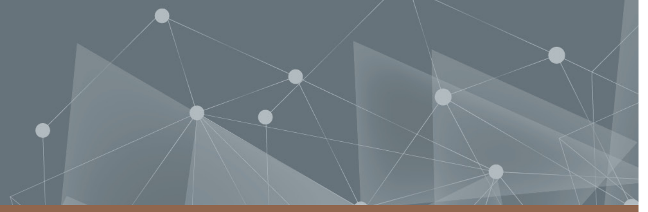
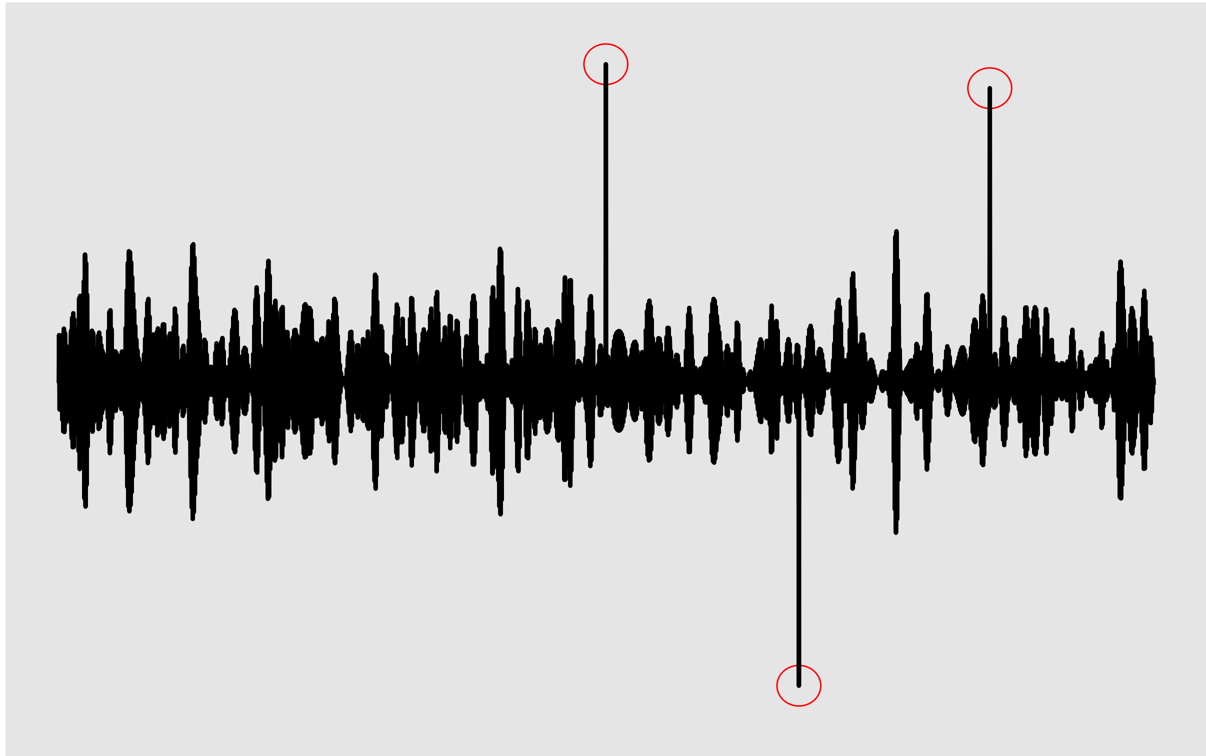




Characterization & Anomaly Detection in Radio Sensors

Algorithms for real-time fault detection and classification

Master's thesis in Engineering Mathematics and Computational Science

OSCAR LILJENZIN

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Characterization & Anomaly Detection in Radio Sensors

Algorithms for real-time fault detection and classification

OSCAR LILJENZIN

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2024

Characterization & Anomaly Detection in Radio Sensors
Algorithms for real-time fault detection and classification
Oscar Liljenzin

© OSCAR LILJENZIN, 2024.

Supervisors:	Jing Liu B & Winston Garcia-gabin, Ericsson AB
	Johan Jonasson, Department of Mathematical Sciences
Examiner:	Johan Jonasson, Department of Mathematical Sciences

Master's Thesis 2024
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A random signal in black with some anomalies circled in red.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

Characterization & Anomaly Detection in Radio Sensors
Algorithms for real-time fault detection and classification
Oscar Liljenzin
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

In the ever-evolving field of radio technology, accurate sensor readings are a necessity. Faults in sensors can be a costly endeavour. Thus, it is interesting to explore the possibility of automatic fault detection using sensor readings. First different types of faults that occur in radion sensors are modeled. Then different algorithms for fault detection were implemented and evaluated. From this evaluation, it was found that algorithms based on simple heuristics and statistics can in most cases, perform similarly or even better than more advanced machine learning methods.

Acknowledgements

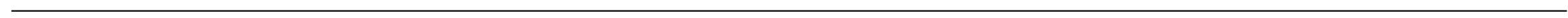
First and foremost, I would like to express my gratitude to my two supervisors at Ericsson, Jing Liu B and Winston Garcia-gabin for their guidance and advice on this project. I would also like to extend my appreciation to my manager Roy Naddaf and everyone else at Ericsson in Kista for a smooth onboarding experience and for supporting me in my work. Moreover, I would also like to thank my academic supervisor Johan for helping me with some of the theory and rigor in this thesis. Finally, I would also like to thank my family members for supporting me through this project. Given that this project mainly took place in Stockholm it would not have been possible without their help and support.

Oscar Liljenzin, Gothenburg, June 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AIC	Akaike's Information Criterion
AICc	Corrected Akaike's Information Criterion
ANN	Artifical Neural Network
AR	Autoregressive
ARIMA	Autoregressive Integrated Moving Average
ARMA	Autoregressive Moving Average
AUC	Area Under Curve
BERNN	Bidirectional Elman Recurrent Neural Network
BGRU	Bidirectional Gated Recurrent Unit
BIC	Bayesian Information Criterion
BLSTM	Bidirection Long Short-Term Memory
BPTT	Backpropagation Through Time
CLT	Central Limit Theorem
CNN	Convolutional Neural Network
DTW	Dynamic Time Warping
FPR	False Positive Rate
GRU	Gated Recurrent Unit
IID	Independent and Identically Distributed
kNN	k-Nearest Neighbors
KAN	Kolmogorov-Arnold Network
LBGFS	Limited-Memory Broyden-Fletcher-Goldfarb-Shanno
LSTM	Long Short-Term Memory
MA	Moving Average
MCC	Matthew's Correlation Coefficient
ML	Machine Learning
MLE	Maximum Likelihood Estimation
MLP	Multilayer Perceptron
MSE	Mean Squared Error
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
TPR	True Positive Rate
WN	White Noise



Contents

List of Acronyms **ix**

List of Figures **xvii**

List of Tables **xix**

List of Algorithms **xxi**

1 Introduction **1**

1.1 Anomalies 1

2 Theory **3**

2.1 Sensor Model 3

2.1.1 Multivariate sensor model 3

2.2 Faulty sensor model 3

2.2.1 Hardover fault 4

2.2.2 Drift fault 5

2.2.3 Spike fault 5

2.2.4 Erratic fault 5

2.2.5 Stuck fault 6

2.2.6 Frozen fault 6

2.2.7 Other faults 6

2.3 Time series modeling 7

2.3.1 ARIMA model 7

2.3.2 Information criteria 8

2.3.3 Dynamic time warping 9

2.4 Machine learning methods for anomaly detection 10

2.4.1 Extremely randomized trees 10

2.4.2 K-nearest neighbors 11

2.4.3 Recurrent neural networks 11

2.4.4 Cross-entropy 13

2.4.5 Multiclass classification 13

2.4.6 Long short-term memory 14

2.4.7 Gated recurrent unit 15

2.5 Evaluation metrics 16

2.5.1 Confusion matrix 16

2.5.2 Precision, Recall and F_1 score 17

2.5.3 Receiver operating characteristic 17
2.5.4 Matthews correlation coefficient 17
2.6 Bootstrap 18

3 Method **19**
3.1 Dataset 19
3.1.1 Fault generation 20
3.1.2 Data generation 21
3.2 Algorithms 22
3.2.1 Sliding window 22
3.2.2 Baseline method 23
3.2.3 Bootstrap method 25
3.2.4 ARIMA method 26
3.2.5 Extra trees method 27
3.2.6 KNN-DTW method 28
3.2.7 LSTM method 28
3.2.8 Elman RNN & GRU method 30
3.2.8.1 Number of parameters 30
3.3 Number of samples analysis 30

4 Result **33**
4.1 Result for binary classification 33
4.1.1 Number of samples analysis 34
4.2 Result for multiclass classification 37
4.2.1 Results for sensor class classification 38
4.2.2 Results for fault class classification 42

5 Discussion **45**
5.1 The dataset 45
5.2 The prevalence of faults 45
5.3 The independence test 45
5.4 Note on the ANN-based methods 46
5.5 Uncertainty in the results 46
5.6 Note regarding the kNN-DTW method 46
5.7 Note regarding the ARIMA method 46
5.8 Future work 47

6 Conclusion **49**

References **51**

A Appendix 1 **I**
A.1 White noise I

B Appendix 2 **III**
B.1 Note on the difference operator applied to the faults III
B.1.1 Hardover fault III
B.1.2 Drift fault III

B.1.3 Conclusions IV

C Appendix 3 **V**

C.1 Pearson's Chi-square test for independence V

D Appendix 4 **VII**

D.1 Results VII

D.1.1 binary classification VIII

D.1.2 multiclass classification XXIV

List of Figures

2.1 The six different types of faults which can occur in a sensor that this thesis will focus on. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Note that the black dotted line in subfigure **e)** shows a predefined value, this is to clarify the distinction between the stuck and frozen faults which is later discussed in Section 2.2.6. Figure created with [34]. 4
2.2 A graphical overview of the difference between computing the **a)** Euclidean distance and **b)** DTW distance between two sequences. Note that in **a)**, the distance is computed between the sequences for each time point, while in **b)**, the distance is computed from an optimal alignment according to Equation (2.20). Figure created with [34]. . . 10
2.3 A graphical overview of an RNN, **a)** shows a single neuron while **b)** shows the network unfolded through time. Note that affine transformations are not explicitly shown. The box labeled A is meant to represent some architecture, a common one is the so-called Elman RNN [21] which can be seen in Figure 2.4. 12
2.4 A graphical overview of the architecture for an Elman RNN with a tanh activation function. Note that there are examples where a *rectified linear unit* (ReLU) activation function is used instead. 12
2.5 A graphical overview of an LSTM memory cell. Note that like in Figure 2.4, affine transformations are not explicitly shown. 14
2.6 A graphical overview of a GRU memory cell. Note that like in Figure 2.4 and Figure 2.5, affine transformations are not explicitly shown. . . 15

3.1 A schematic overview of the method. Note that the split occurs before any data is generated according to Algorithm 3.2 to ensure that no single part of the time series is present in both the training and the test sets. 19
3.2 Illustration of a sliding window of size n_w operating on multivariate time series with n_s sensors. This is a good method for online detection of anomalies. 22
3.3 A graphical overview of the architecture used in this method. By normalization, an instance normalization layer is intended. Note that for the binary classification task, softmax is meant to mean sigmoid, as it can be seen as a special case of softmax (see Section 2.4.5). . . . 29

4.1 Accuracy for the baseline, bootstrap, LSTM and GRU classifiers. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34]. 34
4.2 Accuracy for the ARIMA, Elman RNN, extra trees and kNN-DTW classifiers. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34]. 35
4.3 Probability of making a detection based on the number of samples with faulty data. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The three types of faults are **a)** hardover, **b)** drift and **c)** spike. Figure created with [34]. 36
4.4 Probability of making a detection based on the number of samples with faulty data. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The three types of faults are **a)** erratic, **b)** stuck and **c)** frozen. Figure created with [34]. 37
4.5 Accuracy for the baseline, bootstrap, LSTM and GRU classifiers. Note that the test set is balanced and only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34]. 38
4.6 Confusion matrices for the baseline classifier from Section 3.2.2 with a window size of 12 on a balanced test set. In each matrix, the percentage of 26,200 classifications is shown. Figure created with [34]. 39
4.7 Confusion matrices for the bootstrap classifier from Section 3.2.3 with a window size of 12 on a balanced test set. In each matrix, the percentage of 26,200 classifications is shown. Figure created with [34]. 40
4.8 Confusion matrices for the LSTM classifier from Section 3.2.7 with a window size of 12 on a balanced test set. In each matrix, the percentage of 26,200 classifications is shown. Figure created with [34]. 41
4.9 Confusion matrices for the GRU classifier from Section 3.2.8 with a window size of 12 on a balanced test set. In each matrix, the percentage of 26,200 classifications is shown. Figure created with [34]. 42
4.10 Accuracy for the LSTM and GRU classifiers. Note that the test set is balanced, so it consists of an equal amount of each type of fault and no fault. Figure created with [34]. 43
4.11 Confusion matrix for the LSTM classifier from Section 3.2.7 with a window size of 12 on a balanced test set. In each matrix, the percentage of 19,650 classifications is shown. Figure created with [34]. 44
4.12 Confusion matrix for the GRU classifier from Section 3.2.8 with a window size of 12 on a balanced test set. In each matrix, the percentage of 19,650 classifications is shown. Figure created with [34]. 44

List of Tables

2.1 A common way to organize binary classifications is in a so-called confusion matrix which is a 2×2 contingency table over the four possible combinations of predicted and true labels. 16

3.1 Parameters used when generating faults according to Algorithm 3.1. Note that frozen fault (see Section 2.2.6) is not present in this table since it is non-parametric. The values and intervals of these parameters were verified with field experts at Ericsson. 20
3.2 Parameters used for the baseline method (see Algorithm 3.3). Note that the same parameter is used across all sensors, so $\alpha_{\mathcal{M}}^{(i)} = \alpha_{\mathcal{M}}$ for $i = 1, \ldots, n_s$. 23
3.3 Parameters used for the extra trees method in [53]. 27
3.4 Parameters used for the kNN-DTW method in [65]. 28
3.5 Number of trainable parameters for the BERNN, BLSTM, BGRU. Note that these expressions can be simplified further by using that $n_h = n_s$ as mentioned in Section 3.2.7. However, this simplification has not been made should the reader wish to change the value for n_h. Note that to get the number of parameters for the unidirectional variants, we simply halve the number of parameters for the bidirectional variant. 30

List of Algorithms

2.1 Bootstrap 18
3.1 Fault generation 20
3.2 Data generation 21
3.3 Baseline method 24
3.4 Bootstrap method 25
3.5 ARIMA training 26
3.6 ARIMA method 27
3.7 Determine the number of samples 31

1

Introduction

The ever-evolving field of radio technology is vital in meeting the increasing demands of modern communication. The performance of radio systems is influenced by various factors, both external (e.g., environment, network traffic) and internal (e.g., radio configuration, component behavior). To enhance radio service stability and efficiency, the development of robust radio control systems is imperative. This thesis aims to characterize and detect anomalies (see Section 1.1) in radio sensors using measured data, ultimately contributing to improved radio design and performance.

This thesis is written in collaboration with Ericsson, a world leader in radios currently looking into automating fault detection in radios.

1.1 Anomalies

There are multiple definitions of what is considered an anomaly [22, 58]. This thesis, however, will limit itself to a set of anomalies presented in Section 2.2. These anomalies represent different types of faults that can occur in sensors and thus real-time detection of these becomes crucial for enhancing radio performance. The relevance of the faults presented in Section 2.2 have been verified by field experts.

2

Theory

In this chapter, theoretical concepts are introduced to give the reader a basic understanding of the concepts used in this thesis.

2.1 Sensor Model

We will view the output $x(t) \in \mathbb{R}$ of a sensor as a discrete time series $\{x(t)\}_{t=0}^{T}$, over some partition $0 = t_0 < t_1 < \ldots < t_N = T$ of the interval $[0, T]$. We restrict ourselves to only consider equidistant time points, i.e., $\Delta t = t_i - t_{i-1}$ is constant for all $i = 1, \ldots, N$. Moreover, we will also assume that $\Delta t = 1$.

A more thorough model based on time series analysis will be presented in Section 2.3.

2.1.1 Multivariate sensor model

It is possible to model multiple sensor outputs simultaneously as a multivariate time series, $x(t) = \left(x^{(1)}(t), \ldots, x^{(n_s)}(t)\right) \in \mathbb{R}^{n_s}$ where $n_s \in \mathbb{N}$ denotes the number of sensors. However, the models for the different faults in this thesis, which will be presented in Section 2.2 will only be for a scalar time series. Some multivariate models of these faults are presented in [41].

In some literature [46], the topology of the sensor setup is incorporated into the model. However, no such assumptions regarding the topology of the sensor setup will be made in this thesis.

2.2 Faulty sensor model

In this section, several faults will be presented and modeled which can be seen in Figure 2.1. The models in this section will be based on those presented in [19], except for the spike fault presented in Section 2.2.3 which will be based on the model presented in [57]. We let $\tilde{x}(t)$ denote a faulty sensor output.

Note that the faults presented in this thesis will be a process which starts from some critical time point τ and then continues until T. However, in real-world scenarios, this is not always the case because a radio might recover from a fault. This might

happen due to the radio restarting for various reasons which may or may not be related to the current fault in the sensor output.

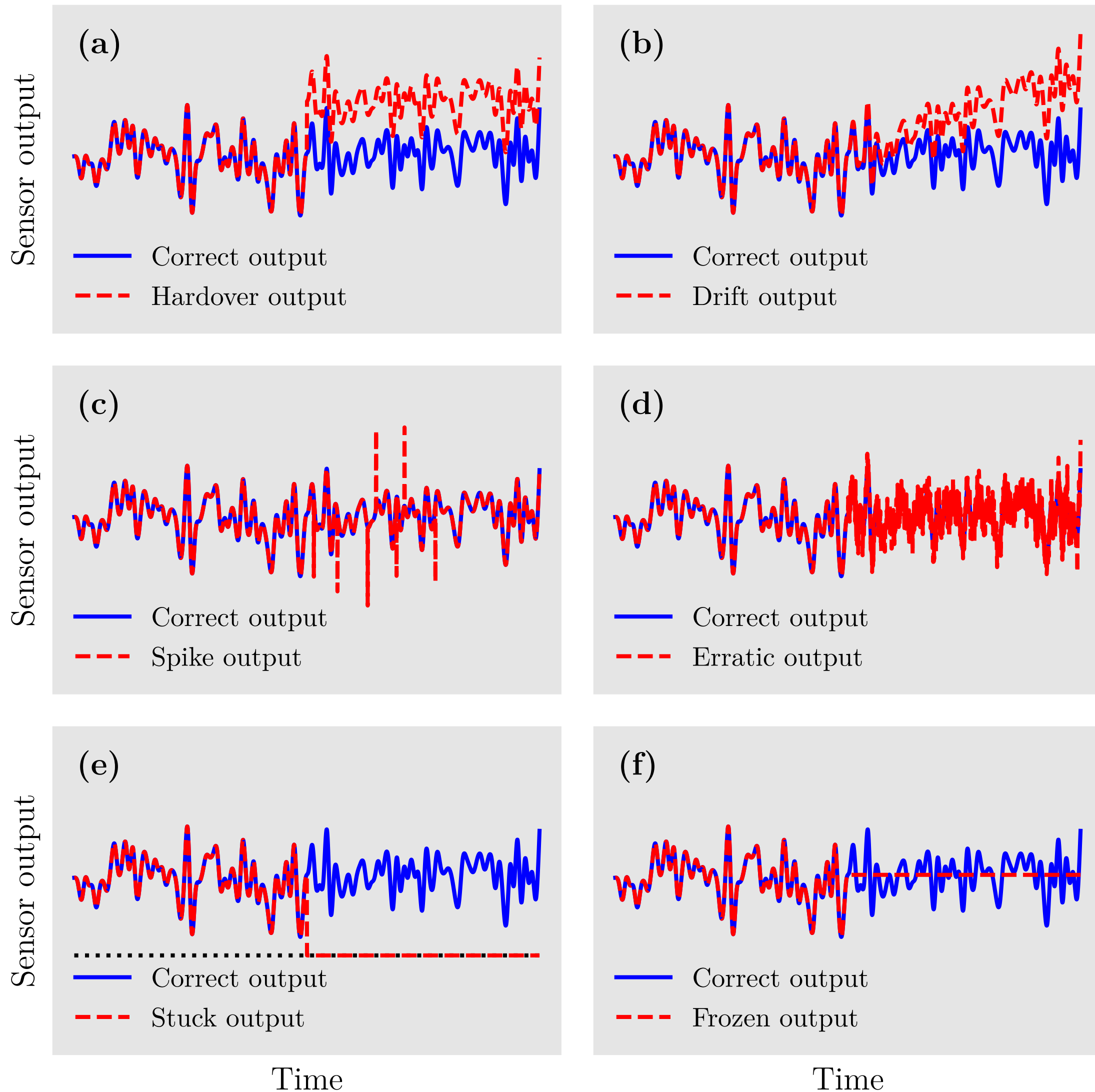


Figure 2.1: The six different types of faults which can occur in a sensor that this thesis will focus on. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Note that the black dotted line in subfigure **e)** shows a predefined value, this is to clarify the distinction between the stuck and frozen faults which is later discussed in Section 2.2.6. Figure created with [34].

2.2.1 Hardover fault

This type of fault is also known as [72, 19] Bias fault, or [49] Offset fault in some literature. This fault [19] is defined by the sensor output being shifted by a positive offset. To model this a constant $\delta > 0$ is added to the original sensor output after

some time point τ. Thus $\tilde{x}(t)$ can be modeled as

$$\tilde{x}(t) = \begin{cases} x(t) & \text{if } t < \tau, \\ x(t) + \delta & \text{if } t \geqslant \tau. \end{cases} \tag{2.1}$$

This fault can be seen in Figure 2.1a. A physical cause for this failure is [48] miscalibration of the sensor.

2.2.2 Drift fault

A drift fault [19] is defined by the sensor output linearly drifting from the correct output. To model this a linear function with constant slope $\lambda \neq 0$ is added to the original sensor output after some time point τ. Thus $\tilde{x}(t)$ is modeled as

$$\tilde{x}(t) = \begin{cases} x(t) & \text{if } t < \tau, \\ x(t) + \lambda(t - \tau) & \text{if } t \geqslant \tau. \end{cases} \tag{2.2}$$

This fault can be seen in Figure 2.1b. A physical cause for this failure is [48] miscalibration of the sensor.

In [58], λ is chosen from an IID uniform distribution.

2.2.3 Spike fault

A spike fault [38] is defined by the occurrences of "spikes" in the sensor output. To model this, randomly spaced "spikes" of some height $h \neq 0$ are added to the original sensor output after some time point τ. Thus $\tilde{x}(t)$ is modeled as

$$\tilde{x}(t) = \begin{cases} x(t) & \text{if } t < \tau, \\ x(t) + hB(t) & \text{if } t \geqslant \tau, \end{cases} \tag{2.3}$$

where $B(t) \sim \text{IIDBernoulli}(p)$ for some value of $p \in (0, 1)$. This model differs compared to the one presented in [38] in that it does not make any assumption regarding the periodicity of these spikes. This fault can be seen in Figure 2.1c. A physical cause for this fault is [48] battery failure, but also other hardware-related failures.

2.2.4 Erratic fault

This type of fault is also known as [19] Precision degradation fault in some literature. This fault [19] is defined by an increase in the variance of the sensor output. To model this *white noise* (see Definition A.1.1) is added to the original sensor output after some time point τ. Thus $\tilde{x}(t)$ is modeled as

$$\tilde{x}(t) = \begin{cases} x(t) & \text{if } t < \tau, \\ x(t) + \sigma W(t) & \text{if } t \geqslant \tau, \end{cases} \tag{2.4}$$

where $W(t) \sim \text{WN}(0,1)$ (WN denotes white noise, see Definition A.1.1) and the standard deviation $\sigma > 0$. This fault can be seen in Figure 2.1d. Physical causes for this failure include [48] hardware-related failures and low battery.

From Proposition A.1.1 we see how white noise can be generated and from this arises the problem of which underlying probability distribution to use. In [57], the white noise is chosen as $\sigma W(t) \sim \text{IID}\mathcal{N}\left(0, \sigma^2\right)$ where $\sigma^2 \gg \mathbb{V}\text{ar}(x(t))$. In [58] the white noise is also generated from an IID normal distribution but with $\sigma = 0.2$ fixed.

However a normal distribution is unbounded, which might not be desirable when modeling an erratic fault. This is not a problem for most practical purposes, due to the light tails of the distribution.

A more accurate model would also incorporate a time dependence t for the standard deviation $\sigma = \sigma(t)$ to let the precision degrade over time. In [58], different time-dependent models for degradation are proposed.

2.2.5 Stuck fault

In most literature, this type of fault [57, 72, 38, 19] is mostly described as the sensor output being stuck at some fixed value. In this thesis, this value will always be a default value $C \in \mathbb{R}$ specified by the manufacturer. To model this the sensor output is set to the constant C after some time point τ. Thus $\tilde{x}(t)$ is modeled as

$$\tilde{x}(t) = \begin{cases} x(t) & \text{if } t < \tau, \\ C & \text{if } t \geqslant \tau. \end{cases} \tag{2.5}$$

This fault can be seen in Figure 2.1e.

2.2.6 Frozen fault

In this thesis, we make a distinction between this fault and the stuck fault. The reason for making this distinction is a difference in the physical cause of the failure and thus a difference in action to mitigate it. In some literature [49] this fault is known as a stuck-at fault. However, due to the similarly named stuck fault, this thesis will go with the name frozen fault to make the distinction clearer. This fault occurs when the sensor output gets frozen at some particular value. We can thus use the same model for this fault as the one presented in Equation (2.5) by letting $C = x(\tau)$. This fault can be seen in Figure 2.1f. A common physical cause for this fault is [48] hardware malfunction.

2.2.7 Other faults

There are other faults which have been identified in the literature such as cyclic fault [71], random fault [57], gain fault [49] and others. However, these faults have been deemed to be out of the scope of this thesis.

2.3 Time series modeling

In time series analysis some operators are introduced to make notation easier. These operators are extensively used in Section 2.3.1 and thus some proper definitions are justified.

Definition 2.3.1 (Lag operator)**.** For a sequence $\{X(t)\}_{t=0}^{T}$ we define the *lag operator* L as

$$LX(t) = X(t-1), \tag{2.6}$$

for $t > 0$. Moreover, powers of L can be defined by applying Equation (2.6) iteratively

$$L^s X(t) = L^{s-1} LX(t) = L^{s-1} X(t-1) = \ldots = X(t-s), \tag{2.7}$$

for $t > s - 1$.

Note that L might also be referred to as the [10] *Backward shift operator*.

Definition 2.3.2 (Difference operator)**.** For a sequence $\{X(t)\}_{t=0}^{T}$ we also define the *difference operator* Δ, as

$$\Delta X(t) = (1-L) X(t) = X(t) - X(t-1), \tag{2.8}$$

for $t > 0$. Similarly, powers of Δ are given by applying Equation (2.8) iteratively

$$\Delta^s X(t) = \Delta^{s-1} \Delta X(t) = \Delta^{s-1} (1-L) X(t) = \ldots = (1-L)^s X(t). \tag{2.9}$$

To justify Equation (2.8) one should not think of 1 as the number, but rather, as the identity operator acting on $X(t)$. This remark applies to expressions presented in Section 2.3.1 as well.

2.3.1 ARIMA model

An ARIMA process is [10] a time series model which is constructed by combining simpler models. Thus, this section will first present these simpler models and build up the ARIMA model to further motivate it.

An *autoregressive* (AR) process computes the next value as a linear combination of the $p \in \mathbb{N}$ previous values. Thus, an AR(p) process $X(t)$ is given by

$$X(t) = \epsilon(t) + \sum_{i=1}^{p} \phi_i L^i X(t), \tag{2.10}$$

where $\epsilon(t) \sim \mathrm{WN}\left(0, \sigma_\epsilon^2\right)$ denotes the error at time point t. Similarly, a *moving average* (MA) process computes the next value as a linear combination of the $q \in \mathbb{N}$ previous errors. Thus, an MA(q) process $X(t)$ is given by

$$X(t) = \sum_{j=1}^{q} \theta_j L^j \epsilon(t), \tag{2.11}$$

where $\epsilon(t)$ is defined as in Equation (2.10). A process which is given by a linear combination of the p previous values and q previous errors is called an *autoregressive moving average* (ARMA) process. Thus, an ARMA (p, q) process $X(t)$ is given by

$$X(t) = \sum_{i=1}^{p} \phi_i L^i X(t) + \epsilon(t) + \sum_{j=1}^{q} \theta_j L^j \epsilon(t), \tag{2.12}$$

which is normally expressed as,

$$\left(1 - \sum_{i=1}^{p} \phi_i L^i\right) X(t) = \left(1 + \sum_{j=1}^{q} \theta_j L^j\right) \epsilon(t). \tag{2.13}$$

To make Equation (2.13) more compact it is common to introduce the polynomials

$$\phi^{(p)}(z) = 1 - \sum_{i=1}^{p} \phi_i z^i, \tag{2.14}$$

$$\theta^{(q)}(z) = 1 + \sum_{j=1}^{q} \theta_j z^j, \tag{2.15}$$

to make notation easier. Thus by substituting in Equation (2.14) and Equation (2.15), Equation (2.13) simplifies to

$$\phi^{(p)}(L) X(t) = \theta^{(q)}(L) \epsilon(t). \tag{2.16}$$

Moreover, by allowing differencing of order $d \in \mathbb{N}$ to be applied to $X(t)$, such that Equation (2.16) models $\Delta^d X(t)$ we get an *autoregressive integrated moving average* (ARIMA) process, which has been used to [63] detect sensor failures. Thus, an [10] ARIMA (p, d, q) process is given by

$$\phi^{(p)}(L) \Delta^d X(t) = \theta^{(q)}(L) \epsilon(t). \tag{2.17}$$

Note that more constraints apply to both the data and the coefficients that are not presented here. To read up on these a good resource is [10].

2.3.2 Information criteria

When selecting the number of parameters such as determining the values p and q for a model, such as the ARIMA (see Section 2.3.1), a method to [10] decide which values to pick can be determined by choosing the model that minimizes an *information criterion*, where common ones include [12] the so-called AIC and BIC, which are presented below.

The first criterion, *Akaike's information criterion* (AIC) is given by [1]

$$\text{AIC} = 2k - 2 \ln \hat{L}, \tag{2.18}$$

where k denotes the number of estimated parameters and $\hat{L}$ denotes the maximized value of the likelihood function. The AIC is [12] asymptotically an estimator of

the expected Kullback discrepancy of the model. Note that refined versions of the AIC such as [35] the *corrected AIC* (AICc) have been published for various models, including time series-based ones. The reason for this correction is that the AIC can be negatively biased for small sample sizes. However, the fact that this thesis will only consider large sample sizes together with the fact that the expression for the AICc is on a per-model basis compared to the AIC makes it less suitable as a metric.

The *Bayesian information criterion* (BIC) [61], also known as Schwarz's information criterion (SIC), is given by

$$\mathrm{BIC} = k \ln n - 2 \ln \hat{L}, \tag{2.19}$$

where k and $\hat{L}$ are defined as in Equation (2.18) and n is the sample size. From its derivation [47] as an asymptotic approximation of a transformed Bayesian posterior probability for a model, the model that is favored by the BIC is the model which is most probable given the observations when the number of observations is large.

The two information criteria are closely related to each other. An advantage of the BIC compared to the AIC is that the sample size is taken into account when comparing models.

2.3.3 Dynamic time warping

A method for determining a "distance" between two time series is to compute the so-called *dynamic time warping* (DTW) distance [59], see Figure 2.2 for a comparison to the Euclidean distance. To compute the DTW distance between two sequences x and y of length m and n respectively one solves the following *dynamic programming* [4] problem defined by the Bellman equation [55]

$$\mathrm{OPT}\,(i,j) = d\,(x\,(i)\,, y\,(j)) + \min \begin{cases} \mathrm{OPT}\,(i-1,j)\,, \\ \mathrm{OPT}\,(i,j-1)\,, \\ \mathrm{OPT}\,(i-1,j-1)\,, \end{cases} \tag{2.20}$$

base case

$$\mathrm{OPT}\,(1,1) = d\,(x\,(1)\,, y\,(1))\,, \tag{2.21}$$

and the constraint that $i, j > 0$. The DTW between x and y is then given by $\mathrm{DTW}\,(x,y) = \mathrm{OPT}\,(m,n)$. Note that in most cases the distance function d in Equations (2.20) and (2.21) is usually defined as [55]

$$d\,(x,y) = \|x - y\|^2. \tag{2.22}$$

This distance metric [55] has become popular to use in classification tasks where the data consists of sequences, an example of which is time series. A popular example is to use the DTW distance in conjunction with another ML method to perform classification tasks, e.g. kNN (see Section 2.4.2). The general idea is that time series labeled with the same class should have a smaller DTW distance between each other, compared to time series labeled with different classes.

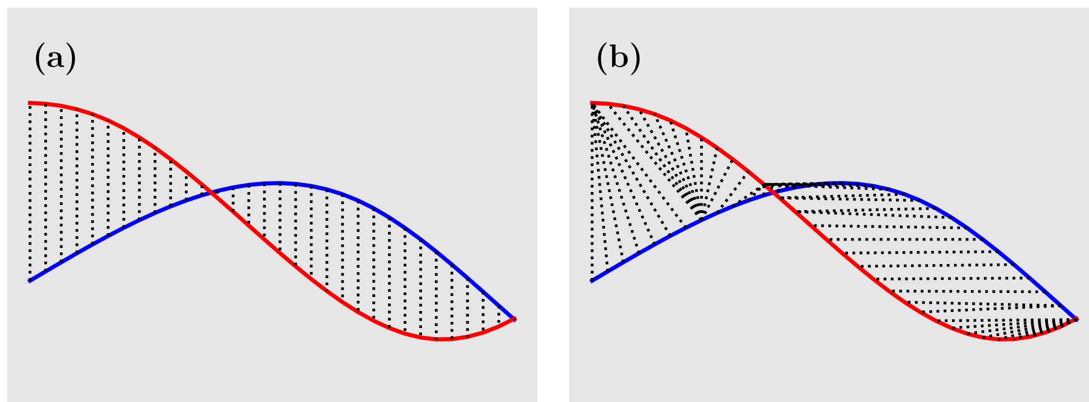


Figure 2.2: A graphical overview of the difference between computing the **a)** Euclidean distance and **b)** DTW distance between two sequences. Note that in **a)**, the distance is computed between the sequences for each time point, while in **b)**, the distance is computed from an optimal alignment according to Equation (2.20). Figure created with [34].

2.4 Machine learning methods for anomaly detection

There is already an abundance of *machine learning* (ML) methods for detecting anomalies in the literature [22], and in recent years numerous new ideas, methods and algorithms have been presented [55], and efforts have been made to apply other algorithms, such as *transformers* [68, 70] to this area. However, we should mainly consider algorithms which provide robustness and reliability and have a proven track record for this task. Moreover, because we have labeled data, only supervised methods will be considered.

Note that methods discussed in Section 2.4.3 and Section 2.4.6 require prior knowledge of *artificial neural networks* (ANN). A good resource for learning about these concepts is [28].

2.4.1 Extremely randomized trees

The usage of *extremely randomized trees* or *extra trees* for detecting anomalies in sensors has been proposed in [57]. Originally proposed in [25] it is an ensemble algorithm very similar to that of *random forests* [9]. The main difference is that splits are picked randomly in extra trees, compared to random forest which uses the best-split strategy.

Other forest-based methods, like gradient-boosted algorithms such as *XGBoost* [13] were considered but did not seem to yield any significant improvement over extra trees.

2.4.2 K-nearest neighbors

A common algorithm in classification tasks is *k-nearest neighbors* (kNN) [23]. It classifies a new observation by picking the majority class of its k nearest neighbors according to some distance metric d. The idea is that similar inputs should correspond to similar labels. For a new point $x \in \mathbb{R}^{n_s}$ and a training set $\mathcal{T} = \{(x', y')\}$ we can, as in [69], define the k-nearest neighbors of x, $S_x \subseteq \mathcal{T}$ where $|S_x| = k$ and

$$d(x, x') \geqslant \max_{(x'', y'') \in S_x} d(x, x'') \text{ for all } (x', y') \in \mathcal{T} \setminus S_x. \tag{2.23}$$

So every point in $\mathcal{T} \setminus S_x$ is at least as far away from x as the furthest away point in S_x with respect to distance metric d. An estimate of the label corresponding to x is then computed as

$$\hat{y}(x) = \text{mode}\left(\{y' : (x', y') \in S_x\}\right). \tag{2.24}$$

Note that we have not specified what happens in case of a draw in Equation (2.24) or if S_x is not unique. This is usually defined by the implementer, although a common strategy in the case of equality in Equation (2.24) is to look at the $k-1$ nearest neighbors while $k > 1$.

Note that in a high-dimensional space, the concept of the nearest neighbor loses its meaning [7], because measures like the Euclidean distance become less meaningful. This is related to the *curse of dimensionality* [4]. For classifying time series, which can be seen as high-dimensional objects, a common choice of this distance metric d is to compute the DTW distance (see Section 2.3.3) between the two time series.

2.4.3 Recurrent neural networks

Recurrent neural networks (RNN) [56] are a family of ANNs that contain recurrent connections, which allow for information to persist in the network. So information from far back can influence the current output. This is done by storing the information in a so-called *hidden state* h. This makes these types of networks a natural choice when working with sequence-based data [28], like time series. An example of such a network can be seen in Figure 2.3a and the same network *unfolded through time* can be seen in Figure 2.3b.

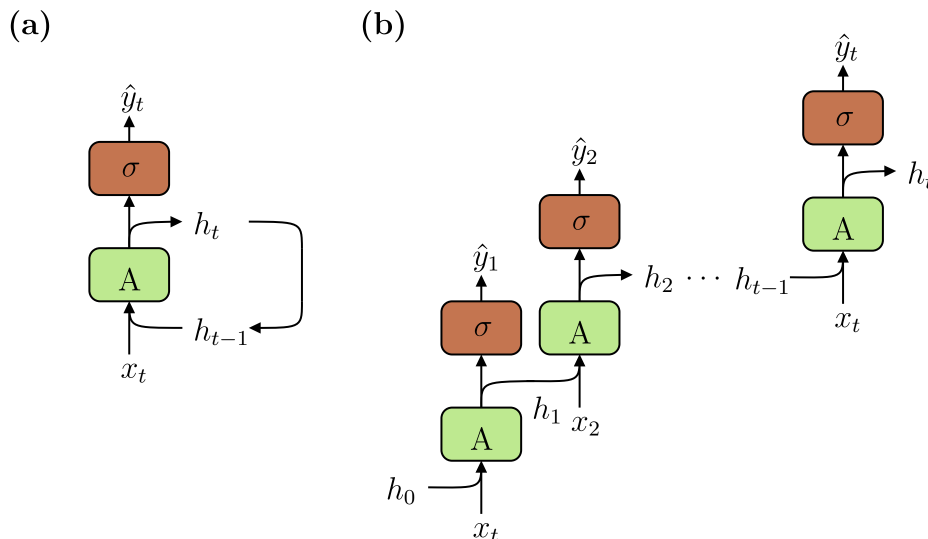


Figure 2.3: A graphical overview of an RNN, **a)** shows a single neuron while **b)** shows the network unfolded through time. Note that affine transformations are not explicitly shown. The box labeled A is meant to represent some architecture, a common one is the so-called Elman RNN [21] which can be seen in Figure 2.4.

To compute the outputs [28] of the network in Figure 2.3 we first compute the current hidden state $h(t)$ from the current input $x(t)$ and previous hidden state $h(t-1)$. This is done through some architecture labeled A in Figure 2.3. A common architecture is to use the so-called *Elman RNN* [21], an example of which can be seen in Figure 2.4.

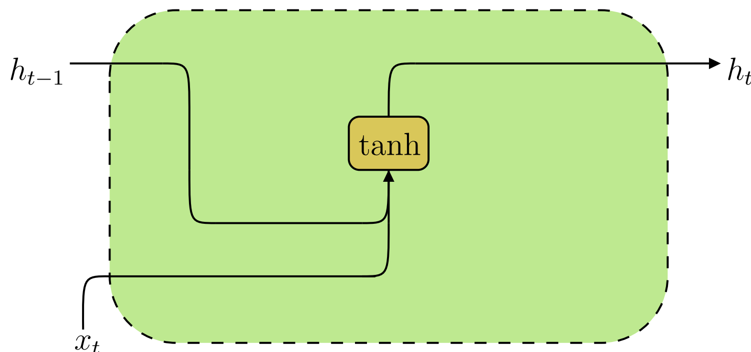


Figure 2.4: A graphical overview of the architecture for an Elman RNN with a tanh activation function. Note that there are examples where a *rectified linear unit* (ReLU) activation function is used instead.

The hidden state in an Elman RNN is updated according to

$$h(t) = \tanh\left(W_x x(t) + b_x + W_h h(t-1) + b_h\right). \tag{2.25}$$

Note that in Equation (2.25), the two bias vectors, b_x and b_h are combined to a single bias vector $b = b_x + b_h$ in practice. This thesis will still present them separately, as

we can think of this as two affine transformations being applied to $x(t)$ and $h(t-1)$ respectively in Equation (2.25). From the hidden states, the current output $\hat{y}(t)$ in Figure 2.3 can be computed as

$$\hat{y}(t) = \sigma\left(W_o h(t) + b_o\right). \tag{2.26}$$

In this context, σ refers to a *sigmoid function*, usually the logistic function

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{2.27}$$

When acting on a multi-dimensional input, the function is assumed to act element-wise. The same holds for the tanh function in this context.

A common reason for using the sigmoid function in Equation (2.27) is [28, 3], that it corresponds to the cumulative distribution function for a $\text{Logistic}(0,1)$ distributed random variable, which means that the output can be interpreted as a probability $\sigma(x) = \mathbb{P}(X \leqslant x)$. This probability is then usually interpreted as input x corresponds to label 1 with probability $\sigma(x)$. Note that in literature, x is often referred to as a *logit* in this context.

2.4.4 Cross-entropy

By allowing the output to be interpreted as a probability, the loss in the network can be measured as the difference between the predicted labels $\hat{y}$ and the true labels y in terms their of distributions. A common metric to use when determining the difference between two distributions is to use the so-called *cross-entropy* [16], or *negative log-likelihood* between the distributions. For this setting, the cross-entropy H can be computed as

$$H(y,\hat{y}) = -y\ln\hat{y} - (1-y)\ln(1-\hat{y}). \tag{2.28}$$

From Equation (2.28) one can define the loss function

$$l = \sum_{(x,y)\in\mathcal{T}} H\left(y, \hat{y}(x)\right), \tag{2.29}$$

where $\mathcal{T}$ is the training set. Here, $\hat{y}(x)$ is thought of as the output from the entire network given the input x, to avoid confusion with the notation in Equation (2.26).

2.4.5 Multiclass classification

Note that the sigmoid function presented in Equation (2.27) can be extended to handle multiclass classification problems [28]. This extension is the so-called *softmax function* $\text{softmax} : \mathbb{R}^K \to (0,1)^K$ and is given by

$$\text{softmax}(x)_i = \frac{e^{-x_i}}{\sum_{j=1}^{K} e^{-x_j}}, \tag{2.30}$$

for $i = 1,\ldots,K$. The interpretation is similar to that of the sigmoid function, in that input x corresponds to class i with probability $\text{softmax}(x)_i$, for $i = 1,\ldots,K$.

Note that the sigmoid function can be seen as a special case of the softmax function when the number of classes $K = 2$.

Similarly, the concept of cross-entropy in Equation (2.28) can be extended to handle K different classes and is given by

$$H(y, \hat{y}) = -\sum_{i=1}^{K} y_i \ln \hat{y}_i, \tag{2.31}$$

where y is the zero vector except for one element which indicates which class it corresponds to. So y corresponds to class i if $y_j = \delta_{ij}$ for $j = 1, \ldots, K$, where δ is the Kronecker delta. From this extension of the cross-entropy a similar loss function to that of the one presented in Equation (2.29) can be constructed.

2.4.6 Long short-term memory

A proven ML method for handling time series data is [33] the *Long short-term memory* (LSTM), seen in Figure 2.5. It is an RNN architecture similar to that of the Elman RNN, but a bit more complex which allows it to capture long-term dependencies very well. Simple RNNs, such as the Elman RNN fail to capture long-term dependencies due to the so-called *vanishing gradient problem* [5] that occurs in gradient-based training methods such as *backpropagation through time* (BPTT). The LSTM however, does not suffer from this problem [50]. In the LSTM, a new *cell state* c is introduced alongside the hidden state.

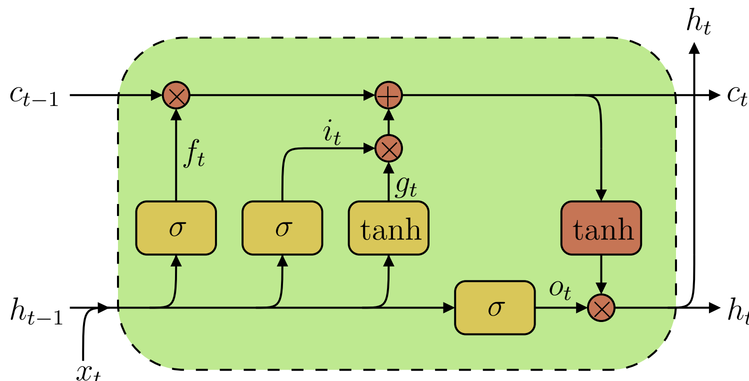


Figure 2.5: A graphical overview of an LSTM memory cell. Note that like in Figure 2.4, affine transformations are not explicitly shown.

The first part of the LSTM consists of a so-called *forget gate* and is given by

$$f(t) = \sigma\left(W_{f,x} x(t) + b_{f,x} + W_{f,h} h(t-1) + b_{f,h}\right). \tag{2.32}$$

In Equation (2.36) we interpret this as $f(t)$ determining how much the current cell state $c(t)$ should be influenced by the previous state $c(t-1)$. So if $f(t) = 0$ the previous state is "forgotten", hence the name forget gate. The second part consists of a so-called *input gate* and is given by

$$i(t) = \sigma\left(W_{i,x} x(t) + b_{i,x} + W_{i,h} h(t-1) + b_{i,h}\right). \tag{2.33}$$

Equation (2.33) gets its name from the fact that it regulates how much the current input data from the so-called *input node*, which is given by

$$g\left(t\right) = \tanh\left(W_{g,i}x\left(t\right) + b_{g,x} + W_{g,h}h\left(t-1\right) + b_{g,h}\right), \tag{2.34}$$

should be taken into account when computing the current cell state $c\left(t\right)$ in Equation (2.36). The last gate is the so-called *output gate* and is given by

$$o\left(t\right) = \sigma\left(W_{o,x}x\left(t\right) + b_{o,x} + W_{o,h}h\left(t-1\right) + b_{o,h}\right). \tag{2.35}$$

It determines how the output should depend on the current cell state in Equation (2.37). Lastly, the current cell state $c\left(t\right)$ and the current hidden state $h\left(t\right)$ is updated by

$$c\left(t\right) = f\left(t\right) \odot c\left(t-1\right) + i\left(t\right) \odot g\left(t\right), \tag{2.36}$$

$$h\left(t\right) = o\left(t\right) \odot \tanh\left(c\left(t\right)\right), \tag{2.37}$$

where $\odot$ denotes the Hadamard product (i.e. element-wise product).

2.4.7 Gated recurrent unit

A more recent RNN architecture that is similar to that of the LSTM is [15] the *gated recurrent unit* (GRU), seen in Figure 2.6. Like the LSTM, it also builds on a similar set of gating mechanisms and has outperformed Elman RNNs on different tasks and shown comparable results to that of the LSTM while having fewer parameters.

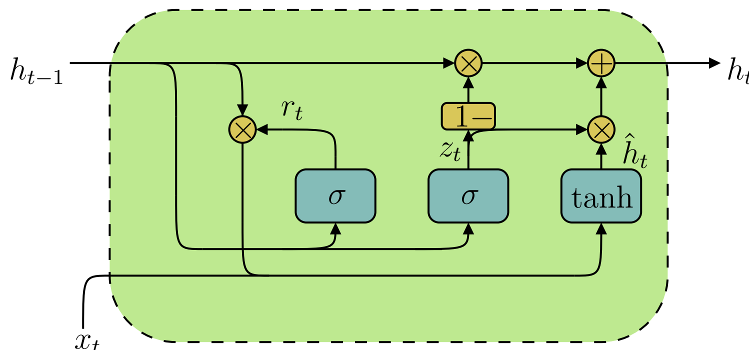


Figure 2.6: A graphical overview of a GRU memory cell. Note that like in Figure 2.4 and Figure 2.5, affine transformations are not explicitly shown.

The first part of the GRU consists of a so-called *reset gate* and is given by

$$r\left(t\right) = \sigma\left(W_{r,x}x\left(t\right) + b_{r,x} + W_{r,h}h\left(t-1\right) + b_{r,h}\right). \tag{2.38}$$

We can think of $r\left(t\right)$ as determining how much the current hidden state candidate $\hat{h}\left(t\right)$ defined in Equation (2.40) should be influenced by the previous state $h\left(t-1\right)$. So if $r\left(t\right) = 0$ the hidden state is "reset", hence the name forget gate. This gate serves a similar purpose to that of the forget gate in Equation (2.32) in the LSTM. The next part consists of a so-called *update gate* which is given by

$$z\left(t\right) = \sigma\left(W_{z,x}x\left(t\right) + b_{z,x} + W_{z,h}h\left(t-1\right) + b_{z,h}\right). \tag{2.39}$$

Equation (2.39) gets its name from the fact that it determines how the next hidden state $h(t)$ should be updated. But before the hidden state can be updated, a new candidate for $h(t)$ is computed as

$$\hat{h}(t) = \tanh\left(W_{h,x}x(t) + b_{h,x} + W_{h,h}\left(r(t) \odot h(t-1)\right) + b_{h,h}\right). \tag{2.40}$$

Lastly, the next hidden state is computed as a convex combination of the previous hidden state $h(t)$ and the new candidate $\hat{h}(t)$ as

$$h(t) = (1 - z(t)) \odot h(t-1) + z(t) \odot \hat{h}(t). \tag{2.41}$$

2.5 Evaluation metrics

In this section, a couple of metrics to be used to evaluate models and classifiers are presented.

2.5.1 Confusion matrix

A common way to organize predicted data for a classification model is to use a 2x2 contingency table [54] called a *confusion matrix* which shows the frequency of each combination of predicted and true labels.

Table 2.1: A common way to organize binary classifications is in a so-called confusion matrix which is a 2×2 contingency table over the four possible combinations of predicted and true labels.

True class \ Predicted class	N̂	P̂	**total**
N	TN	FP	TN + FP
P	FN	TP	FN + TP
total	TN + FN	FP + TP	N + P

It is also possible to extend Table 2.1 for multiclass classification problems. For a problem with K classes, it is possible to organize them analogously in a $K \times K$ contingency table.

One of the most common metrics *accuracy*, can be expressed in terms of the quantities presented in Table 2.1 as

$$\text{accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{P} + \mathrm{N}}. \tag{2.42}$$

2.5.2 Precision, Recall and F_1 score

Other common metrics for evaluating binary classifiers include *precision* and *recall* [54] which can be computed from the values in Table 2.1 as

$$\text{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{2.43}$$

$$\text{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}. \tag{2.44}$$

A common way to capture both precision and recall is to compute the so-called F_1 *score*, which is the harmonic mean of precision and recall. This measurement is also known as the Sørensen-Dice coefficient [64, 17] in statistics. From the Equations (2.43) and (2.44) we arrive at the form

$$\mathrm{F}_1 = \frac{2\mathrm{TP}}{2\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}. \tag{2.45}$$

2.5.3 Receiver operating characteristic

Another common metric used to evaluate binary classification tasks is based on the *Receiver operating characteristics* (ROC) [54] curve, or more precisely, the area under this curve, known as the *area under curve* (AUC). In this thesis, this metric will be referred to as the ROC AUC. The ROC curve is constructed by graphing the *true positive rate* (TPR), which is equal to the recall given in Equation (2.44), as a function of the *false positive rate* (FPR) given by

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}. \tag{2.46}$$

A possible way to extend this metric is to [31] compute the ROC AUC between all classes pairwise and take the average for each class.

2.5.4 Matthews correlation coefficient

Another popular metric for the evaluation of binary classifiers in ML is *Matthews correlation coefficient* (MCC) [45] and is given in terms of quantities obtained from the values in Table 2.1 by

$$\mathrm{MCC} = \frac{\mathrm{TP} \cdot \mathrm{TN} - \mathrm{FP} \cdot \mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})}}. \tag{2.47}$$

There is an advantage [14] to using this metric over, e.g., the F_1 score for binary classification problems, in that it is less optimistic. However, this advantage applies more to problems which involve imbalanced datasets.

In statistics, this measurement is also known as the ϕ coefficient or Yule's ϕ coefficient [73] and can be obtained by computing Pearson's correlation coefficient between the true labels and predicted labels.

There are also possible extensions of the MCC to handle multiclass classification problems with K different classes [29].

2.6 Bootstrap

Bootstrap is a method [20] to estimate statistical properties (e.g., mean, variance etc.) for a population from only a finite sample, without making assumptions about the distribution these samples are generated from. The algorithm can be seen in Algorithm 2.1 and is used in the bootstrap-based method presented in Section 3.2.3.

Algorithm 2.1 Bootstrap

Input: Dataset $\mathcal{X}$, sample size m, bootstrap samples n
Output: Bootstrap samples Θ
1: $\Theta \leftarrow \emptyset$
2: **for** $i = 1, \ldots, n$ **do**
3: Draw a sample $\mathcal{X}^{(\text{sample})}$ of size m with replacement from $\mathcal{X}$.
4: Compute θ from $\mathcal{X}^{(\text{sample})}$
5: Add θ to Θ
6: **return** Θ

3

Method

In this chapter, the method and process in which the results were gathered are presented.

3.1 Dataset

The dataset was provided by Ericsson. It consists of aggregated data points where each data point is the average of 120 measurements measured at 1 Hz. So each data point corresponds to the average over two minutes. In total, the data for six time series, each consisting of eight temperature sensors, over the span of a 48-hour cycle ($2n_x = 1,440$ data points) were provided. This dataset consisted of healthy data, so to obtain any faulty data, this had to be generated. By utilizing the models presented in Section 2.2 faulty data could be generated, see Sections 3.1.1 and 3.1.2. Note that using generated faulty data to train and evaluate anomaly detection algorithms has been used in [57, 58]. An overview of the entire pipeline can be seen in Figure 3.1.

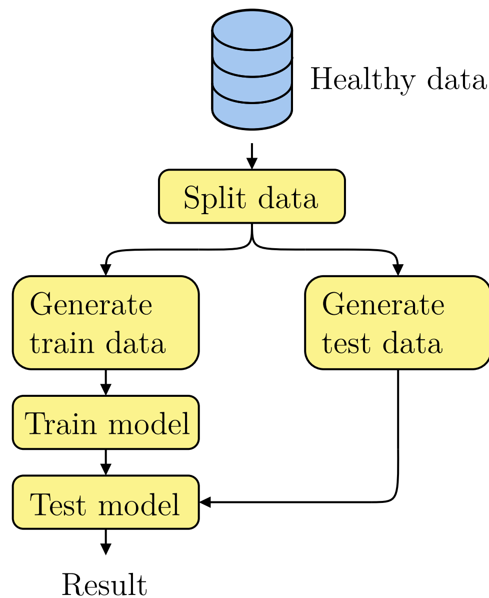


Figure 3.1: A schematic overview of the method. Note that the split occurs before any data is generated according to Algorithm 3.2 to ensure that no single part of the time series is present in both the training and the test sets.

Note that when training the ARIMA model (see Section 3.2.4), the method is different.

The train test split in Figure 3.1 was constructed by taking the first half of each time series and putting it in the training set and putting the second half in the test set, this is a variant of a 50-50 train test split.

3.1.1 Fault generation

To generate faulty data Algorithm 3.1 was used with parameters determining the start time set to $\tau_{\min} = 1$ and $\tau_{max} = \lfloor \frac{n_w + 1}{2} \rfloor$ respectively. The specific parameters for each fault type were chosen according to Table 3.1.

Note that in the following algorithms, we will use the following notation for a submatrix. We let $A[i_1 : i_2][j_1 : j_2]$ refer to the submatrix of A that consists of the intersection of rows $i_1, \dots, i_2$ and columns $j_1, \dots, j_2$. Moreoever, we will also let $A[i][j_1 : j_2]$ refer to the submatrix of A that consists of the intersection of row i and columns $j_1, \dots, j_2$. We will also let $A[i][j]$ refer to the single element that consists of the intersection of row i and column j.

Algorithm 3.1 Fault generation

Input: window of healthy data $x \in \mathbb{R}^{n_s \times n_w}$, fault type f
Output: Faulty data $\tilde{x}$
1: $\tilde{x} \leftarrow x$
2: Pick random sensor $i \sim \mathcal{U}\{1, \dots, n_s\}$
3: Pick random starting time for fault $\tau \sim \mathcal{U}\{\tau_{\min}, \dots, \tau_{\max}\}$
4: $P \leftarrow$ parameters for fault type f given in Table 3.1
5: Generate fault of type f with parameters P in $\tilde{x}[i][\tau : n_w]$
6: **return** $\tilde{x}$

Note that the input data in Algorithm 3.1 is a window and not an entire time series, the reason for this will be explained in Section 3.2.1. Also, note that by generating a fault on line 5, the model corresponding to fault type f in Section 2.2 is applied with parameters P on the row vector $x[i][\tau : n_w]$.

Table 3.1: Parameters used when generating faults according to Algorithm 3.1. Note that frozen fault (see Section 2.2.6) is not present in this table since it is non-parametric. The values and intervals of these parameters were verified with field experts at Ericsson.

Fault type	Parameter(s)
Hardover	$\delta \sim \mathcal{U}[3.0, 5.0)$ [1]
Drift	$\lambda \sim \mathcal{U}[0.2, 0.3)$
Spike	$h = \pm 4.0\,°C$ and $p = 0.1$ [2]
Erratic	$\sigma = 2.0\,°C$ [3]
Stuck	$C = 0.0\,°C$

3.1.2 Data generation

Training and testing data sets were generated according to Algorithm 3.2. On line 6, some random noise is added to the input. The reason for picking a standard deviation of 0.01 for the noise is that $0.01^2 \ll \mathbb{V}\text{ar}(x(t))$, so it will not have too much of an impact on the patterns.

The reason for adding this noise to the input is because we may, depending on the number of rounds n_r that Algorithm 3.2 runs, add the same window pair multiple times to our set of window pairs $\tilde{x}$. This becomes quite probable for high values of n_r and low values of p. To make this less probable, we add a small unique perturbation for each window pair.

Note that a window pair in Algorithm 3.2 is a tuple of $n_s \times n_w$ matrices.

Algorithm 3.2 Data generation

Input: window size n_w, no. rounds n_r, healthy time series $x \in \mathbb{R}^{n_s \times n_x}$, fault type f, fault probability $p \in (0, 1)$
Output: set of window pairs $\tilde{x}$

1: $\tilde{x} \leftarrow \emptyset$
2: **for** $i_r = 1, \ldots, n_r$ **do**
3: **for** $i_w = 1, \ldots, n_x - 2n_w$ **do**
4: $x^{(1)} \leftarrow x[1 : n_s][i_w : i_w + n_w]$
5: $x^{(2)} \leftarrow x[1 : n_s][i_w + n_w + 1 : i_w + 2n_w]$
6: draw $Z^{(1)}, Z^{(2)} \in \mathbb{R}^{n_s \times n_w}$ with elements from a $\mathcal{N}\left(0, 0.01^2\right)$ distribution
7: $x^{(1)} \leftarrow x^{(1)} + Z^{(1)}$
8: $x^{(2)} \leftarrow x^{(2)} + Z^{(2)}$
9: draw X from a Bernoulli(p) distribution
10: **if** $X = 1$ **then**
11: $x^{(2)} \leftarrow$ output from Algorithm 3.1 with inputs $x^{(2)}$ and f
12: add $\left(x^{(1)}, x^{(2)}\right)$ to $\tilde{x}$
13: **return** $\tilde{x}$

The reason for generating window pairs instead of just windows is that some algorithms, like the baseline method (see Section 3.2.2), require the previous window. As can be seen in Algorithm 3.2, only the second window can contain a fault.

Since the algorithms are supposed to work online, no prior data preprocessing, like normalization, can be assumed. So such steps would have to be incorporated into the algorithm should it be desired.

[1]The reason for using this half-open interval is due to the underlying algorithm used by [32] when generating random floating point numbers. Note that for practical purposes this does not matter.

[2]Note that when generating spike faults with Algorithm 3.1, one spike will always appear at τ and spikes will appear with probability p for next $\tau + 1, \ldots, n_w$ time points.

[3]$\sigma = 2.0$ was chosen since $4.0 \gg \mathbb{V}\text{ar}(x(t))$.

To generate data with $K > 2$ classes, Algorithm 3.2 can be used repeatedly. To generate data for the multiclass case where we want to detect in which sensor a fault has occurred, we simply run Algorithm 3.2 for each sensor $i = 1, \ldots, n_s$ and specify that the fault should occur in sensor i instead of picking it at random on line 2, and assign the corresponding class accordingly. Then we simply concatenate all the generated sets. Similarly, to generate data for the multiclass case where we want to detect which type of fault has occurred, we simply run Algorithm 3.2 for each fault type f and specify that the fault type should be f and assign the corresponding class accordingly and concatenate the result.

Note that to generate a balanced set consisting of $K+1$ classes (where healthy data represents one class), we can run Algorithm 3.2 K times and let the probability of fault occurring be $\dfrac{K}{K+1}$.

The reason for not combining the two multiclass classification tasks into one single task is due to the multiplicative number of classes this would produce. Note that in some radios, $n_s \gg 10$. Hence, this is split into two separate tasks.

3.2 Algorithms

In this section algorithms to detect and classify faults in sensors are presented.

3.2.1 Sliding window

The algorithms presented will be based on a *sliding window* method, with n_w denoting the size of the window, see Figure 3.2. This window slides over the original time series with a fixed size and stride. In this thesis, the stride is set to one sample as this is required for the number of samples analysis (see Section 3.3). Note that the window can be seen as a $n_s \times n_w$ matrix.

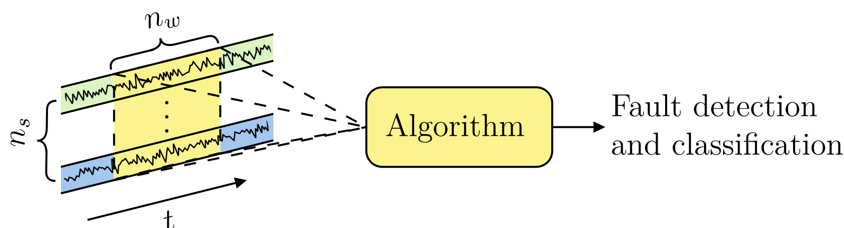


Figure 3.2: Illustration of a sliding window of size n_w operating on multivariate time series with n_s sensors. This is a good method for online detection of anomalies.

It should be noted that, for some of the methods presented in this chapter, a sliding window might not be the most natural way to view these methods. On the other hand, this offers a framework in which the different methods can be compared against each other.

3.2.2 Baseline method

A heuristic baseline method based on simple statistical properties was also developed. The reason for this was to have a basic method which could easily be interpreted, implemented and computed on real data. The method operates on windows of given size n_w for n_s different sensors. The metrics computed for sensor i during the j:th window are

$$\hat{\mu}_j[i] = \frac{1}{n_w}\sum_{k=1}^{n_w} x[i][j+k] \tag{3.1}$$

$$\hat{\sigma}_j[i] = \sqrt{\frac{1}{n_w-1}\sum_{k=1}^{n_w}\left(x[i][j+k]-\hat{\mu}_j[i]\right)^2} \tag{3.2}$$

$$\hat{m}_j[i] = \min_{k=1,\ldots,n_w} x[i][j+k] \tag{3.3}$$

$$\hat{M}_j[i] = \max_{k=1,\ldots,n_w} x[i][j+k] \tag{3.4}$$

Note that the vector $\hat{\mu}_j$ in Equation (3.2) is computed according to Equation (3.1).

Now by also assuming that a fault can occur in at most one sensor in a given window an algorithm can detect faults by detecting changes in these metrics between two consecutive disjoint windows and classify these errors depending on whether or not they exceed some threshold, $\alpha_{\mathcal{M}}^{(i)}$ for sensor i and metric $\mathcal{M}$. Here the absolute difference was used to measure the difference between two consecutive disjoint windows.

However, a simplification was made in that the different $\alpha_{\mathcal{M}}^{(i)}$ are considered to be the same for the different sensors. Thus the number of parameters is reduced from $4n_s$ to just 4. The values chosen for these parameters can be seen in Table 3.2.

Table 3.2: Parameters used for the baseline method (see Algorithm 3.3). Note that the same parameter is used across all sensors, so $\alpha_{\mathcal{M}}^{(i)} = \alpha_{\mathcal{M}}$ for $i = 1, \ldots, n_s$.

Parameter	Value
Mean α_μ	2.0
Standard deviation α_σ	1.0
Minimum $\alpha_{\min}$	1.0
Maximum $\alpha_{\max}$	1.0

For the metrics in Equations (3.1), (3.3) and (3.4) it is reasonable that these would vary to a greater extent when the window size n_w is increased. To capture this, their threshold is multiplied by a factor $\ln(1+n_w)$.

Note that in Algorithm 3.3 we extend Definition 2.3.2 to also work on row vectors. So for $x \in \mathbb{R}^{1\times n}$ where $n > 1$ we define $\Delta x \in \mathbb{R}^{1\times(n-1)}$ as the row vector with elements $\Delta x[i] = x[i+1] - x[i]$ for $i = 1, \ldots, n-1$.

Algorithm 3.3 Baseline method

Input: set of window pairs x
Output: predictions $\hat{y}$

```
1: ŷ ← ∅
2: for each window pair (x^(1), x^(2)) in x do
3:     current prediction ŷ_w ← 0
4:     for each metric type M do.
5:         for each sensor i = 1, ..., n_s do
6:             compute metric M̂^(1) from x^(1)[i][1 : n_w]
7:             compute metric M̂^(2) from x^(2)[i][1 : n_w]
8:             δ^(M) ← |M̂^(2) − M̂^(1)|
9:             threshold factor κ ← 1
10:            if metric type M is not standard deviation then
11:                κ ← κ ln(1 + n_w)
12:            if δ^(M) ⩾ α_M κ then
13:                ŷ_w ← 1
14:    stuck counter s ← 0
15:    for each sensor i = 1, ..., n_s do
16:        compute Δx^(2)[i][1 : n_w]
17:        j ← index of the last nonzero element in Δx^(2)[i][1 : n_w]
18:        if j < n_w/2 then
19:            s ← s + 1
20:    if s = 1 then
21:        ŷ_w ← 1
22:    add ŷ_w to ŷ
23: return ŷ
```

In Algorithm 3.3 the metric $\mathcal{M}$ is one of mean, standard deviation, minimum and maximum. The method also includes a simple step to detect frozen faults, by looking for sensors that measure the same value between any two consecutive time points. The if statement on line 18 looks for sensors that are stuck for the last half of the window. Later on line 20 it checks if the number of sensors that are stuck is 1, the reason for checking for equality instead of inequality is because of the assumption that at most one fault can occur in at most one sensor in a given time point. If several sensors appear to be “frozen” it may very well be the case that the measured value is constant.

For the multiclass classification case, we can extend Algorithm 3.3 by thinking of running it on a single sensor. We can then do that for all n_s sensors. If no sensor is faulty we label the input as healthy, if one sensor is faulty we label the input as corresponding to that class. If multiple sensors are faulty, we label the input corresponding to the sensor with the lowest number. The reason for doing this is that we want to be able to determine the class deterministically.

It should also be noted that this method is very similar to the rule-based method proposed in [63].

3.2.3 Bootstrap method

Another method based on the metrics computed in Section 3.2.2 is a bootstrap-based method. A normal prerequisite to applying bootstrap is that the sampled data are independent [20]. Although there are methods for working with dependent data [40], these methods are significantly more complicated to work with. To determine whether or not the samples are independent a Pearson's χ^2 test [52] was performed (see Section C.1 for details). It was found that independence between the variance of two adjacent windows, see Equation (3.2), could not be ruled out at a 95 % confidence level. For the rest of the measurements, independence between windows could be ruled out.

To perform this test on the sampled data $\{O_i\}_{i=1}^{n}$ the observations were split into two sets $O^{(\text{odd})}$ and $O^{(\text{even})}$ (here $O^{(\text{odd})}$ refers to observations having an odd index i and vice versa for $O^{(\text{even})}$) for which a 4×4 contingency table we constructed. The reason for choosing a 4×4 is that any larger size would have entire rows or columns filled with zeros (this results in a division by zero, see Equations (C.1) and (C.2)). A method to combat this in practice is to add pseudo counts. However, due to the relatively small sample size, this was not deemed to be a viable solution. To assign the measurements to perform this test a 2d histogram was constructed and observations were put into different equidistantly spaced bins.

The quantiles were estimated from $k = 10000$ bootstrap samples (see Algorithm 2.1), each with a sample size of $m = 200$. To compute the quantiles from the bootstrap sample, the default method in [32] was used (See Definition 7 in [36]). The final quantiles were estimated by taking the median of the bootstrap samples [42]. The statistical significance for this test was chosen to be $\alpha = 0.05$.

Algorithm 3.4 Bootstrap method

Input: set of window pairs x, significance level α
Output: predictions $\hat{y}$

```
1: ŷ ← ∅
2: for each window pair (x^(1), x^(2)) in x do
3:     current prediction ŷ_w ← 0
4:     for each sensor i = 1, ..., n_s do
5:         compute σ̂_w^(i) from x^(2)[i][1 : n_w] according to Equation (3.2)
6:         if not q^{-1}(α/2) ⩽ (σ̂_w^(i))^2 ⩽ q^{-1}(1 − α/2) then
7:             ŷ_w ← 1
8:     add ŷ_w to ŷ
9: return ŷ
```

For the multiclass classification case, we can extend Algorithm 3.4 similarly to how we can extend Algorithm 3.3.

3.2.4 ARIMA method

It is possible to use ARIMA models for prediction. Note that Equation (2.17) can be rewritten as

$$\Delta^d X(t) = \epsilon(t) + \sum_{i=1}^{p} \phi_i \Delta^d L^i X(t) + \sum_{j=1}^{q} \theta_j L^j \epsilon(t), \tag{3.5}$$

where $X(t)$ can be computed from $\Delta^d X(t)$ given $LX(t), \ldots, L^d X(t)$. So by assuming a distribution for $\epsilon(t)$, a prediction interval for new observation can be constructed. Common assumptions are $\epsilon(t) \sim \mathcal{N}\left(0, \hat{\sigma}^2\right)$ or $\epsilon(t) \sim t_\nu$. For *maximum likelihood estimation* (MLE), $\epsilon(t) \sim \mathcal{N}\left(0, \hat{\sigma}^2\right)$ is usually assumed. A classifier was constructed by computing an interval for one-step-ahead prediction. Now, for some significance level α, we classify a window as containing a fault if an observation lies outside the prediction interval, i.e. if

$$X(t) \notin \left(\hat{X}(t) - \hat{\sigma}\Phi^{-1}\left(\frac{\alpha}{2}\right), \hat{X}(t) + \hat{\sigma}\Phi^{-1}\left(1 - \frac{\alpha}{2}\right)\right), \tag{3.6}$$

where Φ denotes the *cumulative distribution function* (CDF) of a standard normal distributed variable. This is similar to the one-step-ahead time series method described in [63].

To maximize the likelihood function the *Limited-memory Broyden–Fletcher-Goldfarb Shanno* (LBFGS) algorithm [11, 24, 27, 62, 43] was used. This was the optimizer which showed the best performance. However, it did fail to converge in some cases. In such cases, the AIC on line 5 was set to a `nan` value in Algorithm 3.5. The argmax on line 6 was set to ignore `nan` values.

Algorithm 3.5 ARIMA training

Input: Healthy data $x \in \mathbb{R}^{n_s \times n_x}$, window size n_w
Output: ARIMA model $\mathcal{M}$
1: $A \leftarrow \mathbf{0}^{(n_w+1)\times(n_w+1)}$
2: **for** $p = 0, \ldots, n_w$ **do**
3:     **for** $q = 0, \ldots, n_w$ **do**
4:         $\mathcal{M} \leftarrow$ MLE estimate of an ARIMA $(p, 1, q)$ model fitted to x
5:         $A[p][q] \leftarrow$ AIC for $\mathcal{M}$ computed according to Equation (2.18)
6: $p, q \leftarrow \underset{p,q}{\operatorname{argmin}} A[p][q]$
7: $\mathcal{M} \leftarrow$ MLE estimate of an ARIMA $(p, 1, q)$ model fitted to x
8: **return** $\mathcal{M}$

Note that the model selected in Algorithm 3.5 is only with respect to the AIC. This is because the AIC is a better metric when forecasting is the goal, whilst the BIC is the more important metric when an explanatory model is the goal. However, one should be cautious when the two metrics do not agree.

Also, note that the healthy data x in Algorithm 3.5 is a raw time series from the training set described in Section 3.1. Due to time constraints, this time series was

simply chosen as the first one in the training set. With more time, a K-fold *cross-validation* like method could have been employed, where the ARIMA would train on one of the raw time series in the training set, and then the error (e.g. *mean squared error* (MSE)) between the one-step-ahead predictions and the observed values. Then the model which had the lowest total MSE would have been picked.

Algorithm 3.6 ARIMA method

Input: set of window pairs x, significance level α
Output: predictions $\hat{y}$

1: $\hat{y} \leftarrow \emptyset$
2: **for** each window pair $\left(x^{(1)}, x^{(2)}\right)$ in x **do**
3: current prediction $\hat{y}_w \leftarrow 0$
4: **for** each sensor $i = 1, \ldots, n_s$ **do**
5: Start from the last points in $x^{(1)}[i][1 : n_w]$ compute the upper bounds $x^{(\text{upper})}$ and lower bounds $x^{(\text{lower})}$ at significance level α for a one-step-ahead prediction for each value in $x^{(2)}[i][1 : n_w]$.
6: **if** any observed value is outside of the prediction interval in Equation (3.6) **then**
7: $\hat{y}_w \leftarrow 1$
8: add $\hat{y}_w$ to $\hat{y}$
9: **return** $\hat{y}$

The significance level used in Algorithm 3.6 was $\alpha = 0.001$.

Note that Algorithm 3.6 can be extended to the multiclass classification case in the same way as Algorithms 3.3 and 3.4.

3.2.5 Extra trees method

The data was preprocessed such that each time series would have zero mean and unit variance with [65]. All of the parameters for the extra trees classifier were selected according to those presented in [57], except for `max_depth` and `n_jobs`. The parameter values can be seen in Table 3.3.

Table 3.3: Parameters used for the extra trees method in [53].

Parameter	Value
`bootstrap`	`false`
`n_estimators`	30
`criterion`	`gini`
`max_features`	`none`
`max_depth` [4]	4
`min_samples_split`	2
`n_jobs`	-1

[4]The value used for this parameter in [57] was `none`. However, the resulting model showed problems with generalization. So this parameter was set to 4 to avoid over-fitting.

3.2.6 KNN-DTW method

The data was first preprocessed the same way as described in 3.2.5. After which a one nearest neighbor classifier was constructed with the DTW distance.

The fact that the training set $\mathcal{T}$ is quite large poses a problem for kNN, since it performs classifications in $\mathcal{O}(|\mathcal{T}|)$ time complexity (for a new data point x, it needs to compute the distance between it and every other point $x' \in \mathcal{T}$ to compute S_x in Equation (2.23)). To make the classification step faster the training set was downsampled to just 200 samples, where the sampling was performed without replacement. This was because it took less time to perform the computations while hopefully still maintaining the original distribution of $\mathcal{T}$. Note that common data structures like k-d trees [6] and ball trees for improving this time complexity are not trivially applicable.

The parameters used for the kNN-DTW method can be seen in Table 3.4

Table 3.4: Parameters used for the kNN-DTW method in [65].

Parameter	Value
`n_neigbors`	1
`metric`	`dtw`
`n_jobs`	-1

3.2.7 LSTM method

This method allows for a lot of flexibility. The architecture decided on can be seen in Figure 3.3. Implementation was done in [32, 53, 51]. The first part is a normalization layer since the input data is not normalized. The layer used was an *instance normalization layer* [67] to normalize the input data. The input for sensor $i = 1, \dots, n_s$ is normalized according to

$$x^{(i)}_{\text{norm}} = \beta_i + \gamma_i \frac{x^{(i)} - \hat{\mu}^{(i)}}{\sqrt{\left(\hat{\sigma}^{(i)}_{\text{bias}}\right)^2 + \epsilon}}, \tag{3.7}$$

where β_i and γ_i are learnable parameters, $\hat{\mu}^{(i)}$ is computed as in Equation (3.1) and

$$\hat{\sigma}^{(i)}_{\text{bias}} = \sqrt{\frac{n_w - 1}{n_w}}\hat{\sigma}^{(i)}, \tag{3.8}$$

where $\hat{\sigma}^{(i)}$ is computed as in Equation (3.2), i.e. a biased estimator is used in Equation (3.7) of the standard deviation. A constant $\epsilon = 10^{-5}$ is also added to avoid division by zero. The weights are initialized as $\gamma_i = 1$ and the biases $\beta_i = 0$.

Other normalization layers such as *batch normalization* [37] and *layer normalization* [2] were considered. For batch normalization, since it normalizes all input sequences in a mini-batch together, this type of normalization might not be desirable when working with online classification. The reason for not using layer normalization

is that it estimates the average and standard deviation from the last part of the time series and normalizes the entire input using these estimations. This introduces another hyperparameter: how much of the time series should be used to compute these values.

The LSTM layer used in this method is *Bidirectional LSTM* (BLSTM) [30], which consists of two LSTMs, one processing the sequence in the forward direction and another processing it in the backward direction. Their outputs are then concatenated before the linear layer. Note that this can still be used for online detection since we are working with limited window sizes n_w. The reason for using this over a normal unidirectional LSTM is that it showed better results. The dimension of each hidden state and cell state was set to be the same as the input data, i.e., $\dim x(t) = \dim h(t) = \dim c(t) = n_s$. Both c_0 and h_0 are set to 0.

For the LSTM, the weights W_{ij} are initialized according to the first scheme presented in [26], so

$$W_{ij} \sim \mathrm{IID}\mathcal{U}\left(-\frac{1}{\sqrt{n_h}}, \frac{1}{\sqrt{n_h}}\right), \tag{3.9}$$

and the same holds for the biases according to [51]. The linear layer was initialized analogously.

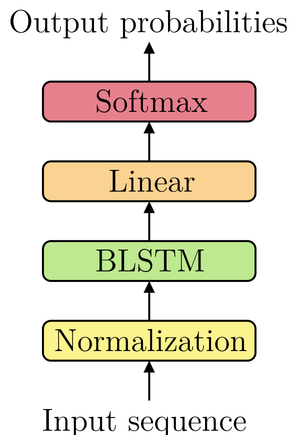


Figure 3.3: A graphical overview of the architecture used in this method. By normalization, an instance normalization layer is intended. Note that for the binary classification task, softmax is meant to mean sigmoid, as it can be seen as a special case of softmax (see Section 2.4.5).

For optimization, the *Adam optimizer* [39] was used. Other optimizers such as *stochastic gradient descent* (SGD) [8], *Nadam* [18] and *RMSProp* [66] were considered but did not yield any significant improvement over Adam. During training the learning rate of 0.002 and the parameters β_1, β_2 and ϵ are set according to [39].

The network was trained for 20 epochs with a batch size of 2. For each epoch, the accuracy on the validation set was computed and the final model was selected as the one that performed best on the validation set over all epochs. The validation set was constructed by performing a 70-30 train test split on the training data.

3.2.8 Elman RNN & GRU method

Methods using an Elman RNN and a GRU were also implemented. These two methods are the same as the LSTM method in Section 3.2.7 except for the BLSTM layer in Figure 3.3, which uses a *Bidirectional Elman RNN* (BERNN) [60] and *Bidirectional GRU* (BGRU) instead of a BLSTM respectively.

3.2.8.1 Number of parameters

An interesting aspect of these models is the number of parameters they require. The first normalization layer in Figure 3.3 has $2n_s$ trainable parameters. The next layer depends on which type of architecture we decide to go with, the number of parameters for each architecture is shown in Table 3.5.

Table 3.5: Number of trainable parameters for the BERNN, BLSTM, BGRU. Note that these expressions can be simplified further by using that $n_h = n_s$ as mentioned in Section 3.2.7. However, this simplification has not been made should the reader wish to change the value for n_h. Note that to get the number of parameters for the unidirectional variants, we simply halve the number of parameters for the bidirectional variant.

Architecture	No. parameters
BERNN	$2n_h (n_s + n_h + 1)$
BLSTM	$8n_h (n_s + n_h + 1)$
BGRU	$6n_h (n_s + n_h + 1)$

The number of parameters for the linear layer in Figure 3.3 depends on whether we are performing binary classification or multiclass classification. If we are performing binary classification, the linear layer will have $2n_h + 1$ trainable parameters, while for a multiclass classification task with $K > 2$ classes, the linear layer will have $K (2n_h + 1)$ parameters.

3.3 Number of samples analysis

Another interesting metric is the number of samples a given algorithm would need to detect a fault. To perform this analysis windows of size $3n_w$ were generated, and then by letting a window pair consisting of windows with size n_w slide over the bigger window the probability of detecting a fault could be estimated as the frequency of faults detected for each sample. Note that for this test the faults always start from time $\tau = 2n_w + 1$ in the bigger window. The method for performing this analysis can be seen in Algorithm 3.7.

The reason for using a sliding window pair is that, for some algorithms like the baseline method (see Section 3.2.2), a window pair is required. Since the last third of the bigger window only contains faulty data, the algorithm will gradually be exposed to more fault samples.

Algorithm 3.7 Determine the number of samples

Input: set of window triplets x, algorithm $\mathcal{A}$
Output: detection probabilities p

1: $\hat{y}^{(\text{sum})} \leftarrow \mathbf{0}^{n_w}$
2: $y^{(\text{sum})} \leftarrow \mathbf{0}^{n_w}$
3: **for** each window triplet $x^{(3)}$ in x **do**
4: $\quad \hat{y} \leftarrow \mathbf{0}^{n_w}$
5: $\quad$ **for** $i = 2, \ldots, n_w + 1$ **do**
6: $\quad\quad x^{(1)} \leftarrow x^{(3)}\,[1 : n_s]\,[i : i + n_w]$
7: $\quad\quad x^{(2)} \leftarrow x^{(3)}\,[1 : n_s]\,[i + n_w + 1 : i + 2n_w]$
8: $\quad\quad \hat{y}\,[i - 1] \leftarrow$ classification by algorithm $\mathcal{A}$ on the window pair $\left(x^{(1)}, x^{(2)}\right)$
9: $\quad \hat{y}^{(\text{sum})} \leftarrow \hat{y}^{(\text{sum})} + \hat{y}$
10: $\quad y^{(\text{sum})} \leftarrow y^{(\text{sum})} + \mathbf{1}^{n_w}$
11: $p \leftarrow \hat{y}^{(\text{sum})} \oslash y^{(\text{sum})}$
12: **return** p

Here $\oslash$ denotes the Hadamard division (i.e. elementwise division). Note that the reason for starting to iterate from $i = 2$ on line 5, is that the algorithm would not be exposed to any fault data when $i = 1$.

Note that Algorithm 3.7 does not compute the probability that algorithm $\mathcal{A}$ detects a fault of type f, but rather the conditional probability that it detects a fault of type f given that a fault of type f has occurred. So this metric is comparable to the recall (see Equation (2.44)).

Algorithm 3.7 can now be used to determine how many samples of a fault are needed to be observed before a detection is made. Given a probability, it can be used as a threshold to determine how many samples are needed to detect a given type of fault with that probability.

4

Result

In this chapter, the results are presented. Results for the binary classification tasks will be presented in Section 4.1 while results from the multiclass classification task are presented in Section 4.2.

In the figures in this section, ERNN is meant to refer to Elman RNNs.

4.1 Result for binary classification

In this section, some initial results from the binary classification can be seen in Figures 4.1 and 4.2. An analysis based on the number of samples required to detect a fault as mentioned in Section 3.3 is also presented in Section 4.1.1. Note that the full results can be seen in Section D.1

From Figure 4.1, a window size of $n_w = 12$ seems to be a sweet spot for the simpler methods (baseline and bootstrap). For the bootstrap classifier, its accuracy for detecting hardover and spike faults in Figures 4.1a and 4.1b decreases with larger window sizes, while its accuracy for detecting stuck and frozen faults in Figures 4.1e and 4.1f decreases with smaller window sizes. We see similar behavior with the accuracy for detecting a frozen fault for the baseline method.

From Figures 4.1b and 4.2b it seems as if group-based analysis is required to detect a drift fault. This is because drift faults are hard to detect since we cannot tell the difference between a drift fault in a sensor and a sensor output measuring values that increase or decrease linearly without a frame of reference.

Note that the accuracy for detecting a hardover, spike or stuck fault for the ARIMA method in Figures 4.2a, 4.2c and 4.2e is very high. However, this is only the case for small window sizes ($n_w \leqslant 4$).

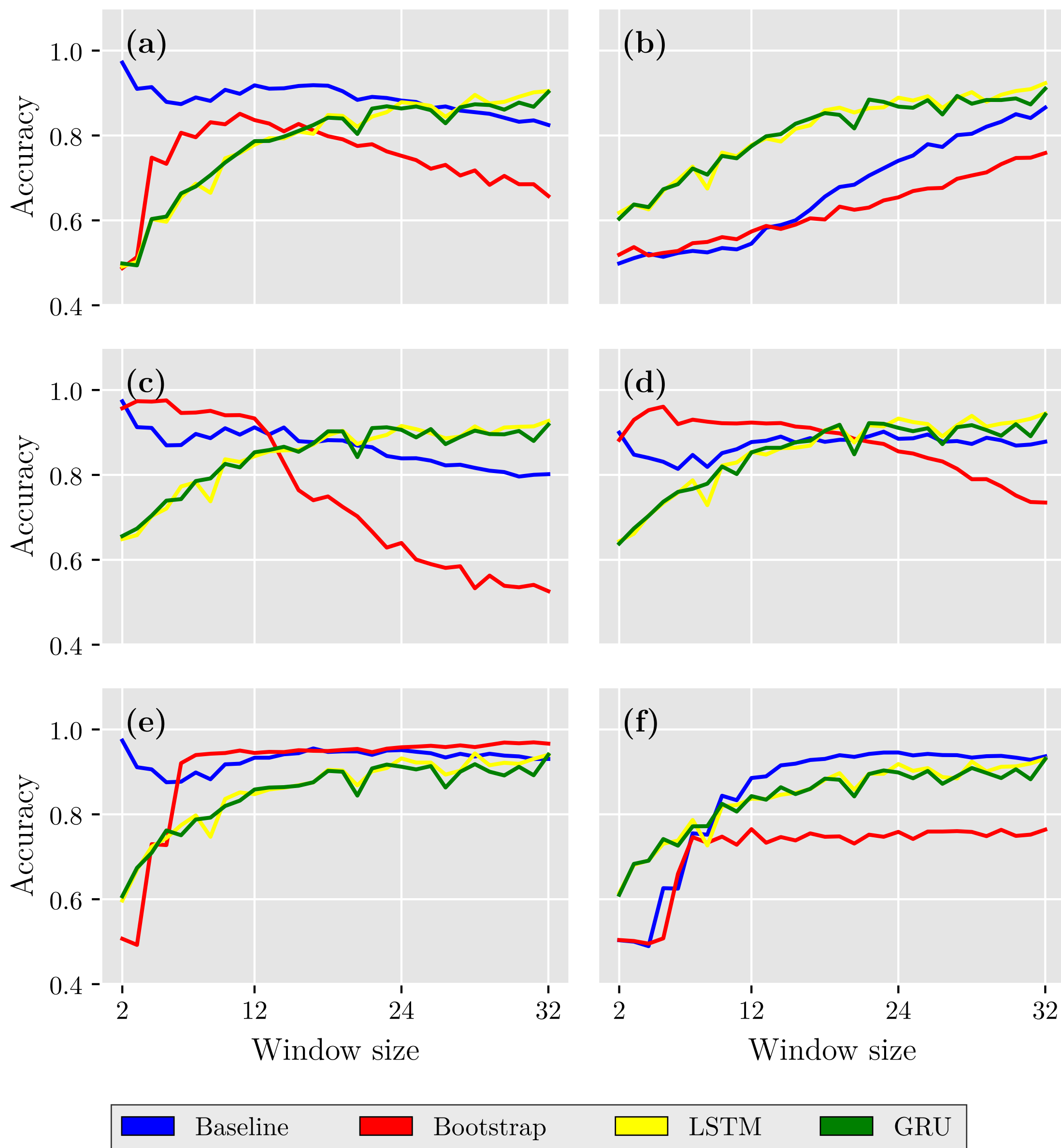


Figure 4.1: Accuracy for the baseline, bootstrap, LSTM and GRU classifiers. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

4.1.1 Number of samples analysis

In this section the results from the number of samples analysis described in Section 3.3 are presented. The algorithms were trained with a window size of $n_w = 12$ and the probability of detecting a fault given a number of samples can be seen in Figures 4.3 and 4.4. The reason for choosing a window size of $n_w = 12$ is because of the observed performance of the baseline and bootstrap method in Figure 4.1.

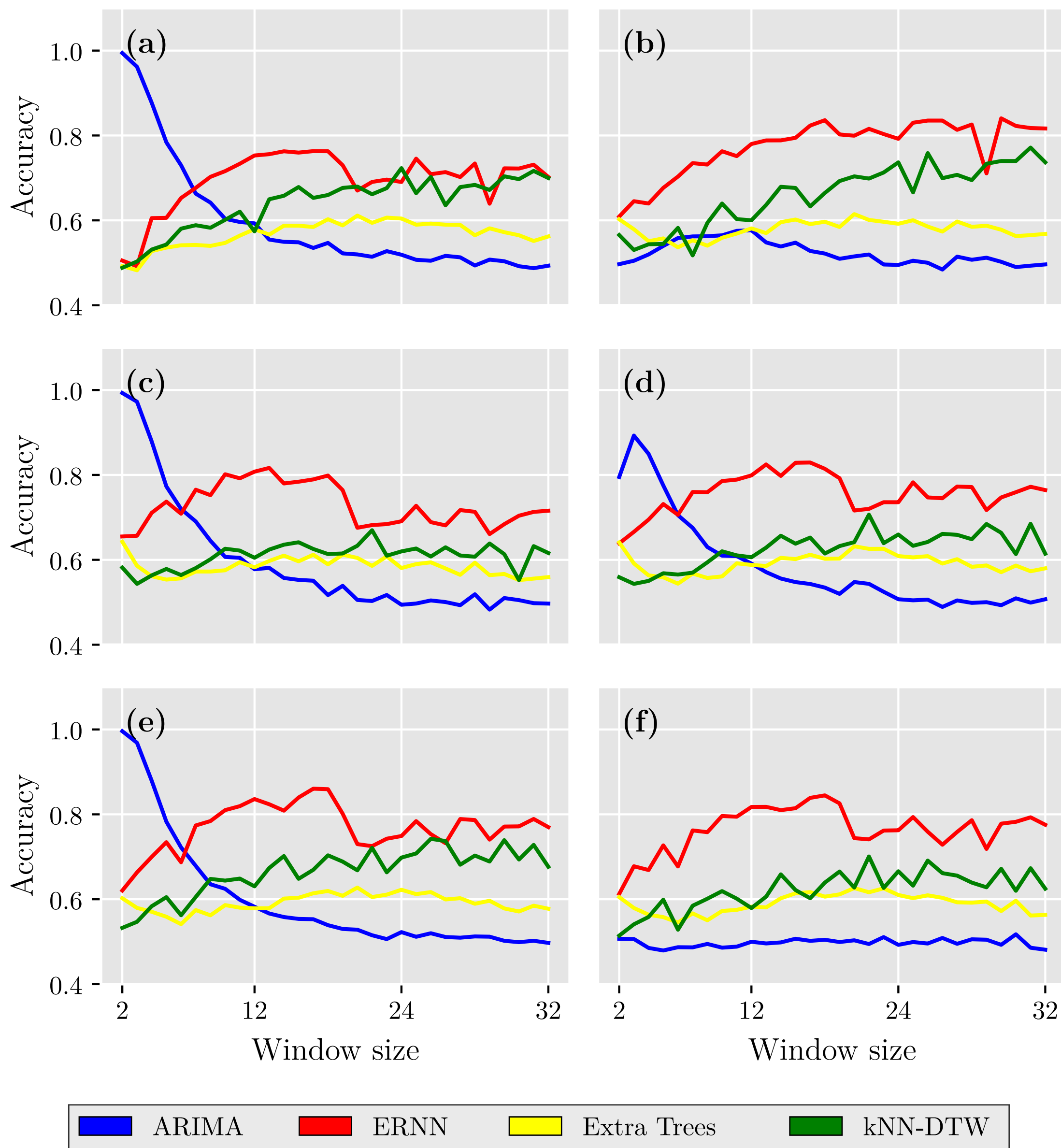


Figure 4.2: Accuracy for the ARIMA, Elman RNN, extra trees and kNN-DTW classifiers. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

The fact that very few algorithms can detect a hardover fault in Figure 4.3a when it sees 12 faulty samples can be explained by the fact that the algorithms operate with a window size of 12. So when the twelfth sample enters the window the initial increase of the hardover fault in Figure 2.1a is no longer visible in the window. The reason that the baseline method (see Algorithm 3.3) performs well is that it uses information from the previous window.

Note that the fact that both the extra trees and ARIMA classifiers were able to detect a fault with a very high probability in Figures 4.3 and 4.4 should not be

interpreted as these classifiers being good at detecting faults. As can be seen in Figure 4.2 both of their accuracies ≈ 0.6 for a window size of 12. This is because these classifiers classify most data as being faulty.

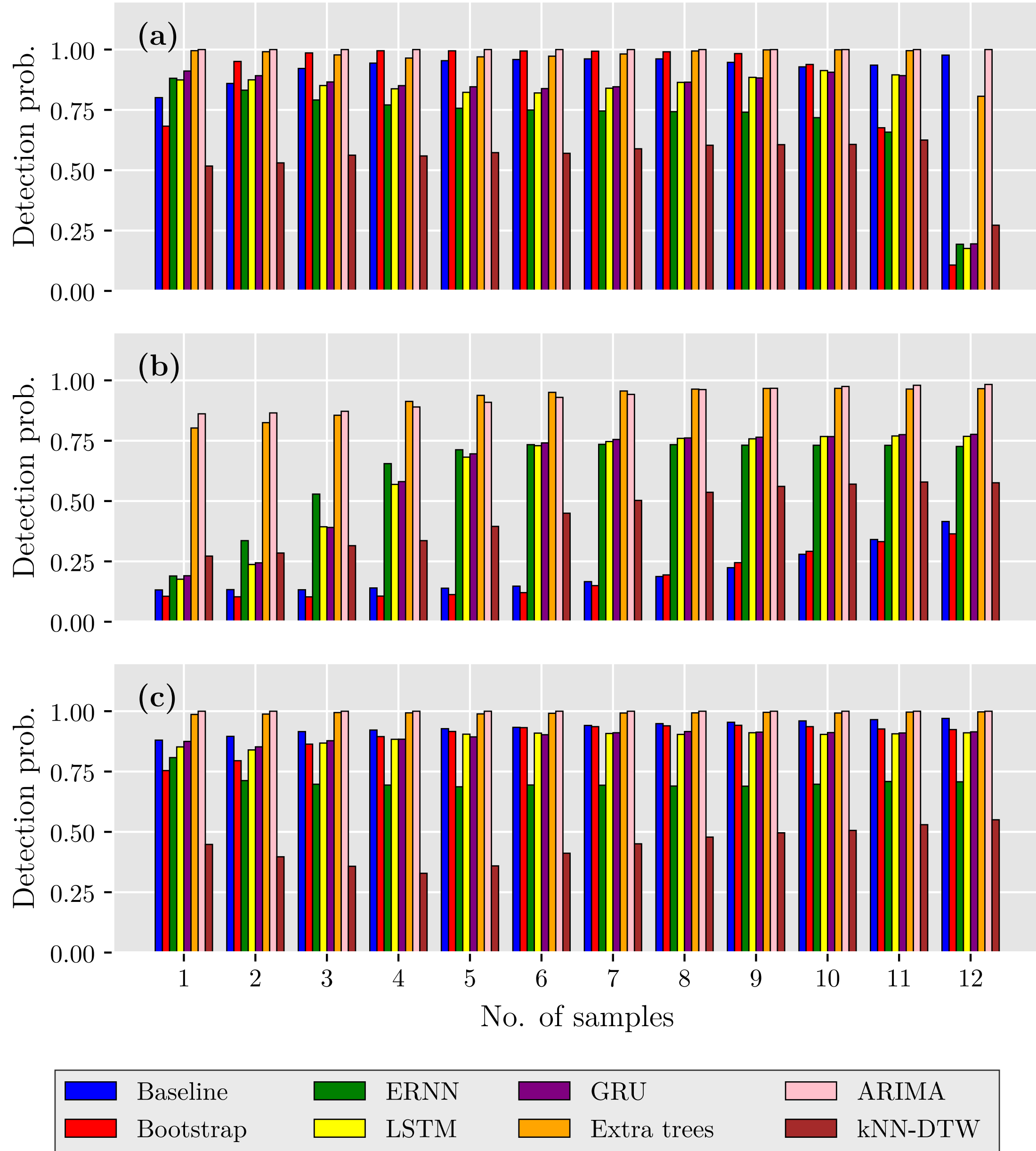


Figure 4.3: Probability of making a detection based on the number of samples with faulty data. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The three types of faults are **a)** hardover, **b)** drift and **c)** spike. Figure created with [34].

The fact that the baseline method sees a significant increase in detection probability around 6 samples in Figure 4.4c is because of the predefined frozen fault detector in Algorithm 3.3.

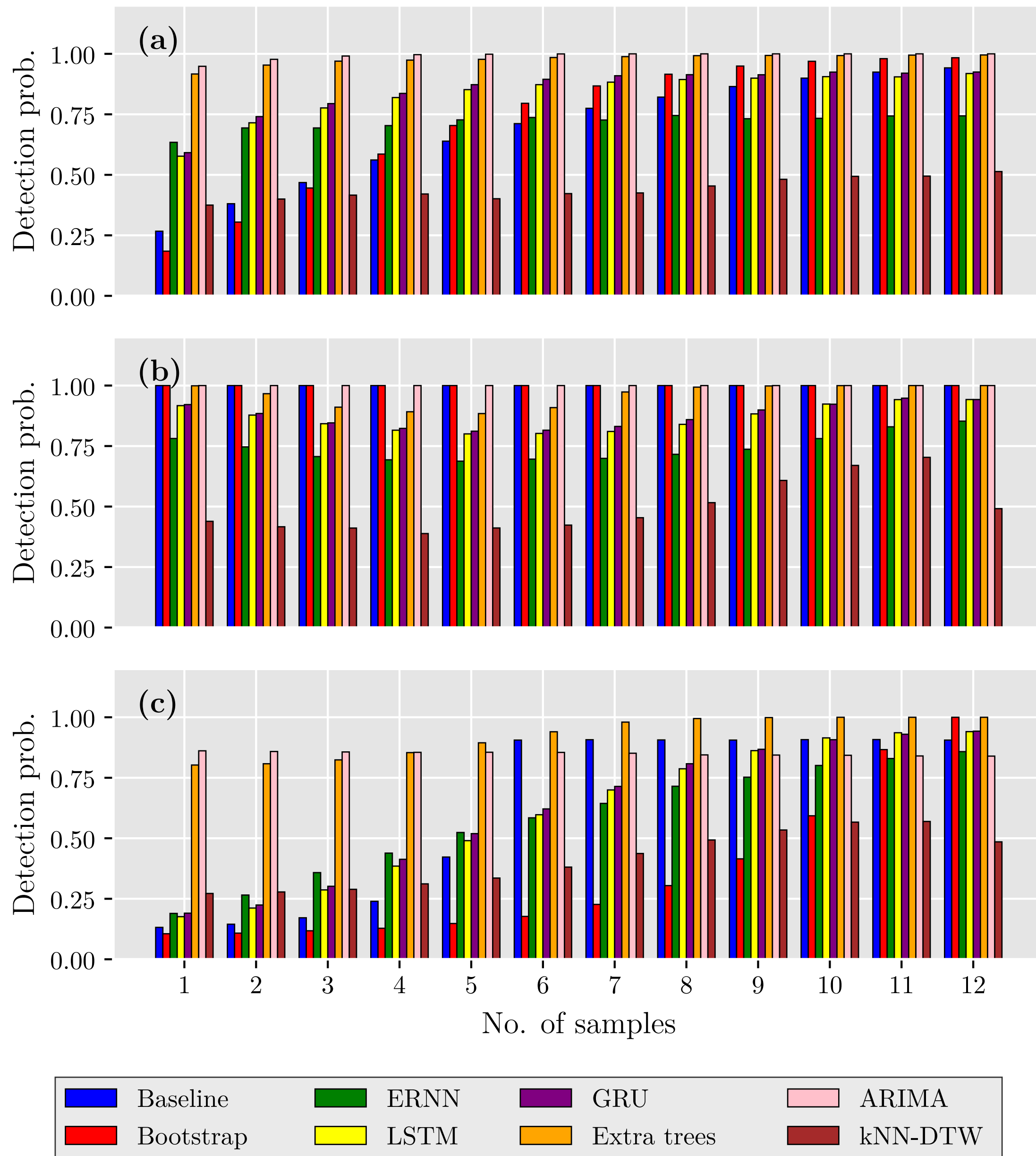


Figure 4.4: Probability of making a detection based on the number of samples with faulty data. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The three types of faults are **a)** erratic, **b)** stuck and **c)** frozen. Figure created with [34].

4.2 Result for multiclass classification

In this section results for the multiclass classification task are presented. In Section 4.2.1 results from determining in which sensor a fault has occurred in are presented while in Section 4.2.2 results from determining what type of fault has occurred are presented.

4.2.1 Results for sensor class classification

The results presented in this section are all based on a balanced test set with nine classes. There is one class for healthy data and one class corresponding to a fault in sensor i for $i = 1, \ldots, n_s$. There are eight different sensors.

In Figure 4.5 the accuracy of the multiclass classification can be seen for different window sizes. Note that many characteristics are shared with the results in Figure 4.1, but that the curves are "smoother" since this test set is larger.

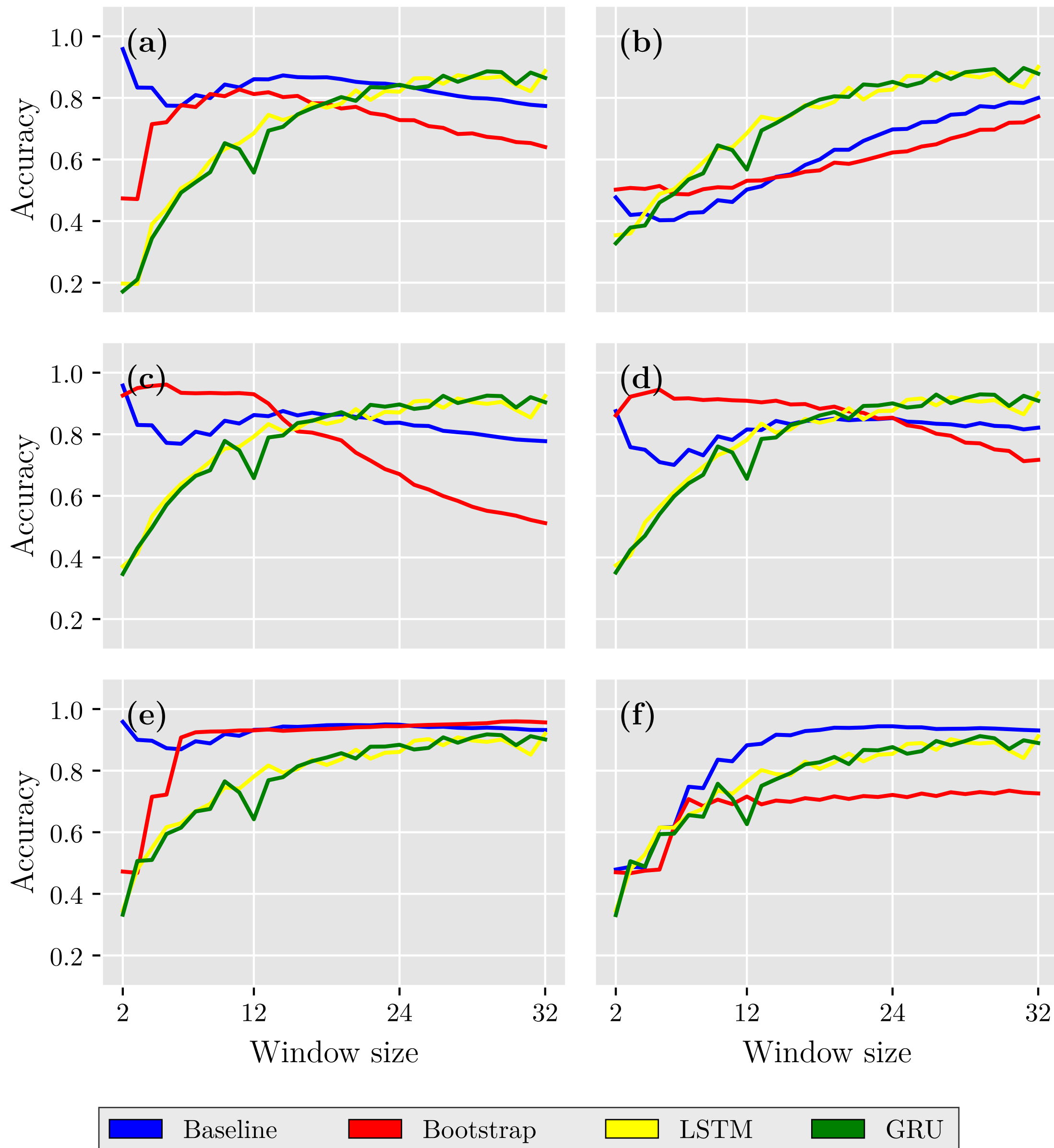


Figure 4.5: Accuracy for the baseline, bootstrap, LSTM and GRU classifiers. Note that the test set is balanced and only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

Note that in contrast to Figure 4.1 this problem consists of nine classes, so a dummy classifier would have an accuracy of approximately 0.11 on a balanced test set, compared to 0.50, hence the different scaling and lower accuracies for some window sizes.

Another way to visualize the performance of the different methods is the confusion matrices. For this a window size has to be chosen. From similar arguments to those presented in Section 4.1.1 a window size of $n_w = 12$ was chosen.

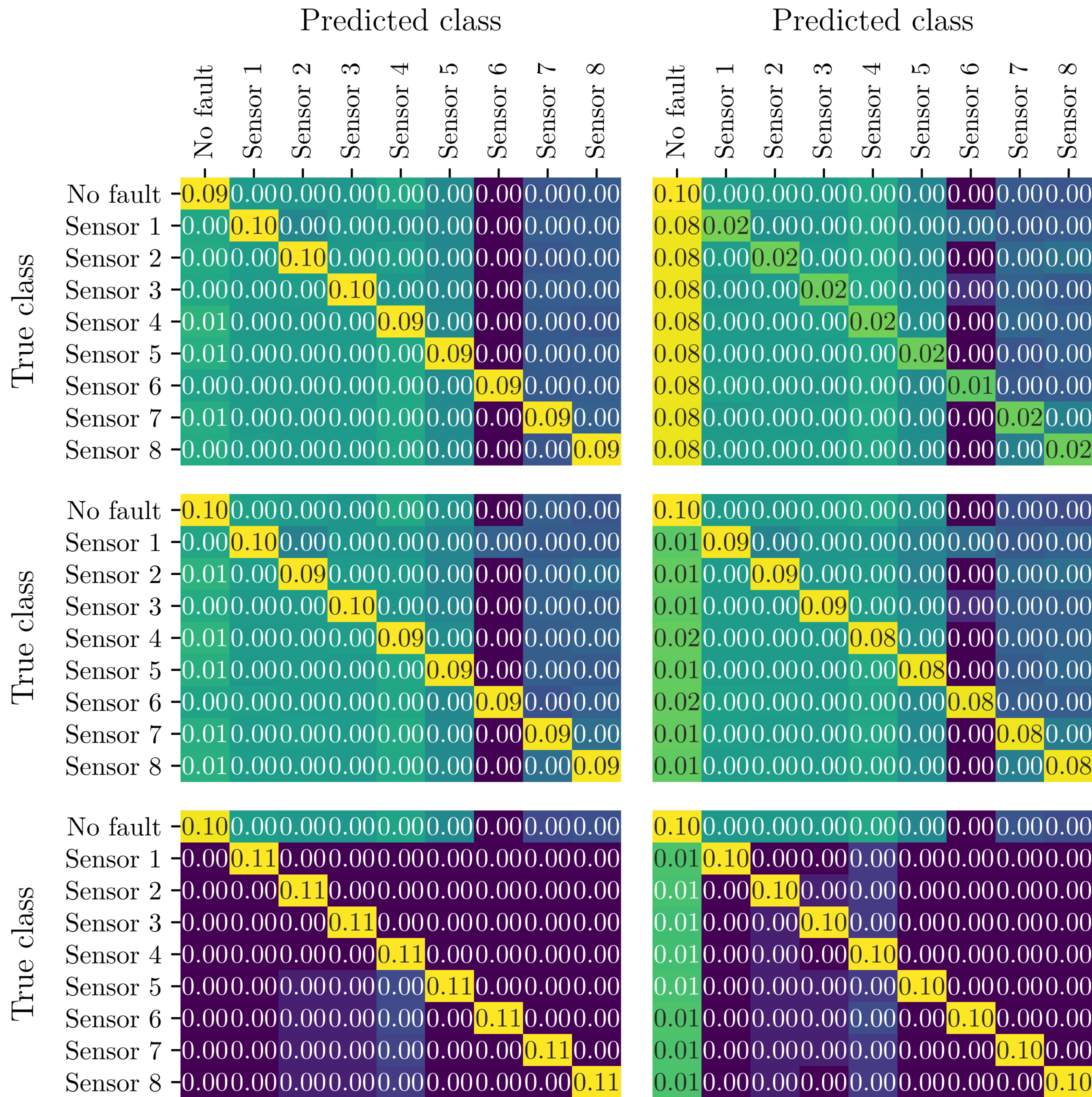


Figure 4.6: Confusion matrices for the baseline classifier from Section 3.2.2 with a window size of 12 on a balanced test set. In each matrix, the percentage of 26,200 classifications is shown. Figure created with [34].

Note that the gradient-like pattern observed among the misclassifications in Figures 4.6 and 4.7 can be explained by the fact that in the case of multiple sensors being classified as faulty, the methods will simply pick the sensor with the lower number.

However, this does not explain the lack of misclassifications for the sixth sensor in both figures.

Note that this pattern is not present in Figure 4.8 and 4.9 since these ANN-based methods make classifications differently.

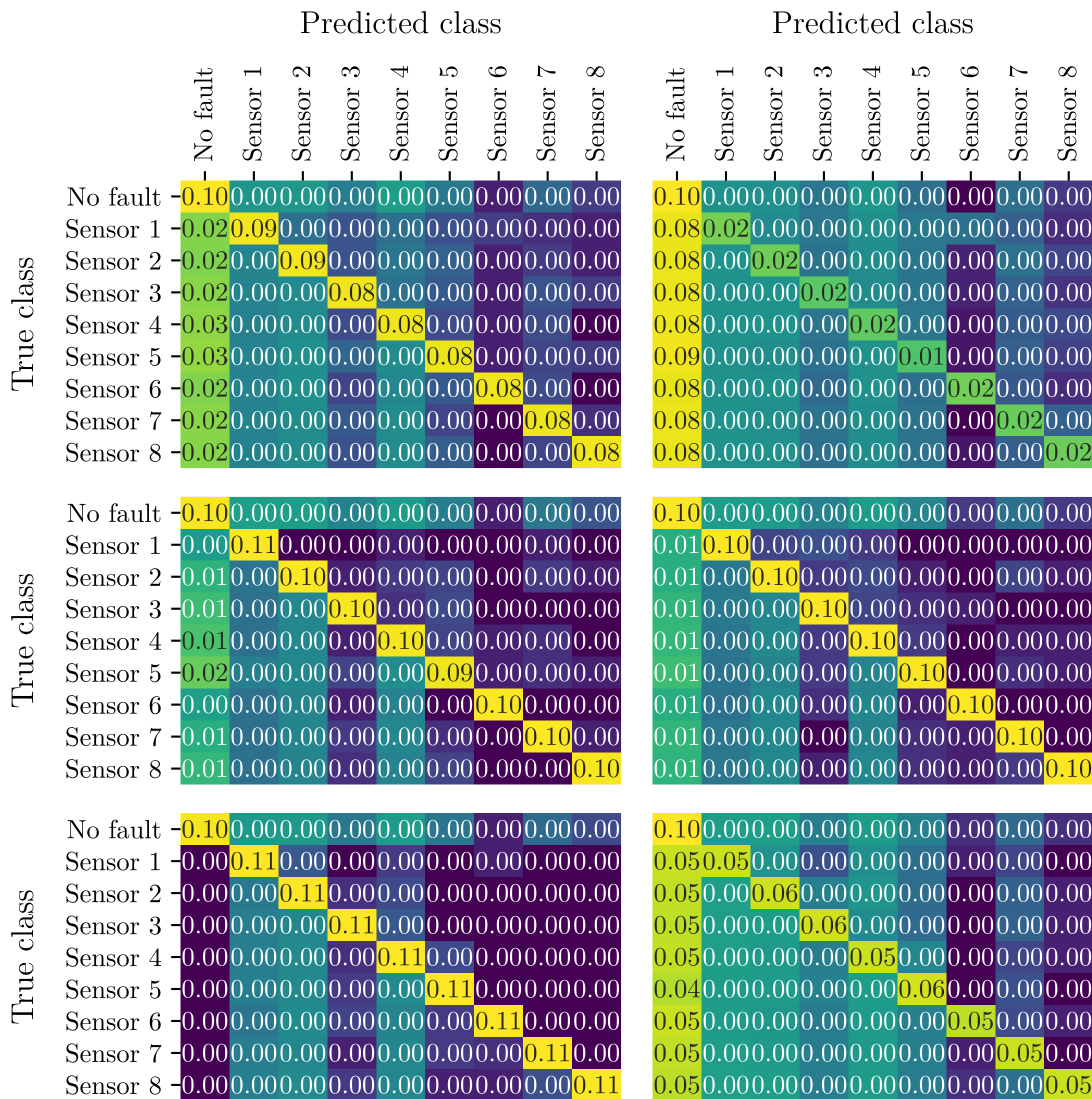


Figure 4.7: Confusion matrices for the bootstrap classifier from Section 3.2.3 with a window size of 12 on a balanced test set. In each matrix, the percentage of 26,200 classifications is shown. Figure created with [34].

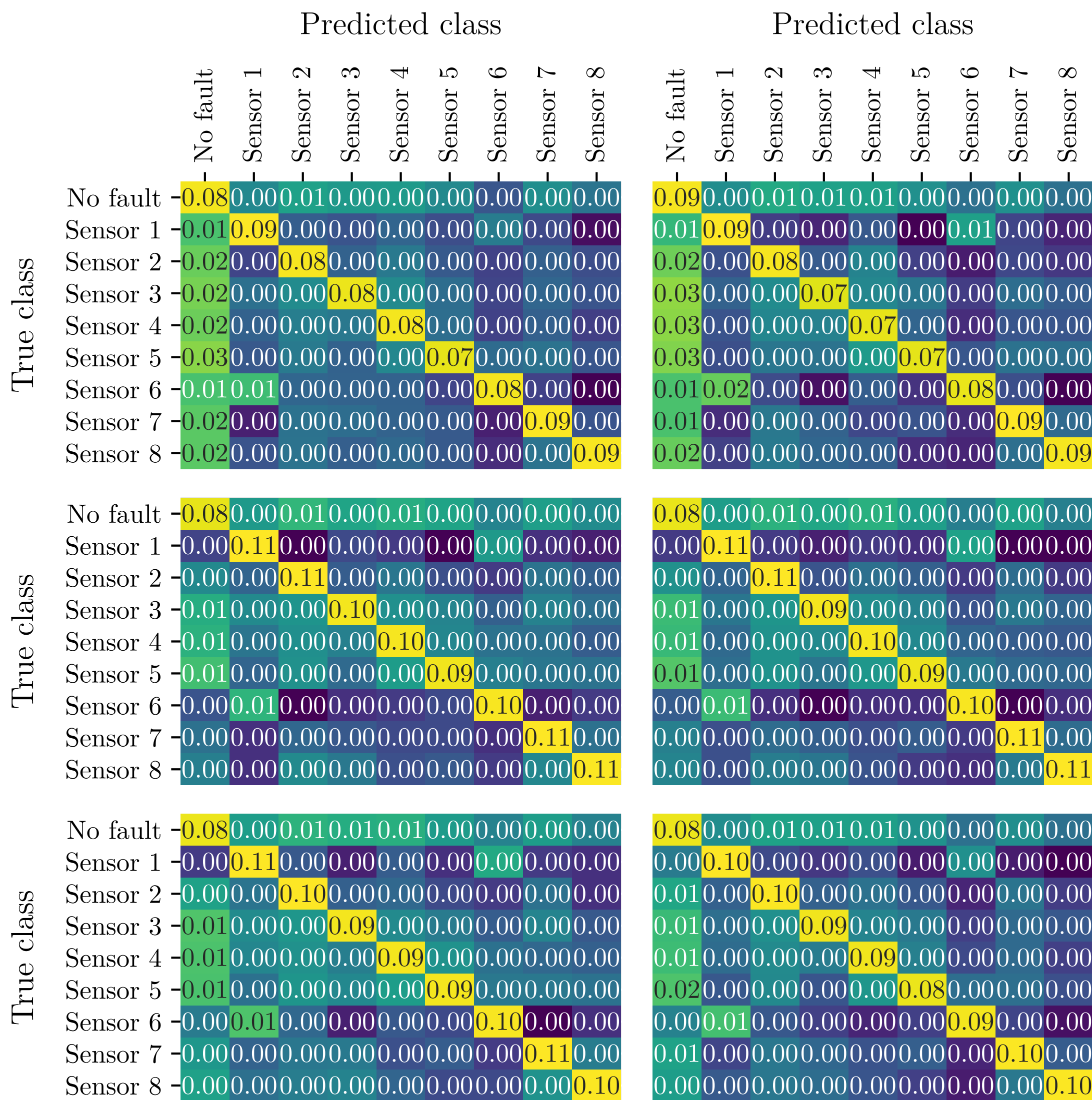


Figure 4.8: Confusion matrices for the LSTM classifier from Section 3.2.7 with a window size of 12 on a balanced test set. In each matrix, the percentage of 26,200 classifications is shown. Figure created with [34].

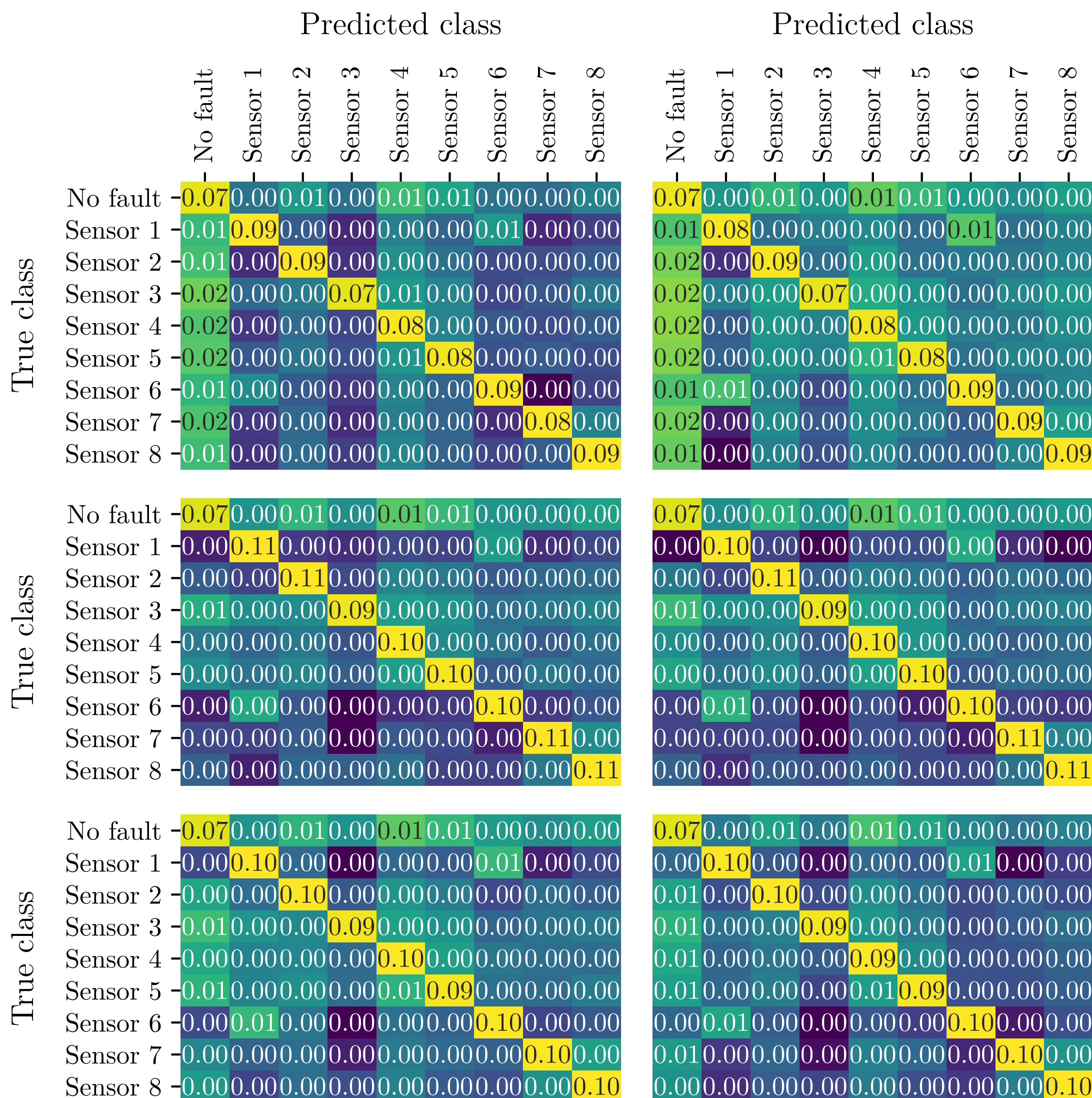


Figure 4.9: Confusion matrices for the GRU classifier from Section 3.2.8 with a window size of 12 on a balanced test set. In each matrix, the percentage of 26,200 classifications is shown. Figure created with [34].

4.2.2 Results for fault class classification

The results presented in this section are all based on a balanced test set with seven classes. There is one class for healthy data and one class corresponding to each of the faults presented in Section 2.2.

From Figure 4.10 we see that all of the metrics increases very slowly with respect to the window size for both the LSTM and GRU methods. So this task seems to be much harder compared with the two previous classification tasks. To get better results for this task, other methods might have to be considered.

One should also note that this problem consists of seven classes, so a dummy classifier would have an accuracy of approximately 0.14 for the balanced test set in Figure

4.10.

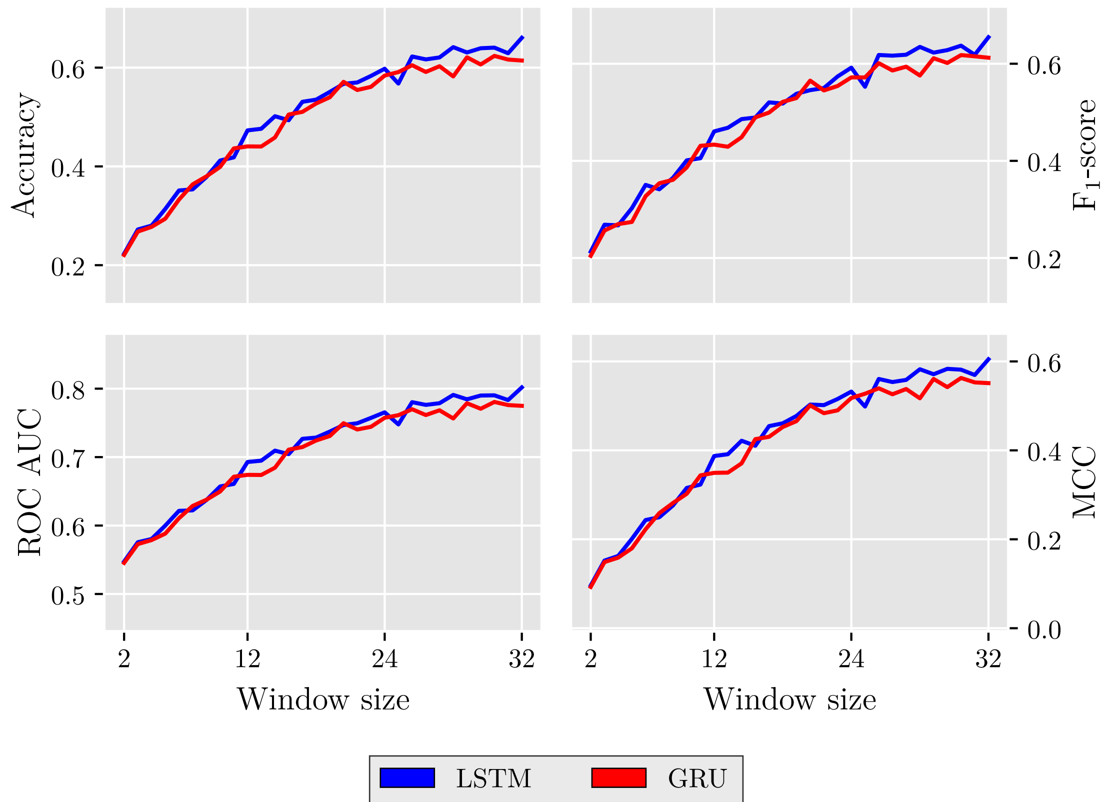


Figure 4.10: Accuracy for the LSTM and GRU classifiers. Note that the test set is balanced, so it consists of an equal amount of each type of fault and no fault. Figure created with [34].

In Figures 4.11 and 4.12, the confusion matrices for the LSTM and GRU methods can be seen. In these figures, a window size of $n_w = 12$ was chosen to be consistent with previous sections.

Predicted class

True class	No fault	Hardover	Drift	Spike	Erratic	Stuck	Frozen
No fault	0.10	0.01	0.01	0.01	0.01	0.00	0.00
Hardover	0.03	0.07	0.01	0.01	0.01	0.00	0.00
Drift	0.03	0.01	0.09	0.00	0.01	0.00	0.00
Spike	0.01	0.02	0.01	0.06	0.03	0.01	0.01
Erratic	0.01	0.01	0.01	0.05	0.05	0.01	0.00
Stuck	0.01	0.00	0.00	0.02	0.01	0.08	0.02
Frozen	0.01	0.02	0.01	0.03	0.01	0.03	0.03

Figure 4.11: Confusion matrix for the LSTM classifier from Section 3.2.7 with a window size of 12 on a balanced test set. In each matrix, the percentage of 19,650 classifications is shown. Figure created with [34].

In Figure 4.11 we see that the LSTM method has a hard time distinguishing between a spike and an erratic fault.

Predicted class

True class	No fault	Hardover	Drift	Spike	Erratic	Stuck	Frozen
No fault	0.10	0.01	0.00	0.00	0.01	0.01	0.01
Hardover	0.03	0.06	0.01	0.01	0.01	0.01	0.02
Drift	0.03	0.02	0.07	0.00	0.01	0.00	0.01
Spike	0.01	0.01	0.01	0.03	0.04	0.03	0.02
Erratic	0.01	0.01	0.01	0.02	0.07	0.02	0.01
Stuck	0.01	0.00	0.00	0.01	0.02	0.08	0.02
Frozen	0.01	0.02	0.01	0.01	0.02	0.04	0.04

Figure 4.12: Confusion matrix for the GRU classifier from Section 3.2.8 with a window size of 12 on a balanced test set. In each matrix, the percentage of 19,650 classifications is shown. Figure created with [34].

5

Discussion

In this chapter, some noteworthy aspects of the thesis and considerations are discussed.

5.1 The dataset

As mentioned in Section 3.1, the data set in this thesis consisted of aggregated data. This may affect the distribution of the data seen in this project, as it was seen that it was close to a Student-t_ν or normal distribution. Although the original data points cannot be assumed to be independent, so a rudimentary version of the *central limit theorem* (CLT), i.e. the Lindeberg-Lévy theorem, cannot be applied to explain this observation. However, it should still be verified that the distribution seen in data aggregated from fewer data points follows the same distribution.

5.2 The prevalence of faults

One should also note that this thesis has mainly considered balanced data sets, where observations of each class are equally prevalent. However, outputs from radio sensors rarely contain faults in practice, and the current methods would probably yield some false positives. Some limited results on unbalanced data are available in Section D.1. Another assumption that has been made in this thesis is that all faults are equally likely to occur. This is not true in practice, so for more accurate results the distribution of the different faults would have to be estimated.

5.3 The independence test

One should also note that in the test performed in Section 3.2.3 we did not show that the variance between two adjacent windows was independent, but only that independence cannot be ruled out with 95 % certainty. One should also note that this is generally not how hypothesis testing is used in practice, where we want to use the data to discard a null hypothesis in favor of some alternate hypothesis.

5.4 Note on the ANN-based methods

From the good performance of the LSTM and GRU in Sections 4.1 and 4.2.1 and their performance in Section 4.2.2, one can see that there is likely no need to employ more computationally heavy state of the art methods such as transformers.

One should also note that the ANN-based methods used in this thesis all have been RNN-based. However, since we are mainly working with short sequences because we are using a sliding window, other ANN-based methods could be considered. One method could be based on a *multilayer perceptron* (MLP) since we can flatten the input vector to a $n_w n_s$-dimensional vector to fit the structure. Another method could be based on a *convolutional neural network* (CNN) where the input could be viewed as an image with $n_s \times n_w$ pixels and a single channel. However, one could make the argument that a sliding window is a special case of a sliding kernel. So the algorithms already incorporate some of the ideas behind CNNs.

Another interesting method to explore would be to use a recently popularized *Kolmogorov-Arnold network* (KAN) [44] instead of an MLP since these have shown comparable performance while requiring fewer parameters. However, they are harder to train and given the time constraints that this project is subdued to, the training of such a network might not be feasible.

5.5 Uncertainty in the results

It should be noted that the results presented in Chapter 4 are computed from a single run. Due to time constraints, multiple runs were not able to be performed. Otherwise, uncertainty could have been taken into account in both training and testing. However, since the curves in the figures presented in Chapter 4 are somewhat smooth, one could expect that these measurements will not show much variance across different runs.

5.6 Note regarding the kNN-DTW method

It is hard to draw any conclusions about the performance of the kNN-DTW method given the results presented in Section 4.1. As mentioned in Section 3.2.6 the original training set $\mathcal{T}$ had to be sampled down because of running time. It is then hard to conclude how the method would have performed should it use the entire data set.

5.7 Note regarding the ARIMA method

As noted in Section 4.1, the ARIMA method (see Section 3.2.4) performs well when detecting hardover, spike and stuck faults in Figures 4.2a, 4.2c and 4.2e for window size $n_w = 2$. These faults would be considered short faults (a fault that is categorized by a notable difference in the value measured between two consecutive data points) in [63], where a one-step ahead time series method also performed well on short faults.

However, it should be noted that the ARIMA model used in [63] also incorporated a seasonal component.

There are however some concerns regarding the performance of this method. Since the data set in this thesis consists of aggregated data points as mentioned in Section 3.1, this will give a smoother time series as the number of aggregated data points increases. So if the number of data points that are aggregated decreases, the resulting time series would be less smooth. An ARIMA model might perform worse in a less smooth setting.

Another, thing that may be considered is the fact the ARIMA method only needs a single observation outside of the prediction interval to classify a window as containing a fault regardless of the window size. Since it was observed that the ARIMA method classified a majority of windows as being faulty for larger window sizes, a solution might be to let the required number of observations outside of the prediction interval to classify a window as being faulty to grow with the window size.

5.8 Future work

The fact that the baseline method (see Section 3.2.2) performed well for some window sizes, e.g. $n_w = 12$, might make it worth investigating other parameters than those presented in Table 3.2. It might be possible, given some training data, to fit these by either performing a grid search or particle swarm optimization. Another possibility would be to estimate upper and lower quantiles via bootstrap, as in the bootstrap method (see Section 3.2.3). However, this would not come with the same theoretical foundation as the null hypotheses were discarded for the other metrics in Section 3.2.3.

One could also consider the creation of a hybrid method, as different methods are good at detecting different fault types. As previously discussed in Section 5.7, a small ARIMA model showed good results in detecting three of the fault types, which is similar to the findings of [63]. The predefined frozen fault detector in the baseline method showed good performance for detecting the frozen faults.

Another thing which should looked at is how these algorithms perform while the number of sensors n_s increases. As mentioned in Section 3.1.2, n_s can be larger than it is in this thesis and it is crucial for successful deployment that scalability of the methods with respect to n_s is evaluated. Another problem which would have to be addressed is how to handle the case when faults are detected in multiple sensors for the baseline and bootstrap methods. In this thesis, the sensor with the lowest number was picked in case faults were detected in multiple sensors. Another method would be to decide based on which sensor is the “most” faulty, i.e., in the baseline method we pick the sensor which showed the biggest absolute difference between two consecutive windows, or in the bootstrap method we pick the sensor whose variance was furthest away from the closest quantile.

As briefly touched upon in Section 5.7 regarding the aggregation of the data. For a method to be deployed its performance would first have to be verified for data

which is aggregated from fewer data points, since less smooth data might have some impact on the methods.

6 Conclusion

In Sections 4.1 and 4.2.1 we see that the baseline and bootstrap methods are comparable with, or even outperform the more advanced methods (except for the drift fault) for some window sizes while requiring less computing power.

Moreover, the success of the LSTM and GRU methods shows that there is likely no need to further explore how more advanced methods can be applied to this problem, as there is not much to be gained. However, one should note that there is a need for another method when determining the type of fault that has happened. Although this would come at a tradeoff.

Before any method were to be deployed their scalability would have to be verified by evaluating how an increase in the number of sensors affect their performance.

References

[1] Hirotsugu Akaike. "Information theory and an extension of the maximum likelihood principle". In: *2nd international symposium on information theory*. Ed. by Frigyes Csáki and Boris Nikolaevich Petrov. Akademiai Kiado. Budapest, Hungary, 1973, pp. 267–281.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: (2016). DOI: 10.48550/ARXIV.1607.06450.

[3] N. Balakrishnan, Vassilly Voinov, and M.S Nikulin. "Chapter 8 – Probability Distributions of Interest". In: *Chi-Squared Goodness of Fit Tests with Applications*. Ed. by N. Balakrishnan, Vassilly Voinov, and M.S Nikulin. Boston: Academic Press, 2013, pp. 167–196. ISBN: 978-0-12-397194-4.

[4] Richard E Bellman. *Dynamic programming*. en. Princeton, NJ: Princeton University Press, Aug. 1957. ISBN: 069107951X.

[5] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994), pp. 157–166. ISSN: 1941-0093. DOI: 10.1109/72.279181.

[6] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (Sept. 1975), pp. 509–517. ISSN: 1557-7317. DOI: 10.1145/361002.361007.

[7] Kevin Beyer et al. "When Is "Nearest Neighbor" Meaningful?" In: *Database Theory — ICDT'99*. Springer Berlin Heidelberg, 1999, pp. 217–235. ISBN: 9783540492573. DOI: 10.1007/3-540-49257-7_15.

[8] Léon Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proceedings of COMPSTAT'2010*. Physica-Verlag HD, 2010, pp. 177–186. ISBN: 9783790826043. DOI: 10.1007/978-3-7908-2604-3_16.

[9] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/a:1010933404324.

[10] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. 3rd ed. Basel, Switzerland: Springer International Publishing, Aug. 2016, pp. 25, 151, 152, 158, 177. ISBN: 9783319298542. DOI: 10.1007/978-3-319-29854-2.

[11] C. G. Broyden. "The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations". In: *IMA Journal of Applied Mathematics* 6.1 (1970), pp. 76–90. ISSN: 1464-3634. DOI: 10.1093/imamat/6.1.76.

[12] Joseph E. Cavanaugh and Andrew A. Neath. "The Akaike information criterion: Background, derivation, properties, application, interpretation, and re-

finements". In: *WIREs Computational Statistics* 11.3 (Mar. 2019). ISSN: 1939-0068. DOI: 10.1002/wics.1460.

[13] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785.

[14] Davide Chicco and Giuseppe Jurman. "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation". In: *BMC Genomics* 21.1 (Jan. 2020). ISSN: 1471-2164. DOI: 10.1186/s12864-019-6413-7.

[15] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* 2014. DOI: 10.48550/ARXIV.1406.1078.

[16] D. R. Cox. "The Regression Analysis of Binary Sequences". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 20.2 (1958), pp. 215–242. ISSN: 00359246.

[17] Lee R. Dice. "Measures of the Amount of Ecologic Association Between Species". In: *Ecology* 26.3 (Aug. 1945), pp. 297–302. ISSN: 1939-9170. DOI: 10.2307/1932409.

[18] Timothy Dozat. "Incorporating Nesterov Momentum into Adam". In: *ICLR Workshop* 1 (2016).

[19] Ricardo Dunia et al. "Identification of faulty sensors using principal component analysis". In: *AIChE Journal* 42.10 (Oct. 1996), pp. 2797–2812. ISSN: 1547-5905. DOI: 10.1002/aic.690421011.

[20] B. Efron. "Bootstrap Methods: Another Look at the Jackknife". In: *The Annals of Statistics* 7.1 (1979), pp. 1–26. ISSN: 00905364.

[21] Jeffrey L. Elman. "Finding Structure in Time". In: *Cognitive Science* 14.2 (1990), pp. 179–211. DOI: 10.1207/s15516709cog1402_1.

[22] L. Erhan et al. "Smart anomaly detection in sensor systems: A multi-perspective review". In: *Information Fusion* 67 (Mar. 2021), pp. 64–79. ISSN: 1566-2535. DOI: 10.1016/j.inffus.2020.10.001.

[23] Evelyn Fix and J. L. Hodges. "Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties". In: *International Statistical Review / Revue Internationale de Statistique* 57.3 (Dec. 1989), pp. 238–247. ISSN: 0306-7734. DOI: 10.2307/1403797.

[24] R. Fletcher. "A new approach to variable metric algorithms". In: *The Computer Journal* 13.3 (Mar. 1970), pp. 317–322. ISSN: 1460-2067. DOI: 10.1093/comjnl/13.3.317.

[25] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees". In: *Machine Learning* 63.1 (Mar. 2006), pp. 3–42. ISSN: 1573-0565. DOI: 10.1007/s10994-006-6226-1.

[26] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye

Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256.

[27] Donald Goldfarb. “A family of variable-metric methods derived by variational means”. In: *Mathematics of Computation* 24.109 (1970), pp. 23–26. ISSN: 1088-6842. DOI: 10.1090/s0025-5718-1970-0258249-6.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* http://www.deeplearningbook.org. MIT Press, 2016, pp. 180–184, 191–192, 367–378.

[29] J. Gorodkin. “Comparing two K-category assignments by a K-category correlation coefficient”. In: *Computational Biology and Chemistry* 28.5–6 (Dec. 2004), pp. 367–374. ISSN: 1476-9271. DOI: 10.1016/j.compbiolchem.2004.09.006.

[30] Alex Graves and Jürgen Schmidhuber. “Framewise phoneme classification with bidirectional LSTM and other neural network architectures”. In: *Neural Networks* 18.5 – 6 (July 2005), pp. 602–610. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2005.06.042.

[31] David J. Hand and Robert J. Till. In: *Machine Learning* 45.2 (2001), pp. 171–186. ISSN: 0885-6125. DOI: 10.1023/a:1010920819831.

[32] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[33] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 1530-888X. DOI: 10.1162/neco.1997.9.8.1735.

[34] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[35] Clifford M. Hurvich and Chin-Ling Tsai. “Regression and time series model selection in small samples”. In: *Biometrika* 76.2 (1989), pp. 297–307. ISSN: 1464-3510. DOI: 10.1093/biomet/76.2.297.

[36] Rob J. Hyndman and Yanan Fan. “Sample Quantiles in Statistical Packages”. In: *The American Statistician* 50.4 (Nov. 1996), pp. 361–365. ISSN: 1537-2731. DOI: 10.1080/00031305.1996.10473566.

[37] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: (2015). DOI: 10.48550/ARXIV.1502.03167.

[38] Sana Ullah Jan et al. “Sensor Fault Classification Based on Support Vector Machine and Statistical Time-Domain Features”. In: *IEEE Access* 5 (2017), pp. 8682–8690. ISSN: 2169-3536. DOI: 10.1109/access.2017.2705644.

[39] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. DOI: 10.48550/ARXIV.1412.6980.

[40] Jens-Peter Kreiss and Efstathios Paparoditis. “Bootstrap methods for dependent data: A review”. In: *Journal of the Korean Statistical Society* 40.4 (Dec. 2011), pp. 357–378. ISSN: 1226-3192. DOI: 10.1016/j.jkss.2011.08.009.

[41] Jyrki Kullaa. “Detection, identification, and quantification of sensor fault in a sensor network”. In: *Mechanical Systems and Signal Processing* 40.1 (Oct. 2013), pp. 208–221. ISSN: 0888-3270. DOI: 10.1016/j.ymssp.2013.05.007.

[42] Kristian Linnet. “Nonparametric Estimation of Reference Intervals by Simple and Bootstrap-based Procedures”. In: *Clinical Chemistry* 46.6 (June 2000), pp. 867–869. ISSN: 1530-8561. DOI: 10.1093/clinchem/46.6.867.

[43] Dong C. Liu and Jorge Nocedal. “On the limited memory BFGS method for large scale optimization”. In: *Mathematical Programming* 45.1–3 (Aug. 1989), pp. 503–528. ISSN: 1436-4646. DOI: 10.1007/bf01589116.

[44] Ziming Liu et al. *KAN: Kolmogorov-Arnold Networks*. 2024. DOI: 10.48550/ARXIV.2404.19756.

[45] B.W. Matthews. “Comparison of the predicted and observed secondary structure of T4 phage lysozyme”. In: *Biochimica et Biophysica Acta (BBA) - Protein Structure* 405.2 (Oct. 1975), pp. 442–451. ISSN: 0005-2795. DOI: 10.1016/0005-2795(75)90109-9.

[46] Thaha Muhammed and Riaz Ahmed Shaikh. “An analysis of fault detection strategies in wireless sensor networks”. In: *Journal of Network and Computer Applications* 78 (Jan. 2017), pp. 267–287. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2016.10.019.

[47] Andrew A. Neath and Joseph E. Cavanaugh. “The Bayesian information criterion: background, derivation, and applications”. In: *WIREs Computational Statistics* 4.2 (Dec. 2011), pp. 199–203. ISSN: 1939-0068. DOI: 10.1002/wics.199.

[48] Kevin Ni et al. “Sensor network data fault types”. In: *ACM Transactions on Sensor Networks* 5.3 (May 2009), pp. 1–29. ISSN: 1550-4867. DOI: 10.1145/1525856.1525863.

[49] Zainib Noshad et al. “Fault Detection in Wireless Sensor Networks through the Random Forest Classifier”. In: *Sensors* 19.7 (Apr. 2019), p. 1568. ISSN: 1424-8220. DOI: 10.3390/s19071568.

[50] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training Recurrent Neural Networks”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 2013, pp. 1310–1318.

[51] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[52] Karl Pearson. “X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50.302 (July 1900), pp. 157–175. ISSN: 1941-5990. DOI: 10.1080/14786440009463897.

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

[54] David M. W. Powers. “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation”. In: *International Journal*

of Machine Learning Technology 2 (2011), pp. 37–63. DOI: 10.48550/ARXIV.2010.16061.

[55] Alejandro Pasos Ruiz et al. "The great multivariate time series classification bake off: a review and experimental evaluation of recent algorithmic advances". In: *Data Mining and Knowledge Discovery* 35.2 (Dec. 2020), pp. 401–449. ISSN: 1573-756X. DOI: 10.1007/s10618-020-00727-3.

[56] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

[57] Umer Saeed et al. "Fault diagnosis based on extremely randomized trees in wireless sensor networks". In: *Reliability Engineering & System Safety* 205 (Jan. 2021), p. 107284. ISSN: 0951-8320. DOI: 10.1016/j.ress.2020.107284.

[58] Saeid Safavi et al. "Multi-Sensor Fault Detection, Identification, Isolation and Health Forecasting for Autonomous Vehicles". In: *Sensors* 21.7 (Apr. 2021), p. 2547. ISSN: 1424-8220. DOI: 10.3390/s21072547.

[59] H. Sakoe and S. Chiba. "Dynamic programming algorithm optimization for spoken word recognition". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26.1 (Feb. 1978), pp. 43–49. ISSN: 0096-3518. DOI: 10.1109/tassp.1978.1163055.

[60] M. Schuster and K.K. Paliwal. "Bidirectional recurrent neural networks". In: *IEEE Transactions on Signal Processing* 45.11 (1997), pp. 2673–2681. ISSN: 1053-587X. DOI: 10.1109/78.650093.

[61] Gideon Schwarz. "Estimating the Dimension of a Model". In: *The Annals of Statistics* 6.2 (1978), pp. 461–464. ISSN: 00905364.

[62] D. F. Shanno. "Conditioning of quasi-Newton methods for function minimization". In: *Mathematics of Computation* 24.111 (1970), pp. 647–656. ISSN: 1088-6842. DOI: 10.1090/s0025-5718-1970-0274029-x.

[63] Abhishek B. Sharma, Leana Golubchik, and Ramesh Govindan. "Sensor faults: Detection methods and prevalence in real-world datasets". In: *ACM Transactions on Sensor Networks* 6.3 (June 2010), pp. 1–39. ISSN: 1550-4867. DOI: 10.1145/1754414.1754419.

[64] Thorvald Sørensen. "A Method of Establishing Groups of Equal Amplitude in Plant Sociology Based on Similarity of Species Content and Its Application to Analyses of the Vegetation on Danish Commons". In: *Biologiske skrifter*. Vol. 5. 4. Kongelige Danske Videnskabernes Selskab, 1948.

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

[66] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[67] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. "Instance Normalization: The Missing Ingredient for Fast Stylization". In: (2016). DOI: 10.48550/ARXIV.1607.08022.

[68] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762.

[69] Kilian Q. Weinberger and Karthik Sridharan. “Lecture 3: K Nearest Neighbors and the curse of dimensionality”. In: *CS 4/5780: Intro to Machine Learning* (Jan. 2023). URL: https://www.cs.cornell.edu/courses/cs4780/2024sp/lectures/pdfs/lecturenote02_kNN.pdf.

[70] Qingsong Wen et al. “Transformers in Time Series: A Survey”. In: *32nd International Joint Conference on Artificial Intelligence* (2022). DOI: 10.48550/ARXIV.2202.07125.

[71] Jing-li Yang et al. “Fault detection, isolation, and diagnosis of self-validating multifunctional sensors”. In: *Review of Scientific Instruments* 87.6 (June 2016). ISSN: 1089-7623. DOI: 10.1063/1.4954184.

[72] Yunluo Yu et al. “A novel sensor fault diagnosis method based on Modified Ensemble Empirical Mode Decomposition and Probabilistic Neural Network”. In: *Measurement* 68 (Apr. 2015), pp. 328–336. ISSN: 0263-2241. DOI: 10.1016/j.measurement.2015.03.003.

[73] G. Udny Yule. “On the Methods of Measuring Association Between Two Attributes”. In: *Journal of the Royal Statistical Society* 75.6 (May 1912), p. 579. ISSN: 0952-8385. DOI: 10.2307/2340126.

A

Appendix 1

A.1 White noise

Definition A.1.1 (White noise)**.** We say that a sequence $\{X(t)\}_{t=0}^{T}$ is white noise with mean μ and variance σ^2 if the two following hold:

(i) $\mathbb{E}[X(t)] = \mu$ for all $t = 0, \ldots, T$ and

(ii) $\mathbb{C}\text{ov}(X(s), X(t)) = \sigma^2 \delta_{st}$ for all $s, t = 0, \ldots, T$,

where δ is the Kronecker delta. To denote that $X(t)$ is white noise with mean μ and variance σ^2 we write

$$X(t) \sim \text{WN}\left(\mu, \sigma^2\right).$$

Proposition A.1.1. *$X(t) \sim \text{IID}\left(\mu, \sigma^2\right) \Rightarrow X(t) \sim \text{WN}\left(\mu, \sigma^2\right)$.*

Proof. The proof is trivial. □

B

Appendix 2

B.1 Note on the difference operator applied to the faults

In this section, some consequences of the difference operator, Δ (see Definition 2.3.2), are discussed and a connection between three of the faults presented in Section 2.2 is presented.

B.1.1 Hardover fault

By applying a first-order differentiation on a faulty signal $\tilde{x}(t)$ with a hardover fault as described by Equation (2.1), we arrive at

$$\Delta\tilde{x}(t) = \begin{cases} \Delta x(t) & \text{if } t \neq \tau, \\ \Delta x(t) + \delta & \text{if } t = \tau. \end{cases} \tag{B.1}$$

We see that $\Delta\tilde{x}(t)$ resembles a spike fault at time τ as described in Section 2.2.3. Thus a method for detecting spike faults on a faulty signal $\tilde{x}(t)$ can also be exploited to detect hardover faults on the same signal by applying it to the differenced signal $\Delta\tilde{x}(t)$.

A drawback of this approach is that all the information about the fault is only available at τ. If $x(t)$ is noisy around τ, a spike might be hard to detect.

B.1.2 Drift fault

By applying a first-order differentiation on a faulty signal $\tilde{x}(t)$ with a drift fault as described by Equation (2.2), we arrive at

$$\Delta\tilde{x}(t) = \begin{cases} \Delta x(t) & \text{if } t < \tau + 1, \\ \Delta x(t) + \lambda & \text{if } t \geqslant \tau + 1. \end{cases} \tag{B.2}$$

We see that $\Delta\tilde{x}(t)$ resembles a hardover fault as described by Equation (2.1). By applying a differencing on Equation (B.2) as in Section B.1.1 we arrive at

$$\Delta^2\tilde{x}(t) = \begin{cases} \Delta^2 x(t) & \text{if } t \neq \tau + 1, \\ \Delta^2 x(t) + \lambda & \text{if } t = \tau + 1. \end{cases} \tag{B.3}$$

We thus see that a method for detecting spike faults on a faulty signal $\tilde{x}(t)$ can be exploited once again to detect drift faults on the same signal by applying the method on the twice differenced signal $\Delta^2 \tilde{x}(t)$. However, in most cases $\lambda \ll \delta$, so another method is most likely more robust and preferable.

B.1.3 Conclusions

One should note that a method based on the results presented in Section B.1.1 and Section B.1.2 probably will not yield a good algorithm for fault detection. An explanation of this is that the difference operator Δ increases the range of values that the time series can take. If $|x(t)| \leqslant M$ then $|\Delta^n x(t)| \leqslant 2^n M$, so spikes will be much harder to detect after applying Δ.

However, this shows an interesting theoretical relation between the three faults. Note that a similar relation would hold in a continuous setting, but instead of using Δ, the *weak derivative* would be applied.

C

Appendix 3

C.1 Pearson's Chi-square test for independence

Pearson's χ^2 test can be used to determine if two categorical variables X and Y are independent, where X has $r \in \mathbb{N}$ categories and Y has $c \in \mathbb{N}$ categories. The test hypotheses for this test are

$$\begin{cases} H_0: & X \text{ and } Y \text{ are independent,} \\ H_{\text{A}}: & X \text{ and } Y \text{ are not independent.} \end{cases}$$

N Observations are then recorded in an $r \times c$ contingency table, where O_{ij} denotes the number of observations in cell (i, j). Now, for this test, we define the expected number of observations E_{ij} for cell (i, j) to be

$$E_{ij} = N\mathbb{P}(X = i)\,\mathbb{P}(Y = j) = \frac{1}{N}\left(\sum_{j=1}^{c} O_{ij}\right)\left(\sum_{i=1}^{r} O_{ij}\right), \tag{C.1}$$

where the probabilities are estimated from the observed frequencies. The reasons for making this definition is that under H_0, X and Y are assumed to be independent, so by definition $\mathbb{P}(X = i, Y = j) = \mathbb{P}(X = i)\,\mathbb{P}(Y = j)$. From this the standard test statistic

$$\chi^2 = \sum_{i=1}^{r}\sum_{j=1}^{c} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}, \tag{C.2}$$

can be computed. Now, for some α we can compute a critical value $\chi^2_{\alpha,\text{df}}$, where the *degrees of freedom* (df) is given by

$$\text{df} = (r - 1)(c - 1). \tag{C.3}$$

We can thus reject H_0 in favor of H_{A}, i.e. that X and Y are not independent, at significance level α if

$$\chi^2 > \chi^2_{\alpha,\text{df}}. \tag{C.4}$$

D

Appendix 4

D.1 Results

In this appendix some results in terms of the metrics presented in Section for different methods are presented.

D.1.1 binary classification

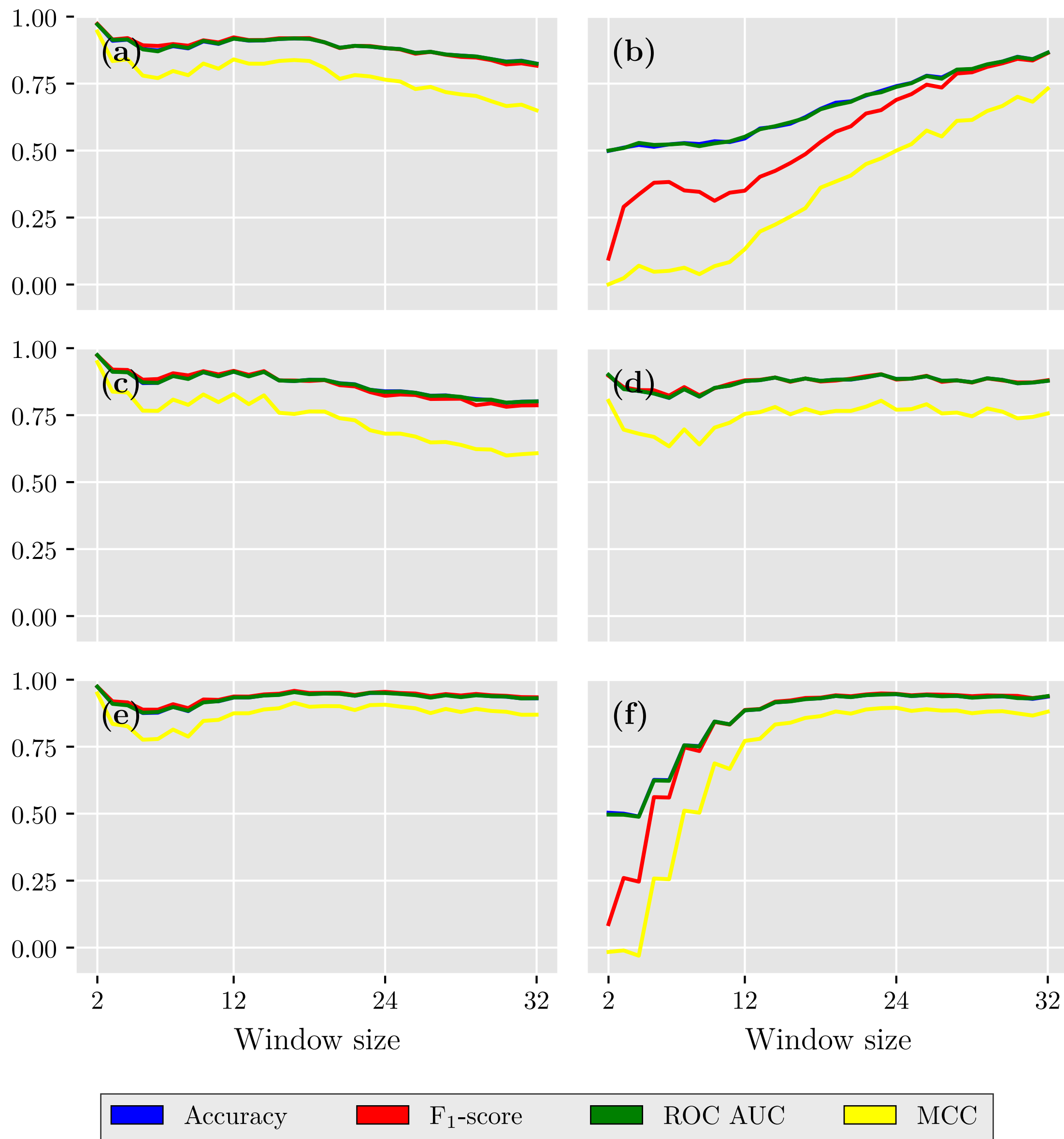


Figure D.1: Performance metrics for the baseline classifier from Section 3.2.2 on a test set that consists 50 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

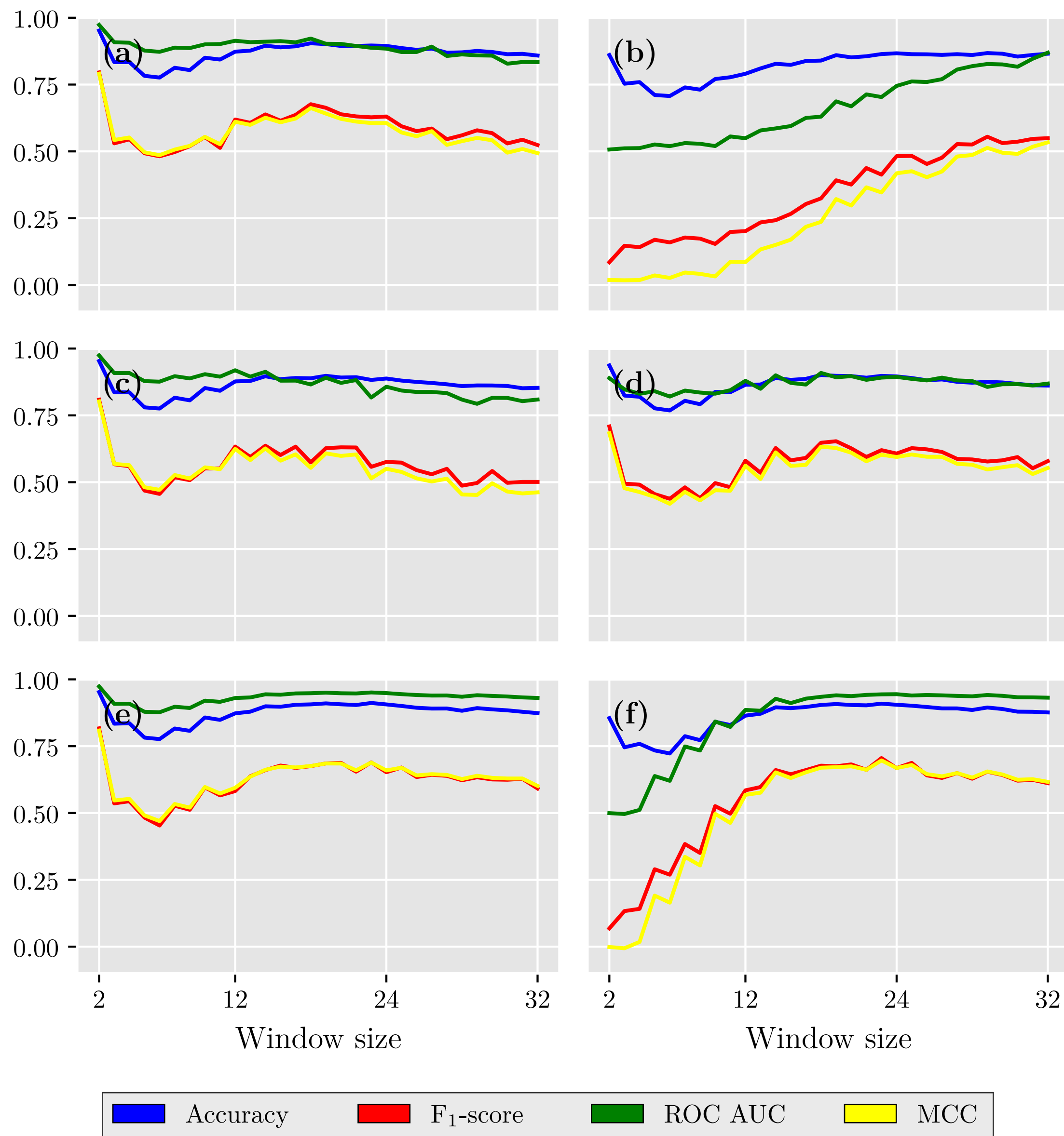


Figure D.2: Performance metrics for the baseline classifier from Section 3.2.2 on a test set that consists 10 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

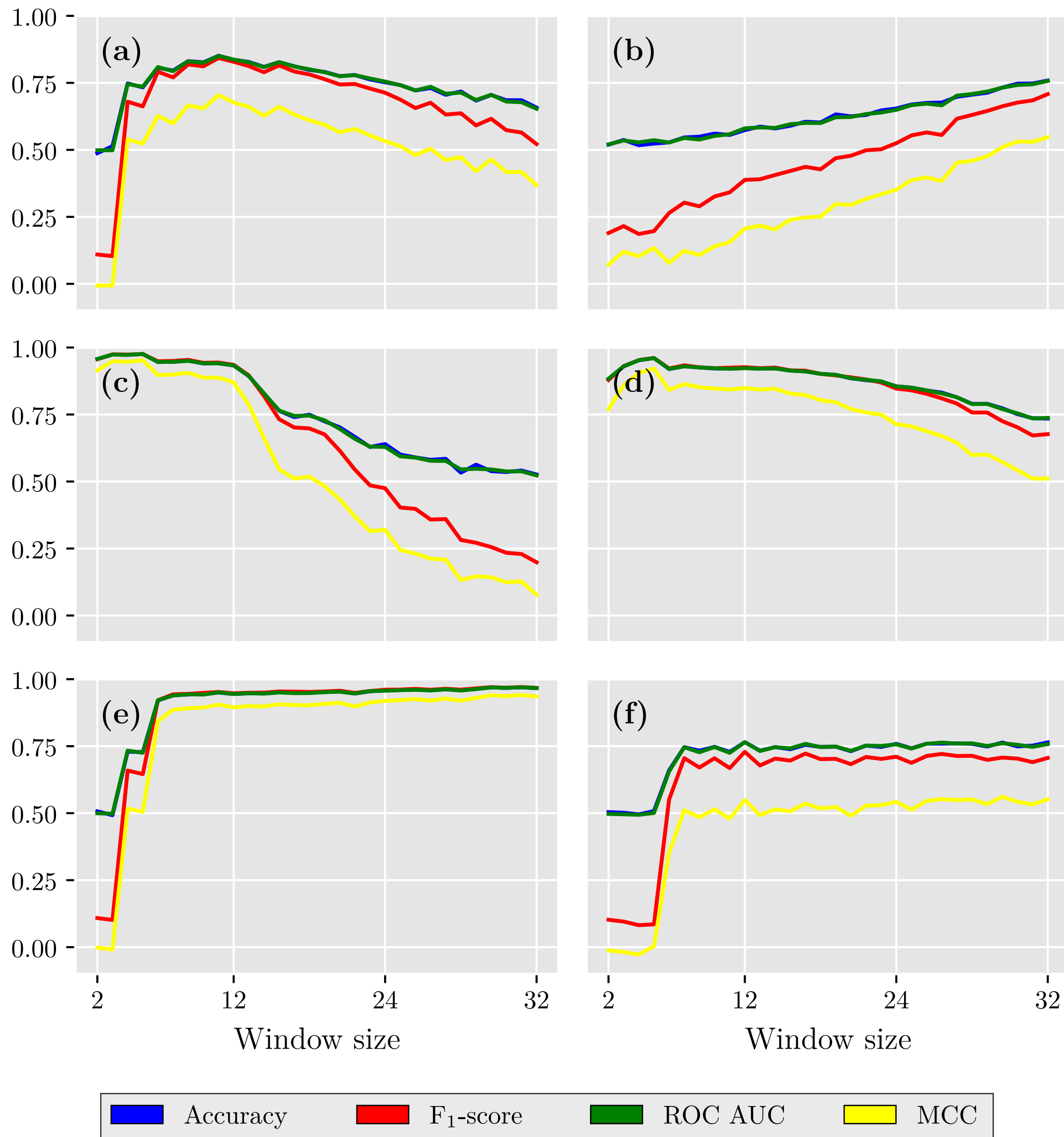


Figure D.3: Performance metrics for the bootstrap classifier from Section 3.2.3 on a test set that consists 50 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

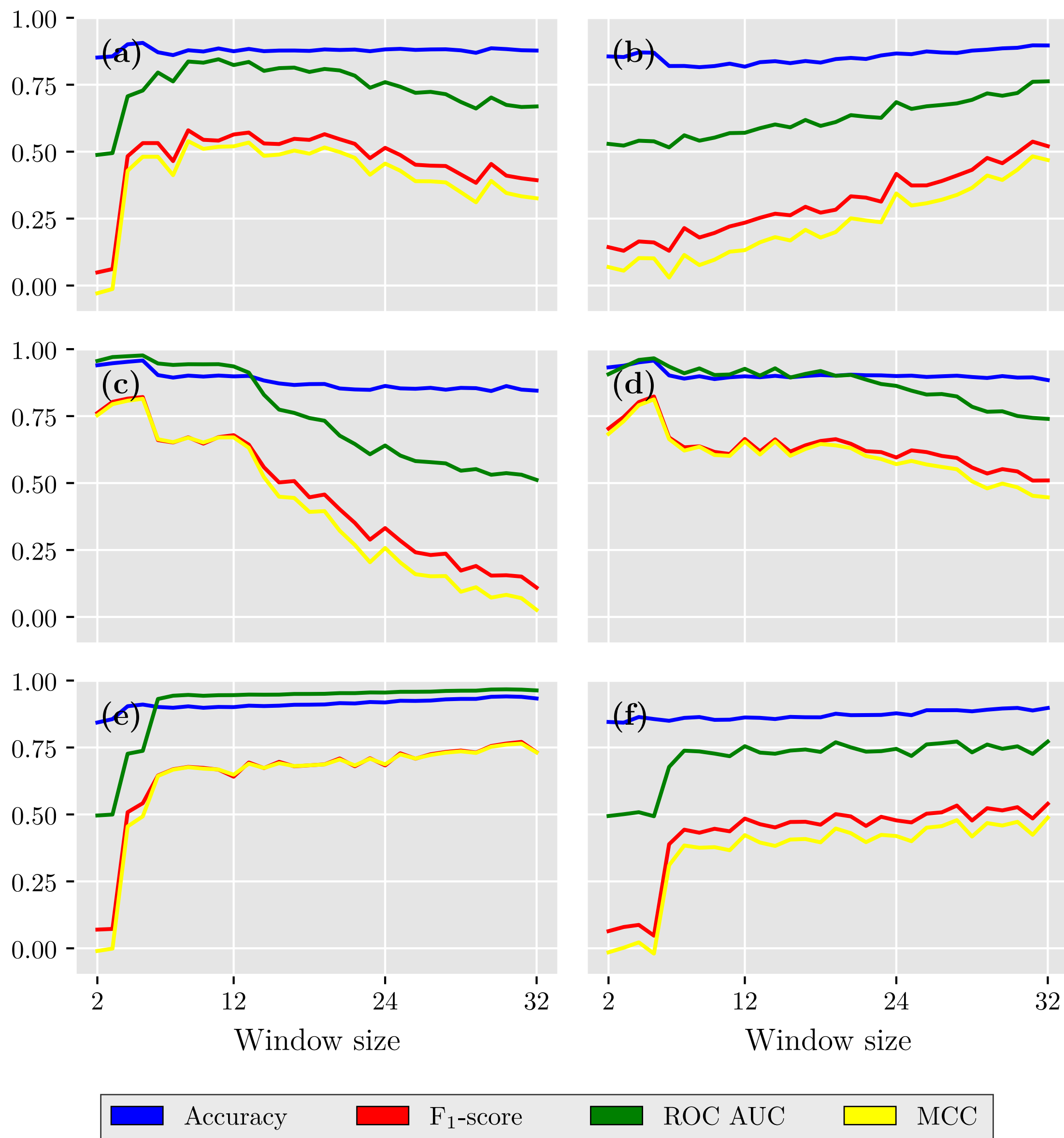


Figure D.4: Performance metrics for the bootstrap classifier from Section 3.2.3 on a test set that consists 10 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

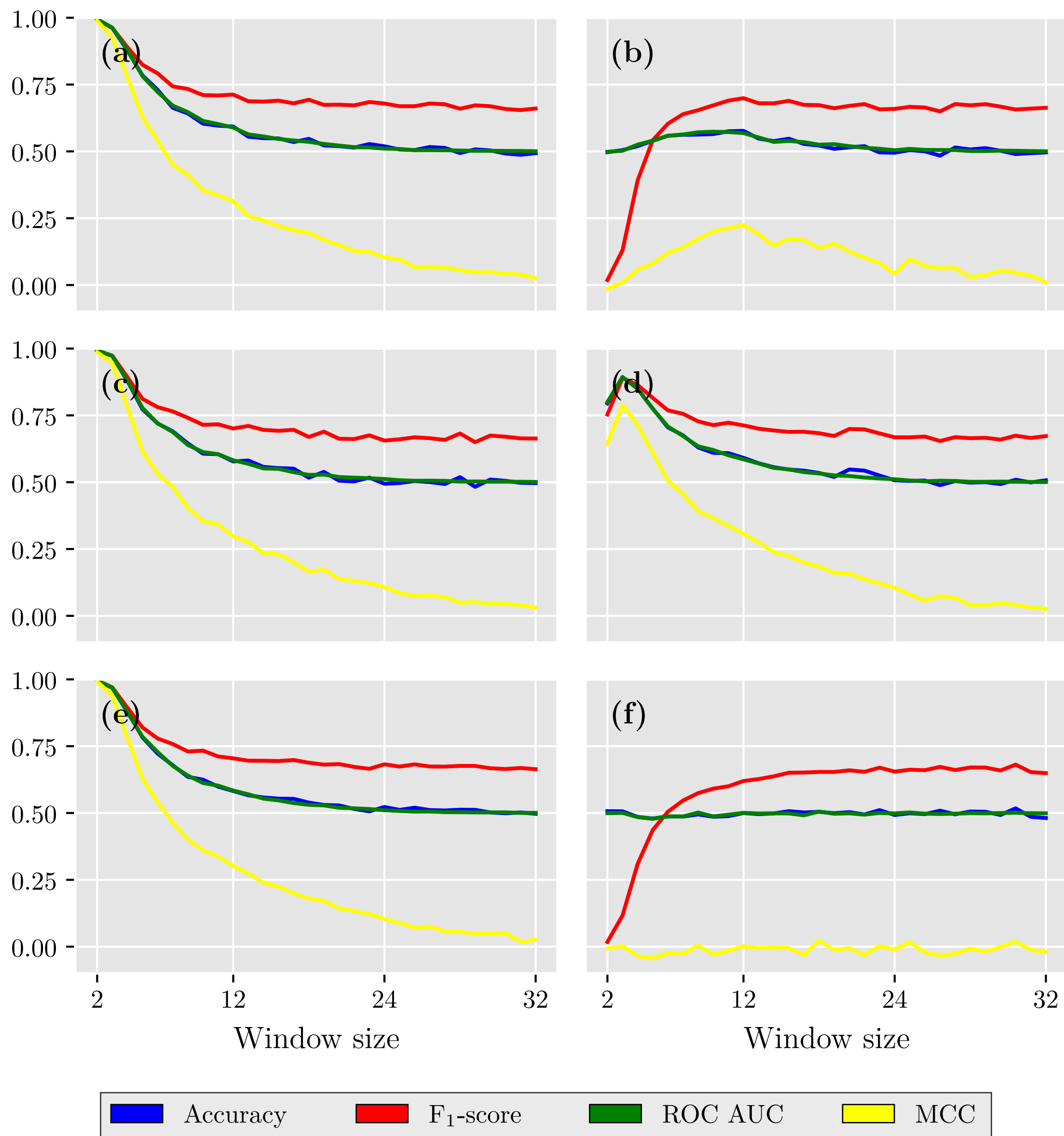


Figure D.5: Performance metrics for the ARIMA classifier from Section 3.2.4 on a test set that consists 50 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

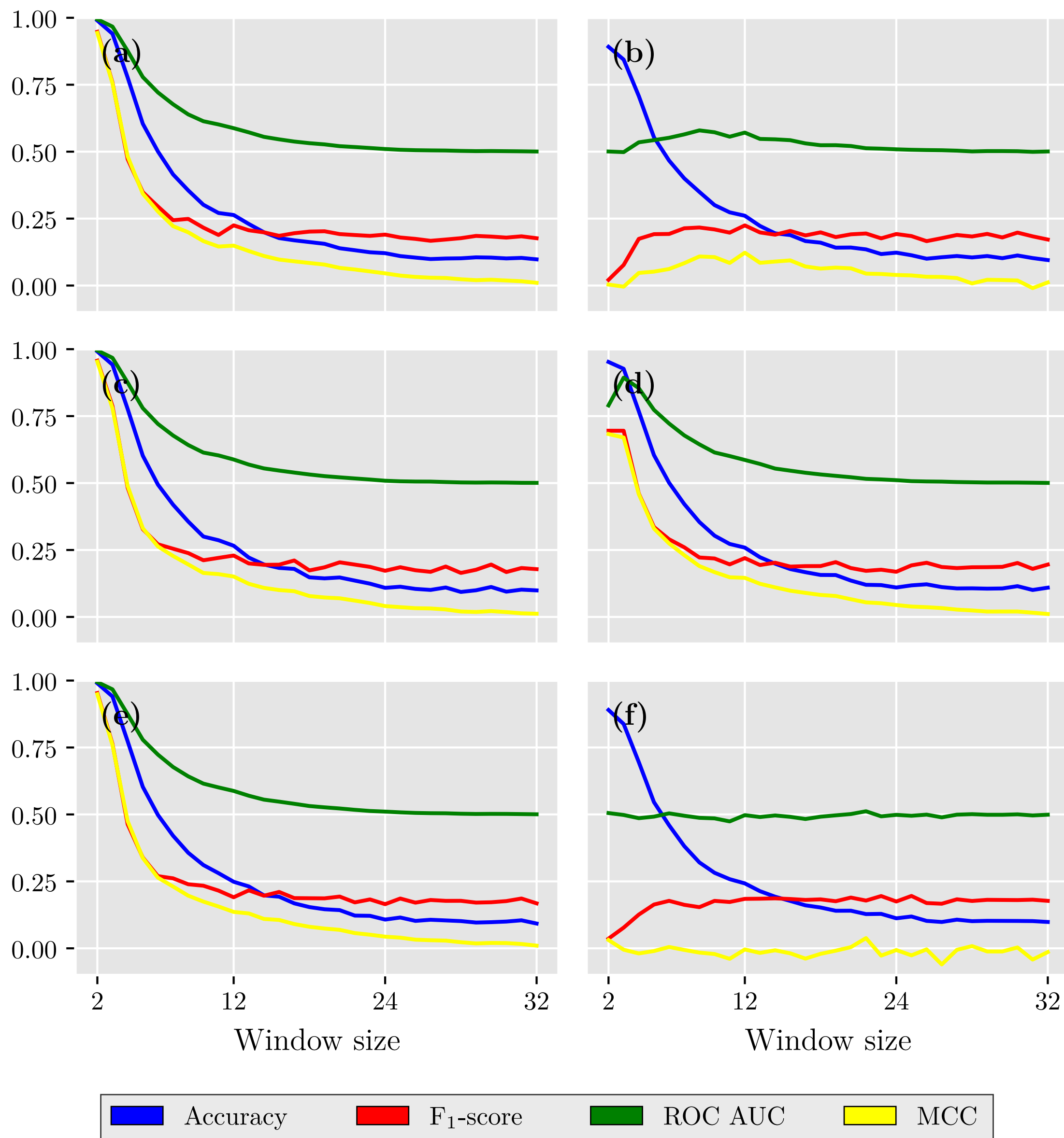


Figure D.6: Performance metrics for the ARIMA classifier from Section 3.2.4 on a test set that consists 10 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

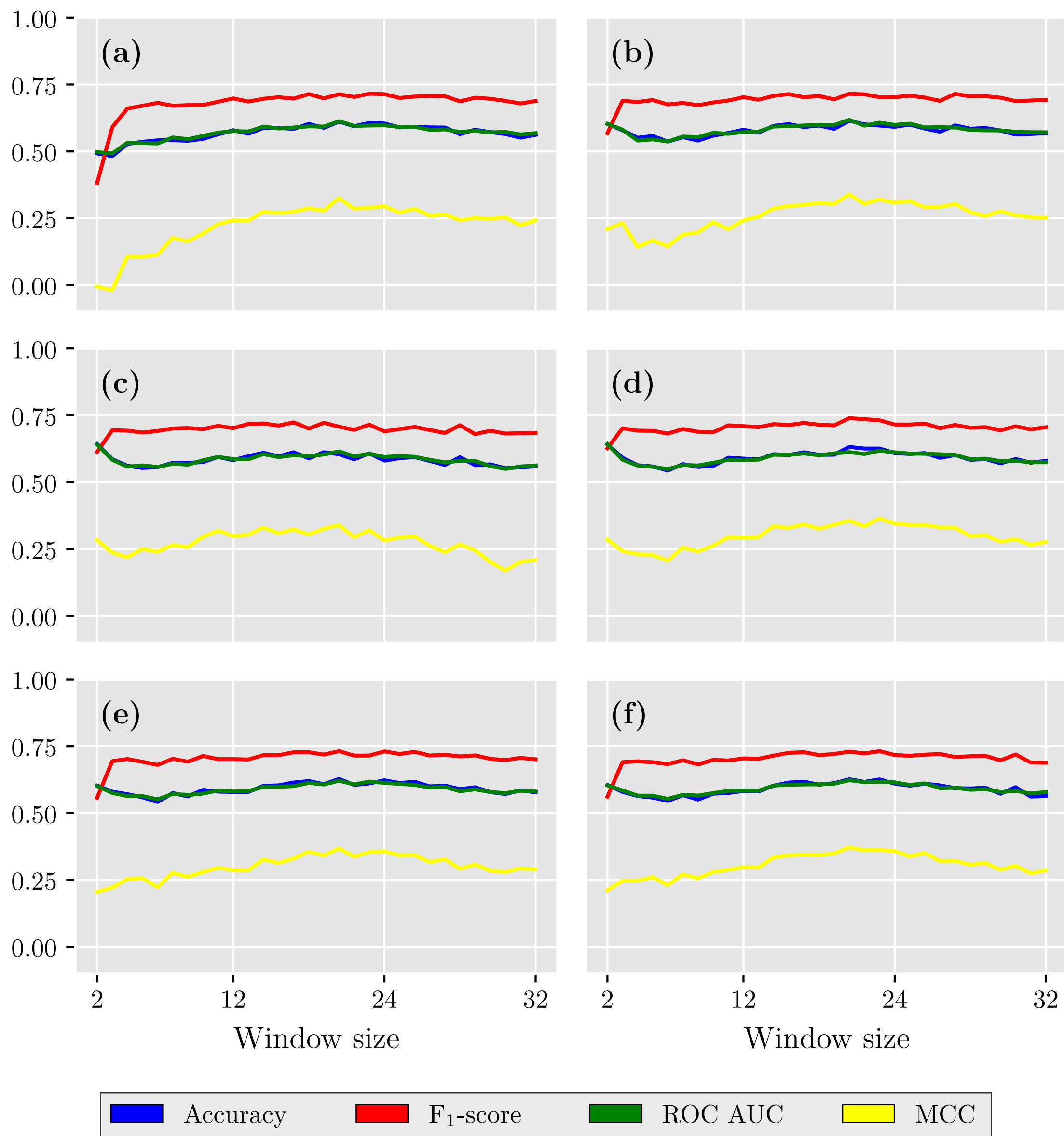


Figure D.7: Performance metrics for the extra trees classifier from Section 3.2.5 on a test set that consists 50 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

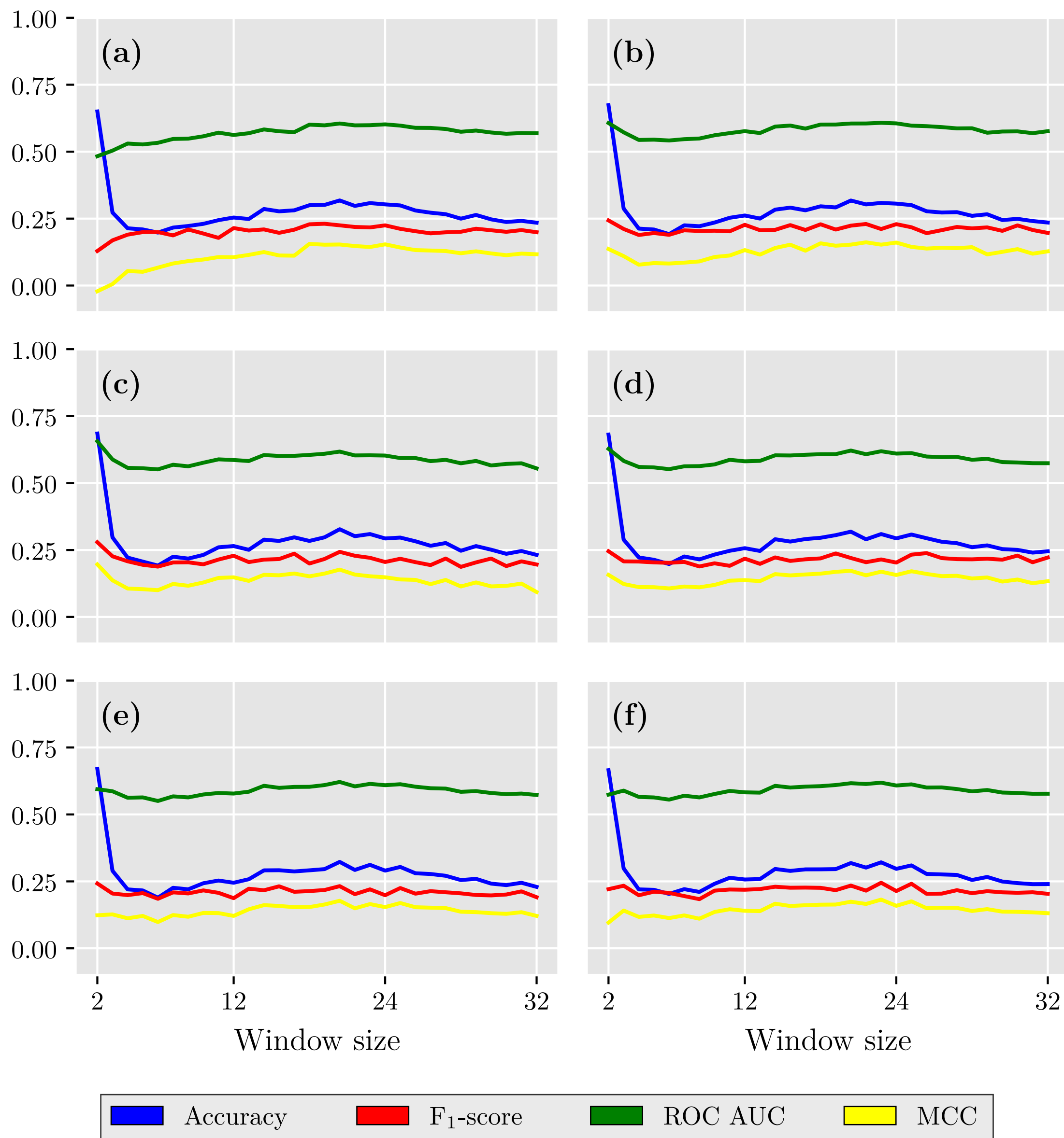


Figure D.8: Performance metrics for the extra trees classifier from Section 3.2.5 on a test set that consists 10 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

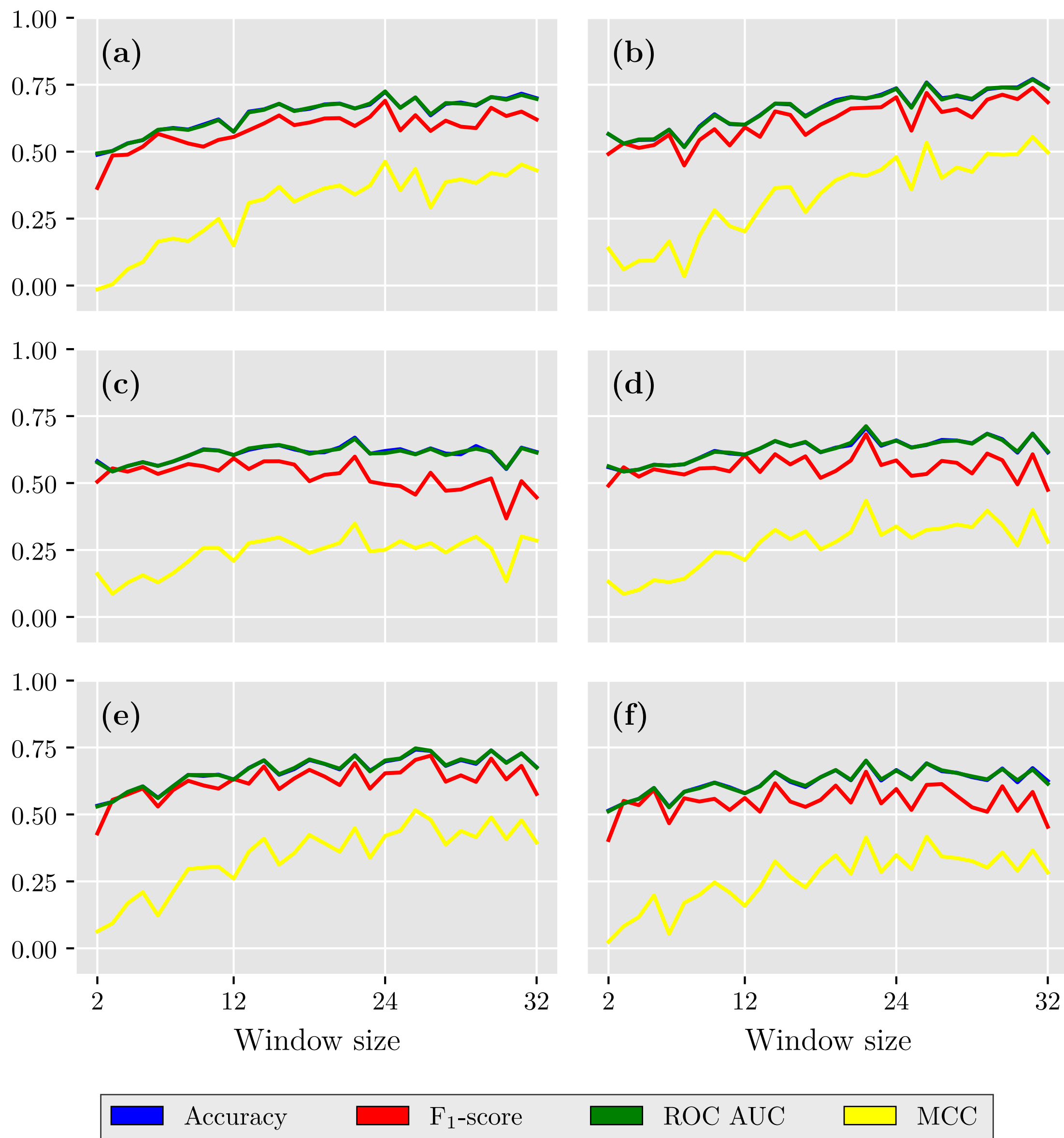


Figure D.9: Performance metrics for the kNN-DTW classifier from Section 3.2.6 on a test set that consists 50 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

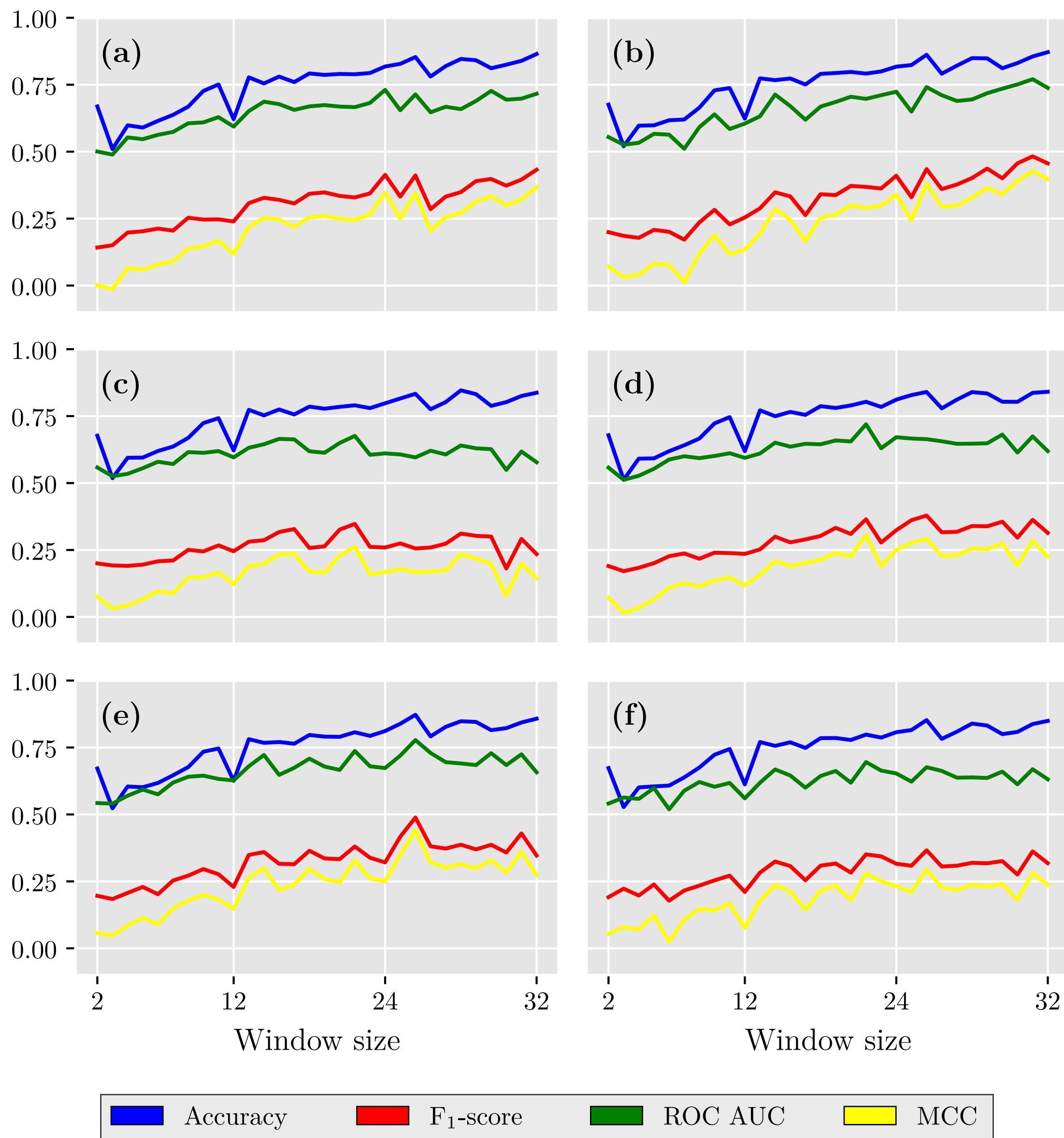


Figure D.10: Performance metrics for the kNN-DTW classifier from Section 3.2.6 on a test set that consists 10 % of faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

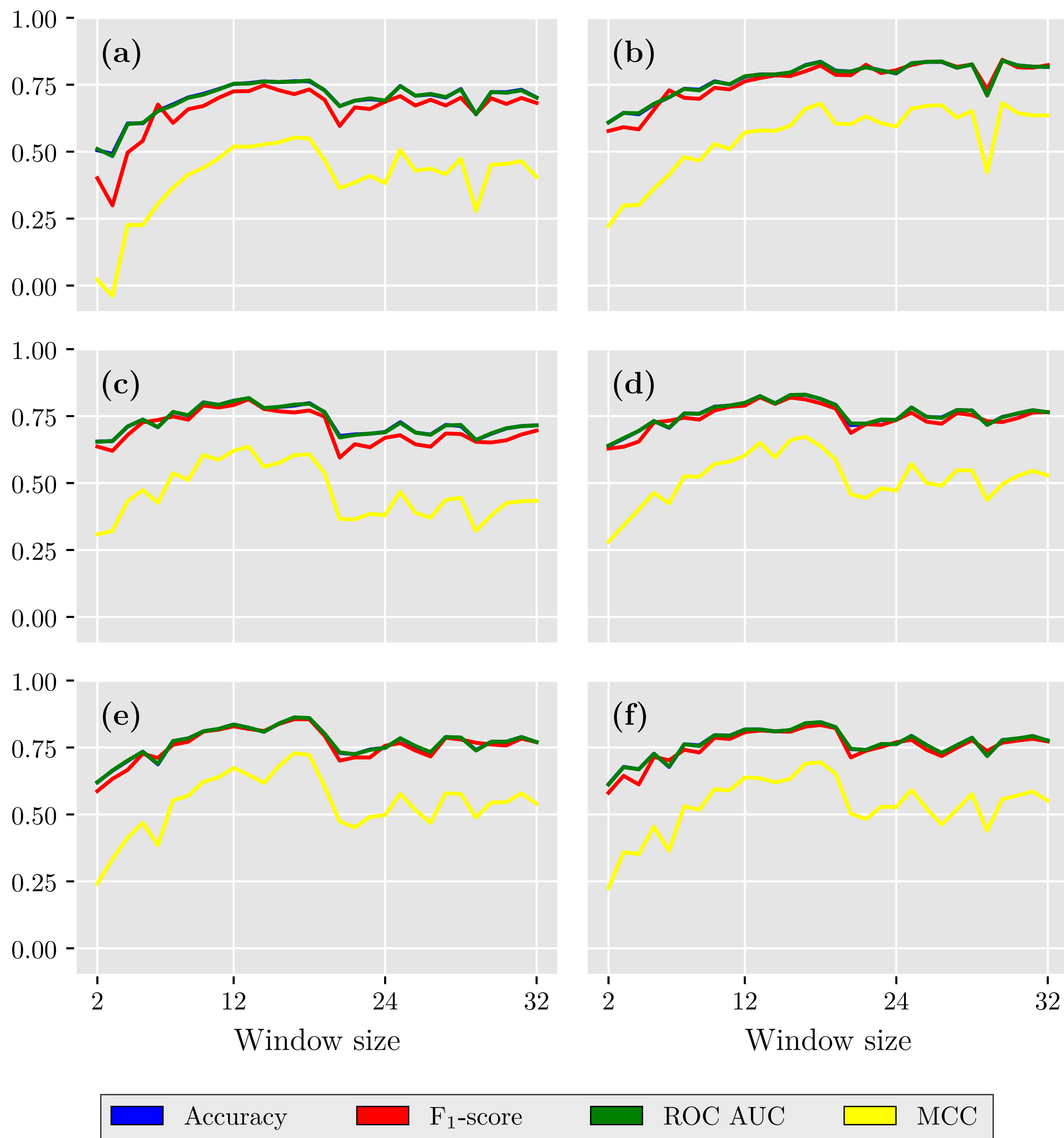


Figure D.11: Performance metrics for the Elamn RNN classifier from Section 3.2.8 on a test set that consists 50 % of faulty data for each of the faults. Note that the test set only consists of on type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

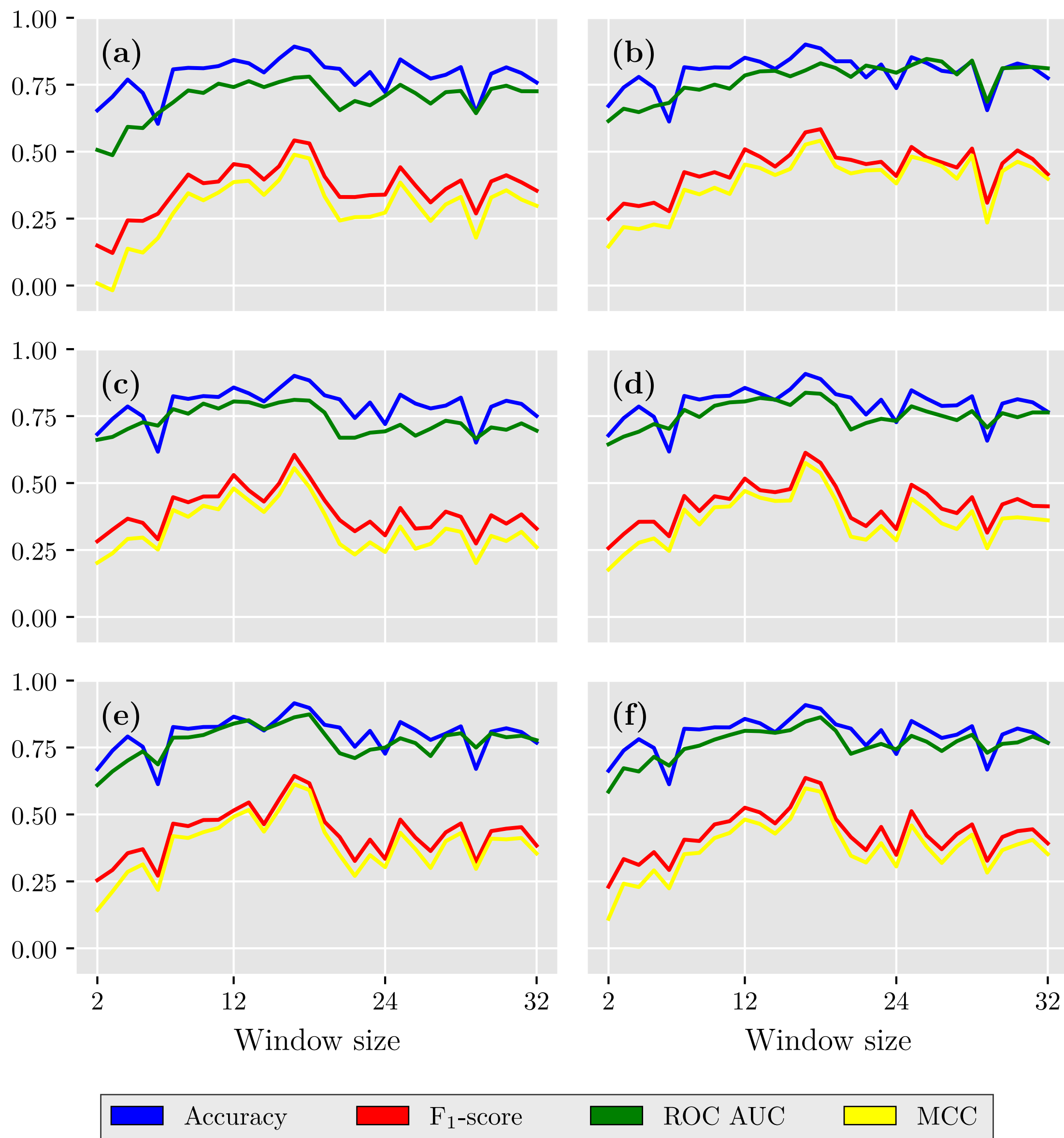


Figure D.12: Performance metrics for the Elman RNN classifier from Section 3.2.8 on a test set that consists 10 % of faulty data for each of the faults. Note that the test set only consists of on type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

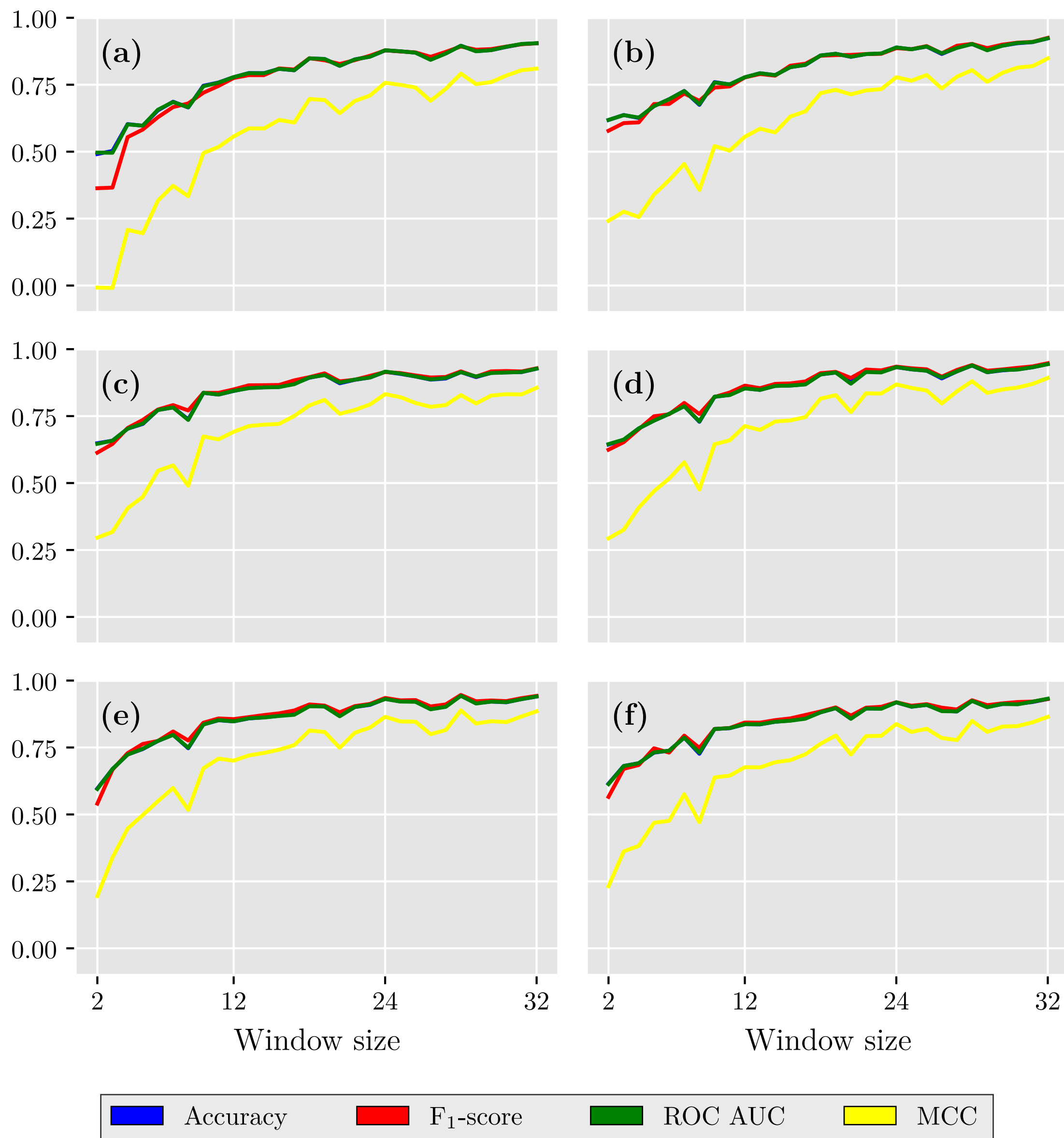


Figure D.13: Performance metrics for the LSTM classifier from Section 3.2.7 on a test set that consists 50 % of faulty data for each of the faults. Note that the test set only consists of on type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

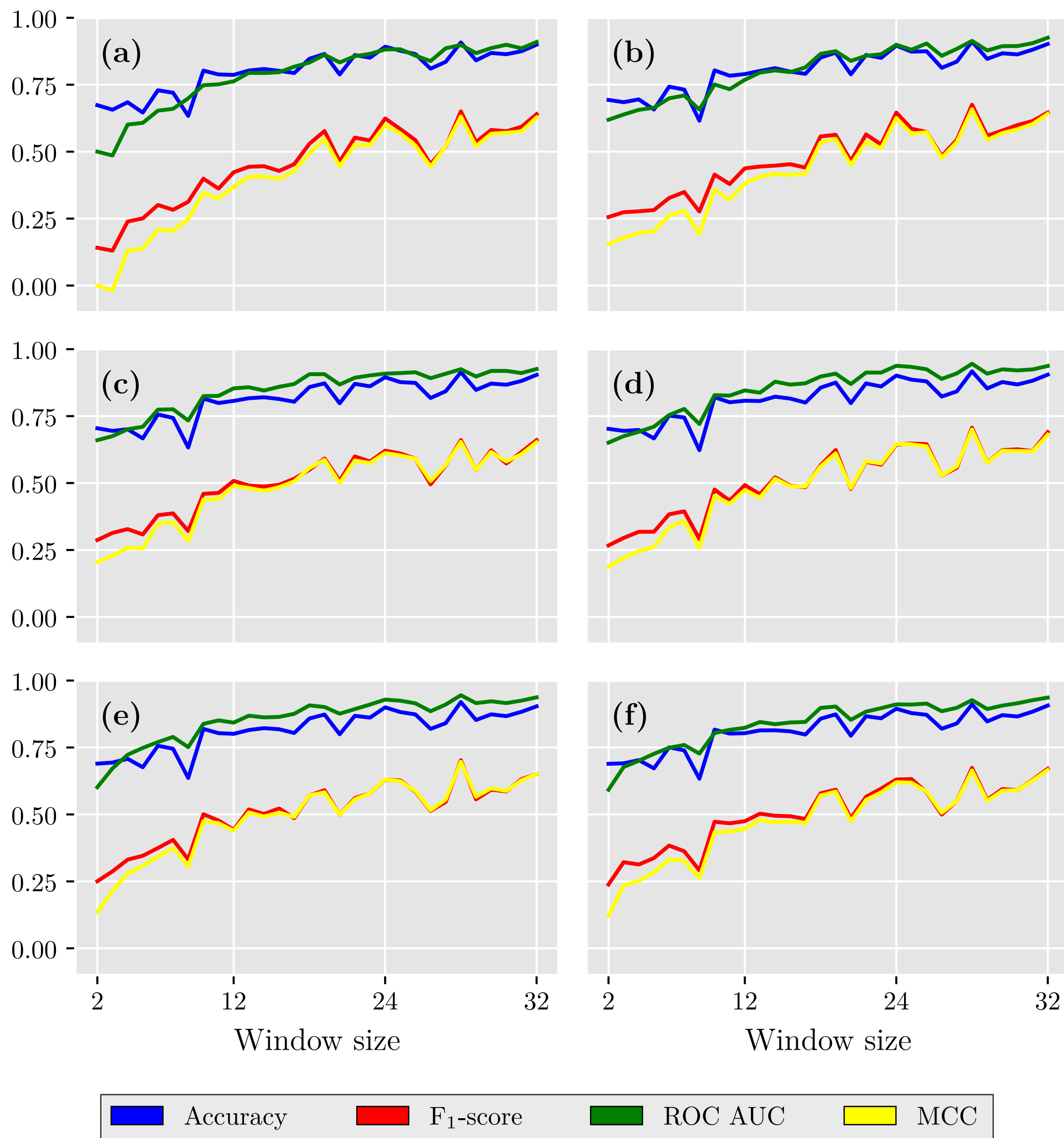


Figure D.14: Performance metrics for the LSTM classifier from Section 3.2.7 on a test set that consists 10 % of faulty data for each of the faults. Note that the test set only consists of on type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

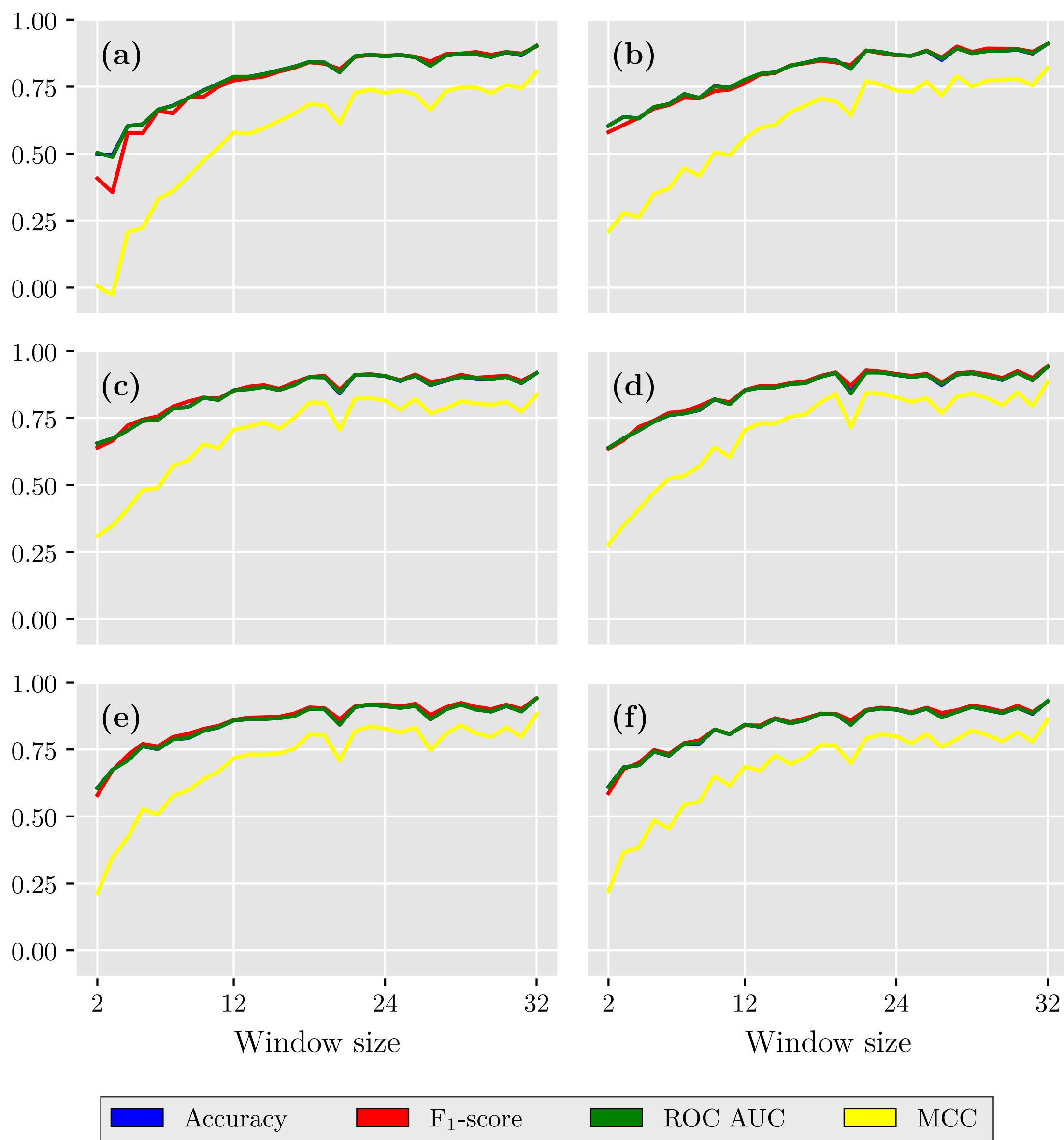


Figure D.15: Performance metrics for the GRU classifier from Section 3.2.8 on a test set that consists 50 % of faulty data for each of the faults. Note that the test set only consists of on type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

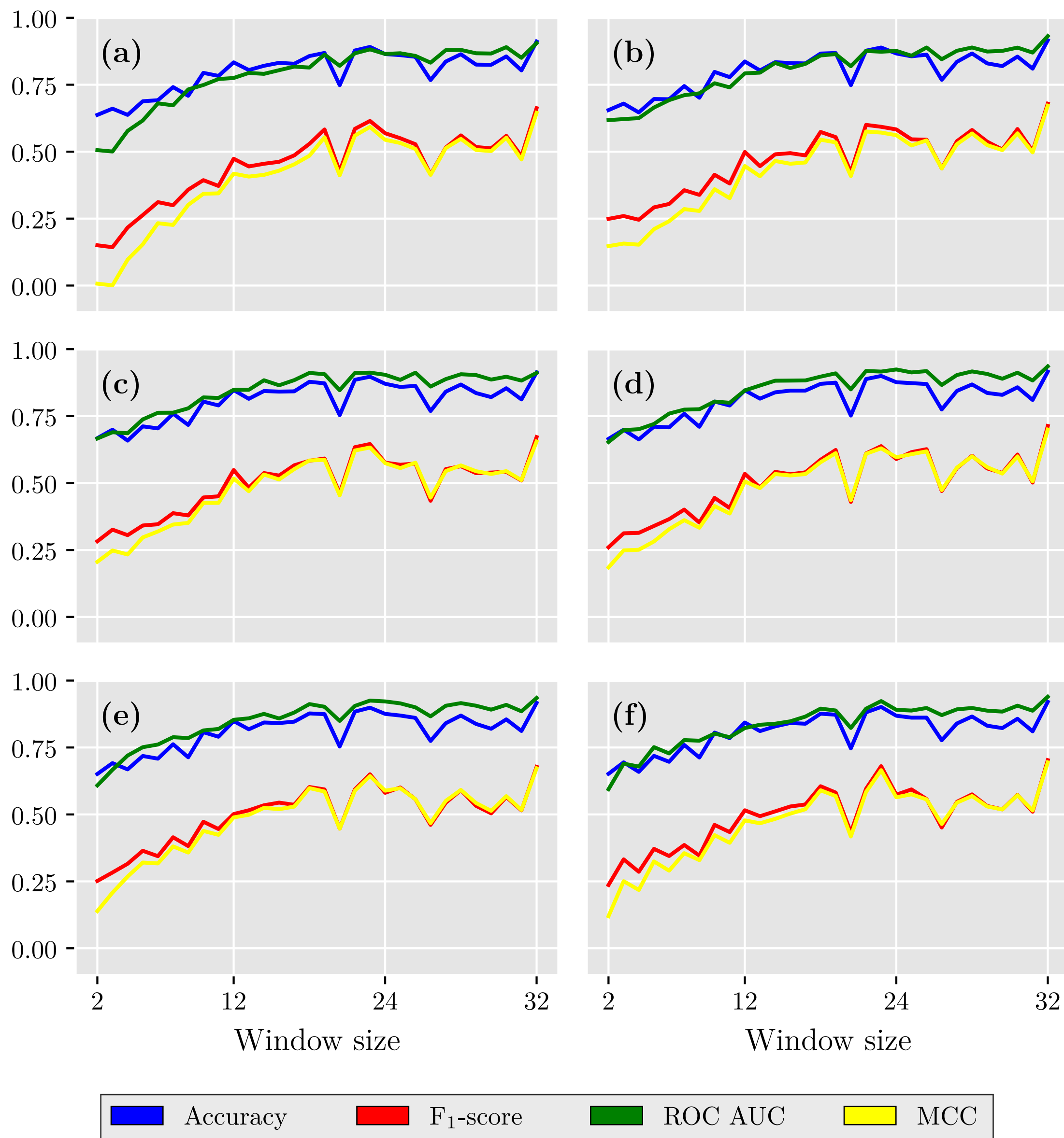


Figure D.16: Performance metrics for the GRU classifier from Section 3.2.8 on a test set that consists 10 % of faulty data for each of the faults. Note that the test set only consists of on type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

D.1.2 multiclass classification

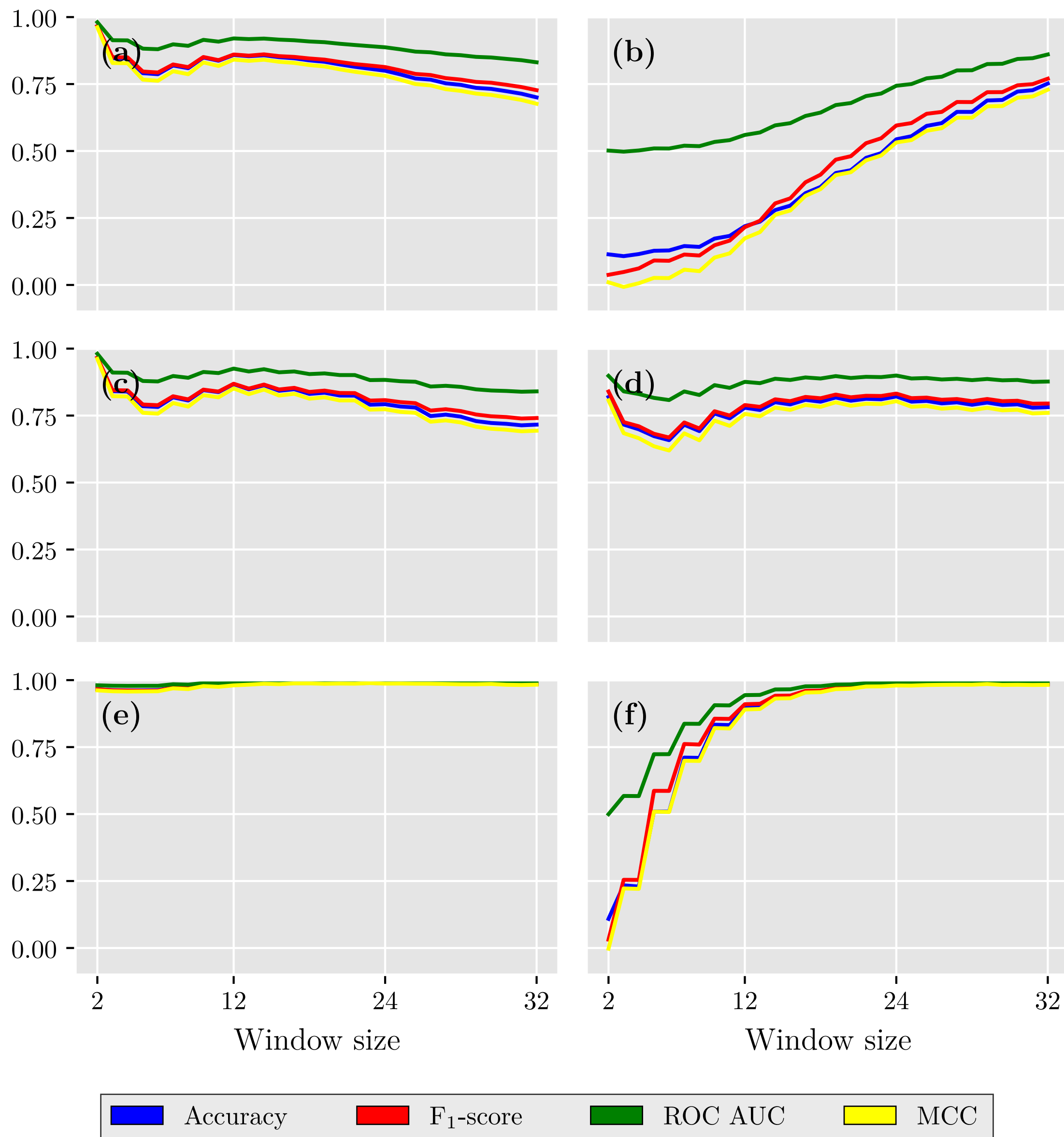


Figure D.17: Performance metrics for the baseline classifier from Section 3.2.2 on a test set that consists of balanced data set (i.e., each class represents ≈11 % of the data set) for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

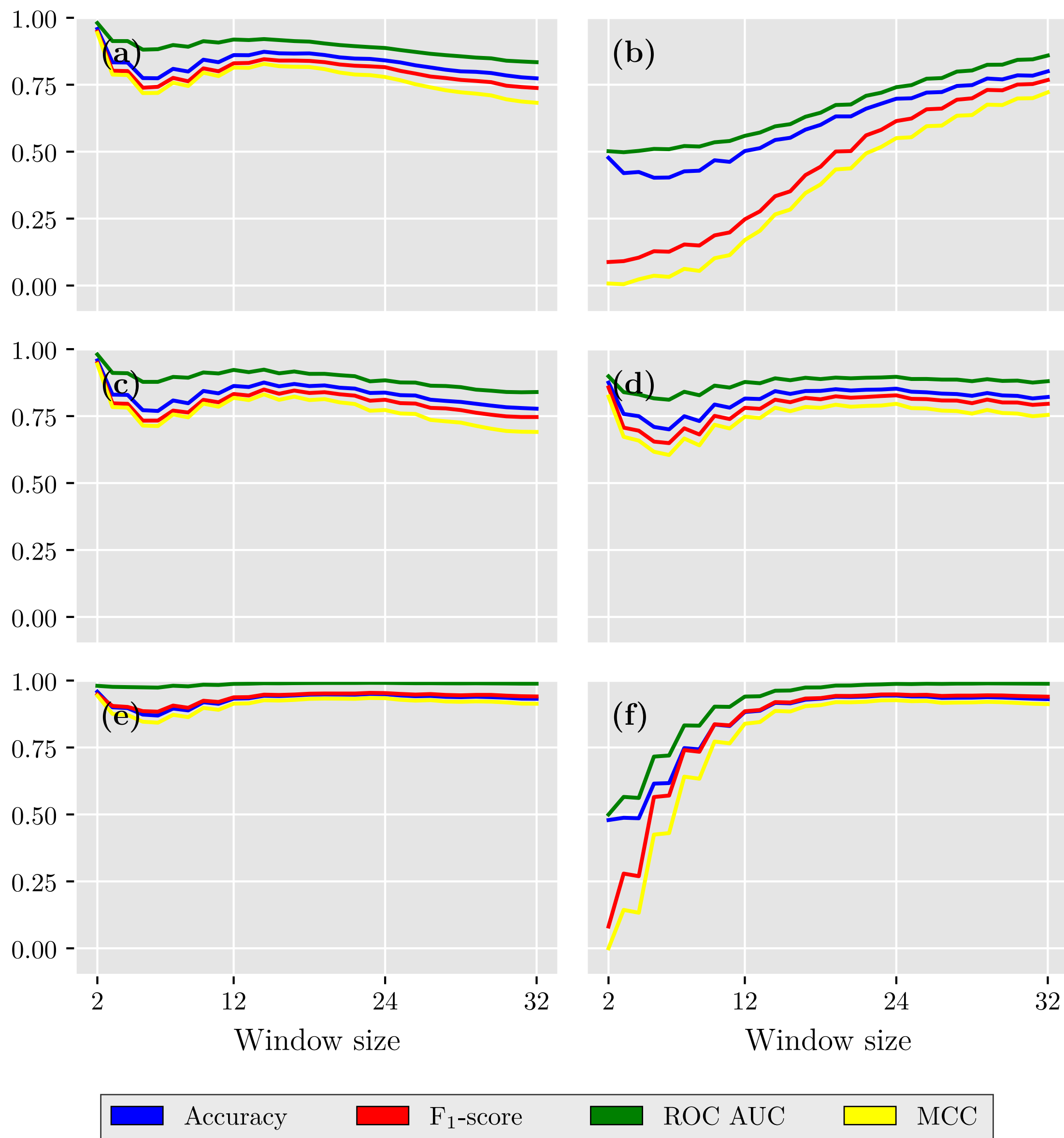


Figure D.18: Performance metrics for the baseline classifier from Section 3.2.2 on a test set that consists 50 % of healthy and 50 % faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

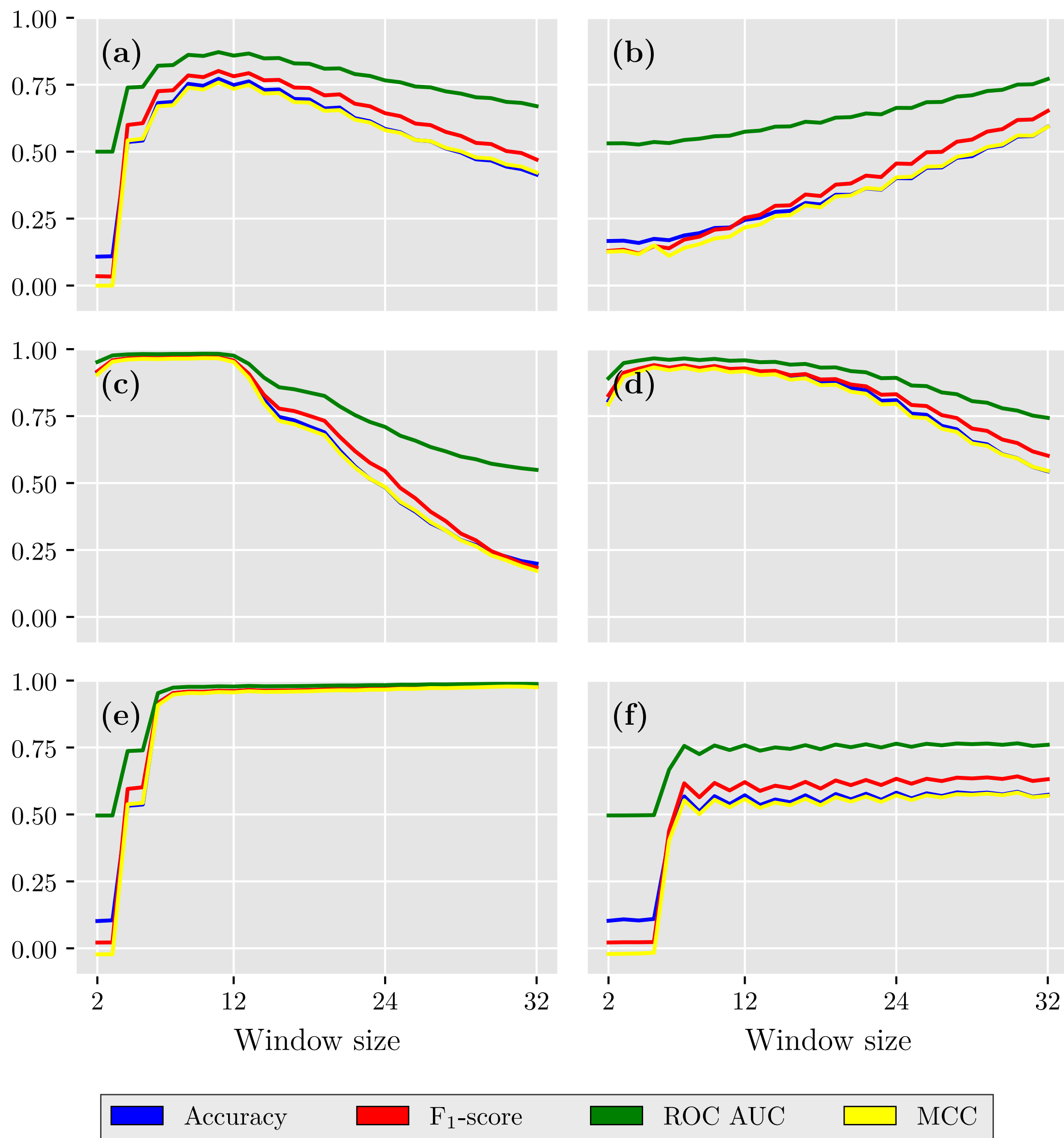


Figure D.19: Performance metrics for the bootstrap classifier from Section 3.2.3 on a test set that consists of balanced data set (i.e., each class represents ≈11 % of the data set) for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

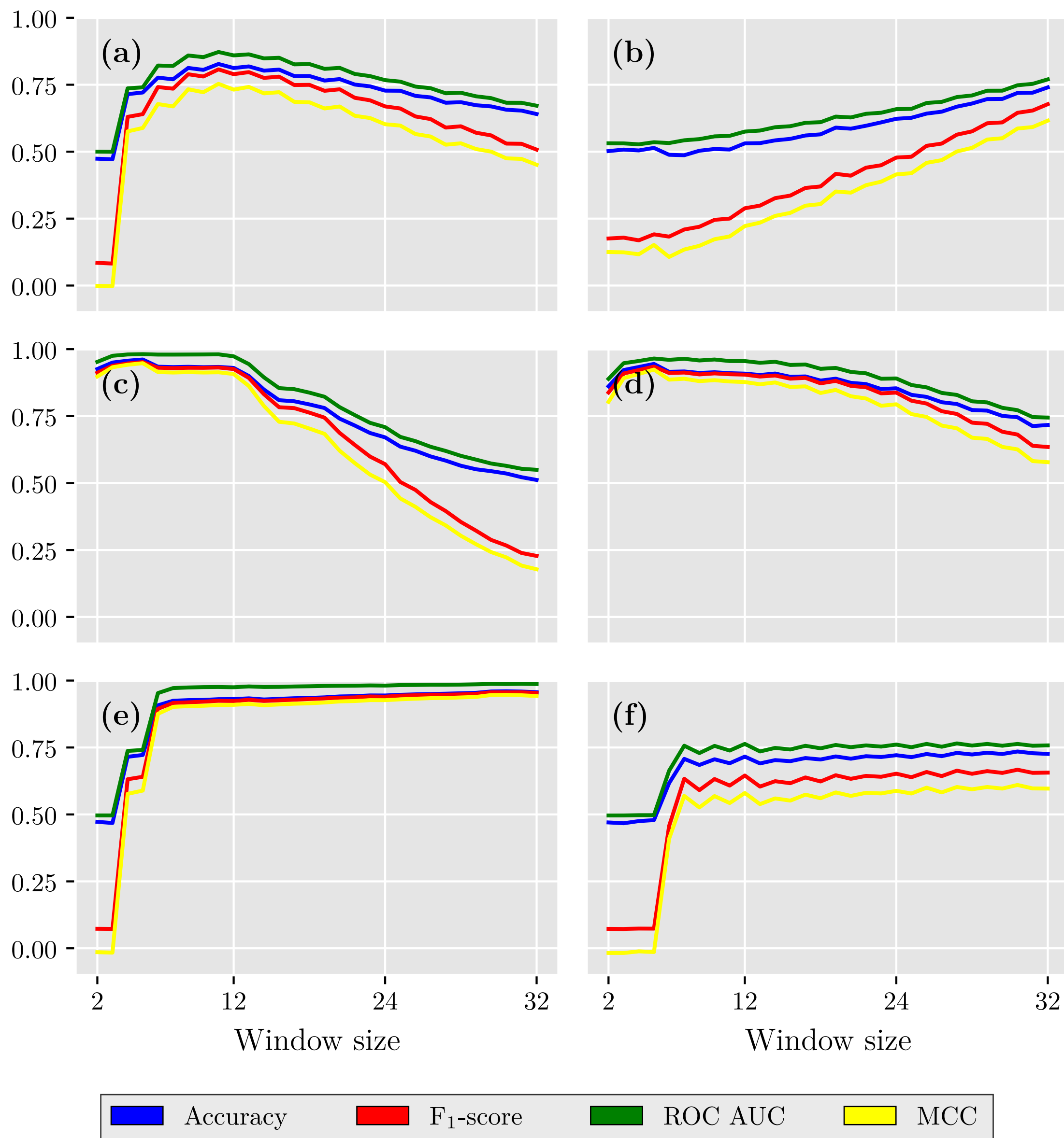


Figure D.20: Performance metrics for the bootstrap classifier from Section 3.2.3 on a test set that consists 50 % of healthy and 50 % faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

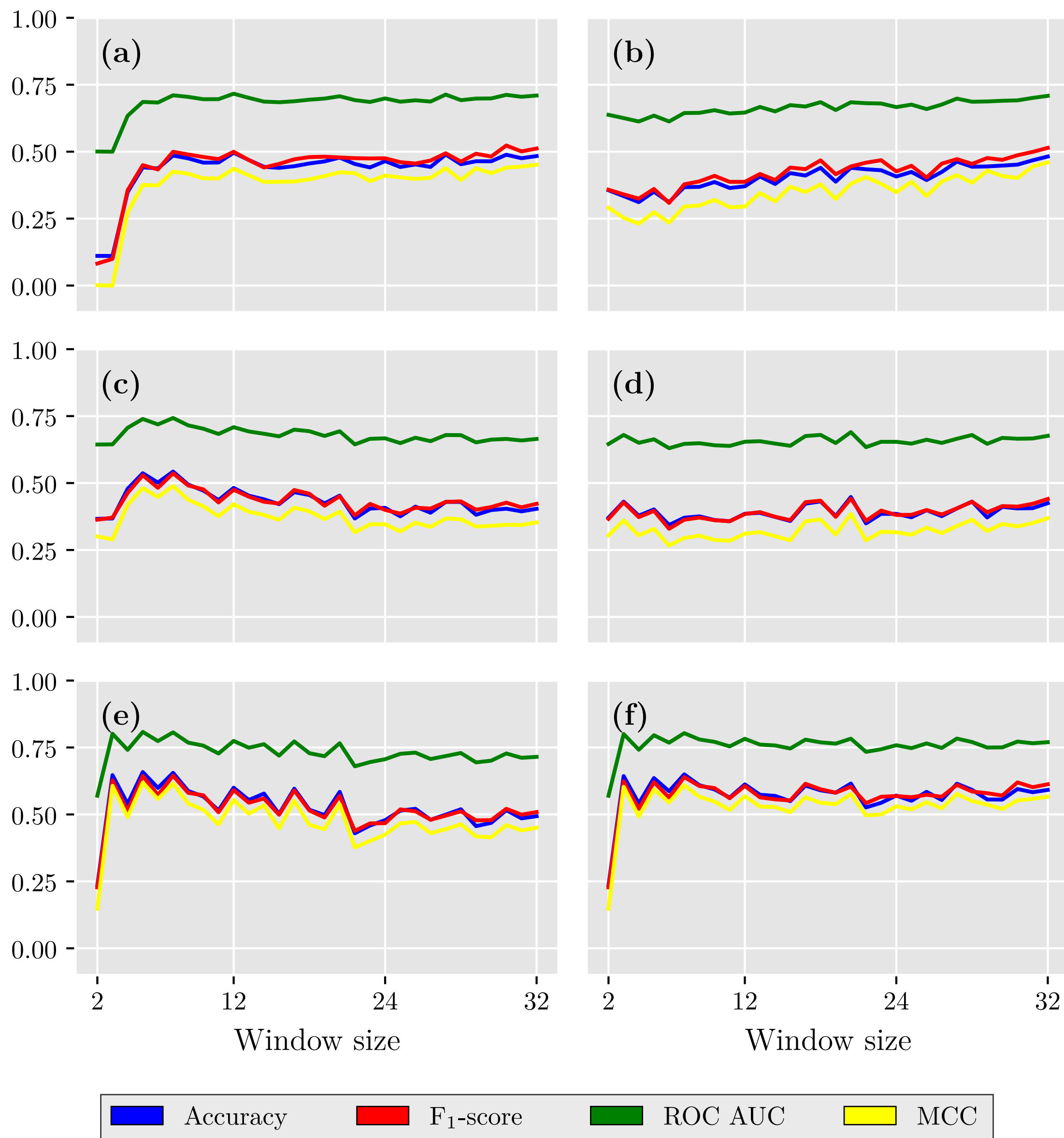


Figure D.21: Performance metrics for the Extra Trees classifier from Section 3.2.5 on a test set that consists of balanced data set (i.e., each class represents ≈11 % of the data set) for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

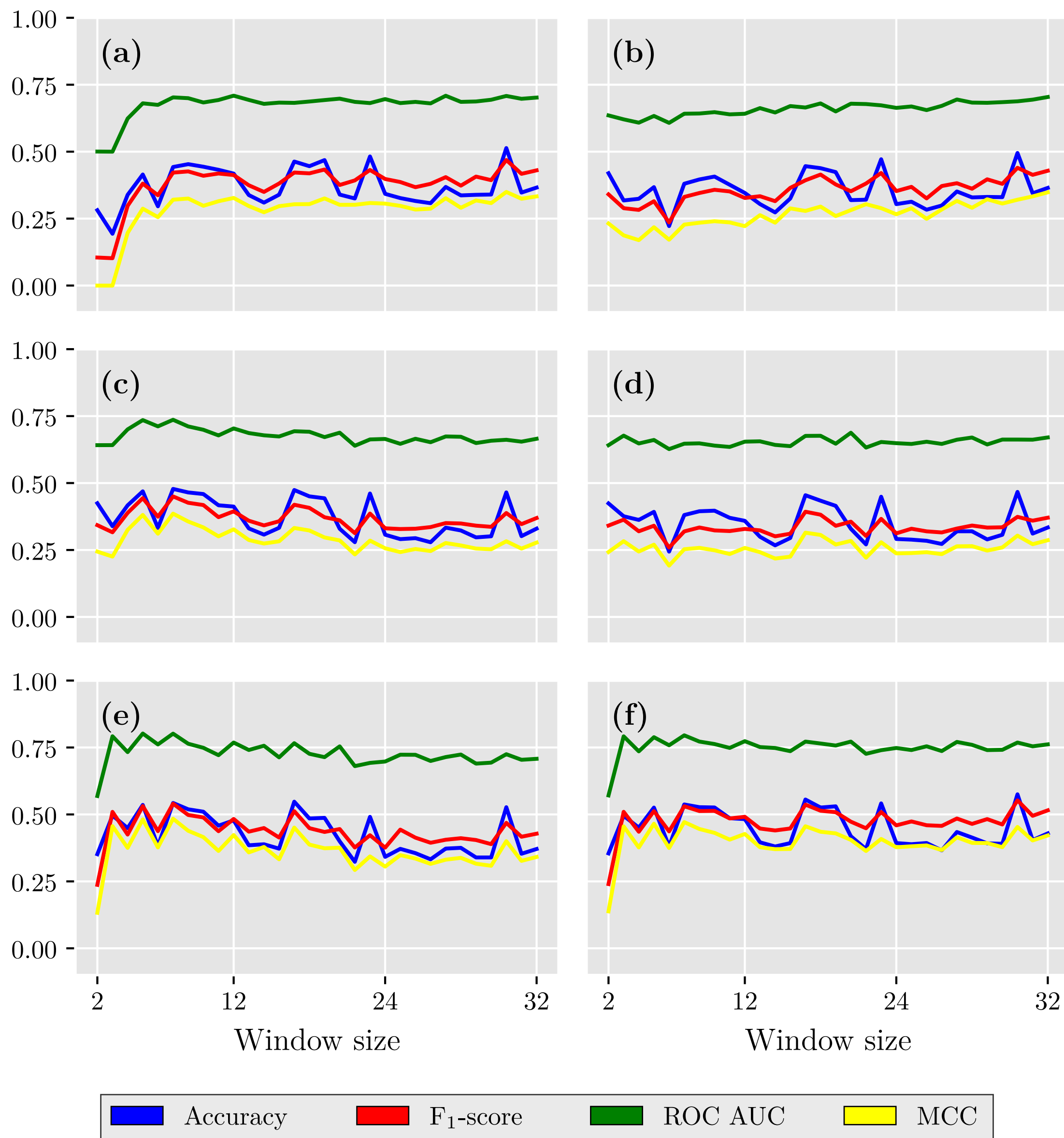


Figure D.22: Performance metrics for the Extra Trees classifier from Section 3.2.5 on a test set that consists 50 % of healthy and 50 % faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

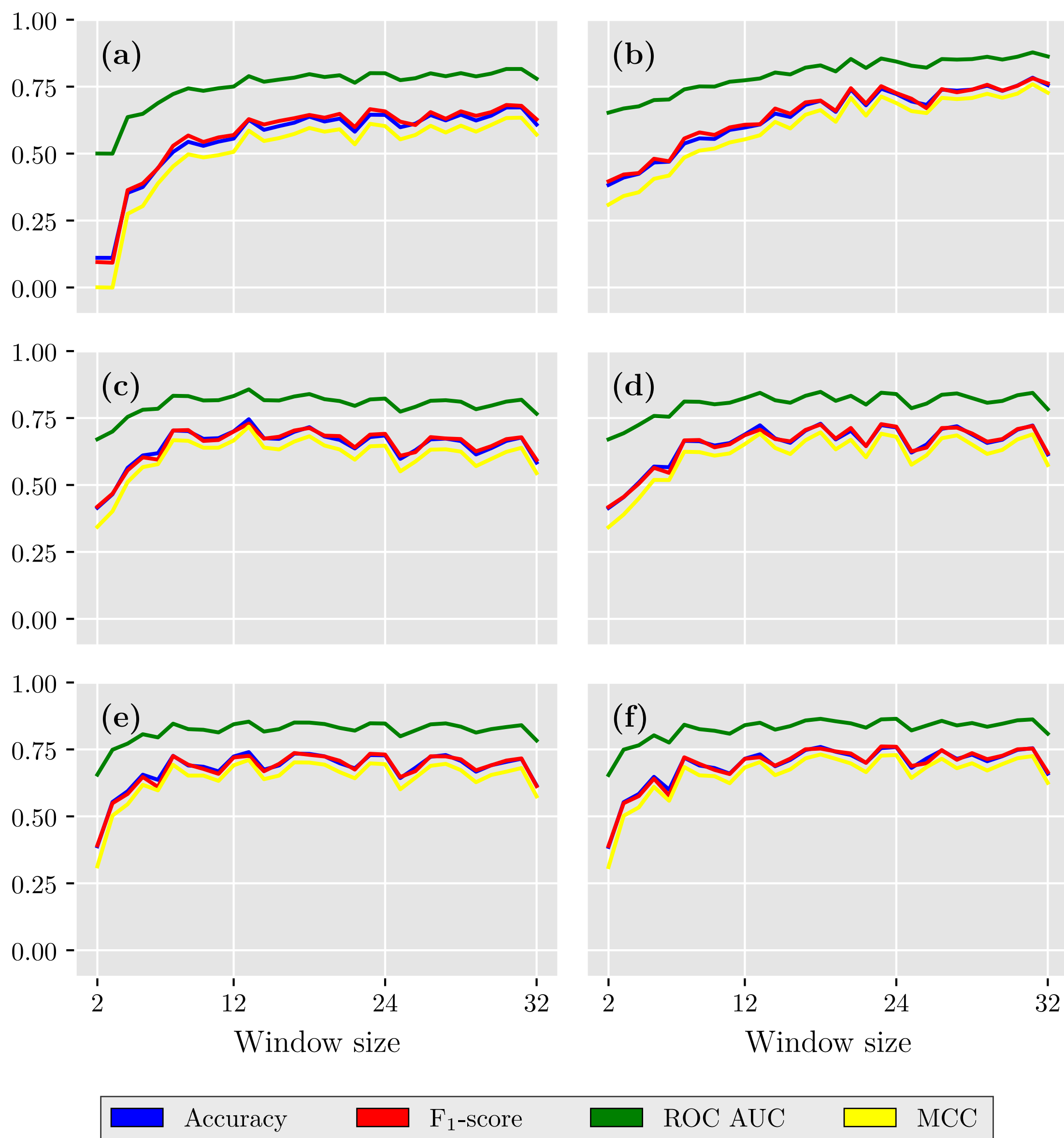


Figure D.23: Performance metrics for the Elman RNN classifier from Section 3.2.8 on a test set that consists of balanced data set (i.e., each class represents ≈11 % of the data set) for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

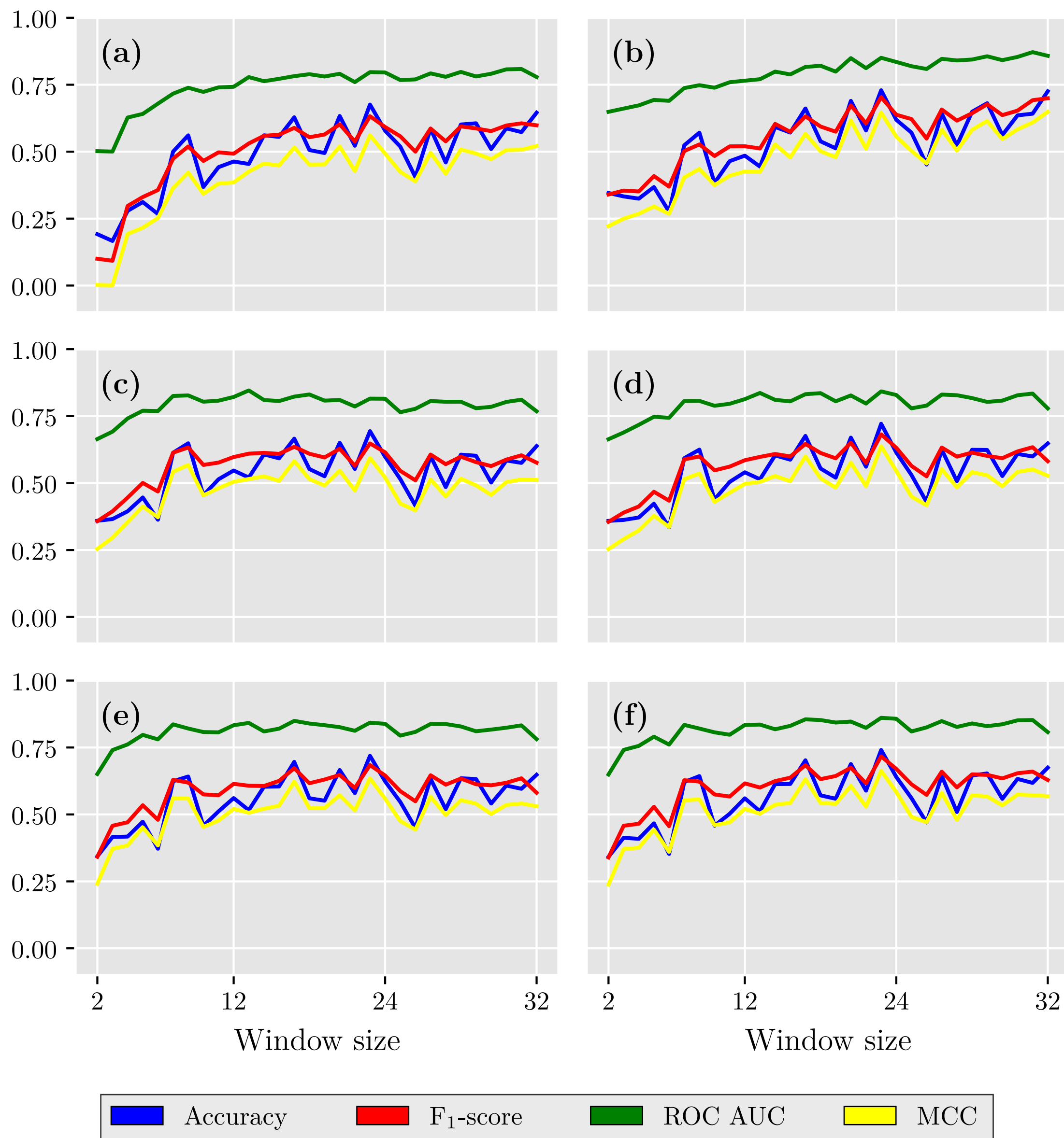


Figure D.24: Performance metrics for the Elman RNN classifier from Section 3.2.8 on a test set that consists 50 % of healthy and 50 % faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

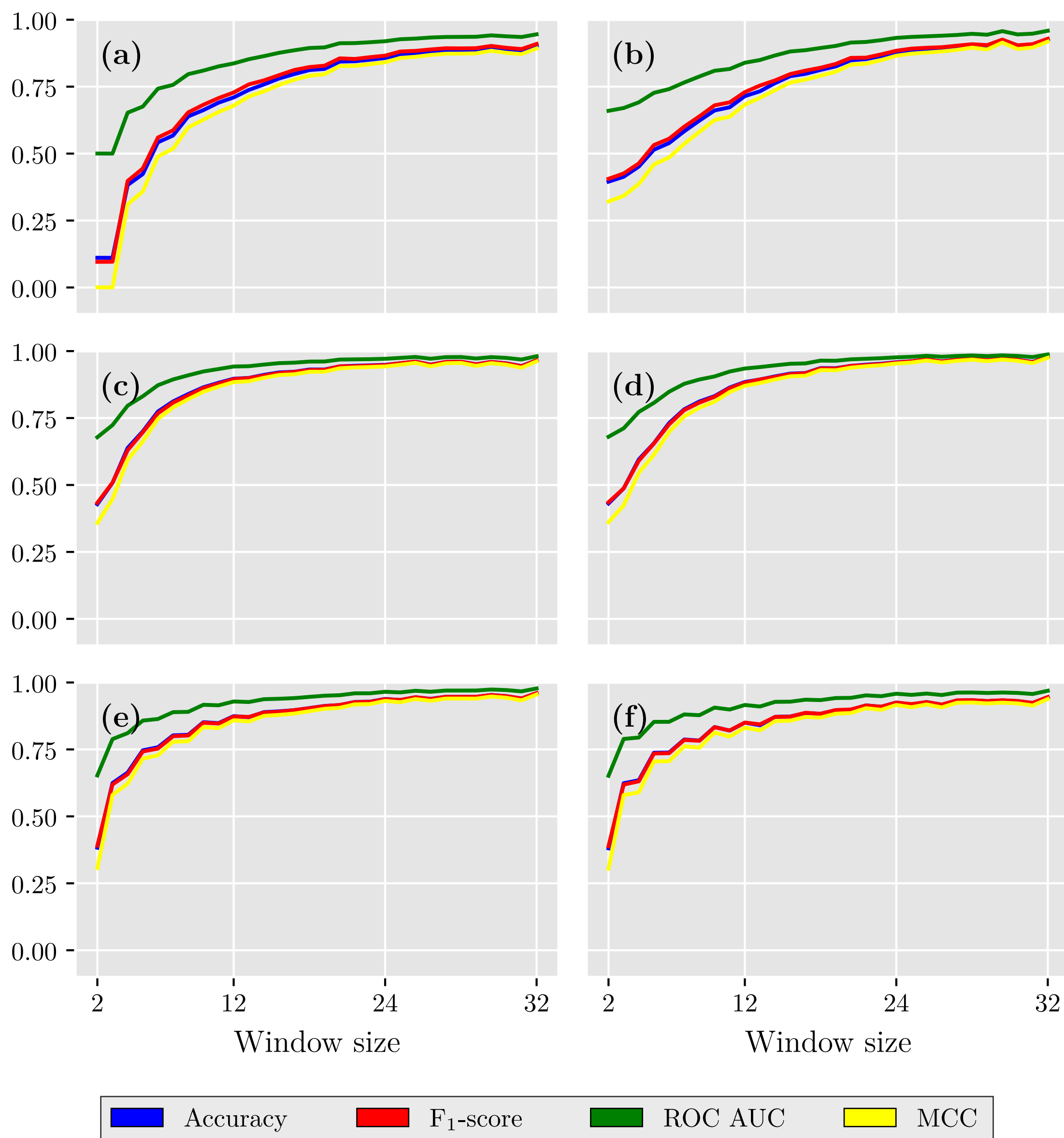


Figure D.25: Performance metrics for the LSTM classifier from Section 3.2.7 on a test set that consists of balanced data set (i.e., each class represents ≈11 % of the data set) for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

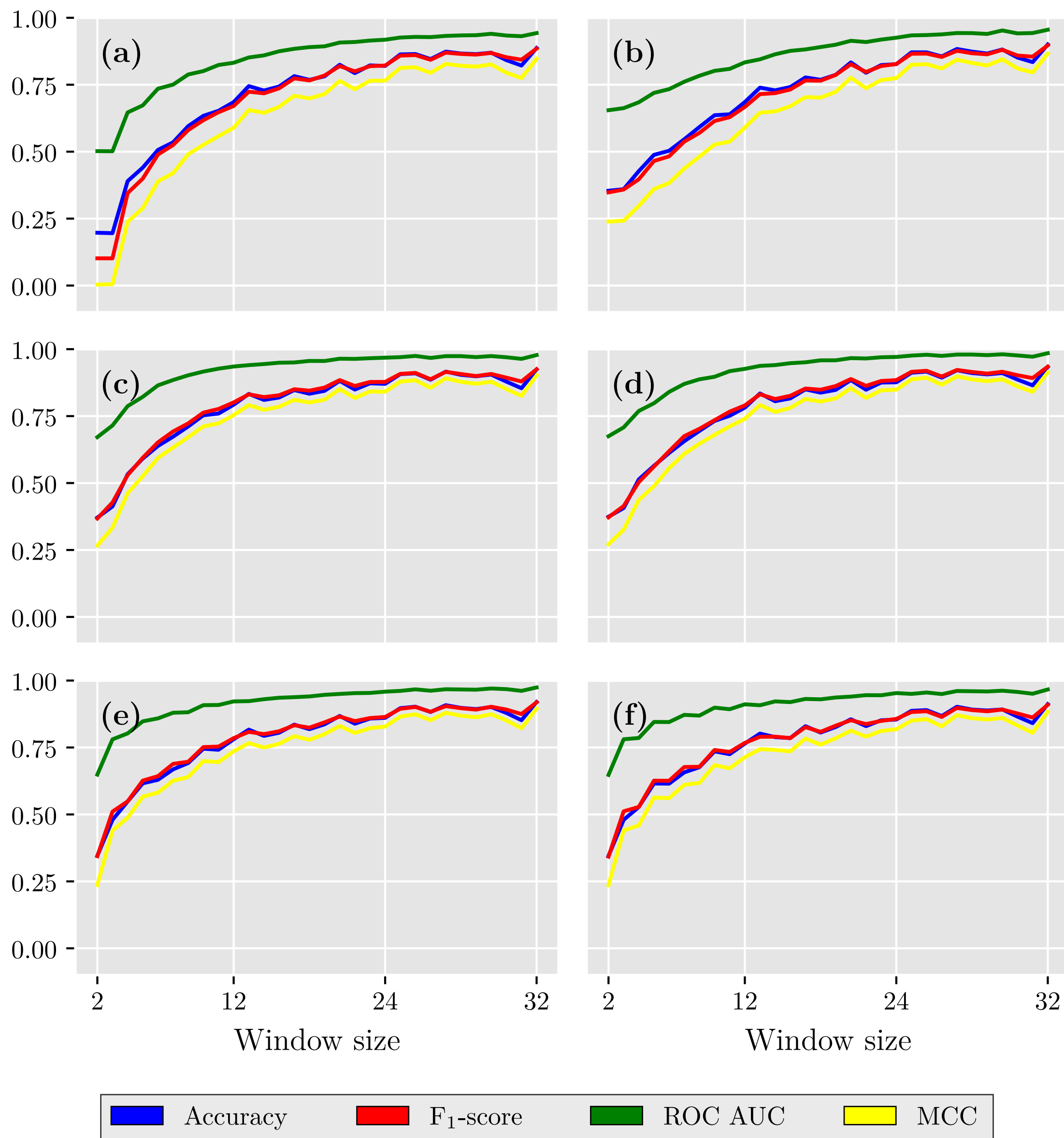


Figure D.26: Performance metrics for the LSTM classifier from Section 3.2.7 on a test set that consists 50 % of healthy and 50 % faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

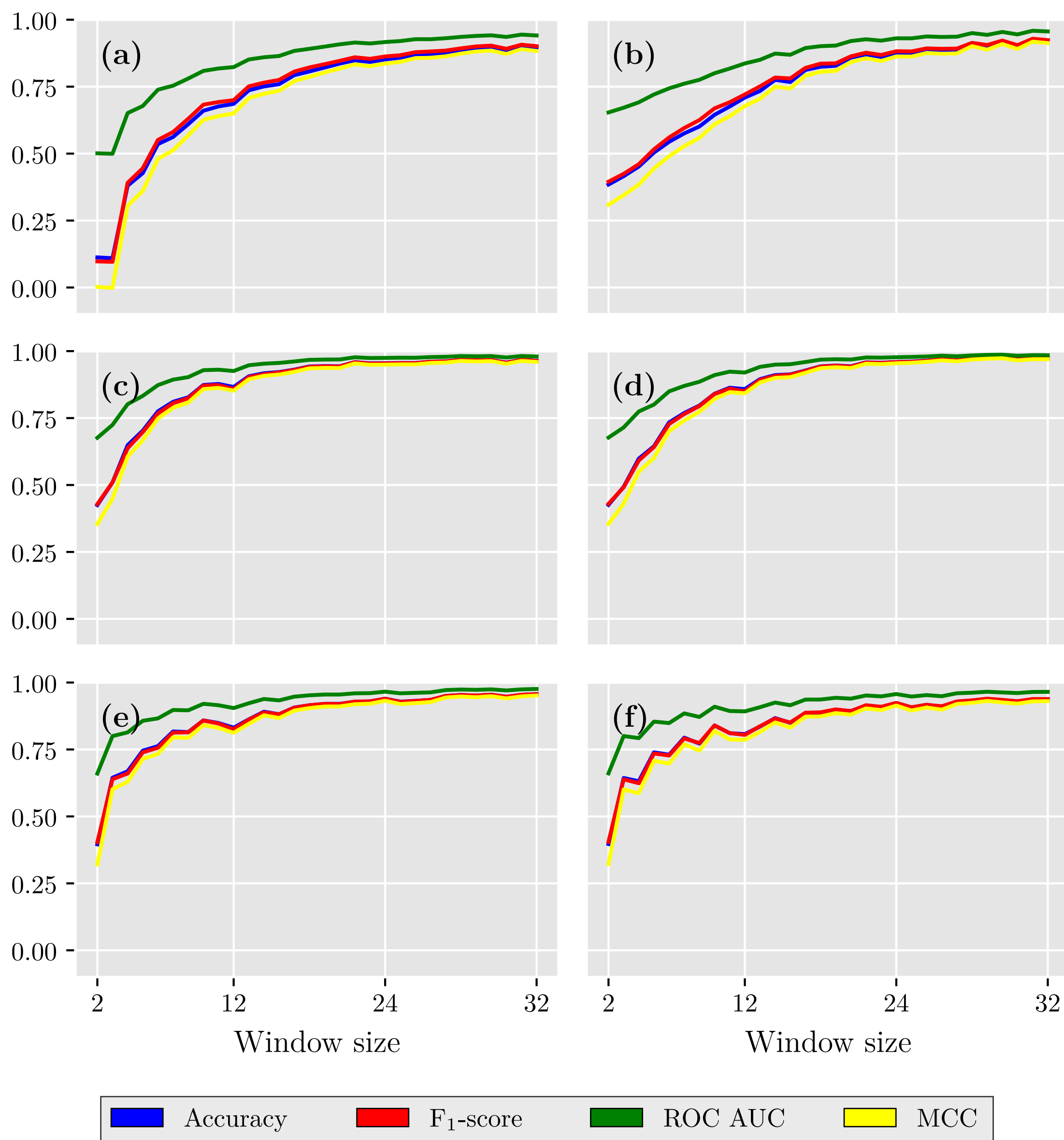


Figure D.27: Performance metrics for the GRU classifier from Section 3.2.8 on a test set that consists of balanced data set (i.e., each class represents ≈11 % of the data set) for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

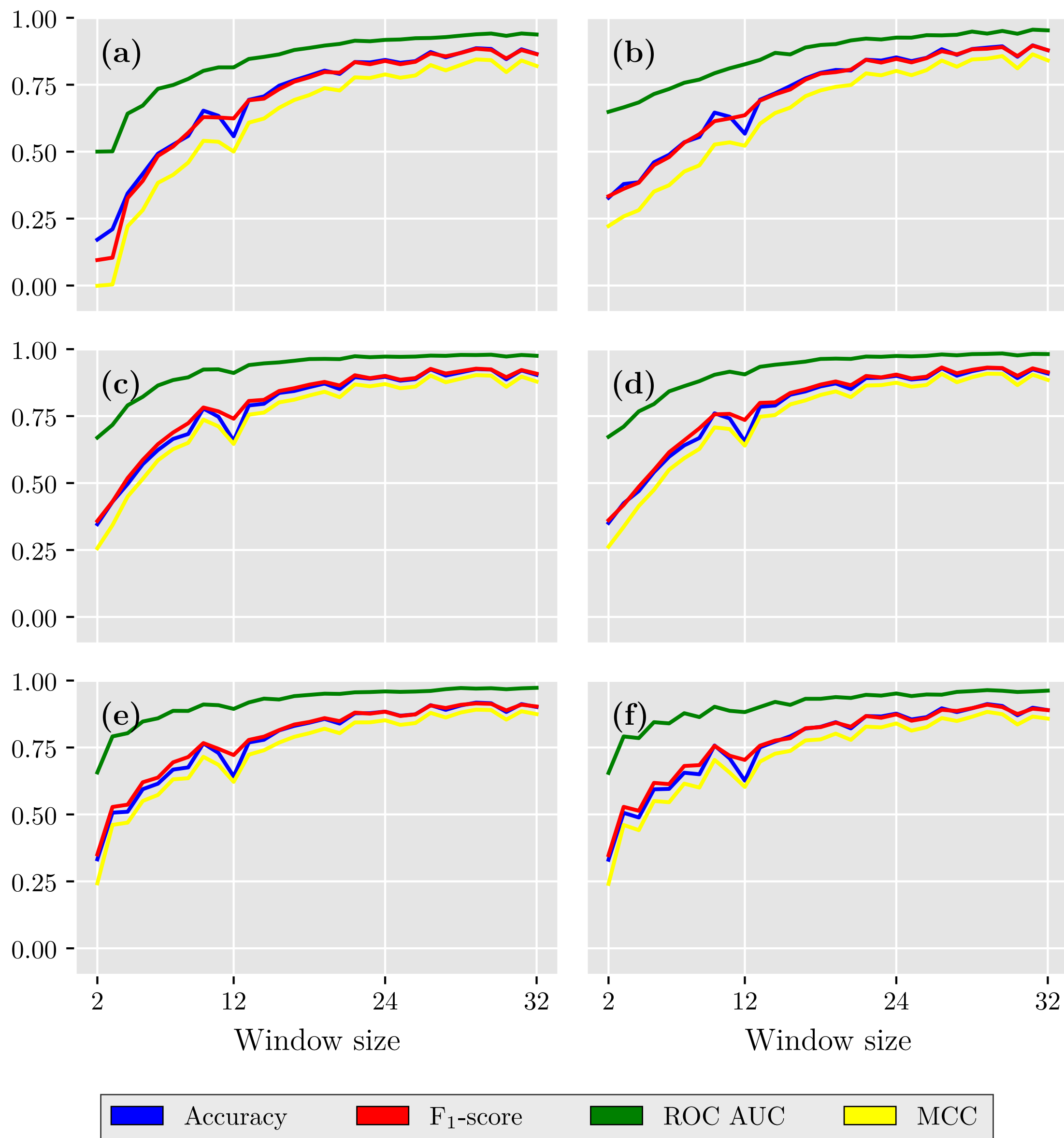


Figure D.28: Performance metrics for the GRU classifier from Section 3.2.8 on a test set that consists 50 % of healthy and 50 % faulty data for each of the faults. Note that the test set only consists of one type of the faults mentioned in Section 2.2 in each subfigure. The six types of faults are **a)** hardover, **b)** drift, **c)** spike, **d)** erratic, **e)** stuck and **f)** frozen. Figure created with [34].

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY