TU WIEN Informatics

A Time-Series Classification Approach to Error Prediction in Hybrid Fiber-Coaxial Networks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Mag.phil. Thassilo Gadermaier, BSc
Matrikelnummer 00507477

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Dr.techn. Peter Filzmoser
Mitwirkung: Dipl.Ing. Dr. techn. Georg Heiler, BSc

Wien, 14. März 2023

Thassilo Gadermaier	Peter Filzmoser

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

A Time-Series Classification Approach to Error Prediction in Hybrid Fiber-Coaxial Networks

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Mag.phil. Thassilo Gadermaier, BSc
Registration Number 00507477

to the Faculty of Informatics
at the TU Wien

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Peter Filzmoser
Assistance: Dipl.Ing. Dr. techn. Georg Heiler, BSc

Vienna, 14th March, 2023

Thassilo Gadermaier | Peter Filzmoser

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Mag.phil. Thassilo Gadermaier, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14.03.2023

Thassilo Gadermaier

Kurzfassung

Die vorliegende Arbeit behandelt die Vorhersagbarkeit gewisser Fehler in Hybrid Fiber-Coaxial (HFC)-Datennetzen, welche in weiten Teilen Österreichs für Internetzugänge genutzt werden. Diese Fehler betreffen überwiegend den Kanal von den Modems der Kunden zum Provider. Die für diese Netze genutzten Kabelmodems sammeln für den Betrieb notwendige die Übertragungsqualität betreffende Daten welche beim Provider zusammenlaufen (Telemetrie).

Die Vorhersage von Fehlern soll dem Betreiber des Netzes ermöglichen, Ressourcen für die Wartung von Netzwerkkomponenten besser planen zu können.

Ausgehend von den Telemetriedaten der Modems (multivariate Zeitreihen) soll die zeitliche Entwicklung einer dieser Variablen, die als Proxy für die Übertragungsqualität und Abwesenheit von Fehlern dient, vorhergesagt werden. Als Vereinfachung des Problems wird nicht der eigentliche zeitliche Verlauf selbst, sondern das Überschreiten eines Schwellwertes vorhergesagt, was einer binären Klassifikation entspricht. Dafür werden dem Stand der Technik entsprechende Modelle des maschinellen Lernens für die Klassifikation von multivariaten Zeitreihen auf die Daten angewandt. Dabei handelt es sich um zwei auf Zeitreihen spezialisierte tiefe künstliche neuronale Netze und ein Modell welches zufällig generierte Faltungskerne zur Ableitung von Merkmalen aus den Daten mit klassischen Klassifizierungsmodellen kombiniert.

Ein wichtiger Aspekt ist das in den Daten stark vorherrschende Ungleichgewicht zwischen den beiden Klassen, der Abwesenheit und Anwesenheit der Fehlerfälle, welches sich stark negativ auf die Leistungsfähigkeit von Klassifizierungsmodellen auswirken kann. Daher werden Methoden zur Behandlung des Ungleichgewichts zunächst theoretisch beleuchtet und dann in praktischen Experimenten angewandt.

Eine Serie von Experimenten überprüft die Eignung der genannten Modelle für die vorliegende Klassifizierungsaufgabe, unter Anwendung weitreichender automatischer Anpassung von Hyper-Parametern sowie Methoden der Regularisierung. Es zeigt sich eine prinzipielle Eignung der Modelle für die Vorhersage von Fehlerfällen, wobei eine weitere Steigerung des positiven Vorhersagewertes (Precision) wünschenswert wäre. Für eine abschließende Beurteilung, welche im Rahmen dieser Arbeit nicht durchgeführt wird, ist die Kenntnis von Kosten für die Fehlerfälle notwendig.

Abstract

This thesis investigates the suitability of machine learning methods for predicting specific types of errors in Hybrid Fiber-Coaxial (HFC)-networks that are used throughout Austria for access to the internet. These errors occur mostly in the upstream path, from customer's modems towards the provider. The modems gather data related to the transmission quality required for the network operation, these are collected further up the network.

Predicting errors aims at enabling the network operator to schedule resources for the (future) maintenance of network components.

Among the telemetry data collected from the modems is one variable that serves as proxy for the network transmission quality. The temporal evolution, especially the exceeding of a threshold, of the proxy variable is of particular interest. The problem is thus formulated as a binary classification problem, where exceeding the threshold is the target event to be predicted. To this end, state of the art machine learning models for the classification of multivariate time-series are employed. In particular, two deep-learning models specialized to the time-series classification task are applied, in addition to a model that generates random convolution kernels for generating features from the data, combined with classical classifiers.

An important property of the data is the strong imbalance between the two classes that are the presence or absence of errors. Strong class imbalance can lead to catastrophically bad performance of classifier models. Thus, methods for treating class imbalance are discussed and applied during experiments.

A series of experiments investigates the suitability of the mentioned models for the given binary classification task, employing extensive automatic tuning of hyper-parameters as well as methods for regularization. The experiments reveal that the models are in general suitable for the task of predicting error cases, but a further improvement of precision is desirable. A final evaluation, requiring concrete cost factors for the error cases, is beyond the scope of this thesis.

Contents

CHAPTER 1

Introduction

Hybrid Fiber-Coaxial (HFC) networks are part of Wide-Area telecommunication Networks (WAN) throughout the world. They were originally designed for TV broadcast service ("Cable TV"), i.e., a one-way service from content deliverer to the customer. Later, these networks were adapted to be additionally used for internet access, that is, as two-way connections. The Data Over Cable Service Interface Specification (DOCSIS) standards are the most widely used for such networks, at least in Europe and the USA. There are different generations of the DOCSIS standard available, some of their aspects will very briefly be discussed where necessary.

As the name suggests, Hybrid Fiber-Coaxial (HFC) networks consist of a part built from fiber-optic components ("glass fibers" and suitable amplifiers, as required) and a part built from electrical components, specifically using coaxial cables, and again suitable amplifiers. More details about the network structure, to a level useful for this thesis, are given later in Chapter 2.

1.1. Scenario and Problem Statement

Network links can fail from time to time, for various possible reasons, at several different nodes present in the network. Naturally, high availability of connections is desired by customers. In case of hardware problems, swift detection of faulty network components by the network operator is required, and subsequently servicing or even replacing them as quickly as possible to avoid major service disruptions. Interruptions may range from "bad performance" from a subscriber's perspective, to total outages.

Hardware (network equipment) failures typically forces dispatched service technicians to manually check several devices on-site until the offending device

is (or devices are) found. Only then remedying actions can be taken, such as replacing a broken link cable, a faulty amplifier, or other components. The devices to be checked may be spread out over several different locations, possibly increasing travel times considerably.

In existing networks that have monitoring systems in place, it is to a certain extent possible to narrow done which components might be at fault when errors occur, thus decreasing the time spent for checking components by field personnel. Identifying regions experiencing bad connection quality, and preferably narrowing down regions so much as to be able to identify broken devices remotely from logging data, is a big problem in its own right. This is not covered in this thesis. Instead, the approach taken here is trying to predict when a certain region will exhibit bad performance in a near future, so as to allocate resources (technicians, test equipment) towards checking and maintenance of network components before the problem actually occurs or becomes more severe. Basing decisions about how to handle future events on given or collected information, that is encoded in data in one way or another, is often called predictive modeling, see e.g. Kuhn et al. (2013, pp. 1 f).

In essence, the goal for the scenario here is to predict whether a target variable that serves as a proxy for connection quality (quality of service / fidelity), will cross a certain threshold within a certain amount of hours into the future. This can in principle be achieved in several ways. Two possible options are:

1. Predict the target variable's values for a given amount of time into the future and check whether a crossing of the threshold (one or several) occurs. This is essentially a regression task, with predicting values into the future.

2. Predict whether a the target variable will cross the threshold (at least once) within a given time frame into the future. This is a classification task, here with binary classes (threshold exceeded: yes or no).

The second option can be viewed as a simplified version of the first: the prediction of the actual values into the future is not of interest, but only whether the target variable will cross the threshold or not, which should simplify the problem. The setup in option 2) above allows for the use of supervised classification methods. Lakshmanan et al. (2020, pp. 80 ff) refer to this re-formulation of a problem as re-framing and discuss reasons for applying this method.

The problem of predicting bad performance of (a segment of) the network is thus framed as a binary Time-Series Classification (TSC) task. Classification here amounts to assigning to a given unlabeled time-series a class label from a fixed, finite set of class labels (Esling et al. 2012, p. 8). Instead of attaching only a class label to a given unlabeled time-series, it is also possible to obtain a prediction of the probability distribution over the possible classes (Ruiz et al. 2021, p. 402). These different approaches and possible benefits are discussed in Chapter 4.

In general, unsupervised machine learning methods may be suitable as well to solve the problem addressed in this thesis. In the context of anomaly detection, a somewhat related problem where typically few instances of the class(es) of interest are available, Aggarwal (2017, p. 26) state that whenever labels are available, it is better to use supervised instead of unsupervised methods. This serves as an additional motivation to approach the problem as a supervised classification task.

1.2. Decisions and Costs

As always when implementing a system, or applying a method to a problem, the question of "how good is it doing?" arises. At a low level, the problem addressed in this thesis is framed as a binary decision (classification) problem: will the network exhibit bad performance (as defined in some way) or not? A confusion matrix is often used to evaluate the performance of binary classifiers (see e.g. Kuhn et al. (2013, p. 254), He et al. (2013, p. 191)). Table 1.2.1 shows an example.

Table 1.2.1.: Confusion matrix for the binary decision problem. See text for details.

Predicted	Actual	
	Event	Non-Event
Event	TP	FP
Non-Event	FN	TN

The terms occurring in such a table are briefly described now. The two possible classes are:

- Event: refers to the occurrence of an incident of bad performance in the network in a certain time frame. While the occurrence of the Event class is a bad thing (bad network performance), it is the class of interest here, often called the *positive* class.
- Non-Event means the absence of such an incident in a certain time frame.

When an instance of class Event occurs, the classifier will predict it to be either of the class Event or Non-Event, and the same holds for the other class, Non-Event. There are thus four different cases to be distinguished, as shown in the table:

- True Positive (TP): True class is Event (the positive class), and it was correctly predicted
- False Positive (FP): True class was Non-Event, but Event was predicted

- False Negative (FN): True class was Event, but Non-Event was predicted
- True Negative (TN): True class Non-Event, correctly predicted

These quantities can be computed by evaluating the (previously trained) classifier on any given data set and recording for each example its true class and the predicted class. A perfect classifier would have zero FP and zero FN cases.

Two more metrics for evaluating the performance of a model, related to the ones just explained above, are Precision and Recall:

- Precision: ratio of the number of TP to the number of instances predicted as class "Event". High precision means that most instances predicted as class "Event" are indeed from this class and not the alternative. For our case this would mean that most predicted cases of bad performance are indeed bad performance in the network that need further investigation. Low precision means that a too large amount of cases, more than truly occurring, are predicted as upcoming bad performance.
- Recall: ratio of the number of TP to the number of instances that are truly of the class "Event". High recall means that most of the instances of the class "Event" are detected. For our case this would mean that most occurrences of bad performance are correctly detected (in advance). Low recall means that too many upcoming cases of bad performance are not detected, delaying allocation of resources for mitigating the problem.

In the context of the scenario described above, the occurrence of an event would trigger the allocation of resources to mitigate the impact of bad network performance. This could e.g. be the closer monitoring of certain sections of the network, either remotely or on-premise by technicians, and search for and replacement of broken components. Knowing about occurring problems in advance can lead to better planning and allocation of resources.

There are certain costs C_{m} (m as in maintenance) associated with the actions to mitigate performance problems. A falsely predicted occurrence of the Event class would thus cost C_{m}, without gain.

Not detecting bad performance would mean delayed allocation of resources and possibly longer periods of bad performance until mitigating actions can be taken. There are costs associated with this case as well, say C_{uc} (uc as in unhappy customer), for e.g. increased call volume to customer service, up to losing customers in the worst case. A falsely predicted occurrence of the Non-Event class would thus cost C_{uc}.

While it may not be possible to precisely estimate such costs properly, it can still be useful to incorporate them into decision processes facilitated by a classifier. Assigning different costs can trade-off between the different error cases.

Table 1.2.2 shows a cost matrix for the binary decision problem in a general form. Similar to the four cases in the confusion matrix, there are four cases here.

Table 1.2.2.: Cost matrix for the binary decision problem.

Predicted	Actual	
	Event	Non-Event
Event	C_{11}	C_{10}
Non-Event	C_{01}	C_{00}

Note that the costs C_{11} and C_{00} for correct classifications can be absorbed in the costs of misclassification C_{01} and C_{10}, or assumed to be zero (Fernández et al. 2018, p. 65).

Two alternative versions of cost matrices are shown in Table 1.2.3. The costs for correct classifications are set to zero, the other cost factors are given meaningful names for different contexts.

Table 1.2.3.: Cost matrices for the binary decision problem with zero costs for correct classifications. Left: the cost factors are named after the associated error cases. Right: the cost factors are named after the cases described in the maintenance scenario described in the text.

Predicted	Actual	
	Event	Non-Event
Event	0	C_{FP}
Non-Event	C_{FN}	0

Predicted	Actual	
	Event	Non-Event
Event	0	C_{m}
Non-Event	C_{uc}	0

A benefit of the possibility to apply costs for the different actions is that they can not only be applied during the modeling process, but also post-modeling, i.e. after training a classifier on labeled data (supervised machine learning setting). This means that a trained model can still be used in an environment with changing conditions (and costs), and can be adapted by applying changes to the costs to shift decisions in one direction or the other. Similarly, if costs were applied during training, but were found afterwards not to be correctly estimated, or not reflecting the true costs properly, different costs can be applied to improve the decisions. As the left matrix in Table 1.2.3 shows, the cost factors can tune between the error cases False Positive (FP) and False Negative (FN).

1.3. Challenges

From a machine learning perspective, several challenges can be stated that need to be addressed to solve the problem at hand:

- Highly imbalanced classes. There are many more data points available for error-free cases than for error-cases. This simply means that the network is

working properly most of the time. In general, special care must be taken in case of imbalanced classes, and even more so in the somewhat extreme imbalance found in the data analyzed in this thesis. The imbalance impacts the choice of models, how to train them in the first place, and secondly how to evaluate their performance. The imbalance present in the data is investigated in Sections 3.5 and 5.8, and the impacts on models and possible approaches to mitigate them are discussed in Chapter 4.

- Large amount of data. While the dimensionality itself is not very high, the data cover a very large time-span, a large spatial area, and thus there are very large amounts of data points. Chapter 3 gives a detailed overview of the data and its extent. If lots of data is available, undersampling, i.e. removing parts of the data, can be a way to mitigate this challenge. Chapter 4 discusses a few points of re-sampling data.

- Heterogeneity of the data that results most likely from the heterogeneity of network components, as well as the different number of users per network segment, and possibly other unknown factors. The heterogeneity is briefly looked at in Sections 3.1 and 3.4.

1.4. Thesis Organization

This thesis is organized as follows: Chapter 2 gives a short overview of HFC-networks, their structure, and typical errors occurring in such a network.

Chapter 3 discusses the data used in this thesis in-depth. An overview is given on how the spatially aggregated data used for the experiments were generated and how they are structured. The extent of the data is shown, missing values discussed, class distributions and statistical properties are investigated.

Chapter 4 describes the problems associated with class imbalance in the data and reviews solutions suggested in the literature. Problems due to class imbalance occur at the level of the data, the models used, and how the performance may be evaluated. Possible methods applying leverage at each of this levels are discussed.

Next, the preprocessing of the data as necessary for the experiments is treated in Chapter 5. Besides typical preprocessing, such as standardization (z-scoring), the processing of the data as necessary for applying Time-Series Classification (TSC) methods is described. This amounts to creating short segments and assigning class labels, thus creating a data set for the task.

Suitable models for the TSC task under the condition of strong class imbalance are discussed in Chapter 6. Basic structure of the models and some aspects of training them are presented. Strategies for model selection and tuning hyper-parameters are addressed as well.

Finally, the experiments conducted are documented in Chapter 7. The most important parameters tested during the experiments are discussed, as are the hyper-parameters used by each model.

The results of the experiments are then summarized, visualized and discussed in Chapter 8. Insights from the model selection are discussed and an exemplary further, more in-depth evaluation of a specific model is shown. Furthermore, some outlook for potential future work is given.

Additional plots and insights across a larger range of settings and models are presented in Appendix A.

CHAPTER 2

Hybrid Fiber-Coaxial (HFC)-Networks

This thesis deals with data obtained at nodes of a Hybrid Fiber-Coaxial (HFC)-network. This network is owned by an Internet Service Provider (ISP) that will be referred to as AnonISP throughout this thesis. Since the structure of the network imposes a certain structure on the data, an is thus important for understanding the data, we shall take a brief look at the network and some of its components. As the name suggests, an HFC-network is a hybrid of two technologies: optical and electrical telecommunication components, where fiber-optics and coaxial cables serve as bearers of signals.

- Fiber optic cabling: transmission is carried out with electromagnetic waves in the optical frequency range.
- Coaxial cabling: transmission is carried out with electromagnetic waves below the optical frequency range.

The specification of the fundamental technical components (the physical layer in the OSI-nomenclature) used in an HFC-network is documented in (CableLabs 2017).

2.1. Network Structure

Figure 2.1 shows a schematic overview of an HFC network. In general, such a network exhibits a tree structure when considering a hub as the root node. At the end, the leaf nodes are customers' Cable Modems (CM). Cables reaching CMs are branching off at splitter devices. Occasional line amplifiers may be

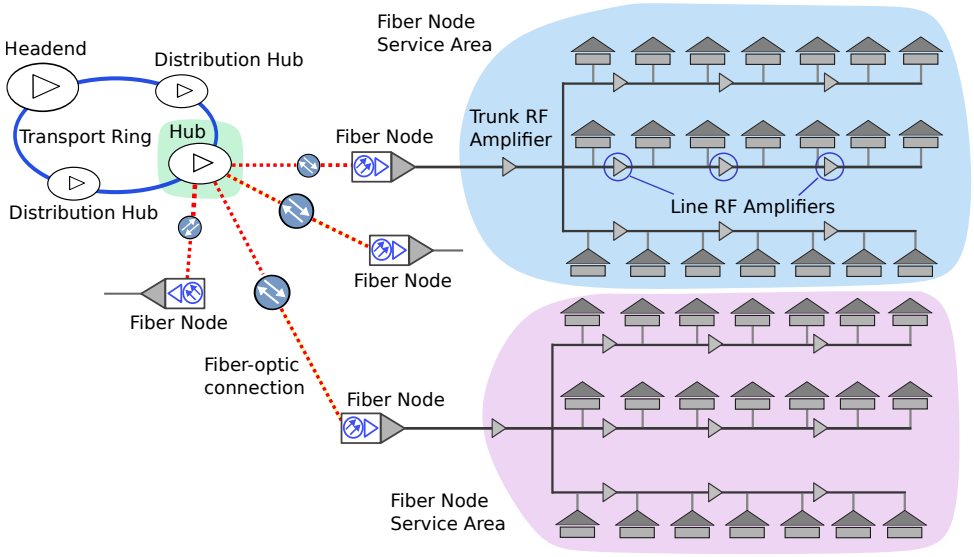

Figure 2.1.: Structure of an HFC network. Each fiber node serves a certain area and thus number of customers (neighborhood). See text for details. Figure adapted from (Unforgettable_fan et al. 2009).

installed within the cable network. The trunk lines serving a neighborhood have a trunk amplifier at their head. Each trunk's electrical cable is finally connected to a fiber node, converting between electrical and optical signals. Several fiber nodes are connected to a hub, containing a Cable Modem Termination System (CMTS). Hubs (and thus CMTS) are connected to the ISP's backbone network. Each fiber node serves a certain region or area, and thus a certain neighborhood of customers. Such a region is often referred to as fiber node area (FNA). Fiber node areas can be considered separate entities for the purposes of this thesis, and this has consequences for handling the data, as is discussed later in Chapter 3 below.

Customers may leave and enter the network through time, terminating or entering contracts with the ISP. Thus, FNAs are not entirely fixed over time. However, for the purposes of this thesis, they will be considered static over time. This assumption is strengthened by aggregating over entities in an FNA, see also Section 3.1 below for further details.

To give an impression of the size of the network, Figure 2.2 shows a pixel map representing the hubs and fiber node areas (FNAs) of AnonISP's network.

2.2. Transmission

Internet connections require two-way communication, preferably simultaneously for both directions. This is usually termed full-duplex. There is thus a channel (in an abstract sense) from customer to service provider, called upstream (US) in HFC-terminology. The channel in the reverse direction, from provider to

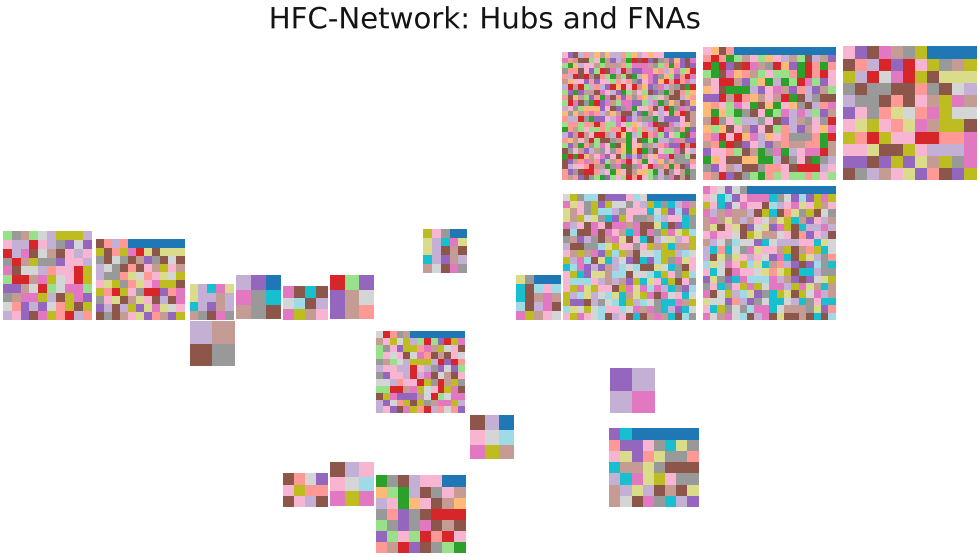

Figure 2.2.: Schematic map of the HFC-network. Each colored pixel represents one fiber node area (FNA), containing potentially hundreds of customers. Each block of pixels represents one hub. Placement of hubs very coarsely follows their spatial locations in AnonISP's country.

customer, is called downstream (DS).

HFC-networks typically use the DOCSIS standards that exist in several generations. Currently, DOCSIS 3.0 and 3.1 are the most recent versions deployed in existing networks. These specify all kinds of parameters of the transmission systems. The upstream and downstream signals are allocated to dedicated frequency bands to allow full-duplex operation. This method of separating the two directions is usually called Frequeny-Domain Duplex (FDD). Typical frequency bands for upstream and downstream channels are given in (CableLabs 2017, p. 29) and are shown in Table 2.2.1 below. Different regions of the world might employ slightly different frequency ranges used for upstream and downstream (Large et al. 2009, p. 217).

Table 2.2.1.: Typical frequency ranges for upstream and downstream directions in HFC-networks (CableLabs 2017, p. 29).

Frequency ranges in MHz	
Upstream	Downstream
5 - 30	54 - (300 to 1002)
5 - 42	54 - (300 to 1002)
5 - 85	108 - (300 to 1002)

Further protocols are employed to define how a single modem (CM) can access

the shared medium. More details are beyond the scope of this thesis and will not be given here.

2.3. Upstream Errors

Transmission of electromagnetic waves in optical fibers, or along electrical cables, in general suffers from different types of noise and interference, to varying degrees. In HFC networks in particular, the electrical part of the network (closer to the leaf nodes, the Cable Modem (CM)), and here the upstream direction, typically is the most cumbersome part. Note that "upstream direction" here refers to the signals traveling from the leaf nodes upwards the network on the electrical cabling that is also used for the downstream signals; as was stated above, these occupy different frequency bands but use the same physical cable.

Noise on the upstream (or return) path in Hybrid Fiber-Coaxial (HFC) networks needs dedicated treatment e.g. on the level of the network components, see for example Chapter 8 in (Large et al. 2009).

One example of noise in the upstream path is Common Path Distortion (CPD). It originates from inter-modulation of downstream and upstream signals, most often caused by corroded contact surfaces of connectors that then act as nonlinear transmission elements due to the corrosion (Large et al. 2009, pp. 249 f). The inter-modulation products generated can occur as comb-like spectrum in both the upstream and the downstream frequency bands, disturbing the actual signal occupying the same regions. The grid-spacing of the combs' components follows the channel-spacing of 6 MHz, 7 MHz or 8 MHz (the grid-spacing depends on the specification typical to the part of the world the system is used in) (Keller 2011, p. 282), (Large et al. 2009, p. 249).

Too strong noise can impact transmission signals such that transmission symbols are detected incorrectly, thus leading to errors in the transmitted data. The Signal-to-Noise Ratio (SNR) is usually used to indicate the state of the transmission (among other figures of merit), and sufficiently high SNR is required for (effectively) error-free transmission. Typically, several thousand transmission symbols together form a transmission codeword. Too many errors in a codeword can lead to seemingly slower connection speeds (because of frequent re-transmissions of erroneous messages) or bad quality of media streams (video, audio content) due to too much corrupted data. The Codword Error Rate (CER) can be used to measure the performance of the transmission at the codeword level.

Since disturbances originating in the upstream path can influence the network performance strongly, it is important to monitor the corresponding signals appropriately. This is the reason why the upstream signals are given a greater focus in this thesis as well, especially when their behavior is used as proxy for the current overall network state and transmission quality, see Section 3.2 and Sections 5.4 and 5.5.

CHAPTER 3

Data

This chapter gives an overview of the data used in this thesis. From the structure of the network briefly discussed above follows a certain structure of the data.

Each cable modem (CM) in the network described above records certain data related to the transmission of signals in the network, and these are fed back to nodes higher up in the network, where they are accessible to the network operator AnonISP.

The derivation of the data used in this thesis, multivariate time series (MTS) sampled on an hourly basis, from the "raw" data recorded from the modems, is described in the following.

The structure of the network shown in Figure 2.1 is depicted again in Figure 3.1, but in a more abstracted fashion more relevant to the understanding of the structure of the data. The fiber node areas (FNA), a collection of modems served from one single fiber node, are depicted as single blob in this figure. The data recorded from all modems within an FNA are (spatially) aggregated to reduce the amount of data to process. The aggregation on the basis of fiber nodes is not just an arbitrary choice, but informed from the functioning of the network: a typical error case in an HFC-network is common path distortion (CPD) that typically affects many or all modems connected to a fiber-node. Typically, more and more modems are affected the longer the error persists.

The next section describes the process of spatially aggregating the per-modem data to per-FNA data.

3.1. Raw and Aggregated Data

This section is meant to describe the data in a more formal way.

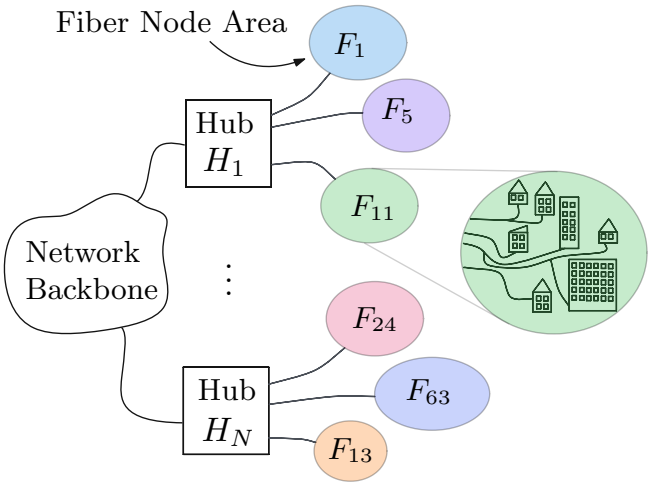

Figure 3.1.: Schematic network hierarchy: Each hub is associated with several fiber node areas. Each fiber node area (FNA) represents data of several (up to thousands of) customers. This Figure should be compared to Figure 2.1.

3.1.1. Notation

The data used in this thesis are discrete-time time series. It is common to denote these as sequences $\{x\}_t$, here with index $t \in \mathbb{Z}$. The data used here are of finite duration: $\{x\}_t = x_0, x_1, \ldots, x_{N-1}$. While x_t is used to denote a single value for fixed t, say x_5 at $t = 5$, this will also be used as a shorter notation for the whole time series, as is common in the literature. It should be clear from the context what is denoted by x_t in each case. Figure 3.2 shows the interpretation of the data as a signal, with a horizontal orientation along a time-axis, and alternatively as a column vector. Both ways to view the data will be used throughout this thesis. While the data are discrete-time, they will be depicted using continuous line-plots in some places.

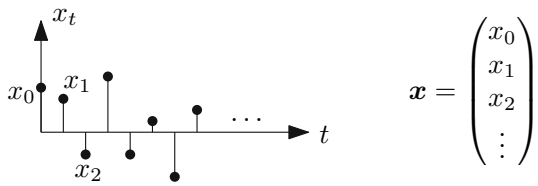

Figure 3.2.: Discrete-time ($t \in \mathbb{Z}$) time series depicted as signal (left) and as column vector (right).

Throughout this thesis, vectors are columns when notated using bold serif typeface:

$$\boldsymbol{x} = (x_0, x_1, \ldots, x_{N-1})^\mathsf{T}. \tag{3.1}$$

When dealing with multivariate time series (MTS), a single variable (univariate

time series) will be denoted with a superscript index:

$$\boldsymbol{x}^{(1)} = \left(x_0^{(1)}, x_1^{(1)}, \ldots, x_{N-1}^{(1)}\right)^\mathsf{T}. \tag{3.2}$$

A multivariate time series can be written as a matrix $\boldsymbol{X}$:

$$\boldsymbol{X} = \begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \ldots & x_0^{(P)} \\ x_1^{(1)} & x_1^{(2)} & \ldots & x_1^{(P)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N-1}^{(1)} & x_{N-1}^{(2)} & \ldots & x_{N-1}^{(P)} \end{pmatrix} = \begin{pmatrix} \mathsf{x}_0 \\ \mathsf{x}_1 \\ \vdots \\ \mathsf{x}_{N-1} \end{pmatrix} = \left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(P)}\right). \tag{3.3}$$

Note that the "time-axis" is in the vertical dimension, along the rows of the matrix $\boldsymbol{X}$.

Since we have P many variables, we can look at a multivariate (row) vector at any given point in time t (as shown in the equation above):

$$\mathsf{x}_t = \left(x_t^{(1)}, x_t^{(2)}, \ldots, x_t^{(P)}\right). \tag{3.4}$$

Row vectors are written using bold upright sans-serif typeface throughout this thesis.

The data matrix $\boldsymbol{X}$ thus represents a multivariate (dimension P) time series:

$$\boldsymbol{X} = \left(\mathsf{x}_0, \mathsf{x}_1, \ldots, \mathsf{x}_{N-1}\right)^\mathsf{T}, \tag{3.5}$$

as above.

3.1.2. Raw and Aggregated Data

This section gives a brief overview of the gestation process of the data that are used in this thesis. The aggregation of the raw data described below was not carried out by the author; the data were handed over in this form for processing during the experiments documented in this thesis. Nevertheless, a short description shall be given to get a better idea of the aggregated data and their structure.

Each fiber node area (FNA) can contain up to a few thousand cable modems (CM). To reduce the amount of data to process, aggregated values are computed across all K devices in an FNA, for each FNA separately. The amount of devices K per FNA varies between different FNAs. Thus, more strictly, the number of devices K should be indexed by the respective FNA, e.g. $K^{(F_1)}$ for FNA F_1. To not clutter the notation too much, this will be refrained from mostly, but should be kept in mind, and will be mentioned again where important.

The process of spatial aggregation across all devices in an FNA on a per-timestamp basis is schematically illustrated in Figure 3.3. This figure should be compared to the Figures 2.1 and 3.1 showing the network structure.

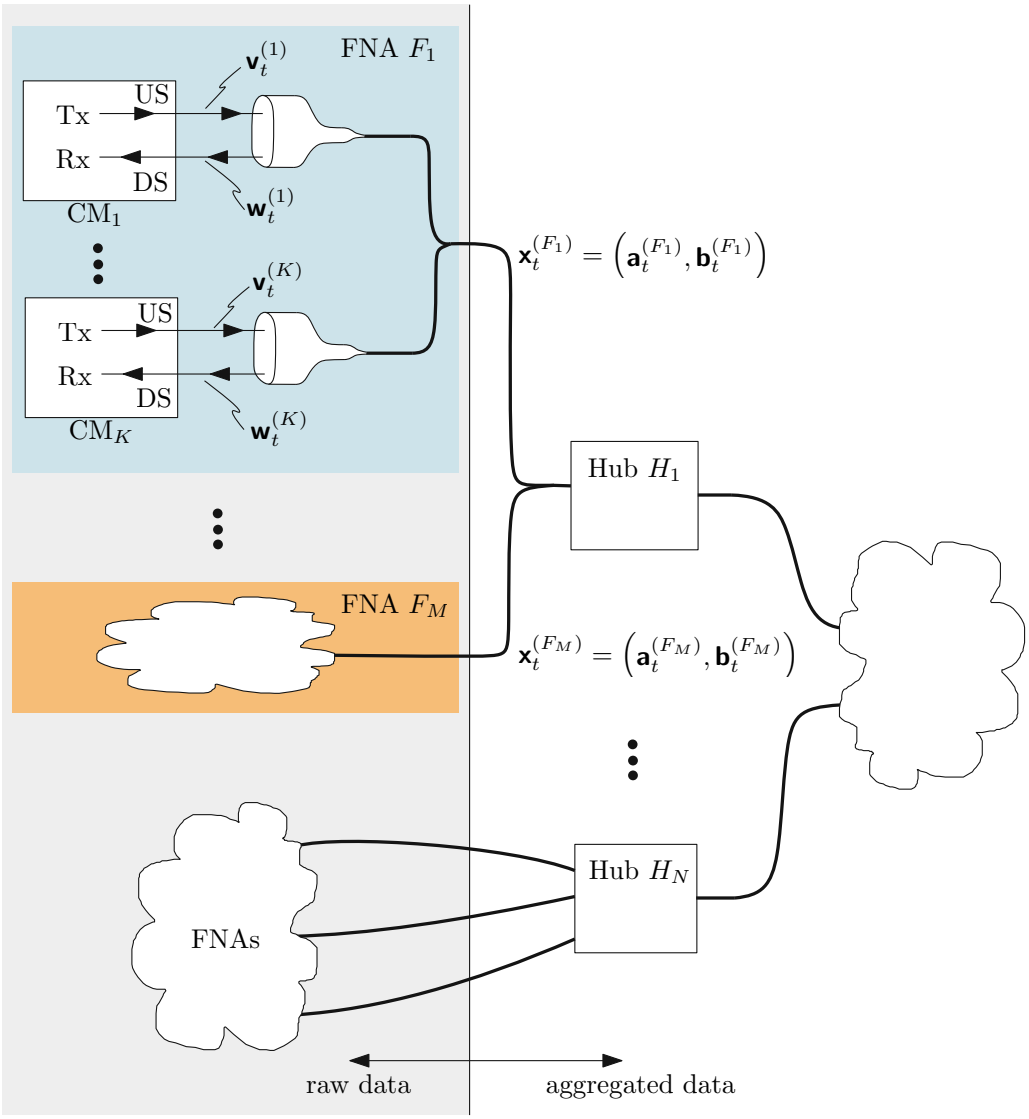

Figure 3.3.: Raw and aggregated data. The raw data are multivariate time series (signals) per cable modem (CM) (Tx: transmitter, Rx: receiver components), separate for upstream (US) and downstream (DS). Several CMs are grouped in each FNA. The aggregated data $\mathbf{x}_t^{(F_i)}$ used in this thesis result from the raw data through spatial aggregation on a per-FNA level.

Table 3.1.1.: Variables and functions involved in spatial aggregation of the raw data. K ... number of Cable Modem (CM) per FNA. See text for details.

	Upstream		Downstream	
Unit	Variable	Dimension	Variable	Dimension
Cable Modem	$\mathbf{v}_t$	$1 \times J$	$\mathbf{w}_t$	$1 \times L$
FNA	$\boldsymbol{V}_t$	$K \times J$	$\boldsymbol{W}_t$	$K \times L$
Single agg. function	$f^{(\text{mean})}$	$K \times J \to 1 \times J$	$g^{(\text{mean})}$	$K \times L \to 1 \times L$
Agg. function	$\boldsymbol{f}$	$K \times J \to 1 \times DJ$	$\boldsymbol{g}$	$K \times L \to 1 \times EL$
FNA, single agg.	$\mathbf{a}_t^{(\text{mean})}$	$1 \times J$	$\mathbf{b}_t^{(\text{mean})}$	$1 \times L$
FNA	$\mathbf{a}_t$	$1 \times DJ$	$\mathbf{b}_t$	$1 \times EL$

Unit	Variable	Dimension
FNA, US & DS	$\mathbf{x}_t = (\mathbf{a}_t, \mathbf{b}_t)$	$1 \times (DJ + EL)$

Table 3.1.1 summarizes the involved variables, functions and their dimensions. The description of the process will in the following focus on the upstream direction, the extension to the downstream is straightforward given the table.

Each modem (CM) logs several quantities at an hourly rate, both for the upstream (US) and the downstream (DS) direction separately. The quantities considered in this thesis are (upstream and downstream, with one exception):

- Signal-to-Noise Ratio (SNR)
- Transmit Signal Power (only US)
- Receive Signal Power
- Codword Error Rate (CER)
- Corrected Codword Error Rate (CCER)
- Micro-reflections

The respective value of each quantity at time t is then, for upstream and downstream, collected in a separate row vector $\mathbf{v}_t$ and $\mathbf{w}_t$, respectively. For the upstream path, per CM, at time t, this is:

$$\mathbf{v}_t = (\text{SNR}_{t,\text{US}}, \text{TxPower}_{t,\text{US}}, \text{CER}_{t,\text{US}}, \ldots), \quad \text{dimension: } 1 \times J, \tag{3.6}$$

and similar for the downstream path. The row vectors $\mathbf{w}_t$ for the downstream are of dimension $1 \times L$. For collecting all vectors $\mathbf{v}_t$ and $\mathbf{w}_t$ for all devices in an FNA, the matrices $\boldsymbol{V}_t$ (upstream) are introduced:

$$\boldsymbol{V}_t = \begin{pmatrix} \mathbf{v}_t^{(1)} \\ \mathbf{v}_t^{(2)} \\ \vdots \\ \mathbf{v}_t^{(K)} \end{pmatrix}, \quad \text{dimension: } K \times J, \tag{3.7}$$

and similarly $\boldsymbol{W}_t$ (dimension $K \times L$) for the downstream.

Note that while J and L (dimensions of $\mathbf{v}_t$ and $\mathbf{w}_t$, respectively) are the same for all modems, the number of devices per FNA K varies from FNA to FNA. This means that aggregations may not be carried out across the same number of devices for different FNAs, respectively.

A simple example for an aggregation function $f_{\text{agg}}^{(d)} : \mathbb{R}^{K \times J} \to \mathbb{R}^{1 \times J}$ is the mean:

$$\mathbf{a}_t^{(\text{mean})} = f_{\text{agg}}^{(\text{mean})}\left(\boldsymbol{V}_t\right) = \frac{1}{K}\, \mathbf{1}_K^{\mathsf{T}} \begin{pmatrix} \mathbf{v}_t^{(1)} \\ \mathbf{v}_t^{(2)} \\ \vdots \\ \mathbf{v}_t^{(K)} \end{pmatrix}, \tag{3.8}$$

where $\mathbf{1}_K$ is a column vector of all 1s of dimension $K \times 1$.

With $d = 1, 2, \ldots, D$ many different aggregation functions like the mean, the standard deviation, or functions like the maximum, minimum, and so on, the component functions $f_{\text{agg}}^{(d)}$ can be composed to an overall aggregation function $\boldsymbol{f}_{\text{agg}} : \mathbb{R}^{K \times J} \to \mathbb{R}^{1 \times DJ}$.

The resulting aggregates are then collected in a single vector $\mathbf{a}_t$ of dimension $1 \times DJ$:

$$\mathbf{a}_t = \left(\mathbf{a}_t^{(\text{mean})}, \mathbf{a}_t^{(\text{stddev})}, \ldots, \mathbf{a}_t^{(D)}\right), \tag{3.9}$$

and similar for the downstream. In short, for the upstream:

$$\mathbf{a}_t = \boldsymbol{f}_{\text{agg}}\left(\boldsymbol{V}_t\right). \tag{3.10}$$

The aggregates of upstream and downstream data are finally stacked horizontally into a single vector:

$$\mathbf{x}_t = \left(\mathbf{a}_t, \mathbf{b}_t\right), \tag{3.11}$$

of dimension $1 \times (DJ + EL)$.

For the experiments conducted in this thesis, only the following spatial aggregations were used (out of several others provided):

- mean,
- standard deviation.

Vectors $\mathbf{x}_t$ as in Equation (3.11), stacked vertically for consecutive timestamps, form the data matrix $\boldsymbol{X}$ (with $P = (DJ + EL)$):

$$\boldsymbol{X} = \begin{pmatrix} x_0^{(1)} & x_0^{(2)} & \ldots & x_0^{(P)} \\ x_1^{(1)} & x_1^{(2)} & \ldots & x_1^{(P)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N-1}^{(1)} & x_{N-1}^{(2)} & \ldots & x_{N-1}^{(P)} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_{N-1} \end{pmatrix} = \left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(P)}\right). \tag{3.12}$$

The aggregated data just described are the outset for further processing and experiments in this thesis.

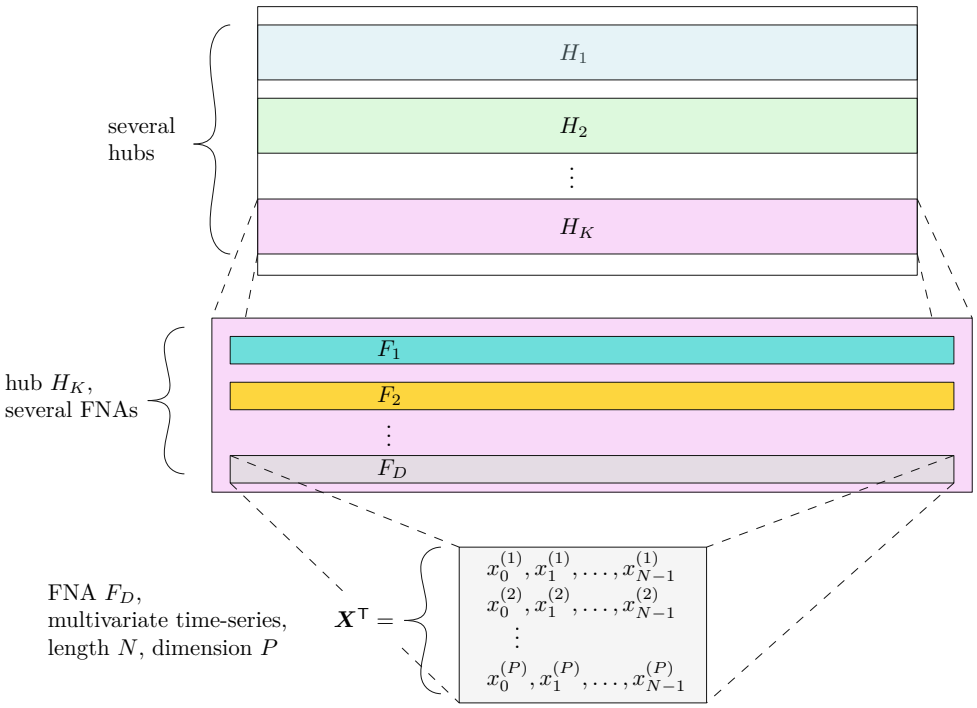

Figure 3.4.: Structure of the data: nested relationship. Each hub is the parent element of several FNAs, and each FNA represents data spatially aggregated across it (compare Figures 3.1 and 3.3), a multivariate time series. Conceptually, each FNA corresponds to a data-matrix $\boldsymbol{X}$.

At this point, the hierarchical network structure as discussed in Section 2 and the spatial aggregations to a multivariate time series per FNA, as described above and shown in Figure 3.3, can be consolidated. From the hierarchical structure of the network, as shown in Figure 3.1, follows a nested structure of the data, as shown in Figure 3.4.

3.2. Target Variable

As established before, the data are multivariate time series (MTS) with component time series $x_t^{(i)}$, $i = 1, 2 \ldots, P$. One of the variables, say $x_t^{(j)}$, is considered the target variable whose evolution is to be predicted for a certain amount of time into the future. As input data for the prediction model, the target variable's past values, as well as the other variables' past values, i.e. the variables $x_t^{(i)}$, with $i = \{1, \ldots, P\} - \{j\}$, can be used. The target variable will be referred to as τ_t throughout in the following.

Figure 3.5 illustrates the situation. The target variable was identified from previous work at AnonISP and is assumed to be a suitable proxy for network state, and ultimately customer experience.

A simplified version of the problem is not concerned with directly predicting the

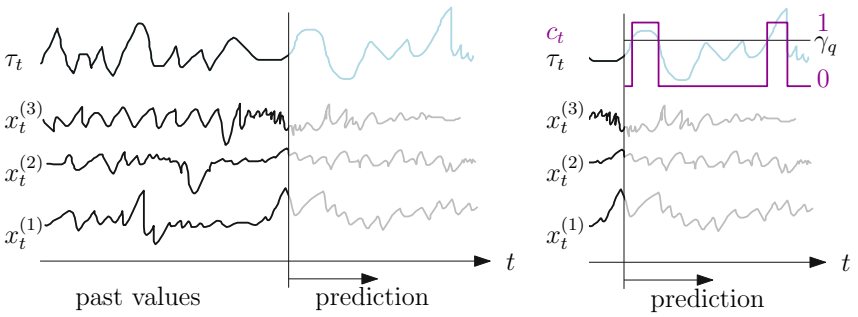

Figure 3.5.: Time series $x_t^{(i)}$ and target variable τ_t. Of interest are future values of the target variable (blue line, top), to be predicted from its own, as well as the other time series', past values. The other variables' future values are not of interest. In a simplified version, not the target signal's future is of interest, but only that of a quantized version $c_t \in \{0, 1\}$, using threshold γ_q.

target variable τ_t, but rather a quantized version $c_t \in \{0, 1\}$. The quantization uses a threshold γ_q. This threshold was identified as well by previous work at AnonISP.

Figure 3.6 shows a plot of the target variable for a single FNA's target variable τ_t as well as the quantized version c_t. Whenever the target variable reaches or crosses the threshold γ_q, the quantized target c_t gets assigned the value 1, which is to be understood as marking the "event" class. This is the first step of assigning class labels for the data, more details are given in Section 5.4 and Section 5.5.

Note that the label 1 is assigned to the class "event", i.e. the class of interest for the scenario treated here. As was briefly mentioned in Section 1.2, the class of interest is called the *positive* class, and it is customary to assign the higher class label (1 vs. 0 for the binary class problem) to the class of interest.[1]

3.3. Data Availability

The data considered have a maximum temporal extent from 21st of January 2021, 23:00, to 28th of June 2021, 21:00, the overall first and last time-stamps t_{start} and t_{end}, respectively. This is a time-span of 157 days and 22 hours, or more than 22 weeks, or more than 5 months. This period is denoted by T in the following. However, not for all hubs and their associated (nested) FNAs, the available data cover this whole extent.

The data for the network consists of 21 hubs with 2220 FNAs in total. The smallest hub serves 4, the largest roughly 600 FNAs. In total, there are 8 148 975 data points available in the aggregated data.

[1] More specifically, for example the well known library scikit-learn (Pedregosa et al. 2011) follows this convention, as do many others.

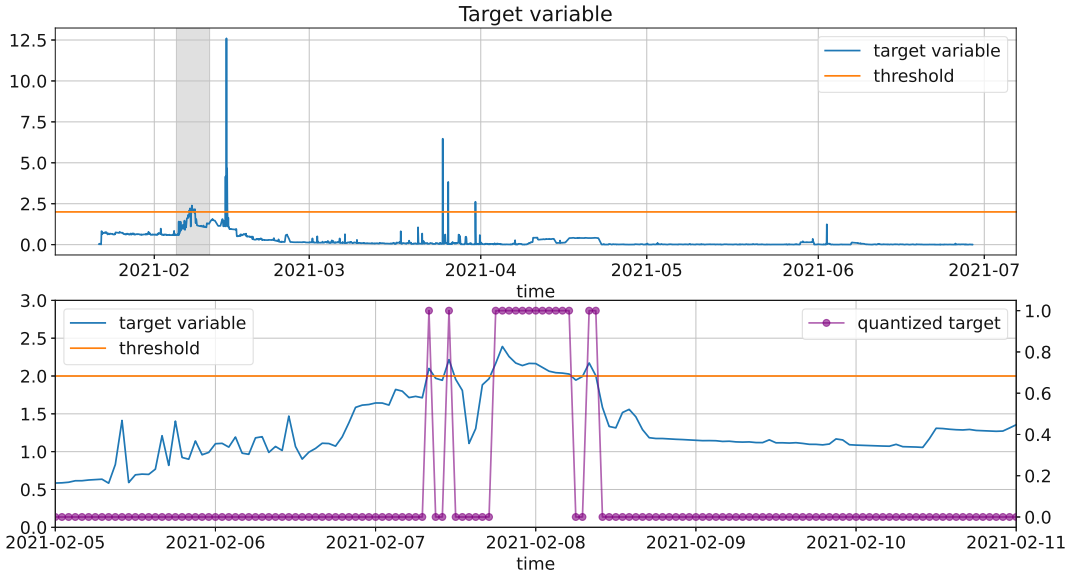

Figure 3.6.: Temporal evolution of the target variable for an FNA. The lower plot shows a zoomed in version of the range marked with a gray bar in the upper plot, with the quantized target variable plotted on top.

3.3.1. Temporal Extent of Data

A schematic representation of the data availability is shown in Figure 3.7. A FNA's data may start or end before the overall start and end time-stamps, and data may be missing in between.

In a first step, all FNAs whose data span less than 80% of T, i.e. for which $T_F = t_{\text{end}} - t_{\text{start}} < 0.8\,T$ holds, were excluded. The reason for this is that the data are split into several parts along the time-axis for cross-validation (CV) procedures, and each part should hold about the same amount of data. See Section 5.7 for further details on data splitting.

Strictly executed, missing data points in between start and end dates should be taken into account, but analyses revealed that typically only few data points are missing for FNAs, so that using FNA's T_F is sufficient for deciding which data to exclude.

A concrete example for a hub's data is shown in Figure 3.8. As can be seen in this example, three FNAs have a later starting date than the rest. Two of those had enough data to be kept, while one (marked in black) was removed from further processing. Several FNAs have a few missing data points, marked by red markers. A red marker here indicates that any variable may be missing; however, it was found that typically, all variables are missing and the gap in the data is detected from discontinuities in the time-stamps. The reasons for missing data points are not known to the author. The depicted hub's data exhibit the largest amount of missing data points in the whole data set (but not the most completely excluded FNAs).

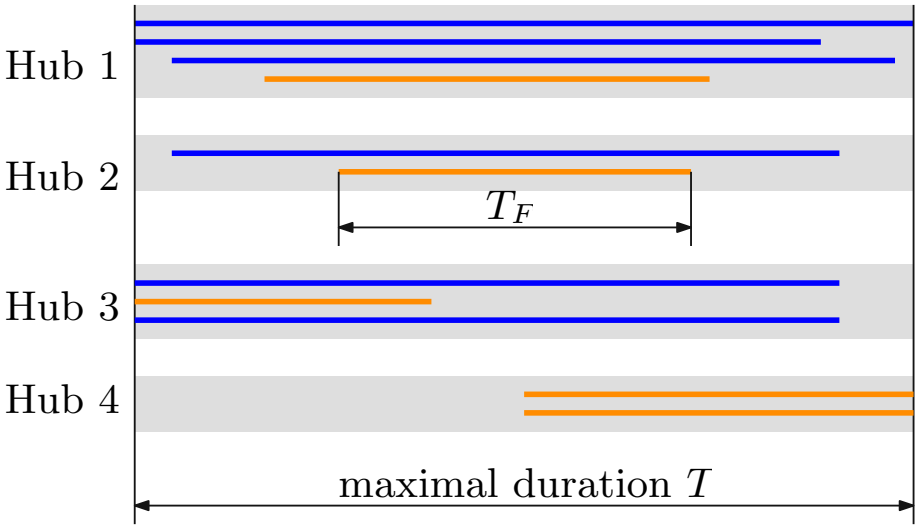

Figure 3.7.: Schematic representation of data availability. The maximal extent of data (first to last occurring in the data) T is used as reference value. Each line (blue or orange) represents an FNA's data. FNAs whose data duration (T_F) is less than 80% of T are excluded and not used for the experiments (orange). Even a whole hub's data may be excluded (hub 4 in this example).

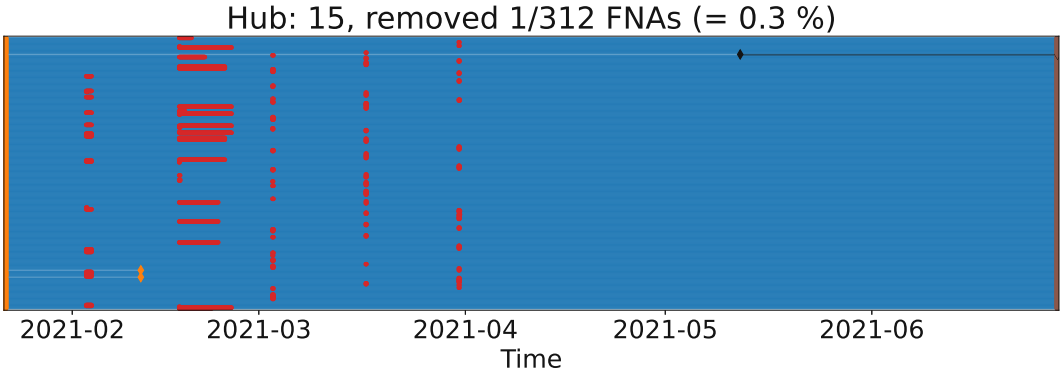

Figure 3.8.: Overview of available data for a single hub (compare to Figure 3.7). Start dates of records are marked by orange diamonds. End of records are marked with brown diamonds (same end date for all records here). There are 3 FNAs that have records starting later than the rest. FNAs excluded are shown in black: there is a single FNA whose data are excluded here due to the short duration (see text). Red markers indicate single data points that are missing. Several FNAs exhibit shorter periods of time where data is missing in between start and end dates. The missing data points occur at similar times for different FNAs, hinting at a common source of error.

Since relatively few data points are missing, and enough data is available overall, it was decided against using methods to impute missing values, but to simply ignore portions of missing data. The handling of missing values, in the form of omitting portions of the data with missing values, is discussed in more detail in Section 5.3.

3.3.2. Missing Onsets

Some FNAs' data show a particular pattern: data is missing a short time before or directly at that time-span where the target variable reaches or crosses the quantization threshold γ_q (marking bad performance). The part of the data that potentially marks the beginning of a problem and may be crucial for the prediction of overshooting the threshold is missing. This makes the usefulness of data that exhibit this pattern more than questionable.

Figure 3.9 illustrates an example of this pattern, where the temporal evolution of the target variable exhibits missing values just before a strong increase, and eventual overshoot of the threshold.

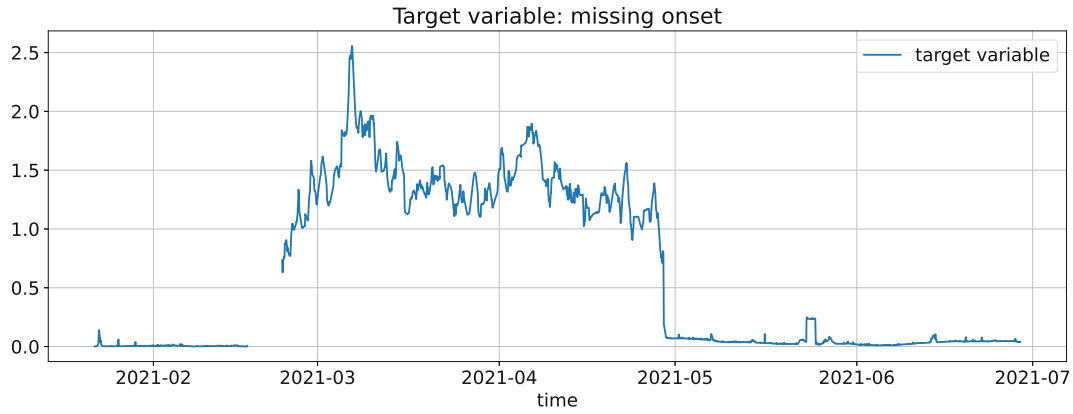

Figure 3.9.: Target variable with the missing onset pattern. See text for details.

3.4. Distributions

This section takes a look at some aspects of the distributions of the variables. Because of the considerable number of variables (20) involved, and especially the large amount of FNAs (2200), extensive and in-depth analysis of the distributions did not seem too meaningful. Furthermore, as will be outlined in Section 5.2, the models chosen (discussed in Chapter 6) do not require specific preprocessing of the data and thus no detailed knowledge of the variables' distributions or relationships. It was therefore deemed sufficient to investigate the following temporal (i.e., computed along the time-axis) summary statistics:

- mean,
- standard deviation,
- skewness.

This is meant to give at least a coarse understanding of the possible heterogeneity of the data collected from the different FNAs, effectively from different parts of the large (country-wide) network.

Figures 3.10 (mean), 3.11 (standard deviation) and 3.12 (skewness) give an overview of the distribution of the aforementioned quantities for all 2200 FNAs. Each dot in one of the plots represents one FNA. Recall that the terms "mean" and "standard deviation" in the variable names refer to the spatial aggregation described in Section 3.1.2.

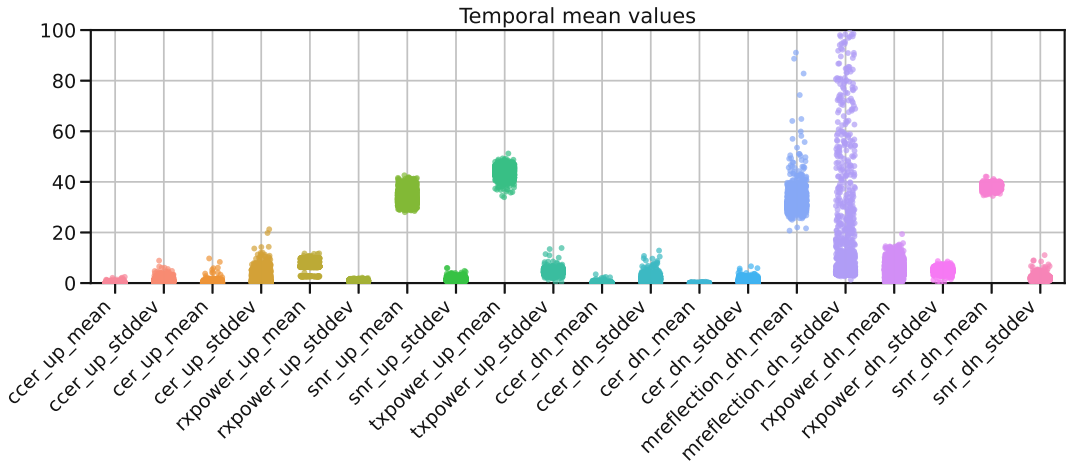

Figure 3.10.: Mean value (computed along the time-axis) of the time-series for each variable. Each FNA is represented by a single dot. The range of values shown on the y-axis is clipped to 100. The variable mreflection_dn_stddev has many instances reaching about 1000, with the largest value around 2500.

As is evident from Figure 3.10, most variables have moderate means, and the values are somewhat grouped closely, except for the variable mreflection.

A similar pattern can be seen for the temporal standard deviations in Figure 3.11, except that the variable cer_up_stddev has a slightly larger spread than other variables. For mreflection_dn_stddev, there seem to two separate groups.

Figure 3.12 shows the skewness. The variables related to CER and CCER show considerable spread in their right skew, as do the variables related to the micro-reflections. The other variables are less skewed and in general more concentrated for the different FNAs.

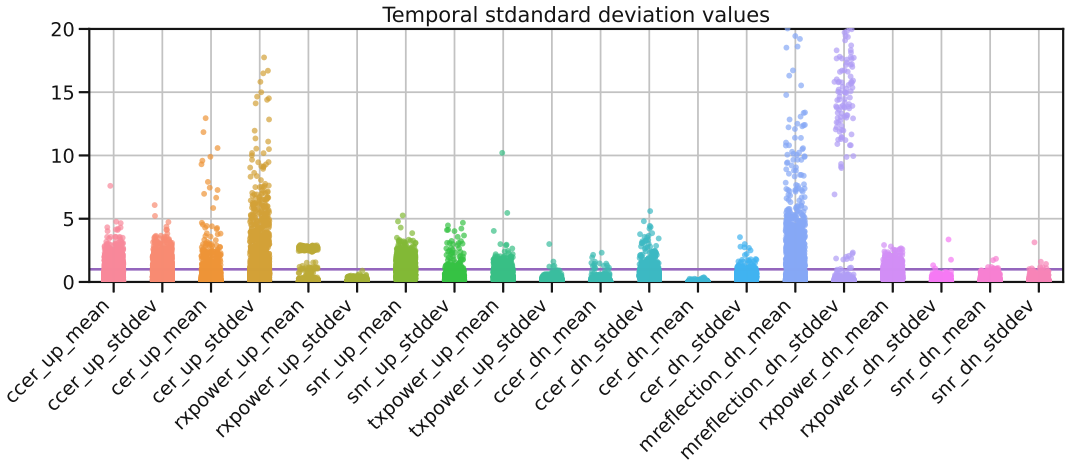

Figure 3.11.: Standard deviation value (computed along the time-axis) of the time-series for each variable. Each FNA is represented by a single dot. The range of values shown on the y-axis is clipped to 20. The variable mreflection_dn_stddev has many instances reaching about 400, with the largest value around 1000. The horizontal line marks a standard deviation of 1.

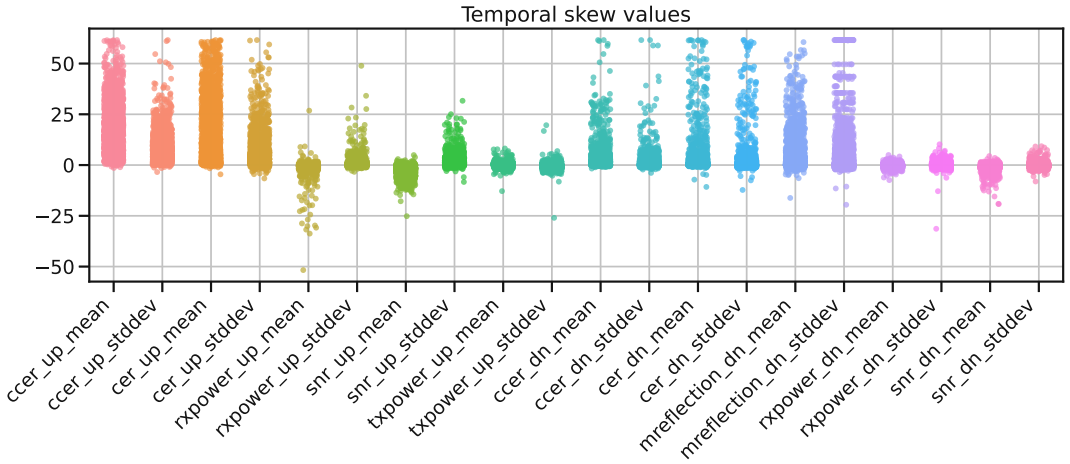

Figure 3.12.: Skewness value (computed along the time-axis) of the time-series for each variable. Each FNA is represented by a single dot.

3.4.1. All-Zero Values

When taking a closer look at the standard deviations computed across time for the FNAs, some showed values of exactly zero for one or more of the following variables:

- CER, both upstream (US) and downstream (DS)
- CCER, US and DS

The Codword Error Rate (CER) reflects the amount of transmitted codewords that contain errors that can be detected at the receiver (it may be that not all errors can be detected, this depends on the code used), and is thus a measure for the quality of the transmission. The Corrected Codword Error Rate (CCER) has a similar meaning, it reflects the amount of codewords whose errors could be corrected after detection (it may be that not all detected errors can be corrected, again depending on the code used).

Both these quantities are non-negative, and spatial means thus would be non-negative as well. It is extremely unlikely that all devices in an FNA have not a single codeword error in 5 months, and far more likely that there is an error in the data. It furthermore is extremely unlikely that these variables exhibit no variability over time, thus having standard deviations of zero, especially as they are spatial means or standard deviations. Therefore, it was assumed that rather there was an issue with the data, and the respective FNAs where removed from the data used for training the models.

Instead of removing the instances (of the affected FNAs), alternatively, the variables could have been removed for the whole data set, keeping the respective instances. However, the amount of FNAs with these cases is only about 13 % of all available FNAs (for more concrete numbers see Section 5.1.3), and furthermore both variables are expected to be in close relationship to the target variable. Under such circumstances, it is favorable to remove instances rather then variables (predictors) (Kuhn et al. 2020, p. 196).

3.5. Class Imbalance Ratio

This section investigates the distribution of classes in the aggregated data (as delivered), i.e. the proportions of instances considered events and non-events. One possible measure to describe the imbalance of classes in the data for two-class problems is the Imbalance Ratio (IR) (Fernández et al. 2018, p. 20). It is defined as the ratio of the number of instances (labeled as) belonging to the negative class (class of no interest) to the number of instances belonging to the positive class (the class of interest).

Figure 3.13 shows for all 2200 FNAs the number of occurring events. As described in Section 3.2 and shown in Figures 3.5 and 3.6, an event is defined by the target variable τ_t reaching or crossing the threshold γ_q. The FNA with

the most events has about 2500 (out of 3790 data points) events, about 65 % of the data points. 128 FNAs show more than 5% of events. 1302 FNAs have no occurrences of events.

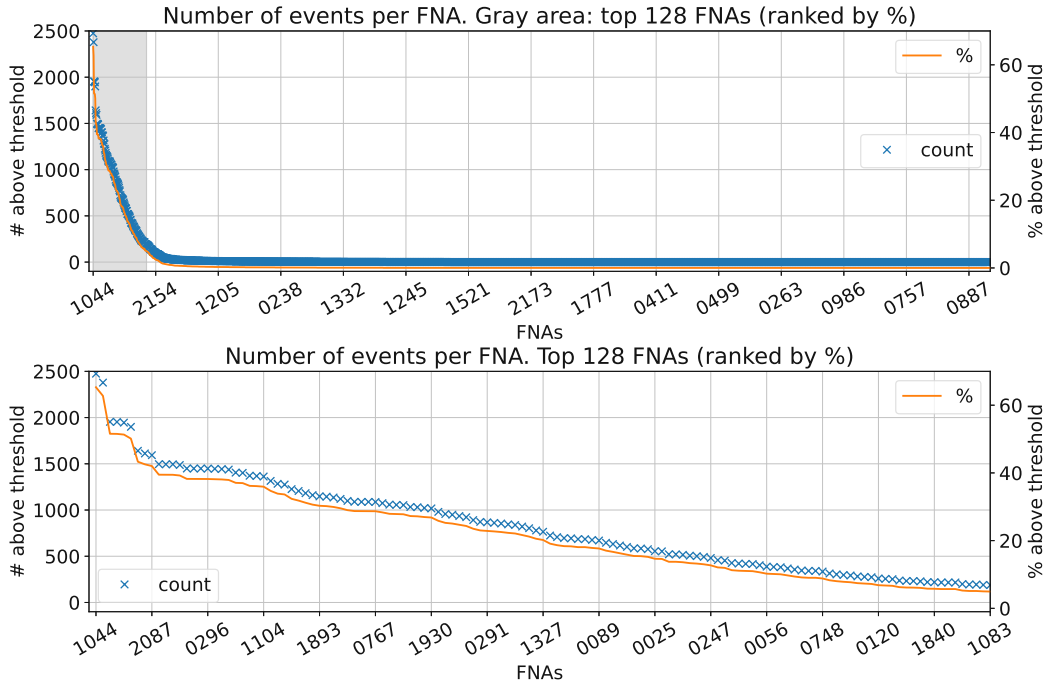

Figure 3.13.: Number and percentage of events per FNA. The lower subplot shows the top 128 (ranked by percentage) FNAs. This corresponds to the area marked in gray in the top subplot. All other of the 2200 FNAs have less than 5% of events.

Figure 3.14 shows the distribution of events over time for all FNAs. This figure is meant to give a rough impression of the overall occurrence of events over time across the whole network.

In absolute numbers, there are 116 857 events in 8 148 975 data points across all FNAs. Averaged across all FNAs, 1.43 % of the data points are of the class "event". This results in an Imbalance Ratio (IR) of about 70:1 for classes "non-event" (0) to "event" (1). There is thus a quite large imbalance between events and non-events in the data, and there are quite few examples for the minority class "event" in total.

Class imbalance ratios will be revisited in Section 5.8, after preprocessing the data, which changes the IR considerably for the worse. The whole next chapter is dedicated to the discussion of the impact of this strong imbalance.

Figure 3.15 shows the total number of events per time-stamp, as obtained by summing over all FNAs. This plot is interesting as it can give an idea of the proportion of events per fixed time-span, say a month. This will be further

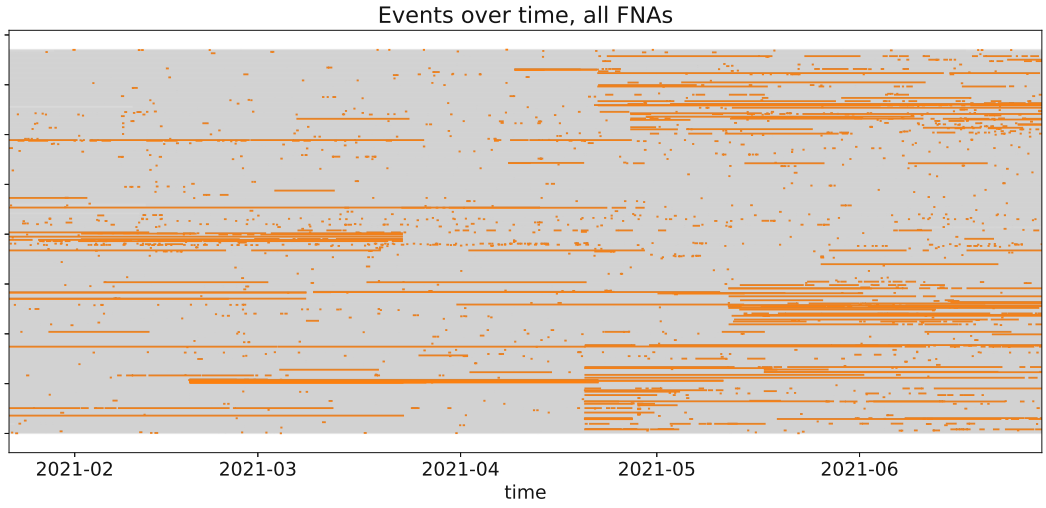

Figure 3.14.: Occurrence of events over time (as indicated by target variable), for all of the 2200 FNAs. Each row of pixels corresponds to one FNA.

discussed below in the context of splitting the data into training, validation and test parts.

Johnson et al. (2019, p. 4) state that besides the Imbalance Ratio (IR), the absolute number of examples in the minority class can be important. There may be simply too few instances of a class in the (labeled training) data to learn the "concept" of the respective class. This case is referred to as absolute rarity in the literature (He et al. 2013, p. 20), as opposed to the relative rarity that is quantified by the IR.

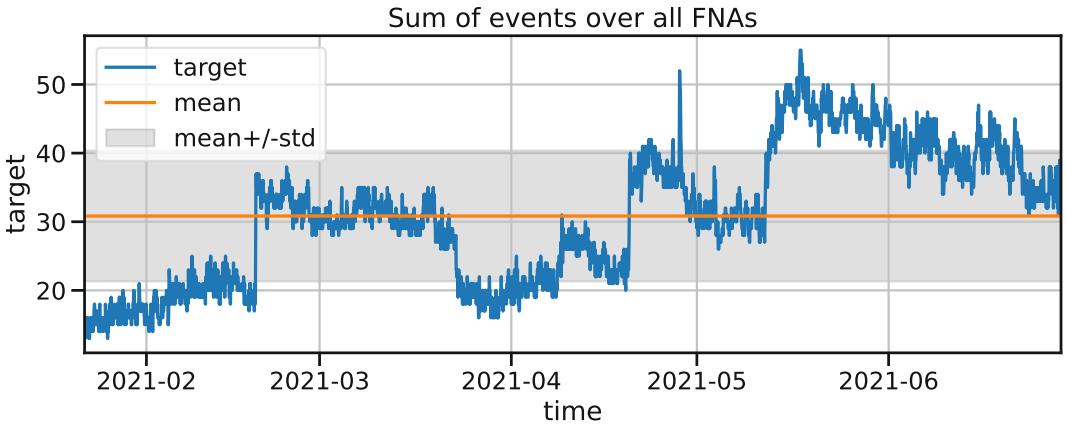

Figure 3.15.: Total number of events per time-stamp, sum over all of the 2200 FNAs. The blue curve is the sum along the vertical axis of Figure 3.14. While there is change over time, the sum does not deviate strongly from the mean ± one standard deviation (orange horizontal line, gray area) most of the time.

CHAPTER 4

Class Imbalance

As the exploratory analysis of the data showed (Chapter 3), the distribution of the classes among the instances is severely imbalanced. Class imbalance is a problem that impacts several points of a machine learning pipeline. It is a well known problem occurring in many real-world data sets and applications that deal with rare events (Johnson et al. 2019). Learning in the presence of imbalanced classes is a research field of its own, see e.g. (Fernández et al. 2018), (He et al. 2013), (Branco et al. 2016).

Johnson et al. (2019) state that the problems of imbalanced learning, and methods to address it, have been researched thoroughly over the last two decades for classical machine learning, but that research in the field of deep learning methods has not progressed very far.

This chapter will briefly look at the problems caused by class imbalance and discuss at which points in a machine learning pipeline mitigating actions can be taken to alleviate them.

There are three important points where leverage can be applied to combat (severe) class imbalance (Fernández et al. 2018), (He et al. 2013):

1. At the level of the data, i.e., applying certain kinds of processing to the data, before using them for training a model.
2. At the level of the models and algorithms used.
3. At the level of the decision rules or thresholds, after fitting a model to data.

These approaches are not mutually exclusive but may be used in combination, however, not all three approaches are applicable to every model. The following sections will take a brief look at these approaches and how they can be applied to the problem at hand.

In some publications, there is another point mentioned as being very important, namely the choice of the evaluation metric(s) used to measure a model's performance, see e.g. (He et al. 2013). The choice of metric is discussed at the end of this chapter in Section 4.4.

It shall be noted that Krawczyk (2016) uses a slightly different list of methods, namely points 1 and 2 as above, but states that a combination of both can be considered a third method. Point 3 from the list above is included in point 2 in (Krawczyk 2016).

Yet another slightly different categorization of methods to combat class imbalance is given in (Branco et al. 2016, p. 12), again by and large covering the above mentioned methods.

4.1. Data Level

Approaches to treat class imbalance at the data level are mostly based on re-sampling the data. Typically, the training data are modified, while other parts of the data used as holdout sets (validation and test sets) are kept as-is. Popular options are (Fernández et al. 2018, p. 80):

- Undersampling the majority class, i.e., reducing the number of instances labeled as the majority class (the class of no interest) in the training data. There are several ways for choosing the instances to be removed.
- Oversampling the minority class, i.e., sampling with replacement from the minority class (the class of interest). Again, several ways for how to choose instances are possible.
- Creating artificial (also termed synthetic) data points for the minority class, either based on existing instances or randomly.

These methods are not mutually exclusive, for example applying both oversampling the minority as well as undersampling the majority class is a common method. Some simple possible approaches will be briefly discussed in the following. In general, there is a vast amount of different re-sampling methods treated in the literature, see e.g. (Fernández et al. 2018) for an overview and discussion.

In general, it is important that the data used for evaluating the quality of the model, i.e. validation and test sets, resemble the distributions of minority and majority cases found "in the world" (Kuhn et al. 2013, p. 427), and thus are not subjected to re-sampling at all for the experiments conducted.

Sampling methods have been shown to mitigate the impact of class imbalance, but there is not one best method among the various available approaches (Kuhn et al. 2013, p. 429).

Furthermore, the optimal amount of re-sampling, in other words the target Imbalance Ratio (IR), may not be clear in advance (Branco et al. 2016, p. 16) and should thus be treated as a hyper-parameter and estimated during the model

selection process. This may, however, lead to unacceptably high additional computational effort.

4.1.1. Undersampling

There are several different ways to apply undersampling to the instances of the majority class. A simple version is to randomly select majority class instances to be dropped from the training data (Branco et al. 2016, p. 16).

Undersampling can have several advantages (Aggarwal 2017, 228f):

- Training on smaller data sets requires less power and is faster.
- Only instances of the class of no interest are removed, and the model can focus on the important instances from the class of interest.
- Due to the improved efficiency, using ensemble methods (i.e., fitting multiple models of the same type and combine output) becomes more feasible.

Reducing the number of instances can be a drawback if there is relatively few data to begin with. Furthermore, there is the risk that instances important for the learning process may be removed (Branco et al. 2016, p. 16).

4.1.2. Oversampling

Again, there exist several methods for oversampling the minority class, such as the simple case of random oversampling by sampling with replacement from the instances of the minority class. There is some agreement that this increases the chance of overfitting (Fernández et al. 2018, p. 80), (Branco et al. 2016, p. 16). As overfitting was found to be a problem during early experiments, already without any alteration of the data, random oversampling was not considered further.

An important point is that random oversampling is not able to mitigate the problem of absolute rarity (He et al. 2013, p. 29). Absolute rarity means that the absolute number of instances of the target class is low, besides a possibly high IR.

Oversampling in general increases the data set size, thus leading to higher effort during training (but this could easily be compensated by undersampling the majority class, if applicable).

4.1.3. Synthetic Oversampling

Synthetic Minority Oversampling Technique (SMOTE) (Chawla et al. 2002) is a popular method for creating synthetic examples for the minority class.

Preliminary experiments were conducted with the SMOTE method, but it was found to be extremely time-consuming: just processing one single fold would

take several hours of computing on a powerful machine (Machine G, see Section 7.1). Processing time for five folds (and one single setting for SMOTE, e.g. for the number of neighbors used) was estimated to be between 20 h and 30 h. Since this would have to be done for each of the five folds, and for each of the several different windowing settings (see Chapter 5), and possibly for different settings (parameters) for the chosen SMOTE method, the necessary processing time was considered to be too much by far for the scope of this thesis. Kuhn et al. (2020, p. 55) state that resampling of data to influence the class distribution and imbalance should be included inside a cross-validation scheme (or other methods that resample data to create training and validation splits), as opposed to doing it outside the scheme, before splitting the data.

Furthermore, the literature suggests that the original SMOTE algorithm has some drawbacks (Fernández et al. 2018, p. 100), and the number of available extensions was considered too vast for screening for alternatives for this thesis: Fernández et al. (2018) lists more than 90 published extensions to the original SMOTE.

Krawczyk (2016, p. 224) state that in cases of extreme class imbalance (they categorize this as imbalance ratios between 1000:1 and 5000:1), the minority class may often be poorly represented in terms of its structure (this relates to the notion of absolute rarity mentioned above). This can lead to decreased performance in classification settings when using methods such as SMOTE, as well as random re-sampling methods.

Johnson et al. (2019, p. 35) argue for the use of model-level (algorithm-level) methods (treated in the next section) in favor of data-level methods due to the increased computational costs of the latter when dealing with large amounts of data.

Due to the points briefly mentioned above (danger of overfitting, computational cost, necessity to include re-sampling methods into hyper-parameter tuning and model selection process), no re-sampling (data level) methods were used in the experiments (see Chapter 7), except for random undersampling where necessary to avoid memory problems with certain models.

4.2. Model Level

Two prominent approaches for dealing with class imbalances at the algorithmic level are:

1. Cost-sensitive learning (Aggarwal 2017, p. 223), (Fernández et al. 2018, p. 63). Several different methods exist under this umbrella, here the case of weights associated with classes is of interest (Fernández et al. 2018, pp. 64 f). The general idea is to include different weights for the different classes directly into the loss function used for training a model. An advantage here is that only changes to the loss functions are necessary, and the rest of the model can be used without any changes (Aggarwal 2017, p. 222).

2. Ensemble methods such as bagging and boosting (Fernández et al. 2018), (He et al. 2013, pp. 66 f). Aggarwal (2017, p. 230) specifically mentions boosting methods. In general, for the case of ensembles, either the base models, the combination scheme, or both together can be geared towards alleviating class imbalance. In this thesis, only the first version will be addressed, more concretely combining base learners sequentially via boosting, where the base learner models are themselves equipped with ways to deal with class imbalance.

In the remainder of this section, only cost-sensitive learning with weights applied on a per-class basis will be discussed further. The boosting models used are discussed in Section 6.5.

4.2.1. Types of Models

Classification models can be distinguished by the way they arrive at, or rather emit, class labels for given input data instances. Following the ideas of Bayesian decision theory (see e.g. (Bishop 2006), (Duda et al. 2001)), some types of classifiers model the posterior probabilities $P(\mathcal{C}_j|\boldsymbol{x})$ of the n_C possible classes $\mathcal{C}_j$ with $j = 0, 1, \ldots n_C - 1$, given a datum (instance) $\boldsymbol{x}$. The model than produces as a prediction a "soft" label score that can be interpreted as a probability for the respective class. A soft label score can be quantized to a "hard" label by applying a decision rule (see Section 4.3 below). Other models directly emit "hard" class labels (e.g. 0 or 1) without the intermediate step of a probability score. An overview of these different types of models is given in (Bishop 2006, pp. 42 ff):

1. Generative models. These model the joint probabilities $P(\mathcal{C}_j, \boldsymbol{x})$ of the classes $\mathcal{C}_j$ and data $\boldsymbol{x}$. The posterior probabilities $P(\mathcal{C}_j|\boldsymbol{x})$ can then be derived from the joint probabilities. An example of such a model is Linear Discriminant Analysis (LDA).

2. Discriminative models are directly modeling the posterior probabilities $P(\mathcal{C}_j|\boldsymbol{x})$. Among the models belonging to this category are logistic regression, decision tree classifiers and neural networks (with sigmoid or softmax output).

3. Models that directly yield a class label only, by learning a discriminative function that maps a datum $\boldsymbol{x}$ to a class label directly.

Classifiers of types 1 and 2 are sometimes referred to as scoring classifiers or probabilistic classifiers, models of type 3 as discrete classifiers (see e.g. (Fawcett 2006), (Bishop 2006)). For the experiments conducted here, models of the type 2 are used, see also Chapter 6.

Using models of type 2, resulting in the availability of the predicted posterior probabilities, has several advantages (Bishop 2006, pp. 44 f). One of them is

that costs (also called weights in some settings) can be applied to the predicted posterior probabilities in a decision rule that finally assigns (hard) class labels to data. The decision for a class label $\hat{y}$ can be viewed as a post-processing step after predicting the posterior probabilities $P(\mathcal{C}_j|\boldsymbol{x})$ for a given instance. This is further discussed in Section 4.3.

4.2.2. Loss Functions and Weights

During training a classifier in a supervised setting, a loss function (or cost function) L is used to compute the overall loss $\mathcal{L}$ across all presented data instances $\boldsymbol{x}$ of a training data set, where each datum has an associated ground truth label y, for the current state of the classifier. The goal of the training procedure is to adapt the model's parameters $\boldsymbol{\theta}$, representing the state of the model, such that the overall loss incurred on the training set is minimized.[1]

For each datum $\boldsymbol{x}_i$ of $i = 1, \ldots, N$ elements in a data set $\mathcal{D}$, with corresponding binary ground-truth class labels $y_i \in \{0, 1\}$, a classifier produces (finally) a predicted label $\hat{y}_i \in \{0, 1\}$, where 1 is again considered the label for the class of interest (target class). We set $\mathcal{C}_1$ to correspond to $y = 1$ and $\mathcal{C}_0$ to correspond to $y = 0$. Probability scores emitted by discriminative models (type 2 above) represent the modeled posterior probabilities $P(\mathcal{C}_j^{(i)}|\boldsymbol{x}_i)$ that are eventually quantized to a hard predicted label $\hat{y}_i$ using a decision rule (the index i will often be omitted in the following for brevity).

For the binary case treated here, the two posterior probabilities are related as $P(1|\boldsymbol{x}) + P(0|\boldsymbol{x}) = 1$. For convenience, we introduce the following simplified notation:

$$P(\mathcal{C}_1|\boldsymbol{x}) = P(1|\boldsymbol{x}) = p_t, \tag{4.1}$$

$$P(\mathcal{C}_0|\boldsymbol{x}) = P(0|\boldsymbol{x}) = 1 - p_t. \tag{4.2}$$

Many different loss functions suitable for classification can be found in the literature, in many different formulations. The notations may also depend on the concrete model the respective function is used in.

Not all loss functions allow for cost-sensitive training. For example, Fernández et al. (2018, p. 64) state that the 0-1 loss (misclassification loss) is not a good loss function for imbalanced problems, as it does not allow to give different costs to different misclassification cases. Rather, it assigns a loss of 1 to mis-classified examples, and 0 to correctly classified.

A loss function that is widely used for classification problems and does allow for cost-sensitive training is the cross-entropy loss. For the cross-entropy loss (also log loss, deviance) (see e.g. (Bishop 2006, p. 206)), we adopt (mostly) the

[1] Note however that the minimization procedure may be stopped at a certain point to avoid overfitting the model to the training set. This is discussed further in Section 6.4.

notation in Lin et al. (2017, p. 3):

$$\mathrm{CE}(p_t, y) = \begin{cases} -\log(p_t), & \text{if } y = 1, \\ -\log(1 - p_t), & \text{if } y = 0. \end{cases} \tag{4.3}$$

For the class labels $y \in \{0, 1\}$ used here, the cross-entropy can also be written as:

$$\mathrm{CE}(p_t, y) = -y \log(p_t) - (1 - y) \log(1 - p_t), \tag{4.4}$$

and all of the loss functions presented in the following can be written in this form as well. The overall loss $\mathcal{L}$ for a given data set is then computed across all examples, $i = 1, \ldots, N$ (see e.g. (Bishop 2006, p. 206)):[2]

$$\mathcal{L} = \sum_{i=1}^{N} \mathrm{CE}(p_{t,i}, y_i) = \sum_{i=1}^{N} -y_i \log(p_{t,i}) - (1 - y_i) \log(1 - p_{t,i}). \tag{4.5}$$

Note that the overall loss $\mathcal{L}$ depends on the data set $\mathcal{D}$, via the data $\boldsymbol{x}$ and labels y, as well as the state of the model (defined by parameters $\boldsymbol{\theta}$), via the predicted probability scores p_t computed from the data $\boldsymbol{x}$.

Multiplicative weights, one scalar weight factor w_j for each class $\mathcal{C}_j$ across all respective instances, can be used to modify the losses for each class separately (Lin et al. 2017, p. 3):

$$\mathrm{CE}_w(p_t, y) = \begin{cases} -w_1 \log(p_t), & \text{if } y = 1, \\ -w_0 \log(1 - p_t), & \text{if } y = 0, \end{cases} \tag{4.6}$$

where weights are suggested to be related as $w_1 = \alpha$ and $w_0 = 1 - \alpha$ in (Lin et al. 2017). Note, however, that in general, class weights for different classes need not be related to each other, see e.g. (Buja et al. 2005, p. 13). By specifying higher weights to increase the losses for a certain class, a model can be tuned towards making less mistakes for that class (see also below).

The weights w_j introduced here may or may not be directly related to the costs C_{kj} discussed in Section 1.2, and hence the different term "weights" instead of "costs" is used here. The application of per-class weights in Eq. (4.6) essentially corresponds to the case shown in Table 1.2.3 (left), where only weights for the error cases are used. However, here, the weights are first and foremost meant to increase the performance of the model during training and may not reflect an actual cost situation describing the overall decision problem; the costs may in any case not be known at all, or precisely enough, at this stage of tackling the problem. Furthermore, it may be the case that the same trained model is to be used in different contexts requiring different cost structures. This is further discussed in Section 4.2.3.

[2] Some authors average the loss accumulated across the respective data set, by dividing the sum of the losses by the number of instances N, see e.g. (Buja et al. 2005, p. 5).

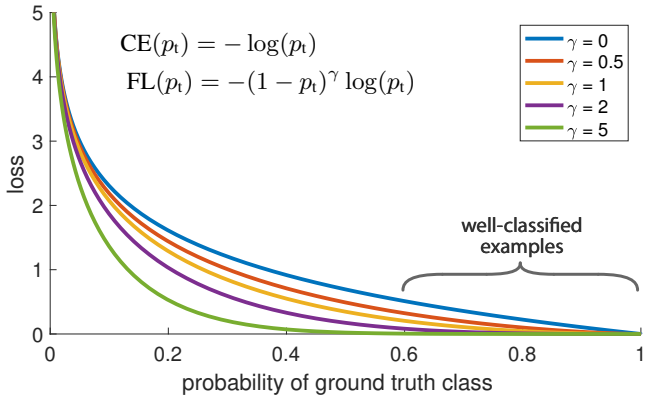

Figure 1. We propose a novel loss we term the *Focal Loss* that adds a factor $(1 - p_t)^\gamma$ to the standard cross entropy criterion. Setting $\gamma > 0$ reduces the relative loss for well-classified examples ($p_t > .5$), putting more focus on hard, misclassified examples. As our experiments will demonstrate, the proposed focal loss enables training highly accurate dense object detectors in the presence of vast numbers of easy background examples.

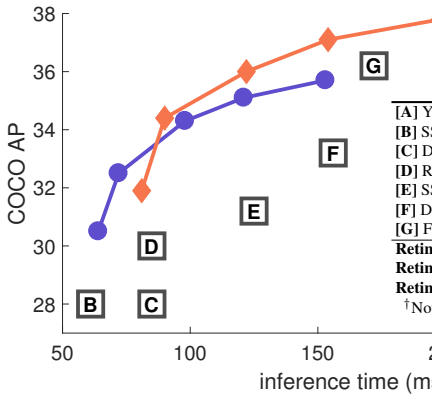

Figure 2. Speed (ms) versus accuracy (AP) o
Enabled by the focal loss, our simple one-s
tor output for all previous one-stage and t
cluding the best reported Faster R-CNN [
We show variants of RetinaNet with ResNet
and ResNet-101-FPN (orange diamonds) a
pixels). Ignoring the low-accuracy regime
forms an upper envelope of all current detec
variant (not shown) achieves 40.8 AP. Detai

Figure 4.1: Loss curves for the focal loss for different values of the focusing parameter γ. For $\gamma = 0$, the focal loss is equivalent to the cross-entropy loss. With increasing γ, the loss for higher values p_t (more confidence in the prediction) is discounted as γ increases. Figure from (Lin et al., 2017).

As an extension to the cross-entropy loss, Lin et al. (2017) introduce the focal loss (FL):

$$\mathrm{FL}(p_t, y) = \begin{cases} -(1-p_t)^\gamma \log(p_t) & \text{if } y = 1 \\ -p_t^\gamma \log(1-p_t), & \text{if } y = 0, \end{cases} \tag{4.7}$$

including the so-called modulating factors $(1 - p_t)^\gamma$ and p_t^γ, respectively. The scalar parameter $\gamma \geq 0$ is termed the focusing parameter and suggested to be in $[0, 5]$, where $\gamma = 0$ falls back to the cross-entropy loss.

A version with weights is introduced in (Lin et al., 2017) as well:

$$\mathrm{FL}_w(p_t, y) = \begin{cases} -w_1 (1-p_t)^\gamma \log(p_t), & \text{if } y = 1, \\ -w_0\, p_t^\gamma \log(1-p_t), & \text{if } y = 0, \end{cases} \tag{4.8}$$

where what was said above for the weights holds here as well.

The modulating factor (together with the focusing parameter) is designed to scale the loss with the confidence of the prediction, the magnitude of p_t, such that instances that are easier to classify will yield lower loss than instances that are harder to classify. This is visualized in figure 4.1. For higher values of the focusing parameter γ, the loss curves decrease stronger for higher values of p_t. Thus, a higher value of $p_t = P(1|\boldsymbol{x})$ for a given instance $\boldsymbol{x}$, interpreted as higher confidence in the prediction, will decrease the loss incurred for this instance.

Lin et al. (2017, p. 3) state an interesting aspect of focal loss: it is designed to down-weight "inliers", referring here to easy to classify instances. This is the opposite behavior of robust loss functions that aim at reducing the influence of outliers by down-weighting them.

Abstract

The highest accuracy object detectors to date are based on a two-stage approach popularized by R-CNN, where a classifier is applied to a sparse set of candidate object locations. In contrast, one-stage detectors that are applied over a regular, dense sampling of possible object locations have the potential to be faster and simpler, but have trailed the accuracy of two-stage detectors thus far. In this paper, we investigate why this is the case. We discover that the extreme foreground-background class imbalance encountered during training of dense detectors is the central cause. We propose to address this class imbalance by reshaping the standard cross entropy loss such that it down-weights the loss assigned to well-classified examples. Our novel Focal Loss *focuses training on a sparse set of hard examples and prevents the vast number of easy negatives from overwhelming the detector during training. To evaluate the effectiveness of our loss, we design and train a simple dense detector we call RetinaNet. Our results show that when trained with the focal loss, RetinaNet is able to match the speed of previous one-stage detectors while surpassing the accuracy of all existing state-of-the-art two-stage detectors. Code is at* https://github.com/facebookresearch/Detectron

1. Introduction

Current state-of-the-art object dete
a two-stage, proposal-driven mechanis
in the R-CNN framework [11], the fir
sparse set of candidate object locations
classifies each candidate location as or
classes or as background using a convo
work. Through a sequence of advances
two-stage framework consistently achie
the challenging COCO benchmark [21]

Despite the success of two-stage
question to ask is: could a simple one-st
similar accuracy? One stage detectors
regular, *dense* sampling of object locat
pect ratios. Recent work on one-stage
YOLO [26, 27] and SSD [22, 9], dem
results, yielding faster detectors with
40% relative to state-of-the-art two-stag
This paper pushes the envelop furthe
stage object detector that, for the firs
state-of-the-art COCO AP of more co

4.2.3. Choosing Weights

As was stated above, the goal of cost-sensitive learning (with per-class associated costs) is making a classifier focus on classes having higher misclassification costs already during training (Fernández et al. 2018, p. 64). Several ways exist to set the weights for the different classes. This sections is mostly concerned with choosing weights used during training (in a loss function) and not choosing costs for the decision stage (used in a decision rule). However, some of the points mentioned here concern that latter case (see Section 4.3) as well.

Fernández et al. (2018, pp. 66 f) point out the importance of choosing the weights appropriately for the effectiveness of training a model and state two ways for obtaining concrete values for weights (costs):

1. Weights provided by domain experts, based on the problem at hand. This could be some estimate of true financial costs associated with the error cases of the problem. In such cases, the weights used for training might conceptually mirror the costs that would otherwise be applied only at the decision stage.

2. Weights estimated from training data. One possibility is to directly apply the Imbalance Ratio (IR) estimated from the training data as weight factors during the training process.

For the problem addressed in this thesis, no such expert knowledge as mentioned in 1), and thus and no cost estimates, was available at the stage of model development. Furthermore, the application of actual financial costs seems much more practically relevant at the decision stage, as we assume that the costs can change over time and with the location of application of the model.

Choosing weights directly based on the Imbalance Ratio (IR) (in the training data) as suggested in 2) may be problematic due to the following reason: the class imbalance may not be the only property of the problem (or data) making the learning of patterns in the data hard; other properties such as noisiness of the data and possible overlap of the classes may add to the difficulties (Fernández et al. 2018, p. 67).

Due to the points just discussed, it was decided to include the selection of methods to assign weights, and assigning concrete weight values, into the model selection process (see Section 6.6). In fact, choosing weights for the classes was included in the hyper-parameter optimization process, see Chapter 7 and Section 7.4 in particular for the methods used for assigning weights. Choosing weights based on the class imbalance was one method, randomly choosing a weight factor for the target class from a given range was another.

Finally, it shall be noted that weights can also be assigned on a per-instance basis, e.g. to increase the influence of certain instances in a data set on the estimated model parameters. This is referred to as case weights in Kuhn et al. (2013, p. 426). Assigning weights to single instances would most likely need

careful inspection of the data if to be done manually, or some automatic method; note that the focal loss cost function discussed above (Eq. (4.7)) can be viewed as such a method, as it assigns different loss to individual instances based on how high the confidence in their class association is. Besides the use of focal loss, this approach was not further considered in this thesis.

4.2.4. Equivalence of Approaches

Branco et al. (2016, p. 16) state the interesting fact that for classification problems, changing the Imbalance Ratio (IR) of the data (e.g. by re-sampling) is equivalent to adjusting the costs for misclassification for the different classes. This is pointed out as well in (Aggarwal 2017, p. 222), where they consider re-sampling as treated in Section 4.1 as an indirect form of cost-sensitive learning, since presenting an increased number of examples (in relation to the original ratio) of a certain class will raise the overall losses incurred by this class during training. This can be understood from Equation (4.5) (sum of losses of the data instances encountered during training) and Equation (4.6) (the weighted cross-entropy loss): increasing the number of instances of the minority class (oversampling) will increase the summed costs due to this class; the same effect can be achieved by applying higher weights (costs) to the instances of this class. Similar relations hold for other re-sampling methods.

Branco et al. (2016, pp. 26 f) additionally state that besides re-sampling and changing costs, tuning the quantization threshold for obtaining class labels from posterior probabilities (discussed in the next section) will yield classifiers of the same performance. However, an important point to consider here is the stage at which the respective method is applied:

1. Re-sampling the data is done earliest, before even training a model, or possibly on the fly during training. Some implications of such methods and the possible risk of overfitting were discussed above.

2. Incorporating weights into loss functions is done during model training, and may be necessary to even make a model pick up relevant patterns in the data.

3. At the latest stage, after training a model (of type 1 or 2 as mentioned above), it is still possible to influence the quantization of the estimated posterior probabilities to class labels that is done by the decision rule, or using a threshold (cutoff). This post-processing of the model outputs can incorporate costs to tune the quantization.

For points 1 and 2 above, the model is fixed after training and changes to either the class (im-)balance or weights for the classes are no longer possible. Of course, the model may be re-trained with a changed setting if affordable. Using a method following point 3, however, changes can be applied quite flexible once a trained

model was obtained. If all of these three methods are able to achieve the same performance, point 3 offers the most flexibility. Note again, however, that the approaches discussed in this chapter are not mutually exclusive. In fact, at least approaches 2 and 3 as listed above were used in all models here.

4.3. Processing Model Output and Decision Rules

As was pointed out above, the output of a discriminative model (type 2), the posterior probabilities of class membership for each data instance, need to be processed further to arrive at actual class labels.

In the last section (Section 4.2.4), it was stated that this post-processing step allows for a flexible tuning of the decision by incorporating costs. This flexibility is viewed as an asset as it can handle cases of changing environment (costs for error cases) and changing application scenarios and locations (Provost et al. 2001).

This section will take a brief look at how to arrive at class labels given the output of a discriminative model (type 2).

4.3.1. Decision Rule

Models of type 2 (discriminative), as discussed above in Section 4.2.1, yield as their output the posterior probabilities for the classes. A method or rule that quantizes the posterior probability to a class label $\mathcal{C}_j \in \{0, 1\}$ (for the binary case) is called a decision rule.

Following the ideas of Bayesian decision theory, the well known Bayes decision rule can be derived using the conditional risk (Duda et al. 2001, p. 25):

$$R(\hat{\mathcal{C}}_k|\boldsymbol{x}) = \sum_{j=0}^{n_C-1} C_{kj}\, P(\hat{\mathcal{C}}_j|\boldsymbol{x}) \tag{4.9}$$

with both k and $j = 0, \ldots, n_C - 1$ and n_C the number of classes. $R(\hat{\mathcal{C}}_k|\boldsymbol{x})$ is the expected cost for deciding for class label $\mathcal{C}_k$ for given datum $\boldsymbol{x}$; the notation $\hat{\mathcal{C}}_k$ is meant to show that this is a predicted probability. Clearly, Eq. (4.9) involves the costs as discussed in Section 1.2 as factors for the posterior probabilities (that are the outputs of a discriminative model here).

For the binary case, Eq. (4.9) yields (for k and $j = 0, 1$):

$$R(\hat{\mathcal{C}}_0|\boldsymbol{x}) = C_{00}\, P(\hat{\mathcal{C}}_0|\boldsymbol{x}) + C_{01}\, P(\hat{\mathcal{C}}_1|\boldsymbol{x}), \tag{4.10}$$

$$R(\hat{\mathcal{C}}_1|\boldsymbol{x}) = C_{10}\, P(\hat{\mathcal{C}}_0|\boldsymbol{x}) + C_{11}\, P(\hat{\mathcal{C}}_1|\boldsymbol{x}). \tag{4.11}$$

Ignoring the costs C_{kk} (for correct decisions), and deciding for the class label that minimizes the conditional risk, the Bayes decision rule can then be written as (Duda et al. 2001, pp. 24 f, 63):

$$\text{decide for class 1 if} \quad C_{01}\, P(1|\boldsymbol{x}) > C_{10}\, P(0|\boldsymbol{x}), \quad \text{else for class 0.} \tag{4.12}$$

Here, C_{10} and C_{01} are the costs for mis-classifying true class 0 as class 1, and vice-versa, respectively. The costs C_{00} and C_{11} for correct classification can be omitted or assumed to be absorbed into the costs C_{10} and C_{01} (it is assumed that the costs C_{kk} are smaller than the costs C_{kj}). This was briefly discussed in Section 1.2, see especially Tables 1.2.2 and 1.2.3.

With the alternative notations for the cost factors shown in Table 1.2.3 (left) (and the shorthand notation for the probability for the target class), the decision rule can be written in a form that more clearly references the error types:

$$\text{decide for class 1 if} \quad C_{\text{FN}}\, p_t > C_{\text{FP}}\,(1 - p_t), \quad \text{else for class 0.} \tag{4.13}$$

Thus, this decision rule can be used to incorporate costs for errors into the decision. Note that the posterior probabilities are obtained from a trained model, and the actual quantization to class labels can be influenced at any point after training in this way. In particular, the behavior of the overall decision system can be adapted to changes in the environment the system operates in, such as changing costs. Furthermore, it could be the case that one model is used in different spatial contexts (at the same time) where costs vary by location. In the scenario of this thesis, this could reflect different costs of maintenance for different parts of Austria, caused e.g. by a varying spatial density of network nodes, and thus a different number of possible components to screen in case of errors.

4.3.2. Threshold Method

In general, as an alternative to using a decision rule, the quantization of posterior probabilities can also be achieved using simple thresholding, involving a decision threshold (cutoff) γ_c:

$$\text{decide 1 if} \quad P(1|\boldsymbol{x}) = p_t > \gamma_c, \tag{4.14}$$

$$\text{decide 0 if} \quad P(0|\boldsymbol{x}) = 1 - p_t \leq \gamma_c. \tag{4.15}$$

This allows to tune between False Positive (FP) and False Negative (FN) error cases by varying the threshold γ_c.

Rearranging the Bayes decision rule, Eq. (4.13), repeated here for convenience:

$$\text{decide for class 1 if} \quad C_{\text{FN}}\, p_t > C_{\text{FP}}\,(1 - p_t), \quad \text{else for class 0,}$$

an equivalent formulation with a threshold based on the costs can be obtained (Fernández et al. 2018, pp. 65 f):

$$\text{decide 1 if} \quad p_t > \frac{C_{\text{FP}}}{C_{\text{FP}} + C_{\text{FN}}} = \gamma_{c,\text{Bayes}}, \quad \text{else for class 0.} \tag{4.16}$$

Trivially, for equal costs, the optimal threshold is 0.5 – carefully note however that this assumes that the (predicted) probabilities are properly calibrated (see next Subsection 4.3.3).

It shall be noted that choosing different thresholds does not improve the classifier itself (Kuhn et al. 2013, pp. 425, 431), (Johnson et al. 2019, p. 32) – it is done after the model is already trained, after all – but rather is useful for trading off between the different error cases (FN and FP). When looking at the confusion matrix (see Table 1.2.1), changing the threshold moves case counts between the rows, but not from the diagonal to the off-diagonals of the confusion matrix (Kuhn et al. 2013, 425f).

In case a threshold is to be selected independent of costs, this should be done based on a separate data set (such as the validation set and not the training or test data), otherwise bias may be introduced (Kuhn et al. 2013, pp. 262, 425).

Visualizations (plots) can be used for guiding the selection of a threshold, in addition to e.g. the confusion matrix, such as the ROC curve plot (see below in Section 4.4) or the discrimination diagram (see (Prati et al. 2011)), as well as the mosaic plot of the confusion matrix (see e.g. (Kuhn et al. 2020, p. 44)).

An interesting possibility is offered when using the threshold method where two thresholds are used instead of just one, the so-called reject option (Bishop 2006, p. 42). Here, a "don't know" region is inserted between the two actual classes, where the classifier can signify that it cannot make a decision with sufficient confidence:

$$\text{decide 1 if} \quad p_t > \gamma_{\text{up}}, \tag{4.17}$$

$$\text{decide "don't know" if} \quad p_t \leq \gamma_{\text{up}} \quad \text{and} \quad 1 - p_t > \gamma_{\text{lo}}, \tag{4.18}$$

$$\text{decide 0 if} \quad 1 - p_t \leq \gamma_{\text{lo}}. \tag{4.19}$$

The inclusion of costs for the decisions is possible as well (Bishop 2006, p. 42).

This concept was not further explored in this thesis, but could be potentially useful for practical applications.

4.3.3. Calibrated Probabilities

We shall briefly mention the notion of calibrated probabilities in relation to discriminative classifier models.

There exists a visualization useful for qualitatively examining the calibration of the classifier model, the so called calibration plot (or calibration curve) (see e.g. (Kuhn et al. 2013, pp. 249 ff)). The process of creating such a plot is instructive for understanding the concept, hence the construction is described in the following: For a set of data instances $\boldsymbol{x}_i$ with a ground-truth (class labels y_i) available, such as a validation or a test set, predict the posterior probabilities $P(\mathcal{C}_1^{(i)}|\boldsymbol{x}_i)$ for the target class 1, in short $p_{t,i}$. The predicted probabilities are then ranked in ascending order; each $p_{t,i}$ has an associated $\boldsymbol{x}_i$ whose true class label y_i is known. Next, the predicted probabilities are binned and the mid-point of each of the $k = 1, \ldots, K$ bins is taken, say $\bar{p}_{t[k]}$, where $[k]$ is meant to denote the respective bin of values. Finally, the $\boldsymbol{x}_{i[k]}$ corresponding to the $p_{t,i}$ values in bin $[k]$ are taken and the rate of true events (number of true class 1 cases in the

bin divided by the number of elements in the bin) is computed, say $r_{\mathcal{C}_1[k]}$. The calibration plot now displays the midpoints on the x-axis and the corresponding rates $r_{\mathcal{C}_1[k]}$ on the y-axis. A model that emits well calibrated probabilities would display the points on a 45° line (Kuhn et al. 2013, p. 249).

Well calibrated probabilities can be helpful for interpreting the model output, if the raw predicted probabilities are used directly, and can be useful for other reasons, see e.g. Guo et al. (2017) for a brief discussion. Furthermore, for being able to choose a decision threshold (as discussed in the previous section) in a meaningful way, (somewhat) well calibrated probabilities can be beneficial, or might even be necessary. Different classification algorithms have their own respective typical shapes of distortion of calibration curves. Several methods exist to remove these distortions, among them is fitting a function to the calibration curve such that the distortion is removed and the predicted probabilities are equalized (Kuhn et al. 2013, pp. 249 ff), (Niculescu-Mizil et al. 2005, pp. 625 f), (Guo et al. 2017).

Figure 4.2 shows examples of calibration plots for different classifiers applied to synthetic data of a binary classification problem, along histograms over the predicted scores (that can be interpreted as posterior probabilities for some of the models shown). The plots are also meant to illustrate the distortions introduced by the different models.

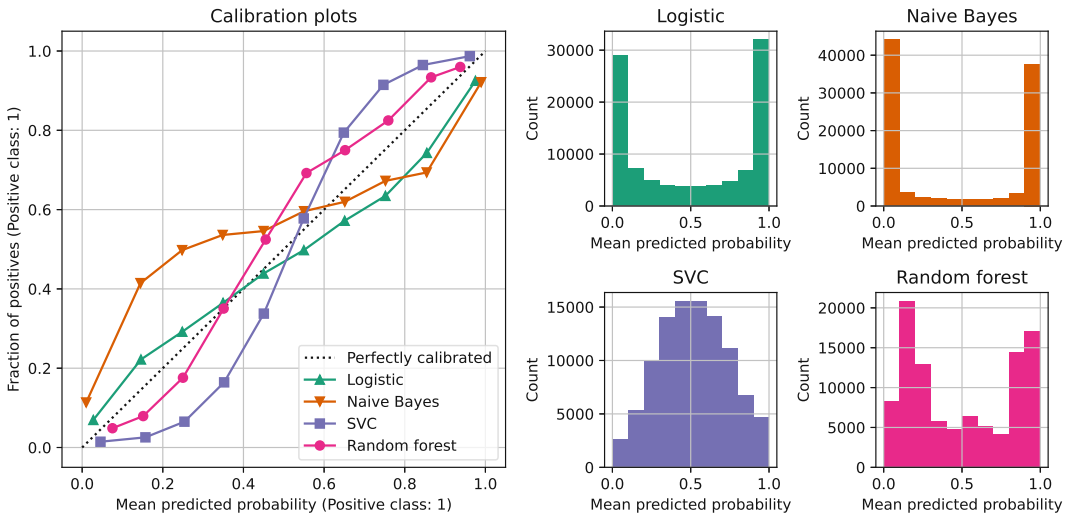

Figure 4.2.: Calibration plots (left) and corresponding histograms of the predicted scores (right) for four different classifiers to illustrate the notion of calibrated probabilities. Perfectly calibrated probabilities would follow the dashed line. Classifiers: Logistic Regression, Naive Bayes, Random Forest, SVC is a Linear Support Vector Classifier. Figure produced with code adapted from (Metzen 2023).

No method for calibrating the predicted probabilities was applied to the output

of the models used in this thesis.

4.4. Evaluation Metrics

Choosing a suitable evaluation metric for quantifying model performance is especially crucial for the case of imbalanced classes. It is well known that for example classification accuracy is not a good measure for classification performance when dealing with data having imbalanced classes (He et al. 2013, p. 20), (Aggarwal 2017, p. 26). Evaluating a model's performance is necessary at several points in a machine learning pipeline, and one or several suitable metric(s) should be used at each point. An evaluation of performance is necessary at several stages, including:

- Monitoring for signs of overfitting during training with iterative methods such as gradient descent, and stopping training early to avoid overfitting.[3]
- Model selection, i.e., choosing between different feature sets, model architectures, etc.
- Hyper-parameter tuning, i.e., choosing values for parameters that can not be learned directly from the data.
- Reporting final model performance on a test (holdout) set, to evaluate generalization behavior.

As can be seen from the list above, development and evaluation of a model are tightly coupled, and thus using the right evaluation metric is very important (see also Aggarwal (2017, p. 222)).

The catalog of available metrics for a certain model depends on the type of model, i.e., the type of predictions it can create. Models of types 2 (probabilistic classifiers, see Section 4.2.1) can output scores that can be interpreted as probabilities for class membership. Hence, a brief discussion of metrics and their relation to the case of imbalanced classes is conducted next.

4.4.1. Types of Metrics

He et al. (2013, p. 189) name three different types (or families) of evaluation metrics used for classification:

1. Threshold metrics
2. Ranking methods and metrics

[3] Early stopping is often done by monitoring the loss computed on a validation set and compared to the loss computed on the training data; however, it is also possible to use other metrics such as the ROC-AUC instead of the loss. See Section 6.4.1 for a discussion of early stopping.

3. Probabilistic metrics

The first two will be briefly discussed in the following.

Threshold Metrics

The first type of metrics above require model output in the form of hard class labels, as e.g. obtained from predicted probabilities fed to a decision rule involving a threshold (that must be fixed at this point). The predicted class labels can then be compared to the true class labels (for e.g. a training set), yielding the four fundamental cases (TP, FP, FN, TN) for the binary classification problem that make up the elements of the confusion matrix (Table 1.2.1, repeated here for convenience as Table 4.4.1), as discussed in Section 1.2.

Table 4.4.1.: Confusion matrix for the binary decision problem.

Predicted	Actual	
	Event	Non-Event
Event	TP	FP
Non-Event	FN	TN

The entries of the confusion matrix are counts and can directly be used to compute threshold metrics, such as Precision, Recall (also called Sensitivity) and Specificity (the former two were described in Section 1.2).

Precision and Recall are defined as (see for example (He et al. 2013, pp. 192 f)):

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{4.20}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{4.21}$$

He et al. (2013, pp. 490 f) distinguish between metrics that focus on all classes or a single class, and consider Precision and Recall to be metrics that focus on the positive class only (as they employ relations of the true positives TP to other quantities). Metrics that focus on a single class are more suitable for problems with imbalanced classes when geared towards the target class.

Ranking Methods and Metrics

These metrics allow the evaluation of models independent of a threshold and/or class imbalance ratios and/or error costs (He et al. 2013, pp. 196 f), (Fawcett 2006, pp. 864, 867). They are thus viewed as especially suitable for the case of imbalanced classes.

The most important metrics in this category are:

- Receiver Operating Characteristic (ROC)
- Precision-Recall-curve (PR-curve)

These are briefly discussed in the following sections.

4.4.2. Receiver Operating Characteristic (ROC)

For obtaining the ROC, the quantities True Positive Rate (TPR) and False Positive Rate (FPR) are needed. These are defined as (see for example (He et al. 2013, p. 197)):

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \text{Recall}, \tag{4.22}$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}. \tag{4.23}$$

These metrics are again dependent on a threshold. For constructing a ROC curve, both metrics are first computed for a range of different thresholds, and then for each threshold the pair (TPR, FPR) is plotted as a point of the ROC curve in a plane (Fawcett 2006, p. 866) (see Figure 4.3 for examples). Therefore, the performance of a classifier can be demonstrated for different thresholds, which reflects the behavior for different levels of class-imbalance as well (He et al. 2013, p. 197). The ROC curve is thus a representation of the performance of a classifier decoupled from thresholds and errors costs (Provost et al. 2001, p. 207).

The ROC curve can then for example be used to choose a threshold for classification (assigning hard class labels by a decision rule) for a specific trade-off between TPR and FPR, see e.g. (Kuhn et al. 2013, p. 263).

Area Under the Curve (AUC)

As a means to reduce the ROC curve to a scalar value metric, it is customary to use the area under the ROC curve, called ROC-AUC. The value for ROC-AUC is in $[0, 1]$, however, any meaningful classifier should have a value > 0.5 (Fawcett 2006, p. 868). The ROC-AUC for different settings of a classifier, or for different classifiers, can be used to rank them against each other (He et al. 2013, p. 202). In fact, the use of ROC-AUC is recommended by many authors for comparing classifiers in the case of imbalanced data, e.g. in (He et al. 2013), (Kuhn et al. 2020). However, this recommendation is not unanimous. Fernández et al. (2018, p. 55) mentions that in the case of absolute rarity (small amount of members of the minority class, see also Sections 3.5 and 4.1), the Precision-Recall-curve may be more suitable. He et al. (2013, p. 202) mention some criticism warning against the use of ROC-AUC, for example as it cannot reveal the case of ROC curves crossing each other: a classifier with a higher ROC-AUC can perform worse than another classifier with a lower ROC-AUC in specific regions of the ROC-plane (Fawcett 2006, p. 868). This can be the case when one ROC curve is below another in some region(s), but above it in other region(s), while covering a larger area in total. This case of crossing ROC curves is illustrated in Figure 4.3.[4]

In general, ROC curves are not strongly restricted in their shape. As Duda et al. (2001, p. 50) note, the ROC curve in the general case (arbitrary, possibly

[4]For a further discussion of more properties of the ROC curve and a statistical interpretation see (Fawcett 2006) and (He et al. 2013).

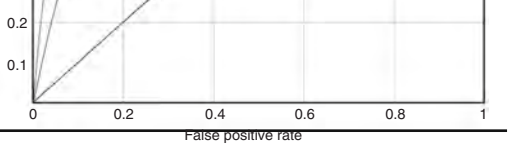

Figure 8.1 The ROC curves for two hypothetical scoring classifiers f_1 and f_2.

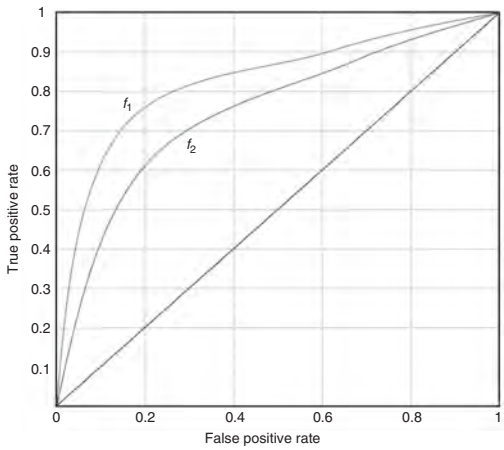

Figure 8.1 The ROC curves for two hypothetical scoring classifiers f_1 and f_2.

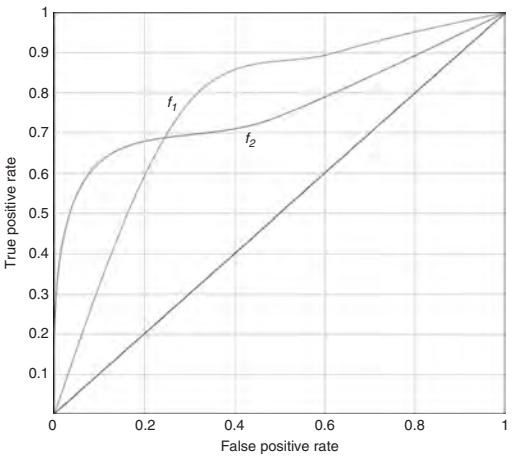

Figure 8.2 The ROC curves for two hypothetical scoring classifiers f_1 and f_2, in which a single classifier is not strictly dominant throughout the operating range.

Figure 4.3.: ROC curves for two (hypothetical) classifiers f_1 and f_2. Left: ROC-AUC$_{f_1}$ > ROC-AUC$_{f_2}$, classifier f_1 dominates f_2. Right: The ROC curves cross, neither classifier dominates the other. While ROC-AUC$_{f_1}$ may be larger than ROC-AUC$_{f_2}$, f_2 shows better performance (higher TPR) than f_1 in regions of low FPR (below about 0.25). Figure from (He et al. 2013, p. 198).

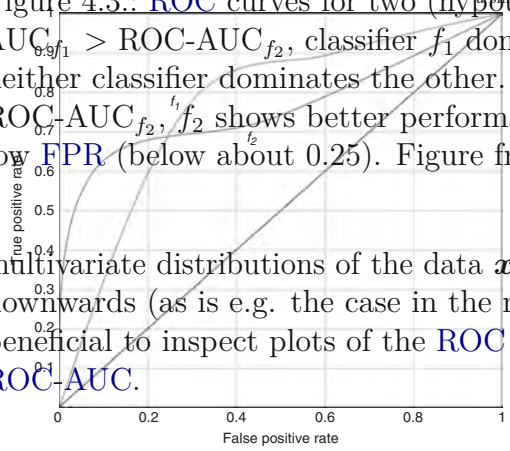

Figure 8.2 The ROC curves for two hypothetical scoring classifiers f_1 and f_2, in which a single classifier is not strictly dominant throughout the operating range.

multivariate distributions of the data $\boldsymbol{x}$) need neither be symmetric nor concave downwards (as is e.g. the case in the right part of Figure 4.3). It may thus be beneficial to inspect plots of the ROC curves when comparing models by their ROC-AUC.

4.4.3. Precision-Recall-Curve (PR-C)

Evaluating Precision and Recall (Equations (4.20) and (4.21)) for different thresholds, similar as for the ROC curve, yields the Precision-Recall-curve (PR-C). Like the ROC curve, the PR-curve as well can be summarized by the area below it, leading to the area under the Precision-Recall-curve PR-AUC.

While it is possible to compare models based on PR-AUC, and optimize for it during model selection, the literature used suggests this only for the case of absolute rarity (as stated above). Furthermore, (He et al. 2013, p. 203) state that PR-AUC is more sensitive to changes in class-imbalance than ROC-AUC. Thus, optimizing for the former may lead to selecting a model that is more sensitive to changes in class-imbalance and may be less robust to changes in the environment the model is used in after training.

Similar to the ROC curve, the PR-curve is not strongly restricted in its shape. For example, (Aggarwal 2017, p. 27) mentions that the PR-curve is not necessarily monotonic and my thus be less intuitive to interpret visually.

There exist certain relationships between the ROC curve and the Precision-Recall-curve.

4.4.4. Relationships between ROC- and PR-curve

A discussion of the relationships between ROC- and PR-curves (for one specific classifier and data set) is given in (Davis et al. 2006), some insights are listed below:

- It is possible to construct the one curve from the other, and identify points on either curve on the other.
- If a classifier dominates another (see Figure 4.3) in either curve, it dominates in the other curve as well.
- Optimizing for ROC-AUC does not necessarily optimize for PR-AUC.

The last point suggests that optimizing for ROC-AUC and PR-AUC jointly, if possible, may be an even better strategy than optimizing for either of the two alone. This path was not followed in this thesis, however.

Section 8.2 briefly explores the actual relationships found for the evaluated experiments by means of visualizations.

CHAPTER 5

Preprocessing

Chapter 3 discussed the data and showed some of their properties. This chapter is concerned with preprocessing the data as necessary for feeding them to suitable models (to be discussed later, Chapter 6) and conducting the experiments. As was stated in the beginning already (Section 1.1), the problem to be solved is framed as a supervised time-series classification task, and the processing steps discussed in the following aim at preparing the data for this task.

5.1. Removing Data

Many models cannot deal with missing data, and using them requires to either remove missing values or fill their spots, e.g. using some form of imputation. Kuhn et al. (2020, p. 196) discuss several aspects of removing data. They distinguish between removing instances (rows) and predictors (columns) when data are missing. Here, however, only removing instances is considered, see below.

5.1.1. Incomplete Data

As was discussed in Section 3.3, for some FNAs, too little data were available when compared to the overall extent of data. In these cases, the whole FNA's data were excluded from further processing. Furthermore, several test-devices not used for actual service of customers were present in the data, and these were completely removed as well. As was already stated in Section 3.3, 2220 FNAs and 21 hubs were present in the data. These numbers already exclude the test-devices.

In total, 66 FNAs' data were removed due to too little data being available.

This includes a complete hub with 4 FNAs, where only roughly 1 month (vs. more than 5 in total) of data were available.

5.1.2. Missing Onsets

As discussed in Section 3.3.2, some FNAs exhibit a pattern where the onset of an increase of the target variable is missing. In total, 69 FNAs exhibit this pattern, however, only 3 of those are not already contained in the 66 with too little date as mentioned in the previous section. These 3 additional FNAs were removed as well.

5.1.3. All-Zero Values or Zero Standard Deviation

293 FNAs have all-zero values for the variables CER and CCER, or standard deviation zero across time, as was discussed in Section 3.4.1. These were also excluded, in addition to the 69 excluded for reasons stated above.

In total, after removing data of 362 FNAs, 1838 out of 2200 FNAs (83.5 %) were kept for further processing, and ultimately for training and evaluating models. Table 5.1.1 summarizes the numbers and reasons for removal of FNAs' data.

Table 5.1.1.: Summary of removed FNAs' data. In total, 362 FNAs' data were removed.

# Removed	Reason
66	too little data
3	missing onset
293	all-zero values
362	

5.2. Transformations

In general, certain (machine learning) models assume or require certain properties of the data, and certain transformations may be necessary to bring the data to meet these requirements. Typical transformations applied to data include some forms of normalization, i.e. compressing the data range to a certain interval, or centering the data and dividing by some (estimate of a) measure of scale. This is covered in the next section under the heading of standardization. Scaling data of all variables to a certain similar range can for example be beneficial for models that use gradient descent methods and can decrease time needed to fit a model, see e.g. (Lakshmanan et al. 2020, pp. 22 f), and can improve the numerical stability of calculations (Kuhn et al. 2013, p. 30).

Other transformations could be removing skewness in the data, or shaping the data such that their distribution becomes more similar to e.g. a (standard) normal distribution. Yet another class of transformations aims at inducing stationarity, a property related to the evolution and change (or lack thereof) of statistical properties of data over time, see e.g. Mills (2019, pp. 20 ff). While more "classical" models applied to time-series data (such as those from the ARIMA family) may more or less strongly rely on assumptions about stationarity and related properties, the models used in this thesis do not require anything related to such properties, and hence they are not further discussed.

As was shown in Section 3.4, the temporal means, standard deviations and skewness differ for the different FNAs, to varying degrees. The following subsections will briefly discuss the necessity to equalize these differences, especially for the models to be applied, as well as methods to achieve this.

In general, care must be taken that parameters used for transforming the data are estimated on training data only, and then applied to all splits of the data, typically the training data itself, validation and test data. This prevents leakage of information from the training data to validation and test data.

5.2.1. Suggested Transformations

In general, some (types of) models may require certain preprocessing steps, others may not. We shall briefly anticipate the (types of) models later used for approaching the Time-Series Classification (TSC) task (see Chapter 6 in particular), to review the suggested or necessary transformations:

- InceptionTime, a Deep Neural Network (DNN): Ismail Fawaz et al. (2020, p. 9) mentions that z-scoring, i.e. subtracting the mean and dividing by the standard deviation, was applied to all time-series data used in their experiments. They cite Bagnall et al. (2017) as suggesting this as a best-practice approach to preprocessing for time-series classification.

- Fully Convolutional Network (FCN)-Long Short-Term Memory (LSTM), a DNN: Karim et al. (2019) used only z-scoring as transformation of the data in their experiments (Karim et al. 2019, p. 9).

- RandOm Convolutional KErnel Transform (ROCKET), the feature extraction part of a DNN: Dempster et al. (2020, p. 1462) state that they assume the input time-series to be standardized to zero mean and standard deviation of one.

- Decision trees: Hastie et al. (2017, p. 352) state that decision trees are invariant under strictly monotone features transformations, including scaling. Furthermore, they are not affected by irrelevant variables, as they perform feature selection as part of their algorithm. A form of boosted regression trees is used together with the ROCKET features.

Note that the last two models mentioned above operate on the output of the third model (ROCKET), so the standardization (z-scoring) is meant to be applied to its output features before feeding them to the classifiers.

5.2.2. Standardization

Summarizing the suggestions for the models used, it is clear that standardization (z-scoring) is a suitable transformation for the data. Thus, z-scoring was applied to all variables across the data set, where the parameters mean and standard deviation were estimated on the respective training part of the data set on a per-FNA level, aiming at removing differences between data stemming from different FNAs.

5.2.3. Skewness

According to Kuhn et al. (2020, p. 31), a skewness of 20 can be considered significant. As can be seen from Figure 3.12 in Section 3.4, several variables exhibit skewness of this magnitude or beyond. Several transformation exist that can remove the skewness in variables, others even aim at shaping a variable's distribution towards another distribution such as the standard normal. The Box-Cox and Yeo-Johnson transformations (see e.g Kuhn et al. (2020)) are examples of the former, while the quantile-transformation is an example of the latter. However, as none of the used models specifically names any such transformation as a requirement, none was applied to the data.

5.3. Windowing

When working with long signals (time series), it is common to split them into smaller segments (blocks, windows) and process them one at a time. Figure 5.1 shows this schematically. This is e.g. very common in signal processing applications. Various terminology is used throughout different communities, such as signal processing, machine learning, time series analysis and statistics, for naming quantities involved in windowing. We will adopt the terms "window length" for the extent of the segments and "stride" for the number of values skipped between two consecutive windows.

The task is to predict the target variable for a certain horizon into the future, given the past of a multivariate time series, as was shown in Figure 3.5 above. This is approached in such a way that the long time series is split into segments, as just described, where each segment is then again split into past values (the state) and the values to be predicted (possibly for the simplified, quantized signal), see the following Figure 5.2. A further simplification is achieved when not the quantized target signal (a simplification from the actual signal) is to

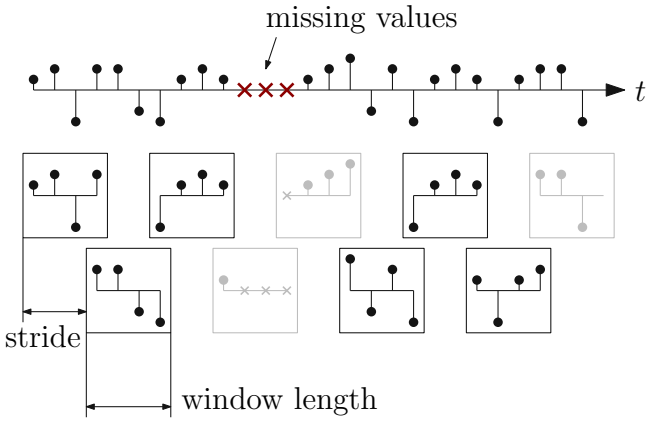

Figure 5.1.: Splitting a long signal into short segments (windows) for processing them one at a time. Parameters involved are the length of the window and the stride, i.e. the amount of values that are skipped between two consecutive windows (here stride = 3). For a stride smaller than the window length, consecutive windows overlap (here 1/4 of the window length). It may not be possible to cover the whole signal with completely filled segments: the rightmost window (gray) is not complete and omitted here; then, values of the signal (three in this example) are left out. Any window containing missing values is ignored.

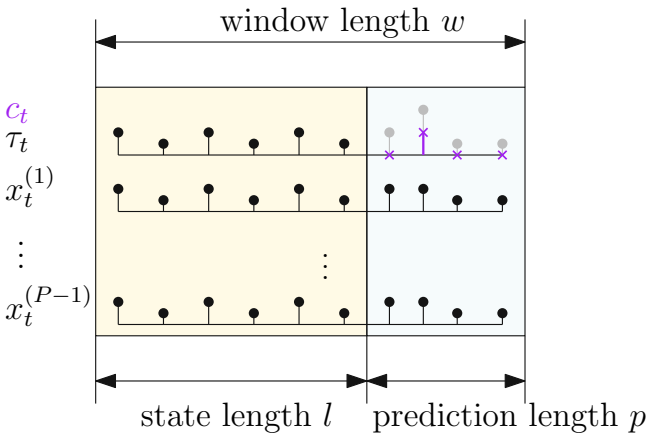

Figure 5.2.: Windows are split into state (considered past values) and prediction parts (values to be predicted). τ_t is the target variable, c_t the quantized version (compare to Figure 3.5). In the example shown, for one time-stamp the target signal is above the threshold.

be predicted, but only whether it crosses the threshold γ at least once in the prediction length p part. This is further discussed and explained in Section 5.4.

The fundamental parameters for the windowing process are thus:

- state length l
- prediction length p
- window length w (sum of the two above)
- stride s

When no periodic structures are present in the data, which we assume to be the case here, using non-overlapping windows may miss important structure in the data (Esling et al. 2012, p. 7), and thus the stride s is always chosen such that two consecutive windows overlap. More concretely, a stride s smaller than half the state length l is used.

The splitting into windows as shown in Figure 5.1 is applied to each FNA's data separately. As described in Section 3.3, data may be missing. The splitting into windows is done such that only completely filled windows are used, and portions of the data with missing values are left out. Again, refer to Figure 5.1 for an illustration.

Depending on the choice of the parameters state length, stride, and prediction length, different cases need to be distinguished, as shown in Figure 5.3.

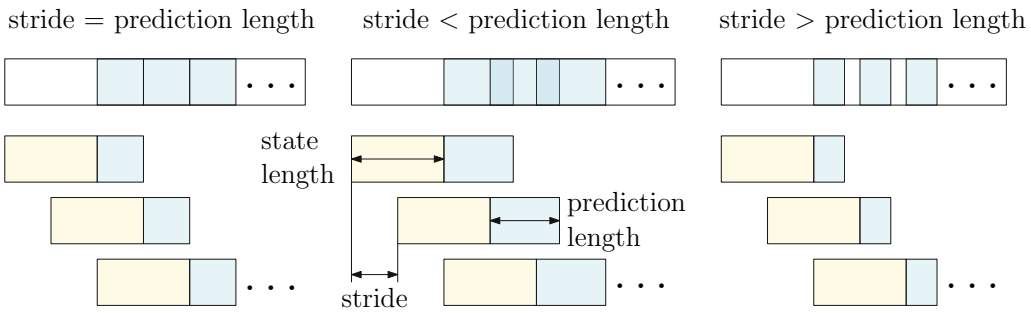

Figure 5.3.: Different relations between stride and prediction length lead to different overlap of prediction parts and covering of the data. See text for details. The leftmost case is used for splitting an FNAs signals so as to cover the whole range of data (except for missing portions and window(s) at the very end).

With appropriate choices of stride and prediction length, the whole extent of a FNA's data can be covered with predictions. This is true for the cases in the left and in the middle of Figure 5.3, where the latter case has overlapping prediction windows.

5.4. Assigning Class Labels

As was already mentioned in Section 3.2, among the recorded variables, one is considered to be a reliable indicator for bad network performance, leading to customer dissatisfaction. Whenever this target variable τ_t reaches or crosses a certain threshold γ_q, bad network performance is to be expected. Figure 5.4 shows a short snippet of the temporal evolution of this variable, for one FNA. Overlaid is the quantized target variable that has value 1 whenever the target variable reaches or crosses the threshold, and is 0 otherwise.

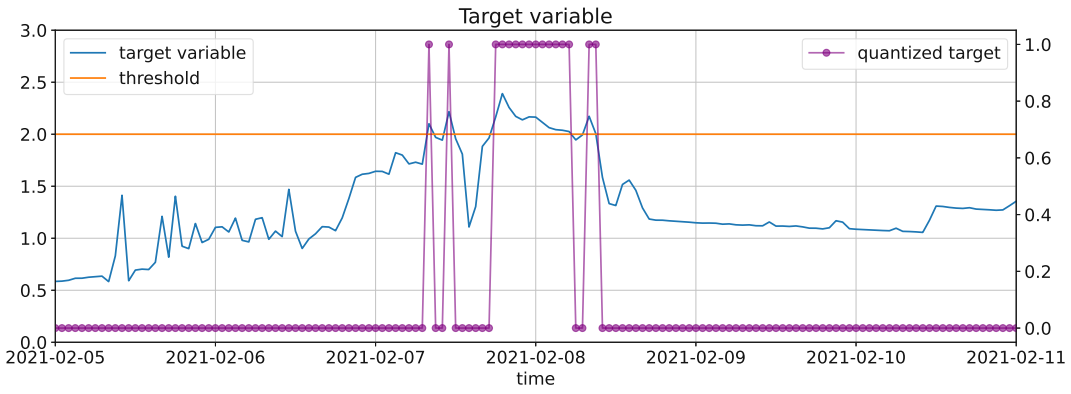

Figure 5.4.: Temporal evolution of the target variable for an FNA, and quantized target variable using a certain threshold. (This Figure is identical to the lower part of Figure 3.6 and is repeated here for convenience.)

The temporal evolution of the target variable in each window's prediction part is used to assign a class label to the corresponding state part. This is illustrated in Figure 5.5. Whenever the quantized target variable c_t has a value of 1 at

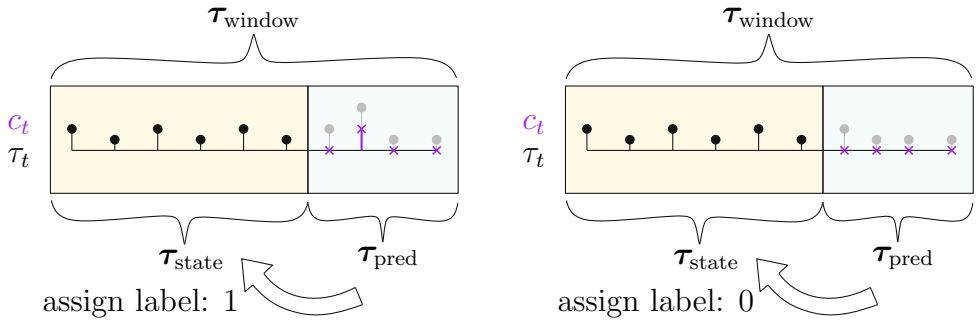

Figure 5.5.: Windows and label assignment. Assigning class labels for the state part of the window based on the quantized target signal c_t. Left: class Event, label is 1. Right, class Non-Event, label 0.

least once in a prediction part, the class label 1 ("event") is assigned to the state block, otherwise class label 0 ("non-event") is assigned. More formally,

if $\boldsymbol{\tau}_{\text{pred}} \in \mathbb{R}^p$ is the prediction part of a window of variable $x_t^{(j)} = \tau_t$, then the function $f : \mathbb{R}^p \to \{0, 1\}$ is:

$$f(\boldsymbol{\tau}_{\text{pred}}) = \begin{cases} 1, & \text{if } \max(\boldsymbol{\tau}_{\text{pred}}) \geq \gamma_q, \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

One consequence of this function, more specifically using $\max(\cdot)$, means that one occurrence of the target reaching or crossing the threshold is enough for assigning class label 1, and that multiple occurrences make no difference for the label assignment. This should be compared to Figures 5.6 and 5.7. The reason for this particular choice is briefly discussed in the next section.

5.5. Class Labels and Network State

When comparing the windows of data to the target signal τ_t, or rather the quantized version c_t, several different cases are possible. These are schematically illustrated in Figure 5.6. Events of c_t crossing the threshold can occur in neither the state nor the prediction horizon, in either of both, or in both. The occurrence of one super-threshold event is sufficient to be considered a "hit" (and consequently assigned class label 1), but several such events may occur in either portion of a window. The task here is to predict the occurrence of such

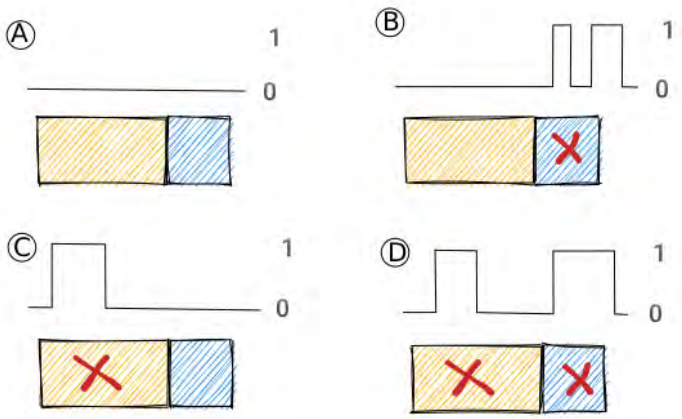

Figure 5.6.: Windows and (quantized) target signal c_t. The occurrence of (at least) one supra-threshold event is marked with a red X. Four cases are possible with respect to the possible occurrence of an event in either the state, the horizon, or both.

an event in the future, i.e., in the prediction part of a window. We are thus not interested in cases C and D: Case C has an event occurring in the state part; we should have predicted this already when the event was in the prediction part of a previous block. Case D has an event occurring in the prediction part, but also in the state part, and what applies to case C applies here as well. For further

processing and usage of data for training of models, only windows of cases A and B are kept, the others discarded.

This choice is meant to put emphasis on detecting the onsets of periods of bad network performance, and ignoring the ability to predict the prolonging of a state of bad network performance by the models to be applied. Furthermore, after some periods of bad performance, mitigating actions may be taken and the network will return to a state of good performance. Since the data used in this thesis contain no information about possible interventions to return the network to a good state, e.g. by maintenance, it did not seem meaningful to keep any windows of case C as shown above; it would not be clear what caused the return to good state.

An example of splitting a time series into windows and the cases discussed above occurring due to quantized the target signal c_t is shown in Figure 5.7.

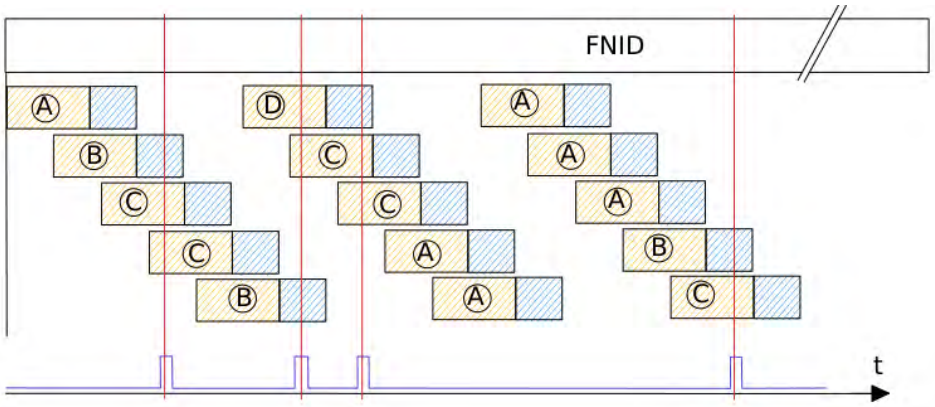

Figure 5.7.: Windows and (quantized) target signal c_t. The occurrence of a supra-threshold event is marked with a red line. This example shows the splitting of data into overlapping windows and occurrences of the four cases shown in Figure 5.6.

5.6. Reshaping Data

The windowing process produces overlapping windows for all settings of windowing parameters used here. While it would be possible to generate windows on the fly, it was decided to prepare the data in advance in appropriate shapes such that the can be loaded and fed directly to the models, without any further processing. This creates redundant data, since overlapping parts of windows are stored at least twice, if not multiple times (depending on the settings). The trade-off here is between time needed for processing the data on the fly and disk space, and eventually longer loading times. Since several different models use the same data, it was decided to include this in preprocessing and store the reshaped data to disk.

5.6.1. 3D-Arrays

For certain models, it is necessary to arrange the windowed data in a certain shape. The multivariate data, depicted as a matrix in Equation (3.3), are windowed as described above and illustrated in Figure 5.2. Each window is itself a matrix.

The windows are arranged into a 3-dimensional array of shape $N \times p \times l$, where n is the number of completely filled windows obtained as discussed in Section 5.3, p is the number of variables, the dimensionality of the data, and l is the length of the state part of a window, i.e. the number of data points in that part. The resulting data structure is schematically depicted in Figure 5.8.

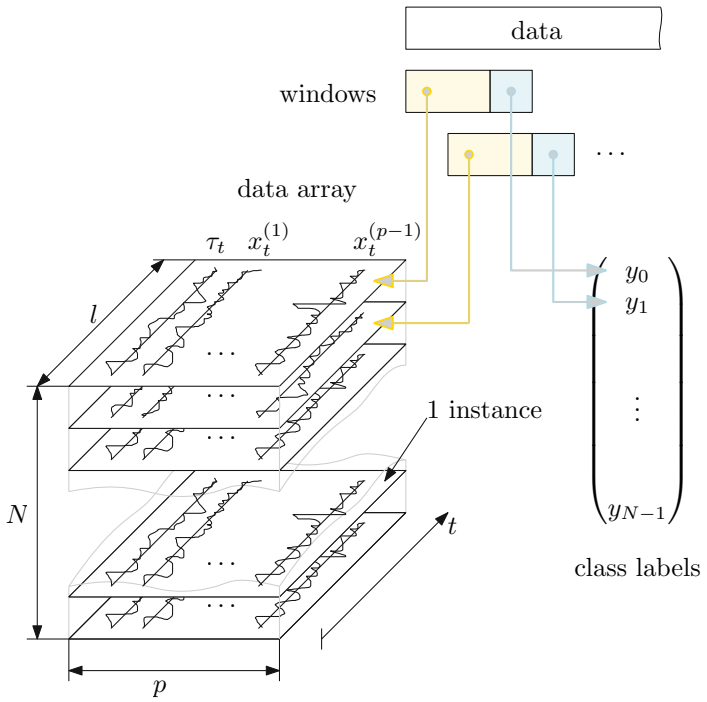

Figure 5.8.: 3D-array structure used for the data after windowing. The associated class labels are collected in a column vector of dimension $N \times 1$. The windowing of the data is shown in Figure 5.3.

The data were stored in this form after all processing was applied. While this requires additional storage capacity due to the redundancy caused by the overlapping windows, it is convenient for the training process, as was mentioned above.

5.7. Data Splitting

Several different ways of splitting data are common for conducting machine learning experiments. Typical approaches are for example splitting the available

data into training, validation and test sets, or using some form of Cross-Validation (CV), possibly with a dedicated test set. k-fold CV is a commonly used method in machine learning, where k usually is chosen as 5 or 10. Splitting data multiple times (as CV does), as opposed to a single splitting, can lead to better performing models (Kuhn et al. 2013, pp. 62, 67) and is thus to favor over a single splitting into training, validation and test sets. A downside of multiple splits is the increased computational effort, as multiple models are fitted and evaluated.

Goodfellow et al. (2016) suggests that CV is useful mostly for smaller data sets (with less than hundreds of thousands of examples), as a performance estimate from a smaller test set implies more statistical uncertainty (Goodfellow et al. 2016, p. 122). Due to the strong imbalance, and a relatively small absolute number of instances of the target class (absolute rarity), the data set used can be considered small, despite the overall larger number of instances. Numbers for instance count and class imbalance are given in Section 5.8 below.

For model selection (see Section 6.6) and tuning of hyper-parameters (see Section 7.4), a 5-fold Cross-Validation (CV) scheme was adopted for the experiments conducted, with an additional dedicated test set split from the data before creating the CV splits. This was mostly a pragmatic choice, balancing between having multiple splits and acceptable computational effort. The splitting of the data is schematically illustrated in Figure 5.9. This figure should be compared to Figures 3.14 and 3.15, showing the number of cases of occurrence of class 1 events over time.

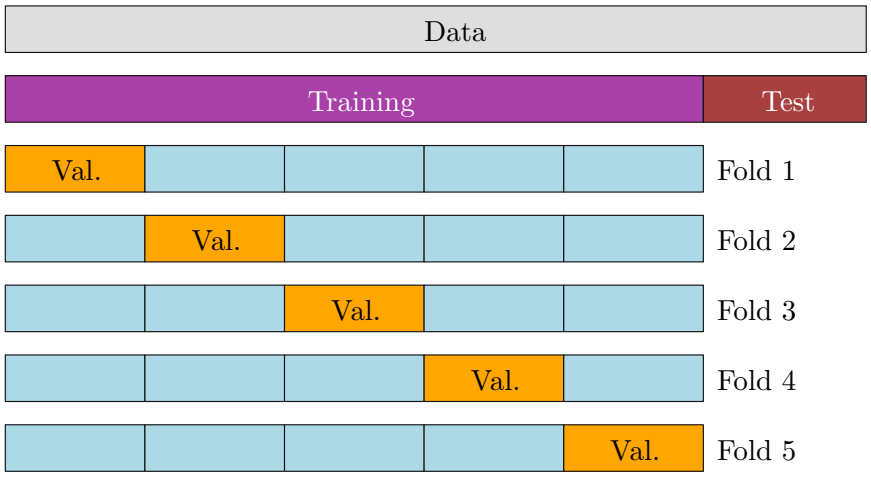

Figure 5.9.: Schema of the data splitting used. The overall range of available data (gray block, top) was split into a block of length 80 % (purple). This part was used for model selection with 5-fold CV, as well as training each final model, respectively. The light blue blocks form the training set, the orange blocks the validation parts of each fold, respectively. The remaining 20 % (brown block) were used as test set for each of the finally selected models.

Depending on the structure of the data, it may be necessary to take precautions

when splitting them into the different subsets. For the case of imbalanced data, He et al. (2013, p. 188) state that care must be taken when using CV to ensure that different folds don't have extremely different Imbalance Ratio (IR). If different folds exhibit strongly different IR, performing the splitting with stratification by the target variable can achieve folds that are (more) balanced with respect to the number of instances of the target in each fold (Kuhn et al. 2013, p. 70).

It was not deemed necessary to use any stratification by the target variable for creating the folds. The folds created for the experiments are examined with respect to this in Section 5.8 below, in particular see Figures 5.11 and 5.10.

5.8. Class Imbalance Ratio after Preprocessing

In Section 3.5, the class Imbalance Ratio (IR) was found to be about 70:1 (class 0 to class 1). The windowing and class label assignment, especially the part incorporating the network state, see Section 5.5, increases (thus worsens) the IR significantly. The settings used for creating the windows have an influence on the number of class 1 events retained in the data: the longer the prediction length, the more occurrences of class 1 events will be lumped into a single class label if they are in sufficient temporal proximity.

5.8.1. Class Imbalance Ratios

Figure 5.10 shows the imbalance ratios for each of the 5 folds generated for the different window size and stride settings. The IR has worsened considerably, from an average 70:1 for the whole data set, to between 500:1 and 1000:1 for the training folds, depending on the windowing settings. Although there doesn't seem to be a consensus on grades or classes of imbalance in the literature, He et al. (2013, p. 15) consider ratios of 1000:1 as extremely imbalanced, as does Krawczyk (2016, p. 224).

There is some spread of the IR for the different folds. When revisiting Figure 3.14 and especially Figure 3.15, the reason is evident: the time-based CV-splitting shown in Figure 5.9 causes the folds to contain quite different amounts of class 1 events. Since the validation parts of the folds have much fewer instances than the training parts (see next section), the spread of count of class 1 instances is larger for the former than for the latter. Furthermore, the median ratio is larger, with the smallest ratio of any validation fold about as large as the smallest ratio of any training fold.

The variability of the IR between the folds is expected to give a good idea of model performance for the obviously variable amount of class 1 events over time, more than a single training, validation and test data split could.

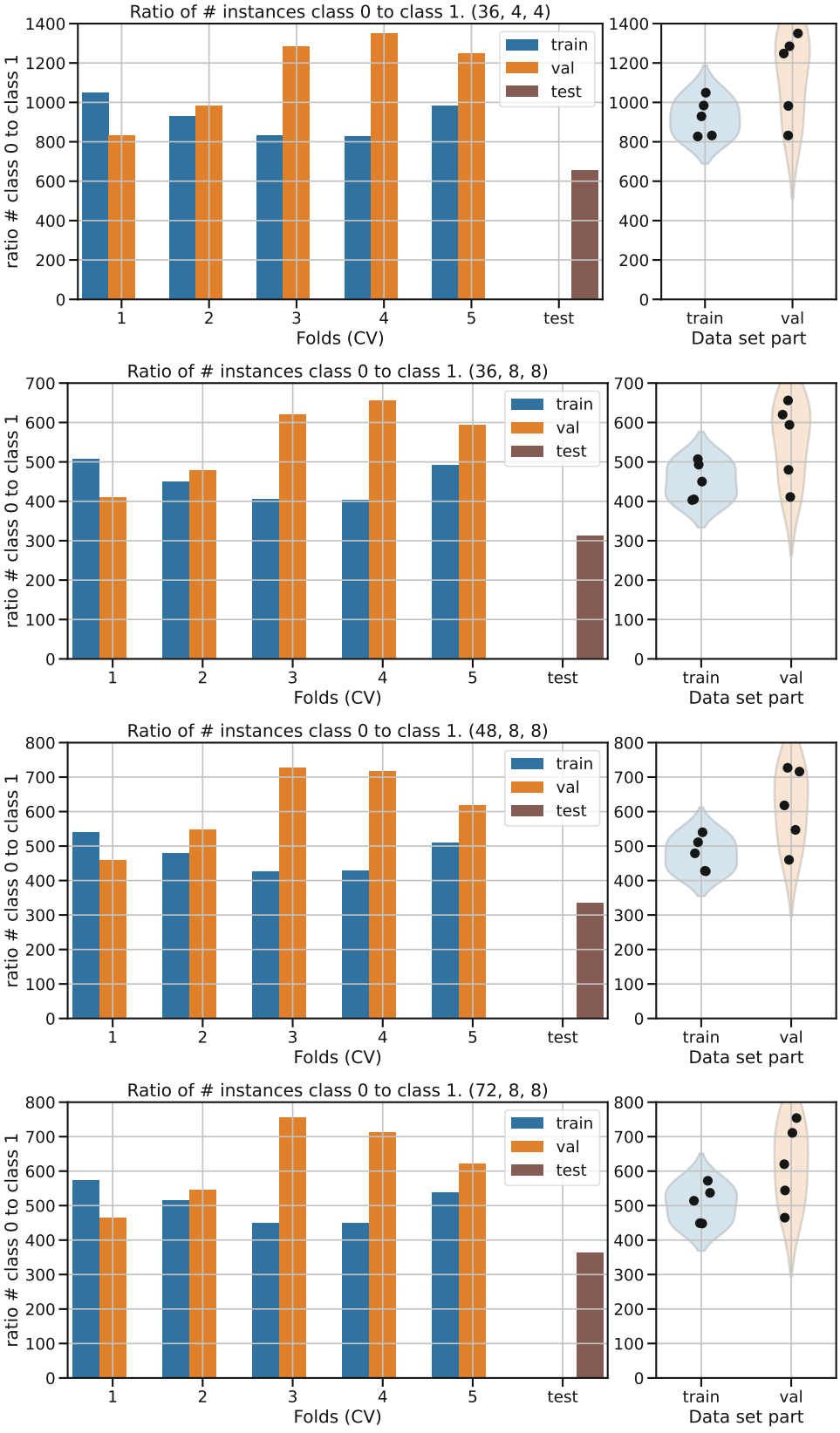

Figure 5.10.: Class imbalance ratios for the different settings, per fold, for training and validation parts as well as the test set split.

5.8.2. Number of Instances

Figure 5.11 shows the number of instances (windows) for each of the 5 folds generated for the different window size and stride settings. As is evident from the plots, the number of instances is mostly influenced by the stride parameter of the windowing process. With smaller stride, more windows are generated for the same time-span (compare the top two plots in Fig. 5.11).

The number of instances does not differ strongly between the folds in each case.

5.8.3. Number of Instances in Target Class

Figure 5.12 shows the number of instances (windows) of the target class (label 1) for each of the 5 folds generated for the different window size and stride settings. The number of instances of the target class does not differ strongly between the folds in each case, and is in a similar range for the different window settings, with roughly at least 1000 instances in each case.

5.9. Processing Pipeline

The processing of the data from the aggregated data discussed in Section 3.1 to the final data set used for the experiments is summarized in the following:

1. Specify session window parameters: state length l, prediction length p and stride s
2. Determine fold boundaries for the folds for 5-fold CV, for each FNA
3. For each fold, determine the windows to be assigned to the training and validation portions, respectively, and extract the windows
4. For the training and validation portions of each fold, estimate the values for mean μ_i and standard-deviation σ_i for each variable $x_t^{(i)}$ from the training part.
5. Standardize the training and validation windows
6. Reshape data as necessary and save data to disk

Additionally, the test set part was processed accordingly.

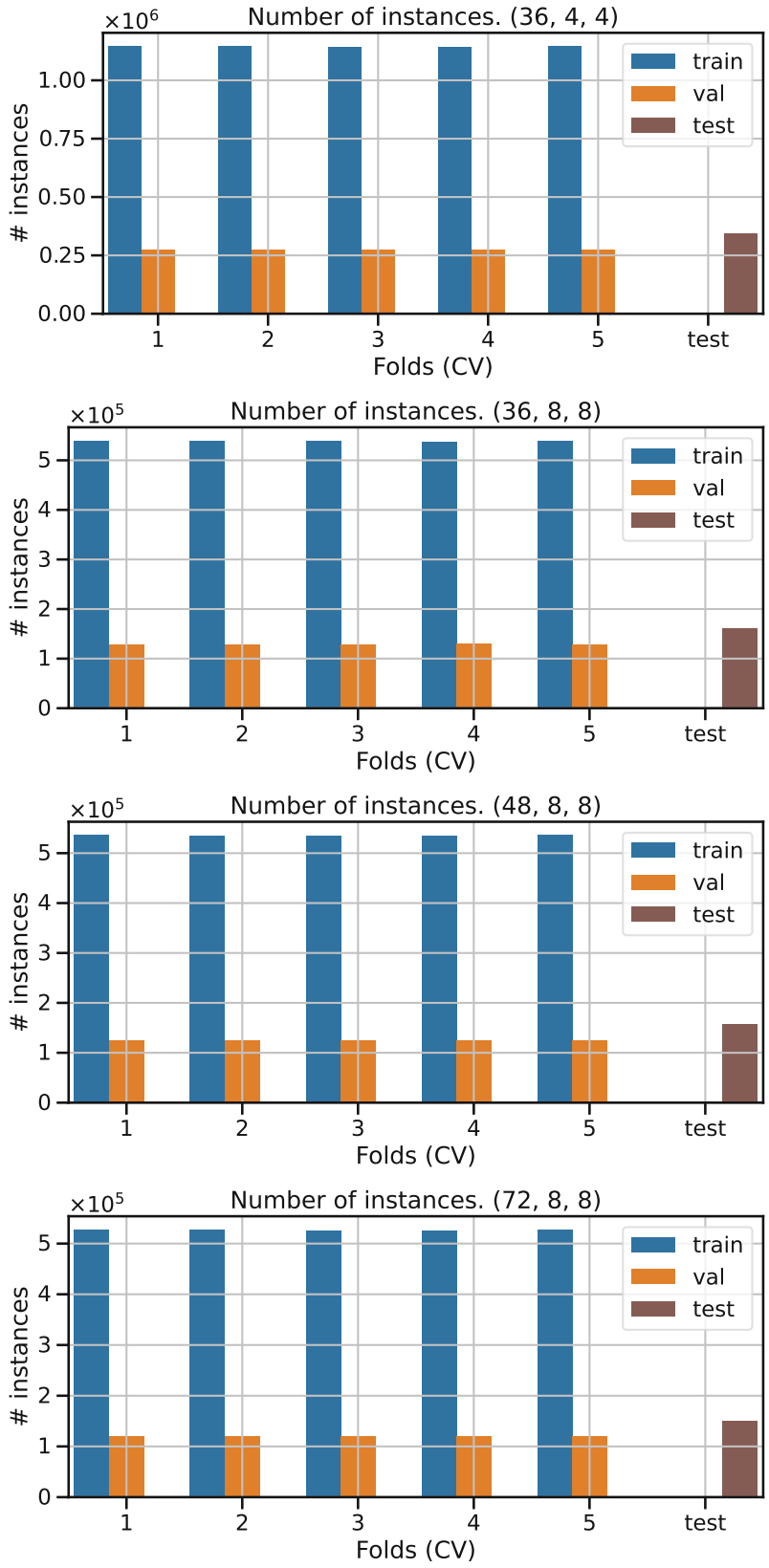

Figure 5.11.: Number of instances (windows) for the different settings, per fold, for training and validation parts as well as the test set split.

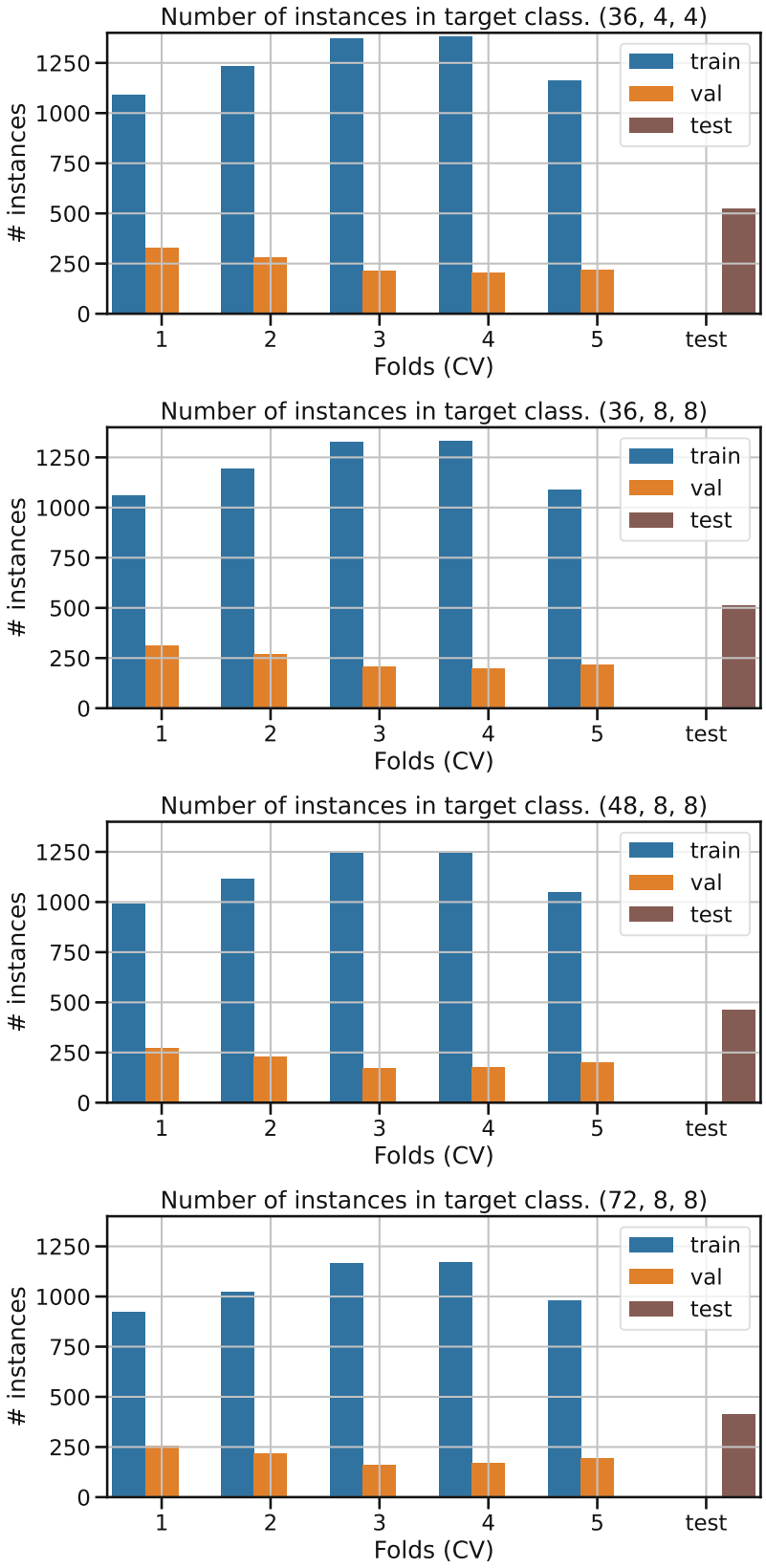

Figure 5.12.: Number of instances (windows) of the target class for the different settings, per fold, for training and validation parts as well as the test set split.

CHAPTER 6

Models

This chapter discusses models suitable for the Time-Series Classification (TSC) task applied to the data described in previous chapters, as well as procedures for training the models, selecting suitable model parameters and hyper-parameters.

Some general requirements for the models used for the given task are given below. The models need to be:

- suitable for time series classification (TSC)
- suitable for multivariate time-series data
- able to handle large amounts of data, thus have acceptable complexity and training time on affordable hardware[1]
- able to work with (heavily) imbalanced classes
- known for good expected classification performance

6.1. Time-Series Classification (TSC)

TSC is the mapping of a given time-series instance to either one class label from a finite set of labels, or to a probability distribution over the classes.

In the literature, often a distinction is made between models tackling classification of univariate time-series or multivariate time-series. The latter case is often explicitly named Multivariate Time-Series Classification (MTSC). In the univariate case, an important aspect is the similarity of the data with itself along the time-axis, as e.g. expressed in the auto-correlation function. In the

[1] "Affordable hardware" is a bit of a clumsy term. See a discussion of the hardware used for the experiments in Section 7.1.

multivariate case, additional complexity can arise from the (possible) relationships of the component individual univariate time-series among each other (Ruiz et al. 2021, p. 403), as e.g. captured by cross-correlation functions.

Not all models are suitable for multivariate time-series data. However, many models designed for univariate data can be adapted to multivariate data. A simple approach to Multivariate Time-Series Classification (MTSC) is to use an algorithm suited to (univariate) TSC and apply it to each variable separately, afterwards combining outputs to an ensemble classifier (Ruiz et al. 2021, p. 401). This approach, however, assumes independence between the component univariate time-series (Ruiz et al. 2021, p. 403) and may thus discard information relevant to the classification task.

Since the data used in this thesis are multivariate, models capable of handling these (preferably per design) are necessary. Many "classical" machine learning methods for Time-Series Classification (TSC) exist. However, most of the classical models showing state of the art performance have very high training complexity. One example is the HIVE-COTE algorithm that has training complexity $\mathcal{O}(n^2\, l^4)$, where n is the number of training instances and l the length of the time-series of the instances (Ismail Fawaz et al. 2020, p. 1939). This complexity can be prohibitively large for big data sets, long time-series, and especially when tuning of hyper-parameters is necessary.

6.2. Deep Neural Networks (DNNs)

In recent years, Deep Neural Networks (DNNs) have shown to be able to beat the previous state of the art in many fields of machine learning. This is also the case for time-series classification tasks, see e.g. (Ismail Fawaz et al. 2020), (Karim et al. 2018). A part of the success of such networks was the availability (or rather, development) of software to utilize commodity Graphics Processing Units (GPUs) for the computations involved.

The most important type of DNN are Convolutional Neural Network (CNN), discussed in the following section.

6.2.1. Convolutional Neural Networks (CNNs)

The typical structure of a CNN used for image classification[2] is shown in Figure 6.1. The lower part of the network is responsible for feature learning (during training) and extraction (after training, for inference). The top part, built from a Multi-Layer Perceptron (MLP), functions as a (linear) classifier.

At the heart of Convolutional Neural Networks (CNNs) is the convolution of input data with a (possibly large) number of kernels (often also referred

[2]We mention CNN as applied to image classification here as they were at the forefront of the modern deep learning revolution, and many models used in other domains, including Time-Series Classification (TSC), were at least in part derived from these.

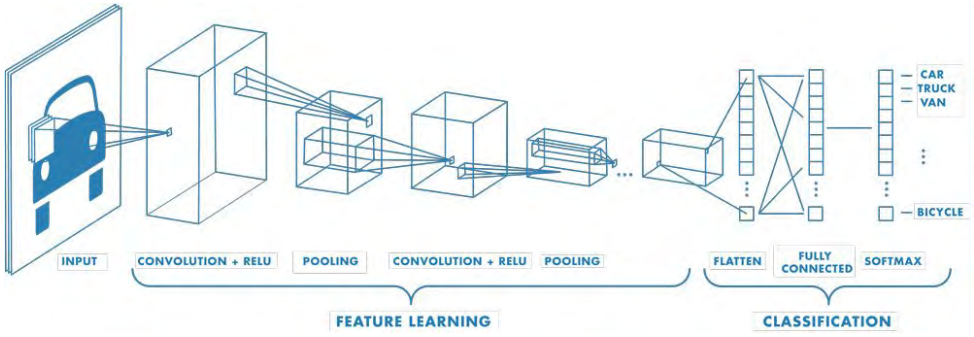

Figure 6.1.: Basic structure of a Convolutional Neural Network (CNN) for image classification. Figure from (Mathworks 2022)

to as filters). Kernels are sequences (vectors) in the case of one-dimensional data, matrices in the two-dimensional case, or n-dimensional arrays (often called tensors) in more general settings.

For example, the discrete time 1-dimensional convolution operation of a time-series x_t (input data) and a (causal) kernel sequence k_t of finite length l_k is:

$$(x * k)(t) = \sum_{j=0}^{l_k-1} x_{t-j}\, k_j = \sum_{j=0}^{l_k-1} x_t\, k_{t-j} = (k * x)(t). \tag{6.1}$$

Usually, further operations are applied to the result of the convolution, such as adding a bias value, and then passing this result through a so-called activation function, e.g. a Rectified Linear Unit (ReLU) in the example in Figure 6.1. The outcome final is often referred to as a feature map (Goodfellow et al. 2016, p. 332).

Typically, a so-called convolution (or convolutional) layer, that is viewed as a building block of a CNN, applies several (e.g. as many as 256) kernels of the same dimension to a given input, resulting in many feature maps (one per kernel). In Figure 6.1, this is shown by depicting the convolution layers (marked "Convolution + ReLU") as boxes. The dimension perpendicular to the input image plane (the input image is the car symbol on the far left) represents the number of feature maps resulting from the convolutions with several kernels. Note that the boxes deeper in the network (when going from input, left, to output, right) get smaller in their front surface area, but gain depth, i.e., have more feature maps. This is not uncommon for CNN architectures.

It shall be noted that the terminology and jargon used for describing the architecture of Convolutional Neural Network (CNN) doesn't seem to be very consistent throughout the literature. As Goodfellow et al. (2016, pp. 339 ff) point out, for example, what is considered a convolutional layer of a CNN can vary, where some may view convolution, an activation function and pooling operation as three separate layers, while others refer to these as stages of a single convolutional layer. Yet, in Figure 6.1, convolution and activation function are considered a convolution layer and pooling a separate layer (or stage).

In general, the concrete implementation and the parametrization of the convolution operations may be more involved than is shown in Equation (6.1), as the range over which the products are summed may differ (padding of the edges), as may the stride, i.e. amount of values the kernel "hops" when moving between values of the input data. A very good overview of convolution arithmetic as used in Convolutional Neural Networks (CNNs) is given in (Dumoulin et al. 2018) with many excellent graphical representations. See also the corresponding code repository (Dumoulin et al. 2019).

The convolution operation is commutative in the sense shown in Equation 6.1. As stated by Goodfellow et al. (2016, pp. 332 f), this commutativity is usually not important for the implementation of CNNs, and thus, often the cross-correlation is used, and can be found in the literature instead of convolution.

Finally, convolution operations can be implemented using products between tensors (as a generalization of vectors and matrices). The one-dimensional convolution shown in Equation (6.1) for example can be implemented as a product between a vector and a matrix of special structure, a Toeplitz matrix, see e.g. (Goodfellow et al. 2016, p. 333).

The key idea of CNNs is to learn the kernels (i.e., the entries of vectors or matrices) during supervised training, together with the weights and biases in the layers of the MLP classifier. The convolution of the kernels (whose values are fixed after training) with the input data is, together with possible further operations, and passing results through activation functions, considered as a form of feature extraction.

This built-in feature extraction, or rather the feature learning during supervised training, is viewed as a major advantage of Deep Neural Networks (DNNs): hand-crafted features and feature selection strategies are not longer necessary, as the features relevant to the task can be learned from the input data directly. Furthermore, the complete process of estimating kernel parameters for feature extraction from raw input data, as well as training a classifier on top of the features, can be optimized together during training. This is usually achieved by using the back-propagation algorithm with a form of gradient descent optimization in a supervised manner (see e.g. (Goodfellow et al. 2016)). Typically, this requires large amounts of labeled training data.

We shall briefly review some more important terms related to a Convolutional Neural Network (CNN) that occur in Figure 6.1:

- Rectified Linear Unit (ReLU), a type of activation function. Activation functions are often applied to the output of a layer, here to the output of a convolution operation. Many activation functions used in deep learning models are nonlinear functions, and the non-linearity is in fact crucial to the performance (see for example (Goodfellow et al. 2016, p. 226)).

- Pooling: a pooling operation computes a summary statistic of its input. As the figure shows, the pooling operation summarizes a certain volume,

or neighborhood, (a part of the 3D-array) into a single value. This summarization is meant to make learned representations of the input data invariant to translations (shifts) in one or several dimensions. For images, this relates to positions of (parts of) objects in an image, in the time-series case this relates to shifts of a pattern along the time-axis. Typical statistics employed are taking the max (max pooling) or the mean (average pooling) (Goodfellow et al. 2016, pp. 339 ff). Note that while pooling is crucial for CNNs used in the image domain, mainly for computational reasons, they can be hindering in the domain of time-series (Ismail Fawaz et al. 2020, p. 1940).

- Flatten: this is an operation that reshapes a multidimensional array into a one-dimensional array (a vector), such that it can be fed into a Multi-Layer Perceptron (MLP).

- Fully Connected: this refers to the layers of an MLP network (serving as a classifier here), where all units are interconnected.

- Softmax: this is a generalization of the logit function. Softmax can be thought of as a special version of the arg max function, where the word "soft" refers to the fact that the softmax is continuous and differentiable (Goodfellow et al. 2016, p. 187).

6.3. CNNs for Time-Series Classification

The basic structure and ideas shown in Figure 6.1 are also valid for, and applicable to, 1-dimensional input data such as time-series, and extensions to multivariate time-series are possible. Figure 6.2 shows in a schematic way the general structure of a Deep Neural Network (DNN) that uses the (possibly) multivariate time-series directly as input.

An advantage that models for time-series have over those for image processing is that convolutions need to be carried out only over one dimension instead of two, which can either save computational power or allow for more complex models given a certain amount of computational power budget. The decreased complexity due to the lower-dimensional convolution allows for removing pooling operations, which benefits performance in TSC tasks (Ismail Fawaz et al. 2020, p. 1940).

Several DNN architectures, i.e. the way how layers are organized and stacked to create the network, have been proposed specifically for Time-Series Classification (TSC) and Multivariate Time-Series Classification (MTSC) tasks. Recent models that yield good performance can be found in the literature, see e.g. (Ruiz et al. 2021), (Ismail Fawaz et al. 2019) for reviews that include quite recent deep learning models and compare their performances on various data sets.

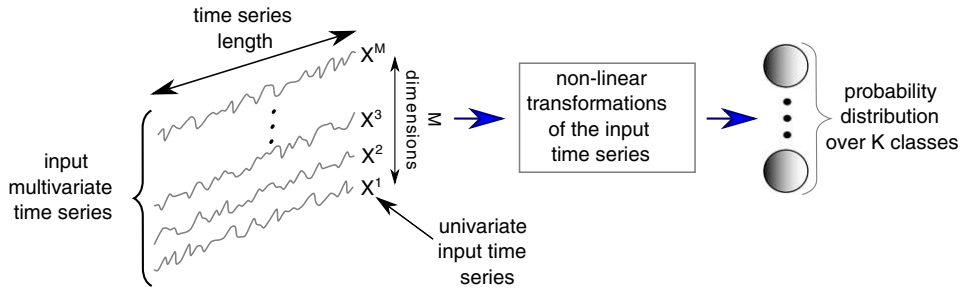

Fig. 1 A unified deep learning framework for time series classification

Figure 6.2.: General schematic structure of a Deep Neural Network (DNN) for Time-Series Classification (TSC). On the left, the input data instance is a multivariate time-series (in the general case) (compare to the shape of the data in Figure 5.8). The data are subject to non-linear transformations, and the output is a probability (score) distribution over the possible classes. Figure from (Ismail Fawaz et al. 2019, p. 921).

2 Background

In this section, we start by introducing the necessary definitions for ease of understanding. We then follow by an extensive theoretical background on training DNNs for the TSC task. Finally we present our proposed taxonomy of the different DNNs with examples of their application in various real world data mining problems.

Bagnall et al. (2017) covers recent "classical" models for TSC and compares their performances.

2.1 Time series classification

Before introducing the different types of neural networks architectures, we go through some formal definitions for TSC.

Definition 1 A univariate time series $X = [x_1, x_2, \ldots, x_T]$ is an ordered set of real values. The length of X is equal to the number of real values T.

Definition 2 A M-dimensional MTS $X = [X^1, X^2, \ldots, X^M]$ consists of M different univariate time series with $X^i \in \mathbb{R}^T$.

Definition 3 A dataset $D = \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_N, Y_N)\}$ is a collection of pairs (X_i, Y_i) where X_i could either be a univariate or multivariate time series with Y_i as its corresponding one-hot label vector. For a dataset containing K classes, the one-hot label vector Y_i is a vector of length K where each element $j \in [1, K]$ is equal to 1 if the class of X_i is j and 0 otherwise.

The task of TSC consists of training a classifier on a dataset D in order to map from the space of possible inputs to a probability distribution over the class variable values (labels).

2.2 Deep learning for time series classification

In this review, we focus on the TSC task (Bagnall et al. 2017) using DNNs which are considered complex machine learning models (LeCun et al. 2015). A general deep learning framework for TSC is depicted in Fig. 1. These networks are designed to learn hierarchical representations of the data. A deep neural network is a composition of L parametric functions referred to as layers where each layer is considered a representation of the input domain (Papernot and McDaniel 2018). One layer l_i, such

Before introducing the different types of neural networks architectures, we go through some formal definitions for TSC.

Definition 1 A univariate time series $X = [x_1, x_2, \ldots, x_T]$ is an ordered set of real values. The length of X is equal to the number of real values T.

Definition 2 A M-dimensional MTS $X = [X^1, X^2, \ldots, X^M]$ consists of M different univariate time series with $X^i \in \mathbb{R}^T$.

Definition 3 A dataset $D = \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_N, Y_N)\}$ is a collection of pairs (X_i, Y_i) where X_i could either be a univariate or multivariate time series with Y_i as its corresponding one-hot label vector. For a dataset containing K classes, the one-hot label vector Y_i is a vector of length K where each element $j \in [1, K]$ is equal to 1 if the class of X_i is j and 0 otherwise.

The task of TSC consists of training a classifier on a dataset D in order to map from the space of possible inputs to a probability distribution over the class variable values (labels).

2.2 Deep learning for time series classification

In this review, we focus on the TSC task (Bagnall et al. 2017) using DNNs which are considered complex machine learning models (LeCun et al. 2015). A general deep learning framework for TSC is depicted in Fig. 1. These networks are designed to learn hierarchical representations of the data. A deep neural network is a composition of L parametric functions referred to as layers where each layer is considered a representation of the input domain (Papernot and McDaniel 2018). One layer l_i, such

The list of possibly useful architectures for TSC and MTSC is quite long when including classical models, but not many deep learning architectures for (multivariate) Time-Series Classification (TSC) exist yet. The following models were on the one hand chosen based on their performance, as documented in (Ruiz et al. 2021), and the respective publications introducing the models (see list below), and the availability of implementations of these networks[3] on the other hand:

- InceptionTime (Ismail Fawaz et al. 2020), an adaptation of the concepts of Inception networks and Residual networks to time series data.
- Fully Convolutional Network (FCN) with (Multivariate) Long Short-Term Memory (LSTM) (Karim et al. 2018) and (Karim et al. 2019).
- RandOm Convolutional KErnel Transform (ROCKET) (Dempster et al. 2020), an architecture used for feature extraction using 1-dimensional kernels, to be used with a separate classifier.

Each of these models will be briefly discussed in the following, to give an idea of how the data are processed. However, no deeper discussion of the architectures is given, as the models were basically used "as-is" from the respective library (implementation), and only a few hyper-parameters were optimized during training (see Section 7.4).

[3] The models were already briefly mentioned in Section 5.2.1 in the context of necessary preprocessing.

6.3.1. InceptionTime

The InceptionTime architecture was introduced in Ismail Fawaz et al. (2020). It follows the ideas of the so-called Inception architectures, and builds on the ideas of Residual Network (ResNet) architectures. Inception architectures are built from (typically several) so-called Inception modules that apply convolutions with kernels of different sizes. A discussion of Inception architectures (for the image recognition domain) and their combination with residual connections can be found in (Szegedy et al. 2016).

The basic architecture of the InceptionTime model is shown in Figure 6.3. The

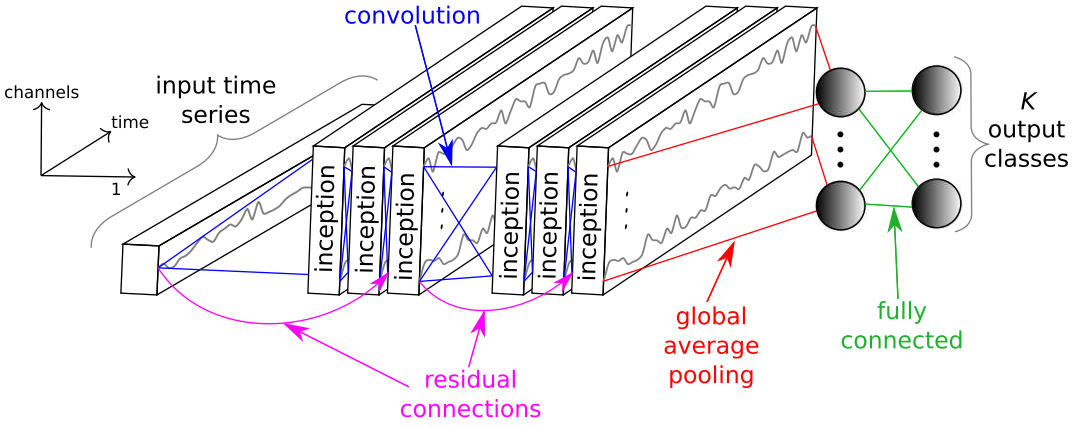

Fig. 1 Our Inception network for time series classification

Figure 6.3.: Basic structure of the InceptionTime architecture. See text for description. Figure from (Ismail Fawaz et al. 2020)

light flux of a region in space as an input MTS for the network (Brunel et al. 2019). However, the authors limited the conception of their Inception architecture to the one proposed by Google for ImageNet (Szegedy et al. 2017). In our work, we explore much larger filters than any previously proposed network for TSC in order to reach state-of-the-art performance on the UCR benchmark.

model contains several Inception modules specialized for time series data. The model shown contains six such modules (labeled with "inception"), where three modules form an Inception block. Between each block of Inception modules, a residual connection is placed. These residual connections were introduced to combat the vanishing gradient problem that occurs in very deep networks with many layers (Ismail Fawaz et al. 2020, pp. 1941 ff). Following the Inception blocks, Global Average Pooling (GAP) is applied which averages the multivariate time series (the output of the last Inception module) along the time axis, but keeps the other dimension. The last element in the architecture again is a single fully connected layer, with as many output neurons as there are classes necessary for the problem. Together with a softmax function, this allows to model the distribution over classes (see discussion in Section 4.2.1 about discriminative models) (Ismail Fawaz et al. 2020, p. 1941).

3 InceptionTime: an accurate and scalable time series classifier

In this section, we start by describing the proposed architecture we call InceptionTime for classifying time series data. Specifically, we detail the main component of our network: the Inception module. We then present our proposed model InceptionTime which consists of an ensemble of 5 different Inception networks initialized randomly. Finally, we adapt the concept of Receptive Field for time series data.

Inception Module

3.1 Inception Network: a novel architecture for TSC

The Inception Module is the core part of the model, as was shown in Figure 6.3. The composition of an Inception network is given in Figure 6.4. *Two* different residual blocks, as opposed to ResNet, which is comprised of *three*. For the Inception network, each block is comprised of three Inception modules rather than traditional fully convolutional layers. Each residual block's input is transferred via a shortcut linear connection to be added to the next block's input, thus mitigating the vanishing gradient problem by allowing a direct flow of the gradient (He et al. 2016). Following these residual blocks, we employed a Global Average Pooling (GAP) layer that averages the output multivariate time series over the whole time dimension. At last, we used a final traditional fully-connected softmax layer with a number of neurons equal to the number of classes in the dataset. Figure 1 depicts an Inception network's architecture showing 6 different Inception modules stacked one after the other.

of the module and its components as described in Ismail Fawaz et al. (2020, pp. 1941 ff) is given in the following:[4]

- Input data: the input is a multivariate time-series of some length l and with M components. Note that this is the very first inception module in the network, as shown in the figure; for the other modules, the input Multivariate Time Series (MTS) is of a different dimension, given by the output of the previous module or block.

- Bottleneck: m filters of length 1 are applied to the input MTS, with stride 1, thus covering the whole MTS along the time-axis. Each filter yields one of m component time-series. For $m \ll M$, this amounts to a (strong) dimensionality reduction. See also Figure 6.5 for an additional depiction of the bottleneck layer.

- To each of the m components, 3 kernels of different lengths (per default: 10, 20, and 40 values) are applied, each forming a new time-series component. Including kernels of different sizes into the module is meant to allow the network to extract features at different time scales, thereby enabling the handling of input time-series of different lengths (Ismail Fawaz et al. 2020, p. 1937).

- In parallel to the bottleneck and following convolutions just described, a sliding window of length 3 with a MaxPooling operation (taking the maximum value of the values within the window) is applied to the input MTS. This is followed by another (different) bottleneck layer, and a convolution with a kernel of length 1, again yielding a component time-series.

- As can be seen in the figure, for each component m, the 3 output time-series (obtained by applying 3 different kernels) are stacked together with the result of the MaxPooling and convolution operation, thus yielding 4 component time-series for each of the m components.

For their experiments, Ismail Fawaz et al. (2020) used an ensemble of five InceptionTime models. This is partially due to the small data set sizes (in relation to what is commonly used for deep learning methods) they used to show the performance of their model (Ismail Fawaz et al. 2020, p. 1943). Their ensemble was created from applying the same architecture to the same data with different initializations. In this thesis, however, no ensemble but only a single instance of the InceptionTime architecture is used, justified by the larger data set size.

To very briefly summarize the network architecture (mostly the part relevant for feature learning and extraction):

[4] The reader is encouraged to compare Figures 1 and 2 in (Crocker et al. 2021) for additional visualizations of a concrete example of an InceptionTime network.

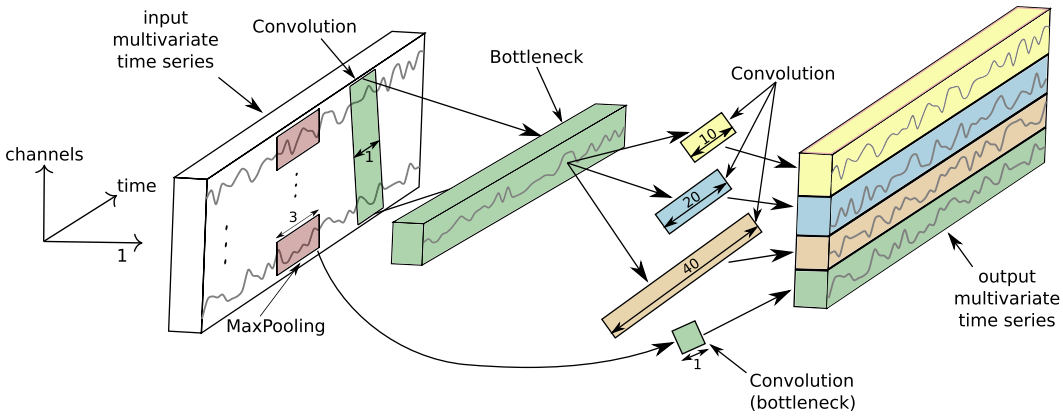

Fig. 2 Inside our Inception module for time series classification. For simplicity we illustrate a bottleneck layer of size $m = 1$

Figure 6.4.: Structure of one Inception module of the InceptionTime architecture. In the center, the bottleneck layer reduces the dimensionality of the input (dimensionality M) to $m \ll M$, here $m = 1$ in particular. Note the three kernels of size 10, 20 and 40, respectively, as well as the second bottleneck followed by a convolution of size 1 that is applied to a max pooled sliding window of length 3. See text for further details, and compare to Figure 6.5 for an illustration of the bottleneck layer for $m > 1$. Figure from (Ismail Fawaz et al. 2020).

of sliding m filters of length 1 with a stride equal to 1. This will transform the time series from an MTS with M dimensions to an MTS with $m \ll M$ dimensions, thus reducing significantly the dimensionality of the time series as well as the model's complexity and mitigating overfitting problems for small datasets. Note that for visualization purposes, Fig. 2 illustrates a bottleneck layer with $m = 1$. Finally, we should mention that this bottleneck technique allows the Inception network to have much longer filters than ResNet (almost ten times) with roughly the same number of parameters to be learned, since without the bottleneck layer, the filters will have M dimensions compared to $m \ll M$ when using the bottleneck layer. The second major component of the Inception module is sliding multiple filters of different lengths simultaneously on the same input time series. For example in Fig. 2, three different convolutions with length $l \in \{10, 20, 40\}$ are applied to the input MTS, which is technically the output of the bottleneck layer. Additionally, in order to make our model invariant to small perturbations, we introduce another parallel MaxPooling operation, followed by a bottleneck layer to reduce the dimensionality. The output of sliding a MaxPooling window is computed by taking the maximum value in this given window of time series. Finally, the output of each independent parallel convolution/MaxPooling is concatenated to form the output MTS. The latter operations are repeated for each individual Inception module of the proposed network.

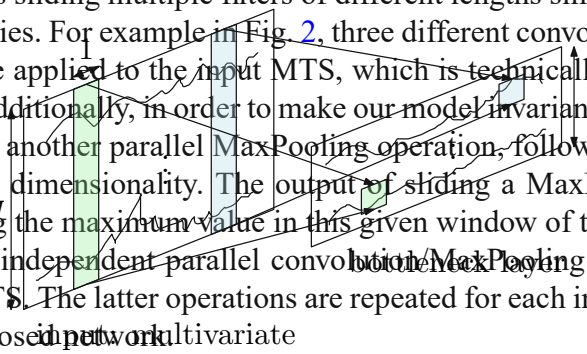

Figure 6.5.: Detail of the bottleneck layer of the Inception module (first stage, acting on the input data). There are m filters, yielding m component time-series. For illustrative purposes, two (of m) filters are shown at different positions along the time-axis. Compare to Figure 6.4.

By stacking multiple Inception modules and training the weights (filters' values) via backpropagation, the network is able to extract latent hierarchical features of multiple resolutions thanks to the use of filters with various lengths. For completeness, we specify the exact number of filters for our proposed Inception module: 3 sets of filters each with 32 filters of length $l \in \{10, 20, 40\}$ with MaxPooling added to the mix, thus making the total number of filters per layer equal to $32 \times 4 = 128 = M$ - the dimensionality of the output MTS. The default bottleneck size value was set to $m = 32$.

- Dimensionality reduction by using bottleneck layers.
- Multiple kernels of different lengths (inside the Inception modules) to allow for input time-series of different lengths, as well as to capture patterns of different temporal extent.
- Residual connections to prevent the vanishing gradient problem to occur.
- Output of probability distribution over classes for each instance.

Implementation

The implementation used in this thesis was taken from the library `tsai` (Oguiza 2020a), where the `InceptionTimePlus` variant was used (Oguiza 2020c).

6.3.2. FCN-MLSTM

The Fully Convolutional Network (FCN) with Multivariate Long Short-Term Memory (MLSTM) model was introduced in (Karim et al. 2018) for univariate data, (Karim et al. 2019) brought its multivariate extension.

An important difference of a Fully Convolutional Network (FCN) to other Convolutional Neural Networks (CNNs) is that the former do not contain any pooling layers (Ismail Fawaz et al. 2019, p. 933). As was briefly mentioned above, pooling layers can actual hinder the performance of CNNs when applying them to time-series data, and hence, FCNs were developed (Ismail Fawaz et al. 2020, p. 1940).

The architecture is depicted in Figure 6.6. The most important components

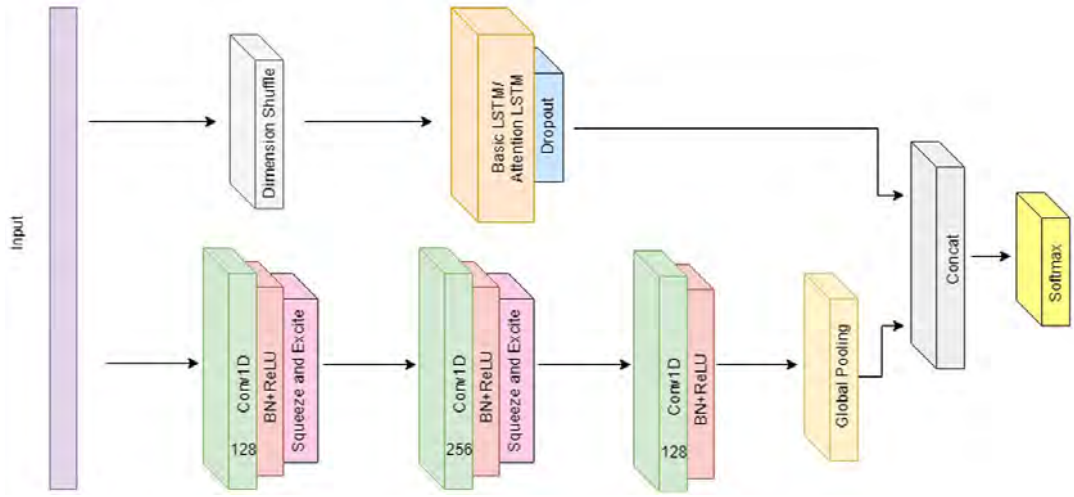

Fig. 1. The MLSTM-FCN architecture. LSTM cells can be replaced by Attention LSTM cells to construct the MALSTM-FCN architecture.

Figure 6.6.: Overview of the architecture of the Multivariate Long Short-Term Memory (MLSTM) - Fully Convolutional Network (FCN) network. See text for description of the elements. Figure from (Karim et al. 2019).

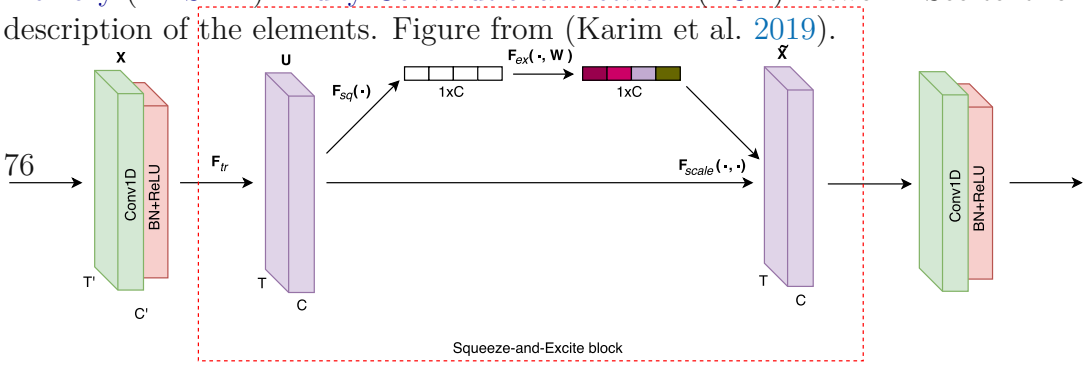

Fig. 2. The computation of the temporal *squeeze-and-excite* block.

will be briefly described, following (Karim et al. 2019). As can be seen, the network is formed from two separate branches that are combined before a final softmax layer:

1. The upper branch with the dimension shuffle element and an LSTM with dropout regularization (see Section 6.4.1 for a short description of dropout).

2. The lower branch holds several blocks, each consisting of a convolution layer followed by a Batch Normalization (BN) layer and a squeeze and excite block (see below and Figure 6.7). Note the absence of the squeeze and excite block from the very last block of layers. Following the convolution blocks is a Global Average Pooling (GAP) layer.

We shall take a closer look at some of these elements, first from the upper branch (Karim et al. 2019):

- Dimension shuffle: this block simply transposes the multivariate time-series input instance (that is a matrix in the multivariate case). This means that each component time-series while be fed as a whole to the LSTM layer, exposing each component's temporal history to the layer at once, and it is stated that this reduces the computational effort.

- LSTM: the concept of Long Short-Term Memory was introduced by Hochreiter et al. (1997). It was designed to improve Recurrent Neural Network (RNN), specifically to combat the vanishing gradient problem in these networks (Karim et al. 2019, p. 238). RNN are special networks designed and used for sequence modeling, but can be costly to train (Goodfellow et al. 2016, pp. 373, 381). No further details are given here, but suffice to say that LSTM can learn temporal dependencies in time-series data, but only to a certain extent (Karim et al. 2019, p. 238). While an RNN with LSTM can be used as a standalone network for time-series data, it is applied as only a small part of a larger network in this architecture.

The elements from the lower branch are:

- Convolutional layer followed by a Batch Normalization (BN) layer and Rectified Linear Unit (ReLU) activation function. The convolutional layers apply convolutions along the time-axis, and have different kernel sizes of 8, 5, and 3 values, respectively. In each of the convolutional layers, 128 or 256 kernels of the respective length are used (thus leading to as many feature maps). The convolutional layers again serve as feature extraction stages (Karim et al. 2019, p. 239).

- Squeeze-and-Excite (SaE)[5] block. See Figure 6.7 for an overview. The general description of this concept can be found in (Hu et al. 2020),

[5]Note that Hu et al. (2020), the authors who introduced this concept, call this squeeze-and-excitation block.

the specialization to time-series data as in (Karim et al. 2019) is briefly described in the following.

Consider the element labeled $\boldsymbol{U}$ in Figure 6.7, it represents feature maps that are the results of convolution operations. The dimensions are T, the length of the time-series, C the number of feature maps (number of kernels), the third dimension representing the components of the multivariate time-series is unlabeled in the figure.[6]

The operation $\boldsymbol{F}_{sq}(\cdot)$ (squeeze) averages each feature map separately over the dimension T, the result is a vector $\boldsymbol{z}$ of dimension $1 \times C$ (this is a form of average pooling). This vector is subjected to the excite operation $\boldsymbol{F}_{ex}(\cdot, \boldsymbol{W})$ that looks like this:

$$\boldsymbol{s} = \boldsymbol{F}_{ex}(\boldsymbol{z}, \boldsymbol{W}) = \sigma(\boldsymbol{W}_2\, \delta(\boldsymbol{W}_1 \boldsymbol{z}^\mathsf{T})), \tag{6.2}$$

where $\sigma(\cdot)$ is the sigmoid-function, $\delta(\cdot)$ is a ReLU-function; $\boldsymbol{W}_1 \in \mathbb{R}^{C/r \times C}$ reduces the dimension to C/r, whereas $\boldsymbol{W}_2 \in \mathbb{R}^{C \times C/r}$ expands the dimension to C again.[7] The scalar r is called the reduction ratio. Both matrices $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ are learned during training.

The output $\boldsymbol{s}$ of $\boldsymbol{F}_{ex}(\boldsymbol{z}, \boldsymbol{W})$ is finally used to scale $\boldsymbol{U}$ channel-wise (operation $\boldsymbol{F}_{scale}(\cdot, \cdot)$) to produce $\widetilde{\boldsymbol{X}}$, the output of the Squeeze-and-Excite (SaE) block.

While somewhat complex structures, the goal of these SaE blocks is simply to apply learned weights to the different feature maps that were extracted by the previous convolutional layers. This enables the network to put more attention on some features over others. As Karim et al. (2019, pp. 239 f) state, the introduction of SaE blocks into this architecture was essential for achieving good performance on multivariate time-series data.

To again summarize this architecture in a few points:

- Combination of dedicated sequence model (LSTM) with convolutional layers.
- Kernels of different lengths to cover different temporal contexts.
- Squeeze-and-Excite blocks for enabling the network to put more attention on certain feature maps over others. This improves the performance for multivariate time-series data.
- Output of probability distribution over classes for each instance.

[6] It is not made explicitly clear in (Karim et al. 2019, pp. 239 f) what the third dimension here is; in fact, from the formulations it is likely that they demonstrate the squeeze-and-excitation concept for a univariate time-series.

[7] Note that (Karim et al. 2019) state dimension $C/r \times C$ for $\boldsymbol{W}_2$ as well, but this would not match with $\boldsymbol{W}_1$.

Fig. 1. The MLSTM-FCN architecture. LSTM cells can be replaced by Attention LSTM cells to construct the MALSTM-FCN architecture.

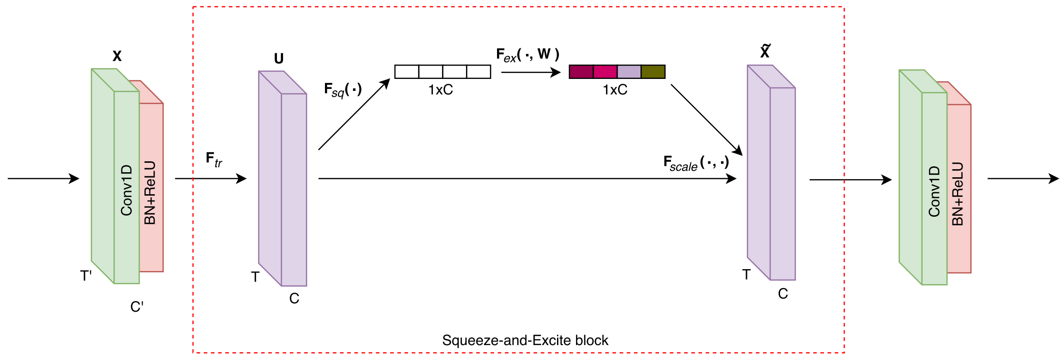

Fig. 2. The computation of the temporal *squeeze-and-excite* block.

Figure 6.7.: Squeeze-and-Excite (SaE) block of the FCN-LSTM architecture. See text for description. Figure from (Karim et al. 2019).

Implementation

The implementation for this model was taken from tsai as well, where the class MLSTM-FCN of (Oguiza 2020b) was used.

6.3.3. ROCKET

The RandOm Convolutional KErnel Transform (ROCKET) model was introduced in (Dempster et al. 2020). It is, like CNNs, based on the convolution of kernel functions (sequences) w_i with data input sequences as a means of feature extraction.

Architecture

ROCKET is not a complete CNN as shown in Figure 6.1, and is not quite following the structure shown there. The model basically consists only of the lower part used for feature extraction, and more specifically only a single convolutional layer, with an unusually large number of kernels (Dempster et al. 2020, p. 1461).

Another important difference is that the kernel parameters of this convolutional layer are in fact not estimated during a training phase, but rather randomly initialized once and then used as-is. The idea of using randomly created kernels for feature extraction has been used in the past successfully (Dempster et al. 2020, p. 1459).

The features generated from convolving the randomly created kernels with the input data are then used to train a separate classifier in a supervised fashion.

The fact that kernels are not learned during training makes the model itself, used as a pure feature extractor, computationally cheap overall. Furthermore, there are no hyper-parameters to tune for this model, except for the number of kernels to generate (Dempster et al. 2020, p. 1460).

Besides the simple structure mentioned above, and how it differs from more

to LSTM-FCN, as it incorporates learned self-attention to the inter-correlations between multiple variables at each time step, which was inadequate with the LSTM-FCN.

In addition, the multivariate time series input is passed through a dimension shuffle layer (explained more in Section 3.2), followed by the LSTM block. The LSTM block is identical to the block from the LSTM-FCN or ALSTM-FCN models (Karim et al., 2017), comprising either an LSTM layer or an Attention LSTM layer, which is followed by a dropout layer.

3.2. Network input

Depending on the dataset, the input to the fully convolutional block and LSTM block vary. The input to the fully convolutional block is a multivariate variate time series with Q time steps having M distinct variables per time step. If there is a time series with M variables and Q time steps, the fully convolutional block will receive the data as such. Variables are defined as the channels of interconnected data streams.

In addition, the input to the LSTM can vary depending on the application of dimension shuffle. The dimension shuffle transposes the temporal dimension of the input data. If the dimension shuffle operation is not applied to the LSTM path, the LSTM will require Q time steps to process M variables at each time step. However, if the dimension shuffle is applied, the LSTM will require M time steps to process Q variables per time step. In other words, the dimension shuffle improves the efficiency of the model when the number of variables M is less than the number of time steps Q.

After the dimension shuffle, at each time step t, where $1 \le t \le M$, M being the number of variables, the input provides the LSTM the entire history of that variable (data of that variable over all Q time steps). Thus, the LSTM is given the global temporal information of each variable at once. As a result, the dimension shuffle operation reduces the computation time of training and inference without losing accuracy for time series classification problems. An ablation test is performed to show the performance of a model with the dimension shuffle operation is statistically the same as a model without using it (further discussed in Section 4.4).

The proposed models take a total of 13 h to process the MLSTM-FCN and a total of 18 h to process the MALSTM-FCN on a single GTX 1080 Ti GPU. While the time required to train these models is significant, one can note their inference time is comparable with other standard models.

4. Experiments

MLSTM-FCN and MALSTM-FCN have been tested on 35 datasets, in Section 4.2. The optimal number of LSTM cells for each dataset was found via grid search over 3 distinct choices - 8, 64 or 128, and all other hyper parameters are kept constant. The FCN block is comprised of 3 blocks of 128-256-128 filters for all models, with kernel sizes of 8, 5, and 3 respectively, comparable with the original models proposed by Wang et al. (2017). Additionally, the first two FCN blocks are succeeded by the *squeeze-and-excitation* block. We consistently chose 16 as the reduction ratio r for all *squeeze-and-excitation* blocks, as suggested by Hu et al. (2017). During the training phase, we set the total number of training epochs to 250 unless explicitly stated and the dropout rate is set to 80% to mitigate overfitting. Each of the proposed models is trained using a batch size of 128. The convolution kernels are initialized by the *Uniform He* initialization scheme proposed by He, Zhang, Ren, and Sun (2015), which samples from the uniform distribution $U \in \left(-\sqrt{\frac{6}{d}}, \sqrt{\frac{6}{d}}\right)$, where d is the number of input units to the weight tensor. For datasets

"typical" Convolutional Neural Networks (CNNs), there are a few more things that distinguish ROCKET (Dempster et al. 2020, p. 1461):

- There are no activation functions applied to the output of the convolutions, in particular no non-linear ones (like ReLU).
- There are no connections between layers (as there is only one).
- There are no kernels with shared parameters (at least not by design), as each kernel is instantiated randomly (see below for details).
- Besides global max pooling, i.e. taking the maximum value of the feature maps (the outputs of the convolutional layers), a newly introduced version of pooling is applied in parallel, namely the proportion of positive values (ppv). This is the average number of positive values in the respective feature map (Dempster et al. 2020, p. 1463). Dempster et al. (2020, pp. 1461, 1475) claim that this additional new pooling operation is crucial to the performance of ROCKET.
- A special variant of the convolution operation, the dilated convolution, is used, see next subsection. While other networks use this as well, they typically have a more structured way of selecting the dilation parameter (see below) than choosing it randomly, as ROCKET does.

Dilated Convolution

The feature extraction makes use of the dilated convolution operation $\cdot *_d \cdot$ (Dempster et al. 2020) (Yu et al. 2016):

$$(x *_d k)(t) = \sum_{j=0}^{l_k-1} x_{t-j\,d}\, k_j, \tag{6.3}$$

where x_t is a data sequence, k_t is a kernel sequence of length l_k, and d is the dilation parameter. Not shown here is the bias b that is added to the result of the convolution operation. The dilation parameter (or dilation rate) d controls the amount of values of x_t omitted from the convolution operation: for $d > 1$, every d-th value of the input sequence x_t is skipped, this amounts essentially to a sub-sampling of the input data. For $d = 1$, the standard discrete time convolution as shown in Equation (6.1) is obtained. Dilated convolution can be used for capturing patterns over different scales (time here) (Dempster et al. 2020, p. 1458).

It shall be noted that Yu et al. (2016) explicitly state that not the kernel is dilated, but the data sequence, and differentiate their definition from the case where the kernel (or filter) is dilated (Yu et al. 2016, p. 2). It would appear that the operation in Equation 6.3 is no longer commutative in the sense shown in Equation 6.1 for $d > 1$.

Dempster et al. (2020) state that applying dilated convolution, with different kernels and multiple levels of dilation per kernel, can be viewed as being similar to several other methods used for extraction of features from time-series for classification:

- Frequency selective features: these represent only certain aspects of the data, e.g. slowly evolving patterns of lower frequency. Random convolution kernels as well allow for a form of frequency selective features, similar to what is used in "classical" approaches. Larger dilation corresponds to selectivity towards lower frequencies, and the opposite for smaller dilation (Dempster et al. 2020, p. 1458).
- Shapelets: shapelets are (sub-)sequences often extracted from the input data (or obtained from other sources). Shapelets are meant to represent patterns in the data that can discriminate between classes, and instances are related to them by means of a distance or similarity measure (Dempster et al. 2020, p. 1460). Discriminative shapelets would ideally exhibit for example high similarity to instances of one class, but not the other(s). The convolution of data with a kernel will yield higher output if the pattern in the data more closely matches the kernel (Dempster et al. 2020, p. 1458), which is conceptually quite similar.

Kernels

The kernels employed in the convolutions are generated randomly, where the following parameters are used for ROCKET (Dempster et al. 2020, p. 1462):

- Length l_k of the kernel: taken from $\{7, 9, 11\}$ with equal probability.
- Weights, i.e. the values of the respective kernel sequence: each value is sampled from a standard normal distribution $\mathcal{N}(0, 1)$. The resulting kernel sequence is mean-centered afterwards.
- Bias b: sampled from the uniform distribution $\mathcal{U}(-1, 1)$.
- Dilation d: for length l_x of data sequence x_t, the dilation is computed as $d = \lfloor 2^m \rfloor$ where m is sampled from $\mathcal{U}(0, A)$ with $A = \log_2 \frac{l_x - 1}{l_k - 1}$. This ensures that the kernel sequence is at most as long as the dilated (down-sampled) input data sequence.
- Padding: a random decision with equal probability is taken for each kernel whether no padding or zero padding is applied. The former variant puts more emphasis on the center part of the respective input data sequence, whereas the latter variant also takes the edges into account, as the center part of the kernel is applied to them with appropriate padding.
- Stride: fixed to a value of 1.

Per default, Dempster et al. (2020) suggest to create $n_k = 10\,000$ kernels randomly. Per kernel, two scalar values are emitted as feature values, one following from max pooling, and one from the pooling computing the proportion of positive values (ppv), so that in total $2\,n_k$ feature values are computed for each data instance.

The huge number of kernels, and the even larger number of resulting features, can be (much) larger than the number of instances in the training data for smaller data sets. Interestingly, Dempster et al. (2020, p. 1464) report good performance of the ROCKET method in these cases as well, where they used a classifier based on ridge regression.

Implementation

The original version of ROCKET is only applicable to univariate time-series (Dempster et al. 2020, p. 1466). However, there are extensions for the multivariate case available, as implemented in the software libraries `sktime` (sktime 2022), (Löning et al. 2019) and `tsai` (Oguiza 2020d), the latter of which was used for the experiments conducted for this thesis.

Classifiers

For the linear classifier that operates on the features generated using the random kernels, Dempster et al. (2020, p. 1464) suggest to use:

- logistic regression with gradient descent for larger data sets,
- ridge regression for smaller data sets, especially when the number of instances is smaller than the number of features.

They furthermore stress the importance of applying regularization when training the classifiers.

For the experiments conducted here, instead of the ones suggested by Dempster et al. (2020), a Gradient Boosted Decision Trees (GBDT) classifier was used, the concrete implementation being XGBoost (dmlc 2021), (Chen et al. 2016). This combination was used in (Silva et al. 2022) for time-series classification tasks and showed better performance than using a ridge classifier. See Section 6.5 for a brief discussion of Gradient Boosted Decision Trees (GBDT) and XGBoost.

6.4. Training CNNs

Typically, training a neural network, i.e. estimating the parameters involved in computing the output for given input data, is done using some form of gradient descent optimization of a cost function (such as those discussed in Section 4.2.2). However, not only neural networks make use of gradient descent optimization, in fact all models that allow estimating parameters involving a cost function can

be trained this way. This is also the case for the other models used in this thesis, see Section 6.5.

Consider a training data set $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots (\boldsymbol{x}_N, y_N)\}$, consisting of pairs of data instances $\boldsymbol{x}_i$ and corresponding class labels y_i.

The discriminative model shown above in Section 4.2.1 can be written as:

$$(P(\mathcal{C}_0|\boldsymbol{x}), P(\mathcal{C}_1|\boldsymbol{x}), \dots, P(\mathcal{C}_{n_C-1}|\boldsymbol{x})) = f(\boldsymbol{x}; \boldsymbol{\theta}), \tag{6.4}$$

where $f(\boldsymbol{x}; \boldsymbol{\theta})$ represents the model whose behavior is defined by the parameters $\boldsymbol{\theta}$, producing the conditional probability of each class $\mathcal{C}_j$ for a given datum $\boldsymbol{x}$. For our binary case this simplifies to:

$$(P(0|\boldsymbol{x}), P(1|\boldsymbol{x})) = f(\boldsymbol{x}; \boldsymbol{\theta}), \tag{6.5}$$

and, using the notation shown in Equation (4.1), further to:

$$(1 - p_t, p_t) = f(\boldsymbol{x}; \boldsymbol{\theta}). \tag{6.6}$$

A loss function $L(\mathcal{D}; \boldsymbol{\theta})$, as discussed in Section 4.2.2, can be used to evaluate the state of the model (defined by its parameters $\boldsymbol{\theta}$) on the given set of data $\mathcal{D}$. Gradient descent methods are iterative procedures, i.e., there is a sequence of steps $k = 1, \dots, K$, where at each step the loss function $L(\mathcal{D}; \boldsymbol{\theta}_k)$, defined by the current set of parameters $\boldsymbol{\theta}_k$, is used to compute a value of the loss $\mathcal{L}_k(\mathcal{D}; \boldsymbol{\theta})$. Thus, updating the model parameters $\boldsymbol{\theta}$ step-wise by some amount $\boldsymbol{\Delta}_{\boldsymbol{\theta}}$,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \boldsymbol{\Delta}_{\boldsymbol{\theta};k}, \tag{6.7}$$

leads to a step-wise change of the loss $\mathcal{L}$ by some amount $\boldsymbol{\Delta}_L$:

$$\mathcal{L}_{k+1} = \mathcal{L}_k + \boldsymbol{\Delta}_{L;k}. \tag{6.8}$$

The goal is to find parameters $\boldsymbol{\theta}$ corresponding to a (possibly only local) minimum of the loss function and thus overall loss computed on the given data set. This can be achieved by updating the parameters such that the loss is minimized step-wise (Bishop 2006, p. 240):

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \, \boldsymbol{\nabla}_{\boldsymbol{\theta}} \, L(\boldsymbol{\theta}_k), \tag{6.9}$$

where $\boldsymbol{\nabla}_{\boldsymbol{\theta}}$ is the gradient in the parameter space and $\eta > 0$ is the learning rate, used to scale the update of the parameters (Bishop 2006, p. 236). More concretely, for $\boldsymbol{\theta} \in \mathbb{R}^J$, the gradient is defined as:

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} = \left\{ \frac{\partial}{\partial \theta_j} \right\}_{j=1}^{J}. \tag{6.10}$$

In Equation 6.9, the parameters are updated by moving a certain amount into the direction of the negative gradient, i.e. a direction of steepest descent.

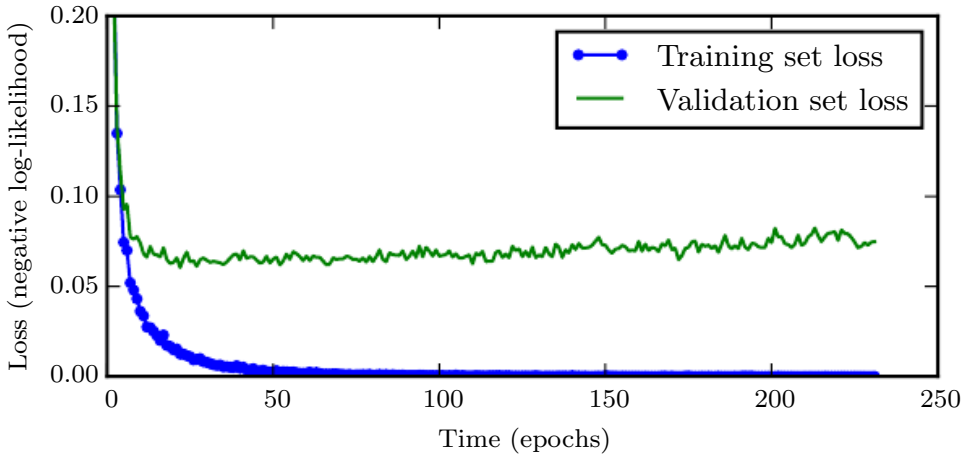

Figure 6.8.: Learning curves: loss values per data set (training and validation), for each epoch, during the training process. Other quantities (performance metrics, such as accuracy) can be used for monitoring the training process. Figure from (Goodfellow et al. 2016, p. 246).

Figure 7.3: Learning curves showing how the negative log-likelihood loss changes over time (indicated as number of training iterations over the dataset, or **epochs**). In this example, we train a maxout network on MNIST. Observe that the training objective decreases consistently over time, but the validation set average loss eventually begins to increase again, forming an asymmetric U-shaped curve.

greatly improved (in proportion with the increased number of examples for the shared parameters, compared to the scenario of single-task models). Of course this will happen only if some assumptions about the statistical relationship between the different tasks are valid, meaning that there is something shared across some of the tasks.

From the point of view of deep learning, the underlying prior belief is the following: *among the factors that explain the variations observed in the data associated with the different tasks, some are shared across two or more tasks.*

7.8 Early Stopping

When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again. See figure 7.3 for an example of this behavior. This behavior occurs very reliably.

This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error. Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters. The

Many different versions and variants of the gradient descent approach exist. One way to categorize them is by the number of instances $\boldsymbol{x}$ processed before the trainable parameters $\boldsymbol{\theta}$[8] are updated (see e.g. Goodfellow et al. (2016, pp. 151 ff., 177 ff., 277 ff), Bishop (2006, pp. 240 ff)):

- Stochastic Gradient Descent (SGD) performs an update step as in Equation (6.9) for each datum (instance) $\boldsymbol{x}$.
- Batch Gradient Descent performs an update only after processing the whole data set $\mathcal{D}$ (this is called an epoch).
- Mini-Batch Gradient Descent is between the two, and performs an update after processing a certain number of instances n_B (where n_B is called the batch-size of the mini-batch.

Learning curves, as shown in Figure 6.8, can be used to monitor the training process. One epoch corresponds to processing the whole training data set (all instances) once. Typically, training lasts at least several, sometimes hundreds of epochs. Evaluating the loss function L on both the training as well as the validation set yields a value for the loss for the respective set (for the current step k). The training set loss should decrease during training, while the validation set loss should decrease as well. Typically, the validation set loss will start to increase again at some point. This typically means that the model is starting to overfit to the training data (Goodfellow et al. 2016, p. 246), (Bishop 2006, p. 259).

[8] With an abuse of notation, the trainable parameters are not distinguished here from other parameters of the respective model that can still influence its performance; these are the hyper-parameters, discussed in Section 6.7.

6.4.1. Regularization

Regularization (for machine learning models) aims at low errors on test data sets, or any other (new) data set not used during the training process for estimating model parameters, in addition to low errors on the training data set (i.e. good model fit) (Goodfellow et al. 2016, p. 228). In other words, regularization is meant to prevent overfitting of a model to the training data. Several different approaches and methods exist for most machine learning models, and two important methods often applied to machine learning models in general, and neural networks in particular, (and used in the experiments) are briefly described.

Early Stopping

Early stopping is closely tied to observing the training curves (see Figure 6.8) during training with a gradient descent method. As was stated above, an increase in loss computed on the validation set during training is a sign of overfitting.

The method of early stopping monitors a metric computed on the validation set (e.g. the loss) and stops training of the model (network), i.e. updating the model parameters, when this metric starts to increase or even stops to decrease (again, as in the case of the loss). This should yield a model that is not overfit to the training data and offers good generalization capabilities (Bishop 2006, p. 259).

Different evaluation metrics may require different criteria, e.g. accuracy is a metric to be maximized rather than minimized, as is the ROC-AUC. Ideally, the model is stored in its current version at e.g. every (n-th) epoch, or whenever the respective metric being optimized changes, and the best version identified by early stopping can directly be used (Goodfellow et al. 2016, 246f)

As learning curves can be quite "rough" (and much less smooth than the ones shown in Figure 6.8), it can pay off to wait for a few epochs before making the decision to stop the training. This number of epochs to wait is usually referred to as patience (Goodfellow et al. 2016, p. 247)

Dropout

Dropout can be viewed as an approximation to an ensemble of several neural networks, more specifically bagging. Bagging is an ensemble method where several different models are trained on the same data, and whose predictions on the same test (or evaluation) data are then combined (Goodfellow et al. 2016, p. 258).

An illustration of dropout is shown in Figure 6.9. The basic idea is to remove certain units from the network randomly to create a sub-network. This selection of units to drop can e.g. be taken anew before each epoch. Instead of training multiple different networks, dropout then leads to training different networks in sequence, epoch after epoch.

Dropout can for example by used for the fully connected layers used as the top part of Deep Neural Networks (DNNs), acting as a classifier. See e.g. the fully connected layers in the classification part of the CNN in Figure 6.1, as well

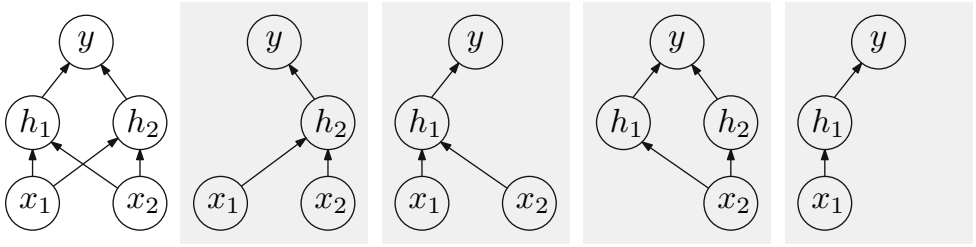

Figure 6.9.: Illustration of dropout. On the left is a very simple Feed Forward Neural Network (FFNN) with two inputs (x_1, x_2) and two hidden units (h_1, h_2). The four subplots to the right show four different subnets resulting from erasing (dropping) one or multiple units from the original network. Figure inspired by Figure 7.6 in (Goodfellow et al. 2016, p. 260).

as the fully connected part of the InceptionTime network in Figure 6.3.

However, dropout can be used for other layers as well, e.g. convolutional layers in Convolutional Neural Networks (CNNs).

6.5. Gradient Boosted Decision Trees (GBDT)

Tree-based methods (or trees for short) are well known and quite simple models that allow for regression as well as classification applications. They have in general a number of desirable properties (only some of which are relevant to the problem at hand) (Hastie et al. 2017, p. 352), (Kuhn et al. 2020, p. 221):

- Trees are invariant under strictly monotone transformations of the predictors (inputs). This was briefly mentioned earlier in Section 5.2 when looking at necessary transformations of the data.
- Tree-based methods can deal with highly correlated data.
- They perform feature selection internally, and can thus deal with the inclusion of (many) irrelevant predictor variables.

The last point mentioned above is of course useful when applying a tree-based classifier to the ROCKET features, as there is a very large number of them, and not all may be relevant or discriminative.

Decision Trees

The main idea of tree-based methods is to partition the input data (feature) space along each dimension (say each of p dimensions for $\boldsymbol{x} \in \mathbb{R}^p$) at certain values (split points) into disjoint regions R_j, $j = 1, \ldots, J$, and approximate the response (say, y) in each region in some way, where the splitting and approximation procedure can be done recursively to more and more "fine-grained" regions. Regions that are not split anymore are called terminal nodes or leaves and contain at least one datum. Other nodes besides the root node are called inner nodes. This

partitioning can be depicted in a binary tree, hence the name (Hastie et al. 2017, pp. 305 ff, 353). The number J of terminal regions, and thus leaves of the tree, is called the size or depth of the tree and is a hyper-parameter (Hastie et al. 2017, pp. 307 f).

During supervised training, for each previously created partition, first the split point at some value of the respective variable (dimension) of the data has to be found to form new regions. Afterwards, the respective value γ that represents (approximates) the response y in each of these new regions has to be estimated. For classification, this representation can for example be the label of the class having the majority in the respective region, or the proportion of the examples belonging to a class of all examples in the region. The procedure of choosing splits and approximating the responses describes a combinatorial optimization problem, and typically a greedy approximation is used to solve it (Hastie et al. 2017, pp. 309, 356).

A single tree is parameterized as follows in (Hastie et al. 2017, p. 356):

$$T(\boldsymbol{x};\boldsymbol{\theta}) = \sum_{j=1}^{J} \gamma_j \, I(\boldsymbol{x} \in R_j) \tag{6.11}$$

where $I(\cdot)$ is the indicator function and γ_j is the respective value associated with (estimated within) the region R_j, and $\boldsymbol{\theta} = \{(R_1, \gamma_1), \ldots, (R_J, \gamma_J)\}$ is the sequence of regions and values. Thus, $T(\boldsymbol{x};\boldsymbol{\theta})$ is an approximation of the response across the regions of the whole input data space. The last pair in $\boldsymbol{\theta}$, (R_J, γ_J), represents the leaves (terminal nodes) of the tree.

However, a single tree alone is often not the best possible model for a given body of data, in terms of e.g. classification accuracy or other figures of merit. Gradient Boosted Decision Trees, also referred to as Gradient Boosted Models (GBM) (see e.g. Hastie et al. (2017)), are a certain type of additive models where the functions of the input data that are added to form the output (a predicted class label for classification), are trees (Chen et al. 2016, p. 786). GBDT can yield drastically improved performance over trees (Hastie et al. 2017, p. 352).

Additive Tree Model

The additive model combining M trees (the approximations $T(\boldsymbol{x};\boldsymbol{\theta})$) can be written as (Hastie et al. 2017, p. 356):

$$f_M(\boldsymbol{x}) = \sum_{m=1}^{M} T(\boldsymbol{x};\boldsymbol{\theta}_m), \tag{6.12}$$

where M is the number of trees to combine, and typically is a hyper-parameter to tune. While each tree could have a different size J_m, it is suggested to use a single value $J_m = J \; \forall m$ (Hastie et al. 2017, p. 362). The value chosen for size J doesn't seem to be very critical when restricted to a range of 4 to 8 (Hastie et al. 2017, p. 363).

The training of a GBDT model starts from a single tree and iteratively adds a new tree until (at most) M trees are obtained, at each step using a greedy strategy to add the currently best possible new tree (Hastie et al. 2017, pp. 361 f), according to a specified criterion. That is, after step $m-1$, a new tree $T(\boldsymbol{x};\boldsymbol{\theta}_m)$ is added in step m:

$$f_{m-1}(\boldsymbol{x}) + \nu\, T(\boldsymbol{x};\boldsymbol{\theta}_m), \tag{6.13}$$

where the factor ν can be used to scale the contribution of each added tree, but can also be interpreted as a form of learning rate (see next section).

Constructing (learning) a tree representation as in Equation (6.11) requires for each split twice the application of a loss function: for getting a new set of regions and for getting the approximation of the response in the newly formed regions (Hastie et al. 2017, pp. 356 f). The cross-entropy loss function as shown in Equation (4.4) for example is a suitable choice for these procedures.

6.5.1. Gradient-Boosting

Boosting methods are additive models, where, as just described, new base learners are added in sequence with the goal that each additional learner instance improves the performance of the ensemble built up so far. This is achieved by applying a new set of instance weights to the training data at each additive step, where the weights are related to the performance of the current ensemble. Instances that were mis-classified before are given greater weight in the next step to make the model focus more strongly on classifying them correctly, instances that were classified correctly get their weights decreased (Bishop 2006, pp. 657 ff), (Hastie et al. 2017, pp. 337 ff).

A special form of this concept is gradient boosting. It is a way to implement the greedy selection of a new tree for the additive model in Equation (6.12), thus the update step of the additive model. This update step, adding a new tree, can be viewed as being similar to a step in a gradient descent algorithm, but here the new tree approximates the negative gradient of the loss function, where the partial derivatives are taken with respect to the function (the tree) to be added (Hastie et al. 2017, pp. 359 f).

Equation (6.13), showing the update of the additive model by an additional tree, can be seen as similar to Equation (6.9), updating the model parameters. Both updates change the behavior of the respective model such that the loss function is minimized successively (until some stopping criterion is reached).

A more detailed discussion is beyond the scope of this thesis, the reader is referred to (Friedman 2001) and (Hastie et al. 2017).

6.5.2. Regularization

Several methods for regularization are available for GBDT models, aiming at decreasing the possibility of overfitting to the training data. An overview shall be given, following (Hastie et al. 2017, pp. 364 ff):

- Monitoring loss on a validation set, effectively again an early stopping strategy, limiting the number of trees (and thus boosting iterations) M.
- Shrinkage, by scaling the contribution of each newly added tree by some constant $\nu \in [0, 1]$ (see Equation (6.13)). This can be considered as an additional hyper-parameter.
- Subsampling (undersampling) the training data instances, without replacement. The amount of subsampling $\eta \in [0, 1]$ (ratio of undersampling) is another hyper-parameter that can also be interpreted as a learning rate.

In general, increasing M increases the model and thus the computational load. There is a trade-off between the shrinkage parameter ν and the number of trees (boosting rounds) M. Friedman (2001, pp. 1203 f) suggests that both parameters should be be optimized in a model selection scheme, and this advice was indeed followed in the experiments conducted.

There is another point to be added the above list not mentioned by Hastie et al. (2017), that is subsampling the variables (dimensions) of the data (Chen et al. 2016, p. 787), where the amount of subsampling again is a hyper-parameter. This regularization approach is implemented in the software library XGBoost that is briefly discussed next.

6.5.3. XGBoost

`xgboost` is a software library implementing Gradient Boosted Decision Trees (GBDT) models (dmlc 2021), (Chen et al. 2016). It was chosen for its offering of computation on GPUs, and its known good performance as a classifier in general (see Chen et al. (2016)).

Model

Besides gradient descent methods that employ the gradient $\nabla L(\boldsymbol{\theta})$ of the cost function, other optimization schemes exist that additionally incorporate higher order partial derivatives. For the case of the second derivatives, this leads to the use of the Hessian matrix $\boldsymbol{H}$ (or approximations of it).

XGBoost employs the gradient as well as the Hessian of the respective cost function used for learning the model parameters from a set of training data (Chen et al. 2016, pp. 786 f), following the concept of the gradient boosting method described above. A more detailed discussion of the XGBoost model is beyond the scope of this thesis, but some properties will be discussed below for the case of handling imbalanced data. Furthermore, the hyper-parameters (including methods for regularization and handling class imbalance) used for the model are listed in Section 7.4.2.

Handling Imbalanced Data

Note that XGBoost, being an ensemble method, can itself already be considered a model-level approach to alleviate the problems of imbalanced data (as was

mentioned in Section 4.2).

Additionally, XGBoost offers two approaches specifically addressing imbalanced data (dmlc 2022a):

1. Specifying class weights, thus offering cost-sensitive learning, a model level approach.

2. Limiting the values of the leaf weights in the update step. The leaf weights correspond to the values $\gamma_J^{(m)}$ representing the terminal regions of the tree to be added at step m (compare Equations (6.13) and (6.11); for more details the reader is referred to Section 2.2 in Chen et al. (2016)). The parameter for influencing the update limit is named `max_delta_step` in the XGBoost library (see (dmlc 2022b)).

Applying class weights (option 1), however, no longer makes it possible to interpret (and use) the model output as posterior probabilities for the classes (dmlc 2022a). This becomes evident in particular when inspecting calibration curves for models where class weights were used during training. The models trained in this way are usable as classifiers, but meaningfully choosing a threshold or applying costs in the decision step may no longer be possible.

Both methods for handling class imbalance mentioned above were tested during the experiments, see Section 7.4.2 for a description of the inclusion of the methods into the hyper-parameter tuning.

6.6. Model Selection

During the model selection phase, several different models, i.e. algorithms, architectures, versions, variants, etc., are trained on the same training data set and validated on the same validation data set, at least as much as possible; after all, different models may require different preprocessing methods.

Besides estimating the actual model parameters that can be learned during the training process, the selection of values for the hyper-parameters happens during this phase as well. Hyper-parameters are parameters of a model (or algorithm) that influence its performance, but cannot be learned directly from the data (see e.g. Goodfellow et al. (2016, p. 120), and the next section for a discussion).

The goal of model selection is to choose the best model according to some specified criterion, e.g. a certain validation metric. This process can be viewed as being two-fold (Kuhn et al. 2013, p. 26): first, there is a selection within a certain (type of) model, where the combination of hyper-parameters giving best performance, as just described, can be chosen. Second, the best model, e.g. neural network architecture such as InceptionTime, can be chosen among the evaluated models.

As was illustrated in Section 5.7, a 5-fold Cross-Validation (CV) scheme was used for the model selection process. For each model, the procedure was as follows:

1. Before any training starts, set the values for each hyper-parameter that defines the respective model.
2. Take for fold j with $j = 1, \ldots, 5$ the associated training and validation set parts.
3. Train the model on the training part, and evaluate on the validation part, store the evaluation metrics computed on both parts (the values computed for the validation part are the more relevant ones here).
4. After all folds are completed, compute the average of each evaluation metric across the folds and store this average metrics for comparing models.
5. Repeat all above steps for a specified number of times (possibly many) to find a set of hyper-parameters that yields good performance of the respective model.

All models trained and evaluated using this procedure can than be ranked against each other, e.g. by the average of the evaluation metric(s) across the folds. A secondary sorting criterion could e.g. be the required training time, or model complexity.

Kuhn et al. (2020, p. 45) suggest to use the ROC-AUC metric to find a good model first, then afterwards tune the cutoff between the two classes (or the decision rule) to achieve specific precision and recall values, possibly incorporating costs in the tuning process. This is the strategy followed here for model selection and hyper-parameter tuning comparing models (see Chapter 8 for the results of the model selection).

The best set of hyper-parameters for each model, thus the one obtained from the within-model selection, can then be used to e.g. retrain on the whole training part across all folds (see Figure 5.9). Alternatively, the model from the fold with best score on the validation part can be used directly without the need for retraining. The latter choice sacrifices a small amount of data for retraining, but will yield a model with known performance across folds. This is the approach chosen here.

6.7. Hyper-Parameters

As was just stated above, hyper-parameters are parameters of a model (or algorithm) that influence its performance, but cannot be learned directly from the data. Typical examples are quantities that influence the regularization of the model, or parameters that control the scaling of updates in gradient descent algorithms (learning rate), among many others.

Hyper-parameters can be set either manually, which requires (possibly quite in-depth) knowledge about the model at hand and its behavior, or they can be selected automatically (Goodfellow et al. 2016, pp. 427 f, 432). Automatic

methods obtain good or optimal values for hyper-parameters by repeatedly training and evaluating (on a separate data set) a model, and choosing the values giving best performance.

If there are several different hyper-parameters that influence the model, and each can take on different values, possibly from a continuous range, the amount of possible combinations can explode very fast. Thus, tuning hyper-parameters is a problem of its own.

Several approaches exist for automatic tuning of hyper-parameters (Goodfellow et al. 2016, pp. 432 ff):

- Grid search: for each hyper-parameter, a set of several values is specified, and taking the Cartesian product forms a "grid" of tuples of values. Each tuple is used to specify a model that is then trained and evaluated. Suitable values have to be taken from e.g. the literature or educated guesses. For m hyper-parameters with n values each, the complexity is $\mathcal{O}(n^m)$ (Goodfellow et al. 2016, p. 434) and can become prohibitively.

- Random search: rather than specifying concrete values, typically a range of values or even a (marginal) probability distribution for each hyper-parameter is supplied from which values are randomly sampled. This again creates tuples of values for the respective hyper-parameters. Random search has been found to be much more effective than grid search, reducing the validation set error in fewer trials (Goodfellow et al. 2016, p. 435).

- Bayesian optimization, a type of model-based hyper-parameter optimization: these create a model of the expected value of the validation set error and associated uncertainty. This allows for a tradeoff between exploring different values for parameters with high uncertainty (according to the model) and suggesting similar values for parameters that have performed well in previous trials seen so far (Hastie et al. 2017, pp. 435 f). This approach thus in a sense allows successive trials to use the information from previous trials about good values for (at least some of) the hyper-parameters.

Note that setting the ranges of values to choose from, or the distributions to sample values from for the respective hyper-parameter, can again be seen as hyper-parameters of the tuning method itself (Goodfellow et al. 2016, p. 432). However, it may be easier to choose these ranges and distributions from past experiments reported in the literature (for at least similar problems, for example) or from suggestions in the documentation of the respective software package.

CHAPTER 7

Experiment Settings

This chapter discusses the experiments carried out with the data processed according to Chapter 5, and the models and specificities for their training described in Chapter 6.

7.1. Computers

In order to illustrate the compute capabilities used for this thesis, a brief description of the 2 mostly used computers is given below in Table 7.1.1.

Table 7.1.1.: Overview of the compute capabilities used for the experiments.

Machine	CPU			RAM	GPU			OS
	Type	#Cores	Freq.		Type	RAM	CUDA cores	
V	4 x Intel Xeon Gold 6138	20 (each)	2 GHz	384 GB	2 x NVIDIA Quadro P2200	5 GB (each)	1280 (each)	Windows
G	AMD EPYC 7343	16	3.9 GHz	512 GB	NVIDIA A40	48 GB	10 752	Linux

The machines were used mostly for the following tasks:

- Machine V was used for data exploration, preprocessing and preliminary tests of models.
- Machine G was used almost exclusively for training and evaluating models. The strong GPU in this machine was very beneficial for carrying out experiments in shorter time.

Both machines were shared with other users, and exclusive use of resources was most of the time not possible.

7.2. Experimental Conditions

The most important conditions were (see next section for additional settings):

- Windowing: 4 different settings
- Models: 3 different models:
  - InceptionTime
  - FCN-MLSTM
  - XGBoost classifier with RandOm Convolutional KErnel Transform (ROCKET) features
- Loss functions

7.2.1. Models

The models used were discussed in Chapter 6, Sections 6.3.1 to 6.3.3 and 6.5.3. Table 7.2.1 shows the combinations of models, loss functions and settings for the ROCKET features (where applicable).

Table 7.2.1.: Combinations of models, loss functions and feature settings (for ROCKET) used in the experiments. The experiment labels are used in many tables and plots in the following.

Experiment Label	Model	Loss Function	Features	# Kernels
ITFL	InceptionTime	focal loss	-	-
ITXE	InceptionTime	cross-entropy	-	-
MLFL	FCN-MLSTM	focal loss	-	-
MLXE	FCN-MLSTM	cross-entropy	-	-
ROXG04k	XGBoost	cross-entropy	ROCKET	4000
ROXG10k	XGBoost	cross-entropy	ROCKET	10 000

7.2.2. Loss Functions

As was shown in Table 7.2.1, different loss functions were used. More concretely:

- Deep Neural Network (DNN) models, InceptionTime and FCN-MLSTM used:
  - cross-entropy (CE), with and without class weights
  - focal loss (FL), with and without class weights
- GBDT model (XGBoost): Cross-entropy (CE)

7.2.3. Windowing

The parameters involved in the windowing process (see Figure 5.2) are state length l, prediction length p, and stride s. Table 7.2.2 shows the four different settings used for generating windows.

Table 7.2.2.: Windowing settings used (numbers are durations in hours).

ID	state length l	prediction length p	stride s
3644	36	4	4
3688	36	8	8
4888	48	8	8
7288	72	8	8

7.2.4. Experiment Labels

In the following Table 7.2.3, the experiment labels for all 24 combinations of models, loss functions, feature settings and windowing settings are shown. Each of these 24 combinations was subjected to a model selection and hpyer-parameter tuning session.

Table 7.2.3.: Overview of all 24 combinations and the respective experiment labels.

InceptionTime	FCN-MLSTM	XGBoost + ROCKET	Windowing Settings
ITFL_3644	MLFL_3644	ROXG04k_3644	3644
ITFL_3688	MLFL_3688	ROXG04k_3688	3688
ITFL_4888	MLFL_4888	ROXG04k_4888	4888
ITFL_7288	MLFL_7288	ROXG04k_7288	7288
ITXE_3644	MLXE_3644	ROXG10k_3644	3644
ITXE_3688	MLXE_3688	ROXG10k_3688	3688
ITXE_4888	MLXE_4888	ROXG10k_4888	4888
ITXE_7288	MLXE_7288	ROXG10k_7288	7288

7.3. Additional Settings

Settings used in addition to the main ones described above are listed in this section. All parameters discussed in the following were used as hyper-parameters.

7.3.1. Addressing Class Imbalance

An overview is given of the methods employed to address the class imbalance problem for each model (refer to Chapter 4 for a discussion). Table 7.3.1 shows which approach for combating class imbalance was used with which model.

Table 7.3.1.: Strategies applied for handling the class imbalance. Refer to Chapter 4 for dicussion. (*): undersampling (of the majority class) was primarily done here to make the model and data fit into the GPU RAM of machine G.

Model	Implementation	Library	Addressing Imbalance	Level
InceptionTime	InceptionTimePlus	tsai	class weights, loss function	model
FCN-MLSTM	MLSTM_FCN	tsai	class weights, loss function	model
ROCKET + XGBoost	ROCKET, XGBClassifier	tsai, xgboost	class weights, boosting, undersampling(*)	model, data

7.3.2. Class Weights

For the cost-sensitive training setting, with weight factors applied to the classes during training, several different methods for choosing the weights were used and included into the hyper-parameter tuning (and model selection). See Section 7.4 below for more details, especially refer to Tables 7.4.1 and 7.4.2.

Some aspects of the methods for choosing weights were discussed in Section 4.2, see especially Section 4.2.3.

7.3.3. Regularization

Several regularization methods were applied to each of the models to decrease the risk of overfitting. Some aspects of the respective regularization methods were subject to hyper-parameter tuning, see Section 7.4 below. For an overview of the regularization methods used refer to Tables 7.4.1 and 7.4.2.

7.3.4. Transformations

For all experiments, the data were standardized by subtracting the mean and dividing by the standard deviation, for each variable (dimension) separately. Both quantities were always estimated on the respective training data part and applied to all parts (training, validation, test).

7.4. Hyper-parameter Tuning

Hyper-parameters and the necessity to tune them were discussed in Section 6.7.

For tuning the models' hyper-parameters, a Bayesian approach was chosen, and the library `optuna` was used (Akiba et al. 2019), (optuna 2021) for conducting the parameter search.

Training a model for search of hyper-parameter values is called a study in optuna's terminology. Each study consists of several trials, where one trial equals one combination of suggested values for all involved hyper-parameters. The time budget for the whole study, and the maximum number of trials in a study, can be specified for the parameter search. The library offers the possibility to prune trials, i.e. abandon the current trial (and the chosen hyper-parameter values) early on when the performance is much worse then what was seen in other trials before. This can save considerable amounts of time during search and allows to use the given time-budget more effectively.

The training scheme was such that at the start of a trial, a concrete value for each hyper-parameter was established ("suggested" in the terminology of `optuna`). These are then used to define the model for each of the 5 folds used in the Cross-Validation (CV) process. The respective model was then trained on each fold's training part and evaluated on the evaluation part. After each fold was completed, the average ROC-AUC score across folds was computed and stored to rank the different trials and their associated combination of hyper-parameters after the study was completed. Note that regularization strength, and for some models the method used for regularization, were considered hyper-parameters as well and included into the tuning procedure.

7.4.1. Deep Neural Networks (DNNs)

Table 7.4.1 gives an overview of the hyper-parameters used for the models InceptionTime and FCN-MLSTM. The first block shows hyper-parameters common to both models, the following blocks show specific paramters for each of the two models.

The inclusion of regularization methods (see Section 6.4.1) into the hyper-parameter tuning scheme is a natural choice, the same holds for the batch size. Including the quantity for monitoring the early stopping regularization should enable the models to obtain good results while ensuring that no overfitting occurs; preliminary experiments showed that using ROC-AUC for monitoring can lead to overfitting. However, for the different models and settings, there was not a clear enough picture to completely exclude the usage of this metric for monitoring early stopping. ROC-AUC was in all cases still used as overall metric to rank the trials against each other.

A similar scenario occurred for the class weights: preliminary experiments did not paint a very clear picture on which method to use for establishing class weights would yield the best results. Thus, several methods were included in the

Table 7.4.1.: Hyper-parameters for InceptionTime and FCN-MLSTM models.

Model	Category	Type/Variable	Value(s)/Case(s)	Data Set
InceptionTime and FCN-MLSTM	Gradient Descent	Batch Sizes		
	Regularization	Early Stopping Monitoring	ROC-AUC or Loss	Validation
	Class Imbalance (data)	Class Weights	Estimation from class distribution	Training
			Randomly chosen for minority class	
			Equal weights	
InceptionTime	Regularization	Dropout	Convolutional Layers	
	Regularization	Dropout	Fully Connected Layers	
	Architecture	# of	Filters in Conv. Layers	
FCN-MLSTM	Regularization	Dropout	RNN Layers	
	Regularization	Dropout	Fully Connected Layers	
	Architecture	Size of	Hidden Layers in FC Network	
	Architecture	# of	RNN Layers	

hyper-parameter tuning scheme.

While the inclusion of these parameters allows for greater flexibility of the models on the one hand, it comes at the cost of increased training time (given the fixed amount of compute power here) on the other hand.

7.4.2. XGBoost

The most important hyper-parameters involved in Gradient Boosted Decision Trees (GBDT) in general and for XGBoost in particular were discussed in Section 6.5.2. Table 7.4.2 holds the hyper-parameters used for the XGBoost models working on the ROCKET features.

For XGBoost, regularization methods (and the coefficients for tuning the amount of regularization applied) were included as well. Model coefficients were shrunk by either limiting their sum by the l_1- or l_2-norm where the amount of shrinkage was a hyper-parameter as well. Note that subsampling at several different points of the model was applied, such as subsampling data instances (rows of the data) as well as subsampling variables (columns).

XGBoost offers two different approaches for handling class imbalance at the model level: including class weights (not in the loss function), and applying weights to the updates in the additive model building process. It was a-priori not clear which of both methods would work better, and again preliminary experiments did not reveal a pattern. Thus, both methods were included (as mutually exclusive options) into the hyper-parameter tuning scheme. Again, the method for choosing class weights was included into the tuning.

Table 7.4.2.: Hyper-parameters used for XGBoost with ROCKET features. (*) Choosing an initial score seems to have helped to speed up training in the first few iterations. The influence of this choice on the model is decreased after a few boosting rounds. (**) The use of Class weights or weighted updates was mutually exclusive and the choice was a hpyer-parameter itself. (***) Undersampling was mainly carried to fit the data and model into the GPU RAM.

Model	Category	Type/Variable	Value(s)/Case(s)	Data Set
XGBoost Classifier	Architecture	Booster	Tree or Dart	
	Regularization	Coefficients l-Norm	l_1 or l_2	
	Regularization	Subsampling	Data instances	Training
	Regularization	Subsampling	Columns by Tree Columns by Level Columns by Node	Training
	Regularization	Early Stopping Monitoring	ROC-AUC or Loss	Validation
	Class Imbalance (model) (*)		Initial Score for Classification	
	Class Imbalance (model) (**)	Class Weights	Estimation from class distribution Randomly chosen for minority class Equal weights	Training
	Class Imbalance (model) (**)	Weighted Update	Max. Delta Step	
	Class Imbalance (data) (***)	Subsampling	Undersampling of majority class	Training
ROCKET	Architecture	# of Kernels	4000 or 10000	

As for the DNN models, the choice of the metric used for monitoring early stopping was included as well.

Additionally to the parameters listed in Table 7.4.2, the learning rate was tuned as a hyper-parameter (see discussion about the interpretation of the learning rate for additive tree models in Section 6.5).

As preliminary experiments showed, the full training data set (and model) would not fit into the RAM of the GPU of machine G. Thus, to enable the use of these models, random undersampling of the majority class of the training data was included. This was different for the cases of 4000 or 10 000 kernels for the ROCKET features: for the 10 000 kernel case, decreasing the data size was mandatory, thus different ratios of undersampling were included into the tuning. For the 4000 kernels case, whether random undersampling was applied at all was hyper-parameter itself, and if chosen the undersampling ratio as well.

CHAPTER 8

Results and Discussion

This chapter shows results of the experiments conducted in forms of tables and plots and discusses the outcomes.

Figure 8.1 shows the number of completed trials for each experimental setting. For each setting, the time budget for the hyper-parameter tuning (using Optuna) was fixed to a certain amount such that at least 10 trials would be completed. The time budget was estimated by extrapolating from running a few preliminary trials in each case. Some cases achieved many more completed trials than others due to inaccurate estimates on the one hand; on the other hand, the machine for running the experiments was not available exclusively at all times, thus leading to possibly longer trial durations. The number of completed trials for a fixed time-budget in each case can reveal a rough estimate of the mean duration of a trial, but due to the fact that the machine was not exclusively available, a comparison of the models based on runtime (and thus consumed compute resources) is not really meaningfully possible. Since for all settings at least 10 trials where completed, the top 10 trials for each setting will be looked at in a bit more detail below. The sorting for better vs. worse trials was done based on the ROC-AUC achieved on the validation set.

8.1. Model Selection: Best Ranked Results

Here, the results of the model selection procedure will be shown and discussed. As was described in Section 6.6, for each model, several hyper-parameters were subjected to automatic tuning with a given time-budget. Each specific setting of hyper-parameters (a set of concrete values) is called a trial.

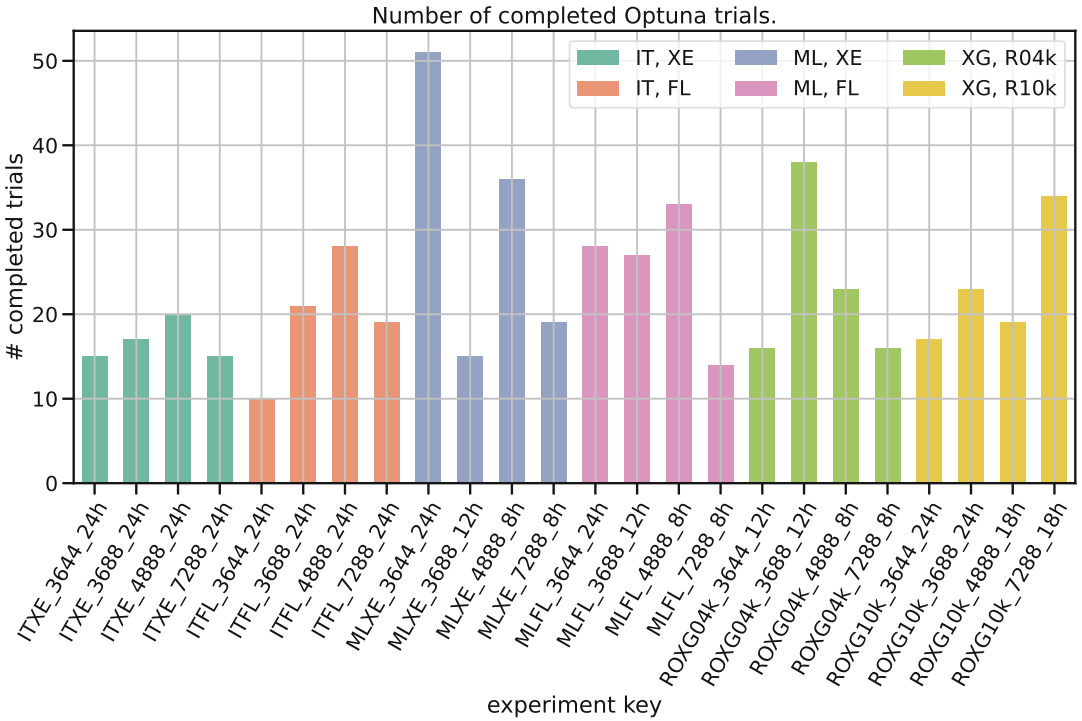

Figure 8.1.: Number of completed hyper-parameter tuning trials using Optuna, per experimental setting (see Tables 7.2.3, 7.2.2 and 7.2.1 for an explanation of the experiment keys). Each trial is one set of values for the respective model's hyper-parameters. IT = InceptionTime, ML = FCN-MLSTM. XE = cross-entropy, FL = focal loss. XG = XGBoost, R04k = ROCKET with 4000 kernels, R10k = ROCKET with 10 000 kernels.

8.1.1. Best Ranked Results Across all Experimental Settings

This is a ranking of results at the level of single trials, i.e., specific settings of hyper-parameters for a model, across all models' trials.

Table 8.1.1 lists the best 10 models, ranked by the mean ROC-AUC across the 5 folds achieved on the validation set.

Table 8.1.1.: Models, windowing parameters and ROC-AUC values for the 10 best results (ranked by mean ROC-AUC on validation set). IT = InceptionTime, ML = FCN-MLSTM. XE = cross-entropy, FL(γ) = focal loss, γ is the focusing parameter. Compare Figure 8.2 for a graphical comparison of the ROC-AUC values for all settings presented here.

Model	Loss Function	AUC Validation	AUC Training	AUC Test	State (h)	Stride (h)	Pred. (h)	Optuna time	Trial Nr.
IT	FL(1)	0.75	0.95	0.73	36	4	4	24h	14
IT	FL(1)	0.74	0.9	0.71	36	4	4	24h	17
IT	FL(1)	0.74	0.92	0.69	36	4	4	24h	4
IT	FL(2)	0.73	0.9	0.66	72	8	8	24h	3
IT	FL(2)	0.72	0.85	0.68	48	8	8	24h	75
ML	FL(2)	0.72	0.9	0.64	36	4	4	24h	1
IT	XE	0.72	0.87	0.65	72	8	8	24h	3
ML	XE	0.72	0.87	0.67	36	4	4	24h	1
IT	FL(5)	0.71	0.84	0.67	48	8	8	24h	1
IT	XE	0.71	0.86	0.64	72	8	8	24h	1

Figure 8.2 shows for each of the trials listed in Table 8.1.1 the ROC-AUC for the training, validation and test sets, for each of the 5 folds, as well as the mean value that was used for ranking.

8.1.2. Insights from Model Selection

- As is evident from Table 8.1.1, as well as from Figure 8.2, the overwhelming majority of the best cases shows the combination of the InceptionTime model with the focal loss loss function. Mostly small values of the focusing parameter γ ($\in [1, 5]$ for the experiments) occur, suggesting that too much influence of hard to classify examples (caused by larger values of the focusing parameter) is not useful.
- The FCN-MLSTM model has two entries with comparable performance, here the loss function does not seem to make a difference. It may be necessary to look at more cases to get an idea of the influence of the loss function for this model.
- The XGBoost GBDT classifier using ROCKET features does not occur anywhere near the top results. The outcomes for the XGBoost models are

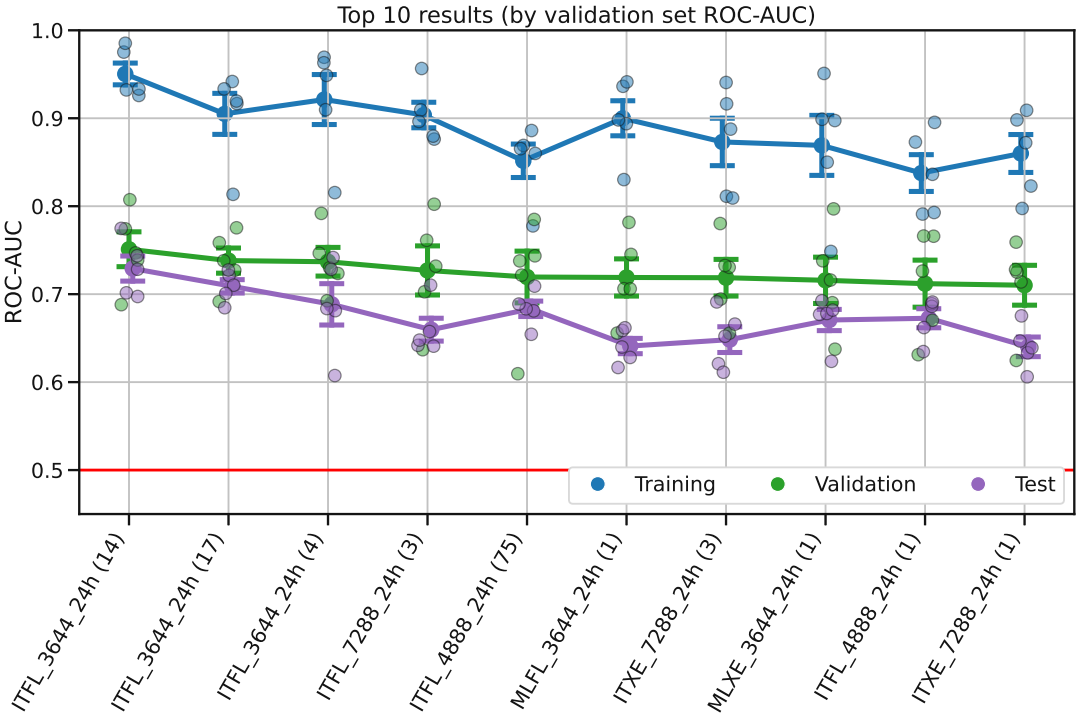

Figure 8.2.: Top 10 best results, ranked by ROC-AUC on the validation set (mean across folds). This plot is the graphical representation of Table 8.1.1. Lines connect mean ROC-AUC values (computed across the respective folds) for each experimental setting / trial (trial number in parenthesis, compare Table 8.1.1). Error-bars show the standard error of the mean. Additionally, each fold's ROC-AUC is plotted as a point for each setting to visualize the spread around the mean. Note that the y-axis is cut below 0.45. IT = InceptionTime, ML = FCN-MLSTM. XE = cross-entropy, FL = focal loss.

shown in more detail in the Appendix in Sections A.1.5 and A.1.6 and discussed in Section A.1.7.

- The windowing setting occurring most often is the (36, 4, 4) hour type. All cases with a prediction length of 8 hours have state length of more than 36 hour setting, i.e., the (36, 8, 8) hour setting does not occur among the top results. This suggests that a state of 36 hours may be to short for predicting more than 4 hours into the future with this high performance (as measeured with ROC-AUC). For the 8 hour prediction length, a state of 48 hours seems to be enough (vs. 72 hours).

- There is a considerable drop in performance from the training data set to the evaluation data set, hinting at possible overfitting. However, there is much less difference in performance between the evaluation and the test set, suggesting that performance on unseen data is to be expected in the range shown here.

8.1.3. Best Ranked Results per Windowing Setting

Figure 8.3 presents the best 10 results per windowing setting. This facilitates the comparison of model trials specific for each windowing setting. Similar to the results shown in Figure 8.2, the performance achieved on the validation set does not differ strongly for the best results, i.e., similar performance can be achieved for a few different models and hyper-parameter settings. For the (72, 8, 8) hour setting, two settings for ROCKET (with 10 000 kernels) occur in the list of best results. While the performance appears acceptable on the validation set, there is a large gap between the the training set and validation set, and a large drop from the validation set to the test set performance. This hints at the strong overfitting that occurs for these models, see Sections A.1.5 and A.1.6 for further details.

8.2. ROC-AUC and Precision-Recall-AUC

While the performance metric ROC-AUC is suitable for selecting the models based on their performance on the validation set, for the predictive capabilities different metrics may be more insightful.

This section will give an overview of the relation of the achieved values for the ROC-AUC metric and the corresponding value for PR-AUC. For a short discussion of the general relationship between ROC-AUC and PR-AUC see also Section 4.4.4.

Figure 8.4 shows plots revealing the relationship between ROC-AUC and PR-AUC, lumped together for all of the top-ranked trials for each experiment setting, with 5 folds each, separately for the training, validation and test sets.

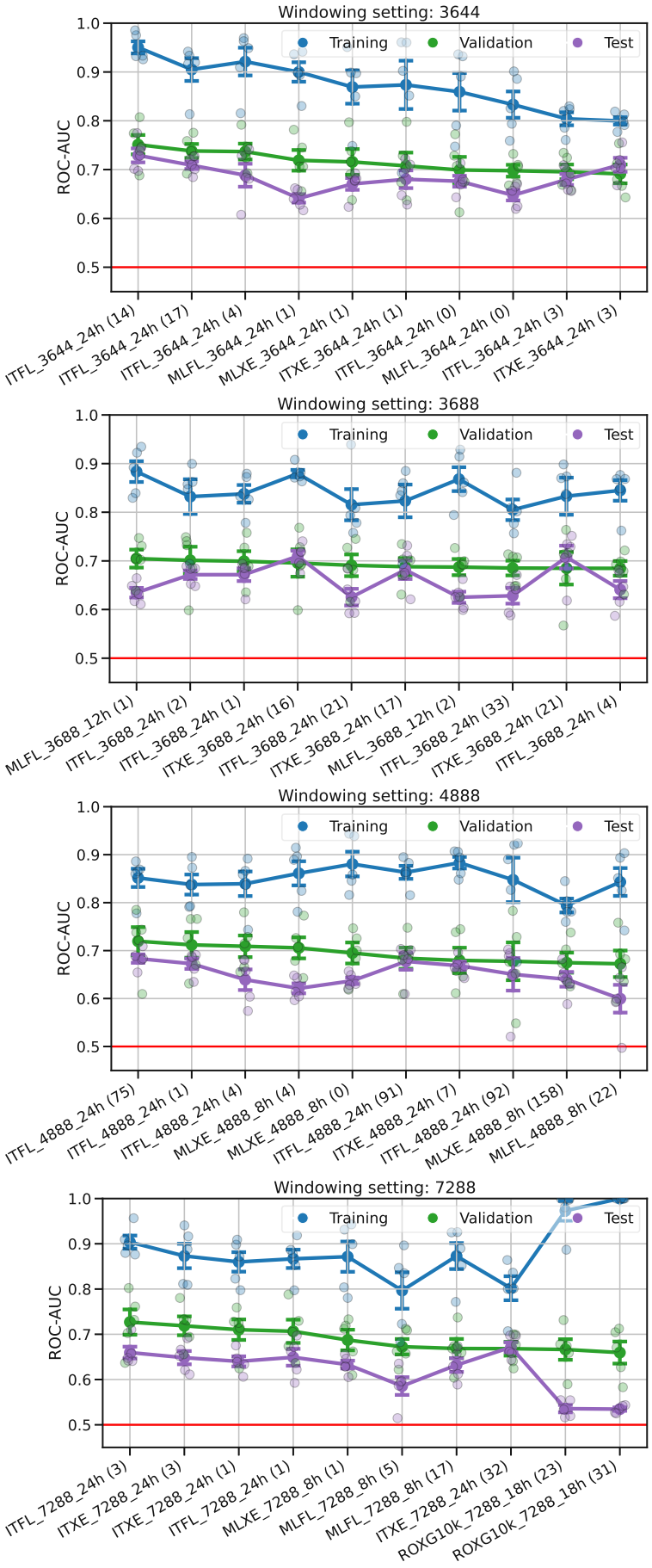

Figure 8.3.: Best trials per windowing setting.

The y-axis of the plots is logarithmically scaled as the PR-AUC values are relatively small.

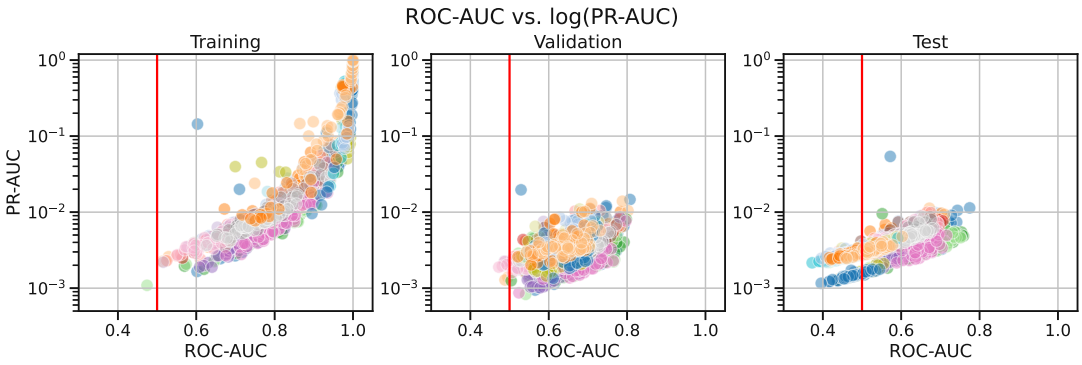

Figure 8.4.: ROC-AUC vs. PR-AUC (area under the Precision-Recall curve). This figure lumps together for each of the 24 settings their top ranked trials, each point in the plot corresponds to 1 single fold (of 5 folds per trial). The plot is meant to show the overall relation between ROC-AUC and PR-AUC for the training, validation and test sets. Note the logarithmically scaled vertical axis. The red vertical line at ROC-AUC = 0.5 corresponds to the no-skill classifier.

In general, there is a monotonic relationship between the ROC-AUC and the PR-AUC metrics. As mentioned in Section 4.4.4, according to (Davis et al. 2006), maximizing ROC-AUC does not necessarily maximize PR-AUC. Joint maximization for both metrics may still be a useful strategy for future experiments, despite the relationship shown for the models used here.

The spread of the values achieved, as well as the disparity of the performance on the different sets (training, validation and test) seen here should also be compared to Figure 8.2.

Section A.2 in the Appendix shows the relationships as depicted in Figure 8.4 for each model family separately.

8.3. Example: Detailed Evaluation

As an example for more detailed evaluation of the performance of a specific model, the ITFL_3644_24h (14) setting is used here, the overall best ranked model and trial (see Figure 8.2 and Table 8.1.1). More concretely, the model corresponding to fold 5 is used, as it shows the smallest gap between training set and evaluation set performance.

Figure 8.5 shows the calibration plot for the model (see Section 4.3.3), for all three data set parts. The model appears not well calibrated, and the distortion away from the line showing perfect calibration is different for the different data set parts. Compensating based on the training data would potentially worsen the situation for the other data set parts, and unseen data in general. Especially

striking is the small range of predicted probabilities, roughly only for the range $[0, 0.4]$ instead of $[0, 1]$.

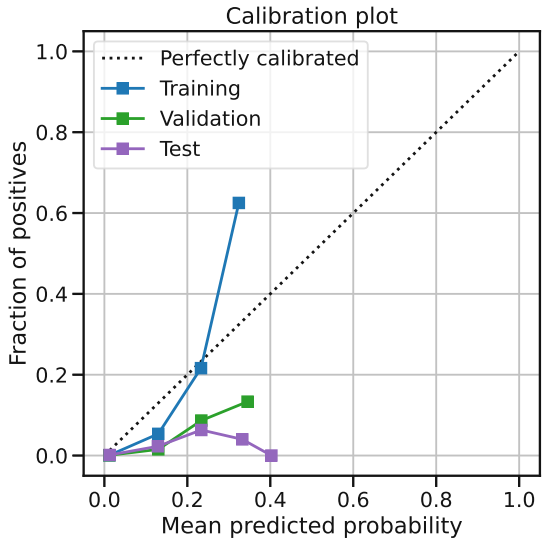

Figure 8.5.: Calibration plot for all three data set parts. Evidently, the model does not properly use the range $[0, 1]$ of probabilities, besides a distortion away from the line showing perfect calibration. Interestingly, the curves differ strongly for the different data sets.

Niculescu-Mizil et al. (2005) state that neural networks predict well calibrated probabilities, but do not specify any details about the networks used for their experiments; very likely, these are only rather simple feed-forward networks and not more complex networks such as CNNs. Thus, it is questionable that this statement holds for more general and complex neural networks, as the results here suggest (similarly bad calibration was observed for most results obtained from the experiments). In fact, Guo et al. investigated more recent deep neural networks (and CNNs in particular) with respect to their calibration and found them to be not well calibrated (Guo et al. 2017, p. 1322) and suggest methods calibrating the outputs.

In general, it may be beneficial to calibrate the outputs of the models used in this thesis, as they are envisioned to be used in a decision system where further processing would be applied to the predicted probabilities (be it only for quantization to class labels, involving costs for the decisions).

Figure 8.6 shows two plots: on the left, for all examples of the training data set, the predicted probabilities are shown, where the true class as well as the predicted class are shown. A threshold was set for quantizing the predicted probabilities to class labels. The right part shows the corresponding confusion matrix with the counts for the four fundamental cases. Both plots clearly reveal that the separation of the classes is not well pronounced for this model.

Figure 8.7 shows the ROC plot (left) and the Precision-Recall curve (right)

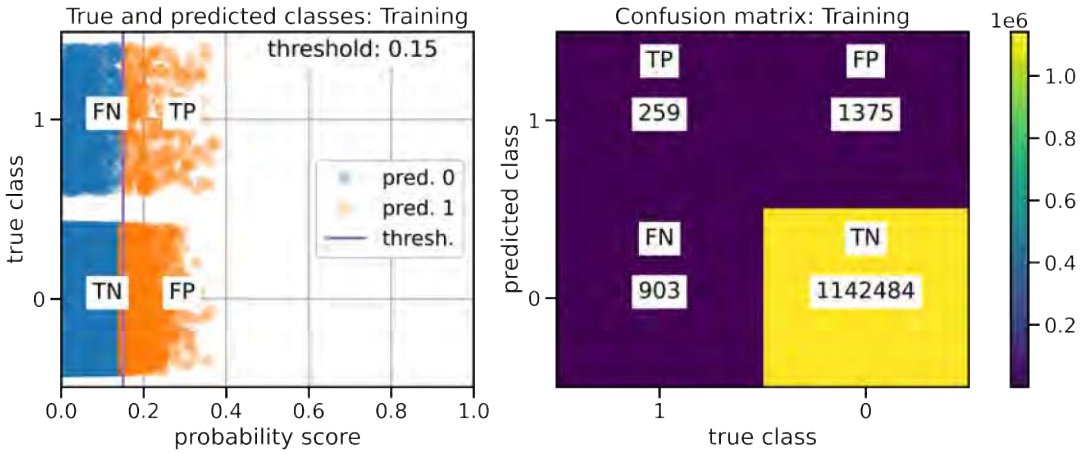

Figure 8.6.: Left: Plot showing the predicted probabilities for the target class for all examples of the training data set. The true class is given by the vertical position, the predicted class by color. Quantization from predicted probability to class label is done using a threshold. Examples quantized to the target class are to the right of the threshold. Right: Confusion matrix, showing the counts for the four fundamental cases.

for all three data set parts. The operating points corresponding to the threshold chosen and used in Figure 8.6 are marked on the respective curves. Note the operating points on the far left in the ROC curve plot. The PR curve shows the rather low values for precision and recall in the range of a few percent, see Table 8.3.1 for details.

Table 8.3.1.: Resulting numbers corresponding to the operating points shown in Figure 8.7, using the threshold shown in Figure 8.6. TPR = True Positive Rate, FPR = False Positive Rate.

Data	Precision	Recall (TPR)	FPR
Training	0.158	0.223	0.001
Validation	0.05	0.096	0.001
Test	0.044	0.054	0.002

8.4. Summary and Outlook

In the following, a few aspects of the results obtained and shown above are discussed, as are some points to be addressed in future investigations related to the overarching scenario of this thesis, i.e. predicting errors in Hybrid Fiber-Coaxial (HFC) networks to facilitate better planning of resources for network maintenance.

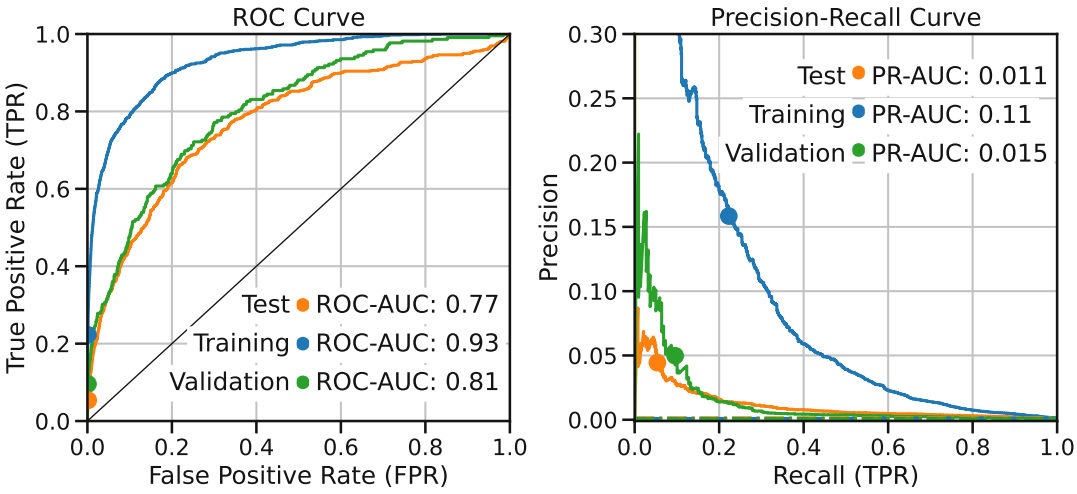

Figure 8.7.: Left: ROC plot for all three data set parts. The points marked on each curve correspond to the specific threshold also used in Figure 8.6. Right: Precision-Recall curve for all three data set parts. The drop in performance from training to validation and test set parts appears more pronounced in this plot.

Model Performance

In general, the model performance based on the ranking using the ROC-AUC seems acceptable at first sight. More detailed evaluation with metrics like Precision and Recall reveal rather poor performance. However, the exemplary evaluation for a specific model shown above should be taken with a grain of salt, as the concrete cost factors are not known and would need to be involved for evaluating the concrete usefulness of the models in a decision system. This is discussed next.

In general, the results suggest that overfitting might be a problem across all models used, despite several methods for regularization employed during model training for all models.

Scenario and Costs, Revisited

We shall briefly return to the overarching scenario as described in Section 1.1. The costs discussed there, C_{m} for maintenance (these are the costs for FP, C_{FP}) and C_{uc} (uc as in unhappy customer) (these are the costs for FN, C_{FN}), could depend on the network segment (location):

- Recall that the network segments, the fiber node areas (FNAs), have different numbers of Cable Modems (CMs) (leaf nodes, customers), see Sections 2.1 and 3.1.2. Thus, errors occurring in larger FNA would lead to a larger number of unhappy customers.
- Larger FNAs typically cover larger spatial regions, this may result in increased distances to travel between the devices to check, besides the larger amount of devices present in such FNAs.

Thus, evaluating the results further based on costs was decided to be far beyond the scope of this thesis; this is a task that would need be carried out with an actual implementation in an existing network with appropriate monitoring and appropriate in-situ evaluation.

Performance Ceiling

Kuhn et al. (2013, p. 79) mention the notion of a performance ceiling for a given data set, manifesting as a number of different models (or different hyper-parameter settings of a specific model) that show similar or equivalent performance. Revisiting Figure 8.2 (and Table 8.1.1), it seems that the model selection carried out established such a performance ceiling for the data set, or data sets, when considering the different windowing settings with the respective (pre-) processing applied. This of course does not exclude the possibility that better performance can be achieved, with either more tuning, different models, different preprocessing or a completely different approach starting from the the aggregated or even the raw data.

Region-Specific Models and Data

As was shown in Chapter 3, the data show some heterogeneity that is related to the different (spatial) network segments. Potentially, it would make sense to use several different models, where each one covers only a certain spatial region of more coherent network state. However, as Sections 5.8 and especially 5.8.3 showed, the number of instances of the positive class is quite low for the data used that span all available regions; when using region specific data, a (much) longer time-span of data would be necessary to cover a sufficient number of positive class cases for meaningfully training models.

APPENDIX A

Additional Plots and Insights

A.1. Best Ranked Results For each Experimental Setting

In the following, for each of the 24 settings shown in Table 7.2.3, a figure similar to Figure 8.2 is shown. Here, the figures show for each setting the top ranked trials (according to ROC-AUC on the validation set) for only this setting, i.e., the (top ranked) results of the model selection per setting.

Lines connect the mean values computed across folds, points show the respective value for each fold. Each trial corresponds to a certain set of values for the hpyer-parameters.

A.1.1. InceptionTime with Cross-entropy (ITXE)

See Figure A.1.

A.1.2. InceptionTime with Focal Loss (ITFL)

See Figure A.2.

A.1.3. FCN-MLSTM with Cross-entropy (MLXE)

See Figure A.3.

A.1.4. FCN-MLSTM with Focal Loss (MLFL)

See Figure A.4.

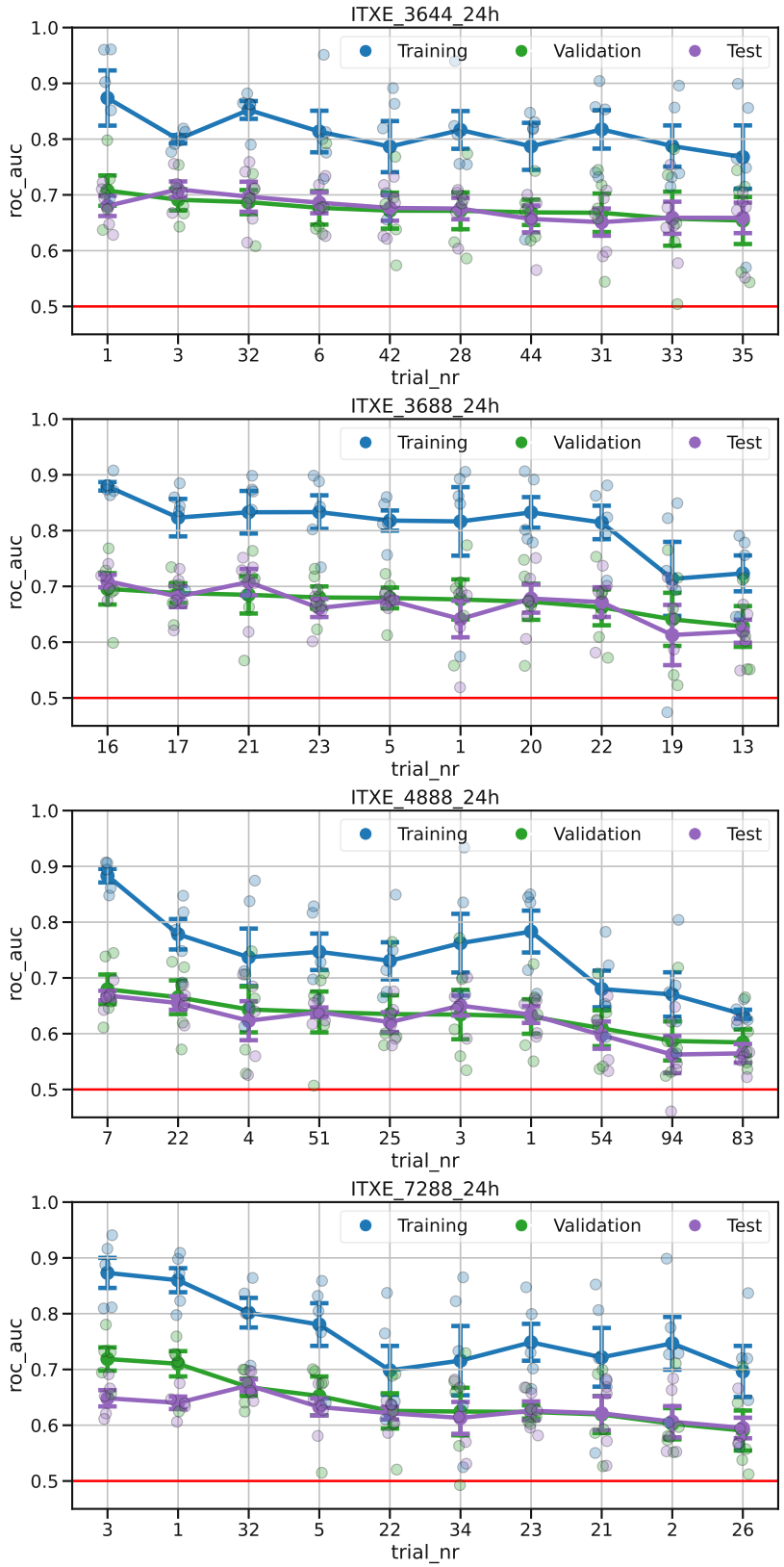

Figure A.1.: InceptionTime with Cross-entropy (ITXE)

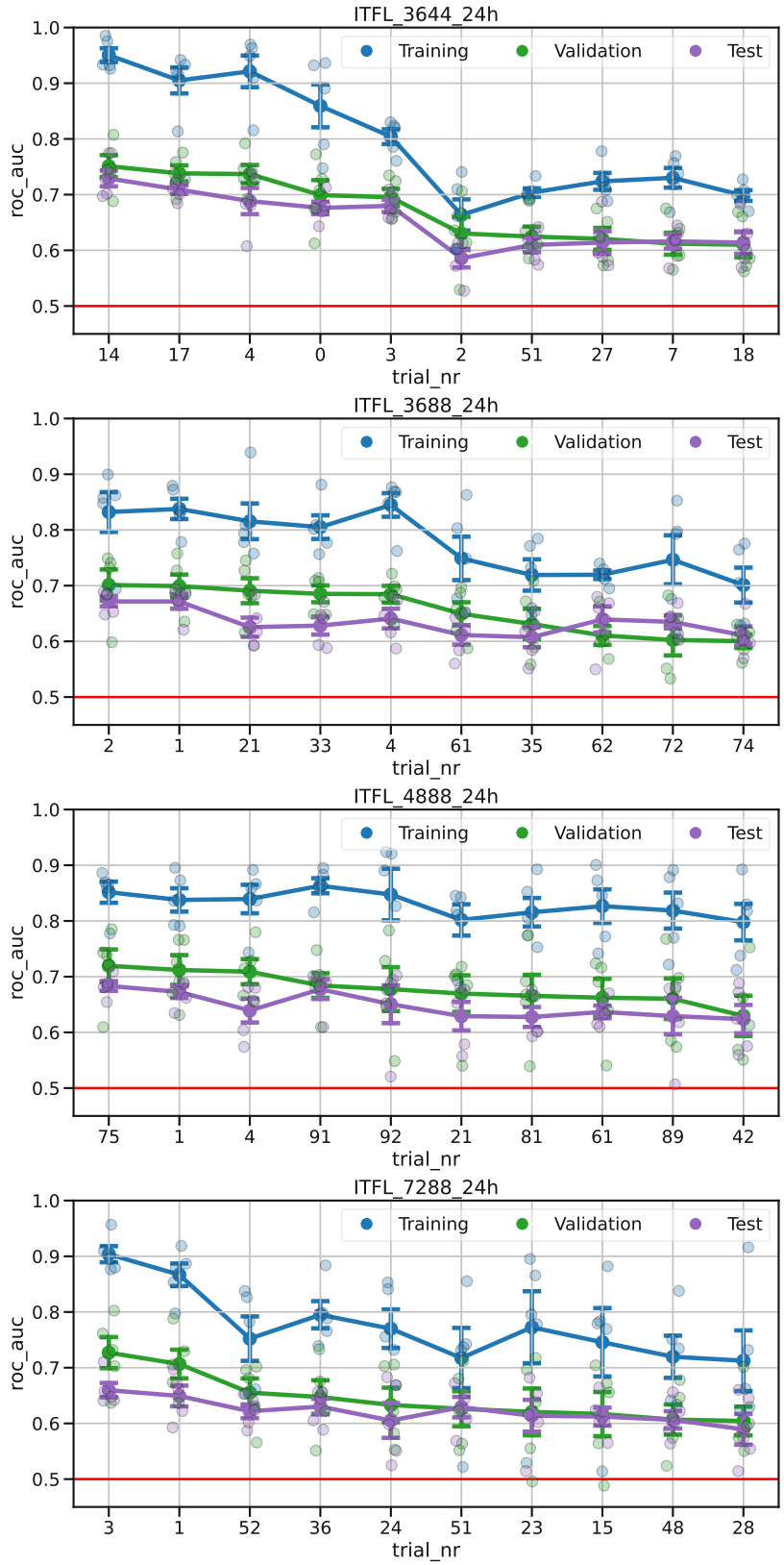

Figure A.2.: InceptionTime with Focal Loss (ITFL)

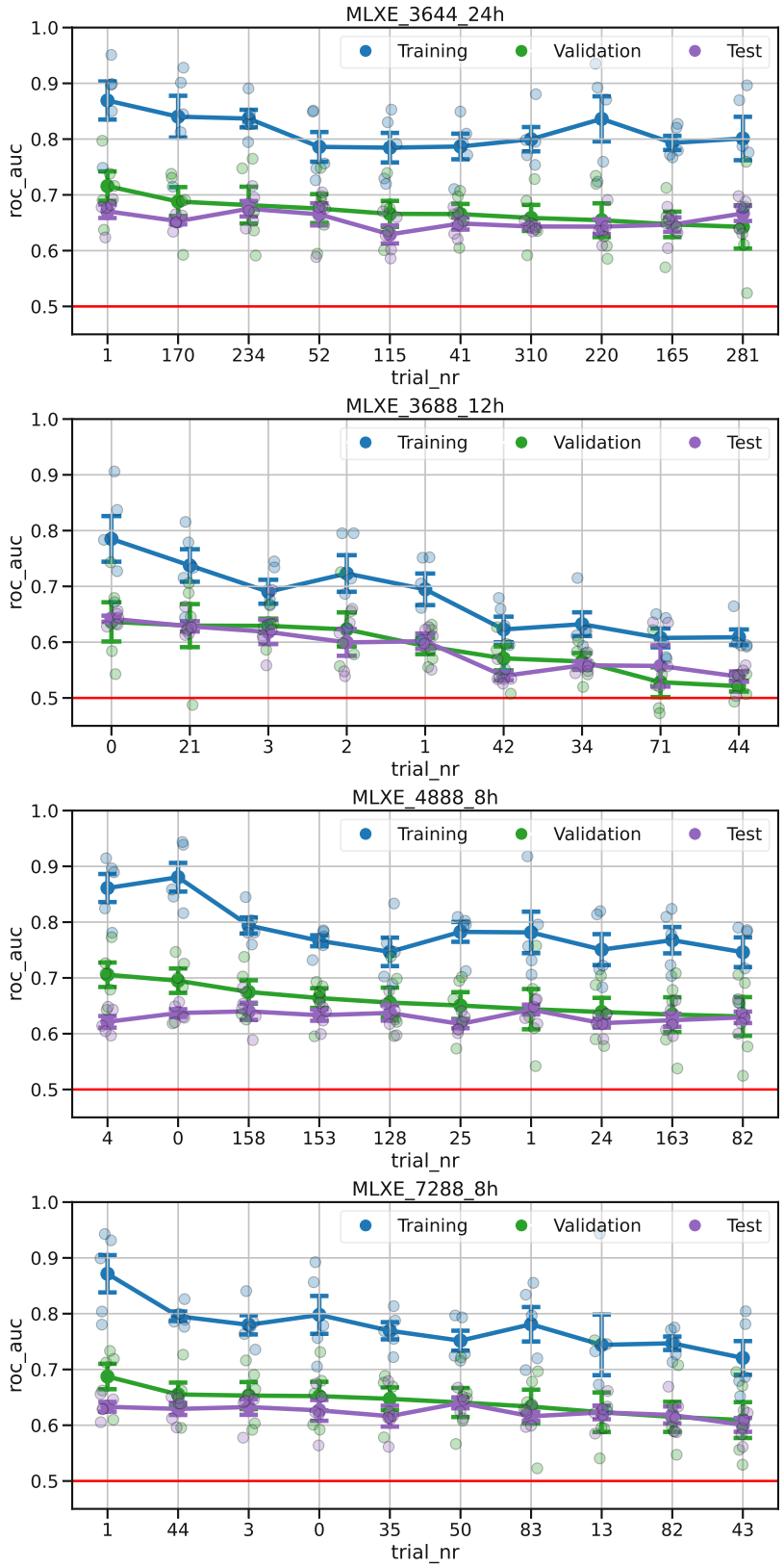

Figure A.3.: FCN-MLSTM with Cross-entropy (MLXE)

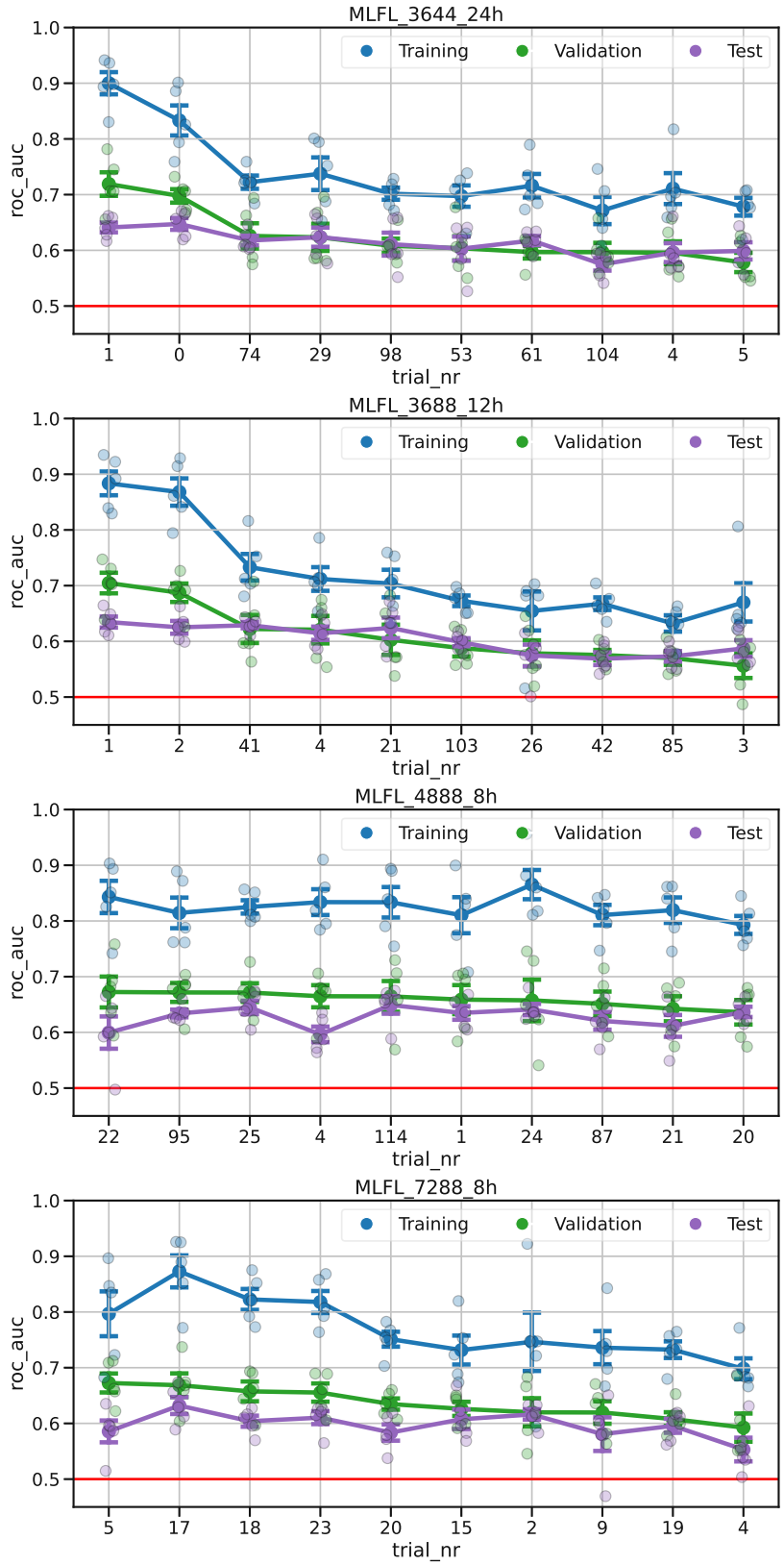

Figure A.4.: FCN-MLSTM with Focal Loss (MLFL)

A.1.5. XGBoost Classifier with ROCKET Features, 4000 Kernels

See Figure A.5.

A.1.6. XGBoost Classifier with ROCKET Features, 10000 Kernels

See Figure A.6.

A.1.7. Discussion of XGBoost Results

The results for the combination of the ROCKET features and the XGBoost GBDT classifier shown in Figures A.5 and A.6 (see also Figure A.9) reveal severe overfitting to the training data. Especially for the version with 4000 kernels, the performance on the test set is worse than random guessing (as is evident from the ROC-AUC being below 0.5). The situation is only marginally better for the 10 000 kernel setting, only for the (72, 8, 8) hour setting there are a handful of trials that show ROC-AUC above 0.5 and thus any skill.

Several methods for regularization were applied to prevent overfitting, see Sections 6.5.2 and 7.4.2 for a discussion of the available options and the methods used, respectively. It is at present not known why overfitting of this magnitude occurred; inspection of the actual values for the amount of regularization applied did not reveal any clear patterns that would seem to prevent or cause the overfitting behavior. Additional inspection of the (hyper-) parameters relevant to the handling of the class imbalance (class weights, limiting upgrade steps or undersampling; see Sections 6.5.3 and 7.4.2) did also not reveal any systematic relationship between these settings and overfitting.

It is not clear whether a different strategy for ranking the models would reveal more successful cases, such as incorporating the performance gap between training and validation set performance instead of only ranking by the validation set ROC-AUC.

Potentially, different settings for the generation of the random kernels could increase the performance and decrease the overfitting here; this would require (possibly extensive) tuning that was beyond the scope of this thesis.

A.2. ROC-AUC and Precision-Recall-AUC per Model Family

In the following, the relationships between ROC-AUC and PR-AUC are shown for each model group separately, these plots should also be compared to the figures in Section A.1, showing the top-ranked trials per model group.

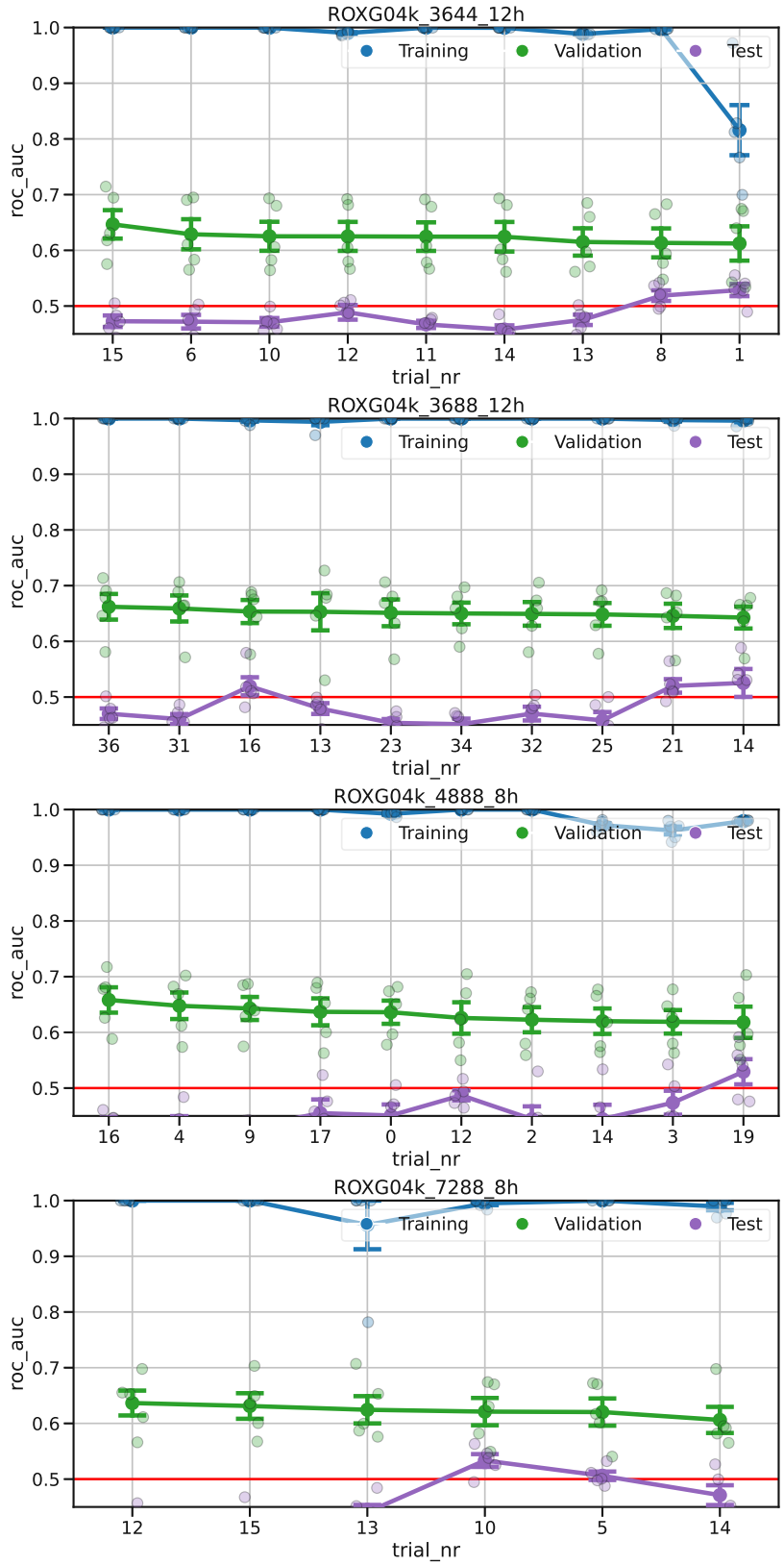

Figure A.5.: XGBoost Classifier with ROCKET Features, 4000 Kernels

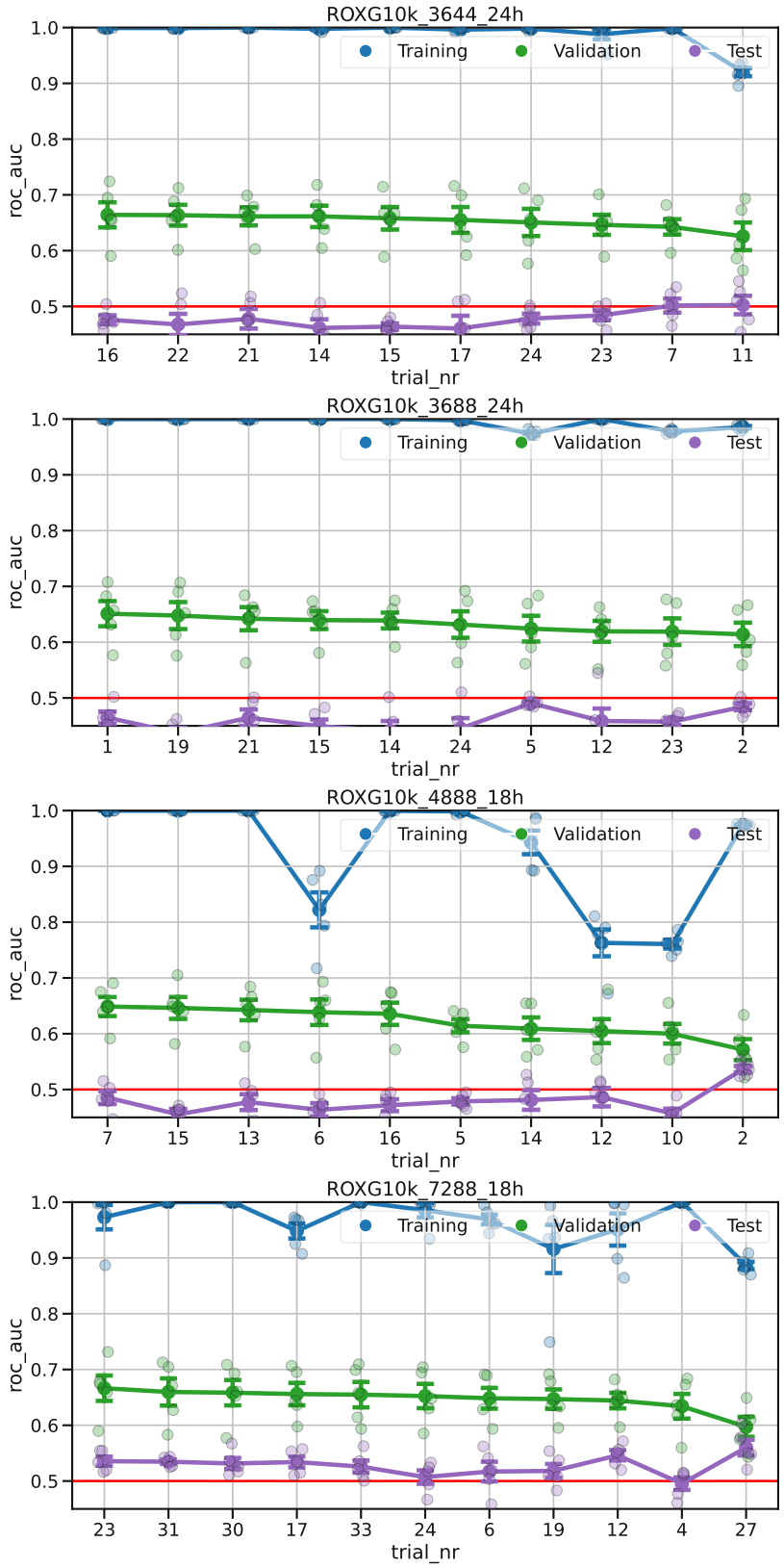

Figure A.6.: XGBoost Classifier with ROCKET Features, 10 000 Kernels

Figure A.7 shows the plots for the InceptionTime models, with cross-entropy and focal loss loss functions. Figure A.8 shows the plots for the FCN-MLSTM models, with cross-entropy and focal loss loss functions.

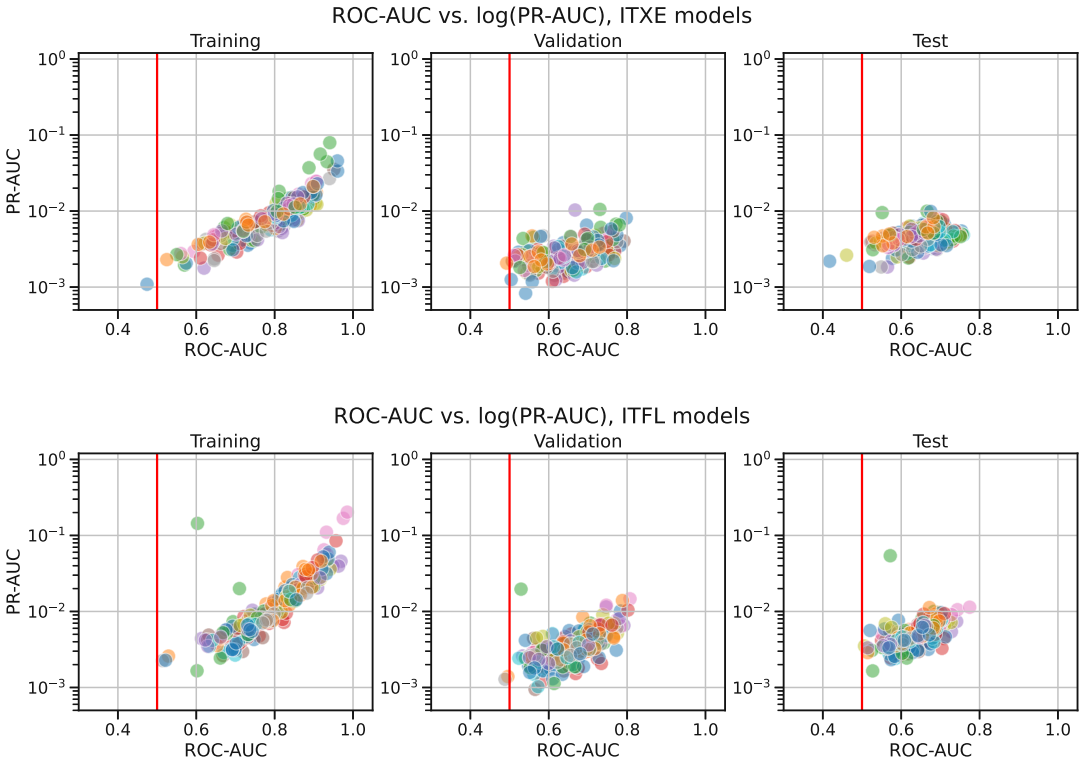

Figure A.7.: InceptionTime (ITXE and ITFL) models. Color is per trial.

Figure A.9 shows the plots for the XGBoost models using ROCKET features. As can be seen, the disparity between both metrics' values achieved on the training and validation sets is striking for these models, hinting at severe overfitting. The test set shows values similar to the validation set.

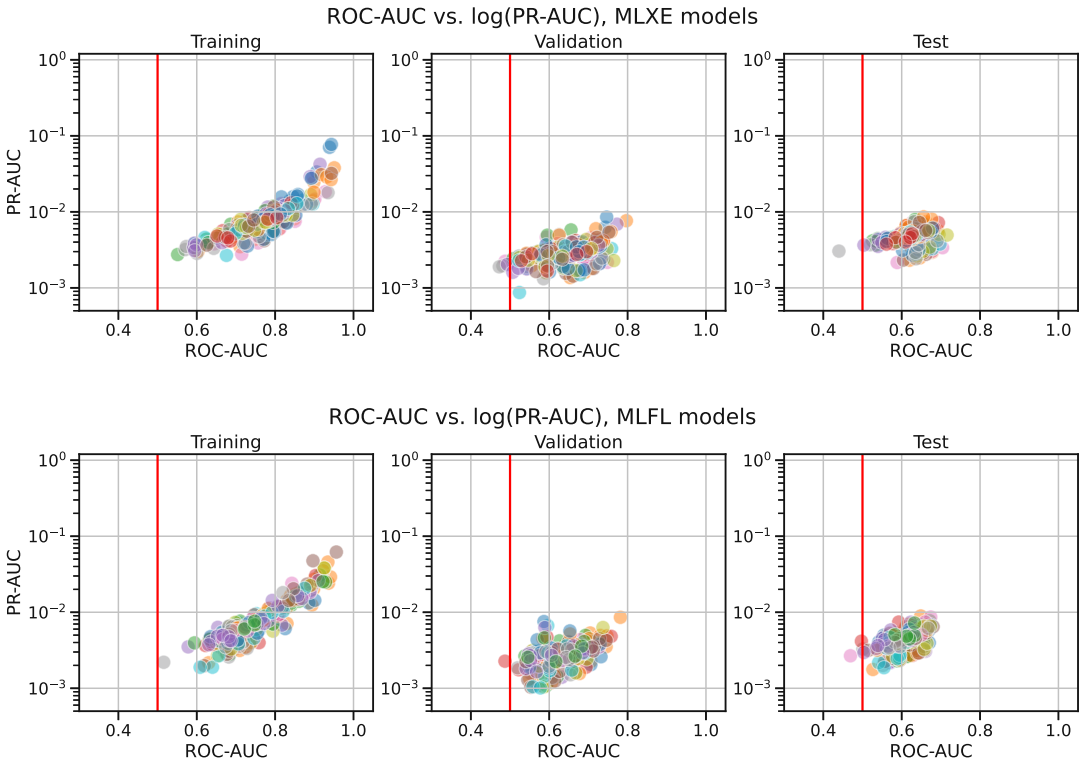

Figure A.8.: FCN-MLSTM (MLXE and MLFL) models. Color is per trial.

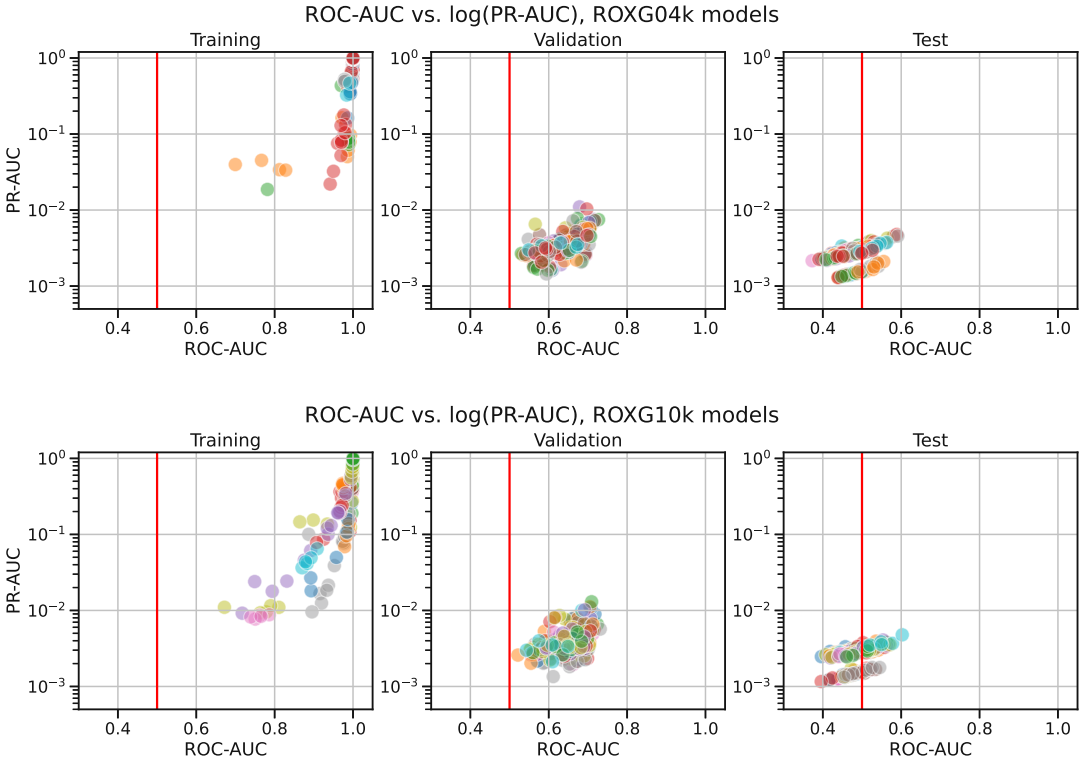

Figure A.9.: XGBoost models with ROCKET features. Color is per trial. For these models, there is a much larger difference between the scores achieved on the training set as compared to the validation and test sets, hinting at severe overfitting.

Acronyms

ARIMA Auto-Regressive Integrated Moving Average. 53

AUC Area Under the Curve. 45, 47–49, 85, 91, 103, 105, 107, 110, 118

BN Batch Normalization. 77

CCER Corrected Codword Error Rate. 17, 24, 26, 52

CER Codword Error Rate. 12, 17, 24, 26, 52

CM Cable Modem. 9, 11–13, 15–17, 110

CMTS Cable Modem Termination System. 10

CNN Convolutional Neural Network. viii, ix, 68–71, 73, 75–77, 79–83, 85, 108

CPD Common Path Distortion. 13

CV Cross-Validation. 61, 62, 90, 97

DNN Deep Neural Network. 53, 68, 71, 72, 94, 100

DOCSIS Data Over Cable Service Interface Specification. 1, 11

DS Downstream. 11, 16, 17, 26

FCN Fully Convolutional Network. viii, 53, 72, 76, 79, 94, 97, 98, 102–104, 121

FDD Frequeny-Domain Duplex. 11

FFNN Feed Forward Neural Network. 86

FN False Negative. 4, 5, 42, 43, 46, 110

FNA Fiber Node Area. See also fiber node area. 10, 11, 13–29, 51–54, 56, 57, 64, 110

FP False Positive. 3–5, 42, 43, 46, 110

FPR False Positive Rate. 47, 48, 109

GAP Global Average Pooling. 73, 77

GBDT Gradient Boosted Decision Trees. ix, 82, 86–89, 94, 98, 103, 118

HFC Hybrid Fiber-Coaxial. iii, v, vii, 1, 6, 9–13, 109

IR Imbalance Ratio. 26–28, 32, 33, 39, 40, 62

ISP Internet Service Provider. 9, 10

LSTM Long Short-Term Memory. 53, 72, 77–79

MLP Multi-Layer Perceptron. 68, 70, 71

MLSTM Multivariate Long Short-Term Memory. viii, 76, 94, 97, 98, 102–104, 121

MTS Multivariate Time Series. 13, 14, 19, 74

MTSC Multivariate Time-Series Classification. 67, 68, 71, 72

PR Precision-Recall. viii, 48, 49, 105, 107, 109, 118

ReLU Rectified Linear Unit. 69, 70, 77, 78, 80

RNN Recurrent Neural Network. 77

ROC Receiver Operating Characteristic. viii, 43, 45–49, 85, 91, 103, 105, 107–110, 118

ROCKET RandOm Convolutional KErnel Transform. 53, 54, 72, 79–82, 86, 94, 98–100, 102, 103, 105, 118, 121

SaE Squeeze-and-Excite. 77–79

SMOTE Synthetic Minority Oversampling Technique. 33, 34

SNR Signal-to-Noise Ratio. 12, 17

TN True Negative. 4, 46

TP True Positive. 3, 4, 46

TPR True Positive Rate. 47, 48, 109

TSC Time-Series Classification. viii, 2, 6, 53, 67, 68, 71–73, 75, 77, 79, 81

US Upstream. 10, 16, 17, 26

WAN Wide-Area Network. 1

Glossary

fiber node Also fiber optical node. A unit within an HFC-network that converts between optical and electrical signals. Interfaces to fiber optic and coaxial cables. 10

fiber node area Also fiber node service area. The area (neighborhood) served by a fiber node within an HFC network. 10, 110, 125

micro-reflections Reflections with very short temporal distance between main signal and reflection (on the order of 1 µs). These can occur in the downstream as well as the upstream direction and have a detrimental effect on the transmission. For details see (CableLabs 2017, p. 22). 17, 24

Bibliography

Aggarwal, Charu C. (2017). *Outlier Analysis*. 2nd ed. Cham: Springer International Publishing. DOI: 10.1007/978-3-319-47578-3. URL: http://link.springer.com/10.1007/978-3-319-47578-3 (cit. on pp. 3, 33–35, 40, 45, 48).

Akiba, Takuya et al. (2019). "Optuna: A Next-generation Hyperparameter Optimization Framework." In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2623–2631. DOI: 10.1145/3292500.3330701. arXiv: 1907.10902. URL: https://github.com/pfnet/optuna (cit. on p. 97).

Bagnall, Anthony et al. (2017). "The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances." In: *Data Mining and Knowledge Discovery* 31, pp. 606–660. DOI: 10.1007/s10618-016-0483-9 (cit. on pp. 53, 72).

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer (cit. on pp. 35–37, 43, 83–85, 88).

Branco, Paula, Luís Torgo, and Rita P. Ribeiro (Aug. 2016). "A Survey of Predictive Modeling on Imbalanced Domains." In: *ACM Computing Surveys* 49.2. ISSN: 0360-0300. DOI: 10.1145/2907070 (cit. on pp. 31–33, 40).

Buja, Andreas, Werner Stuetzle, and Yi Shen (2005). "Loss Functions for Binary Class Probability Estimation and Classification: Structure and Applications." In: URL: http://stat.wharton.upenn.edu/~buja/PAPERS/paper-proper-scoring.pdf (last visited on 2022-04-24) (cit. on p. 37).

CableLabs (2017). *Data-Over-Cable Service Interface Specifications DOCSIS 3.0*. Version C01. Cable Television Laboratories. URL: https://www.cablelabs.com/specifications/CM-SP-PHYv3.0 (last visited on 2022-01-07) (cit. on pp. 9, 11, 127).

Chawla, Nitesh V. et al. (2002). "SMOTE: Synthetic Minority Over-sampling Technique." In: *Journal of Artificial Intelligence Research* 16, pp. 321–357 (cit. on p. 33).

Chen, Tianqi and Carlos Guestrin (2016). "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, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785 (cit. on pp. 82, 87, 89, 90).

Crocker, Harry J. and Aaron W. Costall (2021). "An InceptionTime-Inspired Convolutional Neural Network to Detect Cardiac Abnormalities in Reduced-Lead ECG Data." In: *Computing in Cardiology (CinC) 2021* 48, pp. 1–4 (cit. on p. 74).

Davis, Jesse and Mark Goadrich (2006). "The Relationship between Precision-Recall and ROC Curves." In: *Proceedings of the 23rd International Conference on Machine Learning*. ICML '06. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, pp. 233–240. DOI: 10.1145/1143844.1143874. URL: https://doi.org/10.1145/1143844.1143874 (cit. on pp. 49, 107).

Dempster, Angus, François Petitjean, and Geoffrey I. Webb (Sept. 2020). "ROCKET: Exceptionally Fast and Accurate Time Series Classification Using Random Convolutional Kernels." In: *Data Minining and Knowledge Discovery* 34.5, pp. 1454–1495. ISSN: 1384-5810. DOI: 10.1007/s10618-020-00701-z. URL: https://doi.org/10.1007/s10618-020-00701-z (cit. on pp. 53, 72, 79–82).

dmlc (2021). *XGBoost - eXtreme Gradient Boosting*. URL: https://github.com/dmlc/xgboost (last visited on 2023-03-04) (cit. on pp. 82, 89).

dmlc (2022a). *XGBoost - Parameter Tuning - Handle Imbalanced Dataset*. URL: https://xgboost.readthedocs.io/en/stable/tutorials/param_tuning.html#handle-imbalanced-dataset (last visited on 2023-03-04) (cit. on p. 90).

dmlc (2022b). *XGBoost - Parameters - max delta step*. URL: https://xgboost.readthedocs.io/en/stable/parameter.html#parameters-for-tree-booster (last visited on 2023-03-04) (cit. on p. 90).

Duda, Richard O., Peter E. Hart, and David G. Stork (2001). *Pattern classification*. 2nd ed. Wiley (cit. on pp. 35, 41, 47).

Dumoulin, Vincent and Francesco Visin (Mar. 2018). "A guide to convolution arithmetic for deep learning." In: *ArXiv e-prints*. eprint: 1603.07285. URL: https://arxiv.org/abs/1603.07285 (cit. on p. 70).

Dumoulin, Vincent and Francesco Visin (2019). *Github Repository: Convolution arithmetic*. URL: https://github.com/vdumoulin/conv_arithmetic (last visited on 2022-05-28) (cit. on p. 70).

Esling, Philippe and Carlos Agon (Dec. 2012). "Time-Series Data Mining." In: *ACM Comput. Surv.* 45.1. ISSN: 0360-0300. DOI: 10.1145/2379776.2379788. URL: https://doi.org/10.1145/2379776.2379788 (cit. on pp. 2, 56).

Fawcett, Tom (2006). "An introduction to ROC analysis." In: *Pattern Recognition Letters* 27, pp. 861–874 (cit. on pp. 35, 46, 47).

Fernández, Alberto et al. (2018). *Learning from Imbalanced Data Sets*. Cham: Springer. ISBN: 9783319980744. DOI: 10.1007/978-3-319-98074-4 (cit. on pp. 5, 26, 31–36, 39, 42, 47).

Friedman, Jerome H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine." In: *The Annals of Statistics* 29.5, pp. 1189–1232. ISSN: 00905364. URL: http://www.jstor.org/stable/2699986 (last visited on 2022-11-26) (cit. on pp. 88, 89).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press (cit. on pp. 61, 69–71, 77, 84–86, 90–92).

Guo, Chuan et al. (2017). "On Calibration of Modern Neural Networks." In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, pp. 1321–1330 (cit. on pp. 44, 108).

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2017). *The Elements of Statistical Learning. Data Mining, Inference, and Prediction*. 2nd ed. New York: Springer (cit. on pp. 53, 86–89, 92).

He, Haibo and Yunqian Ma, eds. (2013). *Imbalanced Learning. Foundations, Algorithms, and Applications*. Wiley. ISBN: 9781118646106 (cit. on pp. 3, 28, 31–33, 35, 45–48, 62).

Hochreiter, Sepp and Jürgen Schmidhuber (Nov. 1997). "Long Short-Term Memory." In: *Neural Computation* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (cit. on p. 77).

Hu, Jie et al. (2020). "Squeeze-and-Excitation Networks." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.8, pp. 2011–2023. DOI: 10.1109/TPAMI.2019.2913372 (cit. on p. 77).

Ismail Fawaz, Hassan et al. (July 2019). "Deep Learning for Time Series Classification: A Review." In: *Data Mining and Knowledge Discovery* 33 (4), pp. 917–963. ISSN: 1384-5810. DOI: 10.1007/s10618-019-00619-1. URL: https://doi.org/10.1007/s10618-019-00619-1 (cit. on pp. 71, 72, 76).

Ismail Fawaz, Hassan et al. (2020). "InceptionTime: Finding AlexNet for time series classification." In: *Data Mining and Knowledge Discovery* 34 (6),

pp. 1936–1962. DOI: 10.1007/s10618-020-00710-y (cit. on pp. 53, 68, 71–76).

Johnson, Justin M. and Taghi M. Khoshgoftaar (2019). "Survey on deep learning with class imbalance." In: *Journal of Big Data* 6, pp. 1–54 (cit. on pp. 28, 31, 34, 43).

Karim, Fazle et al. (2018). "LSTM Fully Convolutional Networks for Time Series Classification." In: *IEEE Access* 6, pp. 1662–1669. DOI: 10.1109/ACCESS.2017.2779939 (cit. on pp. 68, 72, 76).

Karim, Fazle et al. (2019). "Multivariate LSTM-FCNs for time series classification." In: *Neural Networks* 116, pp. 237–245. DOI: https://doi.org/10.1016/j.neunet.2019.04.014 (cit. on pp. 53, 72, 76–79).

Keller, Andreas (2011). *Breitbandkabel und Zugangsnetze*. 2nd ed. Berlin: Springer-Verlag (cit. on p. 12).

Krawczyk, Bartosz (Nov. 2016). "Learning from imbalanced data: open challenges and future directions." In: *Progress in Artificial Intelligence* 5.4, pp. 221–232. ISSN: 2192-6360. DOI: 10.1007/s13748-016-0094-0. URL: https://doi.org/10.1007/s13748-016-0094-0 (cit. on pp. 32, 34, 62).

Kuhn, Max and Kjell Johnson (2013). *Applied Predictive Modeling*. New York: Springer. DOI: 10.1007/978-1-4614-6849-3 (cit. on pp. 2, 3, 32, 39, 43, 44, 47, 52, 61, 62, 90, 111).

Kuhn, Max and Kjell Johnson (2020). *Feature Engineering and Selection. A Practical Approach for Predictive Models*. New York: Chapman and Hall/CRC. DOI: 10.1201/9781315108230 (cit. on pp. 26, 34, 43, 47, 51, 54, 86, 91).

Lakshmanan, Valliappa, Sara Robinson, and Michael Munn (2020). *Machine Learning Design Patterns. Solutions to Common Challenges in Data Preparation, Model Building, and MLOps*. O'Reilly. ISBN: 9781098115784 (cit. on pp. 2, 52).

Large, David and James Farmer (2009). *Broadband Cable Access Networks*. The Morgan Kaufmann Series in Networking. Burlington, MA: Morgan Kaufmann. DOI: 10.1016/B978-0-12-374401-2.X1000-5 (cit. on pp. 11, 12).

Lin, Tsung-Yi et al. (2017). "Focal Loss for Dense Object Detection." In: *2017 IEEE International Conference on Computer Vision (ICCV)*. Venice, Italy: IEEE, pp. 2999–3007. DOI: 10.1109/ICCV.2017.324 (cit. on pp. 37, 38).

Löning, Markus et al. (2019). "sktime: A Unified Interface for Machine Learning with Time Series." In: DOI: 10.48550/ARXIV.1909.07872. URL: https://arxiv.org/abs/1909.07872 (cit. on p. 82).

Mathworks (2022). *Basic Structure of a Convolutional Neural Network*. URL: https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html (last visited on 2022-03-26) (cit. on p. 69).

Metzen, Jan Hendrik (2023). *Comparison of Calibration of Classifiers*. URL: https://scikit-learn.org/stable/auto_examples/calibration/plot_compare_calibration.html (last visited on 2023-03-05) (cit. on p. 44).

Mills, Terence C. (2019). *Applied Time Series Analysis*. Academic Press. ISBN: 9780128131183 (cit. on p. 53).

Niculescu-Mizil, Alexandru and Rich Caruana (2005). "Predicting Good Probabilities with Supervised Learning." In: *Proceedings of the 22nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: Association for Computing Machinery, pp. 625–632. ISBN: 1595931805. DOI: 10.1145/1102351.1102430. URL: https://doi.org/10.1145/1102351.1102430 (cit. on pp. 44, 108).

Oguiza, Ignacio (2020a). *tsai - A state-of-the-art deep learning library for time series and sequential data*. URL: https://github.com/timeseriesAI/tsai (last visited on 2023-03-04) (cit. on p. 76).

Oguiza, Ignacio (2020b). *tsai - FCN-MLSTM model*. URL: https://timeseriesai.github.io/tsai/models.rnn_fcn#mlstm_fcn (last visited on 2023-02-25) (cit. on p. 79).

Oguiza, Ignacio (2020c). *tsai - InceptionTimePlus model*. URL: https://timeseriesai.github.io/tsai/models.inceptiontimeplus.html (last visited on 2023-02-25) (cit. on p. 76).

Oguiza, Ignacio (2020d). *tsai - ROCKET feature extractor*. URL: https://timeseriesai.github.io/tsai/models.rocket_pytorch.html (last visited on 2023-02-25) (cit. on p. 82).

optuna (2021). *Optuna: A hyperparameter optimization framework*. URL: https://github.com/optuna/optuna (last visited on 2023-03-04) (cit. on p. 97).

Pedregosa, F. et al. (2011). "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12, pp. 2825–2830 (cit. on p. 20).

Prati, Ronaldo C., Gustavo E. A. P. A. Batista, and Maria Carolina Monard (2011). "A Survey on Graphical Methods for Classification Predictive Performance Evaluation." In: *IEEE Transactions on Knowledge and Data Engineering* 23.11, pp. 1601–1618. DOI: 10.1109/TKDE.2011.59 (cit. on p. 43).

Provost, Foster and Tom Fawcett (2001). "Robust Classification for Imprecise Environments." In: *Machine Learning* 42.3, pp. 203–231. ISSN: 1573-0565. DOI: 10.1023/A:1007601015854. URL: https://doi.org/10.1023/A:1007601015854 (cit. on pp. 41, 47).

Ruiz, Alejandro Pasos et al. (2021). "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), pp. 401–449. DOI:

10.1007/s10618-020-00727-3. URL: https://doi.org/10.1007/s10618-020-00727-3 (cit. on pp. 2, 68, 71, 72).

Silva, Aldomar Pietro Santana et al. (2022). "Machine learning for noisy multivariate time series classification: a comparison and practical evaluation." In: *Anais do XIX Encontro Nacional de Inteligência Artificial e Computacional (ENIAC 2022)* (cit. on p. 82).

sktime (2022). *sktime - A unified framework for machine learning with time series*. URL: https://github.com/alan-turing-institute/sktime (last visited on 2023-03-04) (cit. on p. 82).

Szegedy, Christian et al. (2016). "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning." In: DOI: 10.48550/ARXIV.1602.07261. URL: https://arxiv.org/abs/1602.07261 (cit. on p. 73).

Unforgettable_fan and Jffner (2009). *HFC Network Diagram*. URL: https://en.wikipedia.org/wiki/Hybrid_fiber-coaxial#/media/File:HFC_Network_Diagram.svg (last visited on 2022-01-04) (cit. on p. 10).

Yu, Fisher and Vladlen Koltun (2016). "Multi-Scale Context Aggregation by Dilated Convolutions." In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. URL: http://arxiv.org/abs/1511.07122 (cit. on p. 80).