 3.15: The 20 most useful features for the optimised logistic regression classifier. ALTL is an abbreviation of aggregated linear trend line. The reader is referred to Appendix C for more detailed descriptions of the TSFRESH features.

Rank	Feature	Description
1	DWT__RWE11	RWE, decomposition level 11
2	agg_linear_trend(chunk_len=50, f_agg='max')	TSFRESH, intercept of maximum value ALTL with chunk length 50
3	DWT__RWE12	RWE, decomposition level 12
4	agg_linear_trend(chunk_len=50, f_agg='min')	TSFRESH, standard deviation of minimum value ALTL with chunk length 50
5	approximate_entropy(m=2, r=0.3)	TSFRESH, measure of regularity in time series
6	c3(lag=1)	TSFRESH, measure of non-linearity in time series
7	DWT__RWE10	RWE, decomposition level 10
8	agg_autocorrelation(f_agg="var",maxlag=40)	TSFRESH, variance of autocorrelations for lags up to 40
9	FFT 450.0 Hz	FFT, frequency magnitude at 450.0 Hz
10	FFT 314.3 Hz	FFT, frequency magnitude at 314.3 Hz
11	c3(lag=2)	TSFRESH, measure of non-linearity in time series
12	FFT 442.9 Hz	FFT, frequency magnitude at 442.9 Hz
13	binned_entropy(max_bins=10)	TSFRESH, entropy of the sample magnitude distribution
14	DWT__HWE7	HWE, decomposition level 7
15	FFT 371.4 Hz	FFT, frequency magnitude at 371.4 Hz
16	change_quantiles(f_agg="var",isabs=False, qh=0.6,ql=0.0)	TSFRESH, variance of consecutive changes in measurements between 0.0 and 0.6
17	FFT 350.0 Hz	FFT, frequency magnitude at 350.0 Hz
18	FFT 100.0 Hz	FFT, frequency magnitude at 100.0 Hz
19	FFT 178.6 Hz	FFT, frequency magnitude at 178.6 Hz
20	time_reversal_asymmetry_statistic(lag=2)	TSFRESH, please see appendix C

Table 3.16: The 20 most useful features for the optimised XGBoost classifier. The reader is referred to Appendix C for more detailed descriptions of the TSFRESH features.

Rank	Feature	Description
1	DWT__RWE10	RWE, decomposition level 10
2	time_reversal_asymmetry_statistic(lag=1)	TSFRESH, please see appendix C
3	partial_autocorrelation(lag=7)	TSFRESH, partial autocorrelation with a lag of 7
4	change_quantiles(f_agg="mean",isabs=True, qh=0.4,ql=0.0)	TSFRESH, mean absolute change of all measurements between 0.0 and 0.4
5	DWT__RWE11	RWE, decomposition level 11
6	time_reversal_asymmetry_statistic(lag=2)	TSFRESH, please see appendix C
7	DWT__HWE1	HWE, decomposition level 1
8	c3(lag=3)	TSFRESH, measure of non-linearity in time series
9	FFT 121.4 Hz	FFT, frequency magnitude at 121.4 Hz
10	autocorrelation(lag=3)	TSFRESH, autocorrelation with a lag of 3
11	FFT 450.0 Hz	FFT, frequency magnitude at 450.0 Hz
12	DWT__TWE10	TWE, decomposition level 10
13	autocorrelation(lag=6)	TSFRESH, autocorrelation with a lag of 6
14	autocorrelation(lag=2)	TSFRESH, autocorrelation with a lag of 2
15	DWT__IWE1	IWE, decomposition level 1
16	DWT__IWE10	IWE, decomposition level 10
17	DWT__TWE11	TWE, decomposition level 11
18	approximate_entropy(m=2,r=0.5)	TSFRESH, measure of regularity in time series
19	skewness	TSFRESH, skewness of the sample
20	c3(lag=1)	TSFRESH, measure of non-linearity in time series

3.7 Fault severity assessment

An attempt was also made to determine fault severity using some of the same methodology as above. This was done by creating several one-versus-all classifiers that tried to determine the severity of the fault. This means that the final classification is really made by a collection of several binary classifiers that each check if the sample belongs to a certain class, for example the no fault class. The final classification chooses the class with the associated binary classifier with the greatest confidence that the sample belongs to its class.

The objective of these classifiers was to assess the fault severity of a sample by classifying it as one of several defined fault severities. The exceptions were XGBoost, KNN and MLP classifiers, which are capable of multi-class classification without employing the one-versus-all technique. Since there were so few experiments done, only the no fault condition had more than four experimental cases, it was immediately obvious that it would be fruitless to attempt to determine the exact fault severity due to data constraints. Effectively 4 samples of each case were present in the data set, not counting different sampling rates as separate samples. In an attempt to remediate this, the classifiers were made to classify samples into either *no fault*, *low severity*, *moderate severity* or *high severity*. These degrees of severity were defined so as to split the available samples as evenly as possible between the severity classes while still being informative. They are as shown in Table 3.17.

The resulting classifiers gave either terrible predictions that were near random guesses when the RSS were split according their OSS as detailed in Section 3.5, or close to perfect predictions when the RSS were split randomly.

Table 3.17: The severity degrees of the classifier defined by the number of ITSCs. The rightmost column contains the number of experiments done in that state.

Severity case	**ITSCs**	**Number of experiments**
No fault	0	16
Low severity	1 to 6	12
Moderate severity	7 to 10	8
High severity	Above 10	12

Chapter 4

Discussion

In this chapter, the results from Chapter 3 are discussed in light of the theory presented in Chapter 2 and recommendations for further work are made.

4.1 Data management and pre-processing

The RSS length was chosen to maximise the number of samples from each OSS, exploiting the assumption that the captured signal is stationary and that the signal could be cut easily in zero-passing to capture integer electrical periods. This was possible due to the periodic nature of the air-gap magnetic field. If the same technique is applied to signals originating from other sensor types mounted to the machine such as vibration or stray magnetic field sensors, this may not be the case.

There were two Hall-effect sensors mounted in the generator that made measurements concurrently. The measurements from these sensors have been treated as independent and were assigned to separate OSSes. The motivation for this is, like many decisions in this thesis, to maximise the information extracted from the source data. The justification that the sensors make effectively independent measurements of the same machine state is that their idiosyncrasies and differences in mounting would affect the measurements. One would expect to see improbably high detection rates if they were too similar, but this was not observed.

The classifiers' performance varied greatly between folds of the same cross-validation run and a leap in classification performance occurred when evaluating the final classifiers on the hold-out set. This highlights some issues with small data sets. With so few OSSes to draw from, the splitting of the data set changes the data in the training set drastically. This could also be the reason for the change in performance in the final evaluation. The logistic regression classifier that had performed well previously lost performance when introduced to more data, while the logistic regression stacking classifier improved its performance by relying primarily on its linear SVM base-classifier. Which classifier performs best is thus very dependent upon which parts of the data set it is trained on. The best classifier found for the data set generated in this thesis was the logistic regression stacking classifier presented in Section 3.6.4, but that would likely change with introduction of more data.

Since performance varies between CV-folds, it was important to split the data set in

the same way for every cross-validation for every classifier and feature data set to ensure direct comparability between the results.

4.2 Feature extraction and importance

Some of the features that TSFRESH calculate are very computationally demanding. TSFRESH's computation time to generate all the features for a single RSS was 4.7 seconds on a system with a Intel Core i5-6200U CPU. This would allow a monitoring system of comparable computational power to gauge the machine's condition roughly once every 5 seconds. Nevertheless, including all the features increased the chance that useful features were not overlooked.

Approximate entropy, change quantiles, entropy, and aggregated linear trend lines are the 4 most computationally intensive features among the TSFRESH generated features [20]. These features are more than 2 orders of magnitude more computationally intensive than the average TSFRESH feature [20], and were included among the most useful features for both the optimised logistic regression and the optimised XGBoost classifiers. Considering that the computation time of the DWT wavelet energy and FFT features were insignificant compared to TSFRESH generated features, there is little room to reduce feature extraction time. Due to the non-critical nature of incipient faults, however, a condition monitoring system that diagnoses the machine once every 5 or 10 seconds could be sufficient.

The range of classifiers investigated was intentionally broad, including both linear and non-linear models. The best performers after the grid search were linear models such as the logistic regression, SVM with a linear kernel, and MLP with the identity activation function. It is worth noting that even though Multi-Layer Perceptrons are generally non-linear models, they are linear if the activation function is linear. This is in accordance with the EDA that showed a great deal of linear correlation between the target and several features.

This raises an interesting proposition. The features included in the feature data set for this thesis were the most promising of the ones discovered in [5]. The heuristic used to distinguish useful features in [5] was to compare the features' magnitude to the fault severity, effectively looking for a *linear* relationship to fault severity. This means that since there was a selection bias for linearly correlated features when deciding which features to include, it is only to be expected that linear classifiers can do well on the data set. An implication of this is that features that were discarded in [5] should be investigated anew by inclusion into a feature set for machine learning.

The most useful features among the ones investigated were the high decomposition level DWT RWE features, aggregate linear trend features, approximate entropy features, and change quantile features. TSFRESH generated features performed similarly as the wavelet energy and FFT features did. Correlation to the target value was a strong indication that features would be useful in classification and could thus be used to screen a large number of potential features without having to train classifiers. Screening in such

a manner would come at the risk of missing non-linear relationships as mentioned above.

4.3 Feature selection and target leakage

Due to the small size of the data set, the feature selection algorithms were run on the entire data set excluding the hold-out data set. This likely affected the cross-validation results of the classifiers run on the feature selected data sets. Since the feature selection algorithms selected the best features based on *all* the samples that could appear in each CV-fold, some target leakage occurred. This can cause optimistic cross-validation results. This does not, however, affect the results from the hold-out data set since it was set aside before feature selection.

An alternate approach that would eliminate the aforementioned target leakage is to eliminate the hold-out data set and run a feature selection step as part of each cross-validation fold. This would eliminate target leakage and allow feature selection on a data set of similar size, but it has another draw-back. Without a hold-out data set, the final evaluation of the classifiers would be done on the same samples that were used to choose among the classifiers and their hyper-parameters. This would introduce another source of target leakage and the final evaluation of the classifiers performance on unseen samples would be unrealistically optimistic, something that was avoided with the approach taken.

4.4 Classifier selection

The classifier performances increased markedly as a result of the grid search. However, a general trend among the optimised classifiers was that the less complex classifiers performed better. An example of this is that the optimised hyperparameters of MLP were of the smallest number of neurons included in the search with the identity function as activation function. The hyper-parameters with the greatest performance were, with the exception of the optimised logistic regression classifier, on the lower-complexity-extremes of their associated search-grids. Another grid search with more low-complexity hyper-parameters could likely have yielded better results.

4.4.1 Performance

The performance of the classifiers developed in this thesis has been to some standards unimpressive, which is to be expected when the source data set is reasonably diverse and small. With a small data set comes the challenge of maximising the information extracted from it. As has been mentioned earlier is this report, this involves correct handling of data set splitting and avoiding target leakage. A close to 100 % accuracy is in this regard an indication of mishandled data management, as was exemplified when non-optimised classifiers achieved nearly perfect predictions when samples were randomly split.

4.5 Real-world validity

An experimental generator was used to generate the data set. Even though it is constructed to resemble hydropower generators, it still has its own unique geometry, its sensors were mounted in a certain way and position, it is placed in a laboratory that is a noisy electromagnetic environment, and it is not driven by an actual turbine. All of these discrepancies with any one production hydropower generator makes any classifier trained on measurements taken from the experimental generator useless on machines other than Brutus. While this is nearly universally true for machine learning models, it is worth mentioning that a new model must be made for each machine.

The measurement series that were used contained very little diversity of fault types, fault severities, and operating conditions. This reduces the robustness of the classifier since it can more easily mistake novel operating conditions for fault conditions. Furthermore, it was concluded in [4] that the machine had a 3% dynamic eccentricity at the time the measurements were taken. The eccentricity was thus a constant influence throughout these measurements and would not interfere with the creation of the classifiers, but it reduced the generality of the results gathered in this thesis. The classification rates and the features that worked best are for an eccentric machine, and these features may not be as useful in a machine without eccentricity. This is something that could be rectified by making measurements as originally intended for this thesis, with a variety of operating conditions and faults. A real generator operates under a multitude of load conditions and any evaluation of features should be done using data that reflects this diversity.

The experimental generator was not driven by turbine, but rather an induction motor. Turbines can introduce vibrations into the generator that could affect the features found to work well here. Some of these are hydraulic imbalance [32], cavitation [33], and runner blade damage [32]. These sources of vibration can affect the air-gap magnetic field in a number of ways as they introduce torque variations to the shaft. While cavitation induced vibrations are usually wide band, high-frequency noise, hydraulic imbalance and runner blade damage induced vibrations occur at multiples of the mechanical frequency of the rotor. Since the mechanical frequency of the turbine and the generator is necessarily a common frequency, the state of the turbine would affect the frequency content of features based on the mechanical frequency of the generator's rotor. The FFT feature set generated for this thesis would be particularly susceptible to this.

4.6 Real-world applicability

Two prerequisites exist to apply a fault classification system like the one created in this thesis to a production machine. The machine would have to be outfitted with a Hall-effect sensor in the air-gap, and measurements of the machine in several different load and fault conditions would need to be taken prior to developing the classifier. The sensor itself is small relative to the air-gap of a production machine, and should not be difficult to install, considering it was successfully installed in a much smaller machine. It is, however, unlikely that a production machine could be run with induced faults to construct the data set necessary.

This approach is thus not practical for real-world applications with a few possible exceptions. If the machine is itself of a standardised make and model, then it could be possible to gather sensor data of faulty operation in one or a few machines to train a classifier that could be used in other machines of the same type. Generators can have imperfections that stem from production and/or assembly in addition to differences in the connected turbines that could interfere with the classification. To how great an extent this would be a problem is speculative. Many of the features generated for this thesis are dependent upon machine geometry such as the number of poles, which is shared among machines of a similar make. It would nevertheless be an expensive method of gathering data due to the cost of generators and the small production runs. Some suggestions of how to make more generalised classifiers are presented in the next section.

4.7 Suggested methods

In this section two suggestions for methods of extending fault detection to more machines are presented in increasing order of complexity.

4.7.1 Anomaly detection

Anomaly detection is to develop a model of the machine in healthy operating state, and treat deviations from the model as fault indications. This is feasible to implement in production machines since the training data is only data of the healthy machine operation in different load conditions, something much more easily obtained than fault condition measurements. A machine learning model that could be suited for this is the auto-encoder. It is a multilayered neural net where the input and output layers are of the same size, with smaller intervening layers. The auto-encoder is forced to discard some input information since the intervening layers are smaller than the input layer. It is trained using back-propagation to reconstruct the input in the output layer until it does so with a low error rate. A normal state is then encoded into the neurons of the model. Anomalous inputs would thus be reconstructed poorly and incur a high reconstruction error, indicating an anomaly. The simplest model to produce would also be the least informative. It would only indicate that there is an anomalous operating condition, not what that condition is. This model could use features similar to the ones generated in this thesis, but should have a great deal of healthy training data that spans every acceptable operating condition to reduce the false-positive rate.

4.7.2 Simulated data generation

One of the greatest hindrances to developing fault detection classifiers for industry machines is the lack of labelled fault data. This could be addressed by use of a simulation that includes the generator along with any sensors mounted to it. The finite element simulation could be calibrated against measurements of the real machine, so that the simulation generates sample series in agreement with sample series of healthy operation of the machine. To generate faulty measurement series, the same simulation modified to include induced faults would be used. Sample series from simulated faulty and healthy operation would then be used to train the fault classifier. This would be more easily

implemented in industry since there is very little labelled data of fault conditions in existing machines. In addition the method is very non-invasive, requiring only the sensor installation in the generator. The model's false-positive performance could be assessed by making predictions on a healthy data set gathered from the machine, but a major challenge is that there would not exist faulty testing data from the real machine to assess its efficacy in making true fault detections.

Chapter 5

Conclusion

This thesis has investigated how signal processing and machine learning tools can be used to detect inter-turn short-circuits in rotor field windings. To do this, a fault detection system was implemented to detect ITSC faults based on measurements from a single Hall-effect sensor mounted on stator tooth inside the air-gap of a salient-pole synchronous generator. This was done in three stages, data pre-processing, feature extraction and selection, and classifier development. The objectives were specifically to investigate which features are most useful, which machine learning models perform best in this task, and lastly if a single air-gap magnetic field sensor is sufficient for reliable fault detection or if more sensors are required.

The features extracted were power spectral density of integer multiples of the generator's mechanical frequency extracted by FFT, DWT wavelet energies, and the entire TSFRESH feature extraction suite excluding their FFT features. The most useful features were the Relative Wavelet Energy features and some of the TSFRESH features as presented in Tables 3.15 and 3.16. The performance of TSFRESH generated features paralleled that of the DWT features and surpassed that of the FFT features, indicating that automatic feature extraction is useful for these tasks.

Linear machine learning models were best suited for fault detection on this data set, especially the logistic regression and linear SVM classifiers. KNN was not suited, and did worse than random chance. The performance decreased somewhat on averaged cross-validation when the classifiers were stacked, but generalised better when tested on the hold-out data set. The best classifier was an ensemble stacking classifier with logistic regression as the meta-classifier taking inputs from logistic regression, XGBoost, linear SVM, and MLP classifiers as base-classifiers.

The results indicate that ITSC fault classification using machine learning on air-gap magnetic field measurements from a single sensor can yield good results. The logistic regression stacking classifier had an accuracy of 0.8448, a sensitivity of 0.8456, and a precision of 0.9274. This means that the classifier correctly classified 84.48 % of all the samples in the hold-out data set, and 84.56 % of the faulty samples present were correctly classified as such. Of the samples that were classified as faulty, 92.74 % were correctly classified. Since a large portion of faults go undetected, this fault detection system should therefore not be relied upon as the only detection system. However, if the system alerts of a fault, it would warrant investigation since it is likely to be correct. This is predicated

upon a similar performance on out-of-set samples. Assuming similar performance from this classifier on novel samples is naive due to its limited training data. The robustness of the classifier could likely be improved by creating a more diversified data set.

5.1 Further work

Further work suggested is:

- Combine air-gap magnetic field readings from Hall-effect sensors with concurrent readings from sensors such as voltage over and current through stator and rotor windings, and stator vibration to assess the benefit of combining sensors.
- Implement an anomaly detection system using an auto-encoder.
- Implement an anomaly detection system using an artificial recurrent neural network such as *long short-term memory* (LSTM) or *gated recurrent units* (GRU).
- Create a classification algorithm trained on fault condition measurements from a simulation of the machine and compare its predictions with real measurements of known fault conditions from the physical machine. This would be more easily implemented in industry since there is very little labelled data of fault conditions in existing machines and is very non-invasive.
- Implement a regression model to gauge fault severity.
- Investigate if performance improves when more DWT wavelet energy features using other wavelets is included.
- Implement a convolutional neural network using CWT scalograms or STFT spectrograms as base-classifiers in an ensemble along with the models implemented here.
- Implement feature extraction using order analysis to account for changes in machine speed. Order analysis is a technique to adjust the sample series in such a way so that it is referenced to for example the mechanical speed of the machine. The features extracted would thus be similar even at different machine speeds. This could enable detection of incipient faults such as broken damper bars that rely on acceleration in the machine to be apparent.

Bibliography

[1] D. P. Kothari and I. J. Nagrath, *Electric Machines*, en. Tata McGraw-Hill Education, 2004, Google-Books-ID: axGw7r3SOEMC, ISBN: 978-0-07-058377-1.

[2] *08583: Elektrisitetsbalanse (MWh) 2010m01 - 2019m09*. [Online]. Available: http://www.ssb.no/statbank/table/08583/ (visited on 04/21/2020).

[3] J. Manyika, J. Woetzel, R. Dobbs, M. Chui, P. Bisson, J. Bughin, and D. Aharon. (Jun. 2015). Unlocking the potential of the internet of things | McKinsey. Library Catalog: www.mckinsey.com, [Online]. Available: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world (visited on 06/25/2020).

[4] I. L. Groth, "On-line Magnetic Flux Monitoring and Incipient Fault Detection in Hydropower Generators", Master's thesis, The Norwegian University of Technology and Science, Trondheim, Jun. 2019.

[5] T. N. Skreien, "Application of signal processing and machine learning tools in fault detection of synchronous generators", Department of Electric Power Engineering, NTNU – Norwegian University of Science and Technology, Project report in TET4520, Jan. 2020.

[6] C. Staubach and S. Krane, "Detection of faults in rotor-windings of turbogenerators", in *2016 Conference on Diagnostics in Electrical Engineering (Diagnostika)*, ISSN: null, Sep. 2016, pp. 1–4. DOI: 10.1109/DIAGNOSTIKA.2016.7736500.

[7] J. Yun, S. Park, C. Yang, Y. Park, S. B. Lee, M. Šašić, and G. C. Stone, "Comprehensive Monitoring of Field Winding Short Circuits for Salient Pole Synchronous Motors", *IEEE Transactions on Energy Conversion*, vol. 34, no. 3, pp. 1686–1694, Sep. 2019, ISSN: 1558-0059. DOI: 10.1109/TEC.2019.2905262.

[8] J. A. Antonino-Daviu, M. Riera-Guasp, J. Pons-Llinares, J. Roger-Folch, R. B. Pérez, and C. Charlton-Pérez, "Toward Condition Monitoring of Damper Windings in Synchronous Motors via EMD Analysis", *IEEE Transactions on Energy Conversion*, vol. 27, no. 2, pp. 432–439, Jun. 2012, ISSN: 1558-0059. DOI: 10.1109/TEC.2012.2190292.

[9] Y. Han and Y. Song, "Condition monitoring techniques for electrical equipment-a literature survey", *IEEE Transactions on Power Delivery*, vol. 18, no. 1, pp. 4–13, Jan. 2003, ISSN: 1937-4208. DOI: 10.1109/TPWRD.2002.801425.

[10] R. Priemer, *Introductory Signal Processing*, en. World Scientific, 1991, Google-Books-ID: QBT7nP7zTLgC, ISBN: 978-9971-5-0919-4.

[11] E. O. Brigham and R. E. Morrow, "The fast Fourier transform", *IEEE Spectrum*, vol. 4, no. 12, pp. 63–70, Dec. 1967, ISSN: 1939-9340. DOI: 10.1109/MSPEC.1967.5217220.

[12] C. Torrence and G. P. Compo, "A Practical Guide to Wavelet Analysis", *Bulletin of the American Meteorological Society*, vol. 79, no. 1, pp. 61–78, Jan. 1998, ISSN: 0003-0007. DOI: 10.1175/1520-0477(1998)079<0061:APGTWA>2.0.CO;2. [Online]. Available: https://journals.ametsoc.org/doi/abs/10.1175/1520-0477%281998%29079%3C0061%3AAPGTWA%3E2.0.CO%3B2.

[13] B. Y. Lee and Y. S., "Application of the Discrete Wavelet Transform to the Monitoring of Tool Failure in End Milling Using the Spindle Motor Current", en, *The International Journal of Advanced Manufacturing Technology*, vol. 15, no. 4, pp. 238–243, Apr. 1999, ISSN: 1433-3015. DOI: 10.1007/s001700050062. [Online]. Available: https://doi.org/10.1007/s001700050062.

[14] S. H. Kia, A. M. Mabwe, H. Henao, and G.-A. Capolino, "Wavelet Based Instantaneous Power Analysis for Induction Machine Fault Diagnosis", in *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, ISSN: 1553-572X, Nov. 2006, pp. 1229–1234. DOI: 10.1109/IECON.2006.347461.

[15] L. Guo, D. Rivero, J. A. Seoane, and A. Pazos, "Classification of EEG signals using relative wavelet energy and artificial neural networks", Jan. 2009, pp. 177–184. DOI: 10.1145/1543834.1543860.

[16] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane, "Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming", en, in *Artificial Intelligence in Design '96*, J. S. Gero and F. Sudweeks, Eds., Dordrecht: Springer Netherlands, 1996, pp. 151–170, ISBN: 978-94-009-0279-4. DOI: 10.1007/978-94-009-0279-4_9. [Online]. Available: https://doi.org/10.1007/978-94-009-0279-4_9.

[17] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, en, ser. Springer Texts in Statistics. New York, NY: Springer New York, 2013, vol. 103, ISBN: 978-1-4614-7137-0 978-1-4614-7138-7. DOI: 10.1007/978-1-4614-7138-7. [Online]. Available: http://link.springer.com/10.1007/978-1-4614-7138-7.

[18] I. Guyon and A. Elisseeff, "An Introduction to Variable and Feature Selection", *Journal of Machine Learning Research*, vol. 3, no. Mar, pp. 1157–1182, 2003, ISSN: ISSN 1533-7928. [Online]. Available: http://www.jmlr.org/papers/v3/guyon03a.html.

[19] M. Christ, A. W. Kempa-Liehr, and M. Feindt, "Distributed and parallel time series feature extraction for industrial big data applications", *arXiv:1610.07717 [cs]*, May 2017, arXiv: 1610.07717. [Online]. Available: http://arxiv.org/abs/1610.07717.

[20] M. Christ, N. Braun, J. Neuffer, and A. W. Kempa-Liehr, "Time Series FeatuRe Extraction on basis of Scalable Hypothesis tests (tsfresh – A Python package)", en, *Neurocomputing*, vol. 307, pp. 72–77, Sep. 2018, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2018.03.067. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231218304843.

[21] H. He and E. A. Garcia, "Learning from Imbalanced Data", *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009, ISSN: 2326-3865. DOI: 10.1109/TKDE.2008.239.

[22] R. R. Picard and R. D. Cook, "Cross-validation of regression models", *Journal of the American Statistical Association*, vol. 79, no. 387, pp. 575–583, 1984, Publisher: Taylor & Francis Group.

[23] D. D. Lewis, R. E. Schapire, J. P. Callan, and R. Papka, "Training algorithms for linear text classifiers", in *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, 1996, pp. 298–306.

[24] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot, "Variable selection using random forests", *Pattern Recognition Letters*, vol. 31, no. 14, pp. 2225–2236, Oct. 15, 2010, ISSN: 0167-8655. DOI: 10.1016/j.patrec.2010.03.014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865510000954.

[25] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting", en, *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, Aug. 1997, ISSN: 0022-0000. DOI: 10.1006/jcss.1997.1504. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002200009791504X.

[26] R. Caruana and A. Niculescu-Mizil, "An Empirical Comparison of Supervised Learning Algorithms", in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06, event-place: Pittsburgh, Pennsylvania, USA, New York, NY, USA: ACM, 2006, pp. 161–168, ISBN: 978-1-59593-383-6. DOI: 10.1145/1143844.1143865. [Online]. Available: http://doi.acm.org/10.1145/1143844.1143865.

[27] *Dmlc/xgboost*, en. [Online]. Available: https://github.com/dmlc/xgboost.

[28] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System", ACM, Aug. 2016, pp. 785–794, ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. [Online]. Available: http://dl.acm.org/citation.cfm?id=2939672.2939785.

[29] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A Highly Efficient Gradient Boosting Decision Tree", in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 3146–3154. [Online]. Available: http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf.

[30] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain", *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958, ISSN: 1939-1471(Electronic),0033-295X(Print). DOI: 10.1037/h0042519.

[31] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing", *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995, _eprint: https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/j.2517-6161.1995.tb02031.x, ISSN: 2517-6161. DOI: 10.1111/j.2517-6161.1995.tb02031.x. [Online]. Available:

https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/j.2517-6161.1995.tb02031.x.

[32] R. K. Mohanta, T. R. Chelliah, S. Allamsetty, A. Akula, and R. Ghosh, "Sources of vibration and their treatment in hydro power stations-A review", en, *Engineering Science and Technology, an International Journal*, vol. 20, no. 2, pp. 637–648, Apr. 2017, ISSN: 2215-0986. DOI: 10.1016/j.jestch.2016.11.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2215098616304815.

[33] P. Kumar and R. P. Saini, "Study of cavitation in hydro turbines—A review", en, *Renewable and Sustainable Energy Reviews*, vol. 14, no. 1, pp. 374–383, Jan. 2010, ISSN: 1364-0321. DOI: 10.1016/j.rser.2009.07.024. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364032109001609.

Appendix A

Available data

A summary of the available measurement series is presented in Table A.1.

Table A.1: The measurement series available as well as the machine condition. Second and third columns are the number of turns short-circuited in the field windings of poles 13 and 6, respectively. Each test condition was sampled simultaneously with two sensors.

Case	Pole 13	Pole 6	Sample rate	Loading	Comment
1	0 turns	0 turns	10 kHz	No-load	Cold machine
2	0 turns	0 turns	50 kHz	No-load	Cold machine
3	0 turns	0 turns	10 kHz	Full-load	Cold machine
4	0 turns	0 turns	50 kHz	Full-load	Cold machine
5	10 turns	0 turns	10 kHz	Full-load	-
6	10 turns	0 turns	50 kHz	Full-load	-
7	10 turns	0 turns	10 kHz	No-load	-
8	10 turns	0 turns	50 kHz	No-load	-
9	7 turns	0 turns	10 kHz	No-load	-
10	7 turns	0 turns	50 kHz	No-load	-
11	7 turns	0 turns	10 kHz	Full-load	-
12	7 turns	0 turns	50 kHz	Full-load	-
13	3 turns	0 turns	10 kHz	Full-load	-
14	3 turns	0 turns	50 kHz	Full-load	-
15	3 turns	0 turns	10 kHz	No-load	-
16	3 turns	0 turns	50 kHz	No-load	-
17	2 turns	0 turns	10 kHz	No-load	-
18	2 turns	0 turns	50 kHz	No-load	-
19	2 turns	0 turns	10 kHz	Full-load	-
20	2 turns	0 turns	50 kHz	Full-load	-
21	1 turns	0 turns	10 kHz	Full-load	-
22	1 turns	0 turns	50 kHz	Full-load	-
23	1 turns	0 turns	10 kHz	No-load	-
24	1 turns	0 turns	50 kHz	No-load	-
25	10 turns	10 turns	10 kHz	No-load	-
26	10 turns	10 turns	50 kHz	No-load	-
27	10 turns	10 turns	10 kHz	Full-load	-

Continued on next page

Case	Pole 13	Pole 6	Sample rate	Loading	Comment
28	10 turns	10 turns	50 kHz	Full-load	-
29	10 turns	3 turns	10 kHz	Full-load	-
30	10 turns	3 turns	50 kHz	Full-load	-
31	10 turns	3 turns	10 kHz	No-load	-
32	10 turns	3 turns	50 kHz	No-load	-
33	3 turns	10 turns	10 kHz	No-load	-
34	3 turns	10 turns	50 kHz	No-load	-
35	3 turns	10 turns	10 kHz	Full-load	-
36	3 turns	10 turns	50 kHz	Full-load	-
37	0 turns	0 turns	10 kHz	Full-load	Warm machine
38	0 turns	0 turns	50 kHz	Full-load	Warm machine
39	0 turns	0 turns	10 kHz	No-load	Warm machine
40	0 turns	0 turns	50 kHz	No-load	Warm machine
41	0 turns	0 turns	10 kHz	No-load	Reversed direction of rotation
42	0 turns	0 turns	50 kHz	No-load	Reversed direction of rotation
43	0 turns	0 turns	10 kHz	Full-load	Reversed direction of rotation
44	0 turns	0 turns	50 kHz	Full-load	Reversed direction of rotation
45	0 turns	0 turns	10 kHz	Full-load	Reversed polarity of excitation
46	0 turns	0 turns	50 kHz	Full-load	Reversed polarity of excitation
47	0 turns	0 turns	10 kHz	No-load	Reversed polarity of excitation
48	0 turns	0 turns	50 kHz	No-load	Reversed polarity of excitation

Appendix B

Implementation

This appendix contains the code implementation as written. Imports and dependencies are not included. The complete implementation with accompanying scripts and dependencies is also available in a more accessible form at GitHub, access to which can be given upon request. Note that these are Jupyter Notebook files and not ".py", so Jupyter Notebook in addition to Python is necessary to run them. In the interest of transparency and reproducibility, the entire implementation is included.

B.1 Data management

The following code loads a data set from several CSV files into a data frame, labels and groups each measurement series.

```
import pandas as pd
import numpy as np
import math

import math

def add_CSV(CSV_dataframe, label, description, file_path, channels_of_interest,OSSid):
    # Appends a measurement series extracted from a CSV onto a provided DataFrame.
    # label is the number of short circuited turns.
    # description is a description of the machine state.
    # channels_of_interest are which columns in the CSVs that contain measurements.


    # Finding the sampling frequency.
    description_file = pd.read_csv('{0}.csv'.format(file_path),sep=':',header=None,
    nrows=3)
    sampling_freq = int(math.ceil((1/(float(description_file.iloc[1,1])))))) # All of
    this is needed to read the text and convert it into an integer.

    # Extracting sample series.
    measurements = pd.read_csv('{0}.Wfm.csv'.format(file_path),sep=';',header=None)
    CSV_dataframe = CSV_dataframe.append({'measurements':measurements,
                                          'channels_of_interest':channels_of_interest,
                                          'condition':label,
                                          'description':description,
                                          'sampling_freq':sampling_freq,
                                          'OSSid':OSSid},ignore_index=True)

    return CSV_dataframe


data_set_file_path = 'Data sets/ingrid15052019/' # Location of measurement CSV's in file
     hierarchy.
file_paths = ['{0}{1}'.format(data_set_file_path, i) for i in range(2,49+1)] # Generate
    all the file paths.
channels_of_interest = [3,4] # The columns in the CSVs that contain measurements.
df_CSVs = pd.DataFrame(columns=['measurements','channels_of_interest',
                                'condition', 'description','sampling_freq','OSSid'])

labels = [0,0,0,0,10,10,10,10,
          7,7,7,7,3,3,3,3,
          2,2,2,2,1,1,1,1,
          20,20,20,20,13,13,13,13,
          13,13,13,13,0,0,0,0,
          0,0,0,0,0,0,0,0] # The number of ITSC of each measurement series.
OSSids = [0,0,2,2,4,4,6,6,
          8,8,10,10,12,12,14,14,
          16,16,18,18,20,20,22,22,
          24,24,26,26,28,28,30,30,
          32,32,34,34,36,36,38,38,
          40,40,42,42,44,44,46,46]
descriptions = []

for label in labels:
    if label == 0:
        descriptions.append('Healthy')
    else:
        descriptions.append('Faulty')


for i in range(len(file_paths)):
    df_CSVs = add_CSV(df_CSVs, labels[i], descriptions[i], file_paths[i],
    channels_of_interest, OSSids[i])
```

B.2 Data segmentation

The following code contains functions which segment the OSSes into RSSes in the manner described in Section 3.2. Each original sample series is partitioned into 7-period-long reduced sample series and stored along with their conditions, sampling frequency and other identifiers.

```
def is_valid_crossing(position, data_series, validation_length):
    # Checks that the series of samples after "position" are all positive.
    is_valid = True
    validation_position = position
    sum_of_samples = 0.0
    for i in range(validation_length):
        current_sample = data_series[position + i]
        sum_of_samples += current_sample
        if ((current_sample < 0) or (sum_of_samples < 0)):
            validation_position += i
            is_valid = False
            break

    return is_valid, validation_position


def find_zero_crossing(search_from, data_series,
                       sampling_period, validation_ratio):
    # Finds and returns the first rising zero crossing in the signal after ...
    # the time "search_from", using zero-crossing.
    # "data_series" is the list structure with the signal.
    # "sampling_period" is the signal's sampling period.
    # "validation_ratio" is the length of the validation check, given in ...
    # "synchronous_periods". Should be between 0.1 and 0.35.

    current_pos = int(search_from/sampling_period)


    while (data_series[current_pos] > 0):  # Fast forward to a lightly zero-crossing
    point.
        current_pos += 1
    while (data_series[current_pos] < 0):
        current_pos += 1
    current_pos -= 10

    validated_crossing = False  # True if "current_pos" is a validated zero-crossing,
    False otherwise.
    validation_length = int((0.02/sampling_period)*validation_ratio)

    while not validated_crossing:  # Iterate through the samples looking for a zero-
    crossing.
        has_crossed_upwards = False  # True if "current_pos" has just risen above zero,
    reset every cycle.

        while not has_crossed_upwards:
            if (data_series[current_pos] > 0):
                has_crossed_upwards = True
            else:
                current_pos += 1

        validated_crossing, current_pos = is_valid_crossing(current_pos,
                                                            data_series,
                                                            validation_length)

        if not validated_crossing:
            current_pos += 1

    crossing_int = current_pos
    crossing_time = crossing_int*sampling_period

    return crossing_time, crossing_int
```

```
58
59
60 def return_period(data_series, search_from=0,
61                     sampling_freq=50000, synchronous_periods=7):
62     # synchronous_periods refer to the electrical periods. The mechanical rotation
       period of the machine is 7.1333 Hz. We need at least 50/7.13 = 7 to capture an
       entire period of rotation.
63     # data_series is a list of values.
64     time_window = synchronous_periods*0.02 # The number of electrical periods within the
        window we're looking at.
65     sampling_period = 1/sampling_freq
66     start_time, start_int = find_zero_crossing(search_from, data_series,
67                                                sampling_period, 0.1)
68     end_time, end_int = find_zero_crossing(start_time + time_window - 0.005,
69                                            data_series, sampling_period, 0.1)
70
71     data_snippet = data_series[start_int:end_int]
72
73     time_series = np.linspace(0, end_time - start_time, len(data_snippet)).tolist()
74     return (start_time, end_time), (start_int, end_int), data_snippet, time_series
75
76
77 def partition_sample_series(raw_data, OSSid, condition,
78                             data_frame=False, description=None,
79                             sampling_freq=50000, synchronous_periods=7,
80                             channels_of_interest=[1], skip_one=True,
81                             RSS_per_CSV = -1):
82     # Partitions a multichannel dataframe containing time-series data into...
83     # several several smaller times-series of synchronous_periods length.
84     # Continues until the end of the OSS or RSS_per_CSV.
85     # raw_data is the imported sample data.
86     # OSSid is the sample series number.
87     # data_frame is the data structure the data is added to.
88     # skip_one is an option to skip one electrical period before capturing next sample.
89     # synchronous_periods is the number of electrical periods in each RSS.
90     # RSS_per_CSV is the number of RSS to extract from each OSS. If ...
91     # set to -1, it will extract until end of OSS.
92
93     number_of_samples = raw_data.shape[0]
94     RSSid = 0
95     for channel in channels_of_interest:
96
97         if (RSS_per_CSV < 0):
98             number_of_RSS = float('-inf')
99         else:
100            number_of_RSS = 0
101
102        end_int = 0
103        if channel == 3:
104            data_series = [-i for i in raw_data.iloc[:,channel]] # This is due to the
       sensors being mounted with opposite polatity in the machine.
105        else:
106            data_series = [i for i in raw_data.iloc[:,channel]]
107        while (end_int + (synchronous_periods + 0.5)*0.02*sampling_freq <
       number_of_samples):
108            #print('number_of_RSS is {0}, RSS_per_CSV is {1}, channel is {2}'.format(
       number_of_RSS, RSS_per_CSV, channel))
109
110            end_time = end_int/sampling_freq
111            start_end_time, start_end_int, data_snippet, time_series = return_period(
       data_series, search_from=end_time,
112
       sampling_freq=sampling_freq,
113
       synchronous_periods=synchronous_periods)
114            if skip_one:
115                end_int = start_end_int[1] + int(sampling_freq*0.015) # The 3/4 period
       added shifts the sampling window to make fault conditions appear at new locations in
        each consecutive RSS.
116            else:
117                end_int = start_end_int[1] - int(sampling_freq*0.003)
118
119            samples = pd.DataFrame({'time':time_series,'data':data_snippet})
```

```
120
121             data_frame = data_frame.append({'OSSid':OSSid,
122                                             'RSSid':RSSid,
123                                             'sampling_freq':sampling_freq,
124                                             'condition':condition,
125                                             'description':description,
126                                             'samples':samples}, ignore_index=True)
127             RSSid += 1
128             number_of_RSS += 1
129             if (number_of_RSS >= RSS_per_CSV): # Stop creating RSS when RSS_per_CSV is
    reached.
130                 break
131         OSSid += 1
132     return data_frame, OSSid
```

The functions above were applied in the following manner:

```
1  RSS_data_frame = pd.DataFrame(columns = [
2      'OSSid', 'RSSid',
3      'sampling_freq',
4      'condition',
5      'description',
6      'samples'
7  ])
8
9  OSSid = 0
10 for i, CSV in df_CSVs.iterrows(): # For every OSS extracted from the CSV files.
11     # Partition into RSSes.
12     RSS_data_frame, OSSid = partition_sample_series(
13         CSV['measurements'],
14         CSV['OSSid'],
15         CSV['condition'],
16         data_frame=RSS_data_frame,
17         description=CSV['description'],
18         channels_of_interest=CSV['channels_of_interest'],
19         sampling_freq=CSV['sampling_freq'],
20         synchronous_periods=7,
21         RSS_per_CSV = -1
22     )
```

B.3 Feature extraction

The code for feature extraction is included below.

B.3.1 FFT

The following code applies an FFT to each RSS and stores the results as described in Section 3.3.1.

```
from scipy.fftpack import fft
from scipy import signal
from cmath import exp, pi


def FFT(data_series, sampling_freq, window='hann', plot = False):
    # Performs an FFT of the data_series.
    # window is the window function name. Reference: https://docs.scipy.org/doc/scipy/reference/signal.windows.html#module-scipy.signal.windows
    samples = len(data_series)

    if window:
        w = signal.get_window(window, samples)
    else:
        w = np.ones(samples)
    FFT_raw = fft(data_series*w)

    FFT_transform = np.abs(FFT_raw[:samples//2])/max(np.abs(FFT_raw[:samples//2]))
    FFT_frequencies = np.linspace(
        0.0,
        sampling_freq/(2.0),
        int(samples/2)
    )

    return FFT_transform, FFT_frequencies

def compute_all_FFTs(df, padded_sample_length=-1):
    # Computes FFTs for every time series in the given data frame.
    # df is the data frame containing the time series.
    # padded_sample_length is the length to pad the signal to, ...
    # this is done due to the FFT algorithm returning frequency...
    # bins according to the length of the input signal.

    start = time.time()
    FFTs = []

    for index, sample in df.iterrows(): # Iterate through the data frame and apply the FFT to each RSS.
        # sample_length = int(0.14*sample['sampling_freq'])
        if padded_sample_length > 0:
            #print(sample_length)
            pad_length = padded_sample_length-len(sample['samples']['data'])
            series = (np.pad(sample['samples']['data'], (0,pad_length),'constant'))
        else:
            series = sample['samples']['data']
        FFTtransform, FFTfrequencies = FFT(series, sample['sampling_freq'])
        FFT_trans_freq = {
            'frequencies':FFTfrequencies,
            'transform':FFTtransform
        }
        FFTs.append(FFT_trans_freq)
    df['FFT'] = FFTs

    end = time.time()
    print("- FFT run time is %.4f seconds." % (end - start))
    print("- Equivalent to %.2f seconds run time per sample." % ((end - start)/df.shape[0]))
```

```
55
56 def FFT_selected_frequencies(time_series, sampling_freq, frequencies, window='hann'):
57     # This is an implementation of the DFT. It is less ...
58     # efficient than the FFT, but allows choice of ...
59     # frequencies of interest and does not necessitate ...
60     # padding of the signal.
61     # time_series is the signal to be analysed.
62     # sampling_freq is the sampline frequency of the signal
63     # frequecies is a list of frequencies of interest.
64     # window is the window
65
66     transform = []
67     N = len(time_series)
68     indices = [i for i in range(N)]
69
70     if window:
71         w = signal.get_window(window, N)
72     else:
73         w = np.ones(N)
74
75     windowed_time_series = time_series*w
76
77     for frequency in frequencies:
78         one_freq = 0 + 0j
79         for n in indices:
80             k = windowed_time_series[n]
81             i = exp(-2j*pi*n*(frequency/sampling_freq))
82             one_freq += k*i
83
84         transform.append(abs(one_freq))
85
86     transform = transform/max(transform) # Normalisation
87
88     return transform, frequencies
```

The functions above were applied to the output of the segmentation functions in the following manner, note that the padded sample length is chosen so that the FFT returns frequencies at exact 50/7 intervals:

```
1 compute_all_FFTs(RSS_data_frame, padded_sample_length=7002)
```

B.3.2 Discrete wavelet transform wavelet energies

The following code applies an DWT to each RSS described in Section 3.3.2. Wavelet energies are extracted from each DWT and added as features.

```
1  import pylab
2  import pywt
3  import math
4  from scipy.signal import resample
5
6  def IWE(coeffs): # Instantaneous wavelet energy.
7      # coeffs is a DWT.
8
9      num_decomp = len(coeffs) # The number of decompositions.
10     energy_spec = [0. for i in range(num_decomp)]
11
12     for level in range(num_decomp): # For every decomposition level.
13         num_coeffs = len(coeffs[level]) # The number of coefficients at the current
       level.
14
15         for coeff in range(0,num_coeffs-1):
16             energy_spec[level] = energy_spec[level] + (coeffs[level][coeff])**2
17
18         energy_spec[level] = math.log10(energy_spec[level]/num_coeffs)
```

```
19
20      return energy_spec
21
22  def TWE(coeffs): # Teager wavelet energy.
23      # coeffs is a DWT.
24
25      num_decomp = len(coeffs) #The number of decompositions.
26      energy_spec = [0. for i in range(num_decomp)]
27
28      for level in range(0,num_decomp-1): # For every decomposition level.
29          num_coeffs = len(coeffs[level]) # The number of coefficients at the current
        level.
30
31          for coeff in range(1,num_coeffs-1):
32              energy_spec[level] = energy_spec[level] + abs((coeffs[level][coeff])**2 -
33                                      coeffs[level][coeff-1]*coeffs[level][coeff+1])
34
35          energy_spec[level] = math.log10(energy_spec[level]/num_coeffs)
36      return energy_spec
37
38  def HWE(coeffs): # Hierarchical wavelet energy.
39      # coeffs is a DWT.
40
41      num_decomp = len(coeffs) # The number of decompositions.
42      energy_spec = [0. for i in range(num_decomp)]
43
44      for level in range(0,num_decomp-1): # For every decomposition level.
45          num_coeffs = len(coeffs[level]) # The number of coefficients at the current
        level.
46
47          if level == 0:
48              for coeff in range(0,num_coeffs-1):
49                  energy_spec[level] = energy_spec[level] + (coeffs[level][coeff])**2
50          else:
51              last_num_coeffs = len(coeffs[level-1])
52              for coeff in range(int((num_coeffs-last_num_coeffs)/2-1),
53                                 int((num_coeffs+last_num_coeffs)/2-1)):
54                  energy_spec[level] = energy_spec[level] + (coeffs[level][coeff])**2
55
56          energy_spec[level] = math.log10(energy_spec[level]/num_coeffs)
57      return energy_spec
58
59  def RWE(coeffs): # Relative wavelet energy.
60      # coeffs is a DWT.
61
62      num_decomp = len(coeffs) # The number of decompositions.
63      energy_spec = [0. for i in range(num_decomp)]
64
65      for level in range(0,num_decomp-1): # For every decomposition level.
66          num_coeffs = len(coeffs[level]) # The number of coefficients at the current
        level.
67
68          for coeff in range(0,num_coeffs-1):
69              energy_spec[level] = energy_spec[level] + (coeffs[level][coeff])**2
70
71          energy_spec[level] = energy_spec[level]/num_coeffs
72      total_energy = sum(energy_spec)
73      return [i/total_energy for i in energy_spec]
74
75  def compute_wavelet_energies(time_series, wavelet='haar', level=12):
76      # Computes the wavelet energies of the time series DWT.
77      # time_series is the measurement series to be analysed.
78      # wavelet is the selected wavelet, by default 'haar'.
79      # level is the number of decomposition levels.
80
81      wavelet_energies = {'IWE':[],'TWE':[],'HWE':[],'RWE':[]} # Empty dictionary to hold
        the wavelet energies.
82
83      # Since the measurements are taken of a stationary system, ...
84      # a single measurement series can be repeated in succession.
85      # This is effectively padding with the measurement itself.
86      # This is necessary for the proper functioning of the algorithm.
87      DWT_coeffs = pywt.wavedec(time_series, wavelet, mode='periodic', level=level, axis
```

```
    =-1)
88
89      wavelet_energies['IWE'] = IWE(DWT_coeffs) # Each function returns a list of the
        energies of each level.
90      wavelet_energies['TWE'] = TWE(DWT_coeffs)
91      wavelet_energies['HWE'] = HWE(DWT_coeffs)
92      wavelet_energies['RWE'] = RWE(DWT_coeffs)
93
94      for energy_type in wavelet_energies:
95          wavelet_energies[energy_type].reverse()
96
97      return wavelet_energies
98
99  def compute_all_wavelet_energies(df, wavelet='haar', level=12):
100     # Iterates through the data frame to compute wavelet energies...
101     # for every time series.
102     # df is the dataframe containing the time_series.
103     # wavelet is the selected wavelet, by default 'haar'.
104     # level is the number of decomposition levels.
105
106     start = time.time()
107     wavelet_energies = []
108
109     num_RSS = df.shape[0]
110     print_interval = int(num_RSS/100)
111
112     for index, sample in df.iterrows(): # Iterate through the data frame and extract
        wavelet enegies from each RSS.
113
114         if (sample['sampling_freq'] == 10000):
115             # The 10k sampling frequency time series are upsampled to 50k...
116             # to facilitate reusing the code for both sampling frequencies...
117             # and so that the wavelet energies represent the same frequency...
118             # ranges.
119             sample_length = 5*len(sample['samples']['data'])
120             time_series = resample(sample['samples']['data'], sample_length) # Upsampled
         by a factor of 5.
121             time_series = np.tile(time_series, 4) # Repeated 4 times, equivalent to 4
        mechanical periods.
122         else:
123             time_series = np.tile(sample['samples']['data'], 4) # Repeated 4 times,
        equivalent to 4 mechanical periods.
124
125         wavelet_energies.append(compute_wavelet_energies(time_series,
126                                                          wavelet=wavelet,
127                                                          level=level))
128         if (index%print_interval == 0 or (index+1)==num_RSS): # To not spam print
        statements.
129             # Print the progress
130             num_length = len(str(num_RSS)) # Formatting aid
131             print('\rCalculation is {0:6.2f}% complete. Wavelet energies calculated for
        RSS number {1:{3}} out of {2}.'.format( (((index+1)/num_RSS)*100), index+1, num_RSS,
         num_length), end='' ) # To see progress.
132
133
134     df['wavelet_energies'] = wavelet_energies
135     end = time.time()
136     print("- DWT run time is %.4f seconds." % (end - start))
137     print("- Equivalent to %.2f seconds run time per sample." % ((end - start)/df.shape
        [0]))
```

The functions above were applied to the output of the segmentation functions in the following manner:

```
1 compute_all_wavelet_energies(RSS_data_frame)
```

B.3.3 TSFRESH

The implementation of TSFRESH as described in Section 3.3.3 is shown below. Note that the majority of the code is related to properly formatting the output of the segmentation functions.

```
from tsfresh import extract_features, extract_relevant_features
from tsfresh.feature_extraction import ComprehensiveFCParameters, MinimalFCParameters,
    EfficientFCParameters
from scipy.signal import resample
from numpy import linspace


def return_data(df):
    # Iterates through the RSS data frame and yields a single time step...
    # consecutively.
    # df is the RSS data frame.


    identity = 0
    for index, sample in df.iterrows():

        single_sample = pd.DataFrame(columns = ['id', 'time', 'flux'])
        sample_length = len(sample['samples']['data'])
        if (sample['sampling_freq'] == 50000):
            # The 50k sampling frequency time series are downsampled to 10k...
            # to facilitate reusing the code for both sampling frequencies...
            # and so that the wavelet energies represent the same frequency...
            # ranges.
            downsampled_length = int(sample_length/5)
            single_sample['time'] = linspace(sample['samples']['time'].iat[0], sample['
    samples']['time'].iat[-1], num=downsampled_length)
            single_sample['flux'] = resample(sample['samples']['data'],
    downsampled_length) # Downsampled by a factor of 5.
            single_sample['id'] = {'id':([identity]*downsampled_length)}['id']
        else:
            single_sample['time'] = sample['samples']['time']
            single_sample['flux'] = sample['samples']['data']
            single_sample['id'] = {'id':[identity]*sample_length}['id']

        identity += 1
        yield single_sample

def compute_TSFRESH(df, extraction_settings):

    if extraction_settings is None:
        extraction_settings = EfficientFCParameters()

    # Reformatting to fit with TSFRESH
    time_series_df = pd.concat(return_data(df), ignore_index=True, sort=False)


    start = time.time()
    if __name__ == "__main__":
        tsfresh_df = extract_features(time_series_df, column_id='id', column_sort='time'
    ,
                                      default_fc_parameters=extraction_settings,
                                      impute_function= None)

    end = time.time()
    print("- TSFRESH run time is %.4f seconds." % (end - start))
    print("- Equivalent to %.2f seconds run time per sample." % ((end - start)/df.shape
    [0]))

    return tsfresh_df
```

The functions above were applied to the output of the segmentation functions in the following manner:

```
settings = ComprehensiveFCParameters()
settings.pop('fft_coefficient', None) # Removing fourier calculations since FFTs of the most interesting frequencies are computed above.
settings.pop('fft_aggregated', None)
tsfresh_df = compute_TSFRESH(RSS_data_frame, settings)
```

B.4 Formatting

The outputs from feature extraction was passed into formatting where the FFT, TS-FRESH and DWT features are formatted into the same data structure. This was done as shown below.

```
def initialise_ML_data_frame(source_df, target_df, two_class=True):
    # This function initialises a data frame to hold the ML...
    # features that is suitably formatted for the purpose.
    # source_df is the source data frame.
    # target_df is the target data frame to be initialised.
    # two_class is true if a boolean classification is sought after...
    # and false if it will be used for multi class classification.

    column_labels = ['label','OSSid']
    sample_dict = {k:[ ] for k in column_labels} # Makes an empty dictionary.

    for index, sample in source_df.iterrows(): # Iterated through all the RSS with
    associated FFTs.

        # Adds the OSS id and target label.
        sample_dict['OSSid'].append(sample['OSSid']) # The OSS that the RSS was taken
    from.
        sample_dict['label'].append(sample['condition']) # The label, i.e. the fault
    condition.

    if two_class: # Make labels boolean is two class is true.
        for i in sample_dict['label']:
            if i > 0:
                i = 1

    for k in column_labels:
        target_df[k] = sample_dict[k]

def insert_FFTs(source_df, target_df, freq_range=500, two_class=True):
    # This function extracts the FFTs for every RSS from the source data...
    # frame and inserts them into a target data frame that is returned.
    # source_df is the source data frame.
    # target_df is the data frame the FFTs are inserted into. If none is...
    # provided, it will make one.

    column_labels = []

    if (target_df.empty):
        initialise_ML_data_frame(source_df, target_df, two_class=two_class)

    source_df.query('condition==1')

    for i in source_df.query('sampling_freq==50000').iloc[0]['FFT']['frequencies']:
        if (i > (freq_range+1) ):
            break
        column_labels.append("FFT__{0:.1f}_Hz".format(i))

    sample_dict = {k:[ ] for k in column_labels} # Makes an empty dictionary.

    for index, sample in source_df.iterrows(): # Iterates through all the RSS with
    associated FFTs.

        # Normalising the FFT if not done already.

        if sample['sampling_freq'] == 50000:
            RSS_FFT_values = [sample['FFT']['transform'][k] for k in range(len(
    column_labels))]
        else:
            RSS_FFT_values = [sample['FFT']['transform'][k*5] for k in range(len(
    column_labels))]
            # To only include the integer multiples of the mechanical...
            # frequency because the 10000 Hz samples have 5 times more freq bins in...
            # the relevant freq range than the 50k Hz samples.
```

```
58
59          norm = max(RSS_FFT_values) # Normalising the FFT.
60          RSS_FFT_values = RSS_FFT_values/norm # Normalising the FFT.
61
62          for i in range(len(column_labels)): # Adds the FFT values.
63
64              sample_dict[column_labels[i]].append(RSS_FFT_values[i])
65
66
67
68      for k in column_labels:
69          target_df[k] = sample_dict[k]
70
71  def insert_TSFRESH(source_df, target_df, ts_df, two_class=True):
72      # Inserts the TSFRESH features into a target data frame that is returned.
73      # TSFRESH_df is the source data frame.
74      # target_df is the data frame the features are inserted into. If none is...
75      # provided, it will make one.
76
77      if (target_df.empty):
78          initialise_ML_data_frame(source_df, target_df, two_class=two_class)
79
80      for col in ts_df.columns:
81          target_df[col] = ts_df[col]
82
83  def insert_DWT_energies(source_df, target_df, energy_types=[], levels=[], two_class=True
        ):
84      # Inserts the DWT features into a target data frame.
85      # source_df is the source data frame.
86      # target_df is the data frame the features are inserted into. If none is...
87      # provided, it will make one.
88      # energy_types is a list of strings of the types of wavelet energies to...
89      # include. The options are IWE, TWE, HWE and RWE.
90      # levels is a list of integers of the decomposition levels to include.
91
92      if (target_df.empty): # Initialises the target_df if it not already.
93          initialise_ML_data_frame(source_df, target_df, two_class=two_class)
94
95      if not energy_types: # Includes all wavelet energies if no subset is selected.
96          energy_types = list(source_df['wavelet_energies'][0].keys())
97      if not levels: # Includes all decomposition levels if no subset is selected.
98          levels = [i for i in range(len(source_df['wavelet_energies'][0][energy_types
        [0]]))]
99
100     column_labels = [] # Wavelet energy key list.
101     for energy_type in energy_types:
102         for level in levels:
103             column_labels.append('DWT__{0}{1}'.format(energy_type,level))
104
105
106     sample_dict = {k:[] for k in column_labels} # Makes an empty dictionary.
107
108     for index, sample in source_df.iterrows(): # Iterates through all the RSS with
        associated DWT energies.
109         for energy_type in energy_types:
110             for level in levels:
111                 keyword = 'DWT__{0}{1}'.format(energy_type, level)
112                 energy = sample['wavelet_energies'][energy_type][level]
113                 sample_dict[keyword].append(energy)
114
115     for k in column_labels:
116         target_df[k] = sample_dict[k]
```

The functions above were applied to the output of the segmentation functions in the following manner:

```
1 two_class = False # Whether its boolean or multiclass.
2 ML_df = pd.DataFrame() # Initialises an empty data frame.
3
4 insert_FFTs(RSS_data_frame, ML_df, freq_range=500, two_class=two_class) # Adds frequency
```

```
    magnitudes.

energy_types = ['IWE', 'TWE', 'HWE', 'RWE'] # The energy types to include.
levels = [i for i in range(0, 12+1)] # The decomposition levels to include.
insert_DWT_energies(RSS_data_frame, ML_df, energy_types=energy_types, levels=levels,
    two_class=two_class) # Adds DWT energies.

insert_TSFRESH(RSS_data_frame, ML_df, tsfresh_df, two_class=two_class) # Adds TSFRESH
    features.
```

B.5 Exploratory data analysis

In this appendix section, the code for the EDA is presented. Since the EDA is a series of smaller tests and procedures, it is shown below not as a monolithic block of code but rather as smaller code snippets with explanatory text associated.

B.5.1 Feature pruning

Uninformative features were removed. Any feature that was constant across all samples would not aid in classification and was removed to save training/classification time. The code is shown below.

```
# All features with zero variance are removed using a query statement.
ML_dataset_no_var = ML_dataset_raw.loc[:, ML_dataset_raw.var() != 0.0]

# Print the number of features removed, the total remaining number of features, and a
    list of the removed features.
print('{0} constant features were removed.'.format(ML_dataset_raw.shape[1] -
    ML_dataset_no_var.shape[1]))
print('Number of samples is {0}. Number of features is now {1}, down from {2}.'.format(
    ML_dataset_no_var.shape[0], ML_dataset_no_var.shape[1]-2, ML_dataset_raw.shape[1]-2)
    )
print('The features removed were:\n')
for removed_feature in ML_dataset_raw.columns[ML_dataset_raw.var() == 0.0]:
    print(removed_feature)
```

This yielded the output as shown below. The 50 Hz component of the FFT, the lowest levels of Teager, hierarchical and relative wavelet energies, and 54 of the features generated by TSFRESH were invariant and were thus removed from the data set.

```
58 constant features were removed.
Number of samples is 3552. Number of features is now 417, down from 475.
The features removed were:

FFT__50.0_Hz
DWT__TWE0
DWT__HWE0
DWT__RWE0
flux__augmented_dickey_fuller__autolag_"AIC"__attr_"usedlag"
flux__autocorrelation__lag_0
flux__large_standard_deviation__r_0.05
flux__large_standard_deviation__r_0.1
flux__large_standard_deviation__r_0.15000000000000002
flux__large_standard_deviation__r_0.2
flux__large_standard_deviation__r_0.25
flux__large_standard_deviation__r_0.30000000000000004
flux__large_standard_deviation__r_0.4
flux__large_standard_deviation__r_0.45
flux__large_standard_deviation__r_0.5
flux__large_standard_deviation__r_0.55
flux__large_standard_deviation__r_0.6000000000000001
flux__large_standard_deviation__r_0.65
flux__large_standard_deviation__r_0.7000000000000001
```

```
flux__large_standard_deviation__r_0.75
flux__large_standard_deviation__r_0.8
flux__large_standard_deviation__r_0.8500000000000001
flux__large_standard_deviation__r_0.9
flux__large_standard_deviation__r_0.9500000000000001
flux__number_crossing_m__m_-1
flux__number_crossing_m__m_1
flux__partial_autocorrelation__lag_0
flux__range_count__max_0__min_1000000000000.0
flux__ratio_beyond_r_sigma__r_10
flux__ratio_beyond_r_sigma__r_2
flux__ratio_beyond_r_sigma__r_2.5
flux__ratio_beyond_r_sigma__r_3
flux__ratio_beyond_r_sigma__r_5
flux__ratio_beyond_r_sigma__r_6
flux__ratio_beyond_r_sigma__r_7
flux__symmetry_looking__r_0.0
flux__symmetry_looking__r_0.05
flux__symmetry_looking__r_0.1
flux__symmetry_looking__r_0.15000000000000002
flux__symmetry_looking__r_0.2
flux__symmetry_looking__r_0.25
flux__symmetry_looking__r_0.30000000000000004
flux__symmetry_looking__r_0.35000000000000003
flux__symmetry_looking__r_0.4
flux__symmetry_looking__r_0.45
flux__symmetry_looking__r_0.5
flux__symmetry_looking__r_0.55
flux__symmetry_looking__r_0.6000000000000001
flux__symmetry_looking__r_0.65
flux__symmetry_looking__r_0.7000000000000001
flux__symmetry_looking__r_0.75
flux__symmetry_looking__r_0.8
flux__symmetry_looking__r_0.8500000000000001
flux__symmetry_looking__r_0.9
flux__symmetry_looking__r_0.9500000000000001
flux__value_count__value_-1
flux__value_count__value_1
flux__variance_larger_than_standard_deviation
```

B.5.2 Rough inspection

Below is the code for a rough inspection of the features to spot outliers.

```
# Calculates the mean, standard deviation, minimum, maximum and the 25%, median and 75%
    quartile of each feature across all samples.
description = ML_dataset_no_var.describe()

```

```
# Plots the mean, standard deviation, minimum, maximum and the 25%, median and 75%
    quartile of each feature across all samples.
for j in range(1, description.shape[0]):
    fig = plt.figure(1, figsize=(15,5))
    value_type = description.index.values[j]
    print(value_type)
    col = description.take([j])

    plt.plot([i for i in range(description.shape[1])], description.take([j]).transpose()
    , 'o', markersize=3)

    ax = fig.axes[0]
    ax.set_xlabel('Features')
    if value_type == 'mean':
        ylabel = 'Mean'
    elif value_type == 'std':
        ylabel = 'Standard deviation'
    else:
        ylabel = value_type

    ax.set_ylabel('{0}'.format(ylabel))
    plt.show()
    fig.savefig('Report_figures/method_EDA_feature_{0}.eps'.format(value_type),
                dpi=None, facecolor='w', edgecolor='w',
                orientation='portrait', papertype=None, format=None,
                transparent=False, bbox_inches=None, pad_inches=0.1,
                frameon=None, metadata=None)
```

B.5.3 Correlation

Below is the code for calculation and visualisation of Pearson's correlation. This code generated Figures 3.5 and 3.6.

Calculation of the correlations between all features, including the target values.

```
from sklearn.preprocessing import StandardScaler

# Scales all feature columns for standard mean and variance.
scaler = StandardScaler()
scaler.fit(ML_dataset_no_var)
ML_dataset_no_var_scaled = pd.DataFrame(scaler.transform(ML_dataset_no_var), columns=
    ML_dataset_no_var.columns)

# Calculates auto- and inter correlations of all the features and stores the results to
    a matrix
correlations = ML_dataset_no_var_scaled.iloc[:,:].corr()
```

Printing the features with the highest correlation to the target value.

```
# The correlation to the target value, i.e. the number of ITSC's.
corr_to_target = correlations.iloc[0, 1:ML_dataset_no_var.shape[1]]

# Prints the 20 features most correlated with the target.
print("20 features most correlated with the target :")
print(corr_to_target.abs().sort_values(ascending=False).head(20))
```

This gave an output as shown below.

```
20 features most correlated with the target :
DWT__TWE9                                                0.890734
DWT__TWE8                                                0.886363
DWT__IWE10                                               0.861306
DWT__RWE10                                               0.856935
```

```
DWT__RWE11                                                  0.826443
DWT__HWE10                                                  0.811841
DWT__TWE10                                                  0.810356
DWT__IWE11                                                  0.786659
DWT__TWE11                                                  0.556887
flux__longest_strike_above_mean                             0.549258
flux__approximate_entropy__m_2__r_0.7                       0.412482
DWT__TWE12                                                  0.407188
flux__longest_strike_below_mean                             0.395404
DWT__HWE11                                                  0.359168
flux__approximate_entropy__m_2__r_0.1                       0.300201
flux__linear_trend__attr_"pvalue"                           0.250902
flux__change_quantiles__f_agg_"var"__isabs_True__qh_1.0__ql_0.4  0.217445
flux__cid_ce__normalize_True                                0.211657
flux__time_reversal_asymmetry_statistic__lag_1              0.210167
flux__partial_autocorrelation__lag_1                        0.206603
Name: label, dtype: float64
```

The code below generates and saves a plot of feature correlation to the target value. This code generated Figure 3.7.

```
fig = plt.figure(1, figsize=(19,8))

plt.plot([i for i in range(1,ML_dataset_no_var_scaled.shape[1])], corr_to_target, 'o',
    markersize=6)
plt.axvline(x=72.5,color='r') # Demarking FFT and DWT portions.
plt.axvline(x=120.5,color='r')# Demarking DWT and TSFRESH portions.

ax = fig.axes[0]
ax.set_xlabel('Features')
ax.set_ylabel('Correlation')

plt.show()
fig.savefig('Report_figures/method_EDA_correlation.eps',
            dpi=None, facecolor='w', edgecolor='w',
            orientation='portrait', papertype=None, format=None,
            transparent=False, bbox_inches=None, pad_inches=0.1,
            frameon=None, metadata=None)
```

The code below generates and saves a plot of the correlation matrix. This code generated Figure 3.8.

```
fig = plt.figure(1, figsize=(19,16))
plt.matshow(correlations.abs(), fignum=1, vmax=1,vmin=0,cmap='Blues')
plt.axvline(x=70.5,color='r',lw=3) # Demarking FFT and DWT portions.
plt.axvline(x=120.5,color='r',lw=3)# Demarking DWT and TSFRESH portions.
plt.axhline(y=70.5,color='r',lw=3) # Demarking FFT and DWT portions.
plt.axhline(y=120.5,color='r',lw=3)# Demarking DWT and TSFRESH portions.
ax = fig.axes[0]
ax.set_ylabel('Features')
ax.set_title('Features', fontsize=20)

plt.colorbar()
plt.show()

fig.savefig('Report_figures/method_EDA_correlation_matrix.eps',
            dpi=None, facecolor='w', edgecolor='w',
```

```
            orientation='portrait', papertype=None, format=None,
            transparent=False, bbox_inches=None, pad_inches=0.1,
            frameon=None, metadata=None)
```

B.5.4 PCA and visualisation

Below is the code to compute a PCA and visualise its two first principal components.

Computing a PCA and printing the principal component variance.

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# The data set is first copied.
X_PCA_dataset = ML_dataset_no_var.copy()

# Seperating into features and targets
y_PCA_dataset = X_PCA_dataset.pop('label')

# Remove OSSid labelling.
X_PCA_dataset.pop('OSSid')

# Standardising the data set and putting it into a pandas DataFrame.
scaler = StandardScaler()
X_PCA_dataset = pd.DataFrame(scaler.fit_transform(X_PCA_dataset),
                             columns=X_PCA_dataset.columns,
                             index=X_PCA_dataset.index)

# Initialising PCA model that conserves 95% of the variance, which is correlated with
    information.
pca = PCA(.95)

# The PCA is fit on the data set.
pca.fit(X_PCA_dataset)

# Prints the variance ratios of the principal components generated.
print('The PCA made {} PCA components to encompass 95% of data set variance.'.format(pca
    .n_components_))
print('The explained variance ratios are:')
(pca.explained_variance_ratio_*100)
```

This produced the following output.

```
The PCA made 31 PCA components to encompass 95% of data set variance.
The explained variance ratios are:

array([36.20323537, 15.44963767,  9.78029715,  7.75185411,  4.90343163,
        3.30030116,  2.73598296,  1.95832954,  1.56094035,  1.37906103,
        1.1014838 ,  0.92292232,  0.82971246,  0.70810316,  0.63638894,
        0.60981849,  0.57711614,  0.48254183,  0.47041873,  0.43192233,
        0.40552671,  0.39160754,  0.35448156,  0.30238909,  0.29136357,
        0.27726808,  0.26090209,  0.25614311,  0.25024009,  0.24036458,
        0.23920348])
```

The code below transforms the feature set and plots the positions of each sample in the first and second principal component plane. This code generated Figure 3.9.

```
import matplotlib

# Plotting healthy and faulty samples along the first two principal components.

# The labels are changed from number of ITSCs to 1 for faulty and 0 for healthy.
for i in y_PCA_dataset:
    if i < 0:
        i= 1

# The data set is transformed into PCA space.
X_PCA_dataset_transformed = pca.transform(X_PCA_dataset)
x = X_PCA_dataset_transformed[:,0]
y = X_PCA_dataset_transformed[:,1]

# Each sample is shown as blue or red if it is healthy or faulty respectively.
label = y_PCA_dataset
colors = ['blue','red']

# The first and second principal components are plotted against each other.
fig = plt.figure(figsize=(13,12))
plt.scatter(x, y, c=label,  alpha = 1, cmap=matplotlib.colors.ListedColormap(colors))

ax = fig.axes[0]
ax.set_xlabel('First principal component')
ax.set_ylabel('Second principal component')

# The figure is saved.
fig.savefig('Report_figures/method_EDA_PCA.eps',
            dpi=None, facecolor='w', edgecolor='w',
            orientation='portrait', papertype=None, format=None,
            transparent=False, bbox_inches=None, pad_inches=0.1,
            frameon=None, metadata=None)
```

B.6 Feature selection

In this appendix section the code for the feature selection implementation is shown along with the code for pre-selection data set splitting.

```
# Before any feature selection, the data set must be split into training and test sets.
# This is to prevent that the features that we select are based on the test set, ...
# thereby including information from the test set into the training process.
# It is important that they are independent. If they are not, the results of the ...
# classification will be overly optimistic and unrealistic. In a production setting ...
# there would obviously not be future samples available when features are selected.

from sklearn.model_selection import GroupShuffleSplit

def split(dataset):
    # Splits the dataset into test and train.

    dataset.head()

    dataset_copy = dataset.copy()

    gss = GroupShuffleSplit(n_splits=1, test_size=0.15, train_size=None, random_state=1)
     # Split into training and testing sets, random_state is set to 1 so that the split
    is equal for each split.

    y = dataset_copy.pop('label')
    groups = dataset_copy.pop('OSSid')
    X = dataset_copy


    train_idx, test_idx = None, None

    for train_idx, test_idx in gss.split(X, y, groups):
        train_idx = train_idx
        test_idx  = test_idx

    X_train = X.iloc[train_idx]
    X_test  = X.iloc[test_idx]

    y_train = y.iloc[train_idx]
    y_test  = y.iloc[test_idx]

    group_train = groups.iloc[train_idx]
    group_test  = groups.iloc[test_idx]

    # Resetting indices to ease later work.
    X_train.reset_index(drop=True, inplace=True)
    X_test.reset_index(drop=True, inplace=True)
    y_train.reset_index(drop=True, inplace=True)
    y_test.reset_index(drop=True, inplace=True)
    group_train.reset_index(drop=True, inplace=True)
    group_test.reset_index(drop=True, inplace=True)


    return X_train, X_test, y_train, y_test, group_train, group_test
```

B.6.1 Random forest feature selection

Below is the code for feature selection using random forest.

```
from sklearn import ensemble
from sklearn.feature_selection import SelectFromModel

# The data set is first split before any feature selection.
X_train_RF, X_test_RF, y_train_RF, y_test_RF, group_train_RF, group_test_RF = split(
    ML_dataset)

```

```
# A random forest ensemble is trained on the data.
sel = SelectFromModel(ensemble.RandomForestClassifier(n_estimators = 1000, criterion='
    gini'))

# The features that were most usefull for the RF are selected.
selected_feats_RF = X_train_RF.columns[(sel.get_support())]

# The data set is pruned to only contain the selected features.
X_train_RF = X_train_RF.loc[:,selected_feats_RF]
X_test_RF = X_test_RF.loc[:,selected_feats_RF]

# The feature-selected data sets are stored in a tuple.
RF_data = (X_train_RF, X_test_RF,
           y_train_RF, y_test_RF,
           group_train_RF, group_test_RF)
```

B.6.2 TSFRESH

Below is the code for feature selection using the TSFRESH feature selection algorithm.

```
from tsfresh.feature_selection.relevance import calculate_relevance_table

# The data set is first split before any feature selection.
X_train_tsfresh, X_test_tsfresh, y_train_tsfresh, y_test_tsfresh, group_train_tsfresh,
    group_test_tsfresh = split(ML_dataset)

# The TSFRESH algorithm determines which features are relevant and which are not.
relevance_table = calculate_relevance_table(X_train_tsfresh,
                                            y_train_tsfresh,
                                            hypotheses_independent=False,
                                            ml_task='classification',
                                            fdr_level=0.05)

# A list of the relevant features is generated.
selected_feats_tsfresh = X_train_tsfresh.columns[relevance_table['relevant']]

# The relevant features are extracted from the non-feature-selected data sets.
X_train_tsfresh = X_train_tsfresh.loc[:,selected_feats_tsfresh]
X_test_tsfresh =  X_test_tsfresh.loc[:,selected_feats_tsfresh]

# The feature-selected data sets are stored in a tuple.
tsfresh_data = (X_train_tsfresh, X_test_tsfresh,
                y_train_tsfresh, y_test_tsfresh,
                group_train_tsfresh, group_test_tsfresh)
```

B.7 Fault presence detection

In this appendix section the code the classifier construction, evaluation and selection is shown along with the code for exporting results into LaTeX.

B.7.1 Boolean target values

In this code, the target labels are changed to only indicate fault presence instead of fault severity. Fault detection is binary classification and the severity of the fault needs to be changed to a Boolean indication of fault.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 20}) # Setting the font size of the plots to 20. Use
    plt.rcdefaults() to reset to default.

# Remove fault severity from target values.
# Target equals 1 indicates fault, and 0 indicates no fault.

def make_target_list_boolean(y_set):
    # Replace non-zero values with 1.

    for i in range(len(y_set)):
        if y_set[i] != 0:
            y_set[i] = 1
        else:
            y_set[i] = 0

def make_dataset_targets_boolean(dataset_tuple):
    # This function replaces any target value ...
    # other than 0 with 1.

    X_train, X_test, y_train, y_test, group_train, group_test = dataset_tuple

    make_target_list_boolean(y_train)
    make_target_list_boolean(y_test)
```

The functions above were applied to the output of the feature selection process in the following manner:

```
for dataset in datasets: # Go through every data set.
    make_dataset_targets_boolean(dataset)
```

B.7.2 Classifier cross-validation pipeline

The following code implements the classifier cross-validation training and evaluation used to evaluate classifiers in this thesis. It is created to be able to easily test several models and reduce code repetition.

```
import time
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score
from sklearn.model_selection import GroupKFold
from sklearn.preprocessing import StandardScaler

def single_set_run(X_train, X_test,
                   y_train, y_test,
                   classifier, results_dict):

    # Scaling the data so that it is more suited for SVM and KNN
```

```
    scaler = StandardScaler()
    scaler.fit(X_train)
    X_train_scaled, X_test_scaled = scaler.transform(X_train), scaler.transform(X_test)

    # Training model
    classifier.fit(X_train_scaled, y_train)

    # Making predictions
    start = time.time()
    y_pred = classifier.predict(X_test_scaled)
    end = time.time()
    total_prediction_time = (end - start)*1000
    prediction_time_per_sample = total_prediction_time/len(y_pred)


    # Rating predictions
    results_dict['Accuracy'].append(accuracy_score(y_test, y_pred))
    results_dict['Sensitivity'].append(recall_score(y_test, y_pred))
    results_dict['Precision'].append(precision_score(y_test, y_pred))
    results_dict['F1-score'].append(f1_score(y_test, y_pred))
    results_dict['ROC AUC'].append(roc_auc_score(y_test, y_pred))
    results_dict['Pred. time per sample'].append(prediction_time_per_sample)

    return results_dict


def train_and_test_classifiers(dataset_tuple, classifiers):
    # This function takes in training and test sets along with accompanying...
    # labels and groupings and trains several classifiers on them. It uses...
    # K-fold validation and prints out the average metrics for each classifier.
    # dataset_tuple is a tuple containing X_train, X_test, y_train, y_test, ...
    # group_train and group_test, in that order.
    # classifiers is a list of tuples with a string in the first position and...
    # a classifier or pipeline in the second position.


    # Extract data from tuple
    X_train, X_test, y_train, y_test, group_train, group_test = dataset_tuple


    results = pd.DataFrame()


    for classifier in classifiers:


        cv_results = {
            'Accuracy':[],
            'Sensitivity':[],
            'Precision':[],
            'F1-score':[],
            'ROC AUC':[],
            'Pred. time per sample':[]
        }

        # Do group k-fold split
        group_kfold = GroupKFold(n_splits=5)
        cv_split = group_kfold.split(X_train, y_train, group_train) # Creates several
    splits into training and validation sets.

        print('\nClassifier is {}'.format(classifier[0]))

        for train_index, val_index in cv_split:

            X_train_cv, X_val_cv = X_train.iloc[train_index], X_train.iloc[val_index]
            y_train_cv, y_val_cv = y_train[train_index], y_train[val_index]

            cv_results = single_set_run(
                X_train_cv, X_val_cv,
                y_train_cv, y_val_cv,
                classifier[1], cv_results
            )

```

```
        print('CV results are:')
        print(pd.DataFrame(cv_results).describe().loc['mean',:])

        test_results = {
            'Accuracy':[],
            'Sensitivity':[],
            'Precision':[],
            'F1-score':[],
            'ROC AUC':[],
            'Pred. time per sample':[]
        }

        test_results = single_set_run(X_train, X_test,
                                      y_train, y_test,
                                      classifier[1], test_results)


        print('Test results are:')
        print(test_results)

        results[classifier[0]] = [cv_results.copy(), test_results.copy()]

    return results
```

B.7.3 Feature data set comparison

The following code is the implementation of the feature data set comparison, here a collection of classifiers are compared across the individual data sets.

In the code below, all the non-optimised models are initialised.

```
from sklearn.linear_model import LogisticRegression, LassoCV
from sklearn import svm
from sklearn.neighbors import KNeighborsClassifier
from xgboost import XGBClassifier
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import StackingClassifier


k =  20 # The number of nearest neighbors
knn = KNeighborsClassifier(n_neighbors=k, weights='uniform')

logreg = LogisticRegression(max_iter= 10000)

SVM_rbf = svm.SVC(kernel = 'rbf', gamma ='auto', probability=True)

SVM_linear = svm.SVC(kernel = 'linear', probability=True)

pca_and_logreg = Pipeline(steps=[
    ('pca', PCA(.95)),
    ('logreg', LogisticRegression(max_iter= 10000))
])

pca_and_knn = Pipeline(steps=[
    ('pca', PCA(.95)),
    ('knn', KNeighborsClassifier(n_neighbors=k))
])

pca_and_svm_linear = Pipeline(steps=[
    ('pca', PCA(.95)),
    ('svm', svm.SVC(kernel = 'linear', probability=True))
])

pca_and_svm_rbf = Pipeline(steps=[
    ('pca', PCA(.95)),
    ('svm', svm.SVC(gamma = 'auto', probability=True))
```

```
])

xgbooster = XGBClassifier()

neural_net = Pipeline(steps=[
    ('NN',MLPClassifier(hidden_layer_sizes=(200,100,14), max_iter=1000, random_state=1))
])

stack = StackingClassifier([ # A stacking classifier combining
    ('Logistic Regression', logreg),
    ('Logistic Reg. with PCA', pca_and_logreg),
    ('KNN', knn),
    ('KNN with PCA', pca_and_knn),
    ('SVM (rbf)', SVM_rbf),
    ('SVM (rbf) with PCA', pca_and_svm_rbf),
    ('SVM (linear)', SVM_linear),
    ('SVM (linear) with PCA', pca_and_svm_linear),
    ('XGBoost', xgbooster),
    ('Neural net', neural_net)
])


classifiers = [
    ('Logistic Regression', logreg),
    ('Logistic Reg. with PCA', pca_and_logreg),
    ('KNN', knn),
    ('KNN with PCA', pca_and_knn),
    ('SVM (rbf)', SVM_rbf),
    ('SVM (rbf) with PCA', pca_and_svm_rbf),
    ('SVM (linear)', SVM_linear),
    ('SVM (linear) with PCA', pca_and_svm_linear),
    ('XGBoost', xgbooster),
    ('Neural net', neural_net),
    ('Stack', stack)
]
```

These classifiers were evaluated all the data sets in the following manner:

```
results = []

dataset_names = [
    'no_selection_data',
    'RF_data',
    'tsfresh_data'
]
for dataset_tuple in datasets:
    print('\n\nDataset is {}'.format(dataset_names.pop(0)))
    results.append(train_and_test_classifiers(dataset_tuple, classifiers))
```

The results from this returns are then visualised into box plots using the following code. This generated the plots in Figure 3.11.

```
# Taking the average of every classifier over the data set and plotting into a boxplot.
metrics = [
    'Accuracy',
    'Sensitivity',
    'Precision',
    'F1-score',
    'ROC AUC',
    'Pred. time per sample'
]

# Gather metrics for each data set.
dataset_metrics = { k:[] for k in metrics}
dataset_metrics['Dataset'] = []

dataset_names = ['A', 'B', 'C']

for metric in metrics: # For every metric.

```

```
    for ds_num in range(len(results)): # For every feature data set.
        r = results[ds_num]

        avg_list = []

        for classifier in r.columns: # For every classifier.
            avg_list.append(((r[classifier])[0])[metric])
        for num in avg_list:
            for ft in num:
                dataset_metrics[metric].append(ft)
                if metric == 'Accuracy':
                    dataset_metrics['Dataset'].append(dataset_names[ds_num])

plot_structure = pd.DataFrame(dataset_metrics)

import seaborn as sns
#sns.set(style="whitegrid")


for metric in metrics:
    fig = plt.figure(figsize=(16,10))

    sns.set(style="whitegrid", font_scale=2.5)
    fig.axes[0] = sns.boxplot(
        x=plot_structure["Dataset"],
        y=plot_structure[metric],
        color=sns.color_palette("Blues")[2],
        saturation=0.8
    )
    ax = fig.axes[0]
    ax.set_xlabel('Feature data set')
    ax.set_ylabel(metric)
    plt.ylim((0.35,1.02))


    plt.show()
    fig.savefig(
        'Report_figures/results_model_detection_dataset_{}.pdf'.format(metric),
        dpi=None, facecolor='w', edgecolor='w',
        orientation='portrait', papertype=None, format=None,
        transparent=False, bbox_inches='tight', pad_inches=0,
        frameon=None, metadata=None)
```

The numeric values of the results were aggregated and printed to Latex using the code in the following snippet. This generated Table 3.8.

```
metrics = [
    'Accuracy',
    'Sensitivity',
    'Precision',
    'F1-score',
    'ROC AUC'
]

result_columns = ['Data set', 'Classifier']
for k in metrics:
    result_columns.append(k)

dataset_names = ['A','B','C']
dataset_summary = []

dataset_summary = pd.DataFrame(columns=result_columns)

for i in range(len(results)): # For every feature data set.
    r = results[i]
    for classifier in r.columns: # For every classifier.
        append_dict = {k:[] for k in result_columns}
        append_dict['Data set'] = dataset_names[i]
        append_dict['Classifier'] = classifier
        for metric in metrics: # For every metric.
            append_dict[metric] = np.mean(((r[classifier])[0])[metric])
        dataset_summary = dataset_summary.append(append_dict, ignore_index=True)
```

```
27
28      # Adding the average classifier score for the feature data set.
29      avg_dict = {
30          'Data set': dataset_names[i],
31          'Classifier':'Average classifier score'
32      }
33      for k in metrics:
34          avg_dict[k] = np.mean(dataset_summary[k].iloc[-len(r.columns):])
35      dataset_summary = dataset_summary.append(avg_dict, ignore_index=True)
36
37  # Showing a summary of training results for each feature data set.
38  print(dataset_summary)
39
40  # Printing results to latex format for inclusion in the thesis.
41  print_columns = [result_columns[k] for k in [0,1,3,4,6]]
42  latex_print = dataset_summary.to_latex(
43      columns=print_columns,
44      label='tab:label',
45      float_format="%.4f",
46      multirow=True,
47      escape=False,
48      index=False,
49      header=['\\textbf{{{0}}}'.format(k) for k in print_columns]
50  )
51
52  print(latex_print)
```

B.7.4 Hyper-parameter optimisation

To select a classifier, each one is optimised using grid search CV. The implementation is shown in this section.

In the script below, each classifier is initialised and the hyperparameter search grid is defined and initialised.

```
1   # The initalising the models.
2   logreg = Pipeline([
3       ('scaler', StandardScaler()),
4       ('logreg', LogisticRegression(max_iter= 10000))
5   ])
6   knn = Pipeline([
7       ('scaler', StandardScaler()),
8       ('knn', KNeighborsClassifier())
9   ])
10  SVM_clf = Pipeline([
11      ('scaler', StandardScaler()),
12      ('svm', svm.SVC(probability=True))
13  ])
14  xgbooster = XGBClassifier(
15      objective='binary:logistic'
16  )
17  neural_net = Pipeline(steps=[
18      ('scaler', StandardScaler()),
19      ('neural_net',MLPClassifier(max_iter=1000, random_state=1))
20  ])
21
22  # Creating parameter grids for each model, the combinations...
23  # of hyperparameters to search through.
24  logreg_paramgrid = {
25      'logreg__C': np.power(10.0, np.arange(-10, 10,0.5)),
26      'logreg__penalty':['l1','l2','elasticnet', 'none']
27  }
28
29  knn_paramgrid = {
30      'knn__n_neighbors': np.arange(1,351,2) # Every odd integer between 1 and 51.
31  }
32
```

```
SVM_clf_paramgrid = {
    'svm__C': [0.1, 1, 10, 100, 1000],
    'svm__gamma': [1, 0.1, 0.01, 0.001, 0.0001],
    'svm__kernel': ['rbf', 'linear'] # Note that both rbf and linear kernels are
    included.
}

xgbooster_paramgrid = {
    'learning_rate': [0.01,0.2,0.3,0.5],
    'n_estimators': [100, 400, 700, 1000],
    'colsample_bytree': [0.8, 1],
    'max_depth': [3,10,15,25],
    'reg_alpha': [0.7, 1, 1.3],
    'reg_lambda': [0, 0.5, 1],
    'subsample': [0.6, 1]
}

neural_net_paramgrid = {
    'neural_net__hidden_layer_sizes':[(50,25,3),(100,50,7),(200,100,14),(300,150,21)],
    'neural_net__activation':['identity', 'logistic', 'tanh', 'relu'],
    'neural_net__batch_size':[200,133,66,32],
    'neural_net__max_iter':[200,500,1000,1200]
}


# All the classifiers are stored in a list along with a descriptive...
# label and their search grid.
classifiers = [
    ('Logistic Regression', logreg, logreg_paramgrid),
    ('KNN', knn, knn_paramgrid),
    ('SVM', SVM_clf, SVM_clf_paramgrid),
    ('XGBoost', xgbooster, xgbooster_paramgrid),
    ('Artificial Neural Network', neural_net, neural_net_paramgrid)
]
```

The grid search is then implemented as shown below. This returns a list with optimised classifiers and their hyper-parameters.

```
from sklearn.model_selection import GridSearchCV

dataset = datasets[2] # The TSFRESH feature data set.
X_train, X_test, y_train, y_test, group_train, group_test = dataset # Extract data from
    tuple

classifier_results = pd.DataFrame(columns=[
    'Classifier',
    'Grid search result'
])

for classifier_tuple in classifiers:

    description, classifier, paramgrid = classifier_tuple
    group_kfold = GroupKFold(n_splits=5)
    scoring_fit='f1_weighted'

    grid_search = GridSearchCV(
        estimator=classifier,
        param_grid=paramgrid,
        cv=group_kfold,
        n_jobs=-1,
        scoring=scoring_fit,
        verbose=2,
        refit=True,
        return_train_score=True
    )

    grid_search.fit(X_train, y=y_train, groups=group_train)
    appendix = {
        'Classifier':description,
        'Grid search result':grid_search
    }

```

```
    classifier_results = classifier_results.append(
        appendix,
        ignore_index=True
    )
```

The best hyper-parameters were printed to Latex using the code in the following snippet. This generated Table 3.10.

```
#Print out the best parameters for each classifier and structure into a table.
best_classifier_parameters = pd.DataFrame(columns=[
    'Classifier',
    'Hyperparameter',
    'Value'
])

for classifier in classifier_results.iterrows():

    name = classifier[1].loc['Classifier']
    parameters = classifier[1].loc['Grid search result'].best_params_

    for HP in list(parameters.keys()):
        best_classifier_parameters = best_classifier_parameters.append({'Classifier':
    name,'Hyperparameter':HP, 'Value':parameters[HP]},ignore_index=True)

column_headings = best_classifier_parameters.columns

latex_print = best_classifier_parameters.to_latex(
    columns=column_headings,
    label='tab:label',
    caption='Caption.',
    float_format=lambda x: '%.3f' % x,#'%.4f',
    escape=False,
    index=False,
    header=['\\textbf{{{0}}}'.format(k) for k in column_headings]
)

print(latex_print)
```

The optimised classifiers were evaluated and their results printed to Latex using the code in the following snippet. To compare the best solutions of the classifiers tested, they are evaluated on a 5-fold cross validation set. Note that the models used are the best models found in the grid search above. This generated Table 3.11.

```
optimised_classifiers = []

for classifier in classifier_results.iterrows():

    name = classifier[1].loc['Classifier']
    clf = classifier[1].loc['Grid search result'].best_estimator_
    optimised_classifiers.append((name,clf))

# Running all the classifiers to score them.
optimised_results = train_and_test_classifiers(datasets[2], optimised_classifiers)

# The metrics of interest
metrics = [
    'Accuracy',
    'Sensitivity',
    'Precision',
    'F1-score',
    'ROC AUC'
]

result_columns = ['Classifier'] # Creating the table columns
for k in metrics:
    result_columns.append(k)

optimised_classifier_summary = pd.DataFrame(columns=result_columns)

```

```
for classifier in optimised_results.columns: # For every classifier.
    append_dict = {k:[] for k in result_columns}
    append_dict['Classifier'] = classifier
    for metric in metrics: # For every metric.
        metric_list = []
        metric_list.append(np.mean(((optimised_results[classifier])[0])[metric]))
        append_dict[metric] = np.mean(metric_list)
    optimised_classifier_summary = optimised_classifier_summary.append(append_dict,
    ignore_index=True)

print(optimised_classifier_summary)

# Print to LaTeX for inclusion into the thesis.
print_columns = result_columns[:]
latex_print = optimised_classifier_summary.to_latex(
    columns=print_columns,
    label='tab:label',
    float_format="%.4f",
    escape=False,
    index=False,
    header=['\\textbf{{{0}}}'.format(k) for k in print_columns]
)

print(latex_print)
```

The features that are most important for predictions in XGBoost and Logistic Regression are extracted using the implementation below. This generated Tables 3.15 and 3.16.

```
feats_and_coeffs = pd.DataFrame()
feats_and_coeffs['XGBoost'] = (optimised_classifiers[3][1].feature_importances_).copy()
feats_and_coeffs['XG features'] = datasets[2][0].columns.copy()

XGBoost_coeffs = (feats_and_coeffs.sort_values('XGBoost',ascending=False, inplace=False
    )).reset_index(drop=True)

print(XGBoost_coeffs)

feats_and_coeffs = pd.DataFrame()
feats_and_coeffs['Log. Reg.'] = np.abs(optimised_classifiers[0][1][1].coef_[0]).copy()
feats_and_coeffs['LR features'] = datasets[2][0].columns.copy()

logistic_coeffs = (feats_and_coeffs.sort_values('Log. Reg.',ascending=False, inplace=
    False)).reset_index(drop=True)

print(logistic_coeffs)

feat_summary = pd.concat([logistic_coeffs, XGBoost_coeffs], axis=1)
feat_summary['Rank'] = [k+1 for k in feat_summary.index]
print(feat_summary)

feat_summary = feat_summary.iloc[:20,[4,1,3]]

print(feat_summary.to_latex(
    columns=feat_summary.columns,
    label='tab:label',
    caption='Caption.',
    float_format=lambda x: '%.3f' % x,#'%.4f',
    escape=False,
    index=False,
    header=['\\textbf{{{0}}}'.format(k) for k in feat_summary.columns]
))
```

B.7.5 Stacking classifiers

The stacking classifiers were initialised, trained and evaluated as shown in the implementation below. Note that *datasets*[2] is a reference to the TSFRESH feature data set.

```
optimised_classifiers = []

for classifier in classifier_results.iterrows():

    name = classifier[1].loc['Classifier']
    clf = classifier[1].loc['Grid search result'].best_estimator_


    if name != 'KNN': # Don't add KNN.
        optimised_classifiers.append((name,clf))
        print(name)
        print(clf)

from sklearn.ensemble import GradientBoostingClassifier, RandomForestClassifier

meta_gradient_boost = GradientBoostingClassifier()

meta_random_forest = RandomForestClassifier()

meta_neural_net = Pipeline(steps=[
    ('neural_net',MLPClassifier(hidden_layer_sizes=(15,7), max_iter=1000, random_state
    =1))
])

stack = [
    ('Stacking classifier LR', StackingClassifier(optimised_classifiers)),
    ('Stacking classifier ANN', StackingClassifier(optimised_classifiers,
    final_estimator=meta_neural_net)),
    ('Stacking classifier GBC', StackingClassifier(optimised_classifiers,
    final_estimator=meta_gradient_boost)),
    ('Stacking classifier RF', StackingClassifier(optimised_classifiers, final_estimator
    =meta_random_forest))
]

stack_performance = train_and_test_classifiers(datasets[2], stack)
```

The results from the stacking classifier evaluation were printed to Latex as shown below. This generated Table 3.12.

```
# The metrics of interest
metrics = [
    'Accuracy',
    'Sensitivity',
    'Precision',
    'F1-score',
    'ROC AUC'
]

result_columns = ['Classifier'] # Creating the table columns
for k in metrics:
    result_columns.append(k)

optimised_stacking_classifier_summary = pd.DataFrame(columns=result_columns)

for classifier in stack_performance.columns: # For every classifier.
    append_dict = {k:[] for k in result_columns}
    append_dict['Classifier'] = classifier
    for metric in metrics: # For every metric.
        metric_list = []
        metric_list.append(np.mean(((stack_performance[classifier])[0])[metric]))
        append_dict[metric] = np.mean(metric_list)
    optimised_stacking_classifier_summary = optimised_stacking_classifier_summary.append
    (append_dict, ignore_index=True)

print(optimised_stacking_classifier_summary)

# Print to LaTeX for inclusion into the thesis.
print_columns = result_columns[:]
latex_print = optimised_stacking_classifier_summary.to_latex(
    columns=print_columns,
    label='tab:label',
    float_format="%.4f",
```

```
    escape=False,
    index=False,
    header=['\\textbf{{{0}}}'.format(k) for k in print_columns]
)

print(latex_print)
```

To extract the coefficients of the base-classifiers the code below was used. This generated the information in Table 3.14.

```
print(stack[c][0]) # The name of the stacking classifier.
print(stack[c][1].final_estimator_.coef_) # The meta-classifiers coefficients.
print(stack[c][1].estimators_) # An overview of the base-classifiers
```

B.8 Fault severity assessment

The code for fault severity assessment reused the code from fault presence detection, but implemented another target adjustment to conform to Table 3.17 as shown below. The classifiers were subjected to minor adjustments and implemented a one-versus-all classifiers where necessary.

```
# Remove fault severity from target values.
# Target equals 1 indicates fault, and 0 indicates no fault.

def make_target_list_graded(y_set):
    # Change to severity indications.

    for i in range(len(y_set)):
        if y_set[i] > 10:
            y_set[i] = 3 # Extreme severity.
        elif y_set[i] >= 7:
            y_set[i] = 2 # Moderate severity.
        elif y_set[i] >= 1:
            y_set[i] = 1 # Low severity.

def make_dataset_targets_graded(dataset_tuple):
    # This function replaces any target value ...
    # other than 0 with 1.

    X_train, X_test, y_train, y_test, group_train, group_test = dataset_tuple

    make_target_list_graded(y_train)
    make_target_list_graded(y_test)
```

The functions above were applied to the output of the feature selection process in the following manner.

```
for dataset in datasets: # Go through every data set.
    make_dataset_targets_graded(dataset)
```

Appendix C

TSFRESH features

The following 23 pages describe the features calculated by TSFRESH. All of these features were included, with the exception of TSFRESH's Fourier transform features and their accompanying aggregation functions. This is included to have a reference for the features investigated in this thesis, since the feature extraction package may be subject to future change. This overview is also available in a searchable format at https://tsfresh.readthedocs.io/en/latest/api/tsfresh.feature_extraction.html under the heading "tsfresh.feature_extraction.feature_calculators module".

tsfresh.feature_extraction.feature_calculators.abs_energy(*x*) [source]

Returns the absolute energy of the time series which is the sum over the squared values

$$E = \sum_{i=1,\ldots,n} x_i^2$$

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.absolute_sum_of_changes(*x*) [source]

Returns the sum over the absolute value of consecutive changes in the series x

$$\sum_{i=1,\ldots,n-1} \mid x_{i+1} - x_i \mid$$

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.agg_autocorrelation(*x*, *param*) [source]

Calculates the value of an aggregation function f_{agg} (e.g. the variance or the mean) over the autocorrelation $R(l)$ for different lags. The autocorrelation $R(l)$ for lag l is defined as

$$R(l) = \frac{1}{(n-l)\sigma^2} \sum_{t=1}^{n-l} (X_t - \mu)(X_{t+l} - \mu)$$

where X_i are the values of the time series, n its length. Finally, σ^2 and μ are estimators for its variance and mean (See Estimation of the Autocorrelation function).

The $R(l)$ for different lags l form a vector. This feature calculator applies the aggregation function f_{agg} to this vector and returns

$$f_{agg}\left(R(1), \ldots, R(m)\right) \quad \text{for} \quad m = max(n, maxlag).$$

Here $maxlag$ is the second parameter passed to this function.

Parameters:

- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"f_agg": x, "maxlag", n} with x str, the name of a numpy function (e.g. "mean", "var", "std", "median"), its the name of the aggregator function that is applied to the autocorrelations. Further, n is an int and the maximal number of lags to consider.

Returns: the value of this feature

Return type: float

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.agg_linear_trend(*x*, *param*) [source]

Calculates a linear least-squares regression for values of the time series that were aggregated over chunks versus the sequence from 0 up to the number of chunks minus one.

This feature assumes the signal to be uniformly sampled. It will not use the time stamps to fit the model.

The parameters attr controls which of the characteristics are returned. Possible extracted attributes are “pvalue”, “rvalue”, “intercept”, “slope”, “stderr”, see the documentation of linregress for more information.

The chunksize is regulated by “chunk_len”. It specifies how many time series values are in each chunk.

Further, the aggregation function is controlled by “f_agg”, which can use “max”, “min” or , “mean”, “median”

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {“attr”: x, “chunk_len”: l, “f_agg”: f} with x, f an string and l an int

Returns: the different feature values

Return type: pandas.Series

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.approximate_entropy(*x*, *m*, *r*) [source]

Implements a vectorized Approximate entropy algorithm.

https://en.wikipedia.org/wiki/Approximate_entropy

For short time-series this method is highly dependent on the parameters, but should be stable for N > 2000, see:

Yentes et al. (2012) - *The Appropriate Use of Approximate Entropy and Sample Entropy with Short Data Sets*

Other shortcomings and alternatives discussed in:

Richman & Moorman (2000) - *Physiological time-series analysis using approximate entropy and sample entropy*

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **m** (*int*) – Length of compared run of data
- **r** (*float*) – Filtering level, must be positive

Returns: Approximate entropy

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.ar_coefficient(*x, param*) [source]

This feature calculator fits the unconditional maximum likelihood of an autoregressive AR(k) process. The k parameter is the maximum lag of the process

$$X_t = \varphi_0 + \sum_{i=1}^{k} \varphi_i X_{t-i} + \varepsilon_t$$

For the configurations from param which should contain the maxlag "k" and such an AR process is calculated. Then the coefficients φ_i whose index i contained from "coeff" are returned.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"coeff": x, "k": y} with x,y int

Return x: the different feature values

Return type: pandas.Series

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.augmented_dickey_fuller(*x, param*) [source]

The Augmented Dickey-Fuller test is a hypothesis test which checks whether a unit root is present in a time series sample. This feature calculator returns the value of the respective test statistic.

See the statsmodels implementation for references and more details.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"attr": x, "autolag": y} with x str, either "teststat", "pvalue" or "usedlag" and with y str, either of "AIC", "BIC", "t-stats" or None (See the documentation of adfuller() in statsmodels).

Returns: the value of this feature

Return type: float

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.autocorrelation(*x, lag*) [source]

Calculates the autocorrelation of the specified lag, according to the formula [1]

$$\frac{1}{(n-l)\sigma^2} \sum_{t=1}^{n-l} (X_t - \mu)(X_{t+l} - \mu)$$

where n is the length of the time series X_i, σ^2 its variance and μ its mean. *l* denotes the lag.

References

[1] https://en.wikipedia.org/wiki/Autocorrelation#Estimation

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **lag** (*int*) – the lag

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.binned_entropy(*x*, *max_bins*) [source]

First bins the values of x into max_bins equidistant bins. Then calculates the value of

$$-\sum_{k=0}^{min(max\_bins,len(x))} p_k log(p_k) \cdot \mathbf{1}_{(p_k>0)}$$

where p_k is the percentage of samples in bin k.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **max_bins** (*int*) – the maximal number of bins

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.c3(*x*, *lag*) [source]

This function calculates the value of

$$\frac{1}{n-2lag}\sum_{i=1}^{n-2lag} x_{i+2\cdot lag} \cdot x_{i+lag} \cdot x_i$$

which is

$$\mathbb{E}[L^2(X) \cdot L(X) \cdot X]$$

where $\mathbb{E}$ is the mean and L is the lag operator. It was proposed in [1] as a measure of non linearity in the time series.

References

[1] Schreiber, T. and Schmitz, A. (1997).
Discrimination power of measures for nonlinearity in a time series
PHYSICAL REVIEW E, VOLUME 55, NUMBER 5

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **lag** (*int*) – the lag that should be used in the calculation of the feature

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.change_quantiles(*x*, *ql*, *qh*, *isabs*, *f_agg*) [source]

First fixes a corridor given by the quantiles ql and qh of the distribution of x. Then calculates the average, absolute value of consecutive changes of the series x inside this corridor.

Think about selecting a corridor on the y-Axis and only calculating the mean of the absolute change of the time series inside this corridor.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **ql** (*float*) – the lower quantile of the corridor
- **qh** (*float*) – the higher quantile of the corridor
- **isabs** (*bool*) – should the absolute differences be taken?
- **f_agg** (*str, name of a numpy function (e.g. mean, var, std, median)*) – the aggregator function that is applied to the differences in the bin

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.cid_ce(*x*, *normalize*) [source]

This function calculator is an estimate for a time series complexity [1] (A more complex time series has more peaks, valleys etc.). It calculates the value of

$$\sqrt{\sum_{i=1}^{n-2lag} (x_i - x_{i+1})^2}$$

References

[1] Batista, Gustavo EAPA, et al (2014).
CID: an efficient complexity-invariant distance for time series.
Data Mining and Knowledge Discovery 28.3 (2014): 634-669.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **normalize** (*bool*) – should the time series be z-transformed?

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.count_above(*x*, *t*) [source]

Returns the percentage of values in x that are higher than t

Parameters:
- **x** (*pandas.Series*) – the time series to calculate the feature of
- **t** (*float*) – value used as threshold

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.count_above_mean(*x*) [source]

Returns the number of values in x that are higher than the mean of x

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.count_below(*x*, *t*) [source]

Returns the percentage of values in x that are lower than t

Parameters:
- **x** (*pandas.Series*) – the time series to calculate the feature of
- **t** (*float*) – value used as threshold

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.count_below_mean(*x*) [source]

Returns the number of values in x that are lower than the mean of x

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.cwt_coefficients(*x*, *param*) [source]

Calculates a Continuous wavelet transform for the Ricker wavelet, also known as the “Mexican hat wavelet” which is defined by

$$\frac{2}{\sqrt{3a}\pi^{\frac{1}{4}}}(1-\frac{x^2}{a^2})exp(-\frac{x^2}{2a^2})$$

where a is the width parameter of the wavelet function.

This feature calculator takes three different parameter: widths, coeff and w. The feature calculater takes all the different widths arrays and then calculates the cwt one time for each different width array. Then the values for the different coefficient for coeff and width w are returned. (For each dic in param one feature is returned)

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"widths":x, "coeff": y, "w": z} with x array of int and y,z int

Returns: the different feature values

Return type: pandas.Series

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.energy_ratio_by_chunks(*x*, *param*) [source]

Calculates the sum of squares of chunk i out of N chunks expressed as a ratio with the sum of squares over the whole series.

Takes as input parameters the number num_segments of segments to divide the series into and segment_focus which is the segment number (starting at zero) to return a feature on.

If the length of the time series is not a multiple of the number of segments, the remaining data points are distributed on the bins starting from the first. For example, if your time series consists of 8 entries, the first two bins will contain 3 and the last two values, e.g. *[0., 1., 2.]*, *[3., 4., 5.]* and *[6., 7.]*.

Note that the answer for *num_segments = 1* is a trivial "1" but we handle this scenario in case somebody calls it. Sum of the ratios should be 1.0.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** – contains dictionaries {"num_segments": N, "segment_focus": i} with N, i both ints

Returns: the feature values

Return type: list of tuples (index, data)

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.fft_aggregated(*x*, *param*) [source]

Returns the spectral centroid (mean), variance, skew, and kurtosis of the absolute fourier transform spectrum.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"aggtype": s} where s str and in ["centroid", "variance", "skew", "kurtosis"]

Returns: the different feature values

Return type: pandas.Series

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.fft_coefficient(*x*, *param*) [source]

Calculates the fourier coefficients of the one-dimensional discrete Fourier Transform for real input by fast fourier transformation algorithm

$$A_k = \sum_{m=0}^{n-1} a_m \exp\left\{-2\pi i \frac{mk}{n}\right\}, \qquad k = 0, \ldots, n-1.$$

The resulting coefficients will be complex, this feature calculator can return the real part (attr=="real"), the imaginary part (attr=="imag), the absolute value (attr=""abs) and the angle in degrees (attr=="angle).

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"coeff": x, "attr": s} with x int and x >= 0, s str and in ["real", "imag", "abs", "angle"]

Returns: the different feature values

Return type: pandas.Series

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.first_location_of_maximum(*x*) [source]

Returns the first location of the maximum value of x. The position is calculated relatively to the length of x.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.first_location_of_minimum(*x*) [source]

Returns the first location of the minimal value of x. The position is calculated relatively to the length of x.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.fourier_entropy(*x*, *bins*) [source]

Calculate the binned entropy of the power spectral density of the time series (using the welch method).

Ref: https://hackaday.io/project/707-complexity-of-a-time-series/details Ref: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.signal.welch.html

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.friedrich_coefficients(*x*, *param*) [source]

Coefficients of polynomial $h(x)$, which has been fitted to the deterministic dynamics of Langevin model

$$\dot{x}(t) = h(x(t)) + \mathcal{N}(0, R)$$

as described by [1].

For short time-series this method is highly dependent on the parameters.

References

[1] Friedrich et al. (2000): Physics Letters A 271, p. 217-222
Extracting model equations from experimental data

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"m": x, "r": y, "coeff": z} with x being positive integer, the order of polynom to fit for estimating fixed points of dynamics, y positive float, the number of quantils to use for averaging and finally z, a positive integer corresponding to the returned coefficient

Returns: the different feature values

Return type: pandas.Series

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.has_duplicate(*x*) [source]

Checks if any value in x occurs more than once

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: bool

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.has_duplicate_max(*x*) [source]

Checks if the maximum value of x is observed more than once

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: bool

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.has_duplicate_min(*x*) [source]

Checks if the minimal value of x is observed more than once

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: bool

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.index_mass_quantile(*x*, *param*) [source]

Those apply features calculate the relative index i where q% of the mass of the time series x lie left of i. For example for q = 50% this feature calculator will return the mass center of the time series

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"q": x} with x float

Returns: the different feature values

Return type: pandas.Series

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.kurtosis(*x*) [source]

Returns the kurtosis of x (calculated with the adjusted Fisher-Pearson standardized moment coefficient G2).

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.large_standard_deviation(*x*, *r*) [source]

Boolean variable denoting if the standard dev of x is higher than 'r' times the range = difference between max and min of x. Hence it checks if

$$std(x) > r * (max(X) - min(X))$$

According to a rule of the thumb, the standard deviation should be a forth of the range of the values.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **r** (*float*) – the percentage of the range to compare with

Returns: the value of this feature

Return type: bool

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.last_location_of_maximum(*x*) [source]

Returns the relative last location of the maximum value of x. The position is calculated relatively to the length of x.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.last_location_of_minimum(*x*) [source]

Returns the last location of the minimal value of x. The position is calculated relatively to the length of x.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.lempel_ziv_complexity(*x*, *bins*) [source]

Calculate a complexity estimate based on the Lempel-Ziv compression algorithm.

The complexity is defined as the number of dictionary entries (or sub-words) needed to encode the time series when viewed from left to right. FOr this, the time series is first binned into the given number of bins. Then it is converted into sub-words with different prefixes. The number of sub-words needed for this divided by the length of the time series is the complexity estimate.

For example, if the time series (after binning in only 2 bins) would look like “100111”, the different sub-words would be 1, 0, 01 and 11 and therefore the result is 4/6 = 0.66.

Ref: https://github.com/Naereen/Lempel-Ziv_Complexity/blob/master/src/lempel_ziv_complexity.py

This function is of type: simple

tsfresh.feature_extraction.feature_calculators.length(x) [source]

Returns the length of x

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: int

This function is of type: simple

tsfresh.feature_extraction.feature_calculators.linear_trend(x, param) [source]

Calculate a linear least-squares regression for the values of the time series versus the sequence from 0 to length of the time series minus one. This feature assumes the signal to be uniformly sampled. It will not use the time stamps to fit the model. The parameters control which of the characteristics are returned.

Possible extracted attributes are “pvalue”, “rvalue”, “intercept”, “slope”, “stderr”, see the documentation of linregress for more information.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {“attr”: x} with x an string, the attribute name of the regression model

Returns: the different feature values

Return type: pandas.Series

This function is of type: combiner

tsfresh.feature_extraction.feature_calculators.linear_trend_timewise(x, param) [source]

Calculate a linear least-squares regression for the values of the time series versus the sequence from 0 to length of the time series minus one. This feature uses the index of the time series to fit the model, which must be of a datetime dtype. The parameters control which of the characteristics are returned.

Possible extracted attributes are “pvalue”, “rvalue”, “intercept”, “slope”, “stderr”, see the documentation of linregress for more information.

Parameters:
- **x** (*pandas.Series*) – the time series to calculate the feature of. The index must be datetime.
- **param** (*list*) – contains dictionaries {“attr”: x} with x an string, the attribute name of the regression model

Returns: the different feature values

Return type: list

This function is of type: combiner

tsfresh.feature_extraction.feature_calculators.longest_strike_above_mean(x) [source]

Returns the length of the longest consecutive subsequence in x that is bigger than the mean of x

Parameters: x (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

tsfresh.feature_extraction.feature_calculators.longest_strike_below_mean(x) [source]

Returns the length of the longest consecutive subsequence in x that is smaller than the mean of x

Parameters: x (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

tsfresh.feature_extraction.feature_calculators.max_langevin_fixed_point(x, r, m) [source]

Largest fixed point of dynamics :math:argmax_x {h(x)=0}` estimated from polynomial $h(x)$, which has been fitted to the deterministic dynamics of Langevin model

$$\dot(x)(t) = h(x(t)) + R(N)(0,1)$$

as described by

> Friedrich et al. (2000): Physics Letters A 271, p. 217-222 *Extracting model equations from experimental data*

For short time-series this method is highly dependent on the parameters.

Parameters:
- x (*numpy.ndarray*) – the time series to calculate the feature of
- m (*int*) – order of polynom to fit for estimating fixed points of dynamics
- r (*float*) – number of quantils to use for averaging

Returns: Largest fixed point of deterministic dynamics

Return type: float

This function is of type: simple

tsfresh.feature_extraction.feature_calculators.maximum(x) [source]

Calculates the highest value of the time series x.

Parameters: x (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.mean(*x*) [source]

Returns the mean of x

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.mean_abs_change(*x*) [source]

Returns the mean over the absolute differences between subsequent time series values which is

$$\frac{1}{n} \sum_{i=1,\ldots,n-1} |x_{i+1} - x_i|$$

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.mean_change(*x*) [source]

Returns the mean over the differences between subsequent time series values which is

$$\frac{1}{n-1} \sum_{i=1,\ldots,n-1} x_{i+1} - x_i = \frac{1}{n-1}(x_n - x_1)$$

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.mean_second_derivative_central(*x*) [source]

Returns the mean value of a central approximation of the second derivative

$$\frac{1}{2(n-2)} \sum_{i=1,\ldots,n-1} \frac{1}{2}(x_{i+2} - 2 \cdot x_{i+1} + x_i)$$

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.median(x) [source]

Returns the median of x

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.minimum(x) [source]

Calculates the lowest value of the time series x.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.number_crossing_m(x, m) [source]

Calculates the number of crossings of x on m. A crossing is defined as two sequential values where the first value is lower than m and the next is greater, or vice-versa. If you set m to zero, you will get the number of zero crossings.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **m** (*float*) – the threshold for the crossing

Returns: the value of this feature

Return type: int

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.number_cwt_peaks(x, n) [source]

This feature calculator searches for different peaks in x. To do so, x is smoothed by a ricker wavelet and for widths ranging from 1 to n. This feature calculator returns the number of peaks that occur at enough width scales and with sufficiently high Signal-to-Noise-Ratio (SNR)

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **n** (*int*) – maximum width to consider

Returns: the value of this feature

Return type: int

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.number_peaks(*x*, *n*) [source]

Calculates the number of peaks of at least support n in the time series x. A peak of support n is defined as a subsequence of x where a value occurs, which is bigger than its n neighbours to the left and to the right.

Hence in the sequence

```
>>> x = [3, 0, 0, 4, 0, 0, 13]
```

4 is a peak of support 1 and 2 because in the subsequences

```
>>> [0, 4, 0]
>>> [0, 0, 4, 0, 0]
```

4 is still the highest value. Here, 4 is not a peak of support 3 because 13 is the 3th neighbour to the right of 4 and its bigger than 4.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **n** (*int*) – the support of the peak

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.partial_autocorrelation(*x*, *param*) [source]

Calculates the value of the partial autocorrelation function at the given lag. The lag *k* partial autocorrelation of a time series $\{x_t, t = 1 \ldots T\}$ equals the partial correlation of x_t and x_{t-k}, adjusted for the intermediate variables $\{x_{t-1}, \ldots, x_{t-k+1}\}$ ([1]). Following [2], it can be defined as

$$\alpha_k = \frac{Cov(x_t, x_{t-k} | x_{t-1}, \ldots, x_{t-k+1})}{\sqrt{Var(x_t | x_{t-1}, \ldots, x_{t-k+1}) Var(x_{t-k} | x_{t-1}, \ldots, x_{t-k+1})}}$$

with (a) $x_t = f(x_{t-1}, \ldots, x_{t-k+1})$ and (b) $x_{t-k} = f(x_{t-1}, \ldots, x_{t-k+1})$ being AR(k-1) models that can be fitted by OLS. Be aware that in (a), the regression is done on past values to predict x_t whereas in (b), future values are used to calculate the past value x_{t-k}. It is said in [1] that "for an AR(p), the partial autocorrelations [α_k] will be nonzero for *k*<=*p* and zero for *k*>*p*." With this property, it is used to determine the lag of an AR-Process.

References

[1] Box, G. E., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015).
Time series analysis: forecasting and control. John Wiley & Sons.
[2] https://onlinecourses.science.psu.edu/stat510/node/62

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"lag": val} with int val indicating the lag to be returned

Returns: the value of this feature

Return type: float

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.percentage_of_reoccurring_datapoints_to_all_datapoints(*x*)

Returns the percentage of unique values, that are present in the time series more than once.

len(different values occurring more than once) / len(different values)

This means the percentage is normalized to the number of unique values, in contrast to the percentage_of_reoccurring_values_to_all_values.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.percentage_of_reoccurring_values_to_all_values(*x*) [source]

Returns the ratio of unique values, that are present in the time series more than once.

# of data points occurring more than once / # of all data points

This means the ratio is normalized to the number of data points in the time series, in contrast to the percentage_of_reoccurring_datapoints_to_all_datapoints.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.permutation_entropy(*x*, *tau*, *dimension*) [source]

Calculate the permutation entropy.

Three steps are needed for this:

1. chunk the data into sub-windows of length D starting every tau. Following the example from the reference, a vector

   x = [4, 7, 9, 10, 6, 11, 3

   with D = 3 and tau = 1 is turned into

   > [[4, 7, 9],
   >
   > [7, 9, 10], [9, 10, 6], [10, 6, 11], [6, 11, 3]]

2. replace each D-window by the permutation, that captures the ordinal ranking of the data. That gives

   > [[0, 1, 2],
   >
   > [0, 1, 2], [1, 2, 0], [1, 0, 2], [1, 2, 0]]

3. Now we just need to count the frequencies of every permutation and return their entropy (we use log_e and not log_2).

> **Ref:** https://www.aptech.com/blog/permutation-entropy/
>
> Bandt, Christoph and Bernd Pompe. "Permutation entropy: a natural complexity measure for time series." Physical review letters 88 17 (2002): 174102 .

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.quantile(*x*, *q*) [source]

Calculates the q quantile of x. This is the value of x greater than q% of the ordered values from x.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **q** (*float*) – the quantile to calculate

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.range_count(*x*, *min*, *max*) [source]

Count observed values within the interval [min, max).

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **min** (*int or float*) – the inclusive lower bound of the range
- **max** (*int or float*) – the exclusive upper bound of the range

Returns: the count of values within the range

Return type: int

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.ratio_beyond_r_sigma(*x*, *r*) [source]

Ratio of values that are more than r*std(x) (so r sigma) away from the mean of x.

Parameters: **x** (*iterable*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.ratio_value_number_to_time_series_length(*x*) [source]

Returns a factor which is 1 if all values in the time series occur only once, and below one if this is not the case. In principle, it just returns

unique values / # values

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.sample_entropy(*x*) [source]

Calculate and return sample entropy of x.

References

[1] http://en.wikipedia.org/wiki/Sample_Entropy
[2] https://www.ncbi.nlm.nih.gov/pubmed/10843903?dopt=Abstract

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.set_property(*key*, *value*) [source]

This method returns a decorator that sets the property key of the function to value

---

tsfresh.feature_extraction.feature_calculators.skewness(*x*) [source]

Returns the sample skewness of x (calculated with the adjusted Fisher-Pearson standardized moment coefficient G1).

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.spkt_welch_density(*x*, *param*) [source]

This feature calculator estimates the cross power spectral density of the time series x at different frequencies. To do so, the time series is first shifted from the time domain to the frequency domain.

The feature calculators returns the power spectrum of the different frequencies.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **param** (*list*) – contains dictionaries {"coeff": x} with x int

Returns: the different feature values

Return type: pandas.Series

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.standard_deviation(*x*) [source]

Returns the standard deviation of x

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.sum_of_reoccurring_data_points(*x*) [source]

Returns the sum of all data points, that are present in the time series more than once.

For example

sum_of_reoccurring_data_points([2, 2, 2, 2, 1]) = 8

as 2 is a reoccurring value, so all 2's are summed up.

This is in contrast to `sum_of_reoccurring_values`, where each reoccuring value is only counted once.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.sum_of_reoccurring_values(*x*) [source]

Returns the sum of all values, that are present in the time series more than once.

For example

sum_of_reoccurring_values([2, 2, 2, 2, 1]) = 2

as 2 is a reoccurring value, so it is summed up with all other reoccuring values (there is none), so the result is 2.

This is in contrast to `sum_of_reoccurring_data_points`, where each reoccuring value is only counted as often as it is present in the data.

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.sum_values(*x*) [source]

Calculates the sum over the time series values

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.symmetry_looking(*x*, *param*) [source]

Boolean variable denoting if the distribution of x *looks symmetric*. This is the case if

$$|mean(X) - median(X)| < r * (max(X) - min(X))$$

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **r** (*float*) – the percentage of the range to compare with

Returns: the value of this feature

Return type: bool

This function is of type: combiner

---

tsfresh.feature_extraction.feature_calculators.time_reversal_asymmetry_statistic(*x*, *lag*)

[source]

This function calculates the value of

$$\frac{1}{n-2lag}\sum_{i=1}^{n-2lag} x_{i+2\cdot lag}^2 \cdot x_{i+lag} - x_{i+lag} \cdot x_i^2$$

which is

$$\mathbb{E}[L^2(X)^2 \cdot L(X) - L(X) \cdot X^2]$$

where $\mathbb{E}$ is the mean and L is the lag operator. It was proposed in [1] as a promising feature to extract from time series.

References

[1] Fulcher, B.D., Jones, N.S. (2014).
Highly comparative feature-based time-series classification.
Knowledge and Data Engineering, IEEE Transactions on 26, 3026–3037.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **lag** (*int*) – the lag that should be used in the calculation of the feature

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.value_count(*x*, *value*) [source]

Count occurrences of *value* in time series x.

Parameters:
- **x** (*numpy.ndarray*) – the time series to calculate the feature of
- **value** (*int or float*) – the value to be counted

Returns: the count

Return type: int

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.variance(*x*) [source]

Returns the variance of x

Parameters: **x** (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.variance_larger_than_standard_deviation(*x*) [source]

Boolean variable denoting if the variance of x is greater than its standard deviation. Is equal to variance of x being larger than 1

Parameters: x (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: bool

This function is of type: simple

---

tsfresh.feature_extraction.feature_calculators.variation_coefficient(x) [source]

Returns the variation coefficient (standard error / mean, give relative value of variation around mean) of x.

Parameters: x (*numpy.ndarray*) – the time series to calculate the feature of

Returns: the value of this feature

Return type: float

This function is of type: simple

tsfresh.feature_extraction.settings module

This file contains methods/objects for controlling which features will be extracted when calling extract_features. For the naming of the features, see Feature Calculation.

---

class **tsfresh.feature_extraction.settings.ComprehensiveFCParameters** [source]

Bases: `dict`

Create a new ComprehensiveFCParameters instance. You have to pass this instance to the extract_feature instance.

It is basically a dictionary (and also based on one), which is a mapping from string (the same names that are in the feature_calculators.py file) to a list of dictionary of parameters, which should be used when the function with this name is called.

Only those strings (function names), that are keys in this dictionary, will be later used to extract features - so whenever you delete a key from this dict, you disable the calculation of this feature.

You can use the settings object with

```
>>> from tsfresh.feature_extraction import extract_features, ComprehensiveFCParameters
>>> extract_features(df, default_fc_parameters=ComprehensiveFCParameters())
```

to extract all features (which is the default nevertheless) or you change the ComprehensiveFCParameters object to other types (see below).

---

class **tsfresh.feature_extraction.settings.EfficientFCParameters** [source]

Skreien, Tarjei Nesbø

Application of signal processing and machine learning tools in fault detection of synchronous generators

NTNU
Kunnskap for en bedre verden