Karlsruher Institut für Technologie

Institut für Telematik
Pervasive Computing Systems / TECO
Leiter: Prof. Dr.-Ing. Michael Beigl

Erstellung und Evaluierung stochastischer Regressionsmodelle auf Basis heterogener Messnetzwerke.

Creating and Evaluating Stochastic Regression Models on the Basis of Heterogeneous Sensor Networks

Bachelorarbeit

von

Stanislav Arnaudov

an der Fakultät für Informatik

Verantwortlicher Betreuer:	Prof. Dr. Michael Beigl
Betreuendere Mitarbeiter:	Dr. Johannes Riesterer Dr. Sebastian Lerch
Bearbeitungszeit:	01.06.2018 – 30.09.2018

Karlsruher Institut für Technologie

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und weiterhin die Richtlinien des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, den 30.09.2018

Abstract

This thesis aims to better understand Bayesian machine learning models and their practical use on real world data. We examine two models that incorporate uncertainty in their predictions – Bayesian Neural Networks and Mixture Density Networks. The used data comes from air-pollution sensors. The quality of three of the sensors is known to be high but for the rest of them the quality of measurement is unknown. We aim to build a model that can predict the air pollution at some sensor at a given time. Consideration of the uncertainty in the predicted value is crucial as it allows the precise evaluation of the generated models. We compare the models through evaluation with proper scoring rules. As the quality of the majority of sensors is unknown, we try to find out which of the sensors are most relevant for the prediction through a feature importance technique. We leverage the capabilities of Tensorflow, Edward and GPFlow as machine learning libraries in order to build probabilistic regression models that can be further evaluated.

Contents

List of Figures

1. Introduction

1.1. Motivation

There are a lot of cases where a network of sensors is used to measure a certain quantity over time in different places. Some of the measurements can have better *quality* than others as not all sensors are manufactured equally. With quality we mean the reliability that the measured value indeed reflects the reality. The difference in the quality of the sensors makes the measured data *heterogeneous* and brings certain difficulties in the modeling process. For a machine learning application only some of the data may be relevant for building a reliable predictive model. One of our goals is to asses which part of the input is useful data and to what extend. From another point of view, we can also set another goal. We consider the case where only a few of the sensors are known to produce reliable measurements. In this sense we try to expand a network of *homogeneous* and high reliability sensors with sensors of unknown quality and thus making the network heterogeneous. In this case we increase the spatial resolution of the network and thus we bring the possibility to make more accurate predictions on the basis of data coming from the whole network. On the other hand, the heterogeneity in the quality of the different sources in the network introduces uncertainty in the measurements. It thus becomes necessary to build appropriate model that predicts the measured values from given sensor but also respects the inherent uncertainty in the data and also models it properly.

To concertize the problem – we use air pollution data from a network of sensors around Stuttgart. The *Landesanstalt für Umwelt, Messungen und Naturschutz Baden-Württemberg* (*LU-BW*, see [lub]) has provided data from three high quality air pollution sensors. The other part of the data is from *DIY* (*Do it yourself*) sensors that provide their measurements publicly under *luftdaten.info*. For more details see [luf]. Those are the unreliable parts of the network as described in the previous paragraph. We can make no statements about the quality of the data coming form the DIY sensors. We do not know how are they calibrated and the sensors are deployed from anonymous builders. Furthermore the sensors are cheap. All of this effectively means that we have unreliable features in our input data. We aim to find out which of the features – and with that which of the sensors – are "better" than the others. This is done through a *feature importance* technique described in Section 4.3. We also investigate whether or not we can predict the air pollution values measured by one of the LUBW-sensors based on the combined data coming from all other sensors. If reliable predictions are possible, this would suggest that one can theoretically use the cheap network of sensors as a substitute for the few high quality sensors. Figure 1 illustrates the general overview of the system we aim to develop.

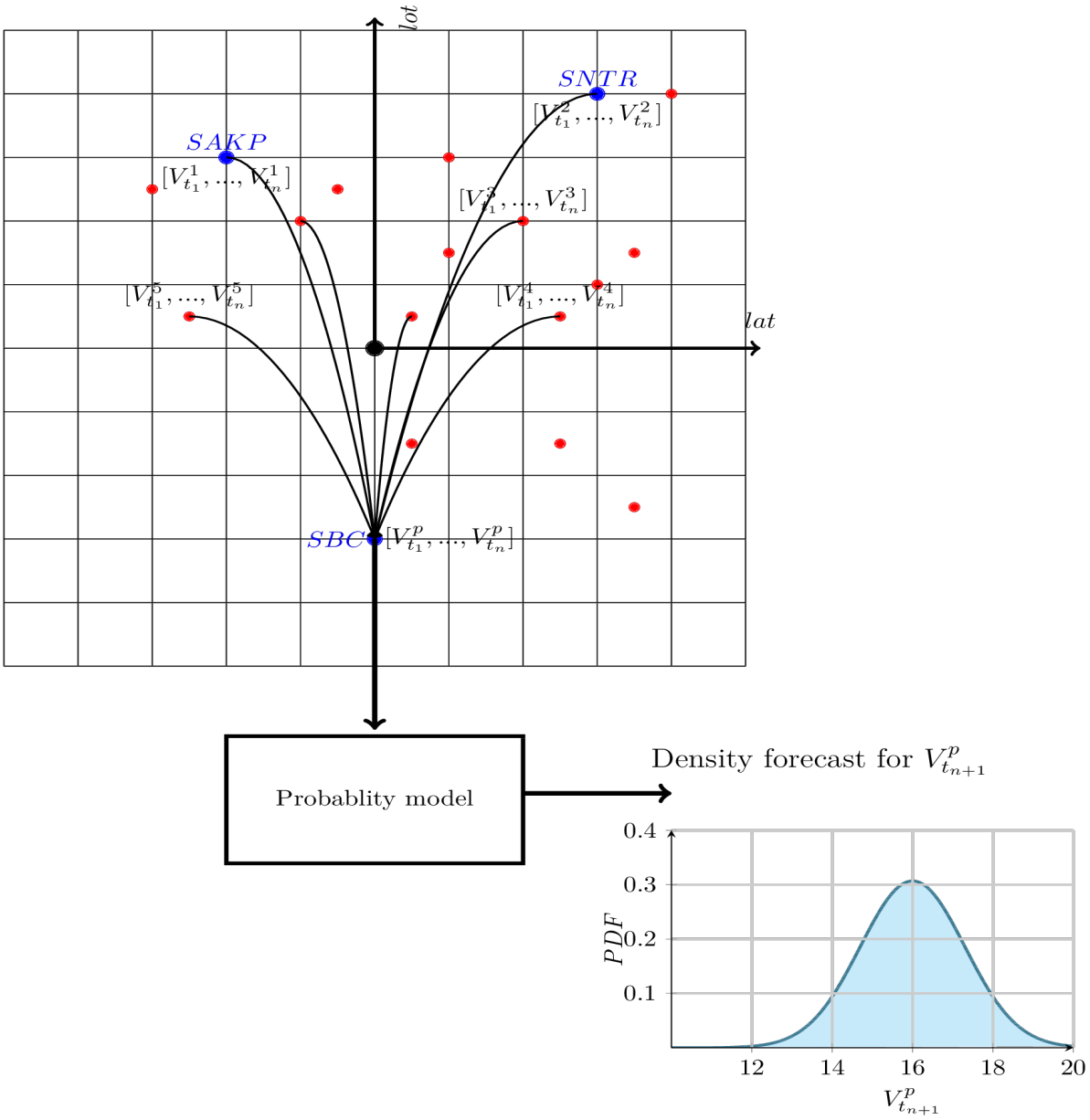


Figure 1: General view of the system we aim to build. The red dots represent unreliable DYI sensors. Data from all the sensors is used in order to make prediction for one (SBC) of the LUBW-Stations (shown in blue). The prediction made is for some future moment $\boldsymbol{t+1}$ and it is on the basis of the past – the values of all moments till $\boldsymbol{t+1}$ from all the stations. The prediction is generated in form of a probability distribution generated by the probabilistic regression model.

1.2. Probabilistic regression

Based on training data, regression models make a real valued prediction given an input data point. Each time a prediction is made, one would desire to have a certain measurement for the uncertainty of the prediction. If this is the case and the prediction is given in a form of some distribution, we say that the forecast is probabilistic. This quantification of the uncertainty allows for optimal decision making. Models that produce such probabilistic forecasts have gained recent popularity in areas such as meteorology, seismology and economics. In Section 2 we look at concrete cases where probabilistic regression is applied.

Probability forecasts are well established for classification models. Through the use of a soft-max layer, first introduced in [DL90], neural networks can assign each class they recognize a certain probability. This thesis is centered, however, around probabilistic regression models. *Probabilistic regression models* produce a probability distribution as their prediction rather than a point estimate. One can then draw samples from this distribution, make statements about the drawn samples and reason about the nature of the possible values that could be predicted. Illustration of this is given in Figure 2. Another way of thinking about probability forecasts is that they define a certain region where the predicted value can fall within. The region is defined in form of a probability distribution. The model tells us then what is the probability of each point within the region to be a predicted value given the input of the model. We say that the model describes the conditional probability of the output given the input (more details in Section 3). Figure 3 illustrates these considerations about probabilistic regression models. We also give Definition 1.1 so that we can reference it later in the thesis.

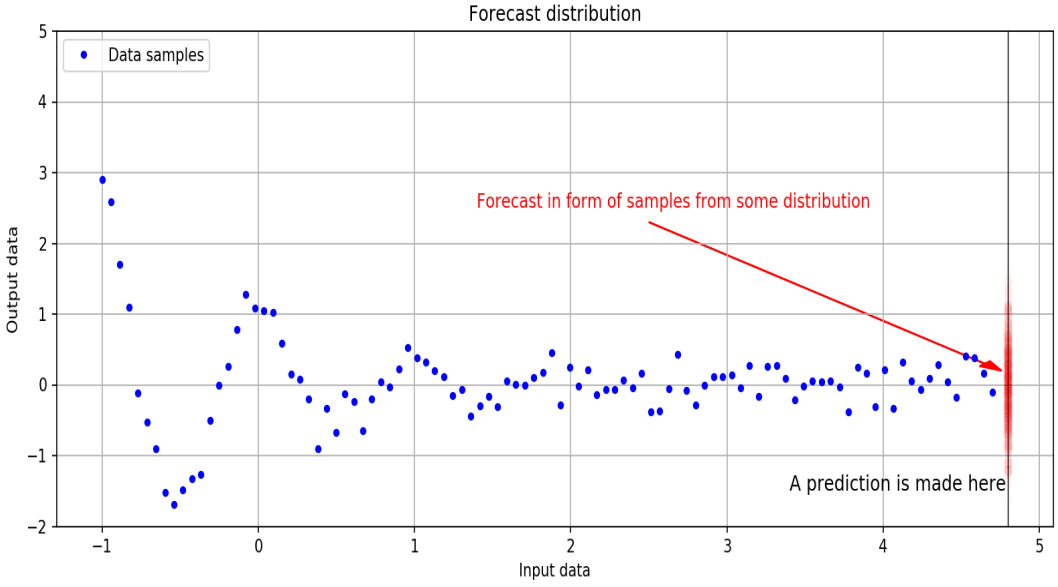


Figure 2: Illustration of what samples from forecast can look like. The blue dots are observations. On their basis a prediction is made. The red dots are drawn samples from the predicted distribution. We can see that those are concentrated near the most probable value of the observation at this place.

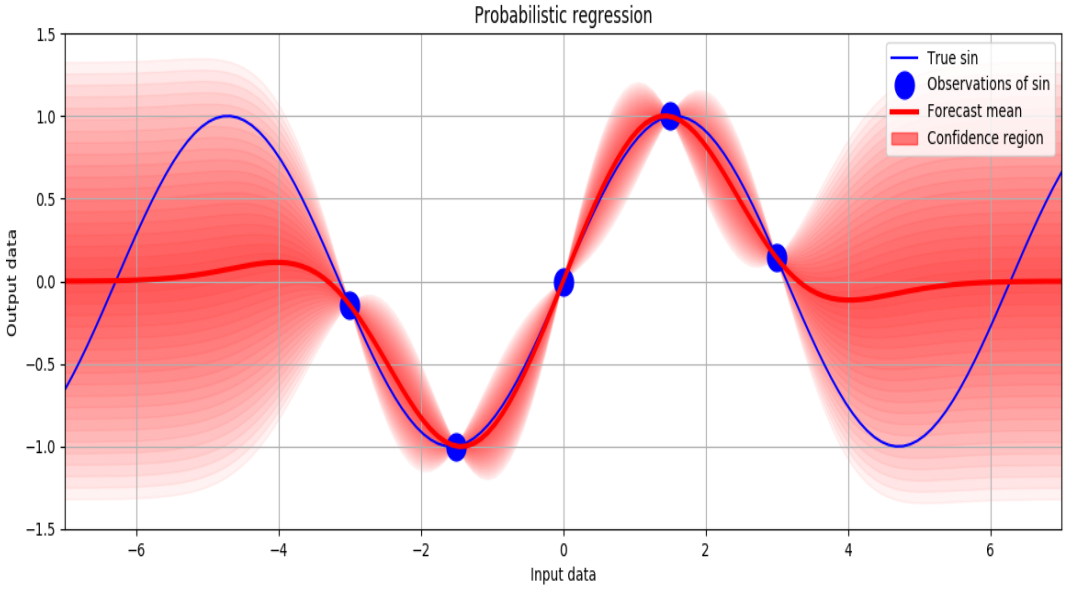


Figure 3: A probabilistic regression model (in red) that tries to model the **sin**-function (in blue). The big blue circles are the observations of the function that the models becomes as training set. On the basis of those observations we construct a probabilistic model that can evaluate the whole input space. The model knows about the given observations of the modeled function so it predicts their values with high certainty. The places where there is no observed data, however, the model cannot make precise prediction and the distribution is wider. We notice, however, that true value of the function lies in the distribution so in sense the model predicts the function somewhat accurately.

Definition 1.1. *Probabilistic forecast: Output of a probabilistic regression model for a single datapoint where the output represents some probability distribution. The distribution determines the probability of each point in space to be the regression value for the given considered input of the model. The distribution can be represented through certain amount of samples drawn from it, its density function or its cumulative distribution function. In the thesis we use probabilistic forecast, predicted distribution, distribution forecast, predictive distribution and forecast interchangeably.*

Predicting distributions requires some consideration by the evaluation of the predictive capabilities of the considered model. When it comes to point estimates, one can simply measure the distance between the predicted and the realized value. Comparing a distribution with a single value, however, is not so straight forward. Figure 4 illustrates the apparent difference between point estimate and distribution prediction and how can one can reason much more, if a distribution for the prediction is present. In our evaluation of the models we employ the use of *proper scoring rules*. A detailed look on the subject is given in [GR07]. The proper scoring rules can measure the error between a predicted probability distribution and a realized observation. We give more details on the proper scoring rules in Section 4.2.

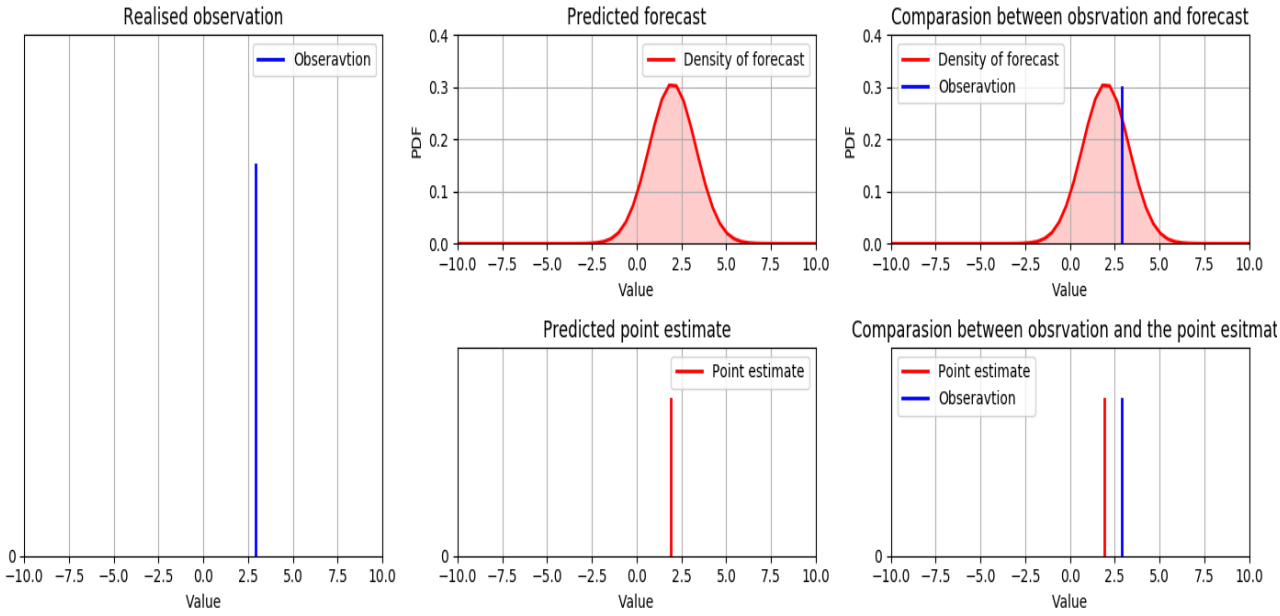


Figure 4: The left most graph shows actual observation that is to be predicted. In the middle there are two prediction made by two different models – a probabilistic forecast (on the top) and point estimate (on the bottom). With the right graphs one can clearly see how there is not a whole lot to compare between the point estimate and the observation other than the distance between the two. With the probability forecast, however, we can evaluate where and how exactly fits the observation in the predicted distribution.

The quality of probabilistic forecasts have two important characteristics:

- Calibration – a measure of the compatibility between the forecast and the observation. This property depends both on the observation and on the forecast. The forecast is assumed to be in a form of a probability distribution.
- Sharpness – a measure of the concentration of the forecast. This property is purely based on the predicted forecast. The forecast is again some predicted probability distribution.

We look at the two properties in details in Section 3.

The statistical framework for inferring probability distributions is *Bayesian Inference*. A good theoretical and practical overview of Bayesian inference can be found in [Tip04]. In Bayesian inference *Bayes theorem* (see [Bay63]) plays a central role in determining the probability for a hypothesis given some evidence. Bayesian inference takes into consideration our prior beliefs about the nature of some data and then combines these beliefs with the actual observed data (denoted as *evidence*) and generates *posterior* probability about the values of the output. We discuss Bayesian reasoning in Section 3.1.

1.3. Data

The air pollution data that we use was recorded by a network of sensors over one year period for the year of 2017. As previously stated, the data comes from two different types from sources.

- LU-BW (Landesanstalt für Umwelt, Messungen und Naturschutz Baden-Württemberg) – Organisation whose activities include air measurements and high quality sensors deployment. LUBW has gathered data from three high quality sensors in Stuttgart for the year 2017. The data is to be used for research proposes only and is not publicly available.
- luftdaten.info – a public network of sensors where everyone can provide data from sensor from certain type. There we can find air pollution data from *sds011*-sensors (see [sds] for specifications). Those are cheap DIY sensors and give no guarantees for the quality of the measured data. Measurements of wide networks of such sensors are publicly available under [luf].

All of the sensors measure two types of air pollution data. The different naming of these values relates to the size of the measured particles. The two data types are:

- *PM2.5* – Particular matter up to 2.5 micrometers.
- *PM10* – Particular matter up to 10 micrometers.

We found that those values are highly correlated – correlation of about 0.95 on average (see Section 5.1 and Appendix A for details). For this reason the built models use either PM2.5 or PM10. That is, with PM2.5 data of all other sensors, we predict the PM2.5 values of a particular sensor or respectively with PM10 data we predict PM10 values.

The format of the data coming straight from the networks was not convenient for direct training of the models. The sensors from *luftdaten* provide measurements for each minute over the whole year while the ones of LUBW integrate their measurements over thirty minutes. Because of this inconsistency we had to "synchronize" the both data sources in order to produce the final data sets. We've also found numerous problems with the data from *luftdaten*. Some of them are:

- Not all sensors are located around Stuttgart.
- Some of the sensors have a lot of missing days without measurements.
- Some of the sensors have missing measurement in random minutes of the day.

The raw data was extensively preprocessed with different techniques. We explain the preprocessing of the data in Section 6.2.

1.4. Models

The models of interest that we are going to look at are *Bayesian Neural Networks* (*BNNs*, see Section 3.2) and *Mixture Density networks* (*MDNs*, see Section 3.3). Here we give brief overview of the both models and we discuss them in detail in the respective sections.

BNNs are relatively new type of model proposed in the 1990s and studied in depth in [Mac92] and [Nea96]. [Gal16] provides a general survey and a study on the uncertainty quantification in deep learning. More recently BNNs have seen a surge in popularity as they combine Neural Networks with Probabilistic Models. The inferred distributions over their parameters allows them to offer a probability distributions for their estimates. This makes BNNs attractive to the machine learning community as those properties give them robustness to over-fitting and the ability to learn from small datasets. These properties of the BNNs are well documented in [Nea96]. BNNs place prior distribution on their weights. The posterior distributions are estimated either by variational inference ([BCKW15], [PBJ12]) or by sampling techniques ([VSL00]). In our work we use variational inference with *Kullback-Leibler divergence* and give details about the method in Section 3.2.3.

With MDNs we hope to construct a model similar to an infinite mixture of Gaussian experts, proposed in [RG02]. The mixture of Gaussian experts is a combination of several Gaussian process regression models and a gating network. Given an example data point the gating network assigns certain probabilities for each of the Gaussian process models. On their turn they generate several probability distributions which are weighted with the probabilities from the gating networks. The end result again is a probability distributions given by the sum of the individual ones. MDNs model this by creating a single network that not only outputs weights for the individual mixture components but also their parameters. MDNs are very similar to standard Neural Networks with the only difference being that final layer is mapped to a mixture of distributions. In our case the mixture model is a weighted sum of Gaussian distributions. MDNs produce especially good results when the input data is such that a single input can correspond to multiple output values. In those cases, the predicted distribution models the corresponding output values by giving some probability on each of them. We investigate what results one can get when this approach is used for simpler data. As MDNs are almost standard neural networks, they are trained with the standard *backpropagation* method, described for example in [BNVP01].

In order to have a baseline with which we can compare the built models we also construct a simple empirical model. The empirical modeling is a technique for making predictions about the values of some process based purely on empirical observations of past values of the process. In this cases there is no consideration of the input of the data but rather a simple looking at the past values of the output value. The past values of the predicted measurement are treated as samples from a random variable. We describe how we infer a probability distribution for a prediction in Section 6.3.3. Similar empirical models are commonly used for simple predictions of meteorological data. Examples of this approach can be found in [KVS11] and [EvOHS15].

2. Related work

Heterogeneous sensor networks are present in a lot of cases. [WC07] examines the deployment and the control of the topology of heterogeneous wireless sensor networks. [KCS] surveys a variety of clustering algorithms aimed at increasing the network's scalability by recognizing the sensors with similar energy requirements. [US14] investigates the kinds of security threats that arise with the use of heterogeneous sensor networks. In the context of machine learning, [LZLQ17] tries to use deep learning techniques and adapt them to the settings of heterogeneous sensor data.

[GK14] has brought a lot of attention to probabilistic forecasting. When it comes to history, [Sti75] describes the transition from point estimates to distribution prediction. Nowadays the technique is widely used in variety of cases as it is further developed ([EG12]). Common areas where probabilistic forecasts are employed are whether and climate prediction. The approach is outlined in [Col07]. There are also other diverse applications. [Krz01] looks how probabilistic models can be used for flood risk assessment, [Tim] describes probabilistic models within the context of economic and financial risk management, [Pin13] investigates the potential to make predictions about the availability of renewable energy resources. [MHW12] shows how distribution forecasting can be even used to make statements about the outcome of elections.

The other focus of our work are BNNs. As models that tie Bayesian reasoning and neural networks together, they have also gained popularity. A good outline of neural networks and deep leaning can be found in [Sch15]. [LV01] first gives a brief overview of the Bayesian approach to neural networks and then surveys different use cases assessing the performance. Fields like astrophysics [BXP+16] and particle physics [PCB15] also profit from the predictive capabilities of BNNs. As medicine is one of the fields that gets much of the attention of the machine learning research, [KP17] applies BNNs in order to quantify the uncertainties and apply them in ischemic stroke lesion segmentation. BNNs are also continuing to be developed. [FBV17] explores the Bayesian approach in Recurrent Neutral Networks developing *Bayesian Recurrent Neural Networks*.

MDNs have also found popularity as an another alternative to combine neural networks with distribution forecast models. [SDD98] uses MDNs as a general framework for identifying stochastic processes and apply them in modeling stock exchange index data. [ZS14] brings MDNs to use in order to improve deep neural networks for acoustic modeling. MDNs have their own variant of recurrent models as described in [MDD03]. We like to note that MDNs have similar structure to other models such as *Adaptive mixture of local experts* – proposed by [JJNH91]) – and *Infinite Mixtures of Gaussian Process Experts* – proposed by [RG02]. MDNs, however, model the data differently. We explain the approach in Section 3.3.

The use of proper scoring rules is standard in probability forecast setting. [JKL17] discusses the **scoringRules** package for R[1] and its usage details by presenting case studies where proper scoring rules find application. [Cle] uses probability integral transform in order to evaluate the UK Monetary Policy Committee's inflation density forecasts. We describe probability integral transform in Section 4.1.2.

[1] Programming language

3. Probabilistic Modeling

3.1. Bayesian regression modeling

In a probabilistic modeling setting we aim to estimate the parameters of a given model by first defining some *prior* on them. The prior distribution describes how likely values of the parameter are without considering any observed data. Then a *conditional likelihood* for the output based on the parameters and the input data is defined. This probability distribution describes the probability of observing certain data given the parameters of the model. The assumption here is that that the model has generated the data. The two distributions are then combined in order to produce the final *posterior* probability distribution on the parameters of the model. A good overview of Bayesian models can be found in [Mit97]. We can summarize:

- **Prior** – uncertainty in form of probability distribution *before* observing the data.
- **Posterior** – uncertainty in form of probability distribution *after* observing the data.

Now we formally write down everything following the approach of [Gal16]. Let

$$\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}, \forall \boldsymbol{i} : \boldsymbol{x}_i \in \mathbb{R}^M \tag{3.1}$$

be the set of input data. The corresponding output is given by the set $\boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N\}$. The dataset can also be seen as tuples of feature vector $\boldsymbol{x_i}$ and an output value $\boldsymbol{y_i}$. We can then write the dataset as $\{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{\boldsymbol{i=1}}^{\boldsymbol{N}}$. As we deal with regression models we assume that $\forall \boldsymbol{i} : \boldsymbol{y}_\mathrm{i} \in \mathbb{R}$ and there is some function $\boldsymbol{f_{true}} : \boldsymbol{X} \rightarrow \mathbb{R}$ with $\boldsymbol{f_{true}}(\boldsymbol{x_i}) = \boldsymbol{y_i}, \forall \boldsymbol{i} \in \{\boldsymbol{1}, \ldots, \boldsymbol{n}\}$. We want to find a model with appropriate parameters that is likely to have generated the data. In fact, through the choice of parameters, we want the observed data to be as probable as possible. We model the mapping function between $\boldsymbol{X}$ and $\boldsymbol{Y}$ through a separate function $\boldsymbol{f_m}(\boldsymbol{x}; \boldsymbol{\omega})$ parameterized by $\boldsymbol{\omega}$. We want to find a "good" set of parameters $\boldsymbol{\omega}$ such that the function $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\omega})$ is *likely* to have generated the output data. We define our prior believes about the parameters through the distribution $\boldsymbol{p}(\boldsymbol{\omega})$. The generalized likelihood distribution is then $\boldsymbol{p}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\omega})$. That is, how likely is output $\boldsymbol{y}$ given model parameters $\boldsymbol{\omega}$ and concrete input $\boldsymbol{x}$. Our assumption here is that the defined model generates the data. The output of a regression model is typically set to be a Gaussian likelihood: $\boldsymbol{p}(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{\omega}) = \boldsymbol{\mathcal{N}}(\boldsymbol{y}; \boldsymbol{f_m}(\boldsymbol{x}; \boldsymbol{\omega}), \boldsymbol{\tau^{-1}I})$ where $\boldsymbol{\tau}$ is the model precision. This can be interpreted as adding small amount of noise to the output as modeled in [Gal16].

We can now write out the posterior by using the Bayes' theorem:

$$\boldsymbol{p}(\boldsymbol{\omega}|\boldsymbol{X}, \boldsymbol{Y}) = \frac{\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\omega})\boldsymbol{p}(\boldsymbol{\omega})}{\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X})} \tag{3.2}$$

This distribution describes the most likely model parameters given the observed data. If we draw parameters from this distribution, the observed data will be most likely. For prediction of new data point $\boldsymbol{x^*}$ we use

$$\boldsymbol{p}(\boldsymbol{y}^*|\boldsymbol{x}^*, \boldsymbol{X}, \boldsymbol{Y}) = \int \boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\omega})\boldsymbol{p}(\boldsymbol{\omega})\boldsymbol{d\omega} \tag{3.3}$$

This integration is called *inference* ([Gal16]). The quantity

$$\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}) = \int (\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\omega})\boldsymbol{p}(\boldsymbol{\omega})\boldsymbol{d\omega} \tag{3.4}$$

is called marginal likelihood or sometimes "model evidence". The first name comes from the fact that we are marginalizing the likelihood over $\boldsymbol{\omega}$. This factor is independent of the parameters so no matter how we choose them, it does not change. Marginalization plays central part in Bayesian modeling and in the prefect case we would want to marginalize over all possible parameter values $\boldsymbol{\omega}$ with respective weight $\boldsymbol{p(\omega)}$. In complex models this is not possible and we resort to approximations.

"Learning" in this Bayesian setting boils down to finding the posterior distribution $\boldsymbol{p(\omega|X, Y)}$ over the parameters. In the general case this cannot be done analytically (a closed formula cannot be given) and thus one must attempt to approximate the distribution.

Inference provides us with the full probability distribution over the parameters and for that reason we are not dealing with point estimates of the parameters. By extension we can also provide some distribution when we evaluate the model on unseen data. In order to "generate" a sample from the resulting distribution for some unseen input data point $\boldsymbol{x^*}$ we first need to sample $\boldsymbol{\omega_s}$ from the parameter distribution and then evaluate the model $\boldsymbol{f_e}$ at point $\boldsymbol{x^*}$ parameterized by the drawn sample $\boldsymbol{f_e(x^*; \omega_s)}$.

The models that we consider next are nothing more than concrete choices for $\boldsymbol{f_e}$ and its parametrization.

3.2. Bayesian Neural Networks

3.2.1. Neural networks

First we give brief overview of conventional neural networks.

Neural networks (or artificial neural networks) are computing systems that model some arbitrary function through their structure and parameters. Neural networks consist of artificial neurons that have a certain number of inputs. They first compute a weighted linear sum of their inputs and then apply a non-linearity in form of **tanh**, logistic or other type of *sigmoid* function. Several stacked neurons are considered a layer of a network. Several consecutive layers where the output of one layer is the input of the next are considered a *neural network*.

Formally the neural network is nothing more than a parameterized function with specific structure that allows it to estimate almost every possible function when given the right parameters. Let $\boldsymbol{W_i} \in \mathbb{R}^{\boldsymbol{A_i} \times \boldsymbol{B_i}}$ and $\boldsymbol{b_i} \in \mathbb{R}^{\boldsymbol{B_i}}$ be weight matrices and biases respectively and let $\boldsymbol{o_i} \in \mathbb{R}^{\boldsymbol{B_i}}$ denote the output of the $\boldsymbol{i}^{\text{th}}$ layer of the network. For an input $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{A_1}}$ we then have:

$$\boldsymbol{o_1} = \mathbf{tanh}((\boldsymbol{W_1 x}) + \boldsymbol{b_1}) \tag{3.5}$$

$$\boldsymbol{o_2} = \mathbf{tanh}((\boldsymbol{W_2 o_1}) + \boldsymbol{b_2}) \tag{3.6}$$

$$\vdots \tag{3.7}$$

In general, for a network with $\boldsymbol{m}$ layers where the $\boldsymbol{m}^{\text{th}}$ layer is the output layer, the outputs of the individual layers can be given as:

$$\boldsymbol{o_{i+1}} = \mathbf{tanh}((\boldsymbol{W_{i+1} o_i}) + \boldsymbol{b_{i+1}}) \tag{3.8}$$

$$\boldsymbol{o_m} = (\boldsymbol{W_m o_{m-1}}) + \boldsymbol{b_m} \tag{3.9}$$

where $\boldsymbol{o_m}$ is the final output of the network. The forward evaluation of the network is just a cascade of applications of liner functions in form of a matrix multiplication and addition of bias term followed up by an application of some non-linear function. For sake of simplicity let us collapse the notation of the output of the network to a single function of the input $\boldsymbol{x}$ and the collection of parameters (weight matrices and bias-terms) to $\boldsymbol{\Omega}$. The evaluation of the network can be given with

$$\boldsymbol{o_m} = \boldsymbol{f_{net}}(\boldsymbol{x}; \boldsymbol{\Omega}) \tag{3.10}$$

A visual representation of a neural network is given in Figure 5.

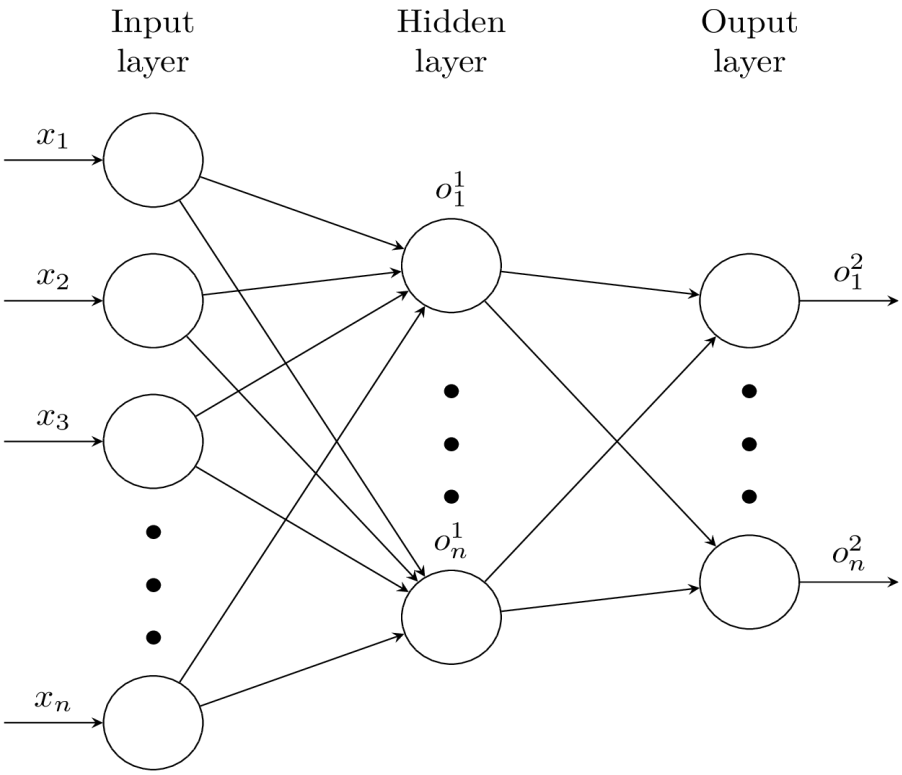


Figure 5: A schematic of a neural network with one hidden layer. The circles represent individual neurons. The data flows from left to right as the neurons sum up their weighted inputs, apply non-linearity and pass their outputs to the neurons in the next layer.

In order to be useful we have to "teach" the neural network our desired mapping. As in previous section we have a set of input data $\boldsymbol{X} = \{\boldsymbol{x_1}, \ldots, \boldsymbol{x_N}\}, \forall \boldsymbol{i} : \boldsymbol{x_i} \in \mathbb{R}^M$ and the corresponding output for each $\boldsymbol{x_i}$ is contained in the set for the output data $\boldsymbol{Y} = \{\boldsymbol{y_1}, \ldots, \boldsymbol{y_N}\}, \forall \boldsymbol{i} : \boldsymbol{y_i} \in \mathbb{R}$. Ideally we would want the following to hold for the neural network

$$\boldsymbol{f_{net}}(\boldsymbol{x_i}; \boldsymbol{\Omega}) = \boldsymbol{y_i} \tag{3.11}$$

In other words, the network has *learned* the exact mapping between $\boldsymbol{X}$ and $\boldsymbol{Y}$ and can predict the output value based on the input one. This is done by choosing appropriate parameter set $\boldsymbol{\Omega}$. In general, finding parameters realising the exact mapping will not be possible but we want to at least approximate this case. This leads us naturally to a definition of measurement of just how wrong the network is. We call this quantity *error function* (also called *loss function*). The

error function is a function of the parameter space and it gets closer to zero as the parameters define a network that better models the data. There are lot of possible choices for loss function but the most common is the *squared error difference*:

$$\boldsymbol{E}(\boldsymbol{\omega}) = \sum_{\boldsymbol{i=0}}^{\boldsymbol{N}} (\boldsymbol{f_{net}}(\boldsymbol{x_i}; \boldsymbol{\omega}) - \boldsymbol{y_i})^{\boldsymbol{2}} \tag{3.12}$$

where $\boldsymbol{\omega}$ is some choice for the parameters for the networks, namely $\boldsymbol{m}$ weight matrices and $\boldsymbol{m}$ bias terms for network with $\boldsymbol{m}$ layers. The squared error difference is a simple sum of all differences between predicted by the network value and the actual value. If the defined function is zero, the network essentially predicts all values correctly. Minimizing this *error function* is what we call *learning* the data or *training* the network. The actual method trough which learning occurs is *backpropagation*. This involves calculating the derivatives of the loss function with the respect to each parameters of the network $\frac{\boldsymbol{\partial E}}{\boldsymbol{\partial W_i}}$. The corresponding parameters are then updated through some gradient descent technique. An overview of different gradient descent techniques can be found in [Rud16]. Some state of the art optimization methods are *LM-BFGS* (Limited Memory Broyden–Fletcher–Goldfarb–Shanno algorithm, see [BLNZ95]), Adam optimization algorithm (see [KB14]) and *SGD* (Stochastic gradient descent, see [RM51]). To note is that the last motioned method – the SGD – does not calculate the gradient of the loss function using all of the training examples rather it uses only a small subset of the training set.

We would like to note that neural networks are considered universal function approximators. This means that given enough complexity of their structure, they can model arbitrary non-linear functions. A prove for this statement is provided in [HSW89].

3.2.2. Neural networks in Bayesian settings

In this section we focus specifically on Bayesian neural networks.

BNNs place a prior distribution over the parameters of a neural network. As described in previous section, once we have distributions on the parameters of a model, we have effectively defined a distribution over functions. Each time when we draw a sample from the parameter space, we have defined a "new" function that models the given data. In this sense, we have sampled a new function from some distribution. Usually, for given weight matrix $\boldsymbol{W_i}$ and bias term $\boldsymbol{b_i}$ in layer $\boldsymbol{i}$ of the network, BNNs place standard Gaussian prior distributions over the matrices, $\boldsymbol{p(W_i) = \mathcal{N}(0, I)}$ and a point estimate for the $\boldsymbol{b}_\text{i}$ is assumed. In this work, however, the standard prior is also placed on the bias vectors so we define the distributions $\boldsymbol{p(b_i) = \mathcal{N}(0, I)}$.

The basic structure of the BNNs is the same as the one of regular neural networks. For this reason we will use the notations from the previous section. The output of the network for a given input $\boldsymbol{x}$ and parameters $\boldsymbol{\omega}$ is

$$\boldsymbol{f_{net}}(\boldsymbol{x}; \boldsymbol{\omega}) \tag{3.13}$$

We have now defined what the basic structure of our considered model is (a neural network) and we know what the priors over the parameters of the model are. As in the previous section we now define a likelihood for each data point. We consider a set of data $\boldsymbol{\{(x_i, y_i), i \in \{1, \ldots, n\}\}}$

where each data point has some features $\boldsymbol{x_i} \in \mathbb{R}^M$ of dimension $\boldsymbol{M}$ and real valued output $\boldsymbol{y_i} \in \boldsymbol{\mathcal{R}}$. We define the likelihood for the BNNs as:

$$\boldsymbol{p(y_i|\Omega, x_i, \sigma^2) = \mathcal{N}(y_i|f_{net}(x_i; \omega), \sigma^2)} \tag{3.14}$$

where $\boldsymbol{\sigma}^2$ is again some known small variance that can be considered as "noise" of the output ([Gal16]).

With this overview we can make several general statements about the BNNn. A BNN is a probability model that utilizes a neural network as universal function approximator. The distributions on the network's parameter allow it to generate a distributional output rather than a point estimate. This the exact same Bayesian setting as described in Section 3.1.

The final piece of having some useful model is to have notion what "learning" means in the case of BNNs. As this is a Bayesian probability model we have to find the posterior distribution over the parameters based on the example data:

$$\boldsymbol{p(\omega|Y, X) = \frac{p(Y|X, \omega)p(\omega)}{p(Y|X)}} \tag{3.15}$$

This distribution describes the most likely parameters given the example data. Direct computation of the posterior is not possible for networks of any practical sizes. The problems come from the term in the denominator as it involves integrating over the whole parameter space:

$$\boldsymbol{p(Y|X) = \int_\Omega p(Y|\omega, X)p(\omega|X)d\omega} \tag{3.16}$$

In the next section we describe a technique that aims to estimate the posterior distribution without actually having to approximate the above integral. Once we have the posterior distribution (or a sufficient estimate of it) we could use it to produce the predicted distribution for unseen data samples. This is done through *inference* as described in Section 3.1.

3.2.3. Variational inference and Kullback-Leibler divergence

As previously stated, it is impractical to compute the full posterior. In fact, the true posterior is intractable and cannot be evaluated analytically. We therefore rely on an estimation technique that transforms the Bayesian inference into an optimization problem – *variational inference* (first proposed in [JGJS99], see an overview in [BKM16], a detailed look in [FR12]). We define a *variational* distribution $\boldsymbol{q_\theta(\omega)}$, parameterized by $\boldsymbol{\theta}$. We assume that this distribution is easy to evaluate. Furthermore we would really like this distribution to be able to approximate other distributions based on the choice of parameters $\boldsymbol{\theta}$. If those assumptions are met, we then try to choose such parameters for $\boldsymbol{q_\theta(\omega)}$ that it maximally resembles the posterior $\boldsymbol{p(\omega|Y, X)}$. For that we also need some measure between two distributions that gives us how "similar" they are. This is slightly analogous to the choice of loss function that is to be minimized. In this case we use the *Kullback–Leibler* (*KL*) divergence proposed by [KL51]. KL is defined as

$$\boldsymbol{KL(q_\theta(\omega), p(\omega|X, Y)) = \int q_\theta(\omega) \log \frac{q_\theta(\omega)}{p(\omega|X, Y)} d\omega} \tag{3.17}$$

KL is a measure of how one probability distributions differs from another. A Kullback–Leibler divergence of $\mathbf{0}$ implies the two distributions are identical. KL divergence does not have a

maximum value as it is unbounded. We aim to minimize this function with respect to $\boldsymbol{\theta}$. Our problem then becomes:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \boldsymbol{KL}(q_{\boldsymbol{\theta}}(\boldsymbol{\omega}), p(\boldsymbol{\omega}|\boldsymbol{X}, \boldsymbol{Y})) \tag{3.18}$$

$$= \arg\min_{\boldsymbol{\theta}} \int q_{\boldsymbol{\theta}}(\boldsymbol{\omega}) \log \frac{q_{\boldsymbol{\theta}}(\boldsymbol{\omega})}{p(\boldsymbol{\omega}|\boldsymbol{X}, \boldsymbol{Y})} d\boldsymbol{\omega} \tag{3.19}$$

$$= \arg\min_{\boldsymbol{\theta}} \boldsymbol{KL}(q_{\boldsymbol{\theta}}(\boldsymbol{\omega}), p(\boldsymbol{\omega})) - \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{\omega})}[\log\, p(\boldsymbol{Y}|\boldsymbol{w})] \tag{3.20}$$

The last cost function is known as the *variational free energy* ([NH98],[BCKW15]) or as the *expected lower bound* ([FR12], [SJJ96]). Thus the function that we want to optimize is

$$\boldsymbol{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{Y}) = \boldsymbol{KL}(q_{\boldsymbol{\theta}}(\boldsymbol{\omega})||p(\boldsymbol{\theta})) + \mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{\omega})}[\log(p(\boldsymbol{Y}|\boldsymbol{\omega}))] \tag{3.21}$$

This loss function is a sum of a data-dependent part and a prior-dependent part. The function embodies a trade-off between satisfying the complexity of the data $\boldsymbol{X}$ and $\boldsymbol{Y}$ and satisfying the simplicity of the prior $\boldsymbol{p}(\boldsymbol{\omega})$(see [BCKW15] for the derivation of the function).

Gradient calculation of this function is not a trivial task. As we use standard Gaussian priors on the parameter $\boldsymbol{\omega}$, the first term in the function (the KL divergence) is traceable and analytically solvable ([PBJ12]). The other term, however, is not that well behaved. Several methods have been proposed to optimize the function with backpropagation techniques. The concrete implementation that we ([TKD+16]) use, utilizes either *Stochastic Gradient Vitiation Bayes* (*SGVB*) as described in [KW13] or the method described in [PBJ12].

We now give a brief overview of SGVB. For the approximation of the posterior $\boldsymbol{q_\theta(\omega)}$ we reparameterize the random variable $\boldsymbol{z} \sim \boldsymbol{q_\theta(\omega)}$ by the use of some differential transformation $\boldsymbol{g_\omega(\epsilon, x)}$ and a noise variable $\boldsymbol{\epsilon}$.

$$\boldsymbol{z} = \boldsymbol{g_\omega(\epsilon, x)}, \quad \boldsymbol{\epsilon} \sim \boldsymbol{p(\epsilon)} \tag{3.22}$$

With those definitions we can now estimate the expected value of some function $\boldsymbol{f(x)}$ with respect to $\boldsymbol{q_\theta(\omega)}$:

$$\mathbb{E}_{q_{\boldsymbol{\theta}}(\boldsymbol{\omega})}[\boldsymbol{f(x)}] = [\boldsymbol{f(g_\omega(\epsilon, x))}] \simeq \frac{\boldsymbol{1}}{\boldsymbol{L}} \sum_{\boldsymbol{l=1}}^{\boldsymbol{L}} \boldsymbol{f(g_\omega(\epsilon^{(l)}, x))} \tag{3.23}$$

We can apply this technique to our cost function and we get a tractable and a differentiable estimate of it

$$\boldsymbol{\mathcal{L}(\theta, Y)} \simeq \boldsymbol{\mathcal{L}(\theta, Y)^E} = \boldsymbol{KL(q_\theta(\omega)||p_\theta(z))} + \frac{\boldsymbol{1}}{\boldsymbol{L}} \sum_{\boldsymbol{l=1}}^{\boldsymbol{L}} \boldsymbol{(\log(p_\theta(x|\omega)))} \tag{3.24}$$

Now that we have a function for which the gradients can be given analytically, we can optimize it with respect to the parameter $\boldsymbol{\theta}$. This is done typically through Stochastic Gradient Descent (see Section 3.2.1). In this case we do not calculate the loss on the whole dataset but rather on s sub-sample of it. Let say we take $\boldsymbol{M}$ datapoints from dataset with $\boldsymbol{N}$ datapoints. We then essentially are making the estimate:

$$\boldsymbol{\mathcal{L}(\theta, Y)} \simeq \boldsymbol{\mathcal{L}(\theta, Y^M)^E_M} = \frac{\boldsymbol{N}}{\boldsymbol{M}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{M}} \boldsymbol{\mathcal{L}(\theta, y_i)^E} \tag{3.25}$$

It is known that SGD leads to faster convergence with less computations ([RM51]). It is also proven and observed ([SW96]) that SGD converges to the result of regular gradient descent. For those reasons, the usage of SGD is almost always preferable.

There is also a somewhat simpler approach of estimating the gradient of the loss function as described in [PBJ12]. Under some regularity conditions, a gradient of the form $\nabla_\theta \mathbb{E}_q \boldsymbol{f}$ where $\boldsymbol{f}$ is some untraceable function, $\boldsymbol{q}$ is some distribution and we take the gradient with respect to $\boldsymbol{\theta}$, can be rewritten as:

$$\nabla_\theta \mathbb{E}_q \boldsymbol{f}(\boldsymbol{\theta}) = \int_\theta \boldsymbol{f}(\boldsymbol{\theta}) \boldsymbol{q}(\boldsymbol{\theta}) \nabla_\theta \ln(\boldsymbol{q}(\boldsymbol{\theta})) d\boldsymbol{\theta} \tag{3.26}$$

Details on how this equation is derived can be found in [PBJ12]. The integral can then be estimated through Monte Carlo integration ([HH64]). This approach can be used to replace the $\mathbb{E}_{q_\theta(\omega)}[\log(\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{\omega}))]$ in the cost function. We can then again apply stochastic gradient descent in order to spare some computations and accelerate convergence. For more detail on the method see [PBJ12].

Other techniques for posterior estimation rely on sampling from the posterior. This can be done for example by *Gibbs sampling* ([GG84]), *Metropolis-Hastings* ([MRR+53], [Has70]) or *Hybrid Monte Carlo* ([DKPR87]). These are all so called *Markov Chain Monte Carlo* (*MCMC*) methods and are popular techniques for sampling from some posterior distribution. We, however, do not consider them here.

3.3. Mixture Density Networks

In this section we describe the theoretical background of our second considered model – mixture density networks.

We first explain the MDNs' approach in modeling the data. MDNs try to describe each data point $(\boldsymbol{x}, \boldsymbol{y}) \in \boldsymbol{X} \times \boldsymbol{Y}$ with a feature vector $\boldsymbol{x} \in \mathbb{R}^{\boldsymbol{R}}$ and a value $\boldsymbol{y} \in \mathbb{R}$ directly through some mixture model.

$$\boldsymbol{p}(\boldsymbol{y}|\boldsymbol{x}) = \sum_{\boldsymbol{m}=\boldsymbol{1}}^{\boldsymbol{M}} \boldsymbol{\alpha}_m(\boldsymbol{x}) \boldsymbol{\phi}_m(\boldsymbol{y}|\boldsymbol{x}) \tag{3.27}$$

where $\boldsymbol{M}$ is the number of mixture components and $\boldsymbol{\phi}_m(\boldsymbol{y}|\boldsymbol{x})$ is the conditional density of the $\boldsymbol{m}^{\boldsymbol{th}}$ kernel. The parameters (or weights) $\boldsymbol{\alpha}_m(\boldsymbol{x})$ are called *mixing coefficients* ([MB94]). To note is that everything in the model depends on the concrete input data $\boldsymbol{x}$. In this sense the parameters for the mixture change for every data point. We choose mixture of Gaussians as with appropriate weights it can model any arbitrary distribution. This means that theoretically MDNs with Gaussian mixture can be universal distribution estimators Similar to regular neural networks which are universal real valued function estimators. Thus all of the density functions become:

$$\boldsymbol{\phi}_m(\boldsymbol{y}|\boldsymbol{x}) = \frac{\boldsymbol{1}}{\boldsymbol{2\pi^{c/2}\sigma}_m(\boldsymbol{x})^c} \exp\{-\frac{\|\boldsymbol{y} - \boldsymbol{\mu}_m(\boldsymbol{x})\|^2}{\boldsymbol{2\sigma}_m(\boldsymbol{x})^2}\} \tag{3.28}$$

Again, the parameters of each mixture component, $\boldsymbol{\mu}_m(\boldsymbol{x})$ and $\boldsymbol{\sigma}_m^2(\boldsymbol{x})$, depend on the input data. The last equation assumes that the components of the output vector are statistically independent within each component of the distribution, and can be described by a common variance

([MB94]). As we deal with regression in this thesis, the output vector is one dimensional and thus the assumption holds true.

With this setup, the MDNs provide conditional density function $\boldsymbol{p(y|x)}$ that can approximate any arbitrary distributions. We treat the parameters of the mixture – the mixing coefficients $\boldsymbol{\alpha}_{\mathrm{m}}\boldsymbol{(x)}$, the means $\boldsymbol{\mu_m(x)}$ and the variances $\boldsymbol{\sigma_m(x)}$ – as continuous functions of the input $\boldsymbol{x}$. It is clear that with the proper parameter functions we can model each data point. The "finding" of this function is achieved through a conventional neural network which takes $\boldsymbol{x}$ as an input. This combination of mixture model and a neural network is what *Mixture Density Networks* (MDNs) are. It is therefore said that MDNs are nothing more than neural network plus a mixture of Gaussians. An overview is given in Figure 6.

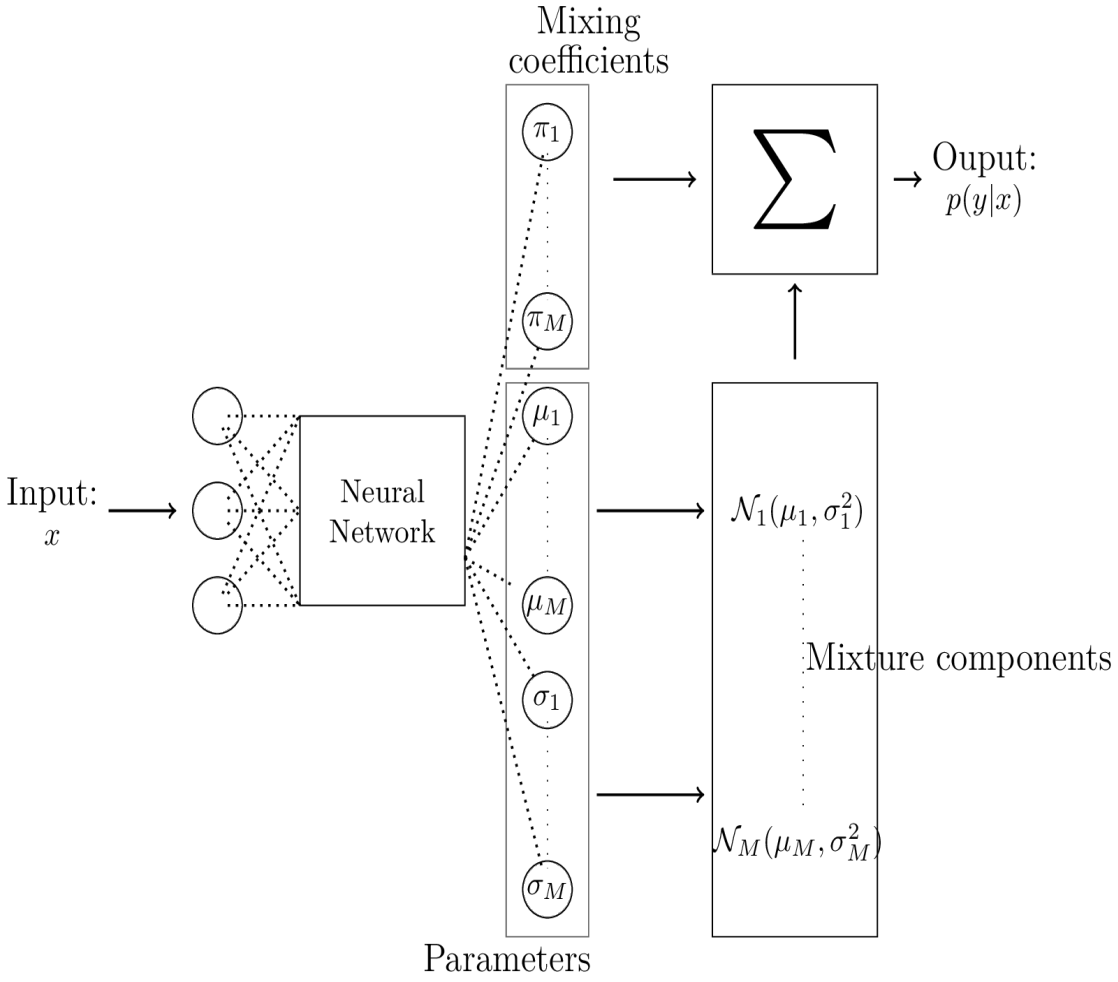


Figure 6: Structure of a Mixture Density Network. The Input $\boldsymbol{x}$ first gets passed through a neural network. The outputs of that are treated as parameters for a mixture model. The mixture model is a weighted linear sum of different distributions – normal distributions in our case. To note is that the mixing coefficients (the weights for the sum) are also generated from the network.

With MDNs we again use the nice property of neural networks as universal function estimators. Before we proceed further, however, we must place some restrictions on the outputs of the network. Let $\boldsymbol{z_m^\alpha}$, $\boldsymbol{z_m^\mu}$ and $\boldsymbol{z_m^\sigma}$ be the outputs of some neural network $\boldsymbol{f(x;\omega)}$ with some

parameter set $\boldsymbol{\omega}$ where $\boldsymbol{z}_m^{\alpha}$ affect the mixing coefficients $\boldsymbol{\alpha}_m$ and $\boldsymbol{z}_m^{\mu}$ and $\boldsymbol{z}_m^{\sigma}$ affect the means and variances of the mixture components respectively.

Now we give the exact relationships between the outputs and the mixture's parameters. The mixing coefficients of the mixture obviously must sum up to one as they are to be treated as probabilities:

$$\sum_{m=1}^{M} \boldsymbol{\alpha}_m(\boldsymbol{x}) = \mathbf{1} \tag{3.29}$$

We can achieve that by placing a *softmax* function on the respective outputs $\boldsymbol{z}_m^{\alpha}$ of the network

$$\boldsymbol{\alpha}_m = \frac{\boldsymbol{e}^{z_m^{\alpha}}}{\sum_{j=1}^{M} \boldsymbol{e}^{z_j^{\alpha}}} \tag{3.30}$$

where $\boldsymbol{z}_i^{\alpha}$ are the outputs of the networks "responsible" for the mixing coefficients $\boldsymbol{\alpha_i}$. Transforming the outputs of a network into a probabilities in this manner was introduced in [Bri90].

The variances of the Gaussian distributions represent *scales* and for that they must be positive. We ensure this through the use of an exponential function

$$\boldsymbol{\sigma}_m = \boldsymbol{e}^{z_m^{\sigma}} \tag{3.31}$$

This also avoids the problem when some of the variances goes to zero. The means $\boldsymbol{\mu}_m(\boldsymbol{x})$ are to be seen as *location* parameters and in order not to introduce bias in the system we model them directly with the network's corresponding outputs

$$\boldsymbol{\mu}_m = \boldsymbol{z}_m^{\mu} \tag{3.32}$$

With those definitions we can summarize – for a network with $\boldsymbol{M}$ mixtures the output would be $\{\boldsymbol{\mu}_m, \boldsymbol{\sigma}_m^2, \boldsymbol{\pi}_m\}, \boldsymbol{m} \in \{\mathbf{1}, \ldots, \boldsymbol{M}\}$. With the outputs we can define a conditional probability distribution for the data

$$\boldsymbol{p}(\boldsymbol{Y}|\boldsymbol{X}) = \sum_{m=1}^{M} \boldsymbol{\pi}_m(\boldsymbol{X}) \boldsymbol{\mathcal{N}}(\boldsymbol{y}|\boldsymbol{\mu}_m(\boldsymbol{X}), \boldsymbol{\sigma}_m^2(\boldsymbol{X})) \tag{3.33}$$

Now we just have to give a loss function to be minimized. Starting from probability perspective, we want to maximize the likelihood of the the observed data $\{(\boldsymbol{x}_n, \boldsymbol{y}_n)\}_1^N$, This is given by a product over the likelihoods of each data point:

$$\boldsymbol{\mathcal{L}} = \prod_{n=1}^{N} \boldsymbol{p}(\boldsymbol{x}_n, \boldsymbol{y}_n) = \prod_{n=1}^{N} \boldsymbol{p}(\boldsymbol{x}_n) \boldsymbol{p}(\boldsymbol{y}_n|\boldsymbol{x}_n) \tag{3.34}$$

The term $\boldsymbol{p}(\boldsymbol{x}_n)$ does not depend on the networks weights and can be spared. As for the $\boldsymbol{p}(\boldsymbol{y}_n|\boldsymbol{x}_n)$ – this is where the mixture model comes into play. The quantity to be maximized thus becomes

$$\tilde{\boldsymbol{\mathcal{L}}} = \prod_{n=1}^{N} \sum_{m=1}^{M} \boldsymbol{\alpha}_m(\boldsymbol{x}_n) \boldsymbol{\mathcal{N}}(\boldsymbol{y}_n; \boldsymbol{\mu}_m(\boldsymbol{x}_n), \boldsymbol{\sigma}_m^2(\boldsymbol{x}_n)) \tag{3.35}$$

Maximizing $\tilde{\boldsymbol{\mathcal{L}}}$ is equivalent to minimizing

$$\boldsymbol{E} = \sum_{n=1}^{N} \boldsymbol{E}^n \tag{3.36}$$

where $\boldsymbol{E^n}$ is the contribution of the $\boldsymbol{n^{th}}$ data point and

$$\boldsymbol{E^n = -\log(\sum_{m=1}^{M} \alpha_m(x_n)\mathcal{N}(y_n; \mu_m(x_n), \sigma_m^2(x_n)))} \tag{3.37}$$

This function $\boldsymbol{E}$ can be minimized with standard backpropagation technique ([BNVP01]) as its derivatives are fully tractable. Exact expressions for the derivatives of $\boldsymbol{E}$ are derived and given in [MB94] but we don't consider them here. Details on the implementation of the MDNs are given in Section 6.3

4. Model evaluation methods

In this section we give formal explanation to our approach in evaluating the models we build.

4.1. Probabilistic Forecast

We first explain the difference between the considered stochastic regression models and classical ones.

4.1.1. Prediction spaces

Classical regression models produce a point estimate $\boldsymbol{y_i^e}$ for each sample data point $\boldsymbol{x_i}$ with an actual realization $\boldsymbol{y_i}$. In this thesis, however, we are interested in regression models whose output is probabilistic forecast $\boldsymbol{F}$ that can be identified with the associated *cumulative distribution function* (*CDF*) as described in Definition 1.1. This introduces the notion of prediction spaces proposed in [GR13]. A prediction space is a probability space tailored to the study of distributional forecasts ([GK14]). Formally speaking we have to consider the realized observation also as a distribution even thou in practice this is a single value. In the most general case, the elements of the prediction space can be seen as tuples $\boldsymbol{(F, Y)}$ where $\boldsymbol{F}$ is the probabilistic forecast and $\boldsymbol{Y}$ is the true distribution. $\boldsymbol{F}$ is a CDF-valued quantity which caries some information about the training data, the models' parameters, certain assumptions that are considered met and other model specific properties. Let $\boldsymbol{\mathcal{A}}$ denote all of the possible information that is available to make a forecast and let $\boldsymbol{\mathcal{L}}$ be some conditional distribution. We can say that the forecast $\boldsymbol{F}$ is ideal relative to the information encoded by $\boldsymbol{\mathcal{A}}$ if $\boldsymbol{F = \mathcal{L}(Y|A)}$. This means that the forecast utilizes the information completely.

In our concrete case, the model – BNN or MDN – is the mechanism which generates the forecast. The models are trained on the training set and thus can "gather" information about the data. When making a prediction about new data point, the models use the gathered data and produce a probabilistic forecast in form of either CDF or a set of samples drawn from the predicted distribution.

4.1.2. Dispersion and sharpness

Dispersion and *sharpness* are properties of predictive forecasts that describe the relationship between the observed value and the prediction. In order to formalize them, however, we need to introduce the notion of probability integral transform. For more information on probability integral transform see [DGT98], [GBR], [Daw84].

Again, let $\boldsymbol{F}$ be the predicted distribution (see Definition 1.1) for an observation $\boldsymbol{Y}$ in the sense of Definition 1.1. The *probability random transform* (*PIT*) is the random variable $\boldsymbol{Z_F = F(Y)}$. Intuitively the PIT is the value of the predictive CDF.

With that we can formally define what dispersion and calibration are. In the following definitions $\boldsymbol{F}$ and $\boldsymbol{G}$ are two forecasts and their PITs are respectively $\boldsymbol{Z_F}$ and $\boldsymbol{Z_G}$. We say that

- the forecast $\boldsymbol{F}$ is marginally calibrated, if $\mathbb{F}(\boldsymbol{y}) = \mathbb{P}(\boldsymbol{Y} \leq \boldsymbol{y})\, \forall \boldsymbol{y} \in \mathbb{R}$.
- the forecast $\boldsymbol{F}$ is probabilistically calibrated, if its PIT $\boldsymbol{Z_F}$ has standard uniform distribution.
- the forecast $\boldsymbol{F}$ is overdispersed, if $\boldsymbol{var(Z_F)} < \frac{\boldsymbol{1}}{\boldsymbol{12}}$ and underdispersed if $\boldsymbol{var}(\boldsymbol{Z}_{\mathrm{F}}) > \frac{\boldsymbol{1}}{\boldsymbol{12}}$.
- the forecast $\boldsymbol{F}$ is more dispersed, if $\boldsymbol{var(Z_F)} < \boldsymbol{var(Z_G)}$.

As mentioned in Section 1.2 calibration has to do with both the observation and the forecast. According to [GK14], if a forecast is ideal relative to some information set then it is both marginally and probabilistically calibrated.

As explained in [GK14] the sharpness of a given forecast describes the concentration of the distribution without consideration of the actual observed value. When it comes to forecasts for a real-valued variable, sharpness can be assessed in terms of the associated prediction intervals. The mean widths of these intervals should be as short as possible.

In next section we'll explain a certain class of scoring rules that can evaluate a predicted forecast against the observed value with respect to the dispersion and sharpness properties i.e. they assign "better" score to forecast with "better" dispersion and sharpness.

4.2. Proper Scoring rules

Proper scoring rules allow us to measure the predictive capability of a given probabilistic forecast. Those rules can consider the sharpness as well as the concentration of the distribution. The generated scores are measurement of differences between a desired and generated distributions and thus we strive to minimize those scores. Proper scoring rules are explained in details in [GBR]. A scoring rule assigns a numerical score $\boldsymbol{S(F, y)}$ to each pair $\boldsymbol{(F, y)}$ of probabilistic forecast $\boldsymbol{F}$ in a form of probability distribution and an observed value $\boldsymbol{y} \in \mathbb{R}$. In general, a scoring rule is some function $\boldsymbol{S : \mathcal{F} \times \mathbb{R} \to \mathbb{R}}$ where $\boldsymbol{\mathcal{F}}$ is some class of probability distributions. A scoring rules generally evaluates a predicted distribution against an realized observation. Let the desired distribution be $\boldsymbol{G} \in \boldsymbol{\mathcal{F}}$. Then we write

$$\boldsymbol{S(F, G)} = \mathbb{E}_{\boldsymbol{G}}[\boldsymbol{S(F, Y)}] \tag{4.1}$$

for the expected score between $\boldsymbol{G}$ and the forecast $\boldsymbol{F}$. The rule $\boldsymbol{S : \mathcal{F} \times \mathbb{R} \to \mathbb{R}}$ where $\boldsymbol{\mathcal{F}}$ is proper if the following property holds

$$\boldsymbol{S(G, G)} \leq \boldsymbol{S(F, G)}, \quad \forall \boldsymbol{G, F} \in \boldsymbol{\mathcal{F}} \tag{4.2}$$

This means that a proper scoring rules yields a minimal score for the true distribution of the observation. We consider three proper scoring rules. The following sections discuss these.

4.2.1. Logarithmic score (LS)

The most straight forward scoring rule is probably the *logarithmic score*. This scoring rule is well established. One of its first propositions is made in [Goo52]. For a given density function $\boldsymbol{f^p(x)}$ that belongs to a predicted forecast, the LS is given by

$$\boldsymbol{LS(f^p, y)} = -\log\, \boldsymbol{f^p(y)} \tag{4.3}$$

where $\boldsymbol{y}$ is the realized observation. The LS is lower (i.e. better), if according to the predicted forecast, the true observation is more probable. The LS would be at its lowest if the observation $\boldsymbol{y}$ is the maximum of the density $\boldsymbol{f^p}$. In this case, the actual observation will be the most probable value for the prediction according to the generated forecast. It is clear that sharper distributions will have lower LS given the observations falls into them. More concentrated distributions have higher values for their densities and thus are more certain for given observation.

The LS is the basis for a class of scoring rules called *local scoring rules*. Those are discussed in detail in [EG12].

A possible disadvantage of the LS is that it requires the density function of the forecast in order to be calculated. If the forecast is not given in density form, the probability density function must be approximated. For simple distributions this may not be a problem but for complicated ones, a considerable amount of samples from the distribution may be required. This can make the LS impractical for some probability models.

4.2.2. Continuously ranked probability score (CRPS)

The mentioned disadvantage of the LS is addressed by the so called *continuously ranked probability score* (*CRPS*). The CRPS uses the predictive cumulative distribution function (CDF) of the forecast to give it score. CRPS is denied as

$$\boldsymbol{CRPS(F, y)} = \int_{-\infty}^{\infty} (\boldsymbol{F(x)} - \mathbb{1}\{\boldsymbol{y \leq x}\})^{\boldsymbol{2}} \boldsymbol{dx} \tag{4.4}$$

$$= \mathbb{E}_{\boldsymbol{F}}|\boldsymbol{Y - y}| - \frac{\boldsymbol{1}}{\boldsymbol{2}}\mathbb{E}_{\boldsymbol{F}}|\boldsymbol{Y - Y'}| \tag{4.5}$$

where $\boldsymbol{Y}$ and $\boldsymbol{Y'}$ are are independent random variables with CDF $\boldsymbol{F}$ and finite first moment and $\mathbb{1}\{\boldsymbol{y \leq x}\}$ is the probability for $\boldsymbol{y}$ to be less than or equal to $\boldsymbol{x}$. The second part of the equation is derived and described in [GR07] and [MW76]. In plain, words the definition says that the CRPS is a quadratic measure of the difference between the forecast's CDF and the empirical CDF of the observation. In contrast to the LS, the CRPS considers the distribution as a whole and doesn't focus on one specific point. CRPS is well studied and it finds application in a lot of cases (see [GR11] for weighted variant and [GGBJ07] for usage for circular variables). Figure 7 gives visual intuition of the CRPS. From the figure one can clearly see that if the variance of the predicted distribution is big, the CDF will have values different form **0** and **1** in bigger interval and thus the CRPS will be higher. This shows how the CRPS encourages forecasts to be sharper with their distribution.

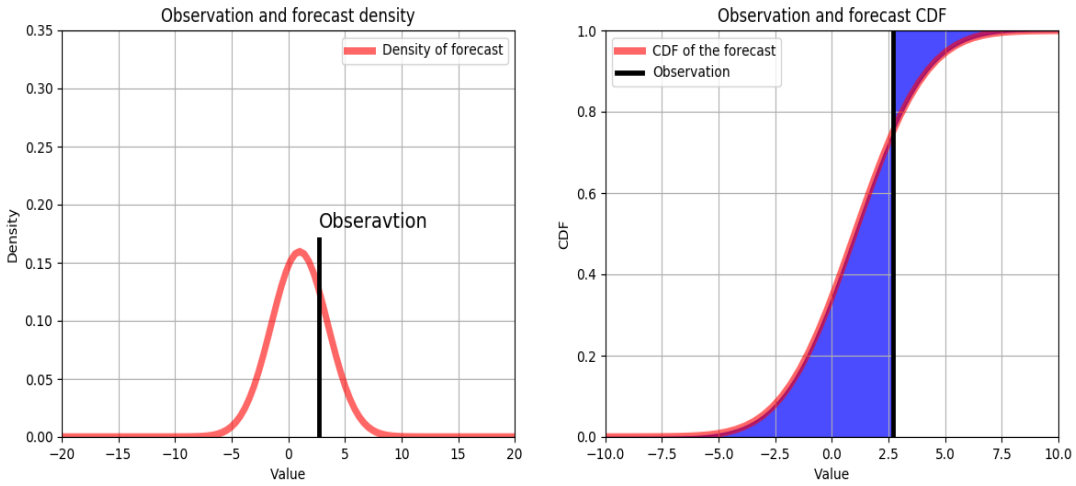


Figure 7: The square of the blue area is the value of the CRPS. Notice how if the predicted CDF were a step function (the case when the prediction is deterministic point estimate), the CRPS will be nothing more than the squared differnece between the observation and the prediction.

A useful property of the CRPS is that it reduces to the *mean absolute error* (*MAE*) if the forecast is deterministic. This allows us to compare probabilistic forecasts with point estimates in consistent manner ([GK14]).

4.2.3. Dawid–Sebastiani score (DSS)

For complex distributions even the CRPS can be hard to evaluated as it requires some estimate of the integral in Equation 4.4. The *Dawid–Sebastiani score* (*DSS*), introduced in [DS99], relies only on the first two central moments of a given predicted forecast $\boldsymbol{F}$. DSS is defined as

$$\boldsymbol{DSS(F, y)} = \frac{(\boldsymbol{y} - \boldsymbol{\mu_F})^2}{\boldsymbol{\sigma_F^2}} + \boldsymbol{2 \log(\sigma_F)} \tag{4.6}$$

Of the three rules the DSS is most simple to calculate as it requires only two simple metrics of the forecast. The only thing we need is the ability to draw samples from the distribution of the forecast and we can estimate the mean and variance.

4.2.4. Computation of scoring rules

In this thesis we deal with sets of data and we need to evaluate the scoring rules on certain number of data points – on the test or train set. For this we take the average over the scores of all examples in a set. Let $\{\boldsymbol{x_n}, \boldsymbol{y_n}\}_{\boldsymbol{n=1}}^{\boldsymbol{N}}$ be our set of interest and some model have generated set of forecasts $\{\boldsymbol{F_n}\}_{\boldsymbol{n=1}}^{\boldsymbol{N}}$ where forecast $\boldsymbol{F_i}$ is based on the features $\boldsymbol{x_i}$ and tries to

predict the observation $\boldsymbol{y_i}$. The overall score $\bar{\boldsymbol{S}}_{\boldsymbol{N}}$ of the scoring rule $\boldsymbol{S}$ then is

$$\boldsymbol{S_N} = \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} \boldsymbol{S}(\boldsymbol{F_i}, \boldsymbol{y_i}) \tag{4.7}$$

This is the final metric on which we compare several probability models.

When it comes to calculating the rules themselves on concrete observations, there are explicit forms for the three rules, if the predicted forecast is a normal distribution. Those are given in Table 1 and taken from [GK14].

Scoring rule $\boldsymbol{S}$	$\boldsymbol{S(\mathcal{N}(\mu, \sigma^2), y)}$
Logarithmic score	$\frac{(y-\mu)^2}{2\sigma^2} + \log(\sigma) + \frac{1}{2}\log(2\pi)$
Continuous ranked probability score	$\sigma(\frac{y-\mu}{\sigma}(2\Phi(\frac{y-\mu}{\sigma}) - 1) + 2\phi(\frac{y-\mu}{\sigma}) - \frac{1}{\pi})$
Dawid–Sebastiani score	$\frac{(y-\mu)^2}{\sigma^2} + 2\log(\sigma)$

Table 1: Closed forms of the scoring rules for normal distribution

All of the examined model in the thesis can generate samples from their predicted forecasts. The DSS can thus be trivially calculated just through the empirical mean and variance of the generated samples. The predicted density function of the MDN is a sum of density functions of the mixture components (see Section 3.3) so it can easily be evaluated with the LS. For the BNN we use density estimation methods described in [Sil86] and [Sco15] to calculate the LS. Empirical CDFs are trivially computable with sample data. For sample set $\{\boldsymbol{x_i}\}_i = \mathbf{1}^n$ the empirical CDF is

$$\bar{\boldsymbol{F}}_{\boldsymbol{n}}(\boldsymbol{t}) = \frac{\mathbf{1}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \mathbf{1}_{\boldsymbol{x_i} \leq \boldsymbol{t}} \tag{4.8}$$

wehre (4.9)

$$\mathbf{1}_{\boldsymbol{a} \leq \boldsymbol{t}} = \begin{cases} \mathbf{1} & \text{, if } \boldsymbol{a} \leq \boldsymbol{t} \\ \mathbf{0} & \text{, if } \boldsymbol{a} > \boldsymbol{t} \end{cases} \tag{4.10}$$

With this function we can also calculate the CRPS in straightforward manner following the definition in Equation 4.4.

4.3. Feature Importance

In order to asses the relative importance of the features used by the build models we use a method called *permutation importance*. It is first described in [Bre01] and it was applied to random forests. The techniques is also successfully applied in assessing the feature importance of ensemble forecasting[2] methods for forecasting ([RL18]). We closely follow the approach of [RL18] by randomly shuffling each feature in the test and training sets and then

[2]Ensemble forecasting is a method used in numerical weather prediction. Instead of making a single forecast of the most likely weather, a set (or an ensemble) of forecasts is produced [Wik18].

look at the mean value of a given scoring rule. For each feature we examine the difference in the mean value of the scoring rules between the permuted and regular datasets.

We give formal definition of the described above. We consider a set of data $\{\boldsymbol{x_n}, \boldsymbol{y_n}\}_{n=1}^{N}$ with feature vectors $\boldsymbol{x_n} \in \mathbb{R}^M$ and realized values $\boldsymbol{y_n} \in \mathbb{R}$. We define a matrix $\boldsymbol{X}$ with the features of all examples as

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \\ \vdots \\ \boldsymbol{x_N} \end{bmatrix} \tag{4.11}$$

In this notation, the $\boldsymbol{i^{th}}$ row of the matrix are all of the features of the $\boldsymbol{i^{th}}$ example and the $\boldsymbol{i^{th}}$ column of the matrix are all of the $\boldsymbol{i^{th}}$ features of all examples. We further define $\boldsymbol{F|x}$ as the conditional forecast distribution given a vector of features of a input data point. We assume that $\boldsymbol{F|x}$ is predicted by some model. In essence we evaluate the importance of the used feature only in the context of the considered model. In this sense we examine which of the features have higher information value for the predictions of the given model.

Now we give formal definition for the permutation of the features. Let $\boldsymbol{X_s}$ denote the $\boldsymbol{s^{th}}$ column of the input data $\boldsymbol{X}$. Then the permuted set $\boldsymbol{X}^{\mathrm{perm_j}}$ for the $\boldsymbol{j^{th}}$ feature is given by

$$\boldsymbol{X}^{\mathrm{perm_j}} = \begin{cases} \boldsymbol{X_s}, & \text{if } \boldsymbol{j \neq s} \\ \boldsymbol{\pi(X_{(s)})}, & \text{if } \boldsymbol{j = s} \end{cases} \tag{4.12}$$

where $\boldsymbol{p(x)}$ is some random permutation of the elements of the vector $\boldsymbol{x}$. Now we can give definition for the quantity that we call feature importance.

$$\textit{Importance}(\boldsymbol{j}) = \frac{\boldsymbol{1}}{\boldsymbol{N}} \sum_{\boldsymbol{n=1}}^{\boldsymbol{N}} (\boldsymbol{SR(F|X^{perm_j,(n)}, y_n) - SR(F|X^{(n)}, y_n)}) \tag{4.13}$$

where $\mathrm{X^{(n)}}$ is the feature vector of the $\boldsymbol{n^{th}}$ datapoint (i.e. the $\boldsymbol{n^{th}}$ row of the matrix $\boldsymbol{X}$) and SR is some scoring rule – CRPS, DSS or LS in our case.

4.4. Rank Histogram

Rank histograms (sometimes called *verification rank histogram*) are a tool for evaluating ensemble forecasts. See [Ham01] for details. In our work, however, we adapt them to evaluate forecasts in form of probability distributions. The underlying assumption is that the ensemble member forecasts are distributed so as that the observed value behaves as a member of the ensemble. Let $[\boldsymbol{y_1^p}, \ldots, \boldsymbol{y_M^p}]$ be a *sorted* by value ensemble where each $\boldsymbol{y_i^p} \in \mathbb{R}$ is a predicted value for the true observation $\boldsymbol{y^t}$. The rank of $\boldsymbol{y^t}$ according to the ensemble is an integer $\boldsymbol{k}$ such that

$$\boldsymbol{rank}(\boldsymbol{y^t}) = \begin{cases} \boldsymbol{0}, & \text{if } \boldsymbol{y^t < y_1^p} \\ \boldsymbol{k}, & \text{if } \boldsymbol{y_k^p \leq y^t < y_{k+1}^p} \\ \boldsymbol{k+1}, & \text{if } \boldsymbol{y_{k+1}^p < y^t} \end{cases} \tag{4.14}$$

In our case the sorted ensemble is just a sorted sequence of drawn samples from the distribution predicted by the given model. We calculated the rank of each observation and then create a

histogram with certain number of bins. To be noted is that the number of bins is considerably less than the number of samples. The amount of samples drawn from each model for each observation is in the order of tens of thousand while the number of bins for the rank histogram is no more than twenty.

For a well calibrated forecast, the rank histogram should be uniform. U-shaped histograms indicate underdispersed predictive distributions, whereas hump or inverse-U-shaped histograms correspond to overdispersed predictive distributions. These considerations are described in [GK14], [DGT98] and [Ham01]. If the histogram is uniform we can say that the observation behaves as a sample from the forecast. This is essentially the behavior which a good forecast should exhibit. Figure 8 illustrates the said about rank histograms.

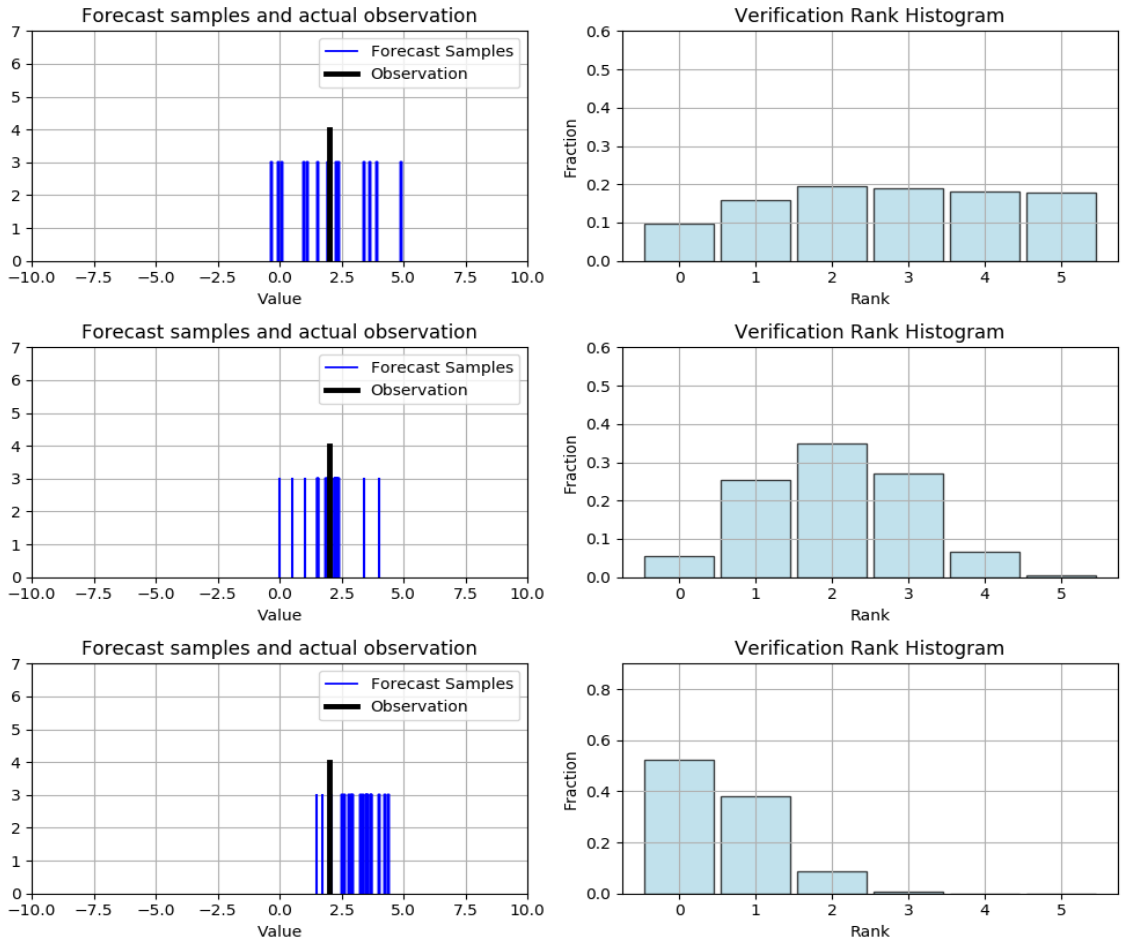


Figure 8: Histograms illustrating different types of distributions with respect to the actual observations. To note is that on left we give only one example of samples drawn form the distribution in respect to a given observation. We assume that the sample sets for the other observations are similar and that this will produce the respective histograms on the right. A rank histogram cannot be given for a single observation and a single set of drawn samples.

4.5. Predictive performance check

In this section we explain how can we formally test if one forecasting model is better than another one.

In order to conclude which one of given models is batter than the other ones, we want something more than just their average score of some rule over a test set. We introduce a statistical test – the *Diebold–Mariano test* – which can asses the predictive performance of two forecasting methods in respect to one another and conclude which of them is better at predicting some data. The use of this particular test is suggested in [GK14] and we follow and explain the approach given there.

Let $\boldsymbol{F}$ and $\boldsymbol{G}$ be two competing forecasting methods and let $\boldsymbol{F_i}$ and $\boldsymbol{G_i}$ be the forecasts each of the corresponding models produces for the actual observation $\boldsymbol{y_i}$. We can asses which of the methods has better predictive capabilities with the following approach. First we calculate the average values of some proper scoring rule over the entire test set with $\boldsymbol{n}$ observations

$$\bar{\boldsymbol{S}}_{\boldsymbol{n}}^{\boldsymbol{F}} = \frac{\boldsymbol{1}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{S}(\boldsymbol{F_i}, \boldsymbol{y_i}) \quad \text{and} \quad \bar{\boldsymbol{S}}_{\boldsymbol{n}}^{\boldsymbol{G}} = \frac{\boldsymbol{1}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \boldsymbol{S}(\boldsymbol{G_i}, \boldsymbol{y_i}) \tag{4.15}$$

where $\boldsymbol{S}$ is some proper scoring rule. Then we calculate the metric

$$\boldsymbol{t_n} = \sqrt{\boldsymbol{n}} \frac{\bar{\boldsymbol{S}}_{\boldsymbol{n}}^{\boldsymbol{F}} - \bar{\boldsymbol{S}}_{\boldsymbol{n}}^{\boldsymbol{G}}}{\hat{\boldsymbol{\sigma}}_{\boldsymbol{n}}} \tag{4.16}$$

where

$$\hat{\boldsymbol{\sigma}}_{\boldsymbol{n}}^{\boldsymbol{2}} = \frac{\boldsymbol{1}}{\boldsymbol{n}} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{S}(\boldsymbol{F_i}, \boldsymbol{y_i}) - \boldsymbol{S}(\boldsymbol{G_i}, \boldsymbol{y_i}))^{\boldsymbol{2}} \tag{4.17}$$

This approach is known as the Diebold–Mariano test and it is first introduced in [DM95]. The method is a statistical test of equal forecast performance. The null hypothesis is that the two forecasting methods predict the data equally bad or good. This means that we cannot prefer one of the methods over the other. If the null hypothesis is rejected, we can say which of the two is better than the other. Under the null hypothesis, the quantity $\boldsymbol{t_n}$ follows the standard normal distribution – $\boldsymbol{\mathcal{N}(0,1)}$. This makes it particularly easy to calculate the corresponding $\boldsymbol{p}$-value of the test. Given the $\boldsymbol{p}$-value is small enough, the method $\boldsymbol{F}$ is preferred if $\boldsymbol{t_n}$ is negative, whereas $\boldsymbol{G}$ is preferred if $\boldsymbol{t_n}$ is positive (see [GK14]).

Detailed information on the Diebold–Mariano test can be found in [Die15].

5. Results

In this section we present the results from the evaluation of the trained models.

5.1. Methodology and data

We first describe exactly what kind of datasets we use to train the model and what feature considerations we make.

As described in Section 1.3, the data comes from different places and further more it is in different formats. Therefore we are responsible for generating the final datasets on which we train the models. This allows us to be flexible in the used data and therefore be able to investigate how the different resolution of the data affects the performance of the models. We describe the generation process in details in Section 6.2. We have generated three different datasets:

- Daily averaged data – For the whole considered period – the year of 2017 – we take the average pollution value (PM10 and PM2.5) of each sensor for each day. We treat the vector of those values as a single entry in the dataset. Naturally this means that we have **365** entries in this dataset.
- Twelve hourly averaged data – Analogous to the daily average but we take the average values over twelve hour periods over the entire year. **730** entries in the entire dataset.
- Hourly averaged data – We consider the average pollution value of each sensor over each hour in the year. The count of entries in this dataset is **8760**.

We look at the performance of the models over each set in the respective sections. In Figure 9 we show plots of the values coming from a single sensor. On the size of the value vectors for entries in each datasets – after the data cleaning (described in Section 6.2) we are left with sixteen sensors from *luftdaten.info* and we also have three LUBW sensors. This adds up to nineteen sensors over all. For each of those we have PM10 and PM2.5 values. This means that for each day, twelve hours or one hour we have thirty eight values. We next describe how we choose subset of those to train models predicting a given another one of them.

As said, we want to investigate how different data affects the performance of the models. For that, we consider several types of orthogonal criteria.

- *Predicted LUBW sensor* – As mentioned, there are altogether three LUBW sensors[3]. The names of those are *SBC*, *SNTR* and *SAKP*. We want to see the predictive performance of the models based on which of these stations we are trying to predict. Plots of the data coming from these sensors can be seen in Figure 10.
- *Pollution value type* – The air pollution values that all of the sensors measure are PM2.5 and PM10. See Section 1.3 for more information on the meaning of these values. In our implementation and considerations we call those *P1* and *P2* respectively (see Section 6.2). While preprocessing the data, we have found out that P1 and P2 are highly correlated.

[3] We use sensor and station interchangeably

On average their correlation is about **0.95**. This means that they effectively measure the same thing and will only make the feature space overly complicated, if both are considered as predictors[4]. For this reason we either train on P1 or P2 values while predicting P1 or P2 respectively.

- *Using the rest of the LUBW senors as predictor* – We want to see the difference in predictive performance of the models based on including the rest of the LUBW stations as predictors. That it, how much better or worse become the models, if we use values of the two other LUBW stations when we predict the values of the third one.

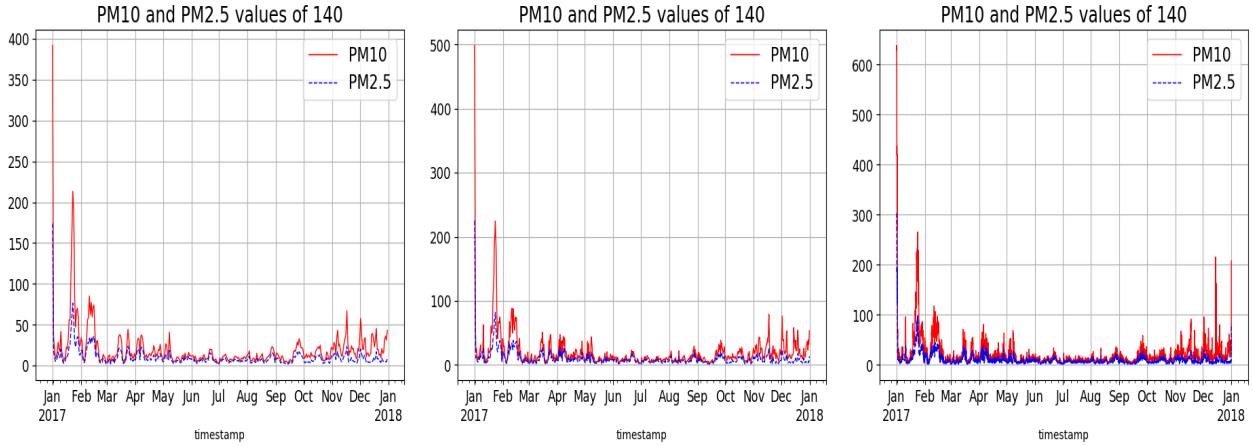
The final distinction about the models that we make is based on hyper-parameters of each of them. For the MDNs we can choose mixture components count and the definitions of the hidden layers. For the BNNs those hyper-parameters are again the definitions of the hidden layers as well as the drawn samples from the model while calculation the stochastic gradient descent. We consider two MDN-models – one with one hidden layer with **512** neurons and three mixture components and one with one hidden layer but with **1024** neurons in it and again three mixture components. Both of the models were trained for maximum of six million and three thousand iterations so we give those models the names `mdn_l[512]_i6300000_mc3`[5] and `mdn_l[1024]_i6300000_mc3` respectively. When it comes to the BNNs, we only trained one model with one hidden layer and **512** neurons in it, where the drawn samples for the gradient calculation were **800**. The BNNs were trained for maximum of **3200** iterations. The id of the BNNs is therefore `bnn_l[512]_i3200_s800`.

Each of these distinctions in the construction of the models can be made independently of the other ones. We consider all possible combinations and we compare the results in Section 5.2. By the training of the models, we used 75% of the available data for the and the rest was used for the evaluation. All of the considered results are over the test set. To note is that the test-train split is not random. With the built models we want to predict future values of the sensors. Therefore we always split the data while preserving the order of the values – from the start of the year till the end. In this sense, we always take the "first" 75% of the values in the corresponding dataset and we use them as train-set and we try to predict the values from the "future" of the year.

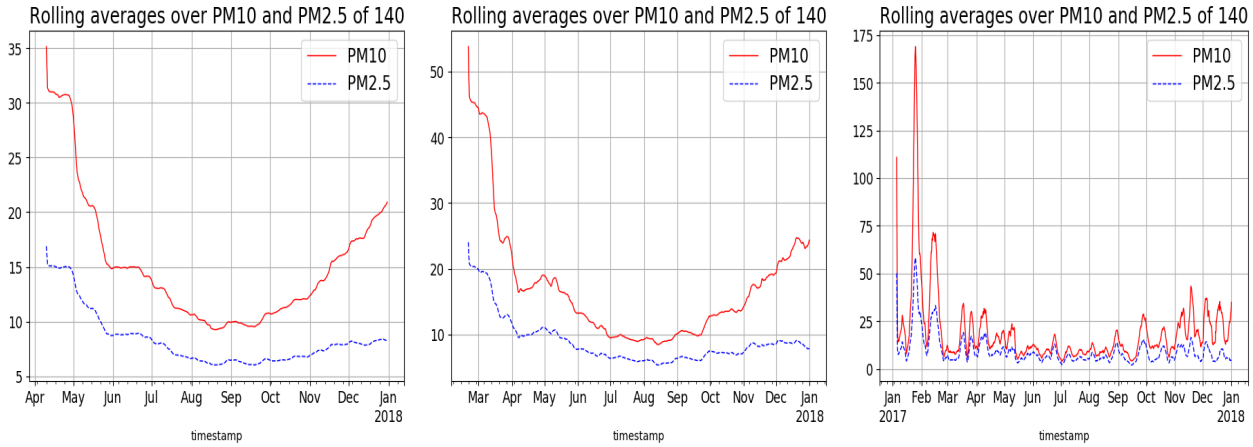
We like to note that the training of the BNN models took significantly longer that than the training of the MDNs. All of the models were trained on a cluster of computers and overall one hundred and forty CPUs were used. The training of all of the MDNs took no longer than four days while the average paste of the BNNs was one trained model per day.

[4] We use predictors and features interchangeably

[5] The `l` signifies the layers, the `i` the iterations and `mc` the mixture components

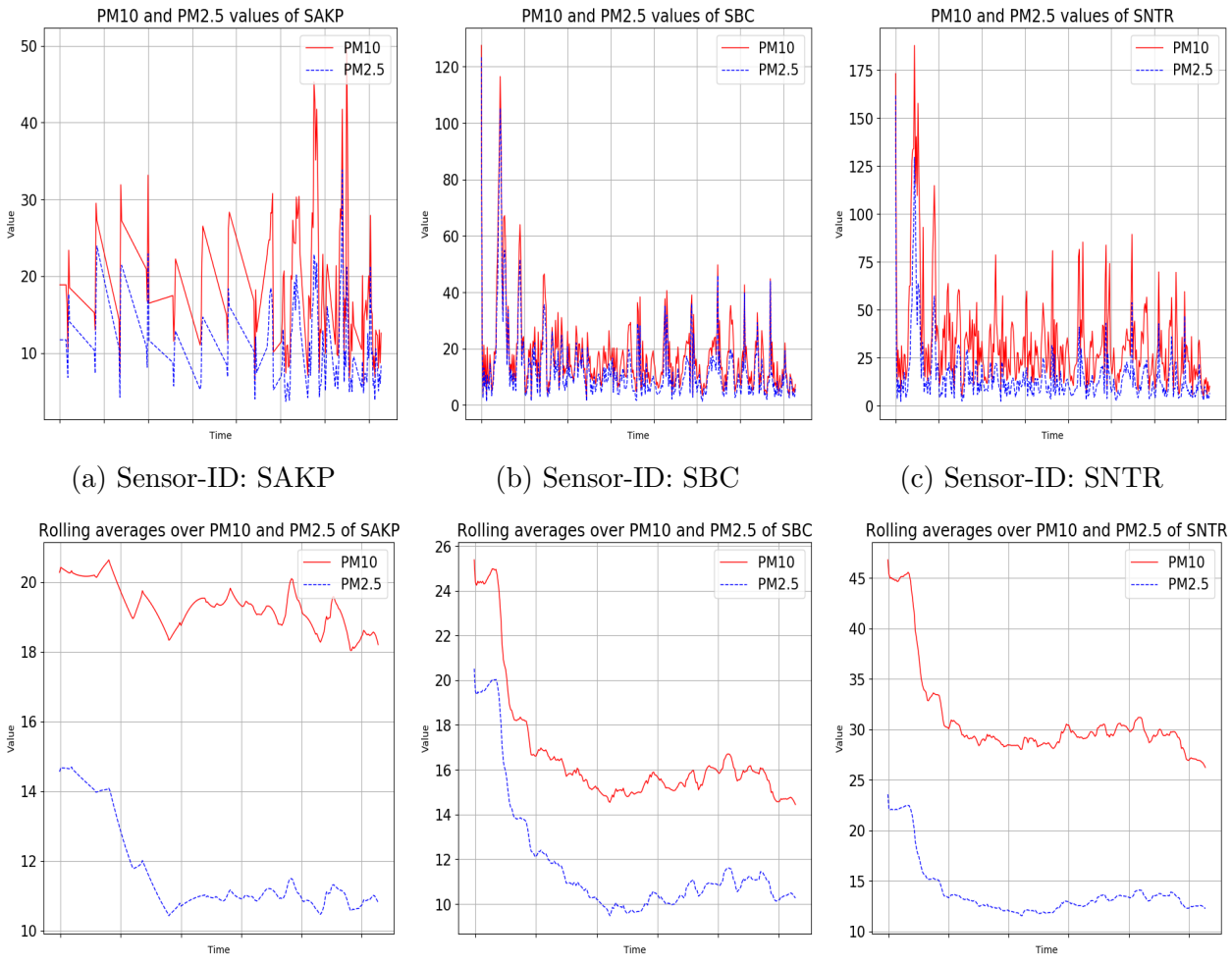


(a) Senor-ID: 140, daily average (b) Senor-ID: 140, twelve hour average (c) Senor-ID: 140, hourly average



(d) Senor-ID: 140, daily average, moving average (e) Senor-ID: 140, twelve hour average, moving average (f) Senor-ID: 140, hourly average, moving average

Figure 9: Different views of the data from a single sensor. The P1 value is displayed in red and P2 in blue. The plots on the top row show the actual values over the entire year depending on the averaging period – daily averaged in (a), twelve hour average in (b) and hour average in (c). (d), (e) and (f) show a moving average of 100 values of the respective plot above.



(a) Sensor-ID: SAKP (b) Sensor-ID: SBC (c) Sensor-ID: SNTR

(d) Sensor-ID: SAKP, moving average (e) Sensor-ID: SBC, moving average (f) Sensor-ID: SNTR, moving average

Figure 10: The plots here show data from the tree LUBW sensors – SAKP, SBC and SNTR respectively – but only for the daily averaging period. The top row again shows the values for each day while the bottom is a moving average of 100 values. For explanation on the values of SAKP, see the note at the end of the chapter.

When it comes to evaluation of the models, we evaluate each one differently generated model with the three proper scoring rules described in Section 4.2. We apply the feature importance technique from Section 4.3 only with respect to the CRPS as in [RL18]. We look at the rank histograms of only some of the models. For details on rank histograms see Section 4.4. We also comment on the results of the predictive performance checks.

Note: During the data preprocessing process we have found out that one of the LUBW sensors – SAKP – has data only for the last three months of the year. We have tried to interpolate the data for the rest of the year. This, however, means that the data from the sensors is almost unusable. We have decided to continue considering the sensor as we would, if this was not the fact. In the evaluation of the modes we hope to see results consistent with the made observation about the sensor. Namely:

- Predicting the values of SNTR is impossible as the values of the the sensor are generated independently of the values of the other sensors.

- The importance of this sensors as predictor is very low for the same reason as above

We reference back this considerations while discussing the observed results.

5.2. Evaluation of the models

As described in the previous section, we consider three different datasets based on the averaging period of the values. We look at the results on each of them in separate sections.

5.2.1. One day period

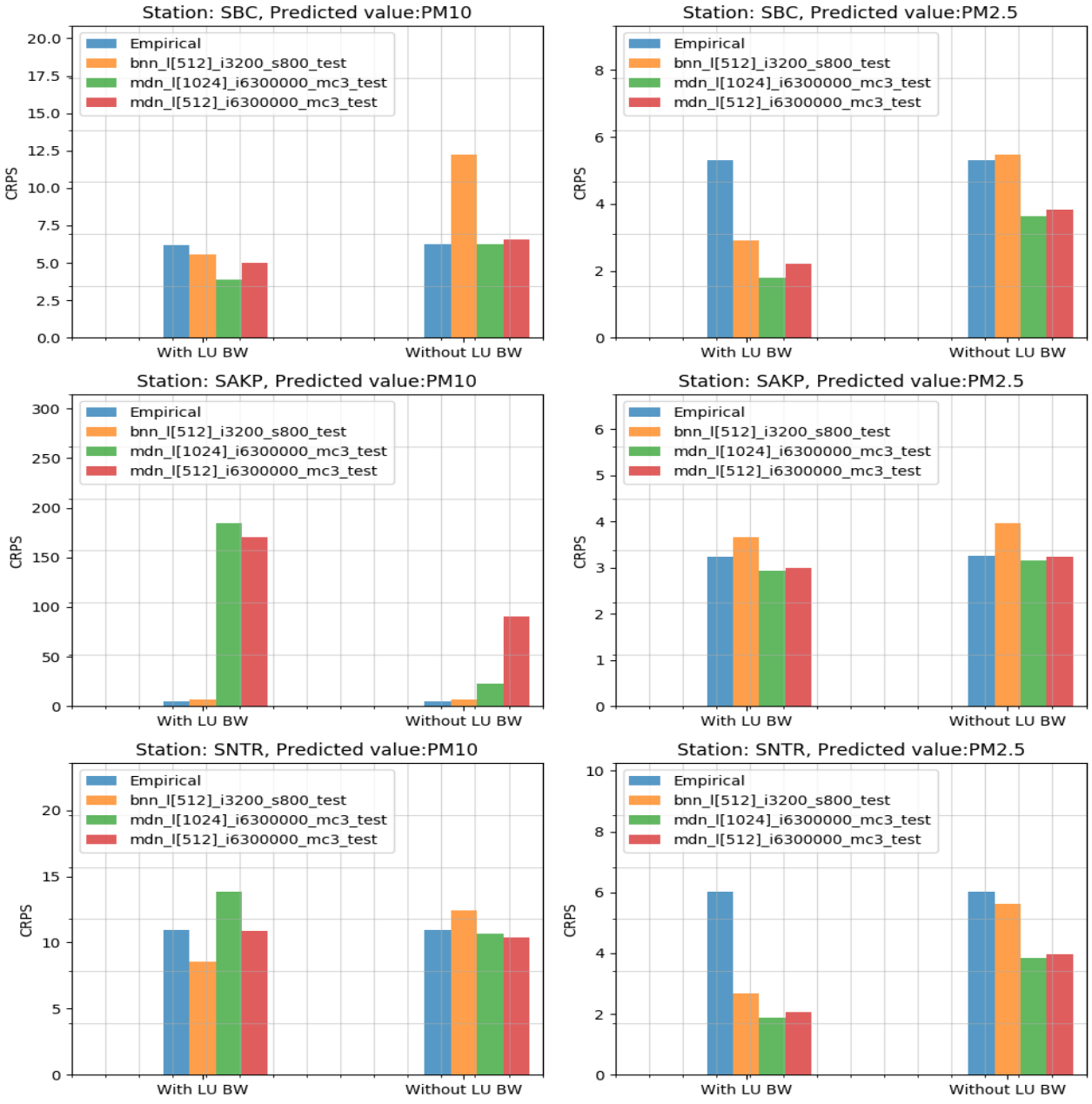


Figure 11: CRPS values of the models (lower is better).

Figure 11 illustrates the results of the models with respect to the CRPS-rule. The plots in the first column show the results models predicting the PM10 value of the corresponding LUBW-sensor whereas those in the second column predict the PM2.5 value. The distinction between the predicted LUBW-sensors is made rowwise.

From the very first plot (predicting PM10 of station SBC) we see how the use of data from the LUBW-senors has effect on the performance of the models. Under the use of the LUBW-sensors the examined models outperform the set by the Empirical model baseline whereas this is not the case without it. This can also be observed while looking at the second plot (predicting PM2.5 of station SBC) where the performance of the BNN model drops significantly without the use of data from the LUBW-sensors. The change by the MNDs is, however, not that drastic and those models even manage to be better than the empirical model without the use of LUBW-data.

The results of predicting the values of SNTR are consistent with the first observations especially in the case of PM2.5. There we can also see the performance drop even though all of the models remain better than the Empirical one. We can again see that the sensibility to the use of LUBW-data is higher by the BNN model. It is interesting to note that in the case of the PM10 prediction, the performance of some of the MDNs is slightly higher without LUBW-data.

As it comes to predicting the values of SAKP, the results are quite inconsistent. This is to be expected as we explained in the previous section. All of the models cannot reliably predict the PM10 value. It is interesting to see that the scores are "good" for the prediction of the PM2.5 value. We assume, however, that because the models were unable to extract any meaningful relationship between the features and the predicted value, the variance of the predicted distributions had become large in order to compensate. By examining the plots of the made by the models predictions, we see that this is indeed the case. See Figure 12

Figure 12: Plot of one of the MDN models where the predicted value is from SAKP. The plot on the top shows the results on the train set and the one on the bottom on the test set. We can clearly see that the variance of the predicted distributions is quite high most of the time. The plots of the other MDN models exhibit similar behavior.

Upon investigating the rank histograms of the models, we have found out that all of them are U-shaped. Rank histograms that are representative of all built models can be found in Appendix G. Rank histograms like these suggest that the models often fail to include the actual observation their confidence region. The observation falls then outside of the predicted distribution. This can be seen in the plots of the predictions of the models in Appendix F. The plots there even suggest that the models are prone to overfit the data and have significantly lower variance on the test set than on the train set. For compression, plots with the predictions of the empirical models are to be found in Appendix H

The results of the predictive performance checks are consistent with the observed CRPS values. Tables with results of the predictive performance checks are to be found in Appendix E. When predicting values of SBC or SNTR, both MDNs and the BNN reliably outperform the empirical model and the results supporting this are statistically significant with $\boldsymbol{p}$-value lower than $\mathbf{0.05}$. According to the checks, the MDN with $\mathbf{1024}$ neurons in its hidden layer is the best model while predicting PM10 of SBC and BNN is the best while predicting PM10 of SNTR.

Plots with the rest of the scoring rules can be found in Appendix C

The results considered in this section can be found in tabular form with concrete numbers in Appendix D.

5.2.2. Twelve hour period

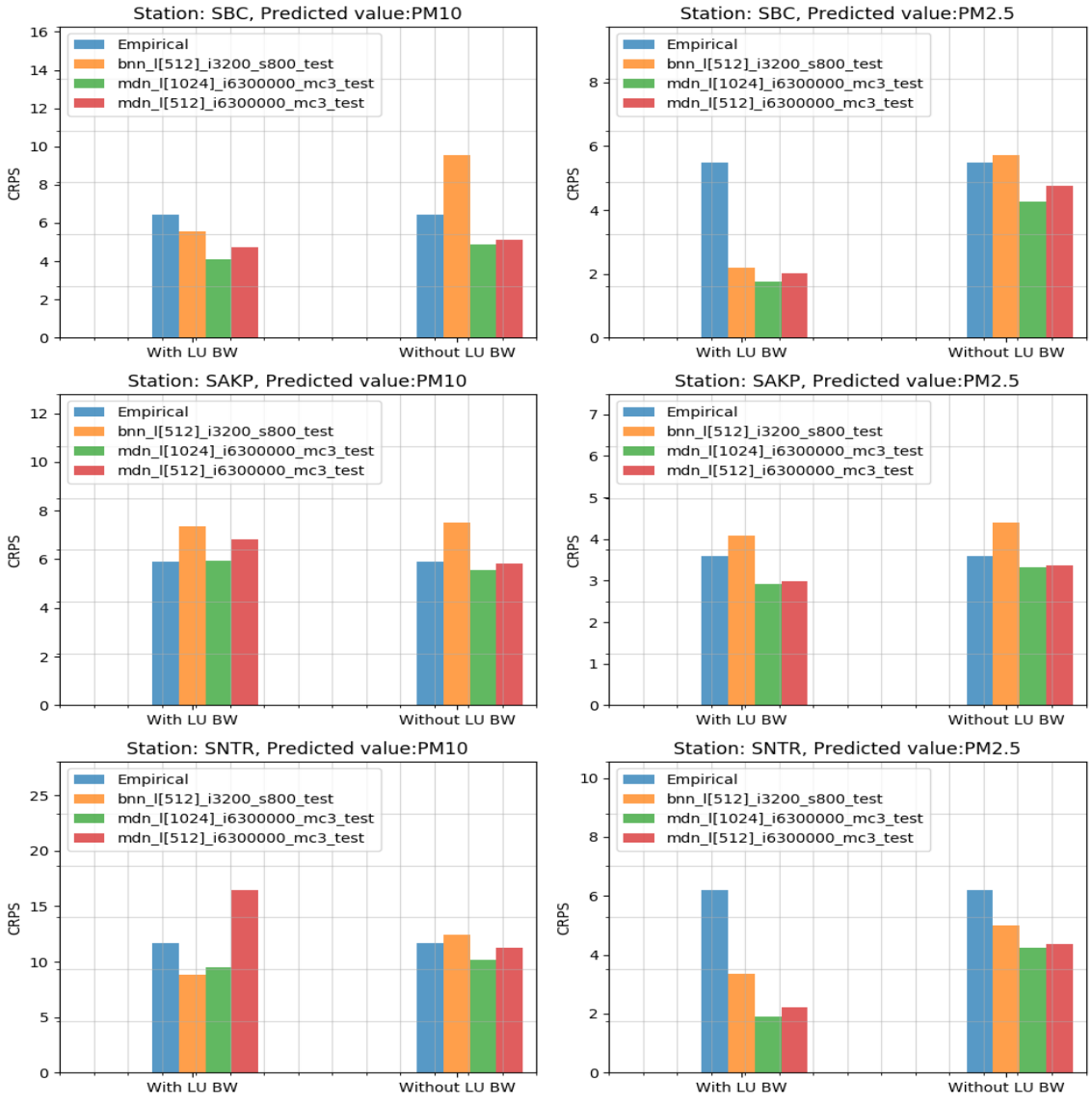


Figure 13: CRPS values of the models (lower is better).

Figure 13 is analogous to Figure 11 but the shown results are from models trained with twelve hourly averaged data. The observed in Figure 13 scores are consistent with the results from the previous section. In the prediction of the air pollution values of SBC we can again see how the BNN is sensitive to the use LUBW-data as features and the performance of the model drops without it in both cases - PM10 and PM2.5. What is interesting is that the both MDN models manage to remain better than the empirical model even without using values from LUBW-sensors. This suggest that the increasing the spatial resolution of the measurements from daily average to twelve hour average does not seem to have notable negative effect on the performance of the MDN models. We can even say that the performance is better as both of the models were worse than the empirical without using LUBW-data when they were trained with daily averaged data.

The tend is also noticeable in the prediction of SNTR values. There almost all of the models remain better than the empirical one regardless in all of the cases. To note is that one of the MDN models (the one given in red) has worse score than the empirical model in the case of predicting PM10 with the use of LUBW-data. We attribute that to poor starting training conditions of the model as its score is clearly not in the general trend of the results.

We again consider the scores for the models predicting SAKP values as unreliable. We see that in this case the scores of the models hardly change. This is consistent with the part of the results in the previous section.

Looking at the predictive performance checks (see Appendix E) of the models, we can see the same trend as suggested by the CRPS scores. The checks of the models trained with twelve hourly averaged data are statistically significant much more often than the ones trained with daily averaged data. This again leads us to believe that the increase of the time resolution of the averaged data has positive effect on the predictive performance of the models. The MDN with **1024** neurons in its hidden layer is the best model in the majority of the cases.

The rank histograms (see Appendix G) of the models continue to be U-shaped. We can, however, see that in the case of MDNs there is also noticeable concentration of values in the middle of the histogram. This means the models often succeed in generating a distribution encompassing the actual observed value.

The problem with overfitting the data is again suggested by the plot with the predictions of the BNN model in Appendix F. We can clearly see how the train data is predicted quite well but the one in the test set is merely approximated. At the start of the thesis we assumed that the considered types of models are robust to overfitting but the observed results suggest otherwise.

5.2.3. One hour period

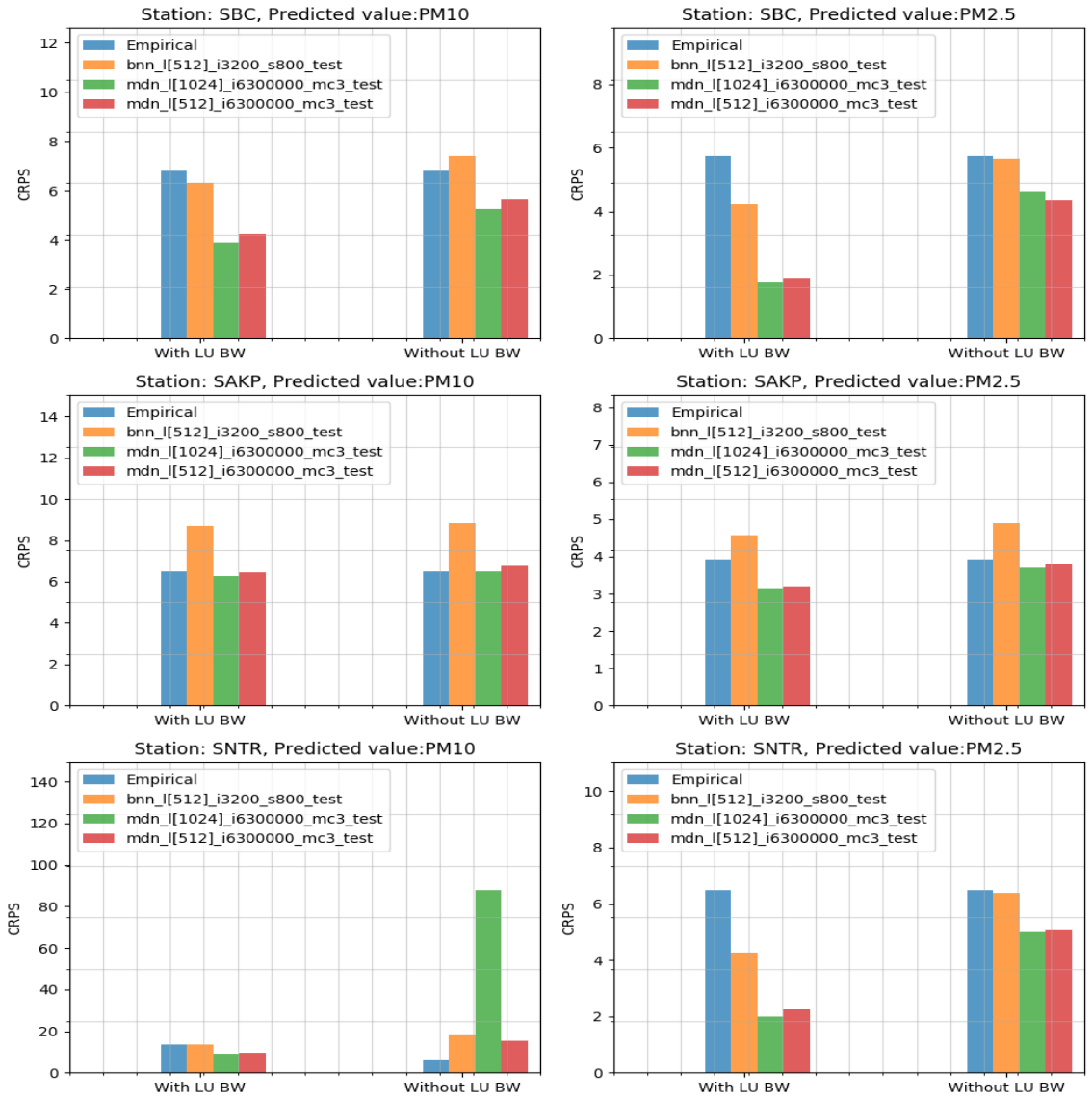


Figure 14: CRPS values of the models (lower is better).

Figure 14 presents the plots of the scores of the models trained with hourly averaged data. In it almost all of the cases are consistent with the results and trends from the previous sections.2

The scores of the models predicting SAKP again have little to no change and we consider them as unreliable. The sensitivity of the BNN models to the use of LUBW-data remains noticeable. We note however that the BNN models is better than the empirical in the case of predicting PM2.5 of SBC. This was not the case with the twelve hourly averaged data. This again suggests that increasing the time resolution does not make the models worse but rather it has positive effect on their CRPS scores relative to the empirical model.

In the case of predicting PM2.5 of SNTR, the scores of all of the models remain better than the empirical. This is, however, not true for the PM10 value. Even though the PM2.5 and PM10 values are highly correlated (see Appendix A) the models seems to have difference

performance while predicting them. We assume this is due to the fact that PM10 varies in bigger intervals than PM2.5. This can be seen in the STD[6]-plots in Appendix B.

The predictive performance checks for this averaging period are even more statistically significant than the ones for the twelve hour average. We can attribute this only to the use of data with higher time resolution. The MDN with **1024** neurons in its hidden layers is yet again the best model in almost all of the cases while looking at the statistically significant test metrics.

The general look of the rank histograms of the models for this averaging period is again less than ideal. The histograms again show that the models often fail to include the actual observation in the generated distribution. The rank histograms of the MDNs are, however, slightly better than the ones of the models from the previous sections.

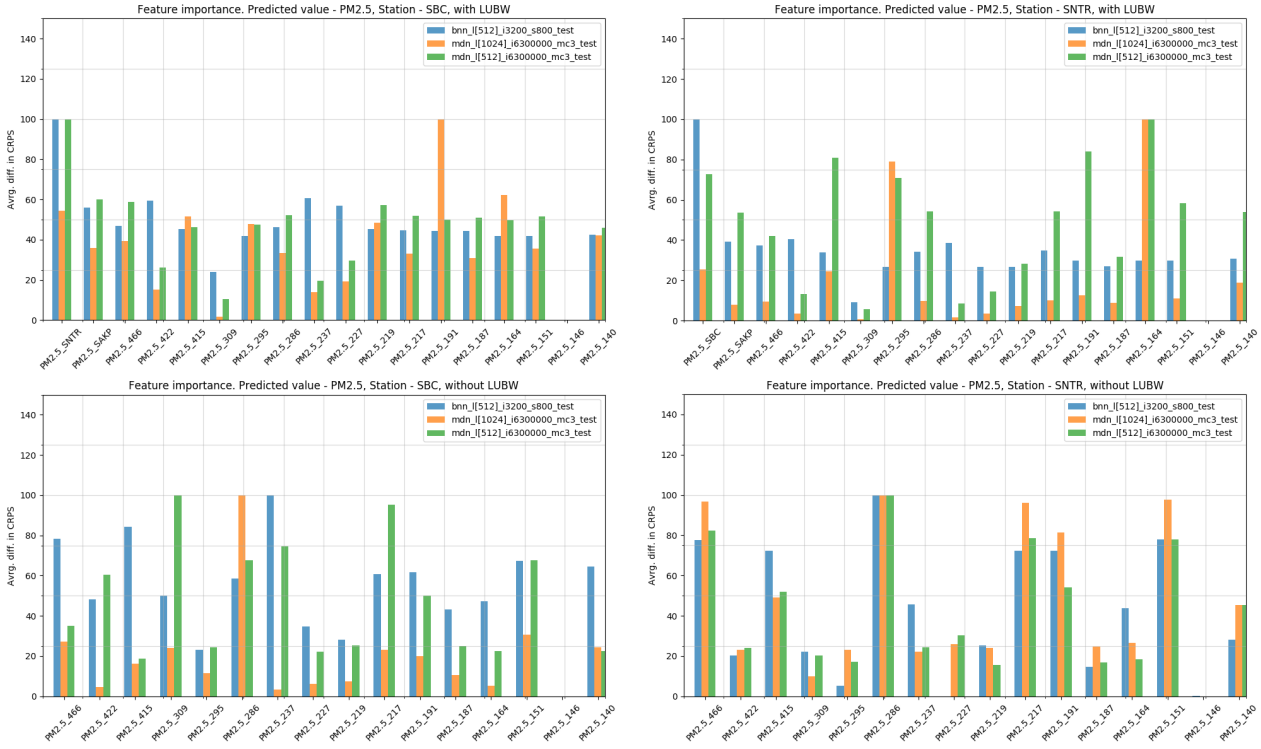
As mentioned in Section 5.2.1, plots with the rest of the scoring rules for each averaging period can be found in Appendix C. All of the considered scores can be found in tabular form with concrete numbers in Appendix D.

5.3. Feature importance

As seen in the previous section, only models trained a certain way achieve acceptable results. For this reason, we only looked at the feature importance results only of selected models. We again make the differentiation based on averaging period and we look at the results from each in three separate sections. We consider the feature importance data only with respect to the CRPS and for models that predict the PM2.5 values of SBC and SNTR.

[6]Standard deviation

5.3.1. One day period



(a) Models predicting PM2.5 of the SBC LUBW-sensor.

(b) Models predicting PM2.5 of the SNTR LUBW-sensor.

Figure 15: The plots on the top show the feature importance results from models that use LUBW-data as features while those on the bottom are from models that do not.

Figure 15 illustrates the considered feature importance results. Looking at the plots we can make several notes about the importance of the different sensors.

A key observation to be made is that the sensor with ID `146` is consistently regarded as irrelevant. For each of the models and in every context this sensors seems to carry no information that can be used for the prediction of the given observations. Another sensors that has little importance in the models that use LUBW-data as feature is `309`.

If LUBW-data is used for the predictions, in both cases of predicting SBC and SNTR, the corresponding other LUBW-sensor[7] has maximum importance over the other sensors. This is to be expected as we consider the LUBW sensors as ground truth. It is interesting to note that the importance of the SAKP sensors is non-trivial and it is not much lower than the importance of the other sensors. We assume this is due to some random correlation between the interpolated data of SAKP and the predicted value. In this sense to think that this high importance of the SAKP sensor is accidental and does note represent reality.

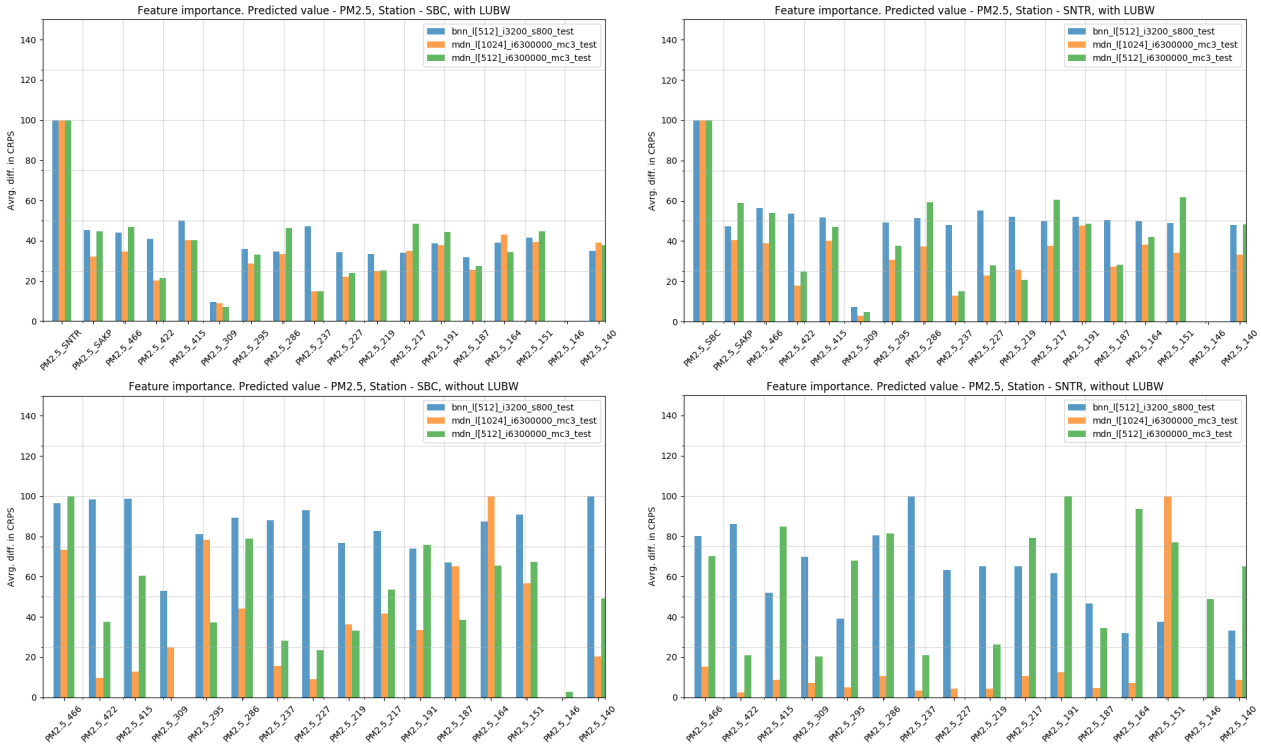
If we focus on models not using LUBW-data we see how the sensor `286` stands out and has consistently high importance for two of the models. This appears to be true for both case of predicting SBC and SNTR. If the importance of this sensors is also high for the other two

[7]SNTR is case of predicting SBC and SBC in case of predicting SNTR

averaging period we would be able to conclude that it carries some relevant for the prediction information.

Another interesting point is how the BNN and one of the MDNs (the one shown in green) consistently have similar sensors with high or low importance. This can be seen as the green and the blue bars often go up and down together. This shows how two different models extract information from almost the same sensors in order to make their predictions. The MDN shown in orange exhibits slightly more irregular behavior but it also follows the general trend of which sensors are important and which are not. This is, however, not the case while predicting PM2.5 of SBC without the use of LUBW-data. There the MDN uses the sensor with id 286 almost exclusively. If we look at the performance of the MDN in Section 5.2.1, we see that in the corresponding case, it is better than the empirical model. This means that this MDN can be better than the baseline even when the majority of the information for the predictions comes from a single non-LUBW sensor.

5.3.2. Twelve hour period



(a) Models predicting PM2.5 of the SBC LUBW-sensor.

(b) Models predicting PM2.5 of the SNTR LUBW-sensor.

Figure 16: The plots on the top show the feature importance results from models that use LUBW-data as features while those on the bottom are from models that do not.

Figure 16 presents plots with feature importance data from models trained with hourly averaged data. We can make almost the same observations s in the previous section but there are some differences.

We notice again how the 146 sensor has very little importance for almost all of the

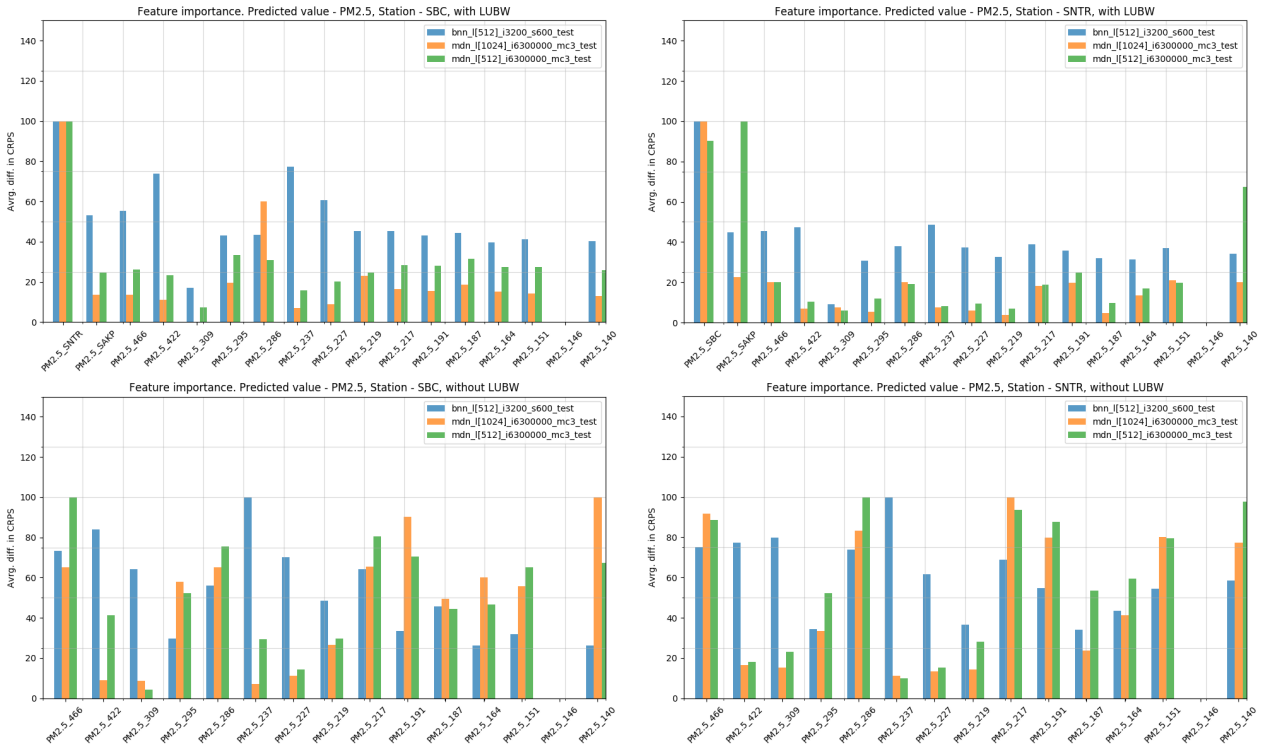
models except one in the case of predicting PM2.5 of SNTR without using LUBW-data as feature. The importance of the sensor with ID `309` also consistent with the results from the previous section.

If we look only at the feature importance plots of the models that use LUBW-data as feature, we see how the sensors tend to have similar importance for every model. This suggests that three different models extract similar connections between the predicted value and the predictors. In the previous section this was obvious only for two of the models. We assume that the increased time resolution of the averaged data allows the models to extract deeper relationships between the value to be predicted and the features.

We no focus on the models on the bottom plots. Here we again observe how the sensor `286` has high importance for two of the models – the ones shown in blue and green. This was also the case in the previous section.

The MDN shown in orange has similar behavior where it almost uses a single sensor for its predictions. This time, however, this happens in the case of preciting PM2.5 of SNTR without the use LUBW-data. There we can see how the sensor with id `151` has very high importance while the other sensors have very low one. Based on this and on the observation from the previous section, we assume that this MDN model is prone to extract information from a single feature and use it for the predictions.

5.3.3. One hour period



(a) Models predicting PM2.5 of the SBC LUBW-sensor.

(b) Models predicting PM2.5 of the SNTR LUBW-sensor.

Figure 17: The plots on the top show the feature importance results from models that use LUBW-data as features while those on the bottom are from models that do not.

Lastly we look at the feature importance results from the models trained on hourly averaged data. Plots of these results are given in Figure 17. The observations that we can made here are essentially the same as the one in the previous sections.

The SBC and SNTR sensors have again high importance. As seen in the previous results, the senor `146` caries no information for all of the models in all of the contexts and the sensor `309` has a relatively low importance when used together with LUBW-data as features.

A surprising result to see is that for the MDN shown in green SAKP has highest importance over all features. This is again probably due to a accentual correlation between the interpolated data of SAKP and the predicted value.

In the case of this averaging period, the sensor `286` has high importance with respect to all of the models. In the previous sections this was true only with respect to the one of the MDNs and the BNN. With the averaging of maximal time resolution, however, it appears that every model can extract some meaningful information from this sensor.

We have clearly seen how across all averaging periods the feature importance results are consistent with one another. Based on this we can conclude that we have successfully identified couple of weak sensors in the network – those with ides `146` and `309` – as well as sensors that appear to have higher value than the others – obviously SNTR and SBC but also probably `151` and `286` given the fact that one of the MDNs can produce reliable predictions almost purely based on one of the sensors. We have also followed the importance of the sensor with id `286` and seen that it consistently has high importance.

6. Implementation

In order to enforce separation between the code and all of the data that the code acts upon, we place all of the generated data in the `env` directory of the project. This directory is added to the `.gitignore` file and the implantation in no way relies on its contents. The contents of the folder are fully generated by the implemented scripts. In the next sections we explain exactly what the generated content is and how it is generated.

6.1. Used libraries

In order to make the data preprocessing, the implementation of the models and their evaluation easier, we use several standard machine learning libraries. For efficient array manipulation we use the popular *python*[8] library *numpy*, described in [Oli15]. The higher level data processing is done through *pandas* [McK10] and for the plotting of the results we employ the use of *matplotlib* [Hun07]. When it comes to the actual machine learning we use a lot of helper methods from *scipy* [JOP+] and *scikit-learn* [PVG+11]. The core library for the implementation of the models is *Tensorflow* [AAB+15]. For the easier handling of the implementation of MDNs and BNNs we use respectively two other libraries built on top of Tensorflow – GPFlow [MvN+17] and Edward [TKD+16].

6.2. Preparing the data

In Section 1.3 we mentioned that the used public data is less than ideal. We have also outlined some of the problems that we found in the data. Now we explain how we deal with these problems. We have written several python scripts dealing with the preprocessing of the data.

The raw data files are located under [luf]. They are organized in folders by day. The folder for a given day contains different files from different sensor types. We are only interested in the air pollution data form the *sds011* sensors. Each data file for this type of sensor is a text file with *comma separated values* (*CSV*) where each row has measurements for the PM2.5 and PM10 value (see Section 1.3), the location of the sensor, the time of day when the measurement was taken as well as the sensor-id. Our final goal in the data preprocessing in to generate a *DataFrame*[9] where each column is labeled with sensor-id and either P1 (for PM2.5) or P2 (for PM10) and each row contains measurements of the corresponding values averaged over some periods of time. The periods of time in which we integrate[10] must be synchronized between all sensors. At the end we generate three different DataFrames from the data – with averaging over whole days, over twelve hour periods and over one hour periods.

We try to automate everything from downloading the data, extracting the relevant pieces of it, sanitizing it and finally generating a single DataFrame object. As the data preprocessing is not the focus of this thesis, we do not explain each script in absolute detail but rather what happens in it.

[8] Programming language.
[9] A *padnas*-object for easy data storage
[10] We use integrate and average interchangeably.

All of the scripts we explain in this section require a configuration file. This file is `config.json`. In it we define certain properties for the scripts so that we can easily change them if necessary. In each of the data preprocessing scripts there is a `_read_config` method that loads the relevant for the script properties from the configuration file. In the next paragraphs we use property and field interchangeably. We also sometimes obey to say that the properties are in the configuration file and we imply it implicitly.

6.2.1. Download module

The amount of files under [luf] is too high for manual downloading. Therefore the first step of the process was to write a script that downloads the relevant files. This is the functionality provided by the `download_module.py` file. In the configuration file there is a section with properties specifically for this script. Through those we can specify:

- `base_url` – the base link where the files are located. In our case `https://archive.luftdaten.info/`
- `start_date`, `end_date` – those specify the considered period of time. Only data files from this period will be downloaded. We've download files for the whole year of 2017.
- `sensor_type` – this fields specifies the type of information we want to download. We are dealing with air pollution and the considered sensor type is sds011. See Section 1.3 for more details.
- `files_list_file` – optional field specifying a file where the links and the filenames of all downloaded files can be saved.
- `list_files` – a boolean value showing weather or not the script should save the links of the downloaded files as a list in the file given in `files_list_file`

The script downloads the desired files in the directory specified by the `raw_down_dir` property of the configuration file. Internally the script uses python bindings for *wget* (see [wge]) in order to preform the actual downloading from the Internet. Another library that comes into use is *BeautifulSoup* (see [bs]). With it we parse the *HTML* of the web pages and extract the relevant the URLs.

6.2.2. Preprocess module

Central part of the preprocessing pipeline is the script in the `preprocess_module.py` file. It aims to select only the data files from the "good" sensors, from those then further filter the data and then for each sensor that is reliable enough, to create a DataFrame in form of a CSV-file. The final CSV-files will be stored in the directory specified by the `data_files_dir` property in the configuration file. The `preprocess_module` script first generates a file with the ids of the sensors that are actually worth processing. In this first phase we perform two checks while iterating over all of the downloaded files:

1. For how many days a given sensor provides measurements. If a certain threshold is passed – specified by the `min_sensor_cnt` property in the configuration file – the examined sensor

is considered *saturated*.

2. Is a given sensor "near" to Stuttgart. In the configuration file we specify a center through the property `center` and a radius through the property `radius`. We consider only the sensors falling within the defined circle.

If a given sensors passes both of those checks, its name is added to the file specified by the `good_sensors_list_file` property. Furthermore, every data file of these "good" sensors is added to a file specified by the `good_sensors_data_files_list` property so that we can later consider only those.

The next step is to process the individual raw files and store their information in a proper form in CSV-files. For each of the raw files listed in the file specified by `good_sensors_data_files_l` we perform the following steps:

- We load the raw data file in memory in the form of a pandas DataFrame.
- We transform the time column in more convenient format and then perform a check to see whether there are any duplicates in the DataFrame with respect to the time of the measurement. If there are, we collapse them to a single entry using the strategy specified by `duplicates_resolution` in the configuration. This can be *MEAN*, *MEDIAN*, *MAX* or *MIN*.
- We then perform the integration over the desired time period specified by `day_integration_period` in the configuration. To note is that we are still working on per file basis and the processed DataFrames contain a day worth of measurements.
- We rename the columns of the DataFrame to reflect the id of the corresponding sensor. The columns with the PM2.5 and PM10 measurements are named `P1_<id>` or `P2_<id>` respectively, where `<id>` it the id of the sensor to which the data file belongs.
- We make one final check in this script. After the values in the DataFrames have been integrated over specific time intervals, there are still can be missing values. If the count of these missing values is higher than `missing_data_cnt_threshold` in the configuration, the name of the sensor is added to a list in the file specified by `bad_missing_data_sensors` property. If the missing values are below the mentioned threshold, then they are interpolated with the strategy specified by the `missing_data_resolution` property. This can either be *MEAN* and the missing values will be filled with the average of the rest of the values or *linear* and the missing values will be linearly interpolated based on the rest of the data.
- After all of that we append the data of the resulting DataFrame to a CSV-file with name `end_data_frame_<id>` in the directory specified by the `data_files_dir` property.

We do this for every relevant raw data file and at the end we have a CSV-file for each sensor with accumulated data from the DataFrames. These CSV-files now contain a year worth of measurements. The final step is to go over these files one more time and sort the data by time.

6.2.3. Description module

The goal of this script is to provide some information about the generated data in from of plots and several DataFrames with statistics as well as to perform one final sanitation on the data. The script is implemented in the file `description_module.py`. It iterates over the generated from the previous step DataFrames and counts the missing values. Its important to note that in this step the missing values are in the context of the whole year we are considering. Depending on the integration intervals there must be exactly certain amount of values in each DataFrame. For integration over a day the exact count is **356**, for integration over **12** hours – **730**, for integration over each hour in the year – **8760** (**365** days with **24** hours per day). If the number of the missing values in the corresponding context are grater than the value of the `missing_entries_threshold` property then the name of the sensor is added to a list in the file specified by `reindexed_frames_file`. If, however, the amount of missing values is under this threshold, the missing values are filled with the average value of the rest of the data. This is done for both the `P1` and the `P2` column. As this is also a data description script, we generate a DataFrame with different statistics about the P1 and P2 values for each sensor. We wanted to know exactly how this last preprocessing step on the data changes it. The DataFrames with the statistics are therefore generated two times – once before the filling of the missing values and one time after the fact. We can then compare exactly what has changed about the data. Compression plots can be found in Appendix B. At the end the scripts also generates plots of the P1 and P2 values of each sensor. Two kinds of plots are generated. One with the pure values as they appear in the final DataFrame. The other plot shows a moving average of 100 consecutive values. All of those description files are saved in the directory specified by the `description_files_dir` property in the configuration.

6.2.4. Preprocess LU BW

Although the data provided from LU-BW is in much better form than the publicly available data, it also needs certain preprocessing. The LU-BW data comes in a form of several spreadsheets in a *xlsx*-file. The script in `process_lu_bw.py` takes this file as an input and generates three DataFrames for each LU-BW station. The measurements in the spreadsheets are on half an hour basis so an integration in the desired intervals is required. The script performs this integration and fills all of the missing values with the mean of the rest of the data. Similar to the previous section, a description file with statistics about the data before and after the missing values filling is generated as well as plots with the integrated values and a moving average of them. The DataFrames are saved in CSV-files in a separate folder in the folder with the rest of the data files. The plots and the description file are saved in the directory with the description files.

6.2.5. Loader module

The final step is to put the individual DataFrames of the sensors in one big DataFrame that can be conveniently used for the training of the models. This is what the script in `loader_module.py` performs. It gathers all the DataFrames defined by the CSV-files in the folder given by `folder` property and appends their columns to the columns to the final big DataFrame. The LU-BW DataFrames are also appended. The final DataFrame is saved in

the `env` directory with name specified by the `final_df_name` property. In the section for this script of the configuration file there is a property with name `ignored_sensors_files`. This is meant to be a list of files, each containing a list of sensor ids. The sensors given in these files will be ignored and won't be loaded in the final DataFrame.

6.3. Models implementations

In the first three sections we explain the classes we've implemented for the two considered models. In the next two sections we describe the scripts that we use for training and evaluating the models. The major libraries we use are build on top of Tensorflow. In our explanations we assume basic knowledge of the working principles and structures behind Tensorflow. A good introduction can be found in [tfn].

6.3.1. Mixture density networks

For the implementation of the MDNs we follow the guidance of [Dut]. We use the *GPFlow* library that internally uses Tensorflow. The library simplifies the creation of Gaussian process models. Although we don't unitize it fully, the features it provides make it easy to define a MDN.

The code implementing the MDN is in the `mdn_model.py` file. In it we define a class – `Mdn` – representing a mixture density network. As previously stated, we follow the approach of [Dut]. Our `Mdn` class inherits from the `Model` class in the `gpflow.models.model` package. This base class makes it possible to integrate our newly defined model in the GPFlow library. In order to explain the structure and the functionality of our class we take a bottom-up approach. We first describe the individual small parts and then how they come to work together.

In the constructor of the class we define the basic attributes of the models that we'll need in the other functions. We first go over the arguments of the constructor. The constructor takes:

- An arbitrary name for the new model in form of a string in the argument `model_id`
- The training data for the model `X` are the input feature vectors (as explained in Section 4.3) and `Y` is the output.
- The definitions of the inner layers of the network. This is nothing more than an array of numbers specifying how many neurons there are in each of the hidden layers. The number of hidden layers is given by the length of this array. In the constructor this is the `inner_dims` argument.
- Activation function that will be used as non-linearity in the neural network (see Section 3.2.1), by default this is the **tanh** function. This is given through the argument `activation`.
- The number of mixture components that the MDN will model. This is a simple integer value in the `num_mixtures` argument with default value of **5**.

- Optional file name of a file from which the class is supposed to load the model.

In the constructor itself we save the provided information in class attributes.

After defining everything we either load the model from a file or construct it with the `_create_network` method. The loading of a model from file is discussed in a later paragraph.

The `_create_network` defines the weight matrices and the bias vectors of the neural network for the MDN. The appropriate dimensions are known thanks to the `dims` array. The weight matrices are assigned random values and we impose restriction on the bias vectors to be positive in order introduce some regularization in the network. This is suggested by [Dut]. The use of similar technique can be found in [CZ15]. All of the network's parameters are in the form of `Param` objects of GPFlow. These encapsulate raw vector variables from Tensorflow that later can be used as model parameters and are managed by GPFlow. The generated parameters are put into class attributes for later use.

we can now describe the `_eval_network` method. In it we iterate over the weight matrices and biases and construct a Tensorflow graph defining the cascade of matrix multiplication, bias addition and non-linear function application as described in Section 3.2.1. The method takes one argument. This is the input to the network. At the end of the matrix multiplications, we split the intermediate output in three parts – mixing coefficients, means and variances. We then place the described in Section 3.3 restrictions on them. The end result of the method is a Tensorflow operation that when evaluated with some input, produces the output of the whole MDN as described in Section 3.3. To note is that the `_eval_network` is decorated with the `@params_as_tensors` decorator which transforms the arguments of the method in tensors automatically on invocation of the method.

The `_eval_network` method is used by `eval_network` to actually provide output of the MDN to the user of the class. `eval_network` is defined with `@autoflow` decorator which automatically evaluates the returned Tensorflow operations.

The most important method to be implemented so that GPFlow can train the model is `_build_likelihood`. It essentially defines the loss function to be optimized. In the method we first evaluate the network with the given input in order to generate the mixing coefficients, means and variances. Next we use these in order to build the sum over the log-likelihoods for each example as described in Section 3.3 and given in Equation 3.36 and Equation 3.37. This is all what we need to do as the automatic differentiation engine of Tensorflow can perform the gradient calculation for any arbitrary operation defined in the operation graph.

The `fit` method is the place where we do the actual training of the model. We use the `ScipyOptimizer` class provided by GPFlow as a wrapper of a *scipy* method for optimizing mathematical functions. This *scipy* method uses the LM-BFGS algorithm (see Section 3.2.1). The `fit` method takes the maximum number of iterations and a optional callback function. We perform the training in three steps. First we optimize the function while we allow changes only in the weight matrices, then we optimize only by changing the bias vectors and finally we optimize with changes in both of the parameter groups. This is again suggested by [Dut]. After each step we invoke the callback if provided, so that the outside user can monitor the changes in network's performance after each step.

We also implement a `save` method that can make a trained model persistent by saving in storage memory. The method uses a `Saver` object provided by GPFlow that can serialize

parameters and parameter lists. In order to fully reconstruct the model later we only need the weight matrices and bias vectors. For this reason these are the only objects that we serialize and save.

The later loading of a given model is done through the `load` method. Its purpose is to load the weight matrices and bias vectors from a given file through the deserialization provided by the same `Saver` object used for saving. The `load` method essentially does the same thing as the `_create_network` method but it loads the weight matrices and bias vectors from a file. The `load` method is in fact invoked instead of `_create_network` by the constructor, if a file name is provided.

6.3.2. Bayesian neural networks

With the implementation of BNNs we again do not reinvent the wheel and use the *Edward* library. Edward is another library built on top of Tensorflow and it provides a lot of functionality that makes Bayesian learning easier to implement. We follow the official guide to BNNs in [Tra] from the authors of Edward. The class implementing BNNs is `Bnn` and it is located in the `bnn_model.py` file. We now look at the individual functions of the class in order to explain it.

The construction is trivial as the only thing it does is save the given name of the model in an class attribute.

First we will look at the methods `generate_prior_vars` and `generate_latent_vars`. These methods are very similar. For given input and output dimensions for the whole network and layer definitions (analogous to Section 6.3.1) they both define weight matrices and bias vectors as Normal distributions (i.e. we can later sample wight matrices and bias vectors from the defined objects) with the appropriate dimensions. The difference between the methods is the context in which they define the distributions. `generate_prior_vars` defines the prior distributions over the wights and biases. These are all standard normal distributions and correspond to the $\boldsymbol{p}$ distributions in Section 3.2.3. `generate_latent_vars` defines distributions with trainable Tensorflow variables that later can be optimized. These distributions build the variational distribution that we will perform inference on and will try to approach the posterior. For details on variational inference see Section 3.2.3. The individual variational variables are assigned random values at the start.

The `_neural_network` method serves almost the same function as the `_eval_network` from the `Mdn` class. For a given input and lists of weight matrices and bias vectors it creates a Tensorflow operation that evaluates to the output of the network defined by the weights and biases and with an input the given input. The non-linearity function that we use is **tanh**.

With all these methods explained we can now look at the central part of the `Bnn` class. The `build` method is what builds the whole Tensorflow graph for the structure of the BNN and the operations we will need for the inference. First off, with the usage of `generate_prior_vars` and `generate_latent_vars`, we generate the prior and latent distributions that we will need for the network. Then we define a tensor placeholder for the input data, operation that defines a normal distribution centered at the network's output and a small variance (see Section 3.1) and a placeholder for the actual observed output data. To note is that here we use the prior distributions while we define the network operation. The next step is to define the operation

necessary for the evaluation of the BNN with new data. For that we define two placeholders – one for the input with which we want to evaluate the trained network and one for the number of samples we want to draw from the posterior distribution (i.e. how many network functions we will generate and evaluate, see Section 3.2.2). After that we define an operation that evaluates to stacked evaluations of networks defined with draws from the latent variables defining the variational distribution. To note is that we evaluate each network on the whole input. The final shape of the output of this operation will be $\boldsymbol{N} \times \boldsymbol{s}$ where $\boldsymbol{N}$ is the count of provided examples to evaluate and $\boldsymbol{s}$ is the number of samples we are supposed to draw from the BNN. The final step in the `build` method is to initialize all the variables in the Tensorflow graph with execution of initialization operation and define a `Saver` object so that later we can later save the trained model to storage memory.

The training of the model happens in the `fit` method. In it the Bayesian inference functionality of Edward comes into play. We first define a dictionary representing a mapping between prior variable and its variational counterpart. This is the format required by Edward. We then define the type of the inference we will use for the training of the model. In our case – Kullback-Leibler divergence (see Section 3.2.3). Edward provides convenient interface for the definition through the class `ed.KLqp`. Once the inference is defined and initialized, we start looping over the data and we feed certain amount of examples to the `update` method of the defined inference object. Every one thousand iterations over the whole data we invoke a callback, if such is provided on calling the `fit` method. This way the outside user can monitor the models during the inference process.

With the `evaluate` methods we can evaluate data with a trained model. The method accepts the input data and number of samples to be drawn for each data point, feeds them to the right placeholders in the Tensorflow graph and evaluates the Tensorflow operation mentioned in one of the previous paragraphs.

The `save` and `load` methods of the class are similar to the ones in the `Mdn` class. One key difference is that here we use the `Saver` object from Tensorflow and not from GPFlow. This means that by serialization we are saving the whole Tensorflow graph and the values of all variables in it. By the loading of the model later, we only need to load the graph and not build anything at all (i.e. not call the `build` method).

6.3.3. Empirical model

The implementation of the empirical model is the simplest one. The class implemented in `empirical_model.py` is a simple wrapper of the Edward class `edward.models.Empirical`. This is a helper class in Edward representing a random variable that can be constructed with a given set of samples. One can then sample new examples from the defined random variable. The implemented by us class – `Emp` – does exactly this. In the `build` method we first construct the random variable and through the use of the `evaluate` method we can later draw certain amount of samples from it.

6.4. Model training

We now explain the scripts that bring together the generated data and the defined models. The code constructing and training the models is in `model_evaluation.py`. The script takes various arguments that configure the type of data that gets used for training, the structure of the models themselves, which model gets trained and where are the results should stored. After training each models gets evaluated. The evaluation is explained in Section 6.5. In this section we explain how the exactly we set up the configuration of the training. We first go over the command line arguments of the script. These are:

- `--model` – the type of the model the script will train. The value of the argument can be *bnn* for BNN, *mdn* for MDNs or *both* for training first BNN and then MDN.
- `--config` – configuration file for the both models. We explain the settings in it in a later paragraph.
- `--station` – the LUBW station whose values the trained model will try to predict. The value of the argument can be *SAKP*, *SBC* or *SNTR*.
- `--predictor` – the air pollution value(s) that should be used as predictor. Can be *P1*, *P2* or *P1P2*. The last one specifies the using of P1 as well as of P2 as predictors.
- `--period` – the averaging period used to generate the dataset. Can be *1D* (daily average), *12H* (twelve hour average) or *1H* (1 hour average).
- `--outvalue` – predicted air pollution value of the model. Can be either *P1* or *P2*.
- `--take_lubw` – a boolean flag whether or the other two LUBW sensors should be used as predictors.
- `--dest` – a folder where the results of the training and from the models' evaluation should be stored.
- `--base-dir` – a folder where the final DataFrames from the three datasets are stored.
- `--load-mdn` – this argument is an optional directory with a stored MDN-model (see Section 6.3.1). If set, the MDN model will be loaded from there and a new one won't be trained.
- `--load-bnn` – this argument is an optional directory with a stored BNN-model (see Section 6.3.2). If set, the BNN model will be loaded from there and a new one won't be trained.

When the script gets called, it extracts the specified by the arguments data and creates a test-train split on it. A description string with all of the settings is created and logged in a log-file. With this we know exactly how the models were trained. The configuration of the models is passed to the script through a *JSON*-file. This configuration is also logged in the log-file. For the BNNs the configuration file contains the definitions of hidden layers and the number of neurons in them, the number of samples drawn when calculating the gradient during optimization as well as the iterations over the entire training set to be performed during the training. The configurations for the MDNs are layer definitions, number of mixture components

and maximum iteration count over the data during training. The script creates the specified by the configuration models and trains them. After that, the models are passed for evaluation. In the next section we explain how the evaluation is performed.

6.5. Models evaluation

In the file `model_evaluation.py` we define a `Evaluator` class that cat evaluate already trained models. The methods that are getting called form outside are `evaluate_mdn`, `evaluate_bnn` and `evaluate_empirical` for generating evaluation information about respectively MDNs, BNNs and empirical models. There are separate methods because the different objects representing different model type generate data in slightly different formats. The methods, however, perform essentially the same actions. First two plots are generated. The plots show the result of applying the model on the train and the test sets and comparing the generated values to the actual observed values. These are the plots like in Figure 9 and Appendix F. After this the rank histograms over the train and the test sets are generated. Rank histograms are explained in Section 4.4 and given in Appendix G. The three proper scoring rules described in Section 4.2 are calculated on the train and test sets and the results are logged in *CSV*-files. To note is that the results are saved as average value over the whole set as well as the individual scores of the individual observations. We later use the latter one to compare the predictive performance of the models based on average score value and the latter one to perform predictive performance checks between different models. Feature importance data based on the evaluated model is also generated in the respective method. This is done again for the test and the train sets. Each column of the considered set is first shuffled and then the scoring rules are calculated again. The difference in scores between the corresponding observations is calculated, summed and averaged exactly as explained in Section 4.3. At the end, for each rule and for each shuffled feature we save the average difference in scoring rules in a *CSV*-file.

6.6. Auxiliary scripts

In this section we mention several script used for automating some of the trivial tasks.

6.6.1. Run Generation pipeline

File: `run_generation_pipeline.sh`

The script performs the whole DataFrame generation with the current `config.json` from start to finish. The following steps are automatically executed:

1. Extracts the integration period `P` for the data from the `config.json` file.
2. Creates a log file with name `LOG_<P>.txt` that will contain the whole output to the standard output of the scripts in the next steps.
3. Runs `preprocess_module.py` with the right flags to generate cache files containing the senors that must be preprocessed.

4. Runs `preprocess_module.py` with the right flags to perform the actual preprocessing of the raw files.

5. Runs `preprocess_lu_bw.py` to preprocess the LU-BW data.

6. Runs `description_module.py` to perform the final checks on the data and generate plots and statistics about it.

7. Counts the lines in each of the generated CSV-files for the data form *luftdaten.info*. All of these counts must be the same if everything is ok.

8. Counts the lines in each of the generated CSV-files for the data form LU-BW. All of these counts must be the same if everything is ok.

9. Runs `loader_module.py` to generate the final big DataFrame with all of the data.

10. Copies the final DataFrame in the `data_frames` folder of the `env` directory.

11. Archives all of the generated files during the whole process including the log file, the `config.json` file, all of the generated data files and all of the generated description files.

The result of running this script is a self-contained archive containing the generated data as well as description of how it was generated. This archive can later be inspected in order to validate the correctness of the data. One can also examine the log file and the configuration file and know exactly what has happened in the generation process.

Some of the steps are placed in separate bash scripts of their own. These scripts are with self expiatory names and we do not go over them in details. The scripts are: `preprocess_new_cache.sh` (step 3.), `preprocess_with_cached.sh` (step 4.), `save_final_dfs.sh` (step 10.), `archive_data_files.sh` (step 11.).

6.6.2. Clean `env`

File: `clean.sh`

The scripts deletes all files generated during the generation process except the final data frame and the created archive. This prepares the `env` directory for a future execution of the preprocessing pipeline.

6.6.3. Training MDN and Training BNN

Files: `training_mdn.sh` and `training_bnn.sh`

The two scripts execute `model_training.py` for each model with all considered combinations of data types as described in Section 5.1. If these two scripts are executed fully, all of the results discussed in this thesis will be generated.

6.6.4. Generating Result Plots

File: `gen_results.py`

The script generates all of the plots with the results of the evaluation of the models as well as the feature importance plots. Furthermore, the LaTeX-tables with numeric results as well as the tables with the predictive performance checks are generated by the script. The tables are written in simple text files. The scripts examines the directories generated by `training_mdn.sh` and `training_bnn.sh`, pull the relevant data from the right files and plots the gathered information in convenient plots. The results of this script can be seen in Appendix C, Appendix D and Appendix E as well as in Section 5.2

A separate bash script – `results.sh` – executes `gen_results.py` three times – once with result data from each averaging period. The shell scripts accepts a single command line argument. The value of the argument should be a directory containing all of the directories generated by `training_mdn.sh` and `training_bnn.sh`.

6.6.5. Data Metrics Changes

File: `data_change_plots.py`

This is a simple python script that generates plots illustrating the change in the statistical metrics of the data before and after the last preprocessing step (see Section 6.2.3). The scripts loads the generated in the preprocessing process DataFrames describing the change in the data and plots the results. The script also examines the generated `env` directory and plots the correlation between the PM10 and PM2.5 values of each sensor. The resulting plots can be seen in Appendix A and Appendix B.

The script get called three times – once for data for each averaging period – form a separated bash script – `gen_data_desc_plots.sh`. This script accepts a single command line argument that should be a directory containing the folders `data_files_1D`, `data_files_12H` and `data_files_1H`. These folders are nothing more than the decompressed archives generated by the `run_generation_pipeline.sh` script.

7. Conclusion

7.1. Summary

In this thesis, we have looked at a concrete use of two probability forecast models – Mixture density networks and Bayesian neural networks. We have considered a heterogeneous network of air pollution sensors in the vicinity of Stuttgart. Through the use of considered models we have tried to predict the air pollution values of a height quality sensors while using data from the rest of the network as a predictor. We aimed to see how reliably such predictions can be made, given the fact that the majority of the sensors in the network are low quality ones. We have also tried to find out which of the sensors from the network have a higher information value for the prediction. This was done through the use of feature importance technique.

In Section 1.1 we have motivated the use of probability forecast models and in Section 1.2 explained their basic principles by differentiating them from classical regression models. In Section 1.3 we have gone over the nature of the used air pollution data and how exactly it was generated by the considered sensor network. The machine learning models were introduced in Section 1.4 with a brief overview for each of them.

In Section 3 we have given detailed look at the theoretical inner workings of the BNNs and MDNs. We have first explained the idea behind Bayesian reasoning and how it can be used to preform regression. We have used the BNNs as example of a Bayesian regression model and we have explained them in detail in Section 3.2. Because BNNs are a type of neural networks we have also given a brief overview of NNs in Section 3.2.1. The training of BNNs was done through a variational inference technique explained in Section 3.2.3. The theory behind MDNs is separately looked at in Section 3.3.

As motioned in a previous paragraph, for the evaluation of the models we have used methods specifically designed to evaluate a probabilistic forecast against an observed value. All of the theory of these methods is explained in Section 4. A central part of the evaluation was the use of proper scoring rules as explained in Section 4.2. The theory behind the used feature importance technique is described in Section 4.3. For the evaluation of the predictive performance of the probabilistic forecasts generated by the models we have also considered rank histograms which are explained in Section 4.4. Direct comparison between two models with respect to their predictive performance was done through the Diebold-Mariano test described in Section 4.5.

In Section 5 we first gave a overview of our approach in the preprocessing of the data and how exactly the models were trained. Then, in Section 5.2 and Section 5.2, we have looked at the results from the different evaluation methods for each differently trained model.

All of the concrete implementations of the models, the data preprocessing scripts and the training and evaluation scripts were gone over in Section 6, where we also mentioned the used libraries.

7.2. Discussion

We have seen that a prediction of air pollution measurements of a high quality sensor under the use of data from cheap sensors is in fact possible. The goal of our work was not to build the best possible model for the job but rather investigate whether the task of predicting high quality sensors with low quality ones is feasible with the considered models – BNNs and MDNs. As seen by the results, we have clearly proven that a reliable predictions can be made even without the use of data from high quality sensors. In our investigation of the problem, we were able to make several important observations about the given task. Based on the results we have seen that the increase in the time resolution of the measurements (i.e. making the averaging period of the data smaller) has no apparent disadvantage. Our assumption is that the models are able to extract more information about the data when it is fine grained. In the evaluating process we also noticed how the MDNs perform better than the BNNs. Even though the both types of models were able to be better than the empirical model in most of the cases, the MDN were managing to keep their advantage over the empirical model even without the use of LUBW data. The BNNs, on the other side, were sensitive to this change in the data. We see this as a key difference between the two types of models.

The other point of this thesis was to pinpoint the weak spots of the sensor network. In the feature importance results we have seen several sensors that were shown to have low information value for the majority of the built models. It is important to note, however, that the feature importance technique cannot make objective statements about the quality of the data coming from a certain sensor. The information value of the considered sensor is always in the context of some built model. We merely observer how the different models give higher information value to different sensors. In this sense, everything is dependent on the models. The identified "weak spots" of the network may not necessarily be bad sensors. They are just not preferred by the models. This implies that some of the features in the data can be dropped and a reliable prediction still can be expected. A more thorough investigation on the differences between the feature importance values of the different sensors can be a topic for further research.

With that being said, our work can used as a basis for further research into the topic of predicting air pollution data with probabilistic regression models. A particular point of interest can be the investigation of models that use only non-LUBW data in order to predict the values of a single or even multiple LUBW-stations. If a reliable model of this type could be built, one would be theoretically able to calibrate a model over a network of low quality sensors with one high quality sensor and then remove the good sensor from the network without losing the measurements of it. The theory here is that the model would have learned to exactly predict the values of the high quality sensor only with the use of data from the other sensors from the network. The high quality sensor can then be used to calibrate another network of sensors. The idea of using different types of models is also worth investigating. In our work we have only looked at BNNs and MDNs and we have seen that the desired predictions were possible. We expect that more sophisticated machine learning models, such as deep convolutions neural networks, can extract even more information from the data coming from the sensors used as predictors. We base this assumption on the fact that the considered models were relatively simple ones but even they were able to show promising results. Another point for further research can be the hyperparameter tuning of the models. To note is that this was not a goal of our work. Therefore, one can theoretically achieve much better results, if an effort to generate good models is made. For example, grid search over the hyperparameter space and splitting the

data in three sets – training, validation and test set – where the test set is used to evaluate the final tuned model only once and be compared with other models, while the validation set is used to tune the hyperparameters of the model. We specifically did not want to play around with the hyper parameters of the models as this would introduce bias into the system and would have causes a ground truth leakage. This was partially because of the scarcity of the data. Thus, training models with more data can also be seen as a possible future prospect. When it comes to the time resolution of the data, an interesting question to be answered is whether it is possible to train a model with hourly averaged data but use is to predict values averaged in even smaller intervals. If this is the case, it would be possible to truly increase the time resolution of the measured data as the LUBW sensors provide data only on half an hour basis. The other sensors, however, provide measurements for each minute. If the data from these is then averaged over fifteen minutes and some reliable model exists, one would be able to predict measurements in time resolution smaller than the actual resolution of the measurements of the high quality sensor.

Appendices

A. Correlation between PM2.5 and PM10

In this appendix we present plots outlining the strong correlation between the PM2.5 and PM10 values of all sensors. We give three separate plots for each averaging period of the data. In the title of each plot we also give the average correlation between the PM2.5 and PM10 values over all the sensors.

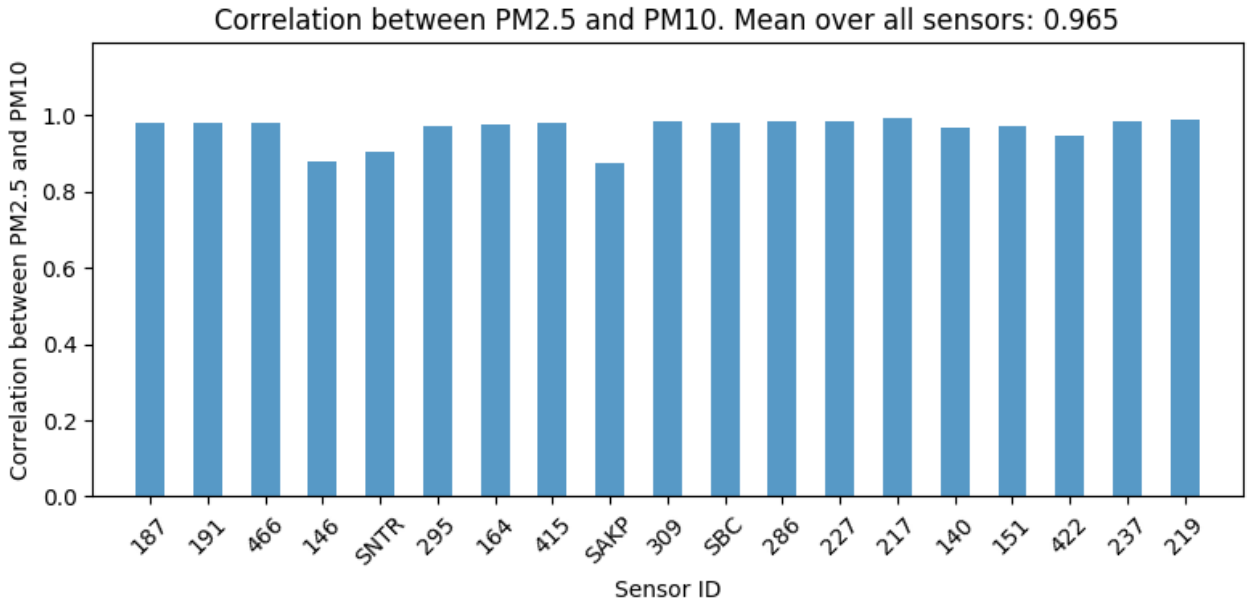


Figure 18: PM2.5-PM10 correlation in the daily averaged data

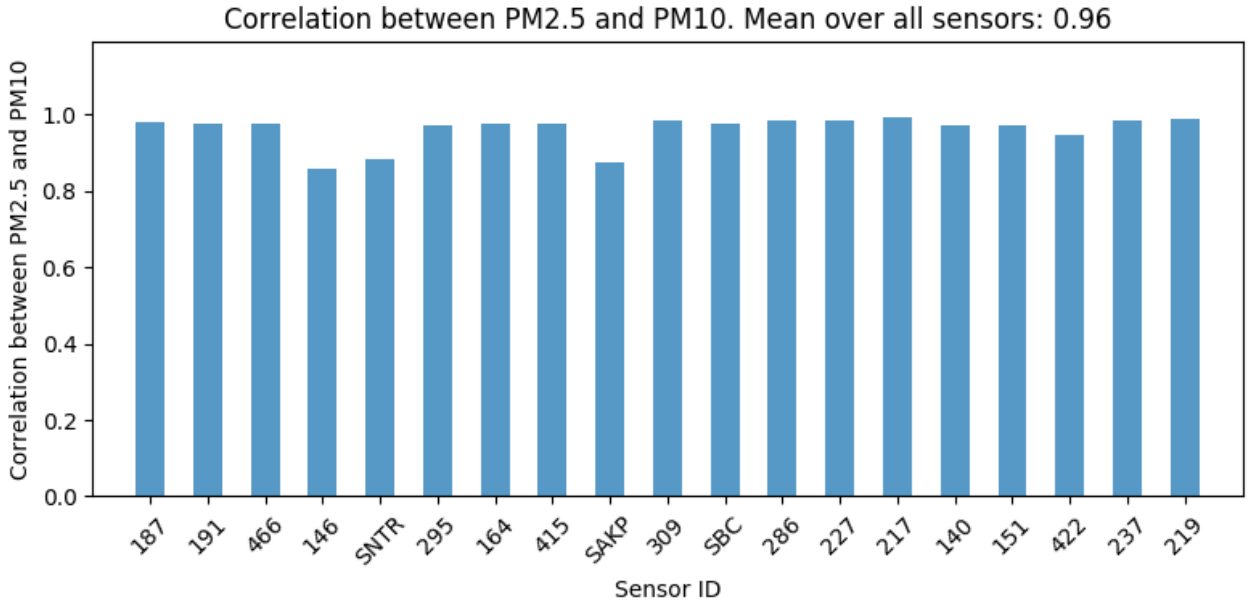


Figure 19: PM2.5-PM10 correlation in the twelve hourly averaged data

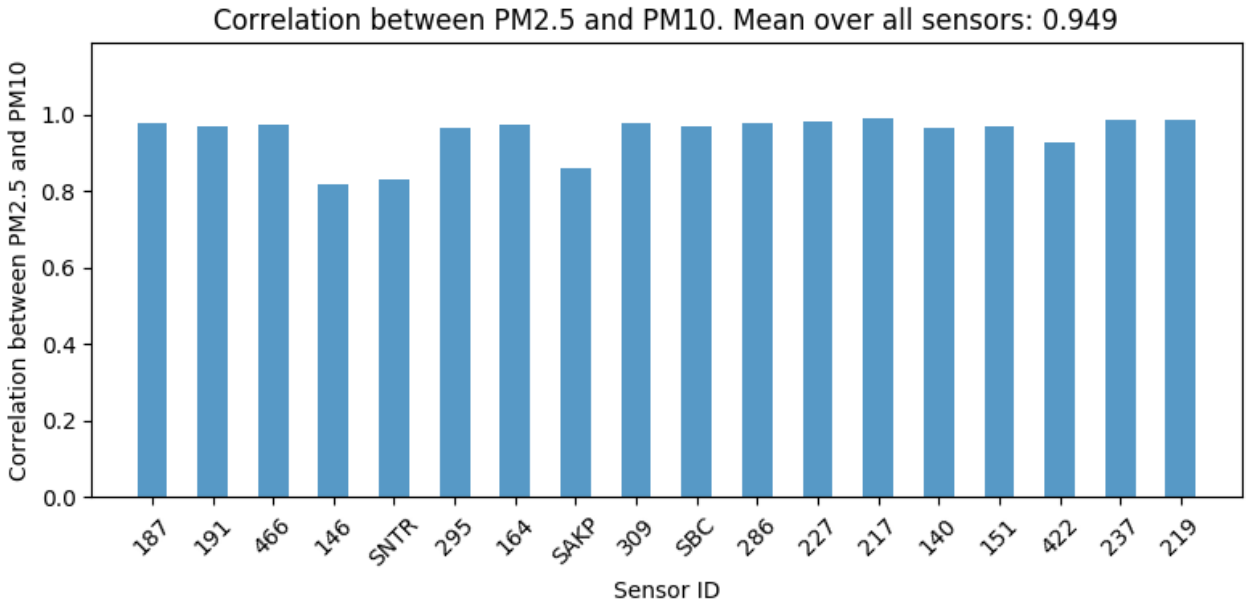


Figure 20: PM2.5-PM10 correlation in the hourly averaged data

B. Data Metrics changes

In this appendix we present plots outlining how the used data was changed during the preprocessing. We compare several statistical metrics of the values of all sensors before and after the applied techniques for missing data filling. The techniques are described in Section 6.2.3. We examine the data from different averaging periods in separate sets of plots. The considered metrics are minimum value, maximum value, mean value and standard deviation.

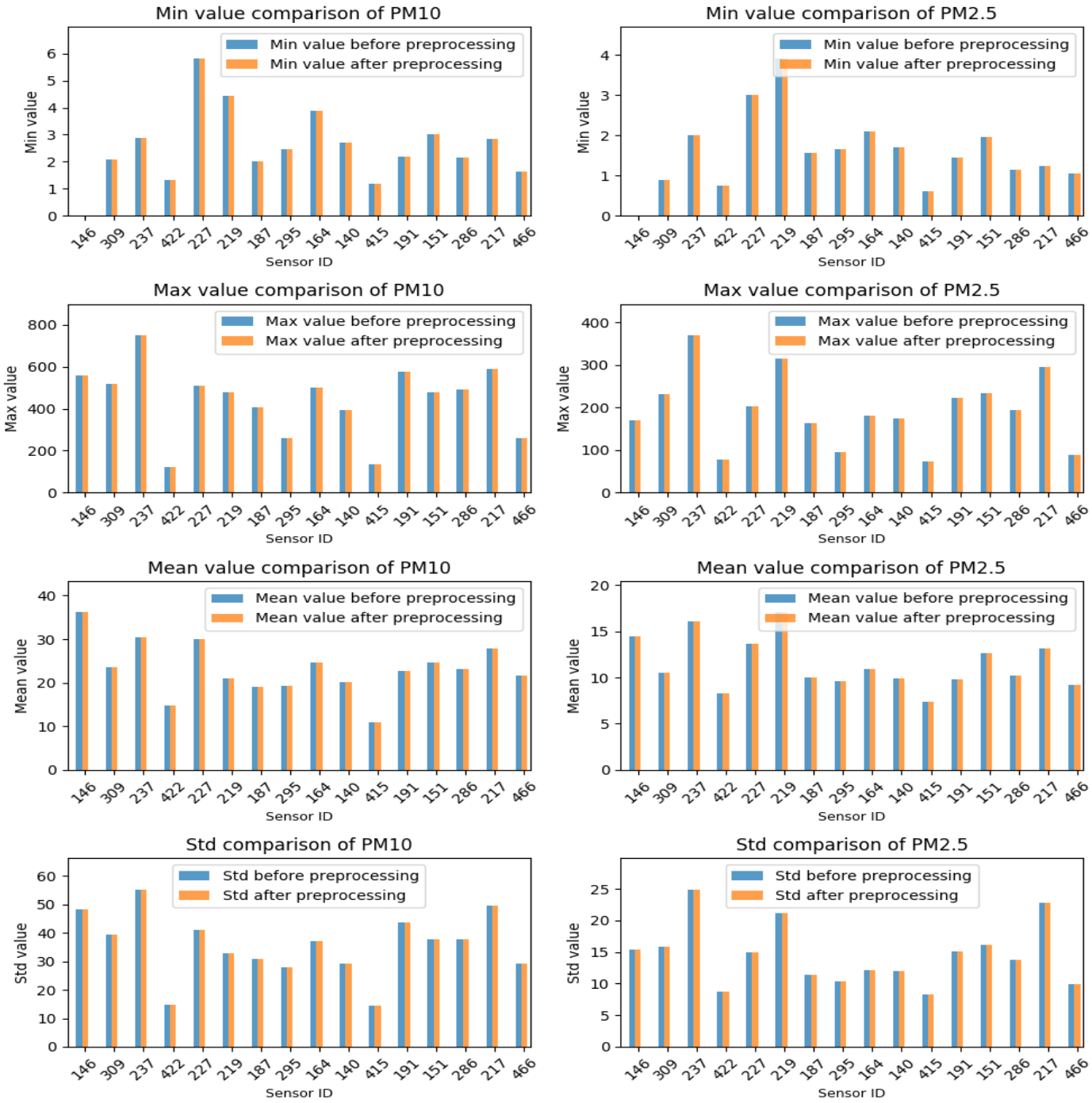


Figure 21: Daily average

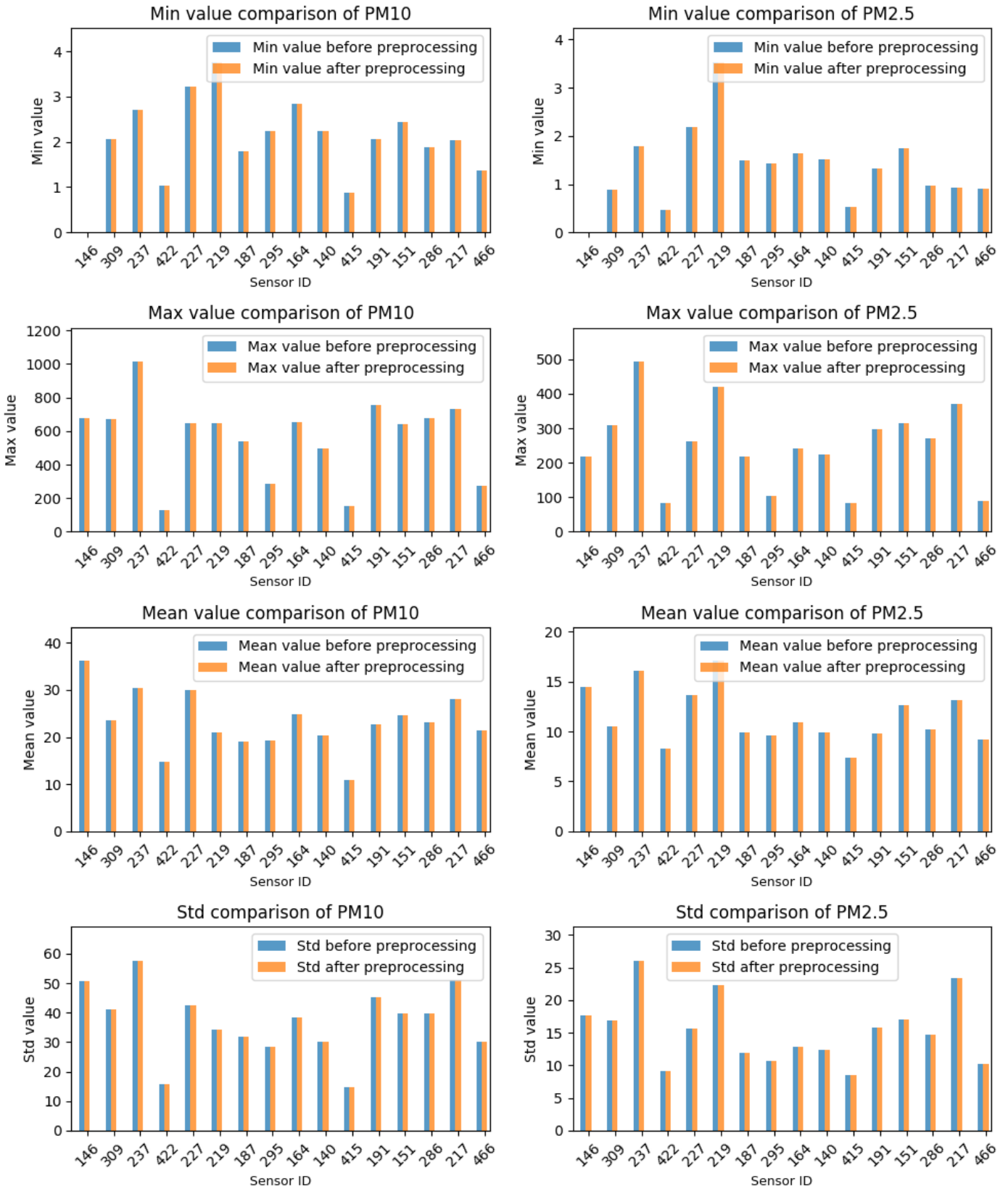


Figure 22: Twelve hour average

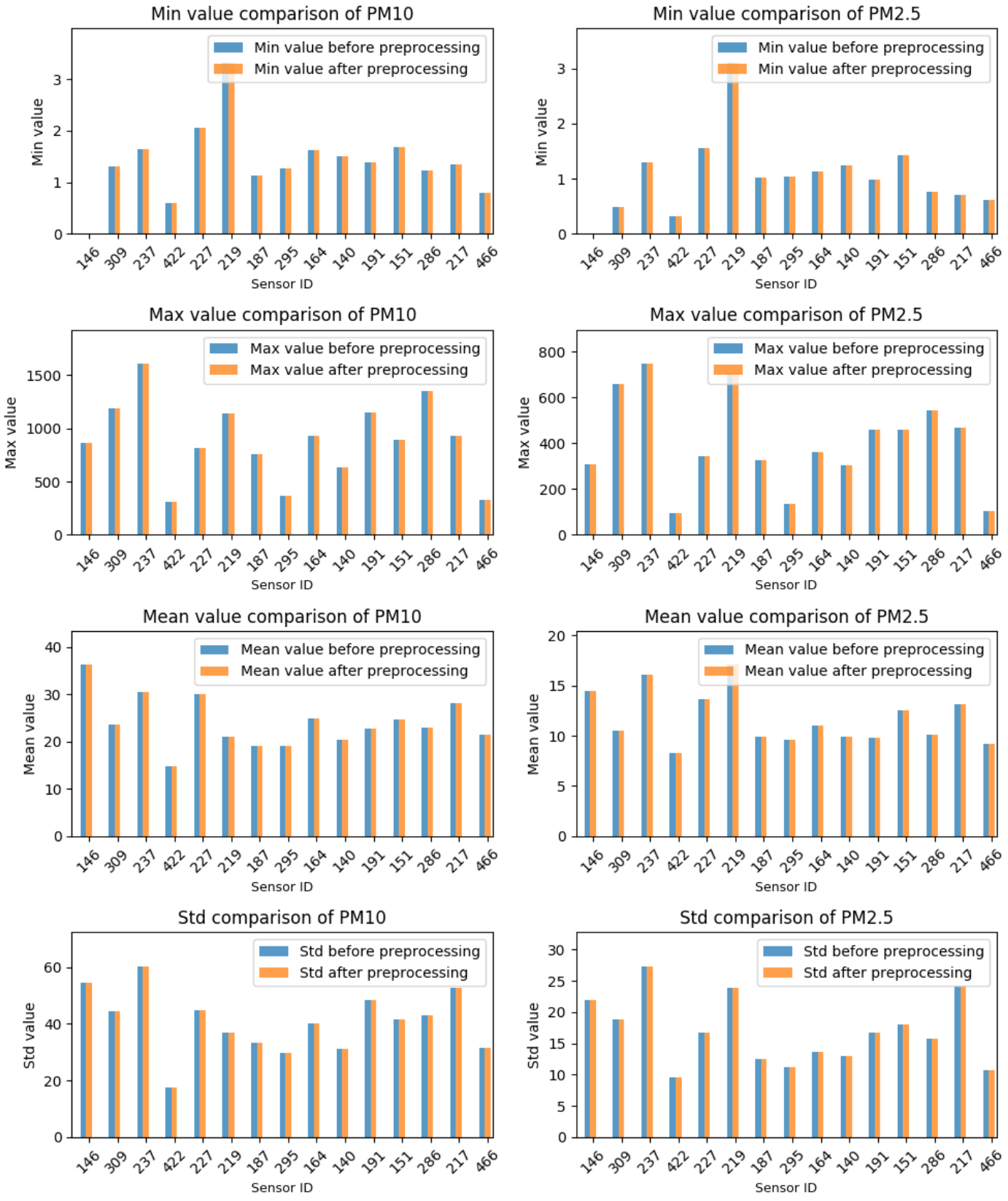
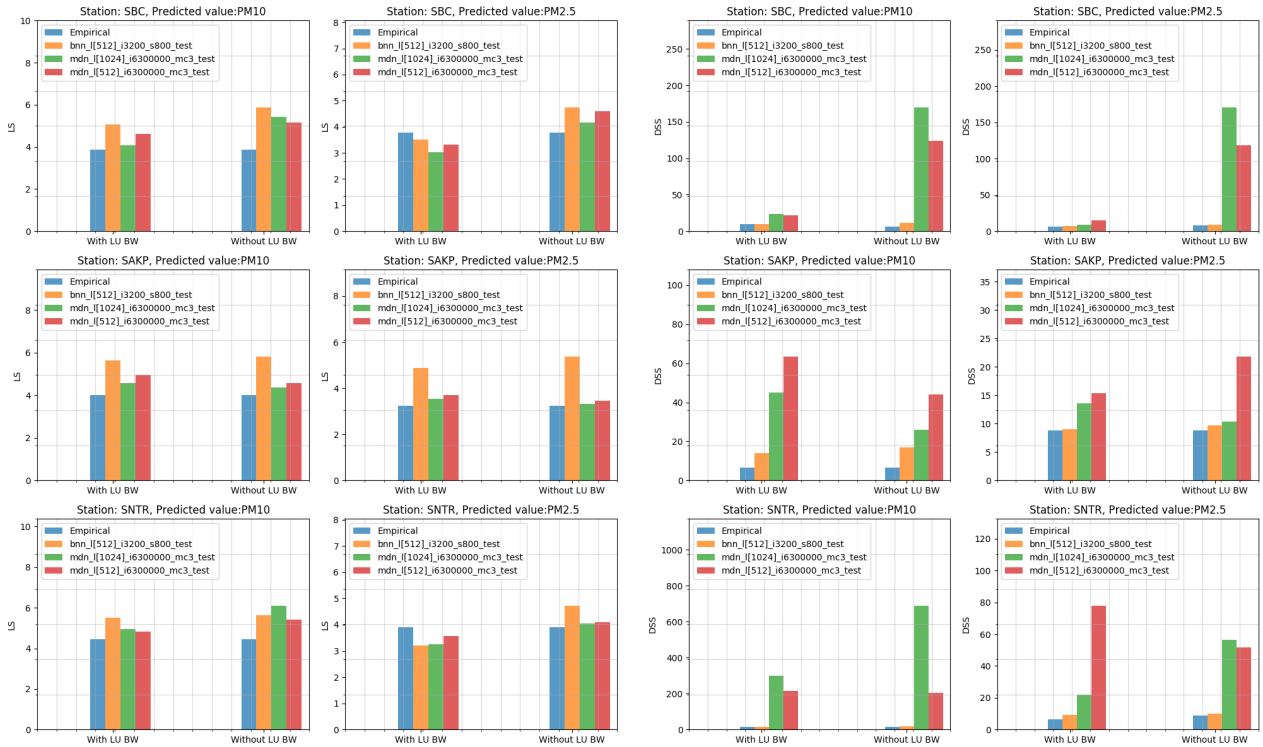


Figure 23: One hour average

C. LS and DSS scores plots

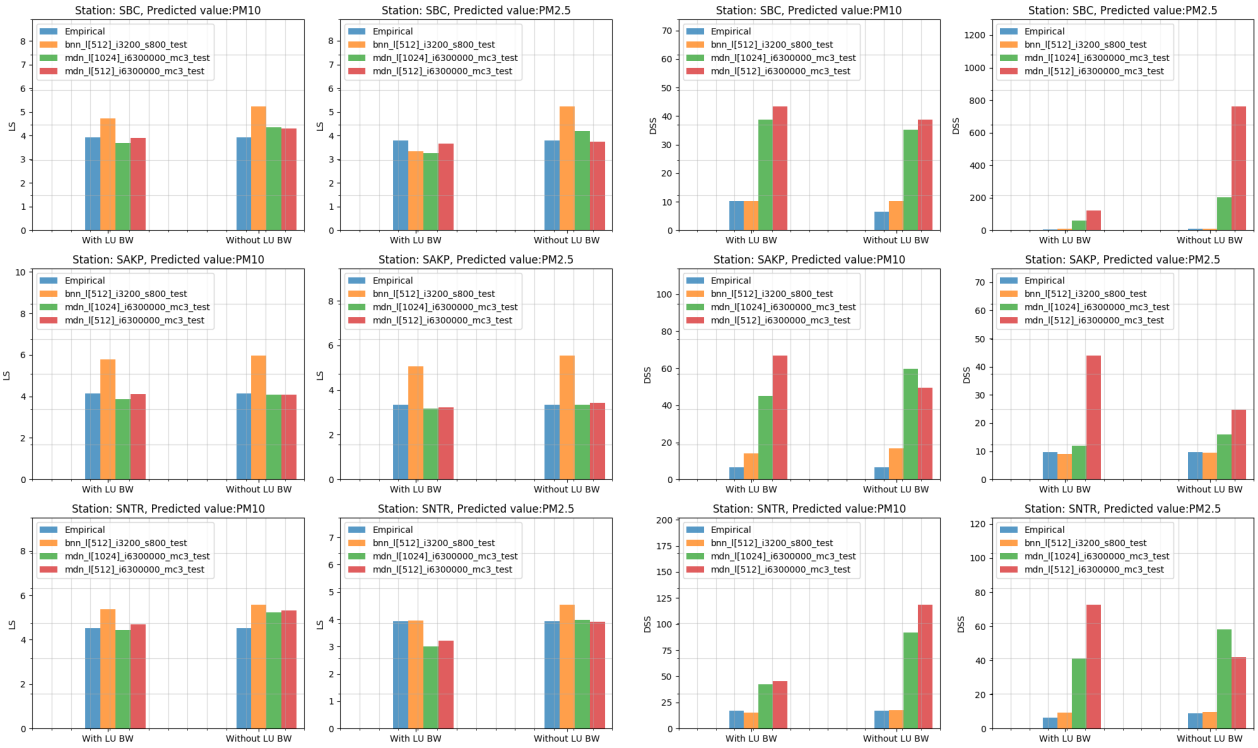
In this appendix we present the summarizing plots of the LS- and DSS-scoring rules over the models. The plots here are analogous to the ones with the CRPS shown in Section 5.2.1, Section 5.2.2 and Section 5.2.3.



(a) Logarithmic scores

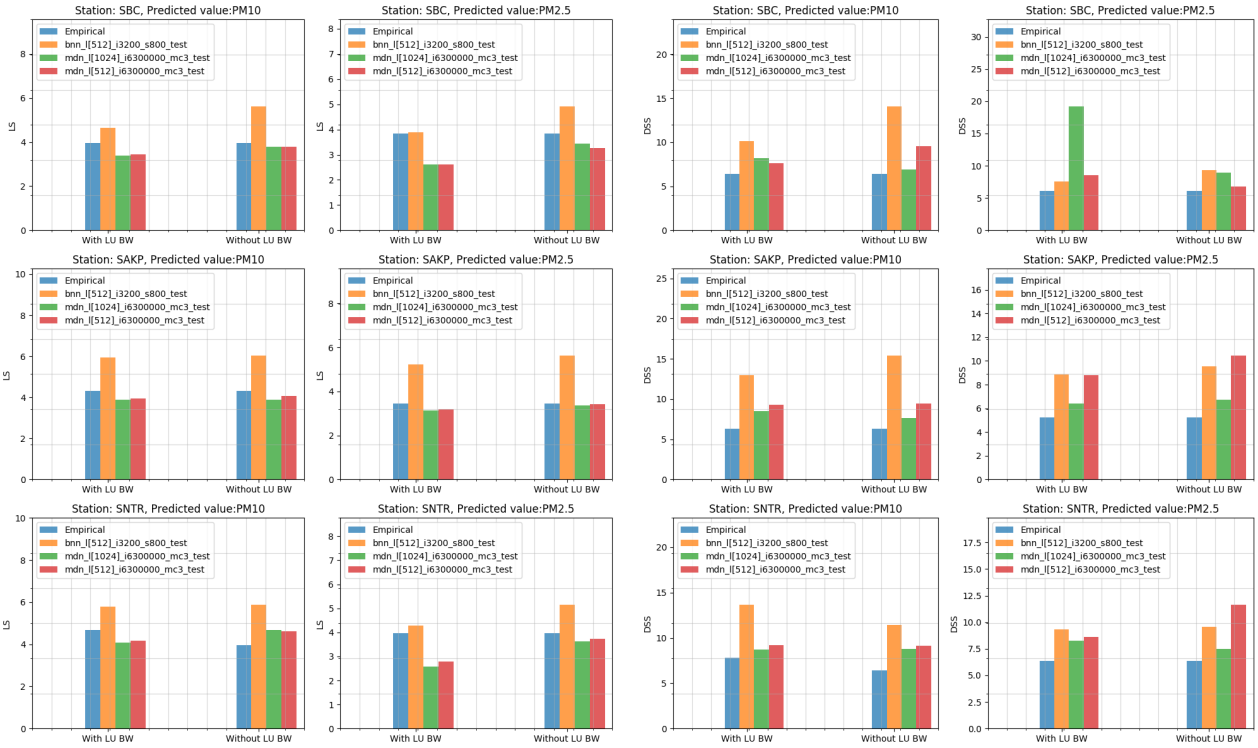
(b) Dawid–Sebastiani scores

Figure 24: DSS and LS values of the models trained with daily averaged data.



(a) Logarithmic scores (b) Dawid–Sebastiani scores

Figure 25: DSS and LS values of the models trained with twelve hourly averaged data.



(a) Logarithmic scores (b) Dawid–Sebastiani scores

Figure 26: DSS and LS values of the models trained with hourly averaged data.

D. Results tables

In this appendix we summarize the results shown with the plots in Section 5.2 by giving them in tabular form. We give three tables, one for each averaging period of the data used. The respective tables are Table 2, Table 3 and Table 4.

The tables are split rowwise in three sections for each of the scoring rules - CRPS, DSS, LS. Each row in these sections is a separate model. The type of the data that was used to train each models is given through the titles of the corresponding columns. The "with LUBW" implies that the corresponding model uses values of LUBW-senors as features whereas "without LUBW" implies the opposite.

All of the values are calculate based on the performance of the models on the test set of the used data.

		SBC PM10		SBC PM2.5		SNTR PM10		SNTR PM2.5		SAKP PM10		SAKP PM2.5	
		with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW
CRPS													
	Emp.	6.22	6.222	5.302	5.304	10.922	10.917	6.027	6.026	5.334	5.332	3.243	3.247
	BNN_1	5.567	12.247	2.918	5.478	8.581	12.439	2.671	5.61	6.681	6.839	3.658	3.966
	MDN_1	3.895	6.255	1.801	3.636	13.85	10.646	1.88	3.854	184.762	22.521	2.923	3.161
	MDN_2	5.02	6.582	2.207	3.836	10.895	10.367	2.07	3.962	170.803	90.188	3.003	3.235
LS													
	Emp.	3.884	3.884	3.764	3.765	4.443	4.442	3.906	3.906	4.017	4.018	3.237	3.241
	BNN_1	5.079	5.882	3.498	4.745	5.504	5.632	3.216	4.723	5.648	5.819	4.891	5.376
	MDN_1	4.087	5.418	3.015	4.173	4.949	6.102	3.26	4.041	4.564	4.377	3.525	3.324
	MDN_2	4.634	5.146	3.328	4.585	4.818	5.4	3.568	4.099	4.951	4.576	3.692	3.452
DSS													
	Emp.	9.909	6.355	5.974	8.325	15.953	15.971	6.308	8.8	6.56	6.561	8.824	8.824
	BNN_1	10.013	11.763	7.49	9.271	15.142	16.951	9.432	9.974	13.978	16.917	9.025	9.645
	MDN_1	23.692	169.926	8.856	170.74	299.41	688.833	21.754	56.629	45.097	25.781	13.59	10.409
	MDN_2	22.189	124.107	15.277	118.104	214.999	205.429	77.95	51.682	63.481	43.886	15.36	21.833

Table 2: Table with results from scoring rules over models trained with daily averaged data.
Emp. : `Empirical`
BNN_1 : `bnn_l[512]_i3200_s800_test`
MDN_1 : `mdn_l[1024]_i6300000_mc3_test`
MDN_2 : `mdn_l[512]_i6300000_mc3_test`

		SBC PM10		SBC PM2.5		SNTR PM10		SNTR PM2.5		SAKP PM10		SAKP PM2.5	
		with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW
CRPS													
	Emp.	6.451	6.451	5.482	5.481	11.645	11.65	6.199	6.203	5.891	5.888	3.585	3.586
	BNN_1	5.565	9.558	2.186	5.727	8.863	12.417	3.362	5.004	7.34	7.518	4.086	4.401
	MDN_1	4.115	4.86	1.743	4.278	9.505	10.163	1.903	4.242	5.935	5.577	2.928	3.311
	MDN_2	4.722	5.138	2.008	4.76	16.477	11.291	2.206	4.367	6.814	5.83	2.978	3.363
LS													
	Emp.	3.916	3.916	3.794	3.793	4.515	4.515	3.931	3.932	4.15	4.148	3.345	3.346
	BNN_1	4.723	5.236	3.351	5.233	5.379	5.58	3.949	4.535	5.793	5.972	5.068	5.548
	MDN_1	3.678	4.353	3.258	4.193	4.442	5.226	3.007	3.98	3.882	4.068	3.158	3.349
	MDN_2	3.902	4.306	3.644	3.738	4.693	5.33	3.209	3.912	4.116	4.068	3.227	3.416
DSS													
	Emp.	10.308	6.385	6.008	8.588	16.816	16.838	6.338	9.047	6.433	6.433	9.777	9.778
	BNN_1	10.319	10.267	8.266	7.828	15.233	17.845	9.246	9.604	13.905	16.741	8.912	9.559
	MDN_1	38.828	35.154	59.36	201.865	42.364	91.931	41.296	58.132	45.138	59.779	11.901	16.055
	MDN_2	43.439	38.79	121.177	762.176	45.287	118.649	72.676	41.84	66.884	49.566	44.0	24.608

Table 3: Table with results from scoring rules over models trained with twelve hourly averaged data.
Emp. : `Empirical`
BNN_1 : `bnn_l[512]_i3200_s800_test`
MDN_1 : `mdn_l[1024]_i6300000_mc3_test`
MDN_2 : `mdn_l[512]_i6300000_mc3_test`

		SBC PM10		SBC PM2.5		SNTR PM10		SNTR PM2.5		SAKP PM10		SAKP PM2.5	
		with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW	with LUBW	without LUBW
CRPS													
	Emp.	6.826	6.825	5.748	5.75	13.435	6.484	6.484	6.483	6.503	6.504	3.925	3.925
	BNN_1	6.321	7.417	4.228	5.662	13.615	18.521	4.281	6.371	8.697	8.842	4.574	4.9
	MDN_1	3.892	5.262	1.754	4.62	8.957	87.812	1.997	4.998	6.247	6.484	3.139	3.695
	MDN_2	4.254	5.652	1.885	4.338	9.554	15.184	2.255	5.089	6.453	6.75	3.203	3.81
LS													
	Emp.	3.968	3.968	3.834	3.834	4.67	3.973	3.973	3.973	4.3	4.299	3.459	3.459
	BNN_1	4.645	5.624	3.896	4.922	5.8	5.883	4.29	5.153	5.937	6.038	5.214	5.634
	MDN_1	3.374	3.786	2.613	3.451	4.093	4.67	2.58	3.64	3.894	3.891	3.124	3.354
	MDN_2	3.451	3.798	2.601	3.266	4.167	4.614	2.784	3.743	3.955	4.057	3.204	3.422
DSS													
	Emp.	6.436	6.436	6.052	6.052	7.788	6.392	6.392	6.391	6.262	6.262	5.232	5.232
	BNN_1	10.164	14.09	7.606	9.302	13.664	11.426	9.303	9.589	12.988	15.413	8.865	9.529
	MDN_1	8.183	6.915	19.213	8.969	8.71	8.817	8.233	7.513	8.515	7.614	6.411	6.736
	MDN_2	7.609	9.586	8.495	6.771	9.224	9.137	8.604	11.652	9.275	9.439	8.776	10.459

Table 4: Table with results from scoring rules over models trained with one hourly averaged data.
Emp : `Empirical`
BNN_1 : `bnn_l[512]_i3200_s800_test`
MDN_1 : `mdn_l[1024]_i6300000_mc3_test`
MDN_2 : `mdn_l[512]_i6300000_mc3_test`

E. Predictive performance checks results tables

In this appendix we give tables with the results of the predictive performance checks over the models, as mentioned in Section 5.1. We again give three separate tables for the three averaging periods. The tables are Table 5, Table 6 and Table 7. The format compares every pair of models. We only consider models that predict the PM10 value of each LUBW sensor and the predictive performance check is calculated with respect to the CRPS.

The tables are split rowwise in three big sections for each LUBW-station – **SBC**, **SNTR**, **SAKP**. Each of these sections is further divided in two smaller ones in which we consider models trained either with or without the use of values from the other two LUBW-stations as features. The column-wise division is only based on the latter consideration. This defines grid of columns and rows where in each column and each row represents a model trained e certain way. At the cell where two models intersect, we give the value of the predictive performance check between them. It is important to note that the predicted station of each column model chages depending on the context in which it is compared. For example, the first column model predicts values of SBC when compared with models from the **SBC** section of the table but it predicts values of SNTR when compared with models from the **SNTR** section of the table. The names of the models are given through aliases that are explained in the caption of each table.

The values given in the tables are the actual metric of the Diebold-Mariano test. The marked with start (*) entries are the ones that we consider as statistically significant. For these entries the corresponding $\boldsymbol{p}$-value is smaller than $\mathbf{0.05}$. If the given metric is negative, the model in the corresponding row is considered better with respect to its predictive capabilities. If the metric is positive – the model in the corresponding column is to be considered as the better one.

All of the values are calculate based on the performance of the models on the test set of the used data.

		PM10 with LUBW				PM10 without LUBW			
		Emp.	BNN_1	MDN_1	MDN_2	Emp.	BNN_1	MDN_1	MDN_2
SBC									
PM10	Emp.	–	1.44	4.73*	3.62*	–	–	–	–
with	BNN_1	-1.44	–	4.13*	1.66	–	–	–	–
LUBW	MDN_1	-4.73*	-4.13*	–	-4.08*	–	–	–	–
	MDN_2	-3.62*	-1.66	4.08*	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-2.88*	-0.1	-0.92
without	BNN_1	–	–	–	–	2.88*	–	2.9*	2.89*
LUBW	MDN_1	–	–	–	–	0.1	-2.9*	–	-1.0
	MDN_2	–	–	–	–	0.92	-2.89*	1.0	–
SNTR									
PM10	Emp.	–	2.51*	-1.48	0.02	–	–	–	–
with	BNN_1	-2.51*	–	-2.73*	-2.18*	–	–	–	–
LUBW	MDN_1	1.48	2.73*	–	1.5	–	–	–	–
	MDN_2	-0.02	2.18*	-1.5	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-1.01	0.43	0.87
without	BNN_1	–	–	–	–	1.01	–	1.28	1.51
LUBW	MDN_1	–	–	–	–	-0.43	-1.28	–	1.42
	MDN_2	–	–	–	–	-0.87	-1.51	-1.42	–
SAKP									
PM10	Emp.	–	-6.94*	-2.21*	-2.37*	–	–	–	–
with	BNN_1	6.94*	–	-2.2*	-2.35*	–	–	–	–
LUBW	MDN_1	2.21*	2.2*	–	0.57	–	–	–	–
	MDN_2	2.37*	2.35*	-0.57	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-7.27*	-2.23*	-2.16*
without	BNN_1	–	–	–	–	7.27*	–	-2.05*	-2.12*
LUBW	MDN_1	–	–	–	–	2.23*	2.05*	–	-2.14*
	MDN_2	–	–	–	–	2.16*	2.12*	2.14*	–

Table 5: Table with results of predictive checks between models trained with daily averaged data.
Emp. : `Empirical`
BNN_1 : `bnn_l[512]_i3200_s800_test`
MDN_1 : `mdn_l[1024]_i6300000_mc3_testl`
MDN_2 : `mdn_l[512]_i6300000_mc3_testl`

		PM10 with LUBW				PM10 without LUBW			
		Emp.	BNN_1	MDN_1	MDN_2	Emp.	BNN_1	MDN_1	MDN_2
SBC									
PM10	Emp.	–	2.37*	6.43*	5.48*	–	–	–	–
with	BNN_1	-2.37*	–	4.43*	2.8*	–	–	–	–
LUBW	MDN_1	-6.43*	-4.43*	–	-5.42*	–	–	–	–
	MDN_2	-5.48*	-2.8*	5.42*	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-2.74*	5.52*	4.98*
without	BNN_1	–	–	–	–	2.74*	–	3.93*	3.72*
LUBW	MDN_1	–	–	–	–	-5.52*	-3.93*	–	-2.76*
	MDN_2	–	–	–	–	-4.98*	-3.72*	2.76*	–
SNTR									
PM10	Emp.	–	4.19*	3.8*	-2.2*	–	–	–	–
with	BNN_1	-4.19*	–	-1.2	-3.48*	–	–	–	–
LUBW	MDN_1	-3.8*	1.2	–	-3.54*	–	–	–	–
	MDN_2	2.2*	3.48*	3.54*	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-0.81	3.09*	0.6
without	BNN_1	–	–	–	–	0.81	–	2.76*	1.19
LUBW	MDN_1	–	–	–	–	-3.09*	-2.76*	–	-2.56*
	MDN_2	–	–	–	–	-0.6	-1.19	2.56*	–
SAKP									
PM10	Emp.	–	-9.34*	-0.14	-2.06*	–	–	–	–
with	BNN_1	9.34*	–	3.9*	1.15	–	–	–	–
LUBW	MDN_1	0.14	-3.9*	–	-3.36*	–	–	–	–
	MDN_2	2.06*	-1.15	3.36*	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-10.06*	1.88	0.29
without	BNN_1	–	–	–	–	10.06*	–	8.05*	6.39*
LUBW	MDN_1	–	–	–	–	-1.88	-8.05*	–	-2.32*
	MDN_2	–	–	–	–	-0.29	-6.39*	2.32*	–

Table 6: Table with results of predictive checks between models trained with twelve hourly averaged data.
Emp. : `Empirical`
BNN_1 : `bnn_l[512]_i3200_s800_test`
MDN_1 : `mdn_l[1024]_i6300000_mc3_testl`
MDN_2 : `mdn_l[512]_i6300000_mc3_testl`

		PM10 with LUBW				PM10 without LUBW			
		Emp.	BNN_1	MDN_1	MDN_2	Emp.	BNN_1	MDN_1	MDN_2
SBC									
PM10	Emp.	–	3.54*	27.9*	24.85*	–	–	–	–
with	BNN_1	-3.54*	–	21.08*	19.45*	–	–	–	–
LUBW	MDN_1	-27.9*	-21.08*	–	-11.58*	–	–	–	–
	MDN_2	-24.85*	-19.45*	11.58*	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-4.26*	18.59*	10.34*
without	BNN_1	–	–	–	–	4.26*	–	18.51*	13.11*
LUBW	MDN_1	–	–	–	–	-18.59*	-18.51*	–	-5.49*
	MDN_2	–	–	–	–	-10.34*	-13.11*	5.49*	–
SNTR									
PM10	Emp.	–	-0.68	19.11*	15.78*	–	–	–	–
with	BNN_1	0.68	–	16.47*	13.7*	–	–	–	–
LUBW	MDN_1	-19.11*	-16.47*	–	-3.34*	–	–	–	–
	MDN_2	-15.78*	-13.7*	3.34*	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-26.57*	-7.0*	-18.48*
without	BNN_1	–	–	–	–	26.57*	–	-5.96*	6.49*
LUBW	MDN_1	–	–	–	–	7.0*	5.96*	–	6.28*
	MDN_2	–	–	–	–	18.48*	-6.49*	-6.28*	–
SAKP									
PM10	Emp.	–	-33.78*	3.84*	0.72	–	–	–	–
with	BNN_1	33.78*	–	22.9*	20.66*	–	–	–	–
LUBW	MDN_1	-3.84*	-22.9*	–	-5.02*	–	–	–	–
	MDN_2	-0.72	-20.66*	5.02*	–	–	–	–	–
PM10	Emp.	–	–	–	–	–	-34.62*	0.38	-4.41*
without	BNN_1	–	–	–	–	34.62*	–	23.79*	21.71*
LUBW	MDN_1	–	–	–	–	-0.38	-23.79*	–	-8.2*
	MDN_2	–	–	–	–	4.41*	-21.71*	8.2*	–

Table 7: Table with results of predictive checks between models trained with one hourly averaged data.
Emp. : `Empirical`
BNN_1 : `bnn_l[512]_i3200_s800_test`
MDN_1 : `mdn_l[1024]_i6300000_mc3_testl`
MDN_2 : `mdn_l[512]_i6300000_mc3_testl`

F. Models plots

In this appendix we present plots that compare the predicted by the models values and the actual observations. We do not give plot for every single model. We have selected plots illustrating the predictive performance of the relatively good models. We have plotted the results of the predictions both on the training and on the test set for each plot. As with previous result considerations, we distinguish between the averaging period of the data with which the models were trained. The first two plots – Figure 27 and Figure 28 – show plots of models trained on daily averaged data. The next two figures – Figure 29 and Figure 30 – show plots from models trained on twelve hourly averaged data and final two plots – Figure 31 and Figure 32 – models trained on one hourly averaged data.

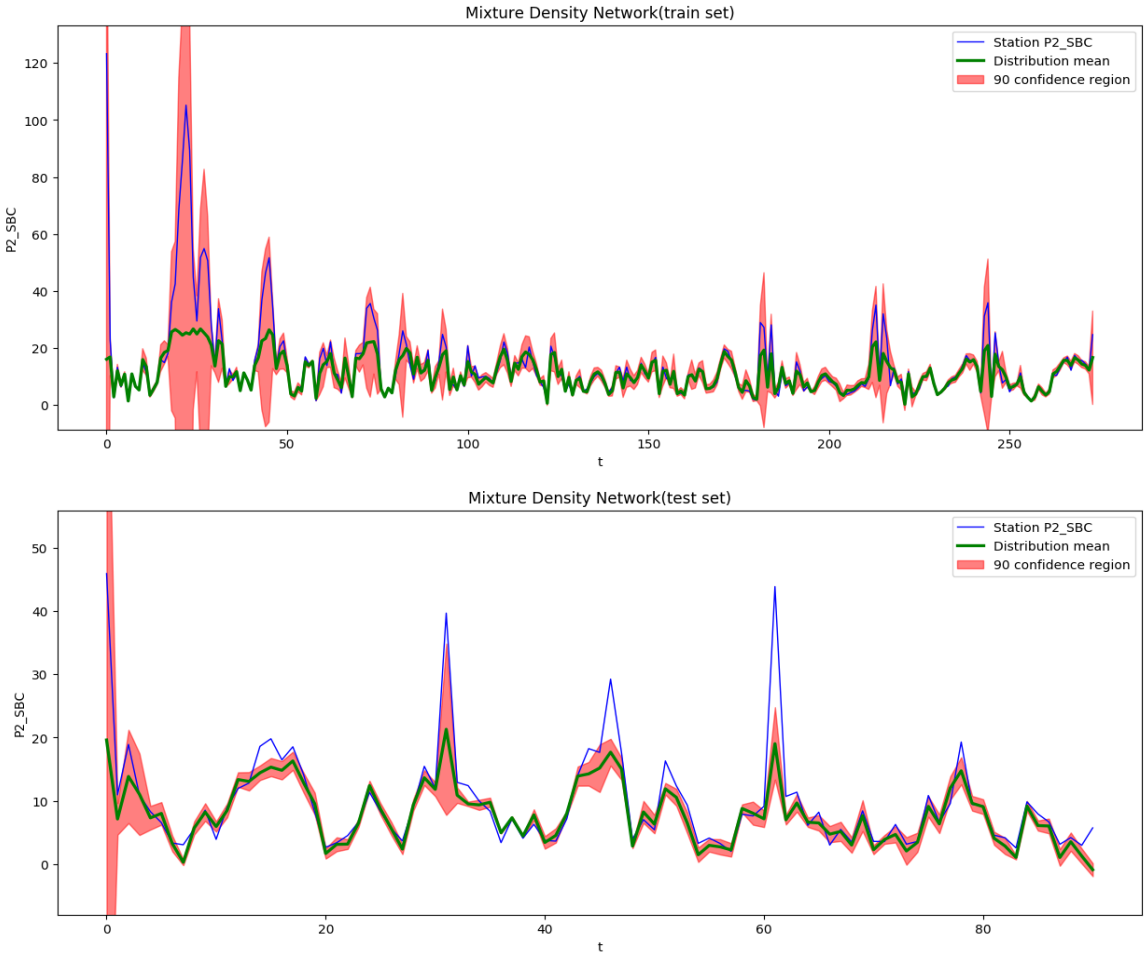


Figure 27: A MDN model with good performance trained on daily averaged data

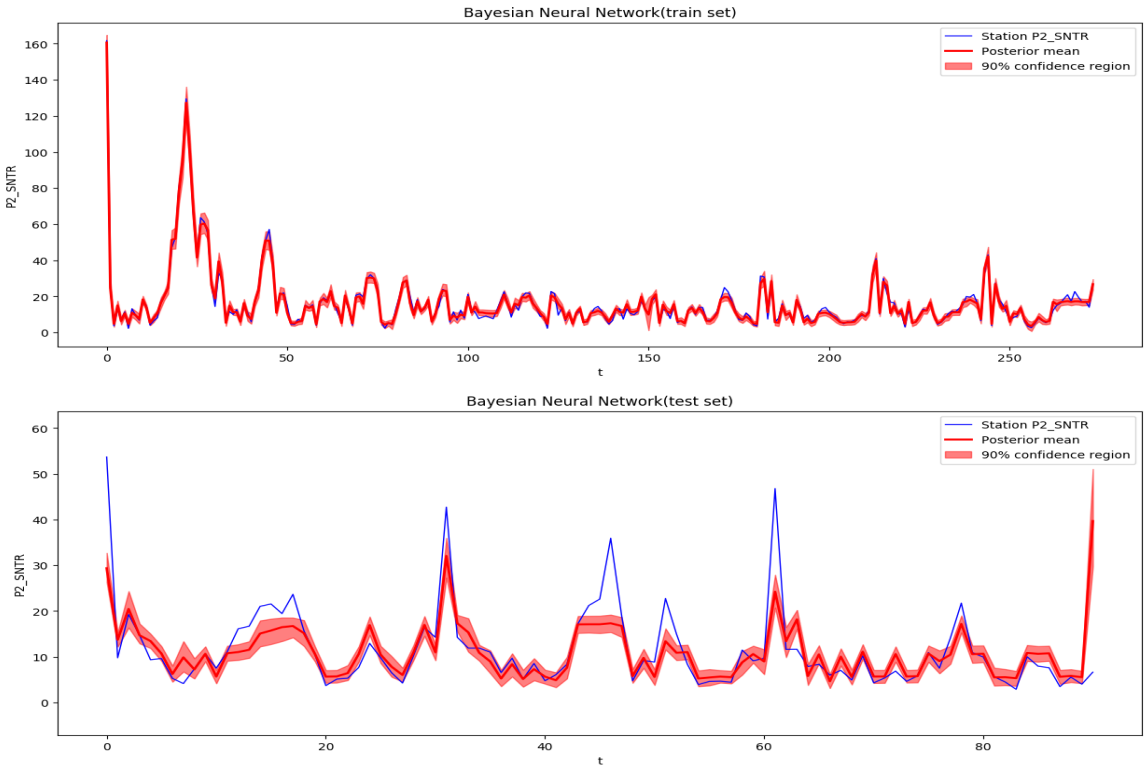


Figure 28: A BNN model with good performance trained on daily averaged data

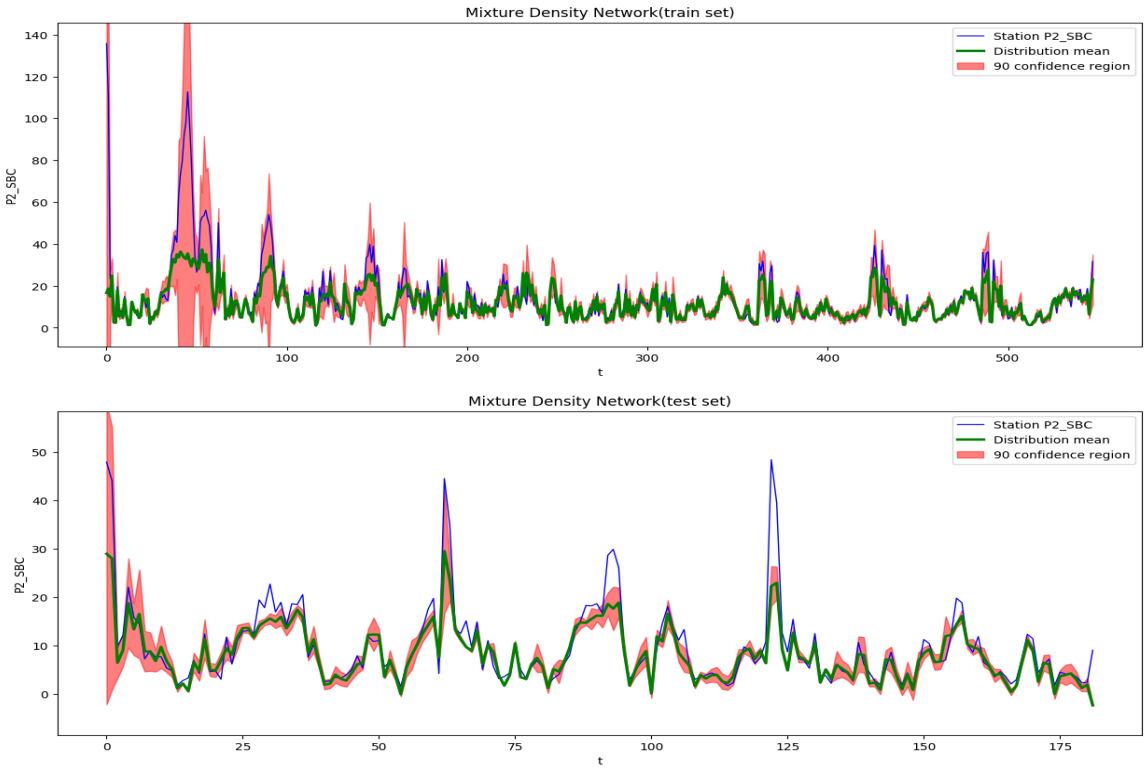


Figure 29: A MDN model with good performance trained on twelve hourly averaged data

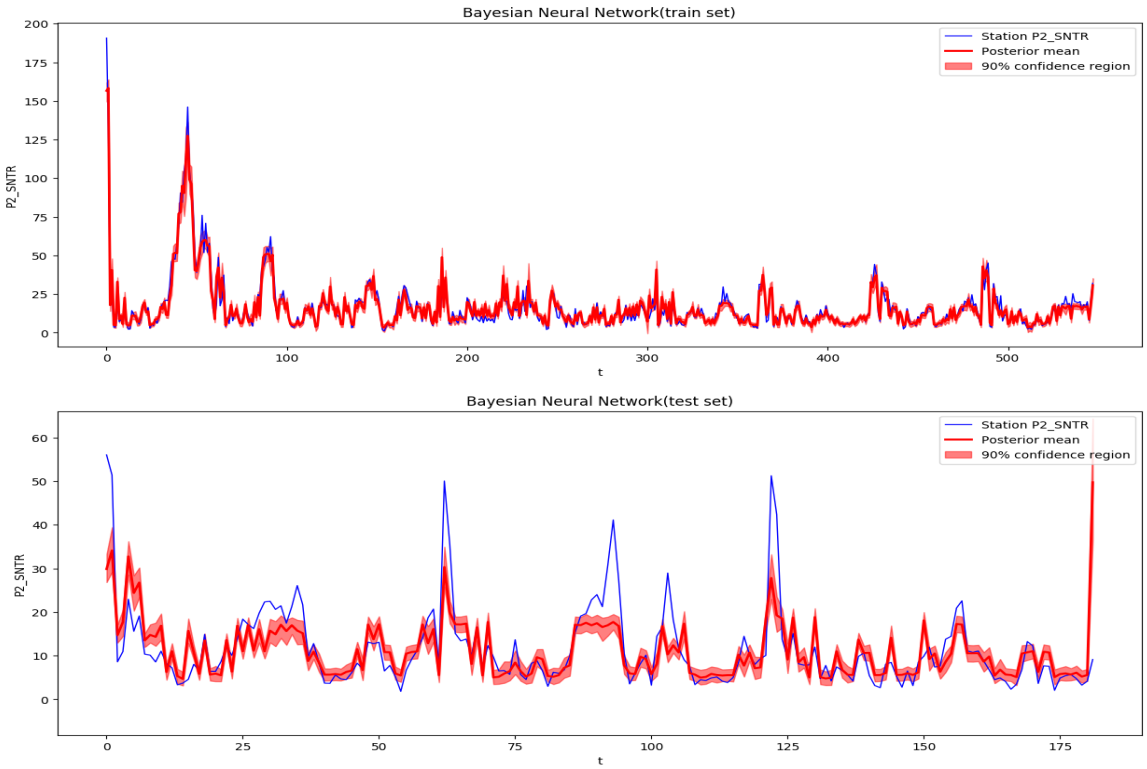


Figure 30: A BNN model with good performance trained on twelve hourly averaged data

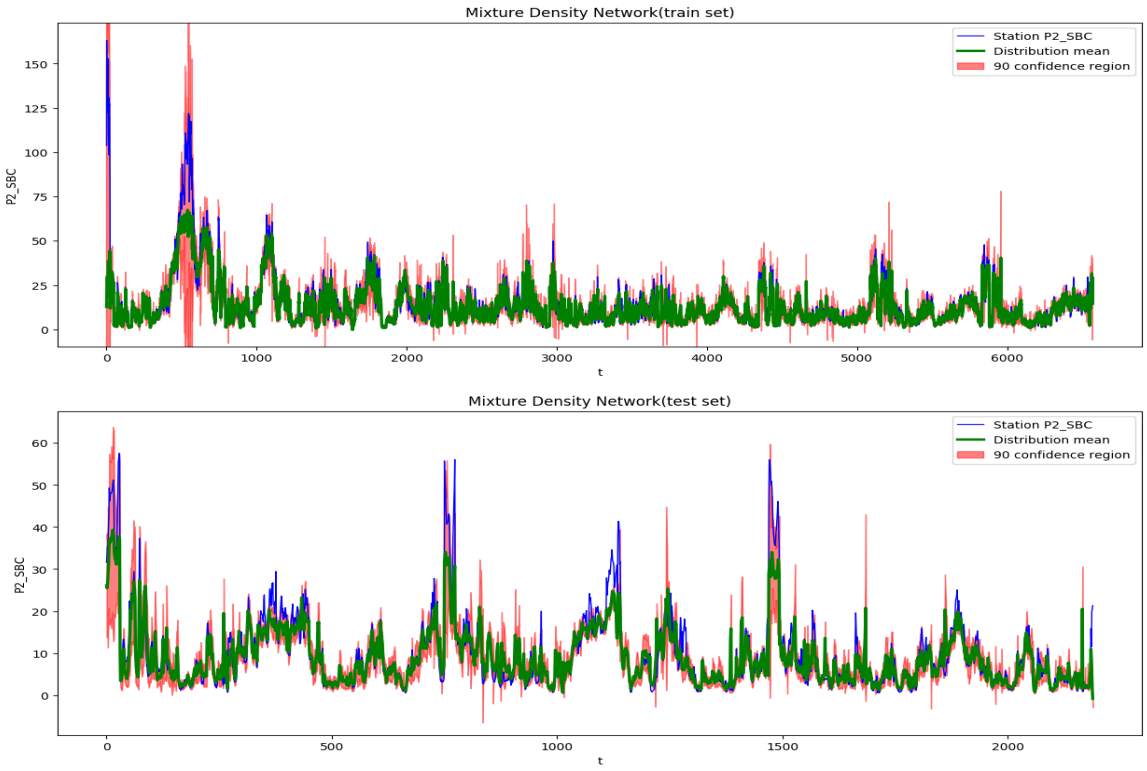


Figure 31: A MDN model with good performance trained on one hourly averaged data

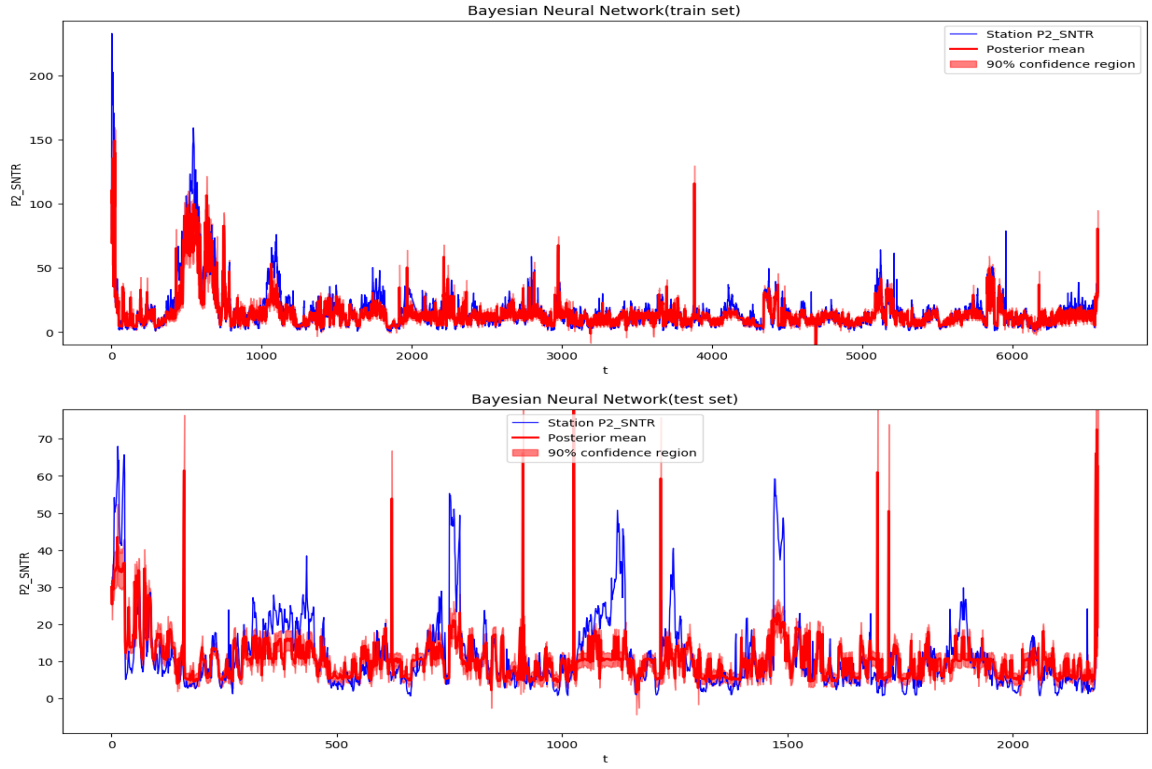
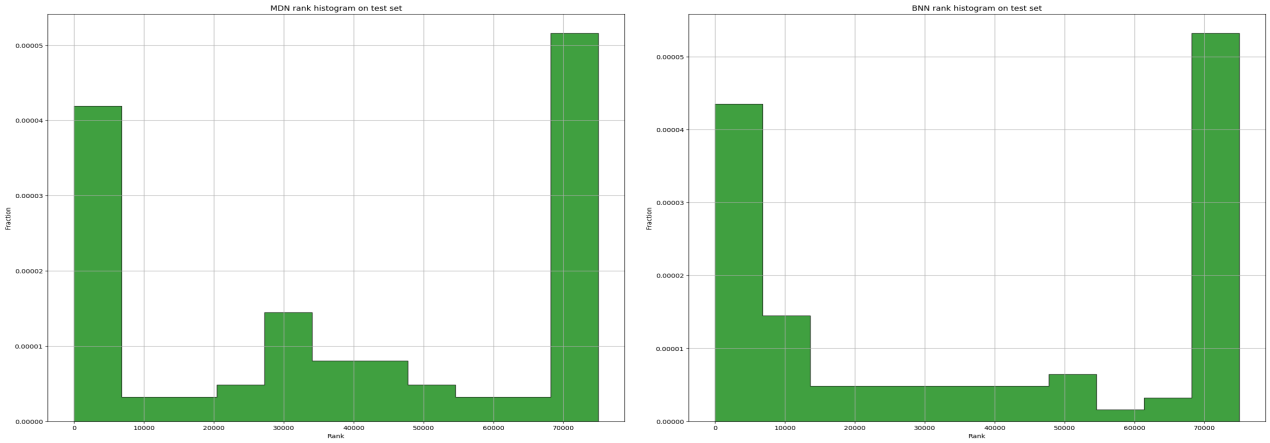


Figure 32: A BNN model with good performance trained on one hourly averaged data. To note is, however, that even though this BNN is considered to have good performance, BNNs struggled to predict the data in the context of one hour average. This plot illustrates this.

G. Models rank histograms

In this appendix we present the rank histograms of some of the models. As discussed in Section 5.2, the rank histograms given here are representative of the rest and illustrate one of the problems of the built models. From all given rank histograms we can see that they are U-shaped which means that the models often miss the actual observation they are trying to predict.

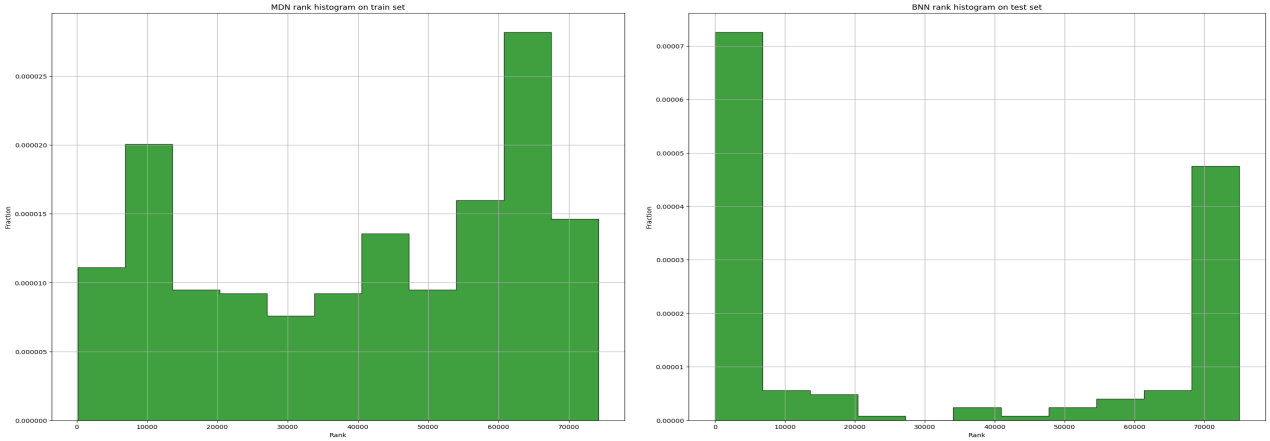
We give one histogram for the BNNs and MDNs per averaging period. Figure 33 illustrates the rank histograms of models trained with daily averaged data, Figure 34 – models trained with twelve hourly averaged data and finally Figure 35 – models trained with hourly averaged data. All of the histograms are calculated on the test set.



(a) A rank histogram of a MDN model trained with daily averaged data.

(b) A rank histogram of a BNN model trained with daily averaged data.

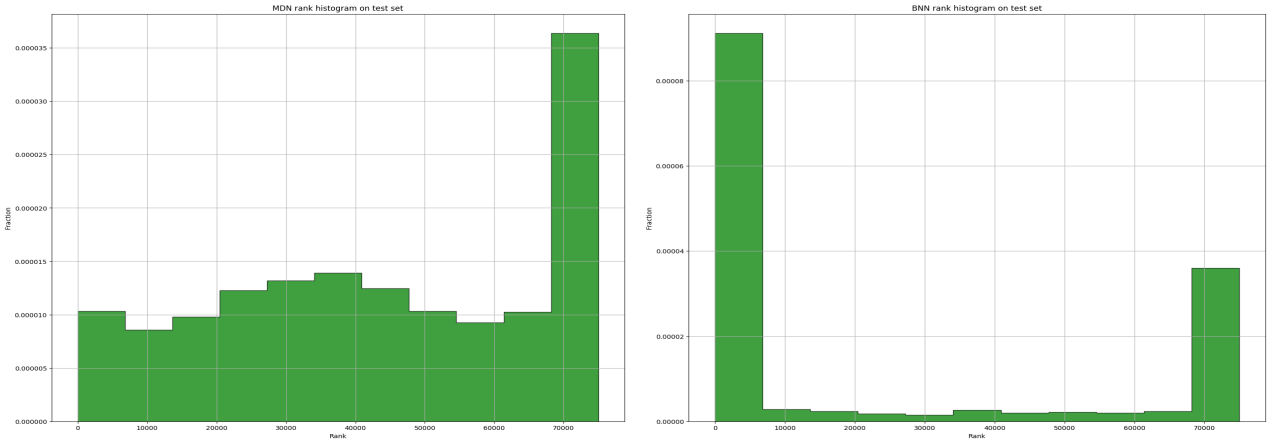
Figure 33



(a) A rank histogram of a MDN model trained with daily averaged data.

(b) A rank histogram of a BNN model trained with twelve hourly averaged data.

Figure 34



(a) A rank histogram of a MDN model trained with averaged data.

(b) A rank histogram of a BNN model trained with hourly averaged data.

Figure 35

H. Empirical models plots

In this appendix we give plots of the predictions made by the empirical models. As these models do not make difference between test and train set, we have evaluated them only on the part of the data that would have been used as a test set for a MDN or a BNN.

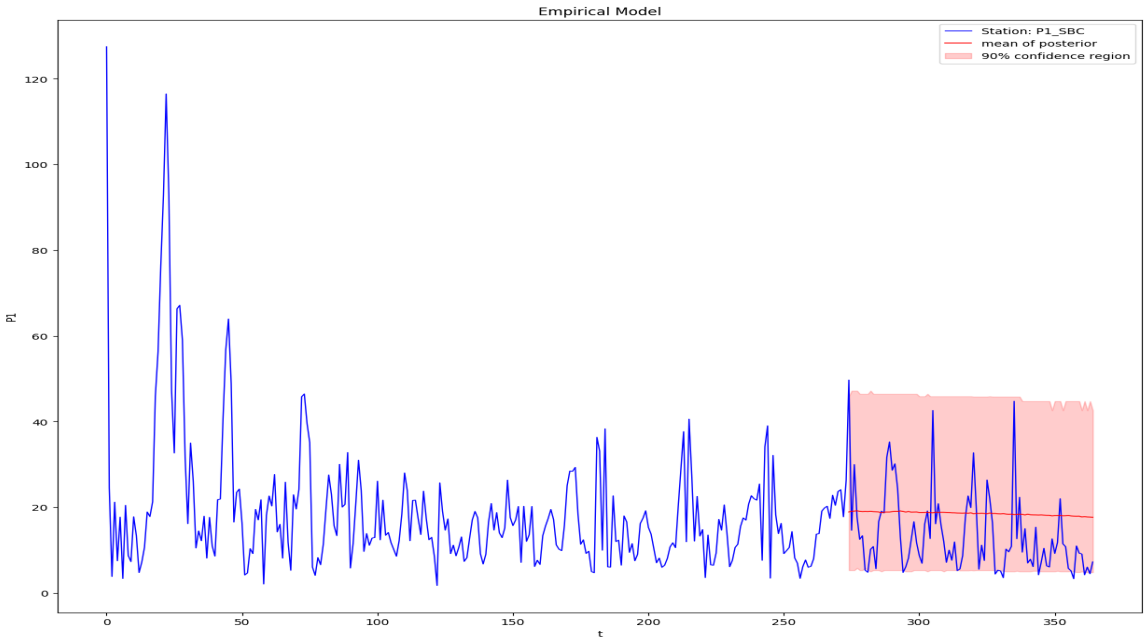


Figure 36: An empirical model predicting daily averaged data.

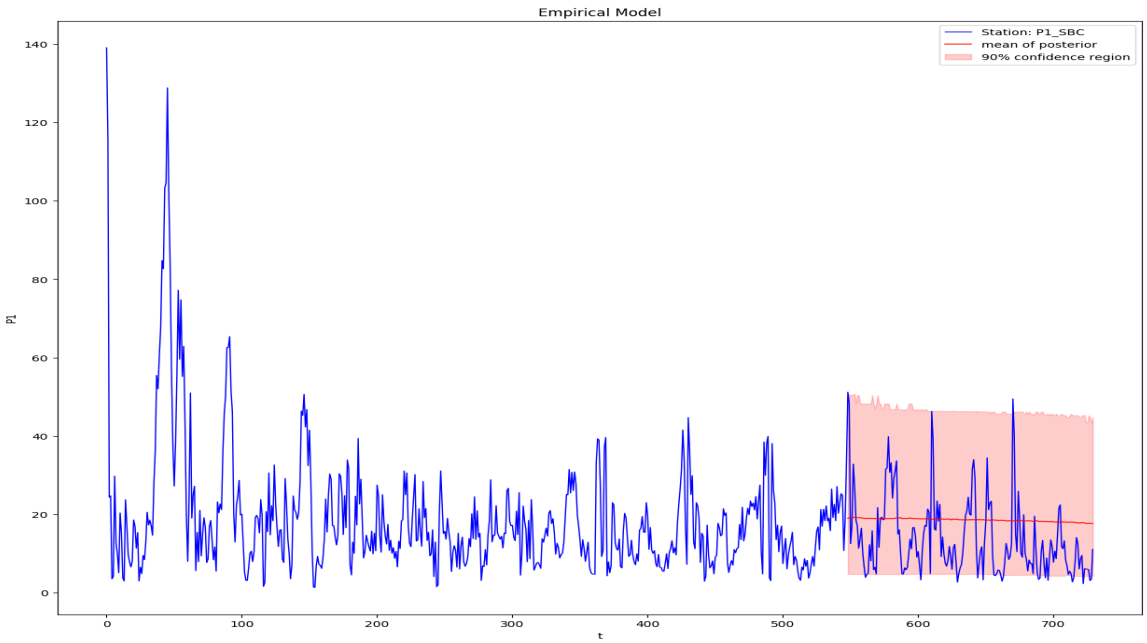


Figure 37: An empirical model predicting twelve hourly averaged data.

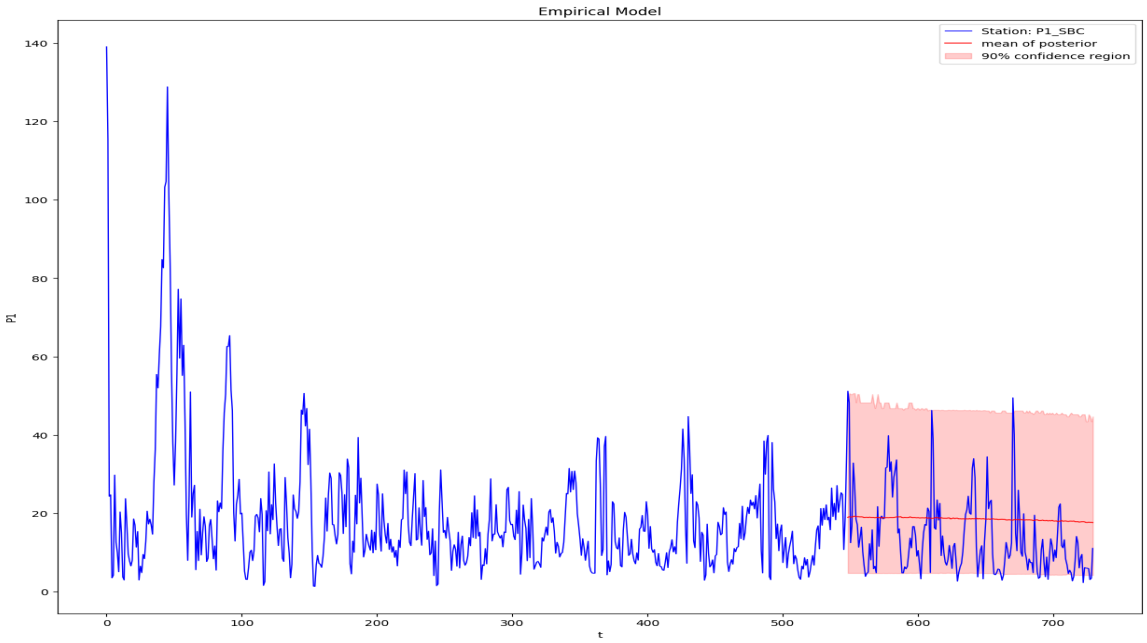


Figure 38: An empirical model predicting hourly averaged data.

References

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. (Page 42).

[Bay63] T. Bayes. An essay towards solving a problem in the doctrine of chances. *Phil. Trans. of the Royal Soc. of London*, 53:370–418, 1763. (Page 5).

[BCKW15] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight Uncertainty in Neural Networks. *ArXiv e-prints*, May 2015. (Pages 7, 14).

[BKM16] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational Inference: A Review for Statisticians. *ArXiv e-prints*, January 2016. (Page 13).

[BLNZ95] R. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995. (Page 12).

[BNVP01] K. Bertels, L. Neuberg, S. Vassiliadis, and D.G. Pechanek. On chaos and neural networks: The backpropagation paradigm. *Artificial Intelligence Review*, 15(3):165–187, May 2001. (Pages 7, 18).

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001. (Page 23).

[Bri90] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. (Page 17).

[bs] Beautiful soup. https://www.crummy.com/software/BeautifulSoup/. Accessed: 2018-08-21. (Page 43).

[BXP+16] Y. Bai, Y. Xu, J. Pan, J.Q. Lan, and W.W. Gao. Application of bayesian neural networks to energy reconstruction in eas experiments for ground-based tev astrophysics. *Journal of Instrumentation*, 11(07):P07006, 2016. (Page 8).

[Cle] Michael P. Clements. Evaluating the bank of england density forecasts of inflation*. *The Economic Journal*, 114(498):844–866. (Page 8).

[Col07] Mat Collins. Ensembles and probabilities: A new era in the prediction of climate change. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 365(1857):1957–1970, 2007. (Page 8).

[CZ15] J. Chorowski and J. M. Zurada. Learning understandable neural networks with nonnegative weight constraints. *IEEE Transactions on Neural Networks and Learning Systems*, 26(1):62–69, Jan 2015. (Page 47).

[Daw84] A. P. Dawid. Statistical theory: the prequential approach (with discussion). *J. R. Statist. Soc. A*, 147:278–292, 1984. (Page 19).

[DGT98] Francis Diebold, Todd A Gunther, and Anthony S Tay. Evaluating density forecasts with applications to financial risk management. *International Economic Review*, 39(4):863–83, 1998. (Pages 19, 25).

[Die15] Francis X. Diebold. Comparing predictive accuracy, twenty years later: A personal perspective on the use and abuse of diebold–mariano tests. *Journal of Business Economic Statistics*, 33(1):1–1, 2015. (Page 26).

[DKPR87] Simon Duane, A. D. Kennedy, Brian J. Pendleton, and Duncan Roweth. Hybrid monte carlo. *Physics Letters B*, 195(2):216 – 222, 1987. (Page 15).

[DL90] John S. Denker and Yann LeCun. Transforming neural-net output levels to probability distributions. In *Proceedings of the 3rd International Conference on Neural Information Processing Systems*, NIPS'90, pages 853–859, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc. (Page 3).

[DM95] Francis Diebold and Roberto Mariano. Comparing predictive accuracy. *Journal of Business and Economic Statistics*, 13(3):253–63, 1995. (Page 26).

[DS99] A. Philip Dawid and Paola Sebastiani. Coherent dispersion criteria for optimal experimental design. *Ann. Statist.*, 27(1):65–81, 03 1999. (Page 22).

[Dut] Vincent Dutordoir. Mixture density networks in gpflow. `https://www.prowler.io/blog/mixture-density-networks-in-gpflow-a-tutorial`. Accessed: 2018-08-20. (Pages 46, 47).

[EG12] Werner Ehm and Tilmann Gneiting. Local proper scoring rules of order two. *Ann. Statist.*, 40(1):609–637, 02 2012. (Pages 8, 21).

[EvOHS15] J. M. Eden, G. J. van Oldenborgh, E. Hawkins, and E. B. Suckling. A global empirical system for probabilistic seasonal climate prediction. *Geoscientific Model Development*, 8(12):3947–3973, 2015. (Page 7).

[FBV17] M. Fortunato, C. Blundell, and O. Vinyals. Bayesian Recurrent Neural Networks. *ArXiv e-prints*, April 2017. (Page 8).

[FR12] Charles W. Fox and Stephen J. Roberts. A tutorial on variational bayesian inference. *Artificial Intelligence Review*, 38(2):85–95, Aug 2012. (Pages 13, 14).

[Gal16] Yarin Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016. (Pages 7, 9, 13).

[GBR] Tilmann Gneiting, Fadoua Balabdaoui, and Adrian E. Raftery. Probabilistic forecasts, calibration and sharpness. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 69(2):243–268. (Pages 19, 20).

[GG84] S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):721–741, Nov 1984. (Page 15).

[GGBJ07] E. P. Grimit, T. Gneiting, V. J. Berrocal, and N. A. Johnson. The continuous ranked probability score for circular variables and its application to mesoscale forecast ensemble verification. *Quarterly Journal of the Royal Meteorological Society*, 132(621C):2925–2942, 2007. (Page 21).

[GK14] Tilmann Gneiting and Matthias Katzfuss. Probabilistic forecasting. *Annual Review of Statistics and Its Application*, 1(1):125–151, 2014. (Pages 8, 19, 20, 22, 23, 25, 26).

[Goo52] I. J. Good. Rational decisions. *Journal of the Royal Statistical Society. Series B (Methodological)*, 14(1):107–114, 1952. (Page 20).

[GR07] Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378, 2007. (Pages 4, 21).

[GR11] Tilmann Gneiting and Roopesh Ranjan. Comparing density forecasts using threshold- and quantile-weighted scoring rules. *Journal of Business Economic Statistics*, 29(3):411–422, 2011. (Page 21).

[GR13] Tilmann Gneiting and Roopesh Ranjan. Combining predictive distributions. *Electron. J. Statist.*, 7:1747–1782, 2013. (Page 19).

[Ham01] Thomas M. Hamill. Interpretation of rank histograms for verifying ensemble forecasts. *Monthly Weather Review*, 129(3):550–560, 2001. (Pages 24, 25).

[Has70] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970. (Page 15).

[HH64] J. M. Hammersley and D. C. Handscomb. *General Principles of the Monte Carlo Method*, pages 50–75. Springer Netherlands, Dordrecht, 1964. (Page 15).

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. (Page 12).

[Hun07] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95, 2007. (Page 42).

[JGJS99] Michael I. Jordan, Zoubin Ghahramani, Tommi S. Jaakkola, and Lawrence K. Saul. An introduction to variational methods for graphical models. *Machine Learning*, 37(2):183–233, Nov 1999. (Page 13).

[JJNH91] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Comput.*, 3(1):79–87, March 1991. (Page 8).

[JKL17] A. Jordan, F. Krüger, and S. Lerch. Evaluating probabilistic forecasts with scor-

ingRules. *ArXiv e-prints*, September 2017. (Page 8).

[JOP+] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>]. (Page 42).

[KB14] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, December 2014. (Page 12).

[KCS] Vivek Katiyar, Narottam Chand, and Surender Soni. Clustering algorithms for heterogeneous wireless sensor network: A survey. *International Journal of Applied Engineering Research*, page 2010. (Page 8).

[KL51] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951. (Page 13).

[KP17] Yongchan Kwon and Myunghee Paik. Uncertainty quantification for ischemic stroke lesion segmentation using bayesian neural networks. 12 2017. (Page 8).

[Krz01] Roman Krzysztofowicz. The case for probabilistic forecasting in hydrology. *Journal of Hydrology*, 249(1):2 – 9, 2001. (Page 8).

[KVS11] Oliver Krueger and Jin-Song Von Storch. A simple empirical model for decadal climate prediction. *Journal of Climate*, 24(4):1276–1283, 2011. (Page 7).

[KW13] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013. (Page 14).

[lub] Landesanstalt für umwelt, messungen und naturschutz baden-württemberg. https://www.lubw.baden-wuerttemberg.de/startseite. Accessed: 2018-08-21. (Page 1).

[luf] Ok lab stuttgart. https://luftdaten.info/. Accessed: 2018-08-21. (Pages 1, 6, 42, 43).

[LV01] Jouko Lampinen and Aki Vehtari. Bayesian approach for neural networks—review and case studies. *Neural Networks*, 14(3):257 – 274, 2001. (Page 8).

[LZLQ17] Zuozhu Liu, Wenyu Zhang, Shaowei Lin, and Tony Q. S. Quek. Heterogeneous sensor data fusion by deep multimodal encoding. *IEEE Journal of Selected Topics in Signal Processing*, 11:479–491, 2017. (Page 8).

[Mac92] David J. C. MacKay. A practical bayesian framework for backpropagation networks. *Neural Computation*, 4(3):448–472, 1992. (Page 7).

[MB94] Christopher M. Bishop. Mixture density networks. 01 1994. (Pages 15, 16, 18).

[McK10] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010. (Page 42).

[MDD03] Tatiana Miazhynskaia, Georg Dorffner, and Engelbert J. Dockner. Risk management application of the recurrent mixture density network models. In Okyay

Kaynak, Erkki Alpaydin, Ethemand Oja, and Lei Xu, editors, *Artificial Neural Networks and Neural Information Processing — ICANN/ICONIP 2003*, pages 589–596, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. (Page 8).

[MHW12] Jacob M. Montgomery, Florian M. Hollenbach, and Michael D. Ward. Ensemble predictions of the 2012 us presidential election. *PS: Political Science; Politics*, 45(4):651–654, 2012. (Page 8).

[Mit97] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. (Page 9).

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. (Page 15).

[MvN+17] Alexander G. de G. Matthews, Mark van der Wilk, Tom Nickson, Keisuke. Fujii, Alexis Boukouvalas, Pablo Le'on-Villagr'a, Zoubin Ghahramani, and James Hensman. GPflow: A Gaussian process library using TensorFlow. *Journal of Machine Learning Research*, 18(40):1–6, apr 2017. (Page 42).

[MW76] James E. Matheson and Robert L. Winkler. Scoring rules for continuous probability distributions. *Management Science*, 22(10):1087–1096, 1976. (Page 21).

[Nea96] Radford M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag, Berlin, Heidelberg, 1996. (Page 7).

[NH98] Radford M. Neal and Geoffrey E. Hinton. *A View of the Em Algorithm that Justifies Incremental, Sparse, and other Variants*, pages 355–368. Springer Netherlands, Dordrecht, 1998. (Page 14).

[Oli15] Travis E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition, 2015. (Page 42).

[PBJ12] J. Paisley, D. Blei, and M. Jordan. Variational Bayesian Inference with Stochastic Search. *ArXiv e-prints*, June 2012. (Pages 7, 14, 15).

[PCB15] Harrison B. Prosper Pushpalatha C. Bhat. Bayesian neural networks. 2015. (Page 8).

[Pin13] Pierre Pinson. Wind energy: Forecasting challenges for its operational management. *Statist. Sci.*, 28(4):564–585, 11 2013. (Page 8).

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. (Page 42).

[RG02] CE. Rasmussen and Z. Ghahramani. Infinite mixtures of gaussian process experts. Max-Planck-Gesellschaft, 2002. (Pages 7, 8).

[RL18] S. Rasp and S. Lerch. Neural networks for post-processing ensemble weather fore-

casts. *ArXiv e-prints*, May 2018. (Pages 23, 30).

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 09 1951. (Pages 12, 15).

[Rud16] S. Ruder. An overview of gradient descent optimization algorithms. *ArXiv e-prints*, September 2016. (Page 12).

[Sch15] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. cited By 1624. (Page 8).

[Sco15] David Scott. *Multivariate Density Estimation: Theory, Practice, and Visualization*. 03 2015. (Page 23).

[SDD98] Christian Schittenkopf, Georg Dorffner, and Engelbert Dockner. Identifying stochastic processes with mixture density networks. 07 1998. (Page 8).

[sds] Sds-011 specifications. https://nettigo.pl/attachments/398. Accessed: 2018-08-21. (Page 6).

[Sil86] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman Hall, London, 1986. (Page 23).

[SJJ96] L. K. Saul, T. Jaakkola, and M. I. Jordan. Mean Field Theory for Sigmoid Belief Networks. *eprint arXiv:cs/9603102*, February 1996. (Page 14).

[Sti75] S. M. Stigler. The transition from point to distribution estimation, 1975. (Page 8).

[SW96] A. Shapiro and Y. Wardi. Convergence analysis of gradient descent stochastic algorithms. *Journal of Optimization Theory and Applications*, 91(2):439–454, Nov 1996. (Page 15).

[tfn] Introduction to tensorflow. https://www.tensorflow.org/guide/low_level_intro. Accessed: 2018-08-21. (Page 46).

[Tim] Allan Timmermann. Density forecasting in economics and finance. *Journal of Forecasting*, 19(4):231–234. (Page 8).

[Tip04] Michael E. Tipping. *Bayesian Inference: An Introduction to Principles and Practice in Machine Learning*, pages 41–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. (Page 5).

[TKD+16] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016. (Pages 14, 42).

[Tra] Dustin Tran. Getting started. http://nbviewer.jupyter.org/github/blei-lab/edward/blob/master/notebooks/getting_started.ipynb. Accessed: 2018-08-20. (Page 48).

[US14] Prachi Uplap, , and Preeti Sharma. Review of heterogeneous/homogeneous wireless sensor networks and intrusion detection system techniques. In *Int. Conf. on*

Recent Trends in Information, Telecommunication and Computing , ITC. Institute of Doctors Engineers and Scientists, ACEEE (A Computer division of IDES), 2014. (Page 8).

[VSL00] Aki Vehtari, Simo Särkkä, and Jouko Lampinen. On mcmc sampling in bayesian mlp neural networks. 1:317–322 vol.1, 02 2000. (Page 7).

[WC07] Chun-Hsien Wu and Yeh-Ching Chung. Heterogeneous wireless sensor network deployment and topology control based on irregular sensor model. In Christophe Cérin and Kuan-Ching Li, editors, *Advances in Grid and Pervasive Computing*, pages 78–88, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. (Page 8).

[wge] Gnu wget. https://www.gnu.org/software/wget/. Accessed: 2018-08-21. (Page 43).

[Wik18] Wikipedia contributors. Ensemble forecasting — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Ensemble_forecasting&oldid=846852066, 2018. [Online; accessed 14-September-2018]. (Page 23).

[ZS14] Heiga Zen and Andrew Senior. Deep mixture density networks for acoustic modeling in statistical parametric speech synthesis. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 3872–3876, 2014. (Page 8).