Empirical Comparisons of Forecasting methods

vorgelegt von
Leonardo Araneda Freccero
geb. in Stockholm, Schweden

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Master of Science

Promotionsausschuss:
Primary Advisor: Prof. Dr. Volker Markl
Advisor: Behrouz Derakhshan
Advisor: Bonaventura Del Monte
Date: 01-03-2021

Berlin 2021

Zusammenfassung

Hier kommt der deutsche Abstrakt rein... ÜÖ sind ok.

Abstract

Put your abstract here...

Dedicated to ...

Acknowledgements

I would like to acknowledge the thousands of individuals who have coded for open-sourceprojects for free. It is due to their efforts that scientific work with powerful tools is possible.

Table of Contents

1 Introduction

Time series forecasting, the ability to predict future trends from past data, is an important tool in science, for businesses and to governments but it can be a double edged sword if the predictions are incorrect. Predicting the future is hard and predictions are only as good as the data and learning capabilities of the forecasting model used. Recently Covid-19 became a global pandemic with disastrous effects on society. Time series forecasting was used to predict the spread of the virus so that governments, hospitals and companies could plan accordingly to minimize its effects. However, many of the forecasts were overestimating the spread of the virus which lead to both organizational and health issues for hospitals, personnel and patients [1].

Improving forecasting accuracy is an active area of research and new forecasting methods are continuously being proposed [2, 3, 4, 5, 6]. As more forecasting methods become available, these need to be compared in an accurate and reproducible way. The currently most popular method for comparing forecasting methods is through evaluating the algorithm on reference datasets [7]. Oftentimes in literature, datasets used are preprocessed before being trained on in order to optimize the dataset for the algorithm at hand. Additionally, the hyperparameters of the algorithm are often tuned to fit the dataset in question. These steps are good practice as it allows the algorithm to perform optimally. However, if the method used to process the dataset is unclear in any way or if configuration details such as which hyperparameters used are missing, reproducing results becomes hard [8].

Some forecasting methods exhibit a non-deterministic behaviour where each subsequent run of the algorithm won't necessarily produce the same output. This is especially the case for deep learning algorithms as they are heavily relying on random processes such as dropout [9] or random initialization of weights. This makes it hard to reproduce results from these algorithms. Additionally, in the field of forecasting, multiple different metrics are employed in order to quantify the error of forecasts. This is due to some metrics being better suited for specific usecases. Often only a few of these metrics are used in papers when presenting new forecasting methods which makes reproducing results even harder.

In other disciplines of machine learning benchmarking suites are common tools for performing automatic reproducible comparisons between different algorithms. Some examples of these are MLBench and MLPerf. In time series forecasting however, no such benchmarking suites exist instead results from competitions such as the M competitions are held as the reference which future papers compare to.

Reproducible results imply that an algorithm needs to produce the same output no matter when some wishes to reproduce them. It should not matter if it is shortly after the algorithm was presented or a long time thereafter. This introduces difficulties as some algorithms are continuously being improved upon by their makers. If any of these improvements would change the predictive power of the algorithm this would make it impossible to reproduce any results. These types of performance changes can be handled by automated tests which ensure that the accuracy of the algorithm remains. However, defining such tests is hard for non-deterministic output and such accuracy regressions can pass through undetected [10]. Changes in predictive performance can also stem from third party updates of dependencies. These things are hard to control and potential problems are hard to foresee ahead of time.

1.1 Motivation

Previous comparisons of forecasting algorithms have found subpar performance of machine learning based approaches both in terms of accuracy and resource consumption when compared to classical methods such as Arima, ETS and Theta. However these comparisons are often made unfairly as the deep learning models which were tested were rather simple neural networks with few bells and whistles. More advanced deep learning methods such as DeepAR, DeepState, WaveNet etc. have been developed since. Thus, there is a clear benefit of thoroughly evaluating these modern algorithms in a fair and accurate way.

Accurate and fair comparisons can however only be made if the algorithms being tested all have an equal opportunity to perform well. Ensuring equal opportunity allowing forecasting methods to perform optimally is however hard and implementing a strategy for doing this is required for a large scale comparison such as this one. Additionally, the choice of error metric to use when comparing forecasting models is important as each error metrics has their own benefits and drawbacks. Using a flawed error metric can invalidate a seemingly fair comparison as some metrics are unreliable for certain applications or datasets. Implementing a strategy for when to use and not use certain error metrics is required in order to allow for fair comparisons.

Reproducibility of results is a core tenet of the scientific method. Despite this, being able to reproduce results from forecasting algorithms is not always straightforward [8]. Much of the difficulties regarding reproducibility stem from two core parts. Firstly, code and datasets are often not public and secondly the non-deterministic behaviour of certain forecasting models causes different runs of the models to produce different results.

This thesis will focus on how fair, accurate and reproducible comparisons of modern forecasting methods can be made. Particular focus will be put on how comparisons can be made in a reproducible way for non-deterministic forecasting models. As part of this thesis, a system is implemented which automates the tuning and benchmarking of forecasting models. This system is then used to perform a large scale comparison of several modern forecasting algorithms over multiple datasets.

1.2 Research Question

This thesis has as an overarching goal to compare modern forecasting models. This question is complex in itself and can be separated into three sub-questions:

1. *Accuracy*: How to perform accurate comparisons between forecasting models?
2. *Fairness*: How can one fairly compare forecasting models?
3. *Reproducibility*: How to make comparisons reproducible for forecasting models which exhibit non-deterministic behaviour?

1.3 Contribution

The main contribution of this thesis is Crayon, an open source library for benchmarking deep learning models in a fair, accurate and reproducible way. Many deep learning based forecasting algorithms are non-deterministic in nature which makes reproducing results hard. Crayon solves this by ensuring that the distribution of error metrics is reproducible. With distributions of error metrics, quantifying how good the accuracy of an algorithm is becomes a problem. To handle this, Crayon implements a custom scoring method, the "crayon score" for ranking forecasting models against eachother.

This thesis also investigates the efficiency of using various hypothesis test such as the t-test and the Kolmogorov-Smirnov test to make non-deterministic output from forecasting models reproducible. Additionally, these tests are evaluated on forecasts generated by several modern forecasting solutions on several datasets. The practical benefits of applying hypothesis testing to verify distributions of forecasts is showcased on several modern forecasting algorithms trained on several real world datasets. Further, as a practical example we show how reproducibility by hypothesis testing can protect modern forecasting solutions from suffering accuracy regressions.

An additional contribution of this thesis is the an analysis of 21 datasets with the purpose of identifying a representable subset of these to use when benchmarking forecasting methods.

The final contribution of this thesis is a large empirical comparison where the predictive performance of multiple modern forecasting algorithms is compared over four popular datasets.

2 Background

A time series is a set of data points with a clear ordering in time. For example, how the exchange rate between two currencies change over time can be seen as a time series, see figure 2. Another example of a timeseries could be how much electricity is generated by photovoltaic solar cells over time see figure 2.

or the hourly electricity consumption of households [11].

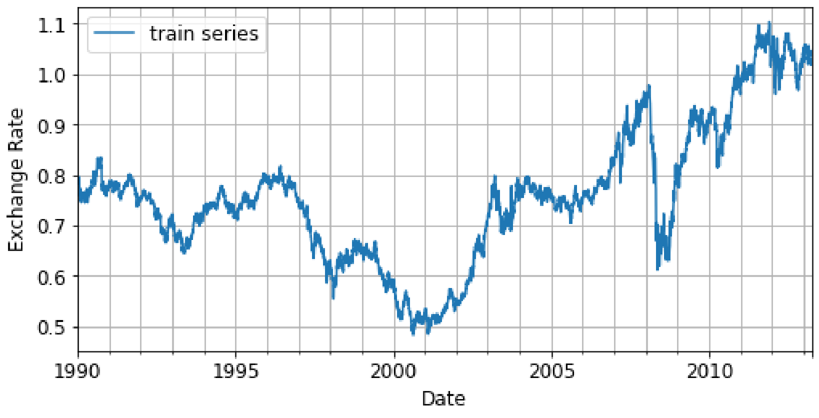


Figure 2.1: The exchange rate of two currencies between 1990 and 2013.

Being able to interpret this data to predict future trends makes it possible to do preventive maintenance or make profitable investments. There are many methods of generating such predictions, ranging from simple heuristics to machine learning based methods.

Before diving into different deep learning forecasting models in Section 2.3 it is important to understand how to quantify the accuracy of a forecasting model. Thus we will in 2.2.1 present some commonly occurring methods for calculating errors of forecasts. In 2.3 we then present some information about how deep learning models are different from classical models as well as some of the benefits and drawbacks associated with deep learning algorithms.

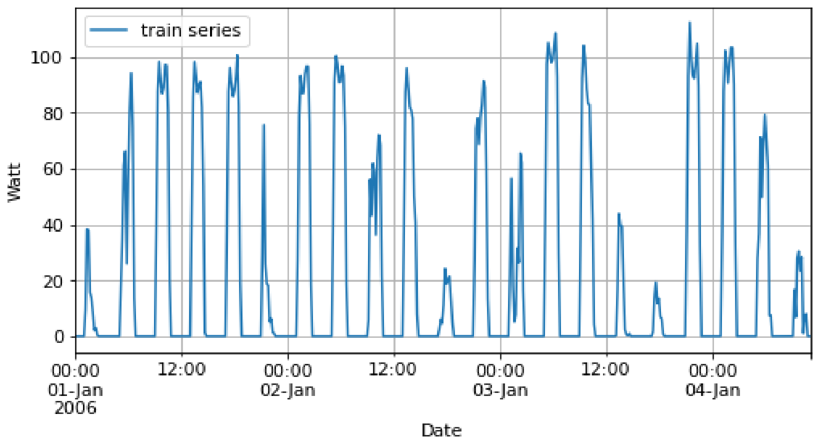


Figure 2.2: First 500 datapoints of a timeseries in the solar-energy dataset.

2.1 Time Series Forecasting

Time series forecasting is the art of predicting future trends of time series based on past observations. It is a key technology within many fields and is fundamental to the business decision process for many companies. Time series forecasting differs from common machine learning tasks such as image recognition or language processing in that predictions are not binary. More specifically, in time series forecasting a prediction is expected to be a close estimate of future values. Thus, quantifying the magnitude of the error from the ground truth is of importance in order to properly evaluate the accuracy of a forecasting model. This is commonly done by calculating an error metric, the choice of which varies depending on what is being forecasted and on preference of the forecasting practitioner. In section 2.2.1 some of these error metrics will be presented.

Timeseries forecasts often comes in one of two forms; probabilistic forecasts or point forecasts. A point forecast is a forecast which consist of only one estimation of the future values. This estimation could be either a single point in time or a series of points for several different timesteps. In figure 2.3 an example of a point forecast for several timesteps can be seen. Point forecasts has the disadvantage that they do not confer how accurate they are which may lead to prectitioners being overly confident in the forecasts. This has caused several issues historically, non the least recently with covid-19 where rellying to heavily on point forecasts led to overly drastic measures for governments and hospitals [1]. Probabilistic forecasts such as the one in 2.4 includes this information by generating prediction intervals of the forecasts.

Many methods exist for generating forecasts on timeseries data, two common groups of these are machine learning based methods and trivial methods. A trivial forecasting method would be, for example, to predict that the next value should be the same as the last recently observed value or the average of the last N observations. For certain timeseries, trivial methods perform competitively and as such these are often held as a baseline to which new forecasting methods are being compared too. Some more complex methods include those based on regression analysis such as linear regression or autoregressive models such as ARIMA [7]. Autoregressive models are a subset of regression models where the temporal ordering of the datapoints matter. I.e. autoregressive models use past observations to predict future values.

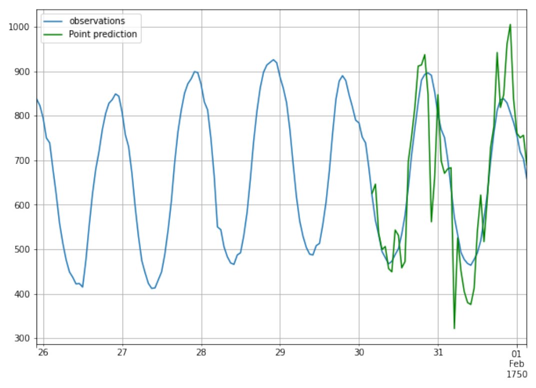


Figure 2.3: Time series point forecast

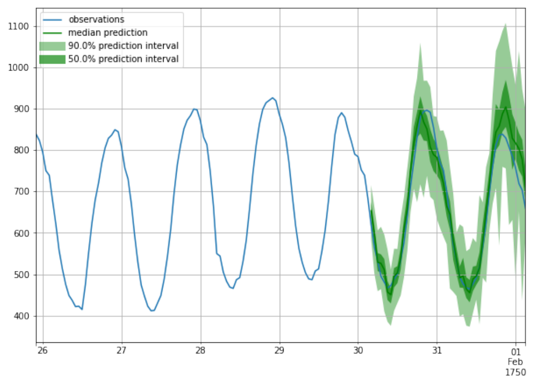


Figure 2.4: Time series forecast with probability distribution

Lately, inspired by the successful use of neural networks in other domains, several neural network based approaches for time series forecasting have been introduced. Early implementations such as simple multi-layered perceptrons did not perform competitively with the more classical approaches such ARIMA. However modern deep

Forecasting models for forecasting can also be split into further sub families such as whether they are local or global models. Local models are trained on individual time series while global models are trained on all time series in the training set. Essentially for a dataset with N timeseries, if using local models, one will have N different local models. If global models would be used only one model would be used for all N timeseries. Generally speaking, local models performs well for timeseries with much historical data. However for short or new timeseries, local models often perform poorly due to the inability to train the model on sufficiently large amounts of data points. This is known as the cold start problem. Since global models train on the entire dataset across all timeseries, any new timeseries added to the dataset suffer less from the cold start issue as the global model can use data from other timeseries as a basis for its predictions. [12]

TODO

- Univariate (i.e. temperature over time) vs Multivariate (temperature, cloud coverage, wind, ..., over time) timeseries https://www.analyticsvidhya.com/blog/2018/09/multivariate-time-series-guide-forecasting-modeling-python-codes/
- Point forecasts vs probabilistic forecasts

2.2 Evaluating forecasting performance

TODO

- Briefly explain backtesting
- Explain k-fold validation with rolling datasets
- Other methods for evaluating performance

2.2.1 Error Metrics

When a forecasting model generates predictions on a timeseries, one needs to be able to quantify the error of the prediction from true observed values. In order to do this many different error metrics exists each with its own benefits and drawbacks. Often the choice of metric is dependent on data, and business application where the prediction will be used. In this section some of the error metrics which are commonly occurring in literature are presented. These metrics are calculated automatically when performing a backtest in GluonTS, see 2.5.1 for details about this process. Note that only a subset of the metrics available in GluonTS is presented here to form a basis for subsequent chapters.

2.2.1.1 Absolute Error

The absolute error is calculated as the distance between the predicted value and the ground truth. An absolute error is positive and a value of 0 is optimal, i.e. the prediction fit perfectly. The absolute error is an intuitively simple error, thus it can easily be understood and discussed when comparing algorithms.

However intuitive the absolute error is, there is a a key issue with this error and that is that it is scale dependent. I.e. the greater the values of the dataset being predicted on, the larger the absolute error will be. This makes it impossible to compare the accuracy of an algorithm across different datasets unless they have the same scale. [**hyndman_forecasting_nodate**] Furthermore, scale dependent metrics such as these may not be suitable even for comparing between timeseries within a single dataset. Imagine a dataset such as the electricity dataset (see 2.1) where each timeseries is the electricity consumption from a household. One household may contain many people and appliances while another may be a vacation home which is seldom visited. These two timeseries would have two different scales thus making it hard to compare the accuracy of the predictions using scale dependent metrics such as the absolute error.

2.2.1.2 Root Mean Squared Error - RMSE

The root mean squared error is another scale dependent error such as the absolute error. RMSE is calculated by taking the square root of the mean of the squared error. TODO The RMSE thus punishes larger errors more than smaller errors due to it taking the power of the error before averaging. The RMSE is more complicated to interpret due to the non-linear nature, despite this, it is widely used in practise. [**hyndman_forecasting_nodate**].

2.2.1.3 Mean Absolute Percentage Error - MAPE

Percentage based errors such as the Mean Average Percentage Error normalizes the errors between 0 and 1 thus MAPE can be compared across datasets with different scales. While this is beneficial, MAPE suffer from other issues. For example MAPE becomes infinite or undefined if the ground truth is 0 for any parts of the prediction. Further issues exists for example that it penalizes negative errors more than positive errors. This has lead to variations of this metric to be created with their own pros and cons. [**hyndman_forecasting_nodate**]

2.2.1.4 Mean Average Scaled Error - MASE

The MASE metric does not scale the errors based on the dataset used. Instead it scales the error on the error of a naive baseline forecasting model. An example of a baseline model could be one which predicts the future values to be the same as the last value. While this causes the MASE metric be scale independent MASE can still behave oddly in certain scenarios. For example if the baseline model would perfectly predict a , this would cause the MASE to tend towards infinity. A MASE of 1 corresponds to that the algorithm under test is equally good as the naive model. A MASE below 1 means that the current model performs better than the naive model while a MASE above 1 means that the model performs worse than the naive baseline model. [**hyndman_forecasting_nodate**]

2.3 Deep Learning forecasting methods

Historically deep learning based methods for forecasting has been limited by the computational power available. Thus it is only recently that deep learning based methods has been used for forecasting.

- Benefits of deep learning
- drawbacks of deep learning
- Local versus global models(cold start)
- common architectures (RNN(error accumulates when recursive strategy is used), LSTM, Fully connected, Hybrid approaches)

2.3.1 Sources of randomness

- Random initialization of weights
- Random data shuffling
- Random scheduling for multiprocessors, causing random data shuffling
- Dropout

2.3.2 Handling randomness

- Fixing the seed
- Running on the same hardware
- K-fold validation
- Ensambles
- Bagging

2.3.3 Common approaches for ensuring reproducible results

- everything in 2.3.2
- public datasets
- public code

2.4 Related Work

2.4.1 Benchmarking

- Why benchmarking
- General info about benchmarking and how one would go ahead with benchmarking forecasting models.
- Specific difficulties with benchmarking deep learning models
- Discussing some papers
- Discussing another master thesis doing this

2.4.1.1 Deep learning benchmarking suites

- MLBench
- MLPerf
- DLBS

2.4.1.2 Competitions

- Basically present some info about the m competitions, what their purpose was. How they were implemented and why these are the closest thing available to a benchmarking suite.
- Maybe briefly present what Kaggle offers.

2.5 Gluon-TS

2.5.1 Overview

Gluon-TS is an open source library which includes several deep learning algorithms, datasets and tools for building and evaluating deep learning based forecasting methods. In GluonTS, an *Estimator* corresponds to the implementation of a forecasting method. An *Estimator* can be trained on a dataset in order to generate a *Predictor*. This *Predictor* can then ingest timeseries in order to produce forecasts for the future values of the timeseries.

In order to make it easy to evaluate the performance of an algorithm, GluonTS offers the *backtest_metrics* function. This function trains an *Estimator* on a train dataset, the generated

Predictor is then used to generate predictions on a test dataset. Each of the timeseries in the test dataset will have the final N datapoints removed prior to passing them to the *Predictor*. The *Predictor* then generates a prediction of length N for each of the timeseries passed to it. Error metrics (see 2.2.1) are then calculated from the prediction and the datapoints which were removed earlier. In this example, N corresponds to how many timesteps in the future one wishes to predict.

GluonTS also contains Dockerfiles which makes it easy to create docker images with installations of GluonTS inside. These images are compatible with AWS SageMaker which allows them to be executed both in the cloud as well as locally. These images calls *backtest_metrics* when run and can thus be used to generate error metrics for any *Estimator* class in GluonTS.

2.5.2 Algorithms

GluonTS offers several forecasting models which can be used to generate predictions on timeseries data below I present most of these algorithms here. Only algorithms which were well documented in GluonTS [13] or which I could explain by sifting through the source code [14] are presented below.

2.5.2.1 CanonicalRNNEstimator

This estimator is a bare bones recurrent neural network (RNN) estimator, with a single layer of LSTM cells [14]. LSTM stands for Long-Short-Term-Memory and is a cell architecture that allows RNNs to remember information from many timesteps back. An LSTM cell contains functionality which allows it to forget, ignore and store long term information. [15]

2.5.2.2 DeepFactorEstimator

The DeepFactorEstimator is presented in [12] and consists of two main parts, a global and a local model. The global model is trained across all timeseries in order to capture complex non-linear patterns between the timeseries. The local model is meant to capture the local random effects of individual timeseries. This hybrid architecture is thus able to leverage the DNN capability of learning complex patterns as well as the computational efficiency of local models. In [12] three different versions of the DeepFactorEstimator is presented, the one implemented in GluonTS is the DF-RNN version. This version uses another RNN to model the local timeseries. The DF-RNN presented in the paper performs better than Prophet and MQ-RNN on the Electricity, Taxi, Traffic and the Uber dataset both for short and long term forecasts. Some of these are presented in table 2.1 DeepAR however performed worse than DF-RNN on average but was more accurate on the short term forecasts for the Uber dataset. The Uber dataset is not available as part of GluonTS however it is mentioned here for completeness.

2.5.2.3 DeepAR

DeepAR presented in [2] is an RNN which produces probabilistic forecasts. The model learns a global representation across the timeseries and learns to identify seasonal behaviours in the data. DeepAR automatically adds certain meta information to the timeseries such as *day-of-the-week*, *hour-of-the-day*, *week-of-the-year*, *month-of-the-year*. Normally this type of meta information is added by the data scientist using the model, thus automating this procedure minimizes manual feature engineering. Another feature of DeepAR is the possibility to choose a likelihood distribution according to the data that it is to be trained on. In [2] the authors use two different distributions, the Gaussian Likelihood for real valued data and the negative binomial likelihood for postive count data. e

2.5.2.4 DeepStateEstimator

The DeepStateEstimator combines linear state space models for individual timeseries with a jointly learned RNN which is taught how to set the parameters for the linear state space models. This has the benefit that the labour intensive tuning of multiple state space models is done automatically by the RNN without sacrificing perfromance. This approach was shown to perform better than DeepAR 2.5.2.3 on the electricity dataset 2.1. Additionally, it performed better than DeepAR on 4 out of 6 metrics on the traffic dataset. Furthermore, this approach outperformed the ARIMA method and the ETS method on all the datasets **??**.

2.5.2.5 GaussianProcessEstimator

The GaussianProcessEstimator is a local model where each timeseries in the dataset is modeled by a single gaussian process. The gaussian process takes a kernel to use for the gaussian processes as a parameter. [13]

2.5.2.6 GPVAREstimator and DeepVAREstimator

The GPVAREstimator was presented as GP-Copula in this paper [6]. It is a RNN model for handling multivariate timeseries which uses a gaussian copula technique for automatic data transformation and scaling . Further, a dimensionality reduction technique is employed which minimizes the parameters needed to calculate a covariance matrix from $O(n^2)$ to $O(n)$. This allows for much larger multivariate timeseries datasets. In [6] the authors evaluated the GPVAREstimator on six datasets, all available in GluonTS see 2.1 for details about these. Additionally, the GPVAREstimator was compared against other multivariate forecasting algorithms, amongst which one was a version of the DeepAREstimator 2.5.2.3 which was altered to handle multivariate timeseries. In in GluonTS this algorithm is named DeepVAREstimator and was in the paper referred to as *vec-LSTM*.

2.5.2.7 LSTNetEstimator

The LSTNetEstimator can be seen as a hybrid approach where long term patterns of the data is captured by a neural network architecture, these long term patterns are combined with a classical autoregressive model when generating predictions. The neural network architecture consists of four parts, a CNN, a RNN a novel RNN-skip layer and a fully connected layer. The RNN-skip layer makes up for the incapability of the LSTM cells in a RNN to remember very long term information by only updating the cells periodically [16].

2.5.2.8 NBEATSEstimator and NBEATSEnsembleEstimator

In [5] a univariate deep learning model for forecasting named N-BEATS is presented. The authors of N-BEATS expressed that their goal with this algorithm was disprove the notion that deep learning models had inferior performance compared to classical models. The authors reported a higher accuracy with the N-BEATS model over all the models it was compared to.

The N-BEATS algorithm is an ensemble of multiple smaller algorithms which are all trained with different goals in mind. In [5] the ensemble consists of 180 smaller algorithms. These were optimized for different metrics (MASE, sMASE, MAPE etc.) and six different context lengths. Furthermore, bagging was used by running multiple runs of the algorithms with random initializations of the networks. A prediction of the N-BEATS model is the median value of the predictions of the algorithms in the ensemble.

A high level overview of the structure of each of the models in the ensemble is that each consists of multiple fully connected layers chained together to form blocks of networks. Each block in turn is chained together in order to form a stack. These stacks are also chained in order to form the final network. Thus this is a nested architecture consisting of fully connected layers.

GluonTS implements the N-BEATS model as the NBEATSEnsambleEstimator and the smaller algorithms in the ensemble as the NBEATSEstimator. There are some differences between the implementation in [5] and that of GluonTS. Specifically, the training data is sampled differently in GluonTS than in the paper [13].

2.5.2.9 Naive2Predictor

the Naive2Predictor is a GluonTS implementation of the Naïve 2 forecasting method used as a reference in the m4 competition [8]. The Naive2Predictor predicts the future values to be that of the last datapoint in the timeseries adjusted by some seasonality. In GluonTS this seasonality can be deduced from the frequency of the data used or by passing a custom seasonality via the *season_length* parameter [13].

2.5.2.10 NPTSPredictor

The NPTSPredictor predicts future values by sampling from previous data in the timeseries. The way that the samples are selected from the previous data can be modified via the hyperparameters. One can sample uniformly across all previous values in the timeseries. One can also bias the sampling to more often sample from more recent datapoints by choosing an exponential kernel. Additionally it is possible to only sample values from seasonal values [13].

2.5.2.11 ProphetPredictor

Prophet is a nonlinear regression model developed at Facebook which frames the timeseries forecasting problem as a curve fitting exercise [7]. This is different from many other forecasting models for timeseries which leverage the temporal aspect of timeseries when forecasting. Prophet was created with three goals in mind; it should be easy to use for people without much knowledge about timeseries methods, it should work for many different forecasting tasks which may have distinct features. Finally it should contain logic which makes it easy to identify how good the generated forecasts of Prophet are [17].

2.5.2.12 RForecastPredictor

The RForecastPredictor is a wrapper which allows a user of GluonTS to use the popular forecasting package *forecast* inside of GluonTS. The forecaster to use inside of the R package can be selected by passing the name of the method as a hyperparameter[13, 18].

2.5.2.13 SeasonalNaivePredictor

The SeasonalNaivePredictor is a naive model which predicts the future value to be the same as the value of the previous season. This is a very simple model however an example explains its functionality best. If I want to use the SeasonalNaivePredictor to forecast the average temperature of June. The SeasonalNaivePredictor will return the average temperature for last year in June. In GluonTS, if not enough data exists (i.e. no data for last year in June) the mean of all the data is returned[13, 7]

2.5.2.14 MQCNNEstimator, MQRNNEstimator

The GluonTS seq2seq package contains two forecasting algorithms, MQ-CNN and MQ-RNN, as presented in [19] as two implementations of their proposed MQ framework. The MQ framework is based on the Seq2Seq architecture[20] which consists of an encoder and a decoder network. A Seq2Seq architecture encodes the training data into a hidden state which the decoder network then decompresses. Normally in Seq2Seq architectures, the RNN models tend to accumulate errors as the forecasts of an RNN for a timepoint t will be reused in order to generate a forecast for time *t+1*. By instead training the model to generate multiple point forecast for each timepoint in the horizon one wishes to forecast on, the errors tend to grow smaller. This is called *Direct Multi-Horizon Forecasting* and it is one of the changes introduced in the MQ framework. Further, the MQ framework allows for different encoders to used. Two of these are implemented in GluonTS, one with a CNN and one with a RNN. These are the MQCNNEstimator and the MQRNNEstimator respectively [13].

2.5.2.15 SimpleFeedForwardEstimator

The SimpleFeedForwardEstimator in GluonTS is a traditional Multi Layer Perceptron (MLP) which can be dynamically resized based on user parameters. Per default it has two densely connected layers with 40 nodes in the input layer and 40 times the prediction length number of cells in the hidden layer. Furthermore it has an added layer for performing probability based predictions instead of only point predictions. This layer consists of a number of sublayers equal to the desired prediction length. Each of these layers consists of 40 nodes (i.e. the size of the output layer). After each hidden layer an optional batch normalization layer can be attached [14].

2.5.2.16 TransformerEstimator

The transformer estimator is a sequence to sequence model which replaces the classical RNN encoder and decoders with more parallelizable NN components. A RNN requires data to passed sequentially in order to learn dependencies between datapoints. This introduces a bottleneck and makes RNNs harder to train efficiently on modern hardware accelerators such as GPUs. The Transformer architecture presented in [21] alliviates this by leveraging parallelizable architectures such as feed forward networks as well as making heavy use of attention. Attention means that the architecture automatically can identify which parts of the input data is most relevant to the value we are trying to predict[21]. I.e. for the next value we are predicting, the most relevant previous values may be any or all of the previous n observations. In GluonTS the Transformer is implemented under as the TransformerEstimator. [13]

2.5.2.17 Wavenet

Based on the PixelCNN algorithm. Wavenet synthesizes speech on the waveform layer. Wavenet has many techniques allowing for extremely deep networks without unreasonable execution overhead.

2.5.3 Datasets

GluonTS offers 17 datasets ready to be used for training and evaluation, in table 2.1 these are presented.

2.6 Runtool

2.6.1 Config files

2.6.2 Defining experiments

2.7 Statistical tests for distributions

2.7.1 T-test

TODO present t test briefly

2.7.2 Welch's T-Test

TODO present Welchs T-test briefly

2.7.3 Kolmogorov-Smirnov Test

TODO present ks test

2.7.4 Continous Kolmogorov-Smirnov test

TODO this is more of an optimization as the ks-test seem to work just fine

Name	Freq	Description
Electricity	Hourly	Hourly electricity consumption of 370 clients sampled between 2011 - 2014. The original dataset on UCL was sampled each 15 minutes whilst the dataset in GluonTS had the data resampled into hourly series [13, 6].
Exchange Rate	B	Daily currency exchange rates of eight countries; Australia, Britain, Canada, China, Japan, New Zealand, Singapore and Switzerland between 1990 and 2016 [16].
Traffic	H	This dataset contains the hourly occupancy rate of 963 car lanes of the San Francisco Bay are freeways [14].
Solar Energy	10 Min	The solar power production records in the year of 2006, sampled every 10 minutes from 137 solar energy plants in Alabama [16].
Electricity NIPS	H	The Electricity dataset with additional processing [6].
Exchange Rate NIPS	B	The Exchange Rate dataset with additional processing [6].
Solar Energy NIPS	H	The Solar Energy dataset with additional processing [6].
Traffic NIPS	H	The Traffic dataset with additional processing [6].
Wiki Rolling NIPS	D	The Wiki dataset contains the amount of daily views for 2000 pages on Wikipedia [6].
Taxi	30 Min	Number of taxi rides taken on 1214 locations in New York city every 30 minutes in the month of January 2015. The test set is sampled on January 2016 [6].
M4 Hourly	H	Hourly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [8].
M4 Daily	D	Daily timeseries used in the M4 competition randomly sampled from the ForeDeCk database [8].
M4 Weekly	W	Weekly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [8].
M4 Monthly	M	Monthly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [8].
M4 Quarterly	3M	Quarterly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [8].
M4 Yearly	Y	Yearly timeseries used in the M4 competition randomly sampled from the ForeDeCk database [8].
M5 Dataset	D	Daily Walmart sales for 3049 products across 10 stores [14, 22].

Table 2.1: Datasets available in GluonTS.

2.7.5 Dvoretzky–Kiefer–Wolfowitz inequality

This will probs be discussed as part of KS test instead

2.8 Dataset characteristics

2.8.1 STL - decomposition

2.8.2 Seasonality

2.8.3 Trend

2.8.4 Stationarity

3 Approach

3.1 Overview

-

3.2 Ensuring reproducible results

The ability to reproduce findings is a core tenet of the scientific method. Deep learning often has some randomness associated to it which makes reproducing results more difficult. In 3.2.1 we define reproducibility for deep learning. Section 3.2.2 dives deeper into how distributions of error metrics can be used to ensure reproducibility of experiments.

3.2.1 Defining reproducibility

As discussed in 2.3.2 deep learning models often suffer from random fluctuations in their accuracy due to inherent random processes of their architectures. This makes it hard to accurately and reproducibly compare the accuracy of different algorithms.

A common approach to solve this randomness is to fix the random seed prior to training the algorithm. While this does ensure that a specific configuration of the model is reproducible it does not necessarily ensure that the behaviour of an algorithm in general is reproducible.

By not fixing the seed we will instead receive a distribution of errors for a given hyperparameter configuration. This distribution provides insights into what range of accuracies one could expect for a specific algorithm running on a specific dataset. I.e. is it heavy tailed, is it wide, does it follow a normal distribution etc. By instead comparing distributions of algorithm errors we can more generally verify if two algorithms perform the same. Thus by our definitions, if two algorithms perform the same they are, logically equivalent in performance.

It is somewhat common that code needed for running algorithms are provided by by their authors. However, it is unfortunately not always the case that the code is frozen at the time of publication. I.e. additional changes may have been added to the algorithm since its publication. This complicates matters as different implementations of the same algorithm may behave differently (gluonts bug on deepar). Furthermore, different implementations of the same algorithm may have different parameters exposed/different internal architecture thus possibly resulting in different results on the same workload (SageMaker DeepAR vs GluonT DeepAR).

Due to this I believe that persistent artifacts such as Docker images containing algorithms are needed for true reproducibility. In 2.6 I present a purpose built tool which through a domain specific language allows defining experiments on algorithms in Docker images. By sharing the docker image, a configuration file, the dataset and a small python script reproducible runs of an algorithm can be made.

To summarize, in this thesis, reproducibility is defined as being able to verify whether two independent sets of training jobs are sampled from the same distribution. Furthermore, for strong reproducibility I propose that algorithm artifacts such as Docker images are used in conjunction with a reproducible way of running them.

3.2.2 A look on distributions

Suppose that I would publish a report detailing that on error metric X on dataset D it receives a value of E. This approach is analogous to saying that "this method with these specific parameters outputs this value". By limiting the representation of the algorithm to this extent one is unable to capture additional characteristics of the algorithm such as how consistent its errors are.

Some randomness cannot be controlled by setting the seed such as OS related scheduling of threads. This can effect the performance of ML models using multi-threading for fast calculations (SEE SECTION XYZ FOR MORE). If this kind of error occurs, it may be hard to reproduce results.

TODO look through the paragraphs below and bind them together better.

Running an algorithm on a dataset N times would result in a set of N error metrics being produced. These errors would be samples taken from a distribution of the errors this specific algorithm would have on this specific dataset. One could imagine that an algorithm which generally performs well on this dataset would have an error distribution with a peak close to the metrics optimum value (i.e. 0 for MASE, MAPE, AE etc.). Furthermore, algorithms with a higher variance in their results, should logically result in a wider distribution.

In the following sections I will discuss possible approaches to solve the following questions:

1. How can one verify that two sets of error metrics; N and M are sampled using the same forecasting algorithm on a specific dataset.

2. How accurate is the method in terms of false positives.

3. What sample size does it need to work sufficiently well. TODO define sufficiently well...

4. How does it perform empirically

3.3 Enabling fair and accurate comparisons

3.3.1 Equal tuning effort

When machine learning algorithms are employed in real life these algorithms are tuned thoroughly on the data it should predict on.

In order to allow for fair comparisons between models, it is required that the algorithms has been tuned on the dataset in question. Performing tuning in a fair way is hard as there are many strategies one could employ to ensure a fair tuning effort. However none of these strategies are flawless and thus, some algorithms may be better tuned using one strategy than another.

One strategy which could be employed to ensure fair tuning could be to fix the total training time available for tuning. This would be fair in the sense that each algorithm would have the same time to find good hyperparameters. However this would give an unfair advantage to fast algorithms. Further, even the programming language used to develop a system could bias these results. I.e. the programming language C generally executes faster than for example Python.

Another strategy could be to set a fixed amount of parameter combinations to test. This ensures that the search space of the parameters are equally traversed for different algorithms. However it is not fair in the sense that the search space of some algorithms hyperparameter configurations are larger than others. Thus an algorithm with two hyperparameters would be more comprehensively tuned than an algorithm with hundreds of parameters.

3.3.1.1 User tuned

3.3.1.2 Automated tuning

3.3.2 Metrics

3.4 Designing a benchmarking system for forecasting methods

3.4.1 Choosing datasets

3.4.2 Handling reproducibility

3.4.2.1 Naive approach

Random noise follows a normal distribution if enough samples are taken. Thus if the noise added to the errors is purely random noise, it follows that the distributions of the error metrics should be normally distributed. TODO verify/come with examples

If this is the case, then one should be able to detect if N is sampled from the same distribution as M by comparing the mean of N with the mean of M +- some standard deviations of M. This approach has the benefit of being simple and intuitive.

4

Designing a benchmark - Analysis and implementation

4.1 Dataset Analysis

It is common that new forecasting methods are compared on several datasets as to showcase their predictive power in different scenarios. Different datasets exhibit different characteristics such as trend and seasonality. By identifying a representable set of datasets with complementary characteristics, one can evaluate how robust a forecasting algorithm is to different datasets. Choosing a representable subset of datasets is also needed as the time to train an algorithm increases linearly with the amount of datasets that it should be trained on. For a empirical comparison such as this, it unfeasible to run training and tuning jobs for all algorithms on all datasets.

4.1.1 Methodology

For each dataset in GluonTS I generated a plot of the average time series along with one standard deviation from it. This was done in order to get a overview of what the datasets looked like. Plotting timeseries for this purpose is common practice in time series forecasting, or as Hyndaman Et al so eloquently put in Forecasting: Principles and practice: *"The first thing to do in any data analysis task is to plot the data."*.[7]

Another common tool for visualising timeseries is to generate a histogram of the timeseries. However as the datasets available in GluonTS contains tens of thousands of timeseries, plotting each of these individually is unfeasible. Thus for each dataset I aggregate the timeseries values so that it can be be displayed in a single plot. Instead of generating histograms I choose to generate violin plots as these capture both the distribution of values just as histograms do as well as the median value and the interquartile range. However as each timeseries in a dataset is on its own scale, the aggregate violin plot becomes hard to read. By scaling each timeseries in the dataset by the maximum value of that timeseries before plotting it the violin plot becomes more easy to read.

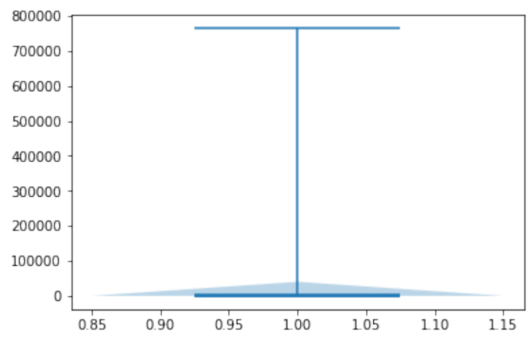


Figure 4.1: Unscaled violin plot of the electricity dataset

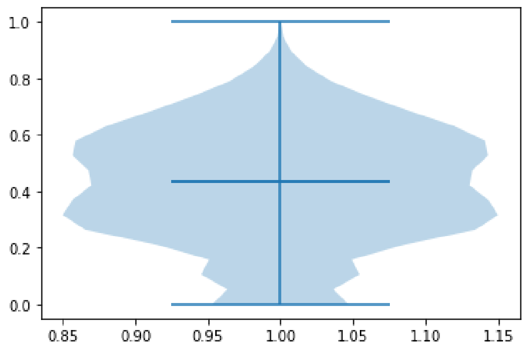


Figure 4.2: Scaled violin plot of the electricity dataset where all timeseries have been scaled by their maximum value.

In addition to these visual tools I calculate the following statistics for each of the datasets in GluonTS:

- Mean value
- Max value
- Min value
- Number of timeseries
- Number of datapoints
- Length of shortest timeseries
- Length of longest timeseries
- Strength of the trend (mean and standard deviation)
- Strength of the seasonality (mean and standard deviation)

Simple metrics such as the maximum, mean and minimum values for a dataset are useful in order to get a high level overview of the dataset. Furthermore, simpler statistics such as these can identify possible issues which can surface further down the line. For example, the metric used when evaluating forecasting methods can become unstable for some values in the dataset. One such example is the MAPE metric, which become unstable for timeseries close to 0 [7].

The number of timeseries are important for global models as these then have more data which they can learn across which can help to combat overfitting. The number of timeseries is however not important for local models as these are only impacted by the quality and length of each individual time series. Longer timeseries however, benefit forecasting models which can handle longer horizons [8]. For these reasons, the number of timeseries and the lengths of the shortest and longest timeseries are necessary metrics when comparing datasets. Optimally one would have both long timeseries and many series as this would benefit both global and local models, however this is not always possible.

In chapter **??** we dove into seasonality and trend and how these describe the stationarity of timeseries. The strength of the seasonality and the trend of a timeseries can be calculated by: TODO add the equation here [7]. The trend and the seasonality X and Y in equation Z can be extracted through STL decomposition. In my case I used the STL method of the python Statsmodel package. In order to apply this to an entire dataset, the strengths need to be calculated for each individual timeseries. Thereafter, the average value and standard deviation is recorded. Complimentary datasets should exhibit different values of strength for trend and seasonality. Additionally, the variance of these strengths are important as a higher variance indicates that the timeseries within a dataset are very mixed regarding the strength of trend and seasonality.

Four types of datasets could be identified as a representable subset of datasets:

1. One dataset with high trend and low seasonality
2. One dataset with low trend and high seasonality
3. One dataset with low trend and low seasonality
4. One dataset with high trend and high seasonality

4.1.1.1 Limitations

The M4 dataset does not significantly differ when compared to the m3 dataset except for its size [23] due to this, and that the m3 dataset has been largely superseeded by the m4 dataset, the m3 dataset was early discarded as the dataset of choice for this comparison. Another set of datasets which are not going to be used is the, the NIPS datasets. This is due to them having had some postprocessing applied to them. As the exact post processing is not known, comparing any findings when training on these datasets with other papers becomes hard and introduces uncertainty. In order to avoid this, these datasets are not considered.

4.1.2 result

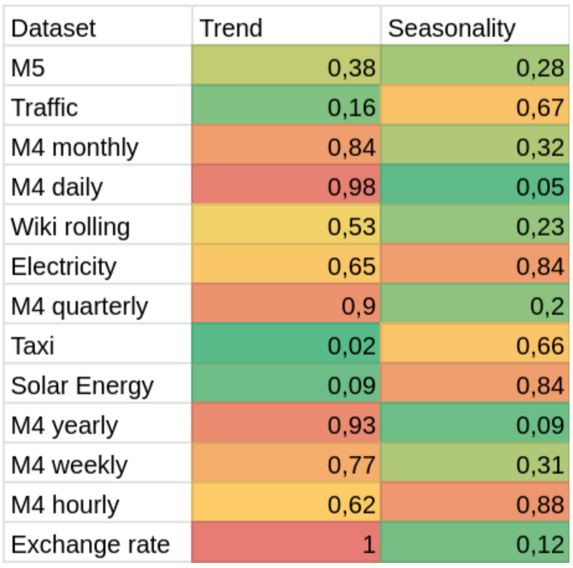
The complete plots and statistics are available in the appendix as they were to numerous to be displayed here. However a summary of the extracted statistics is presented in XXX.

The strength of the trend and seasonality can be easily viewed by generating a heatmap of the values as can be seen in figure 4.1.2.

There is only one dataset in 4.1.2 which exhibits low trend and low seasonality and that is the m5 dataset. However the m5 dataset does show the most variance of all the datasets which implies that it contains many timeseries with and without trend and seasonality.

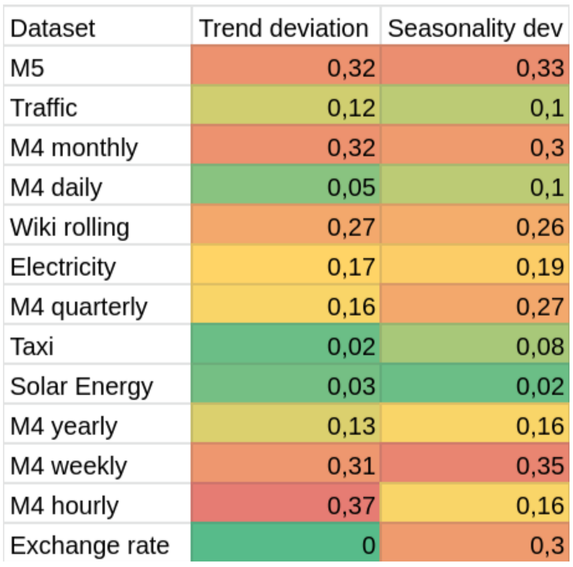
In order to choose a dataset which fits into the category of high seasonality and low trend there are three options; Solar Energy, Taxi and Traffic. Of these the Solar Energy dataset has the lowest variance for both the trend and the seasonality. Further the solar energy dataset shows the highest seasonality of them all.

The datasets with the lowest seasonality and the highest trends are the m4 datasets except for the m4 hourly and the Exchange rate dataset. Of these the one with the lowest variance as well as the highest trend is the Exchange Rate dataset closely followed by the m4 daily dataset. The size of the Exchange Rate dataset is much smaller than the m4 Daily dataset, with only 8 timeseries and 48k datapoints in comparison to the 4227 series and 9.9M datapoints.



Dataset	Trend	Seasonality
M5	0,38	0,28
Traffic	0,16	0,67
M4 monthly	0,84	0,32
M4 daily	0,98	0,05
Wiki rolling	0,53	0,23
Electricity	0,65	0,84
M4 quarterly	0,9	0,2
Taxi	0,02	0,66
Solar Energy	0,09	0,84
M4 yearly	0,93	0,09
M4 weekly	0,77	0,31
M4 hourly	0,62	0,88
Exchange rate	1	0,12

Figure 4.3: Heatmap of the strength of the trend and the seasonality sorted by the size of the datasets.



Dataset	Trend deviation	Seasonality dev
M5	0,32	0,33
Traffic	0,12	0,1
M4 monthly	0,32	0,3
M4 daily	0,05	0,1
Wiki rolling	0,27	0,26
Electricity	0,17	0,19
M4 quarterly	0,16	0,27
Taxi	0,02	0,08
Solar Energy	0,03	0,02
M4 yearly	0,13	0,16
M4 weekly	0,31	0,35
M4 hourly	0,37	0,16
Exchange rate	0	0,3

Figure 4.4: Heatmap of the standard deviation of the strength of the trend and the seasonality for the datasets.

When it comes to finding a dataset which both has a high trend and a high seasonality, there are only two options, the Electricity dataset and the m4 hourly dataset. They both have a high variance, however the variance of the Electricity dataset is slightly lower than that of m4 hourly. In addition, the Electricity dataset has almost twice the amount of datapoints as m4 hourly.

To summarize the four datasets which most closely fit the criteria defined in 4.1.1 are:

	Low trend	**High Trend**
Low seasonality	M5	m4 daily
High Seasonality	Solar Energy	Electricity

Table 4.1: Datasets with complimentary strength of trend and strength of seasonality

4.2 Compairing statistical tests for reproducibility

In this section the tests are compared in order to answer the questions in 3.2.2. Each of the tests are executed on data collected over 300 runs of DeepAR for three different hyperparameter configurations on the Electricity dataset 4.8. Thus the statistical tests will be evaluated on in total 900 datapoints. In statistical tests are compared across a couple of different criteria.

- Can it accurately identify that two sets of samples are from the same distribution when one of the sets is small.
- Does it have a high false error rate, i.e. does it often mistake to identify samples from the same distribution.
- Does it accurately detect when two distributions has different shape.
- How sensitive is it, i.e. can it detect differences in visually similar distributions.

4.2.1 Methodology

4.2.2 Results

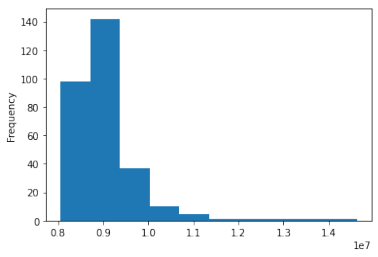


Figure 4.5: Student-T

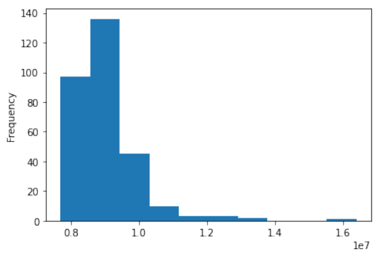


Figure 4.6: Neg-Binomial

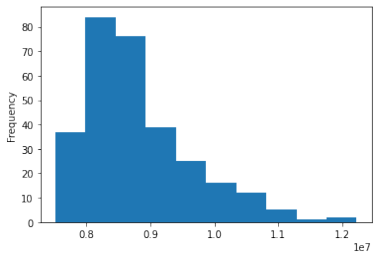


Figure 4.7: Poisson

Figure 4.8: Histograms of the absolute error for 300 runs of DeepAR on the Electricity dataset The three plots are generated through 300 runs of DeepAR on the Electricity dataset with three different values of the hyperparameter *distr_output* see section 2.5.2.3 for more info

As can be seen in 4.8 the distribution of the absolute error is skewed and more closely reflects a lognormal distribution than a normal distribution. This contradicts the key assumption made for the T-Test, Welch-s T-test and the naive approach. However as will be shown, these still perform well in the scenarios we consider below.

TODO write this using some neat mathematical notation. Given a small sample N, how does the different methods compare when evaluating if the sample is from the same distribution of another larger sample M? To answer this, I ran each test 297 times (0 <= |M| < 300 - |N|). N contained the 3 data points not in M. M contained 0, 1, ..., 297 samples of the 300 runs. The results are presented in 4.3.

Name	**Errors**	**%**	**Runs**	**configuration**
Naive	52	17%	297	negative binomial
Naive	62	20%	297	student t
Naive	47	15%	297	poisson
T-Test	24	8%	297	negative binomial
T-Test	19	6%	297	student t
T-Test	4	1%	297	poisson
Welchs T-Test	43	14%	297	negative binomial
Welchs T-Test	39	13%	297	student t
Welchs T-Test	23	7%	297	poisson
Kolmogorv-Smirnov	15	5%	297	negative binomial
Kolmogorv-Smirnov	18	6%	297	student t
Kolmogorv-Smirnov	3	1%	297	poisson

Table 4.2: False positive rates of statistical tests for one small sample and one larger

As can be seen in 4.3 the naive approach have a higher error rate, i.e. it reports that the two distributions are sampled from different distributions more often when compared to the other approaches. The Kolmogorov Smirnov statistic performs the best with the lowest amount of errors out of all the tests.

Calculating the general false positive rates is done similarly as above, however instead of fixing the size of N, it will contain random samples of the remainder after M is sampled from the 300 datapoints. I.e. tests will be applied for all combinations of samples where $0 < |M| < 300$ and $0 < |N| <$ 300-$|M|$

Name	Errors	%	Runs	configuration
Naive	241	<1%	106900	negative binomial
Naive	313	<1%	106900	student t
Naive	366	<1%	106900	poisson
T-Test	5489	5%	106900	negative binomial
T-Test	5893	5%	106900	student t
T-Test	34869	32%	106900	poisson
Welchs T-Test	12334	11%	106900	negative binomial
Welchs T-Test	9037	8%	106900	student t
Welchs T-Test	33971	31%	106900	poisson
Kolmogorv-Smirnov	4250	3%	106900	negative binomial
Kolmogorv-Smirnov	5658	5%	106900	student t
Kolmogorv-Smirnov	23417	21%	106900	poisson

Table 4.3: False positive rates of statistical tests

5 Crayon - A time series forecasting benchmark

In this section I introduce Crayon, a toolkit for benchmarking ML forecasting models. Crayon contains a set of tools for making fair, accurate and reproducible comparisons between forecasting models. Crayon is available on GitHub under the XXX license. TODO

In 3.2, I argue that in order to achieve proper reproducibility it is necessary that algorithms need to be trained and evaluated on datasets multiple times in order to build up a distribution of error metrics. Crayon is built with this in mind, and the benchmarking functionality described in Section 5.7 makes use of the distribution of errors when benchmarking. In its core, Crayon is built on three core techniques; Runtool configuration files for defining reproducible training jobs, Docker images containing the algorithm to benchmark and the datasets that the algorithm is to be benchmarked on. In the following sections the core functionality of crayon is presented.

5.1 Architecture

TODO

5.2 Algorithms

In crayon, the *Algorithms* class contains all the information needed in order to execute an algorithm contained in a docker image either on SageMaker or on the local machine. *Algorithm* objects are used internally in Crayon and they are a basic building block for performing hyperparameter tuning and for creating runtool configuration files.

An *Algorithm* object contains information about the *image* to run, the *hyperparameters* to use. If the algorithm should be executed on SageMaker, the specific instance it should be run on can also be set using the *instance* parameter. The *Algorithm* class also optionally accepts a dictionary of regexes to the *metrics* parameter. These regexes are used by SageMaker to extract the any metrics from an algorithms output. Default regexes for GluonTS algorithms are available in the *utils* module of Crayon. In Listing 5.1 an example of how one can use the *Algorithm* class is presented.

```python
from crayon import Algorithm
Algorithm(
    name="myalgo",
    image="my_image",
    instance="local",
    hyperparameters={...},
    metrics={"abs_error": ... }
)
```

Listing 5.1: Example of the Algorithm class.

5.3 Datasets

The Crayon Dataset class wraps all the data needed to locate a dataset on your local machine or on AWS S3. There is only two required parameters when creating an object of the Dataset class; the *name* of the dataset & the *path* leading to the dataset.

The *path* is a dict which points to the file location(s) of the dataset. If one has a train, test and validation dataset, the items in the *path* will point to these versions of the dataset. Any *meta* information one wish to provide about the dataset in question can be passed through the *meta* parameter. In 5.2 an example of how to use the Dataset class is presented.

```python
from crayon import Dataset
Dataset(
    name="electricity",
    path={
        "train": "file:///datasets/electricity/train/data.json",
        "test": "file:///datasets/electricity/test/data.json",
    },
    meta={...}
)
```

Listing 5.2: Example of the Dataset class.

5.4 Config generation

This section discusses the different tools available in Crayon in order to create runtool compatible config files. These config files are a core part of the Crayon functionality and is used in several internal systems. Thus it is of importance to get an understanding of how these are generated and how these files look like. For further information about how the config files function and additional functionality of them, see the section about the runtool (2.6).

Through providing an *Algorithm* object & a *Dataset* object to the *generate_config*, Crayon can generate a config file compatible with the runtool. The intention of this file is that by providing the config, the image used and the dataset to another person, that person could rerun your experiments with the exact same configuration. Internally Crayon uses the *generate_config* function in order to create tuning and benchmarking training jobs.

This function accepts an *Algorithm* and a *Dataset* object as presented in 5.2 and 5.3. In 5.3 an example of how a config can be generated using Crayon is presented.

```python
from crayon import Algorithm, Dataset, generate_config

# generate runtool configuration file
config = generate_config(
    Algorithm(
        name="myalgo",
        image="my_image",
        hyperparameters={"epochs": 10},
    ),
    Dataset(
        name="my_ds",
        path={
            "train": "/datasets/electricity/train/data.json",
            "test": "/datasets/electricity/test/data.json",
        },
    ),
    "/Users/freccero/Documents/config.yml",
)
```

Listing 5.3: Config generation using Crayon

After running the code above, a config file is stored to the path provided to *generate_config*, in this case the config file is stored to */Users/freccero/Documents/config.yml*. An example of the config file which would be generated by the code in 5.3 is displayed in Listing 5.4

```yaml
my_ds:
  meta: {}
  name: my_ds
  path:
    test: file:///datasets/electricity/test/data.json
    train: file://datasets/electricity/train/data.json
myalgo:
  hyperparameters:
    epochs: 10
  image: my_image
  instance: local
  name: myalgo
```

Listing 5.4: Generated configurations file.

If one does not have a custom dataset but instead wishes to use a reference dataset for training and evaluating the algorithm on, it is possible to use the datasets in GluonTS. The *get_gluonts_datasets* function downloads any number of GluonTS datasets onto the local machine and generates a list of Datasets objects which can be passed to the *generate_config* function. Example code for doing so is shown in 5.5 with the output of the code displayed in 5.6.

```python
from crayon import Algorithm, Dataset, generate_config, get_gluonts_datasets
import yaml

```

```python
# generate runtool configuration file
config = generate_config(
    Algorithm(
        name="myalgo",
        image="my_image",
        hyperparameters={"epochs": 10},
    ),
    get_gluonts_datasets(["electricity"]),
    "/Users/freccero/Documents/config.yml",
)

print(yaml.dump(config))
```

Listing 5.5: Config generation using Crayon with gluonts datasets

```yaml
electricity:
  meta:
    feat_dynamic_cat: []
    feat_dynamic_real: []
    feat_static_cat:
    - cardinality: '321'
      name: feat_static_cat
    feat_static_real: []
    freq: 1H
    prediction_length: 24
    target: null
  name: electricity
  path:
    test: file:///Users/freccero/.mxnet/gluon-ts/datasets/electricity/test/
   data.json
    train: file:///Users/freccero/.mxnet/gluon-ts/datasets/electricity/train/
   data.json
myalgo:
  hyperparameters:
    epochs: 10
  image: my_image
  instance: local
  metrics: null
  name: myalgo
```

Listing 5.6: Output of 5.5

5.5 Training

Given a config file Crayon can use this file to start training jobs either locally or on AWS SageMaker. The provided config file can be either written by hand or generated using Crayon as described in 5.4. Crayon uses the runtool to read config files and dispatch training jobs.

A training job is started by calling the *crayon.run_config* method. *crayon.run_config* has two required parameters, the config that is to be used along with which algorithm-dataset combination to run. The experiment is defined using the mathematical operators + and * as described in section 2.6.

5.5.1 Jobs

The *crayon.run_config* function returns a *Jobs* object. A *Jobs* object makes it easy to investigate training results as it appends any metrics reported by the algorithms across all started jobs to a pandas DataFrame. This makes it easier to analyse the recorded metrics. Furthermore, each item in a *Jobs* object contains all the information required to rerun the job with the exact same configuration.

5.5.2 Local Training

If a config is to be run locally, algorithms in the config needs to point to an image which is accessible locally on the users machine. For details of the image requirements, see section TODO. Furthermore, any datasets which the algorithm should be trained on should also be available on the local machine.

In listing 5.7 a configuration file which can be used for local training is displayed. In 5.8 code for training the algorithm on the dataset using this config is presented.

```yaml
my_algo:
    image: gluonts:cpu_new
    hyperparameters:
        freq: H
        prediction_length: 24
        forecaster_name: gluonts.model.deepar.DeepAREstimator
    instance: local

my_dataset:
    path:
        train: file:///datasets/electricity/train/data.json
        test: file:///datasets/electricity/test/data.json
```

Listing 5.7: Config file for running local training jobs

```python
from crayon import run_config

jobs = run_config(
    config="config.yml",
    combination="config.my_algo * config.my_dataset",
)
```

Listing 5.8: Code for running local training jobs using Crayon

By executing the code in 5.8 Crayon will use the runtool to load the config in 5.7 and start training jobs locally through the *local_run* method of the runtool. The jobs will be executed sequentially. After the jobs finish, Crayon looks for a file named *agg_metrics.json* or *metrics.json* in the output directory for each job. This file should contain any metrics reported by the algorithm being trained. An example json file is shown in 5.9 for reference.

```json
{
    "MSE": 6703174.123610994,
    "abs_error": 37402126.36717987,
    "MASE": 1.62229213532052,
    "MAPE": 0.23736356524438723
}
```

Listing 5.9: Example of a training job output file such as *metrics.json* or *agg_metrics.json*

5.5.3 Cloud training

Starting training jobs in the cloud is done similarly as when starting them locally, however some changes needs to be made to the config file as well as to the python script.

Config file changes

The config file needs to refer to an image available in ECR of the AWS account that is to be used. Further, the *instance* in the config file must match one of the instance types which SageMaker offers. Any metrics which the algorithm outputs during training such as accuracy metrics or similar must have a corresponding regex under the metrics tag in the config file. This regex is used by SageMaker to parse the algorithm output for metrics. The final change to the config which needs to be done for running on SageMaker is that any datasets needs to be stored in an AWS S3 bucket instead of locally.

Python script changes

In addition to the above changes to the config file, three further parameters has to be passed to the *run_config* function in the Python script.

1. *role_arn* - The AWS IAM Role which authorizes Crayon to start training jobs on SageMaker.
2. *bucket* - The AWS S3 bucket where any artifacts of the training job should be stored.
3. *session* - A *boto3.Session* object which Crayon will use to interact with SageMaker.

The IAM Role requires permissions to start training jobs on sagemaker, pull AWS ECR images, and read/write access to the provided S3 bucket. For information about how to create a IAM Role with the proper permissions as well as an S3 bucket please refer to the AWS documentation for these two services. Boto3 is a python package which makes it easy to interact with AWS services from Python code, please also refer to the boto3 documentation for further information.

In Listing 5.10, an example configuration file is displayed which can be used to train on AWS SageMaker. In 5.11 an example python file is presented which given a config dispatches training jobs to SageMaker using the *run_config* function of Crayon.

```
my_algo:
  image: 012345678901.dkr.ecr.eu-west-1.amazonaws.com/gluonts:2020-11-01
  hyperparameters:
    freq: H
    prediction_length: 24
    forecaster_name: gluonts.model.deepar.DeepAREstimator
  instance: ml.m5.xlarge
  metrics:
    abs_error: 'abs_error\): (\d+\.\d+)'

```

```
my_dataset:
  path:
    train: s3://gluonts-run-tool/gluon_ts_datasets/constant/train/data.json
    test: s3://gluonts-run-tool/gluon_ts_datasets/constant/test/data.json
```

Listing 5.10: Config file for running training jobs on SageMaker.

```
from crayon import run_config
import boto3

jobs = run_config(
    config="config.yml",
    combination="config.my_algo * config.my_dataset",
    role_arn="arn:aws:iam::012345678901:role/service-role/my_role",
    bucket="my_bucket",
    session=boto3.Session(),
)
```

Listing 5.11: Code for running training jobs on AWS SageMaker using Crayon

5.6 Tuning

In order to be able to use Crayon as a benchmarking tool, it is advised that each algorithm has been suitably tuned to the datasets which it should run on. This is required since comparing a finely tuned model with an undertuned model will result in unfair comparisons being made. As discussed in 3.3.1 it is not easy to ensure that different models receive a fair tuning effort without introducing biases towards certain properties of the algorithms.

Grid search is a common strategy used when tuning algorithm. When tuning using grid search, one creates for each hyperparameter a list of values which that hyperparameter can take. Thereafter the algorithm is then run for each combination of these parameters and the hyperparameter combination which yielded the best results is then returned as the optimal hyperparameter combination to use.

Below I present how an algorithm can be tuned using the grid-search functionality in Crayon.

5.6.1 Grid-Search

Crayon offer grid search functionality through the *grid_search* function in the *Crayon.Tuner* module. An overview of the features of the *grid_search* module is displayed below:

- Performs grid search with static and changing parameters
- Each hyperparameter combination can be rerun multiple times to handle non-deterministic algorithms
- Allows passing a custom aggregation function in order to customize how multiple runs of a combination should be merged. (i.e. should the average be taken, should the mode be taken or the minimum?)

- Allows passing of a custom scoring function for handling different optimizations. (i.e. for MASE a lower error is better while for some other metric this may not hold true.)
- Generates runtool configuration files for each combination of hyperparameters.

In 5.12 the signature for the *grid_search* method of Crayon is displayed. Each of the parameters of this method will here be presented and in 5.13 an example of how performing grid search locally is presented. In 5.14 an example of how one could execute the grid search using SageMaker is shown.

Grid Search function overview

```python
def grid_search(
    changing_hyperparameters: dict,
    target_metric: str,
    dataset: Dataset,
    algorithm: Algorithm,
    output_dir: str = crayon_dir().resolve(),
    aggregation_function: Callable = statistics.mean,
    evaluation_function: Callable = lambda new, old: new < old,
    run_locally: bool = True,
    **kwargs,
) -> GridSearchResults:
```

Listing 5.12: Parameters of the grid search functionality in Crayon.

changing_hyperparameters are the hyperparameters which are to be tuned. This parameter takes a dictionary where the keys are the hyperparameters to tune and the value is a list of values which the hyperparameter should take. The *target_metric* parameter selects which metric that the algorithm outputs should be optimized for. Further, the grid search takes a *crayon.Dataset* and a *crayon.Algorithm* (see 5.2 & 5.3). Any hyperparameters defined for the algorithm stays the same for each run in the training loop. All configuration which are run will generate a configuration file usable with Crayon for subsequent runs. The *output_dir* determines where these files will be stored.

The *aggregation_function* will be used to merge the results if each configuration should be run more than once. For example, if you wish to tune an algorithm and you wish to run it N times. The *aggregation_function* converts these N recorded metrics into a single value. The default way that this is done is through simply taking the mean of the recorded values. Through passing a custom function to the *aggregation_function* parameter one can override this behaviour. This function needs to take a list of numbers as a parameters and it needs to return a single value.

The *evaluation_function* is used to determine whether the results of a specific combination performed better than the best run so far. The default behaviour defines that if the new value of the target metric is lower than the best so far, it is better. This makes sense whenever one wishes to minimize the target metric, i.e. an absolute error of 10 is better then an absolute error of 100. However for other target metrics, different behaviour may be more suitable. by providing a function which takes two parameters, the new value and the old value and compares these as a the *evaluation_function* parameter one can alter this behaviour. The *run_locally* parameter determines if the jobs should be executed on the local machine or on AWS SageMaker. Any additional parameters will be passed to the *crayon.run_config* function upon starting the training jobs.

Grid search examples

In the example presented in 5.13 we perform grid search on an image containing gluonts. The algorithm being tuned in GluonTS is the DeepAREstimator with the hyperparameters *freq* and *prediction_length* being the same for each training job. The DeepAREstimator should be optimized by minimizing the average value of the abs_error metric over two runs for each combination. The hyperparameters that generate the grid to search over are the *epochs* and the *context_length*.

```
grid_search(
        algorithm=Algorithm(
            name="deepar",
            image="gluonts_cpu",
            hyperparameters={
                "freq": "D",
                "prediction_length": 7,
                "forecaster_name": "gluonts.model.deepar.DeepAREstimator",
            },
        ),
        dataset=Dataset(
            name="electricity",
            path={
                "train": "file:///datasets/electricity/train/data.json",
                "test": "file:///datasets/electricity/test/data.json",
            },
        ),
        runs=2,
        target_metric="abs_error",
        aggregation_function = statistics.mean,
      evaluation_function = lambda new, old: new < old,
        changing_hyperparameters={
          "epochs": [1, 5],
          "context_length": [1, 5],
      }
    )
```

Listing 5.13: Grid search running locally.

By setting the *run_locally* parameter to False we tell Crayon to execute the training on SageMaker instead of locally. In order to make this work, we need to provide the an AWS S3 bucket, an AWS IAM role and a boto3 Session object. See section 5.5.3 for more information about these. An example of how grid search can be executed on SageMaker is presented in 5.14.

```
from crayon import GLUONTS_METRICS, Algorithm, Dataset, grid_search
import boto3

grid_search(
        role_arn="arn:aws:iam::012345678901:role/service-role/my_role",
        bucket="my_bucket",
        run_locally=False,
        session=boto3.Session(),
        algorithm=Algorithm(
            name="deepar",
            image="012345678901.dkr.ecr.eu-west-1.amazonaws.com/gluonts/cpu",
            hyperparameters={
                "freq": "D",
                "prediction_length": 7,
                "forecaster_name": "gluonts.model.deepar.DeepAREstimator",
            },
            metrics=GLUONTS_METRICS,
            instance="ml.m5.xlarge",
        ),
        dataset=Dataset(
            name="electricity",
            path={
                "train": "s3://gluon_ts_datasets/electricity/train/data.json",
                "test": "s3://gluon_ts_datasets/electricity/test/data.json",
            },
        ),
        runs=2,
        target_metric="abs_error",
        aggregation_function = statistics.mean,
        evaluation_function = lambda new, old: new < old,
        changing_hyperparameters={
          "epochs": [1, 5],
          "context_length": [1, 5],
      }
    )
```

Listing 5.14: Grid search running on SageMaker.

5.7 Benchmarking

One of the key components of Crayon is the benchmarking module. This module allows algorithms to be benchmarked on a couple of datasets. (TODO write which datasets it is being run on.) One can use crayon to benchmark an algorithm by providing a config file containing the algorithm. Furthermore, one must pass which target metric should be the target for the benchmark, i.e. absolute error, MASE or anything else which the algorithm outputs. Crayon will then run the algorithm on each dataset 20 times in order to build up a distribution of the target metric. Thereafter, a score is calculated summarizing the distribution of errors into a single value. This is done in order to rank the algorithm against any other benchmarked algorithms. The score is calculated as follows TODO Decide how to calculate the score. Probably using the mode will work best.

As the benchmark should run on several different datasets, it is important to provide tuned models for each of these datasets. One can do so by naming the algorithms in the config file using the structure: *<algorithm_name>_<dataset_name>*. In 5.15 an example config which can be used for benchmarking is presented. Note that there are two hyperparameter configurations of DeepAR in the config, this indicates that different hyperparameters will be used for different datasets.

```yaml
deepar:
    name: deepar
    image: gluonts:cpu_new
    hyperparameters:
        epochs: 1
        freq:
            $eval: $trial.dataset.meta.freq
        prediction_length:
            $eval: 2 * $trial.dataset.meta.prediction_length
        context_length:
            $eval: 2 * $trial.algorithm.hyperparameters.prediction_length
        forecaster_name: gluonts.model.deepar.DeepAREstimator
    instance: local

deepar_electricity:
    $from: deepar
    hyperparameters:
        epochs: 2
```

Listing 5.15: Config file for benchmarking using Crayon. Note that *deepar_electricity* has a different hyperparameter configuration thus these hyperparameters will be used when benchmarking the algorithm on the electricity dataset.

```python
from crayon import benchmark

benchmark(
    algorithm_config="config.yml",
    algorithm_name="deepar",
    target_metric="MASE",
)
```

Listing 5.16: Python script for starting a benchmarking run using Crayon.

After all the training jobs has finished the recorded metrics of the run is stored to a file on the local machine. This file also contains the *id* of the benchmark, as well as the config used to create the training jobs, and the *Jobs* object which was generated from the benchmark. Essentially everything one would need in order to rerun the exact experiment as well as the recorded results is stored in this file. In the future, this should arguably be handled by some centralized storage server instead however this is out of the scope of the thesis.

A benefit of this file is that it can be used to verify that an algorithm behaves the same further down the line i.e., it can be used to detect if the behaviour of an algorithm is the same or if it changed since the benchmark was run. How this can be done using Crayon will be discussed in section 5.8.

5.8 Verifying results

In 3.2 I delved into how one can statistically detect whether two distributions of values behave the same, i.e. if they are sampled from the same underlying distributions. Crayons verification module is designed based on the results and definitions made in that chapter. Lets walk through the use of the verification module using an example:

User A has created a machine learning algorithm and documents the performance of the algorithm in a paper. In the paper, User A also provides the link to a git repository containing the algorithm. Some time later, user B tries to reproduce the findings of user A. However, user B is unable to achieve the same accuracy as was reported in the paper of user A even though the same code was executed. Now the question is, was the accuracy recorded in the paper false or, was it a lucky run of the algorithm? Perhaps user A continued working on the algorithm since the paper was presented, and introduced some changes which caused the accuracy of the algorithm to change? Or maybe the hyperparameters used by User B are not the same as those used by user A?

Traditionally it would be hard to verify the results of user A in this scenario. Theoretically, if user A would have run the algorithm several times in order to build up a distribution of error metrics which he then would have shared. User B could have compared the distributions that she observed with those that User A shared. However, sharing the raw data for several runs of algorithms is rarely done in practice and even less so in a machine readable format.

The benchmarking module of Crayon generates these distributions and includes tooling for sharing and verifying them. Any benchmarks made using Crayon will have their results stored into a *results.yml* file on the local machine. Crayon can use the data in this file in order to verify that two different benchmarks are sampled from the same distribution. Given a *results.yml* file, Crayon can also rerun the benchmark with the exact same hyperparameter configuration. Since this *results.yml* file can be shared and compared across several installations of Crayon it can easily be added as an artifact to a git repository. Let us now show how the two users in the above example could have used Crayon to make the results of User A reproducible.

Assuming that user A would have provided her local *results.yml* file as part of the git repository. User B could then run a local benchmark using Crayon to generate a local *results.yml* file. Thereafter user B could use Crayon to verify that the new benchmarks behave the same as what was reported in the source paper. Thus, even though user B is unable to reproduce the exact same values as user A, user B is able to show that the distribution reported by user A is still the same and thus that algorithm behaves the same. Note that User B does not need to know anything about which hyperparameters was used for the benchmark as these are stored in the *results.yml* file which was shared by User A.

In 5.17 example code for verifying if two benchmarks are of the same algorithm is presented. As noted in the section presenting the benchmarking functionality each benchmark was given an *id* which is unique within a *results.yml* file. This *id* is used to select which benchmarks should be compared within one *results.yml* file.

If one wishes to compare different *results.yml* files, the paths to the two *result.yml* files can be passed to the *verify_benchmarks* function along with the banchmark *id* that should be compared in each file. If only local benchmarks are to be compared, the paths to the *result.yml* files can be omitted.

```python
from crayon import verify_benchmarks

verify_benchmarks(
    id_1="2021/01/25/22-09-04",
    id_1_results_file="result_1.yml",
    id_2="2021/01/25/19-50-43",
    id_2_results_file="result_2.yml",
    target_metric="MASE",
)
```

Listing 5.17: Python code to verify whether two benchmarks are of the same algorithm.

Internally Crayon would load the two *results.yml* files, extract the results of the benchmarks and then perform the Kolmogorov-Smirnov test in order to compare the two distributions. If the p-value reported by the Kolmogorov-Smirnov test would be less than *0.05*, the test fails and thus the two samples are with a 95% confidence not sampled from the same distribution.

Note that we cannot detect if the algorithm used is the exact same one but only if the behaviour of the two algorithms is the same.

5.9 Examples

5.9.1 Generating Gluon-TS artifacts

Generating docker images containing GluonTS algorithms can easily be done by running docker build on suitable docker files in the GluonTS repository. After an image is built one can access the different

5.9.2 Catching regressions of algorithms in gluon-ts

5.9.3 Running benchmarks of three different algorithms

6 Empirical comparison of forecasting methods

6.1 Evaluation

7 Evaluation

TODO write some good discussion here

8 Discussion

TODO write some good discussion here

References

[1] John P.A. Ioannidis, Sally Cripps, and Martin A. Tanner. “Forecasting for COVID-19 has failed”. In: *International Journal of Forecasting* (2020). ISSN: 0169-2070. DOI: `https://doi.org/10.1016/j.ijforecast.2020.08.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0169207020301199`.

[2] David Salinas, Valentin Flunkert, and Jan Gasthaus. “DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks”. In: *arXiv:1704.04110 [cs, stat]* (Feb. 22, 2019). arXiv: `1704.04110`. URL: `http://arxiv.org/abs/1704.04110` (visited on 06/23/2020).

[3] Syama Sundar Rangapuram et al. “Deep State Space Models for Time Series Forecasting”. In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 7785–7794. URL: `http://papers.nips.cc/paper/8004-deep-state-space-models-for-time-series-forecasting.pdf`.

[4] Aaron van den Oord et al. “WaveNet: A Generative Model for Raw Audio”. In: *arXiv:1609.03499 [cs]* (Sept. 19, 2016). arXiv: `1609.03499`. URL: `http://arxiv.org/abs/1609.03499` (visited on 08/02/2020).

[5] Boris N. Oreshkin et al. “N-BEATS: Neural basis expansion analysis for interpretable time series forecasting”. In: *arXiv:1905.10437 [cs, stat]* (Feb. 20, 2020). arXiv: `1905.10437`. URL: `http://arxiv.org/abs/1905.10437` (visited on 08/02/2020).

[6] David Salinas et al. “High-Dimensional Multivariate Forecasting with Low-Rank Gaussian Copula Processes”. In: *arXiv:1910.03002 [cs, stat]* (Oct. 24, 2019). arXiv: `1910.03002`. URL: `http://arxiv.org/abs/1910.03002` (visited on 08/02/2020).

[7] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice.* 3nd edition. Melbourne, Australia: OTexts. URL: `https://otexts.com/fpp3/` (visited on 03/06/2021).

[8] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. “The M4 Competition: 100,000 time series and 61 forecasting methods”. In: *International Journal of Forecasting* 36.1 (Jan. 2020), pp. 54–74. ISSN: 01692070. DOI: `10.1016/j.ijforecast.2019.04.014`. URL: `https://linkinghub.elsevier.com/retrieve/pii/S0169207019301128` (visited on 06/23/2020).

[9] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014). Publisher: JMLR.org, pp. 1929–1958. ISSN: 1532-4435.

[10] github user: jsirol. *DeepAR + NegativeBinomialOutput performance degradation after upgrading from v0.4.2 to 0.5.0.* URL: `https://github.com/awslabs/gluon-ts/issues/906`. (accessed: 26.06.2021).

[11] Alexander Alexandrov et al. "GluonTS: Probabilistic Time Series Models in Python". In: *arXiv:1906.05264 [cs, stat]* (June 14, 2019). arXiv: `1906.05264`. URL: `http://arxiv.org/abs/1906.05264` (visited on 06/23/2020).

[12] Yuyang Wang et al. "Deep Factors for Forecasting". In: *arXiv:1905.12417 [cs, stat]* (May 28, 2019). arXiv: `1905.12417`. URL: `http://arxiv.org/abs/1905.12417` (visited on 08/02/2020).

[13] *GluonTS - Probabilistic Time Series Modeling.* GluonTS - Probabilistic Time Series Modeling. URL: `https://ts.gluon.ai/` (visited on 02/23/2021).

[14] *GluonTS - GitHub repository.* GluonTS - Probabilistic Time Series Modeling. URL: `https://github.com/awslabs/gluon-ts` (visited on 02/23/2021).

[15] Alex Sherstinsky. "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network". In: *Physica D: Nonlinear Phenomena* 404 (Mar. 2020), p. 132306. ISSN: 01672789. DOI: `10.1016/j.physd.2019.132306`. arXiv: `1808.03314`. URL: `http://arxiv.org/abs/1808.03314` (visited on 09/24/2020).

[16] Guokun Lai et al. "Modeling Long- and Short-Term Temporal Patterns with Deep Neural Networks". In: *arXiv:1703.07015 [cs]* (Apr. 18, 2018). arXiv: `1703.07015`. URL: `http://arxiv.org/abs/1703.07015` (visited on 08/02/2020).

[17] Sean J Taylor and Benjamin Letham. *Forecasting at scale.* preprint. PeerJ Preprints, Sept. 27, 2017. DOI: `10.7287/peerj.preprints.3190v2`. URL: `https://peerj.com/preprints/3190v2` (visited on 06/23/2020).

[18] *forecast package in R.* URL: `https://pkg.robjhyndman.com/forecast/` (visited on 02/23/2021).

[19] Ruofeng Wen et al. "A Multi-Horizon Quantile Recurrent Forecaster". In: *arXiv:1711.11053 [stat]* (June 28, 2018). arXiv: `1711.11053`. URL: `http://arxiv.org/abs/1711.11053` (visited on 08/02/2020).

[20] Alex Graves. "Generating Sequences With Recurrent Neural Networks". In: *CoRR* abs/1308.0850 (2013). arXiv: `1308.0850`. URL: `http://arxiv.org/abs/1308.0850`.

[21] Ashish Vaswani et al. "Attention is All you Need". In: (), p. 11. URL: `https://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf`.

[22] Spyros Makridakis et al. "The M5 Uncertainty competition: Results, findings and conclusions". In: (Nov. 2020).

[23] Evangelos Spiliotis et al. "Are forecasting competitions data representative of the reality?" In: *International Journal of Forecasting* (Dec. 2018). DOI: `10.1016/j.ijforecast.2018.12.007`.

A Appendix

A.1 Example code

A.2 Dataset Plots and statistics

A.2.1 Electricity

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	2510.68	321	6755124	21044	21044	0.0	764000.0	1H
test	2509.92	2247	47501580	21068	21212	0.0	764000.0	1H

Table A.1: Statistics of the Electricity dataset.

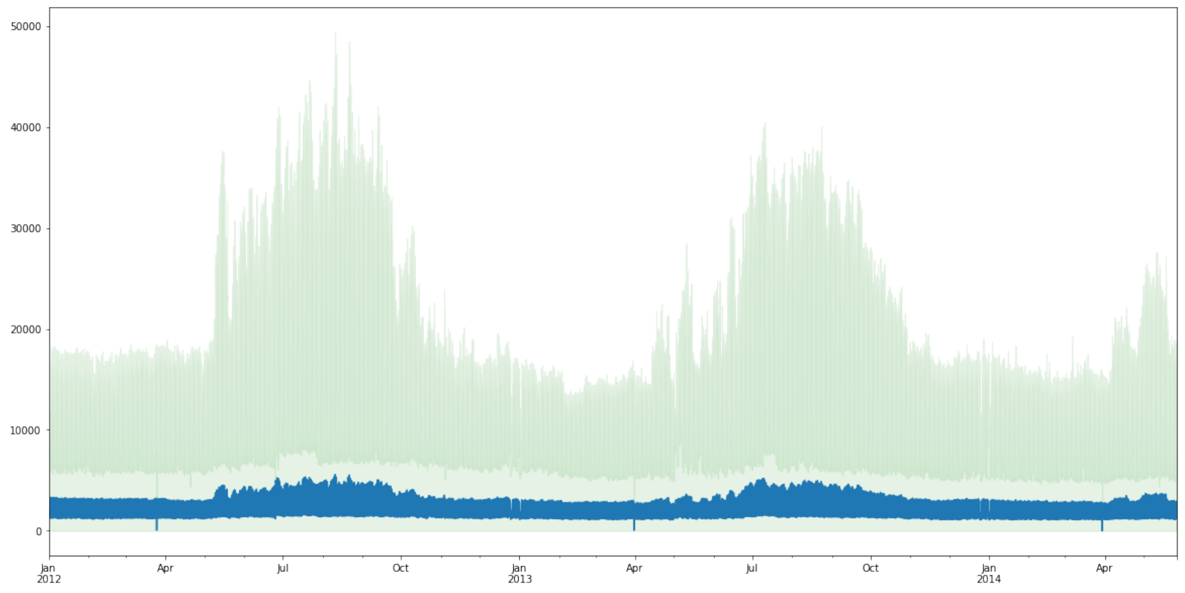


Figure A.1: Plot over the average timeseries in the electricity dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

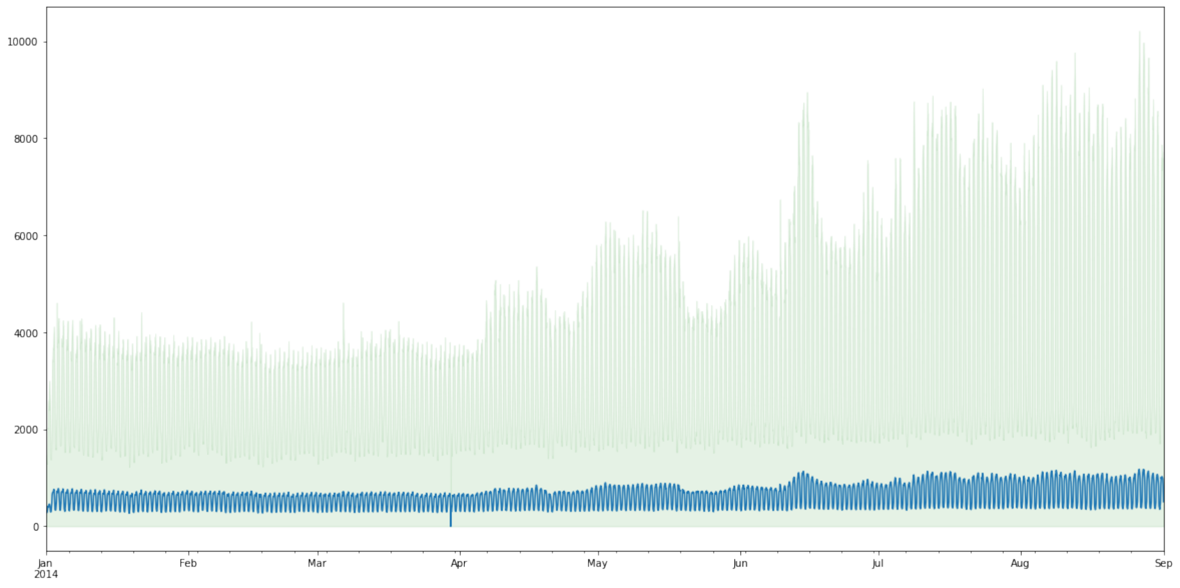


Figure A.2: Zoomed version of A.1

statistic	mean	deviation	max	min
trend	0.65	0.17	1.0	0.09
seasonality	0.84	0.19	1.0	0.0

Figure A.3: Strength of trend and seasonality of the Electricity dataset

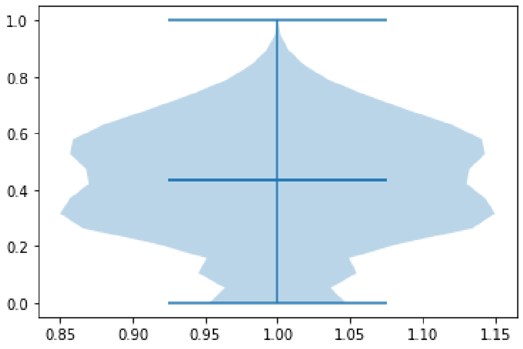


Figure A.4: Scaled violin plot of the electricity dataset.

A.2.2 Exchange Rate

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	0.68	8	48568	6071	6071	0.01	2.11	1B
test	0.68	40	246440	6101	6221	0.01	2.11	1B

Table A.2: Statistics of the Exchange Rate dataset

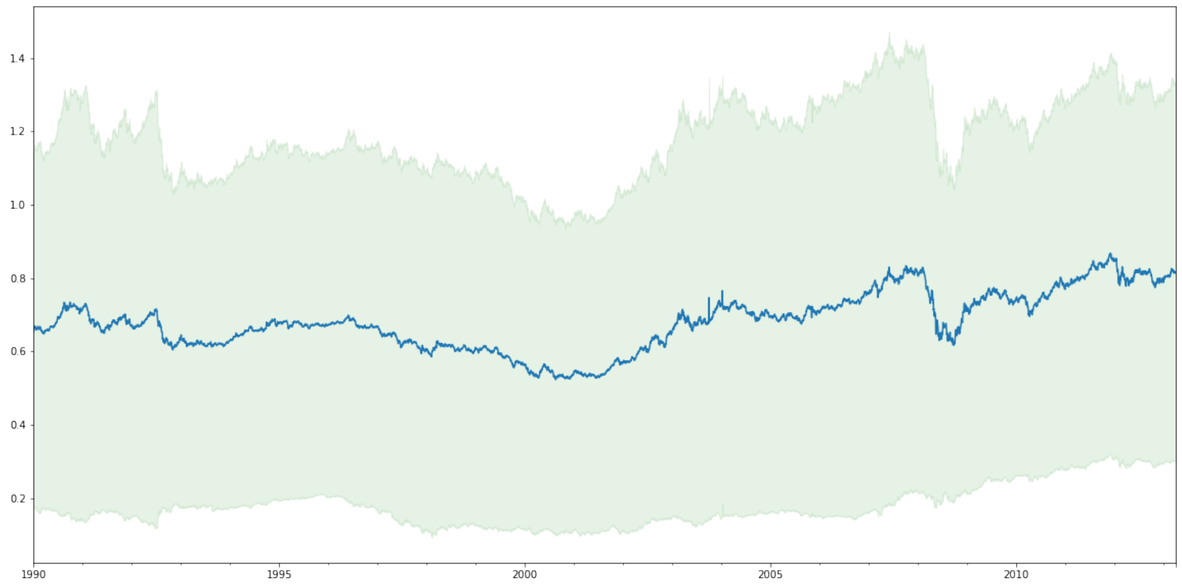


Figure A.5: Plot over the average timeseries in the exchange rate dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	1.0	0.0	1.0	0.99
seasonality	0.12	0.30	0.90	0.0

Figure A.6: Strength of trend and seasonality of the exchange rate dataset

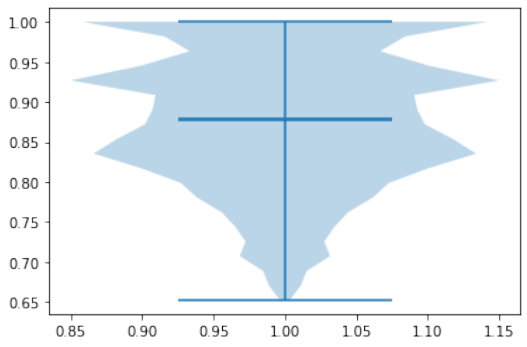


Figure A.7: Scaled violin plot of the exchange rate dataset.

A.2.3 Solar Energy

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	40.35	137	960233	7009	7009	0.0	509.05	10min
test	40.25	959	6813695	7033	7177	0.0	509.05	10min

Table A.3: Statistics of the Solar Energy dataset

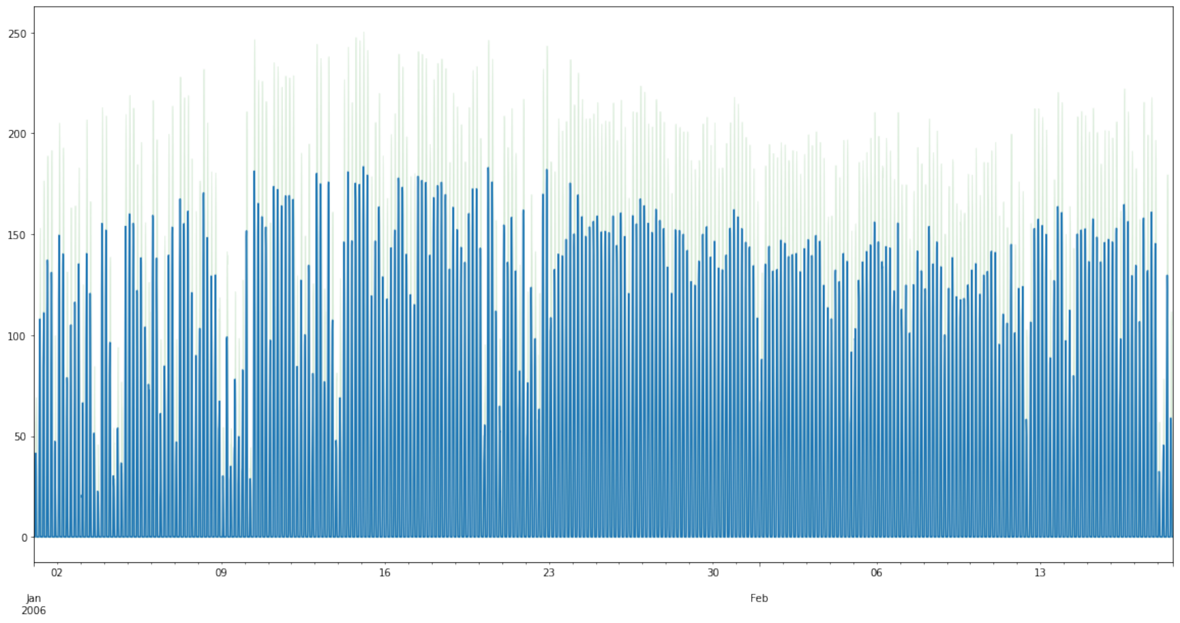


Figure A.8: Plot over the average timeseries in the solar energy dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.09	0.03	0.15	0.0
seasonality	0.84	0.02	0.87	0.79

Figure A.9: Strength of trend and seasonality of the solar-energy dataset

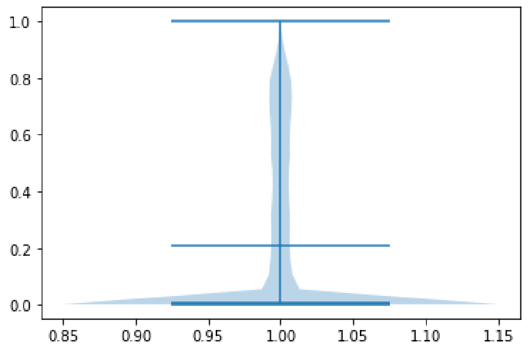


Figure A.10: Scaled violin plot of the solar energy dataset.

A.2.4 Traffic

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	0.06	862	12099032	14036	14036	0.0	0.72	H
test	0.06	6034	85272488	14060	14204	0.0	0.72	H

Table A.4: Statistics of the Traffic dataset

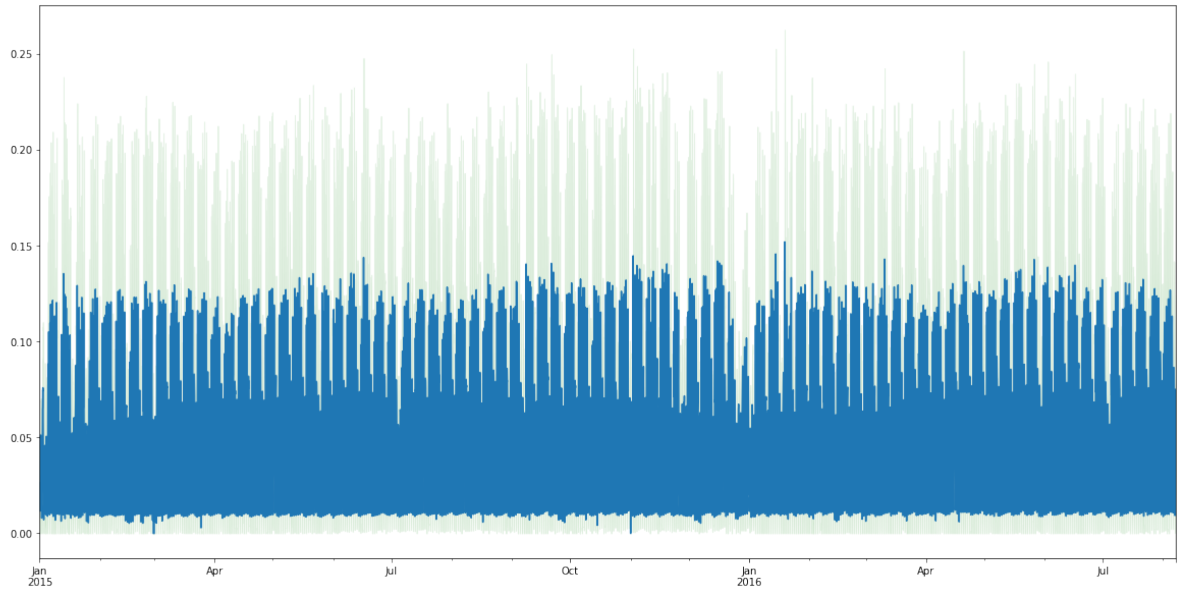


Figure A.11: Plot over the average timeseries in the traffic dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.16	0.12	0.79	0.0
seasonality	0.67	0.10	0.93	0.12

Figure A.12: Strength of trend and seasonality of the traffic dataset

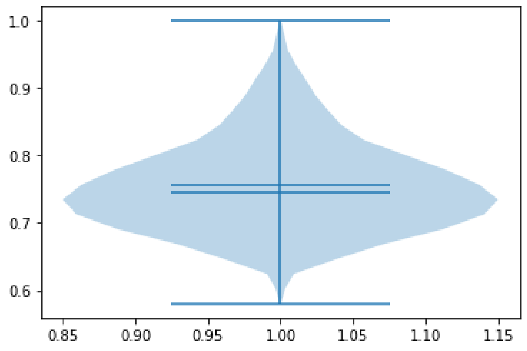


Figure A.13: Scaled violin plot of the traffic dataset.

A.2.5 Exchange Rate NIPS

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	0.68	8	48568	6071	6071	0.01	2.11	B
test	0.68	40	246440	6101	6221	0.01	2.11	B

Table A.5: Statistics of the Exchange Rate NIPS dataset

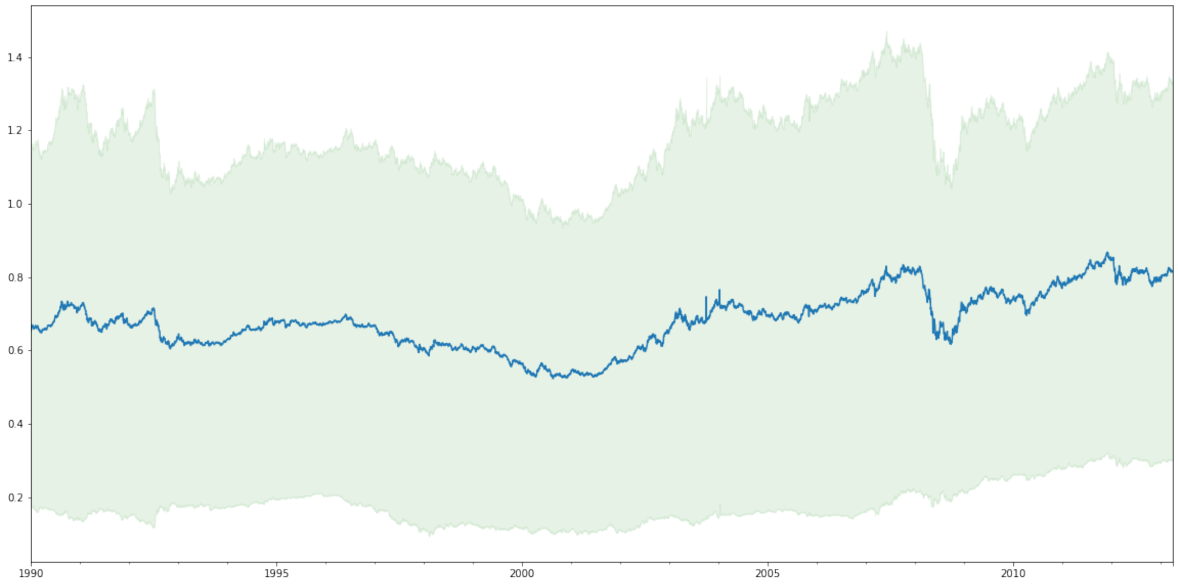


Figure A.14: Plot over the average timeseries in the exchange rate nips dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	1.0	0.0	1.0	0.99
seasonality	0.12	0.30	0.90	0.0

Figure A.15: Strength of trend and seasonality of the exchange rate nips dataset

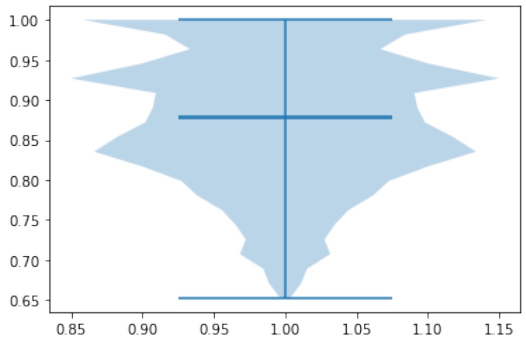


Figure A.16: Scaled violin plot of the exchange rate nips dataset.

A.2.6 Electricity NIPS

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	607.95	370	2142282	1081	5833	0.0	168100.0	H
test	652.36	2590	10340239	1105	4000	0.0	168100.0	H

Table A.6: Statistics of the Electricity NIPS dataset.

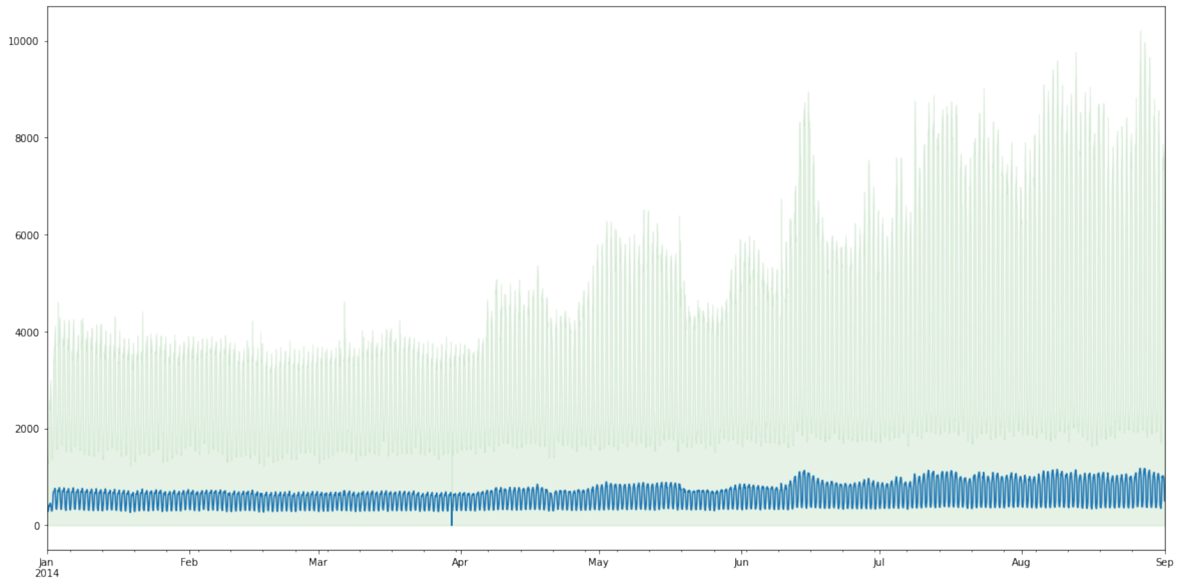


Figure A.17: Plot over the average timeseries in the electricity nips dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.54	0.20	1.0	0.0
seasonality	0.86	0.16	1.0	0.0

Figure A.18: Strength of trend and seasonality of the electricity nips dataset

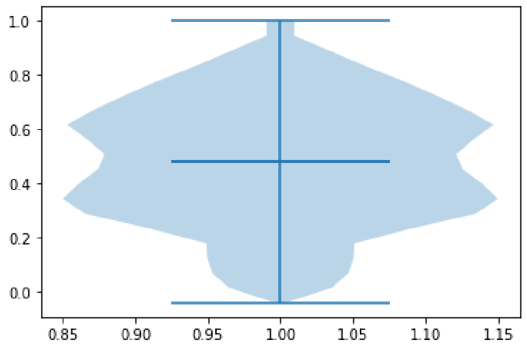


Figure A.19: Scaled violin plot of the electricity nips dataset.

A.2.7 Solar Energy NIPS

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	40.35	137	960233	7009	7009	0.00	509.05	H
test	40.25	959	6813695	7033	7177	0.00	509.05	H

Table A.7: Statistics of the Solar Energy NIPS dataset

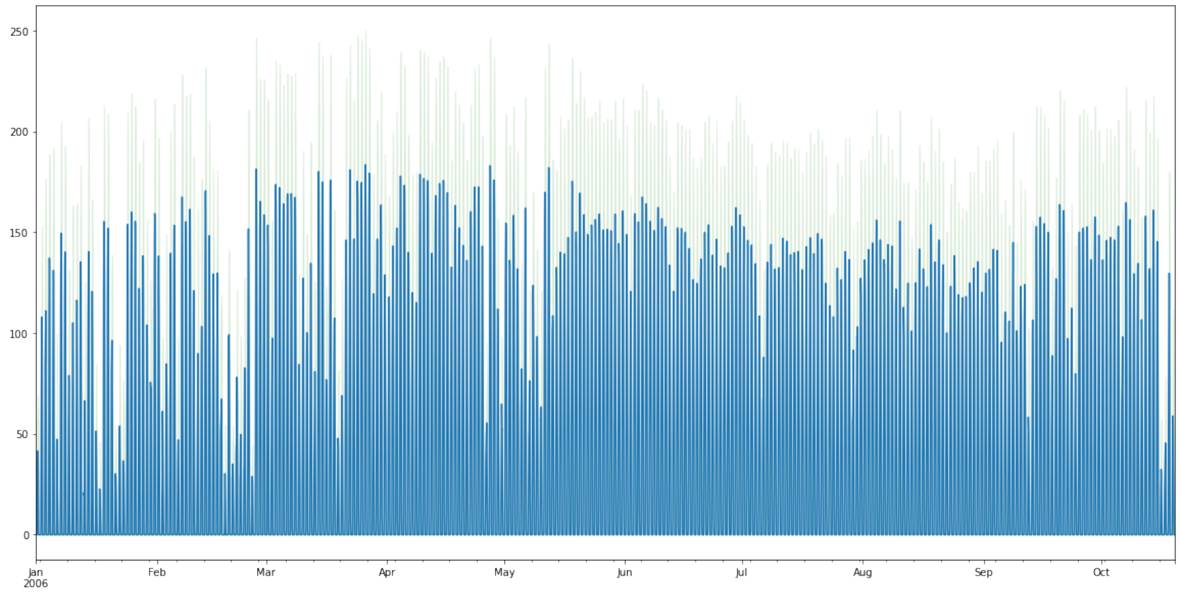


Figure A.20: Plot over the average timeseries in the solar nips dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.17	0.02	0.24	0.11
seasonality	0.86	0.02	0.89	0.80

Figure A.21: Strength of trend and seasonality of the solar nips dataset

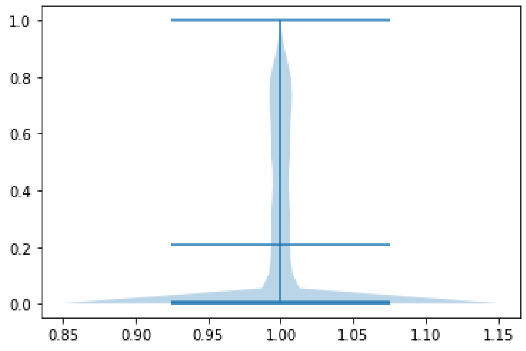


Figure A.22: Scaled violin plot of the solar nips dataset.

A.2.8 Traffic NIPS

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	0.05	963	3852963	4001	4001	0.00	1.00	H
test	0.05	6741	26964000	4000	4000	0.00	1.00	H

Table A.8: Statistics of the Traffic NIPS dataset

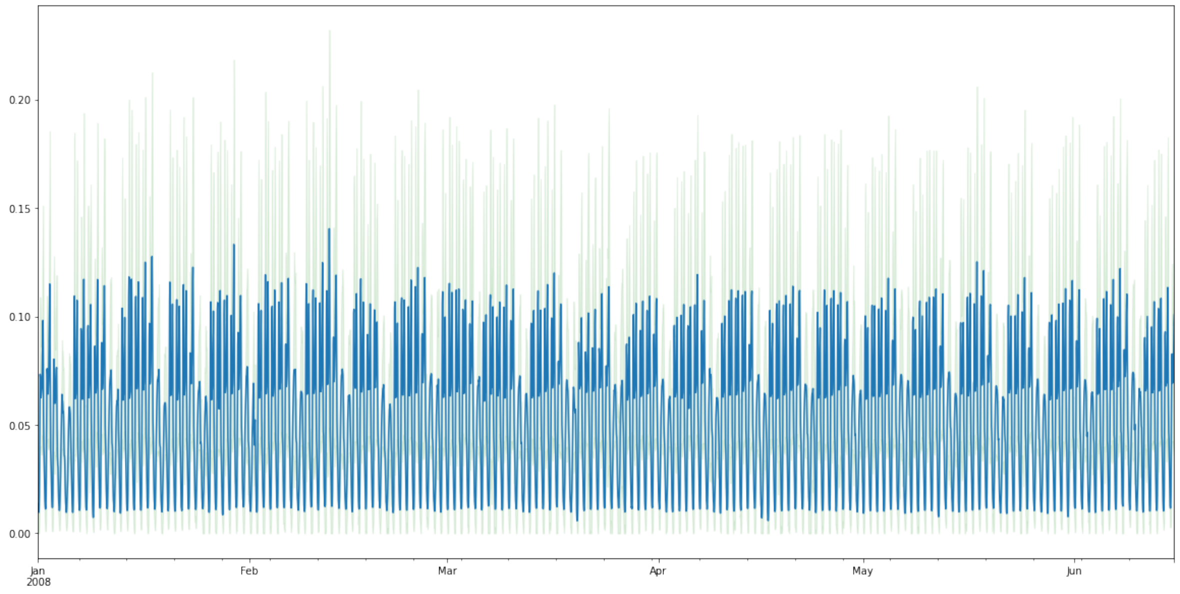


Figure A.23: Plot over the average timeseries in the traffic nips dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.29	0.18	0.87	0.0
seasonality	0.76	0.12	0.94	0.0

Figure A.24: Strength of trend and seasonality of the traffic nips dataset

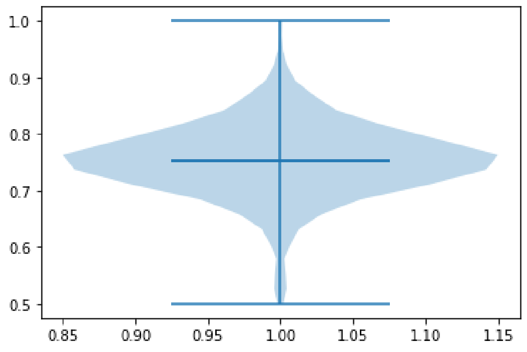


Figure A.25: Scaled violin plot of the traffic nips dataset.

A.2.9 Wiki Rolling NIPS

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	3720.54	9535	7551720	792	792	0.00	7752515.00	D
test	3663.55	47675	40619100	792	912	0.00	7752515.00	D

Table A.9: Statistics of the Wiki Rolling NIPS dataset

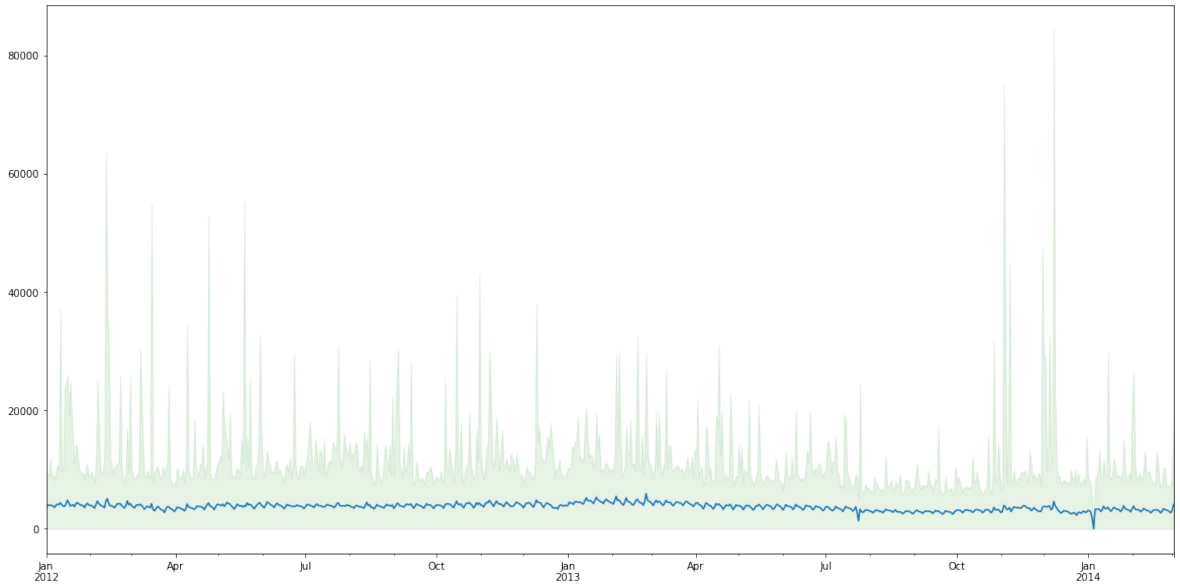


Figure A.26: Plot over the average timeseries in the wiki-rolling nips dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.53	0.27	1.0	0.0
seasonality	0.23	0.26	1.0	0.0

Figure A.27: Strength of trend and seasonality of the wiki-rolling nips dataset

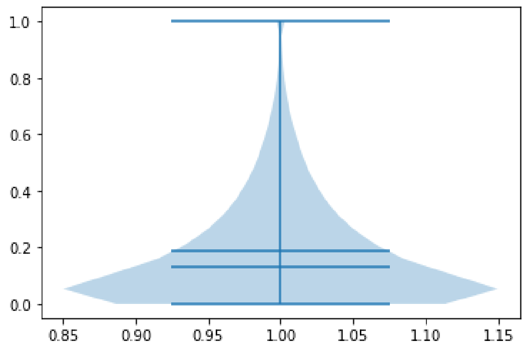


Figure A.28: Scaled violin plot of the wiki-rolling nips dataset.

A.2.10 Taxi

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	8.79	1214	1806432	1488	1488	0.0	265.0	30min
test	7.41	67984	54999056	149	1469	0.0	225.0	30min

Table A.10: Statistics of the Taxi NIPS dataset

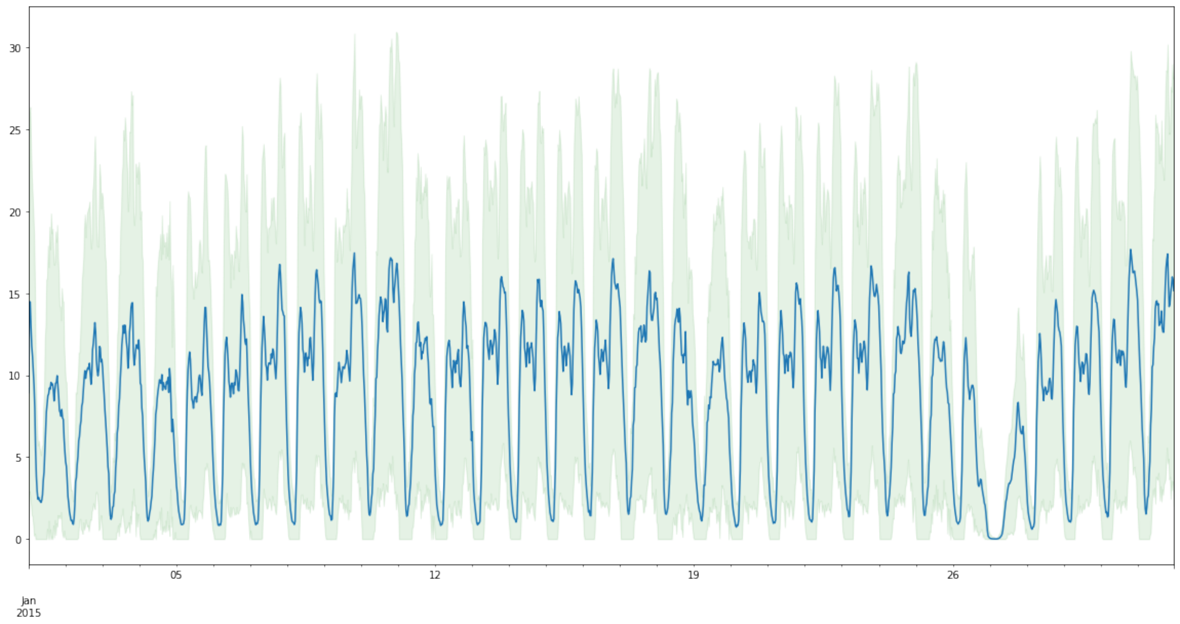


Figure A.29: Plot over the average timeseries in the taxi dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.02	0.02	0.16	0.0
seasonality	0.66	0.08	0.92	0.41

Figure A.30: Strength of trend and seasonality of the taxi dataset

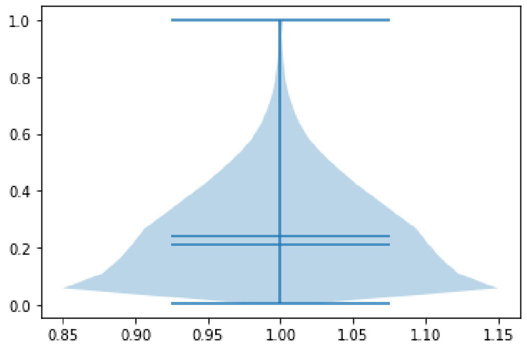


Figure A.31: Scaled violin plot of the taxi dataset.

A.2.11 M3 Monthly

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max
train	4928.47	1428	141858	48	126	80.00	86730.00
test	4971.28	1428	167562	66	144	-1200.00	86730.00

Table A.11: Statistics of the M3 Monthly dataset.

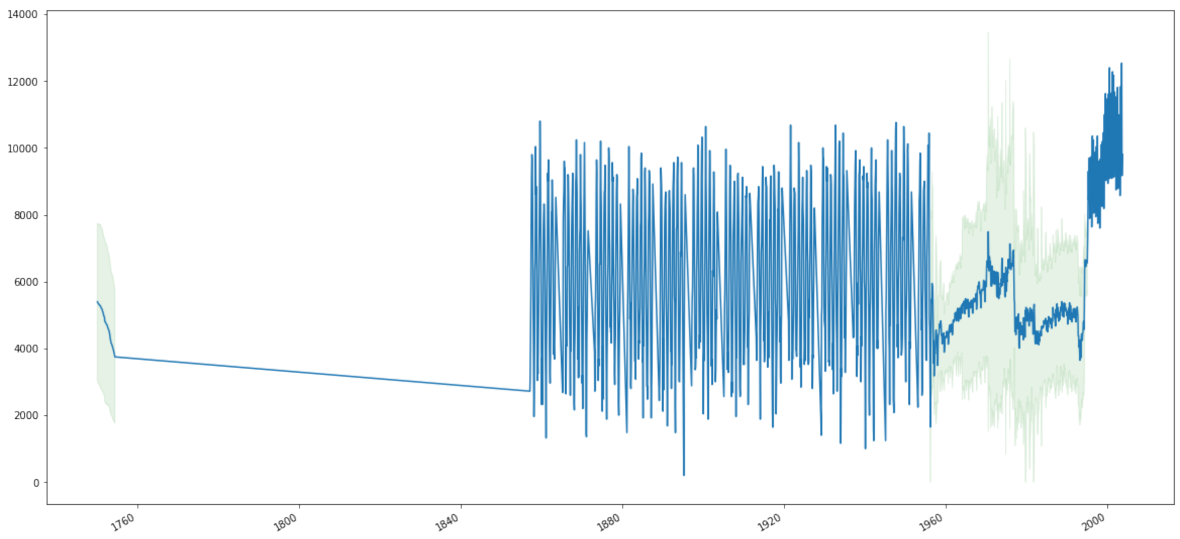


Figure A.32: Plot over the average timeseries in the m3 monthly dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.68	0.31	1.0	0.0
seasonality	0.35	0.29	1.0	0.0

Figure A.33: Strength of trend and seasonality of the m3 monthly dataset

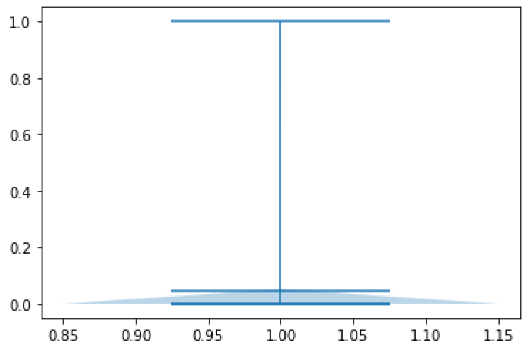


Figure A.34: Scaled violin plot of the m3 monthly dataset.

A.2.12 M3 Quarterly

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max
train	4819.27	756	30956	16	64	126.00	20245.00
test	4983.53	756	37004	24	72	121.00	20375.00

Table A.12: Statistics of the M3 Quarterly dataset.

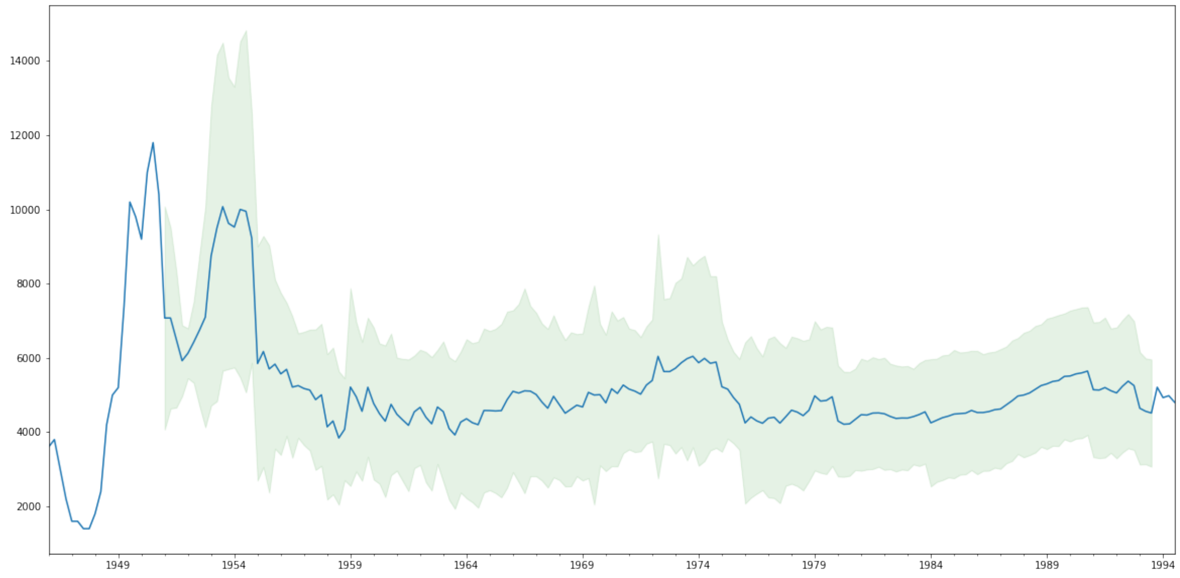


Figure A.35: Plot over the average timeseries in the m3 quarterly dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.88	0.19	1.0	0.0
seasonality	0.33	0.35	1.0	0.0

Figure A.36: Strength of trend and seasonality of the m3 quarterly dataset

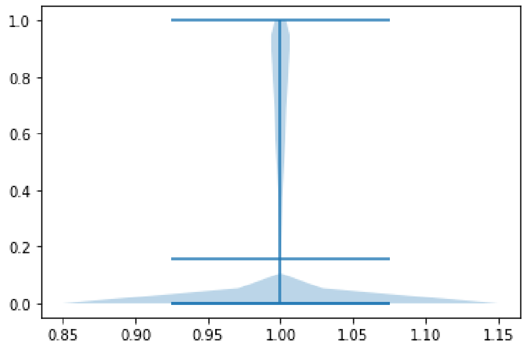


Figure A.37: Scaled violin plot of the m3 quarterly dataset.

A.2.13 M3 Yearly

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max
train	4417.05	645	14449	14	41	30.00	39666.22
test	4815.77	645	18319	20	47	30.00	45525.66

Table A.13: Statistics of the M3 Yearly dataset

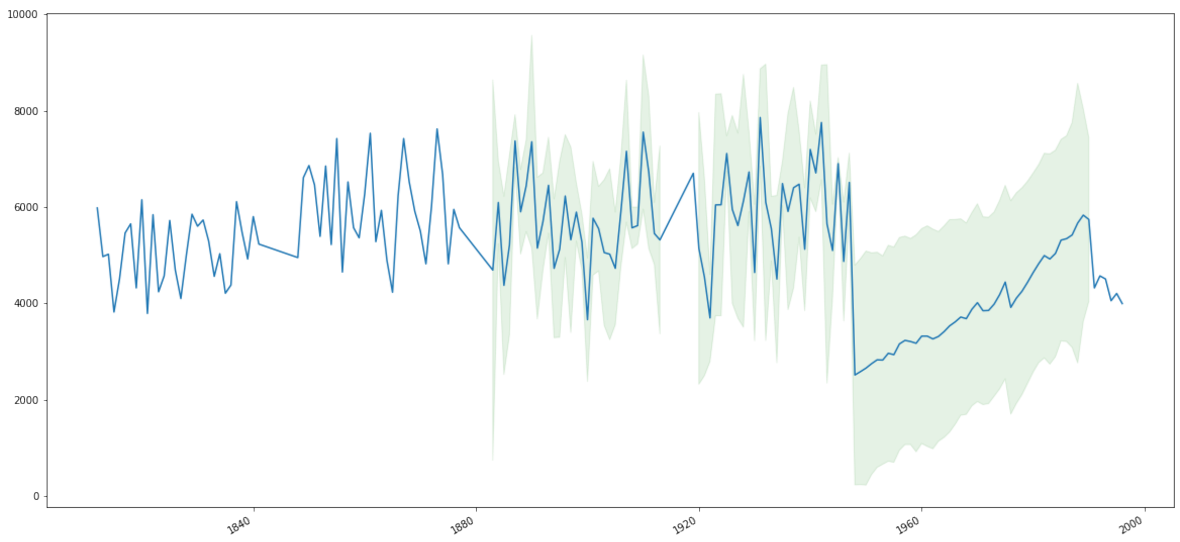


Figure A.38: Plot over the average timeseries in the m3 yearly dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.89	0.17	1.0	0.0
seasonality	0.09	0.17	0.96	0.0

Figure A.39: Strength of trend and seasonality of the m3 yearly dataset

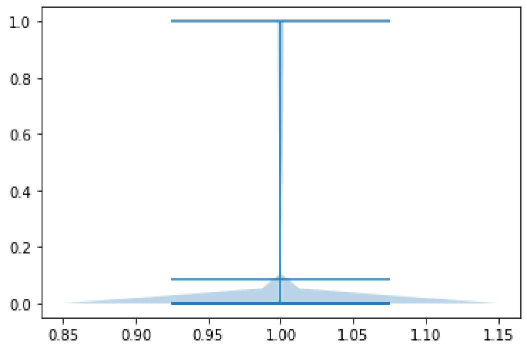


Figure A.40: Scaled violin plot of the m3 yearly dataset.

A.2.14 M3 Other

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max
train	6152.18	174	11933	63	96	28.00	59472.00
test	5999.87	174	13325	71	104	28.00	59472.00

Table A.14: Statistics of the M3 Other dataset

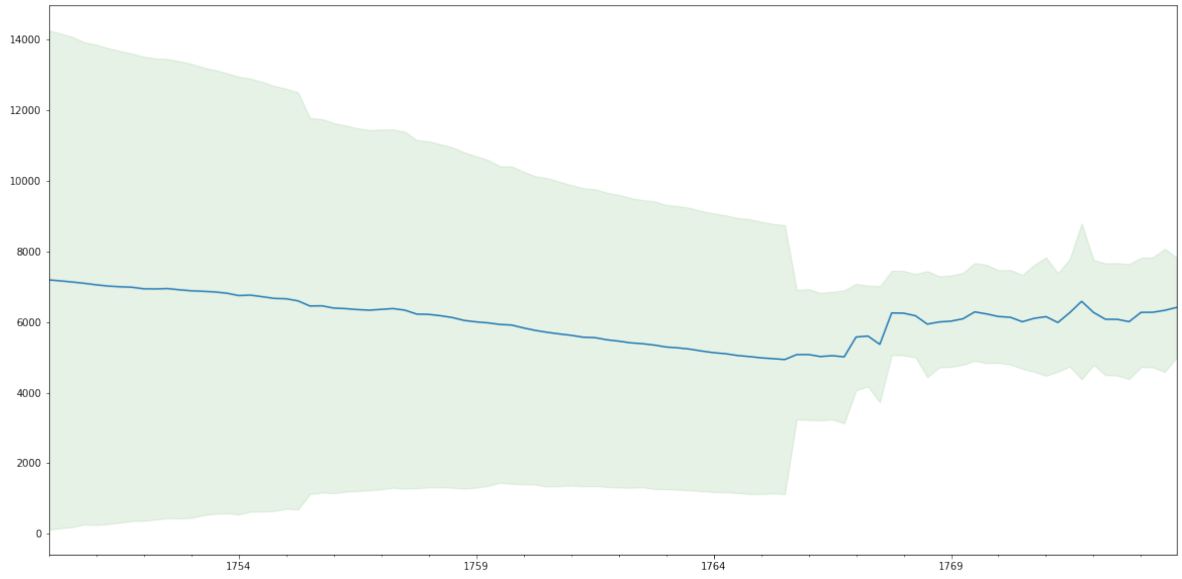


Figure A.41: Plot over the average timeseries in the m3 other dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.94	0.17	1.0	0.35
seasonality	0.07	0.08	0.44	0.0

Figure A.42: Strength of trend and seasonality of the m3 other dataset

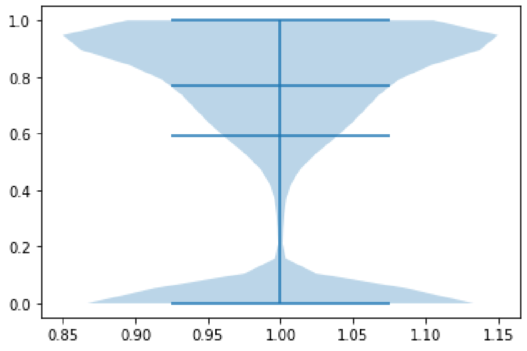


Figure A.43: Scaled violin plot of the m3 other dataset.

A.2.15 M4 Hourly

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	6827.69	414	353500	700	960	10.00	703008.00	H
test	6859.56	414	373372	748	1008	10.00	703008.00	H

Table A.15: Statistics of the M4 Hourly dataset

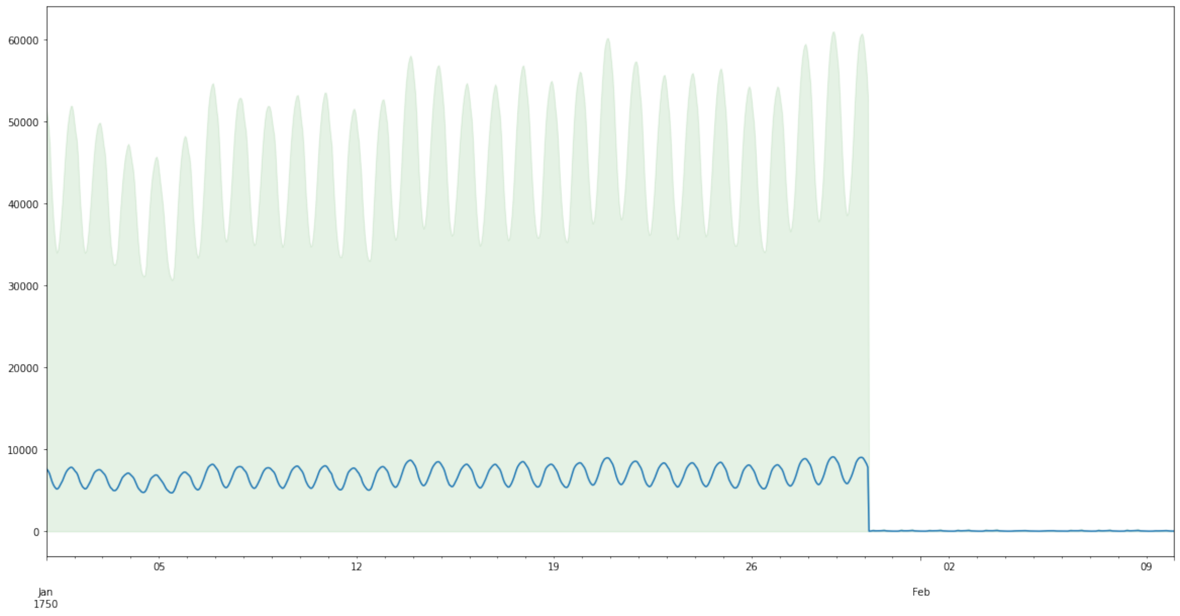


Figure A.44: Plot over the average timeseries in the m4 hourly dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.62	0.37	1.0	0.0
seasonality	0.88	0.16	1.0	0.0

Figure A.45: Strength of trend and seasonality of the m4 hourly dataset

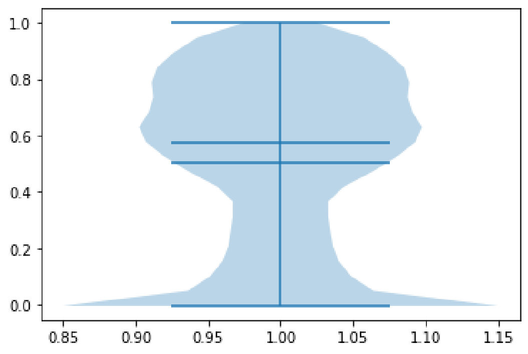


Figure A.46: Scaled violin plot of the m4 hourly dataset.

A.2.16 M4 Daily

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	4951.40	4227	9964658	93	9919	15.00	352000.00	D
test	4960.15	4227	10023836	107	9933	15.00	352000.00	D

Table A.16: Statistics of the M4 Daily dataset

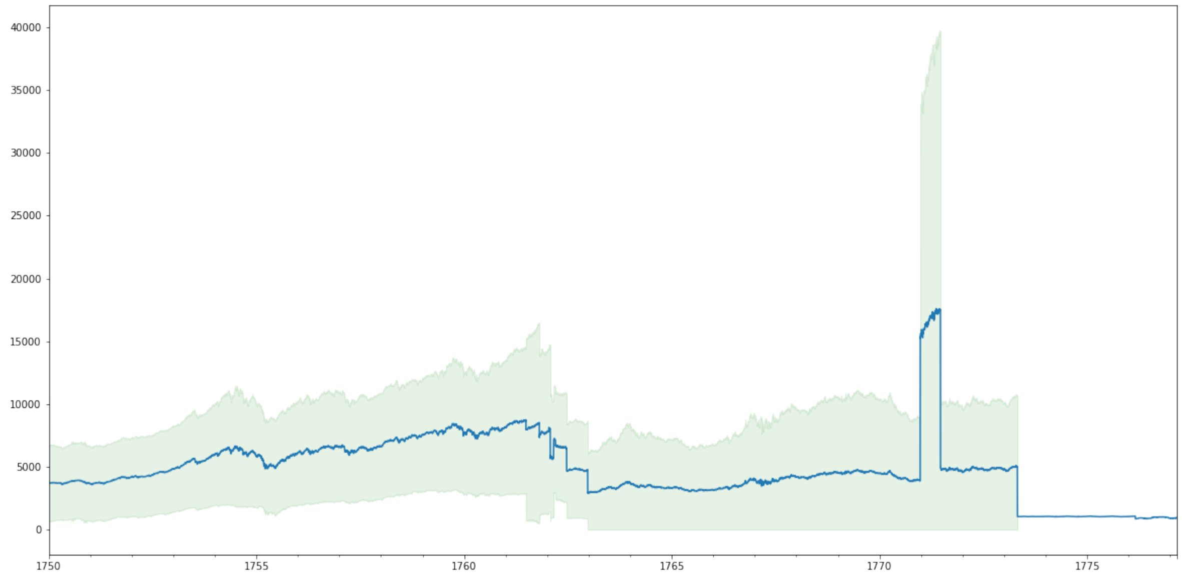


Figure A.47: Plot over the average timeseries in the m4 daily dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.98	0.05	1.0	0.0
seasonality	0.05	0.10	1.0	0.0

Figure A.48: Strength of trend and seasonality of the m4 daily dataset

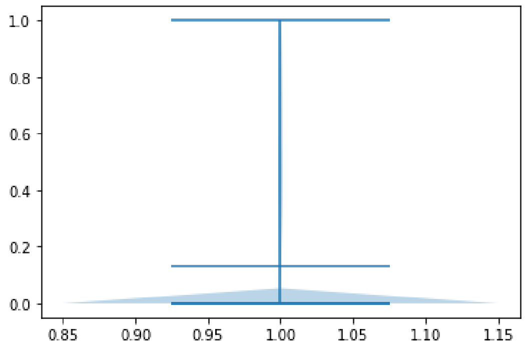


Figure A.49: Scaled violin plot of the m4 daily dataset.

A.2.17 M4 Weekly

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	3738.52	359	366912	80	2597	104.69	51410.00	W
test	3755.97	359	371579	93	2610	104.69	51410.00	W

Table A.17: Statistics of the M4 Weekly dataset

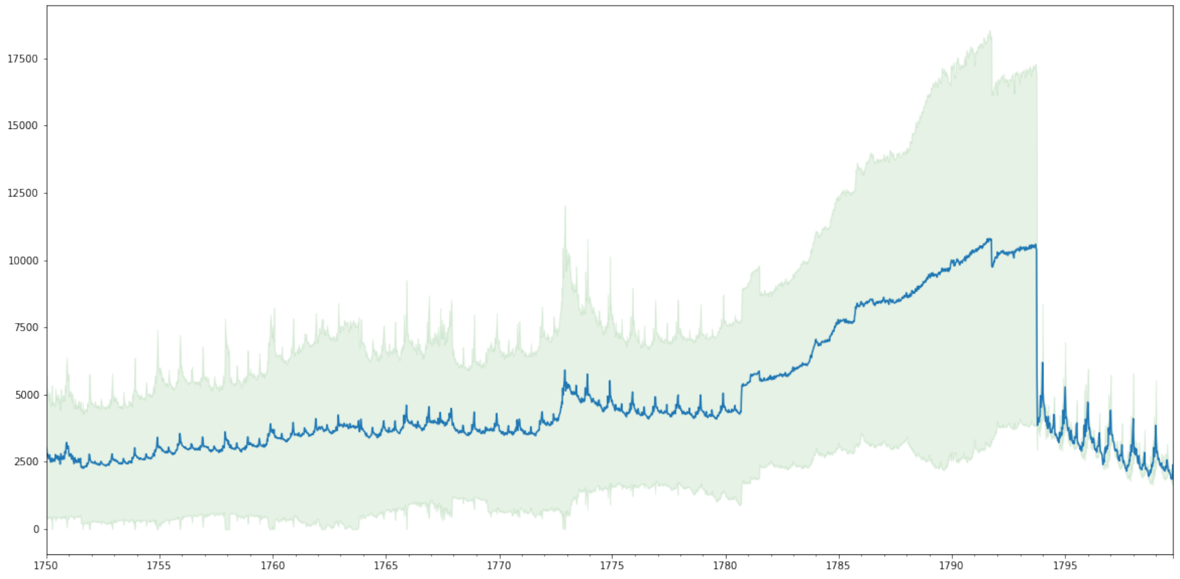


Figure A.50: Plot over the average timeseries in the m4 weekly dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.77	0.31	1.0	0.0
seasonality	0.34	0.35	1.0	0.0

Figure A.51: Strength of trend and seasonality of the m4 weekly dataset

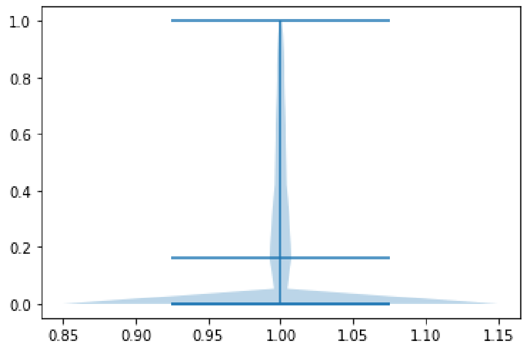


Figure A.52: Scaled violin plot of the m4 weekly dataset.

A.2.18 M4 Monthly

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	4193.28	48000	10382411	42	2794	20.00	132731.31	M
test	4207.51	48000	11246411	60	2812	20.00	177950.00	M

Table A.18: Statistics of the M4 Monthly dataset

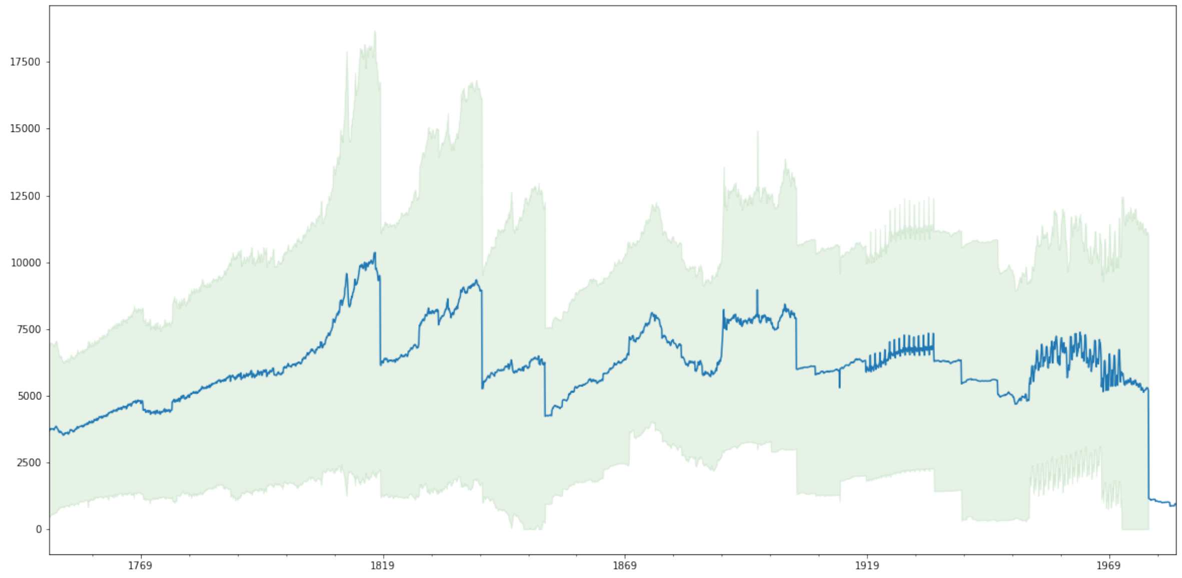


Figure A.53: Plot over the average timeseries in the m4 monthly dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.84	0.23	1.0	0.0
seasonality	0.32	0.30	1.0	0.0

Figure A.54: Strength of trend and seasonality of the m4 monthly dataset

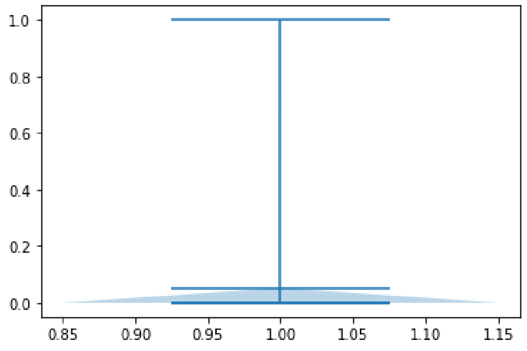


Figure A.55: Scaled violin plot of the m4 monthly dataset.

A.2.19 M4 Quarterly

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	4141.00	24000	2214108	16	866	19.50	82210.70	3M
test	4287.13	24000	2406108	24	874	19.50	82210.70	3M

Table A.19: Statistics of the M4 Quarterly dataset

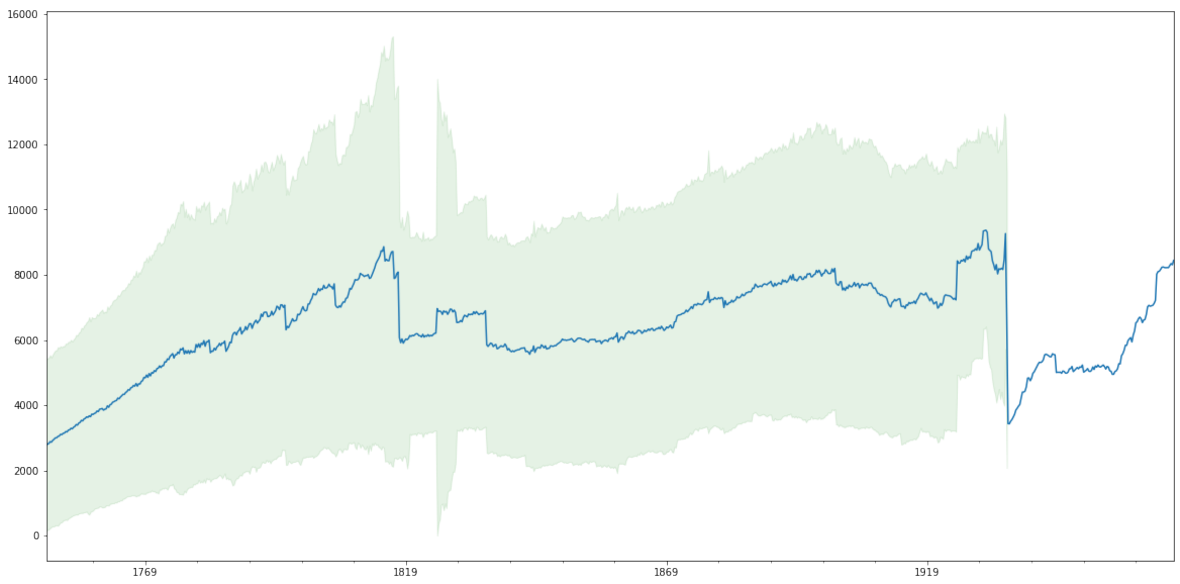


Figure A.56: Plot over the average timeseries in the m4 quarterly dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.90	0.16	1.0	0.0
seasonality	0.20	0.27	1.0	0.0

Figure A.57: Strength of trend and seasonality of the m4 quarterly dataset

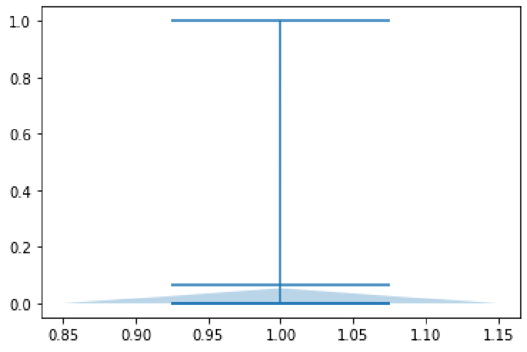


Figure A.58: Scaled violin plot of the m4 quarterly dataset.

A.2.20 M4 Yearly

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	3630.52	23000	715065	13	300	22.10	115642.00	12M
test	4076.24	23000	852909	19	300	22.00	158430.00	12M

Table A.20: Statistics of the M4 Yearly dataset

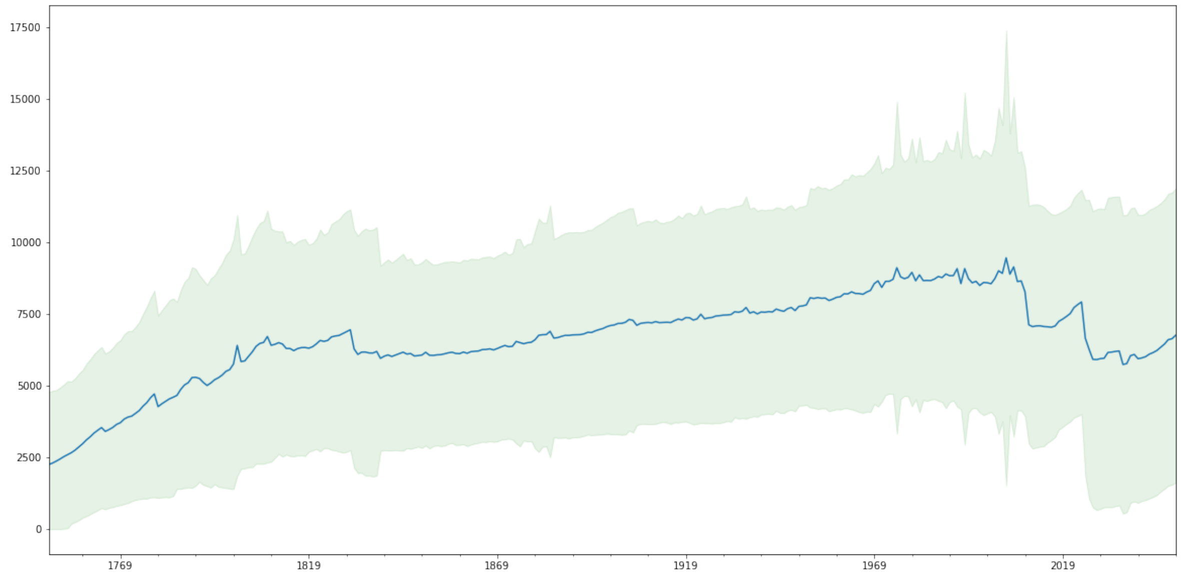


Figure A.59: Plot over the average timeseries in the m4 yearly dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.93	0.13	1.0	0.0
seasonality	0.09	0.16	0.98	0.0

Figure A.60: Strength of trend and seasonality of the m4 yearly dataset

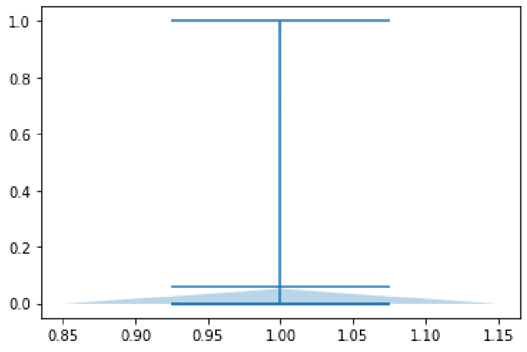


Figure A.61: Scaled violin plot of the m4 yearly dataset.

A.2.21 M5

Dataset	Mean	Series	Items	Shortest	Longest	Min	Max	Frequency
train	1.12	30490	57473650	1885	1885	0.00	763.00	D
test	1.13	30490	58327370	1913	1913	0.00	763.00	D

Table A.21: Statistics of the M5 dataset

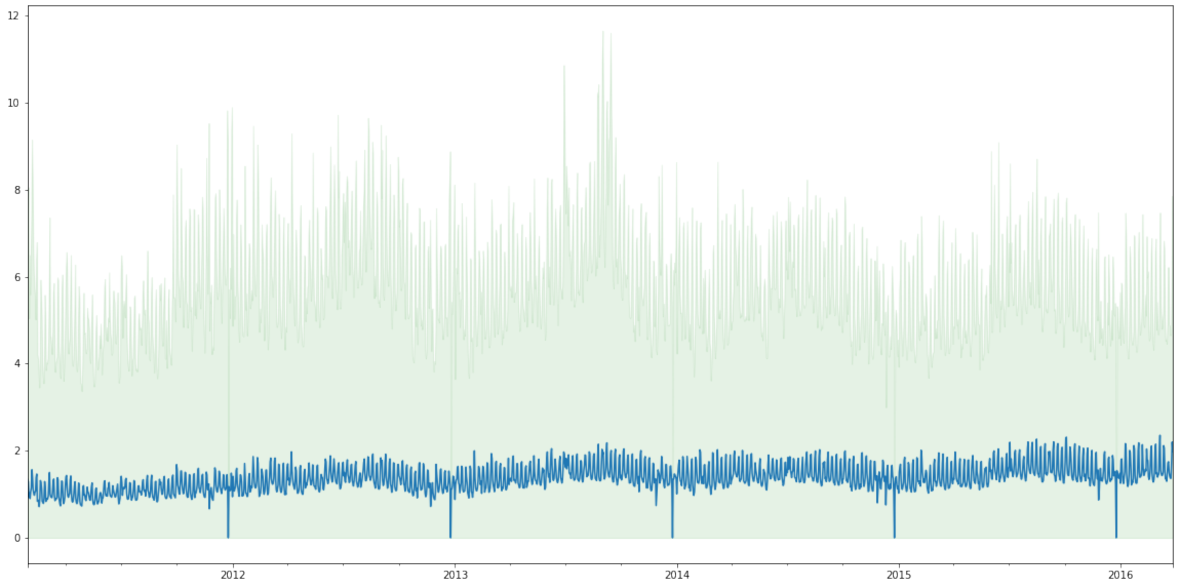


Figure A.62: Plot over the average timeseries in the m5 dataset with one standard deviation shown in green. Only positive values are shown as the dataset is non-negative.

statistic	mean	deviation	max	min
trend	0.38	0.32	1.0	0.0
seasonality	0.28	0.33	1.0	0.0

Figure A.63: Strength of trend and seasonality of the m5 dataset

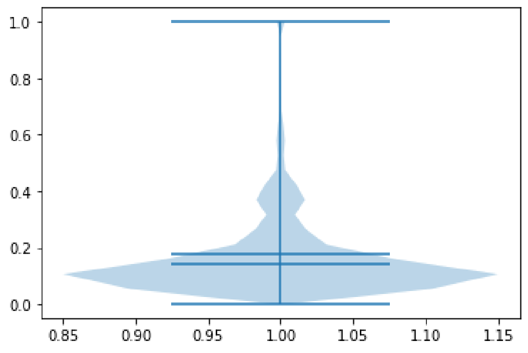


Figure A.64: Scaled violin plot of the m5 dataset.