Big Data Preprocessing for Multivariate Time Series Forecast

UNIVERSITY OF TURKU
Department of Future Technologies
Master of Philosophy Thesis
Computer Science
June 2020
Mikael Kylänpää

Supervisors:
Tapio Pahikkala

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos
MIKAEL KYLÄNPÄÄ: Big datan esikäsittely usean muuttujan aikasarjojen ennustamiseen
Pro Gradu -tutkielma, 79 s.
Tietojenkäsittelytieteet
Kesäkuu 2020
Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -järjestelmällä.

Big data -alustat helpottavat isojen datamäärien talletusta ja hallintaa. Niiden haittapuolena on kuitenkin laaja data-analyysiin vaadittava esikäsittelyn tarve, mikäli halutaan käyttää tavanomaisia analyysimenetelmiä. Erityisen haastavaksi todetaan aikasarjojen muuntaminen alustan tarjoamasta muodosta ohjatun koneoppimisen vaatimaan taulumuotoon, koostuen ennustettavasta kohdemuuttujasta sekä muista ominaisuusmuuttujista. Tässä tutkielmassa tutkitaan usean muuttujan aikasarjadatan esikäsittelyä, sekä käsitellyn datan ennustamista koneoppimismenetelmillä, kuten neuroverkoilla ja tukivektorimallinnuksella. Tutkimusmenetelmät perustuvat kirjallisuuteen datan esikäsittelystä ja aikasarja-analyysistä, mutta myös uusia menetelmiä kehitetään, kuten lokitasoon perustuva kohdemuuttuja sekä muuttujien arvojakaumaan perustuva karsiminen. Ennustustulokset jättävät kuitenkin toivomisen varaa, mikä kertoo big datan mallinnuksen vaikeudesta. Epäiltyinä syinä ovat liian vähäinen malliparametrien ja esikäsittelyvalintojen optimointi, joiden täydentäminen vaatisi resursseihin nähden liian kattavaa testausta.

Avainsanat: big data -alustat, datan esikäsittely, aikasarja-analyysi, ohjattu koneoppiminen

UNIVERSITY OF TURKU
Department of Future Technologies
MIKAEL KYLÄNPÄÄ: Big Data Preprocessing for Multivariate Time Series Forecast
Master of Philosophy Thesis, 79 p.
Computer Science
June 2020
The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

Big data platforms alleviate collecting and organizing large datasets of varying content. A downside of this is the heavy preprocessing required to analyze their data by conventional analysis techniques. Especially time series data is found challenging to transform from platform-provided raw format into tables of feature and target values, required by supervised machine learning models. This thesis presents an experiment of preprocessing a data-platform-extracted collection of multivariate time series and forecasting it by machine learning models such as neural networks and support vector machines. Reviewed techniques of data preprocessing and time series analysis literature are utilized, but also custom solutions such as log level-based target variable, and value-distribution-based feature elimination are developed. No significant forecasting accuracies are achieved, which indicates the difficulty of modelling big data. The expected reason for this is the inadequate validation of model parameters and preprocessing decisions, which would require excessive testing to improve.

Keywords: Big data platforms, data preprocessing, time series analysis, supervised machine learning

Table of Contents

Abbreviations and Acronyms

AC	Autocorrelation
ANN	Artificial neural network
AR	Autoregression
ARIMA	Autoregressive integrated moving average
ARMA	Autoregressive moving average
AUC	Area under curve
B2B	Business-to-business
BP	Backpropagation
FE	Feature extraction
FPR	False positivity rate
FS	Feature selection
GRU	Gated recurrent unit
IR	Instance reduction
IS	Instance selection
JMX	Java management extension
kNN	K-nearest-neighbors
LSTM	Long short-term memory
MA	Moving average
ML	Machine learning
MLP	Multi-layer perceptron
MSE	Mean squared error
NoSQL	Not only SQL
OHE	One-hot-encoding
PAC	Partial autocorrelation
RNN	Recurrent neural network
RVM	Relevance vector machine
SRs	Secure relays
SVM	Support vector machine
SVR	Support vector regression
TPR	True positivity rate
VAR	Vector autoregression

1 Introduction

Many of today's digital services are built around data. Varying data is collected from not just the main processes but from all around the services for later uses, including knowledge discovery by data analysis and machine learning (ML). Traditional data storage techniques such as relational databases have proven incompatible with the volume and dynamics of these large scale, distributed, and simultaneously increasing data collections, often referred to as big data. This has led to use of stand-alone data platform software such as Hadoop [1] and Elasticsearch [2], that are especially designed to operate such collections. One of the reasons enabling their advantage is the relieved restriction on data format. This is also known as schema-less approach, where all sorts of data can be collected without preorganization of table structures. A common storage format is a JSON-based document store, where the documents can contain highly altering contents, yet capable of fast and programming-oriented querying without requiring external database transformations.

Consequently, there is a certain disadvantage in analyzability of the less-structures data collections. Most data analysis and ML methods operate with table-like data, consisting of columns and rows. Therefore, applying ML on big data requires additional transformation from raw data into analyzable table-format. In contrast, traditional databases are table-oriented by nature and hence much easier to prepare for analysis. Another key issue of big data analysis is the size of the collections. Many ML-methods are computationally demanding already on small datasets, so applying them on big data insists means of transforming the data into smaller scale. To overcome such issues, this study presents a review on various data preprocessing methods, along with a practical experiment of preparing a dataset of such scope for ML analysis.

A special factor is that the experimented dataset contains no predefined target variables to predict by ML. Instead, selecting or crafting this variable is considered a part of the preprocessing. In that regard, this study represents a nowadays common data analysis task, where a client provides data and invests in exploratory data analysis, hoping for potential findings unknown in beforehand. In this specific case the provided dataset is a time series, and the goal is to preprocess it by selecting as informative as possible *feature* or *input* variables and the *target* or *output* variable to forecast. As the learning task, this study focuses on multivariate time series analysis and forecast, where multivariate refers to analyzing multiple concurrent time series, and their effects on each other.

Time series analysis has traditionally referred to statistical techniques by Box and Jenkins [3], but an increasing amount of ML alternatives have been published, due to recent advance in the field. ML has unique characteristics as opposed to traditional methods. One of them is the preliminary parameter learning process i.e. model training for which there are two categories or learning setups. In *supervised learning* setup the dataset is labeled, making each instance consist of feature values along with a class label or a numerical target value. The former label type is known as *classification* and the latter as *regression*. *Unsupervised learning*, on the other hand, operates with unlabeled data, and these algorithms are used for structuring data, for instance by clustering. Time series forecast is considered supervised, as its task is to predict the target variable i.e. future, based on feature variables i.e. history of the series. Another specialty of ML is that it is usually completely data driven. For example, rather than explicitly computing the trends and cycles in a time series, as part of the method, ML-models can learn them implicitly from training data.

Amongst ML techniques, especially deep learning has proven advantageous in modelling large datasets or big data. A specific type of deep learning is recurrent neural networks (RNNs), especially designed for learning to predict sequences such as time series. RNN techniques like *long short-term memory* (LSTM) [4] and *gated recurrent units* (GRUs) [5] have shown success with non-real-time sequences like text and speech recognition, making them promising candidates also for real-time sequences focused on this study. However, a certain difference between real-time and non-real-time sequences is that the latter is much easier to scale up. For instance, a text dataset can be increased by inserting more sentences, whereas with real-time sequences the only means of increasing is to count in a longer history or more concurrent features, if such are collected. It is hence uncertain if a time series dataset will reach the level where deep learning can outperform other methods. In addition to preprocessing techniques, this study reviews also the most common time series analysis methods, from traditional to deep learning, aiming to provide which of these are best suitable for learning real-time multivariate sequences.

2 Motivation

This thesis is done in collaboration with a Finnish IT company, henceforth referred as *company A*. One of *A*'s services is maintaining support services for clients' software and business-to-business (B2B) integrations. The services operate on clients' log and metric data, providing real time monitoring, alerting of errors, and minimization of downtime on production environments. As its data platform, *A* utilizes Elasticsearch, that is distributed JSON-based querying specialized document database. For monitoring, *A* uses Kibana [6] and Grafana [7] that are also provided by Elasticsearch. One of the disadvantages of this system is the lack of root cause analysis capabilities, due to which the discovering of the causes must be done manually after the occurrence of the errors. As the service lacks knowledge of root causes, it is also incapable of predicting an upcoming error when a root cause appears. This study aims to provide the missing predictability by developing forecasting model for a dataset extracted from *A*'s platform. If successful, the model could be used as early warning system, forecasting the state of the operation for time ahead. If the predicted state were erroneous, an early warning could be triggered, enabling prohibitive processes to prevent the errors.

In order forecast on large scale data, a heavy amount of preprocessing is required. It is acknowledged that Kibana, for instance, contains a built-in ML library that requires no excessive preprocessing. However, this study focuses on which processes are required if not using platform provided functionality. This allows the use of state-of-the-art techniques that platforms such as Kibana do not necessarily utilize. As mentioned, a key step in this project is also determining the target variable which is the variable to forecast. Since the goal is to predict the state of *A*'s client operation, this variable should reflect the health of the operation as well as possible. As the product of the preprocessing, three feature variable subsets are prepared along with a target variable to predict. Each subset is forecasted, and their performance is evaluated by the accuracy of the forecasts.

As the forecasting technique, also three candidate methods are experimented. Their performances are compared, to see which method performs best on given data. Additional parameters of forecast are *lag* distance, i.e. how much history to calculate for a prediction, and *horizon* length, i.e. how long into future to predict. Based on *company A*'s experience, the optimal lag distance should be within one hour, and horizon from 15 to 30 minutes. The lag distance affects especially the computational complexity, as counting in more instances naturally requires more computation. A longer distance, however, could increase the accuracy of the forecast. The horizon, on the other hand, defines how far ahead to predict. It hence determines how much time there would be for prohibitive actions after a warning is triggered. A longer horizon is not necessarily a benefit, as the farther the horizon, the less accurately a model likely predicts. Thus, two horizon lengths, 15 and 30 minutes are experimented. This makes the final experimental setup of this study a cross-validation of ***a)*** *2x datasets,* ***b)*** *3x preprocessed subsets,* ***c)*** *3x forecasting methods,* and ***d)*** *2x parameter combinations*. Finally, the best performing models are evaluated also as warning systems, assessing not just their overall forecast capability, but also accuracy on triggering warnings correctly.

Study objectives summarized:

- Discover preprocessing techniques for transforming multivariate big data time series into analyzable format.
- Discover means for selecting/crafting a forecast target variable for the dataset provided.
- Discover methods for forecasting time series data of such scale.
- Apply forecast and analyze the success of the set objectives.

Disregarding the objectives set above, an overall goal of this thesis is to discover universally compatible preprocessing techniques that are applicable to not just the case data, but any large-scale time series type of data. The experiments are exploratory acknowledging the null hypothesis, which is that no efficient preprocessing can be applied, and no reliable forecasts can be made on them.

The chapters are divided into two parts. Part I contains the theory of big data preprocessing, and its practical application on a client dataset provided by *company A*. Part II contains time series theory, along with forecast experiments on the client data that was prepared in part I.

Part I

Big data preprocessing

3 Background

3.1 Big data platforms

Many applications used for collecting, storing, and analyzing data can be considered as big data platforms. [8] describe the key features of them by size, complexity, and certain associated technologies such as *not only SQL* (NoSQL) databases e.g. Cassandra [9], MongoDB [10] and file distribution frameworks e.g. Hadoop, Elasticsearch. A key difference compared to traditional data storage is that these techniques usually require no strict schema or structure for content they store. This allows any available data source to be collected without restriction or prior organization. Big data platforms typically consist of a *data lake*, containing the data in raw format and a database or equivalent application, that provides indexing, organization, and better user access to this raw content. Most applications are horizontally scalable, i.e. their data can be distributed on multiple server nodes to increase storage size. Scaling horizontally allows practically infinite increase of volume. In addition, it usually provides content replication, meaning that the data is stored physically in multiple locations, which allows high availability and reliability, even if single nodes fail.

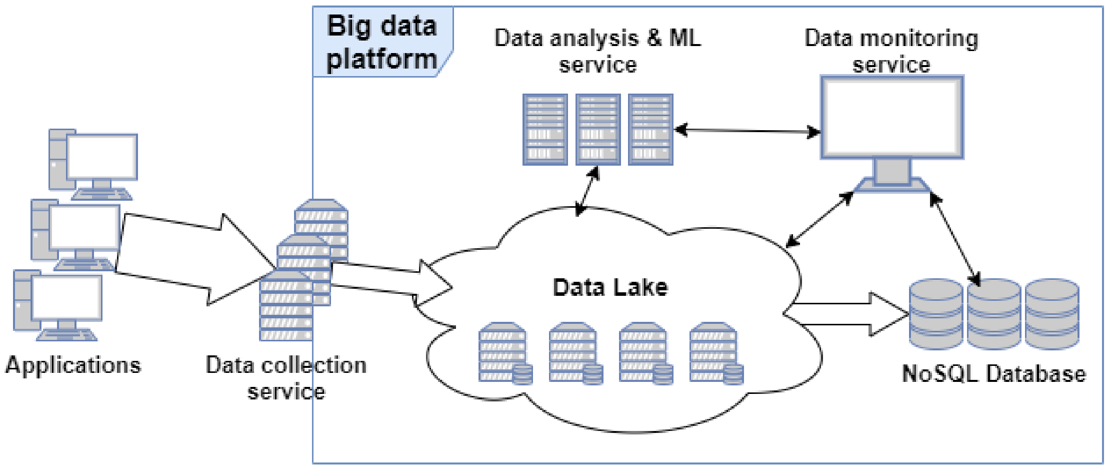


Figure 1 Big data platform example

One of the most common NoSQL database types is document database. Instead of standing for the storage format, the documents function as wrapper of content, providing a uniform format for handling any contents. For example, MongoDB and Elasticsearch use JSON as the document format. As compared to traditional data storage, e.g. relational databases, JSON is faster to search, more flexible towards varying or dynamic content and supports constructs familiar to most programming languages, such as lists [11]. Another benefit of JSON is that the documents can naturally contain nested elements i.e. parents and children, whereas a relational database would require separate tables and foreign keys for handling such recursion. Lastly, querying JSON is fast, as most programming languages have ready-made implementations for operating with the format.

An additional feature of some data platforms is being real time. This means that collection of data, from source to storage is continuous. Examples of real time data sources are application log files and sensor readings that a collecting service "listens to" and transmits into the platform simultaneously. This type of collection allows utilizing the data for real time alerting and monitoring purposes. Associating instances with timestamps also turns data into time series, enabling time series analysis. Many big data platforms provide a complete software, consisting of a storage unit or framework e.g. Hadoop, Elasticsearch, a UI client for control and monitoring e.g. Splunk [12], Kibana, and a separate component for collecting real time data e.g. Sqoop [13], Logstash [14]. Additional modules are often provided as plugins, for example for advanced visualization e.g. Grafana and ML purposes e.g. Mahout [15].

3.2 Issues with data analysis & ML

The schema-less approach has a lot of advantages concerning data collection and storage. However, as a downside, it may hinder data analysis capabilities. This chapter presents such issues, affecting especially ML-modelling on real time and big data platforms. Five main issues, namely size, dimensionality, format, missing values, and data outages are focused. In practice, the issues are solved by various steps of data preprocessing, which is a fundamental part of any data analysis study and challenging for big data.

3.2.1 Size

The foremost issue with large datasets is their size. Although general computational power has increased due to distributed and parallel computation techniques like GPU-processing, a combination of large data with complex models might still be a restriction to many studies. In turn, cloud computing providers offer high-performance computing e.g. Google Colab, as a service, but this study focuses on capabilities without using such services. If unable to utilize high-performance computing, a practical approach to deal with size issues is to reduce the volume in preprocessing.

The processes of reducing data by instances are known as instance reduction and selection [16]. Instance reduction (IR) means selecting instances to drop out, whereas instance selection (IS) means selecting the instances to keep in data. The problem of IR/IS is the loss of information value that it might cause. For example, if a dataset consists of many instances of class A but only few instances of B, a bad IR could drop all instances of B, hence completely corrupting the information. This problem affects especially unlabeled datasets, where stratification, i.e. ensuring correct distribution of labels cannot be applied. A basic application of IR for example for survey-datasets, is to remove noisy and redundant instances. This process is also known as data cleaning.

List of common IS/IR methods [16,17]:

- **Random sampling** – Drops instances randomly. The only goal is to reduce volume.
- **Stratified sampling** – Drops instances randomly amongst classes, maintaining original class distribution.
- **Outlier detection** – Reducing the data size by dropping statistical outliers. This is applicable only on numerical features, and with datasets where outliers exist.
- **RT1, RT2, RT3** – Advanced reduction techniques by [18]. These methods calculate the effect of instances for the learning task, dropping out redundant ones. The performances on multiple learning tasks were reportedly increased, utilizing them.

3.2.2 Dimensionality

Another issue related to data size is dimensionality, which means the number of features for an instance. In addition to increasing computation time, dimensionality also causes issues known as *curse of dimensionality* [19]. A practical example of this is if a dataset has few instances but many features, each instance is likely to appear distant of each other, if all features are concerned. This makes it difficult to find similarity between any instances of data. Because of this, high dimensionality also requires high number of instances for similarities to appear. High dimensionality also restricts the model selection available for an analysis task. For example, distance-based methods' such as support vector machine's (SVM) and k-nearest-neighbors (kNN) classification's computational complexity is directly proportional to number of features in data, making them easily too inefficient for highly dimensional datasets. High dimensionality can be dealt with dimensionality reduction techniques, the most common of which being feature selection and feature extraction [16].

Feature selection (FS) refers to similar selection or drop-out than IR/IS, but amongst the feature space. The foremost part of FS is to remove features which certainly have no effect in the analysis. For example, if only persons of specific age are analyzed in a study, the feature representing age is then redundant and can be removed, as it would always contain the same value. Another example are metadata fields such as object identifiers or dates of last update, common in data platform datasets. If counted in, the redundant fields may reveal spurious correlations, i.e. coincidental correlations without actual causality. FS is also used to remove non-redundant but highly correlated features. For example, if a dataset contains individual fields for age and birthdate, the two would be 100% positively correlated, making it unnecessary to utilize both fields. Even lower levels of correlation could indicate that two features represent the same event, hence only requiring one feature for recognizing it.

[20] divide FS-techniques into filter, wrapper, and embedded methods, based on the technique's relation to the actual model. Filter methods analyze the data independently, generating the final feature subset for models. Wrapper methods function together with the model, providing candidate subsets for the model, that are then evaluated to discover

the best performing one. This approach is also known as FS-cross-validation. Lastly, embedded methods refer to learning models, that have a built-in FS process. Below are presented lists of common FS methods by category.

Filter methods [16,17,21]:

- **Manual** – Selecting best features based on expert knowledge.
- **Correlation coefficient** – Selecting features that correlate most with output value or classes. Functions with numerical values.
- **Chi-Squared selector** – Chi-Squared test ranks categorical features' correlation to output value. This is like correlation coefficient but for categorical features.
- **Fast Correlation-Based Filter (FCBF)** – Advanced, computationally light, correlation-based filter method for highly dimensional data by [21].

Wrapper methods [20]:

- **Sequential forward selection or backwards elimination** – Systematically goes through each feature, evaluating its effect on performance. Selects features increasingly if they improve performance or eliminates features decreasingly if discarding them improves performance.
- **Genetic algorithm** – Advanced, natural selection inspired FS method. The method starts by creating random subsets and evaluating them. The best subset is then used to create a pool of new candidate subsets, and the process is repeated for the new candidates. This evolution converges to best possible subset.

Embedded methods [20,22]:

- **Decision trees & Random forests** – Decision tree modelling and its expansion, Random Forests are user friendly and intuitive models for example for classification tasks. Additionally, they provide a straightforward measure of a feature's importance on decisions, which acts as embedded FS.
- **Least absolute shrinkage and selection operator (Lasso)** – A regression analysis method that applies feature selection. It can also be used to enhance other regression methods such as Ridge regression.

The other approach of dimensionality reduction is feature extraction (FE), which means creating artificial features based on original ones. This enables the use of high-dimensionality-incompatible analysis methods despite the original data contains too many features. Another benefit of reducing dimensionality is that it can be used to project data into two or three dimensions that are visually observable. This could reveal patterns such as clusters, already as such. A common subcategory of FE are space transformation methods. These create projections of high dimensional data to greatly lower dimensions. In contrast to regular FE, space transformations make the projections based on all original features, preventing information loss. One could still have the output of a model as original feature but use extracted features as input for learning tasks. A downside of FE is that it makes the analysis closer to *black box*, as one cannot so easily determine a single original feature's effect, since it is not used as such.

List of common FE methods [16]:

- **Matrix factorization** – Matrix factorization methods like Single Value Decomposition (SVD) are used in recommender systems. However, they can also be used as feature selection algorithms. For example [23] represent a matrix factorization-based unsupervised feature extraction algorithm.
- **Principal Component Analysis (PCA)** – Perhaps the most common space transformation method. It projects the data to the desired number of dimensions that maximize the variance in the data.
- **Independent Component Analysis (ICA)** – A space transformation method like PCA but maximizes independence instead of variance. It is typically used in signal processing for separating independent sources of data.

3.2.3 Format

The schema-less approach improves the flexibility and scalability of big data platforms. However, there are multiple issues affecting the analyzability of such data. First, most data analysis methods operate with columns and rows. Each instance must populate every column with a value, or leave it empty, i.e. populate it with *null*. JSON for example, requires an additional preprocessing step of converting the documents into tables. Moreover, a dynamically altering JSON or equivalent dataset, might yield a large portion of empty values, requiring more strategies for handling the nulls. Second, as JSON or equivalent might contain nested objects, they must be flattened to fit it into a table. This means unfolding all nested objects into single level, which could make the representation more complex. The unfolding process is illustrated in below, where the braces refer to parent-child relations:

```
input:              output:
A                   A
B{C, E}       →     B.C, B.E
F{G, H{I}}          F.G, F.H.I
```

Like format of data, also the type of features affects analysis, especially with big data. Since ML algorithms are math-based, textual features must be transformed into numerical format before analysis. Feature indexers and encoders are techniques that can be used for such transformation [24]. A discrete amount of categorical textual field values, for example species names in a dataset of plants, are straightforward to index into numbers where each number represents one species. However, more complex textual features such as actual text entries are harder to index or encode. Text mining presents techniques aiming to categorize texts to index them into numerical scale [16].

List of common indexer and encoder methods [16]:

- **String indexing** – Transforms a discrete set of textual labels into indexed numbers. The order of which can be defined for example by the total number of labels; the most frequent class or label being either first or last.
- **One-Hot-Encoding (OHE)** – A common strategy, also used for representing output classes with neural networks. In a discrete set of labels, each label is given a binary feature, yielding value 1 if an instance represents the label and 0 otherwise.

The transformation to numerical features is fundamental to any size of dataset that ML is applied to. However, especially large scale and highly dimensional data is problematic. For instance, if a dataset is already highly dimensional, OHE increases the dimensionality even further, which could introduce dimensionality issues. This has been studied [24] and [25] who propose alternative techniques as well. [24] demonstrate issues with non-standardized categorical variables. An example of such is a person's title-field which could contain multiple entries like *PhD* and *Ph.D*, despite all referring to the very same entity. With basic OHE, these would yield two or more non-lapping features, which could corrupt the analysis as the model would interpret them as different entities. They propose their own method, especially targeted to "dirty data" i.e. datasets with spelling mistakes that make same entities represented by multiple accidental values. [25], on the other hand, present a comprehensive comparison of OHE against binary encoding and feature hashing, more detail of which is given below.

List of OHE alternatives [16,24,25]:

- **Term frequency search** – This approach is eligible for continuous textual fields. A specific term is defined as keyword, and the frequency or amount of it is scored as the new feature value.
- **Stemming** – This approach is like term frequency search, but instead of searching a term within texts, searches roots of terms within terms. An example is root *standard* in term set *standards*, *standardize* and *standardized*. There are many extensions to basic stemming, such as [24]'s similarity encoding, which provide more advanced indexing or scoring of roots.
- **Binary encoding** – Where OHE represents features as one-bit binary, binary encoding represents them as multi-bit. For example, where OHE encodes a feature of 4 classes into 4 binary features (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0) binary encoding requires only two; (0,0), (0,1), (1,0), (1,1). This approach yields much less dimensions when encoding discrete categorical variables.

- **Feature hashing** – In this approach, feature values of any type are sent through a hash function, which denotes an output integer within a set of desired length. The acquired integers act as the new encoded features. Interestingly, hash collision, i.e. two separate values being hashed to same output does not drastically decrease prediction performance as heavily as expected, according to [25]'s study.
- **Clustering** – An ML-oriented alternative to apply feature encoding or any dimensionality reduction is to perform unsupervised learning such as clustering on desired set of features. The formed clusters can then be interpreted as feature labels.

For an even more advanced encoding strategy, [26] represent a neural network approach, resembling an autoencoder. Autoencoders are neural networks that learn complex functions for altering data representation and are therefore used for example for encoding and decoding purposes. [26]'s network functions like an autoencoder but also reducing the output dimensionality to a smaller scale. It essentially falls into the category of space transformations, but it can be also used to convert non-desired feature types such as categorical textual fields into numerical representation, which allows better modelling capability.

Feature types especially affect time series analysis. For example, statistical regression methods like ARMA-models (see 5.2) operate only on ordinal numerical values. Ordinality means that the feature can be measured as high or low compared to other instances. Many categorical features are nominal or not ordinal, meaning that their values cannot be compared to each other. An example of such is *species name* that represent category rather than measure. There are exceptions of ordinal categorical features such as log level. It is a textual feature commonly associated with log messages, denoting the severity of the event with values like *debug*, *warning,* or *error*. Since the categories measure the severity, they can be indexed into increasing order, i.e. *debug*: *1*, *warning*: *2* and *error*: *3*, making the feature ordinal and applicable to statistical modelling.

3.2.4 Missing values

Another category especially related to the schema-less storage are missing values. Efficiently handling missing values is critical, as bad handling strategies could bias the analysis [16]. One of the primitive handling techniques is to drop out instances that have missing features. However, this is only applicable if the portion of instances with missingness is small. If not completely dropping instances, the other approach is to fill their empty fields with artificial values. This method is also known as imputation. There are various imputation strategies, i.e. criteria for what to fill the empty values with.

List of imputation strategies [17]:

- **Mean or median** – The most common and basic imputation strategy is to fill empty values with feature's or table column's mean or median value. This is easily extendable, for example to instead using a mean or median within a class or category.
- **Hot deck imputation** – Find an instance that most resembles the instance with missing value. Use this instance's value to impute the missing one.
- **Regression or classification** – An advanced imputation strategy is to build a complete separate classification or regression model, that takes as input the other feature values and predicts the missing value based on statistics or model parameters if ML.

Like format, also missing values cause difficulties especially withs time series datasets for multiple reasons. First, one cannot completely drop out only instances with missing values because time series must always have fixed time intervals between instances. Second, one cannot impute empty values with 0, -1 or other fixed values, because they could be in the natural scale of some variable, for instance, temperature in Celsius. Third, one cannot impute empty values with averages or most common values, at least in a non-stationary time series (see 5.1). This is because the series could be on a peak of a trend during the missing value's occasion. Inserting a much smaller average value between two peaking values would probably corrupt the time series.

More complex imputation strategies for time series have been reviewed by [27,28]. [27] studied the impact of missing values in political data analysis. The key issue affecting time series imputation in this field is the type of the missingness, e.g. informative or non-informative missingness. In the former case, a problem is that there is likely a reason why a feature value is missing, which should be concerned in the imputation. The process should also concern other features, which have values for the moment. For instance, if every other feature dive at a specific moment, the imputed value should likely dive as well. A similar problem is presented by [28], who studied missing values in multivariate scientific time series forecast with RNNs. They found missing values correlated to prediction performances also in sensorial data, e.g. health care, biological and geoscientific datasets. However, instead of applying imputation, they propose a completely new kind of GRU neural network (see 5.3.3), namely GRU-D, that functions with the empty values, rather than guessing the real value of them. This approach showed promising results; where regular RNN with imputation performed at the same level with SVM, GRU-D was able to deliver the increased performance that deep learning generally provides. The network seemingly learns to interpret null as a special class to detect patterns of. However, this likely requires a substantial portion of missing values, to learn their pattern. Relatedly, for example [29]'s RNN-experiment concerned a small amount of empty values, so imputation could be considered a minor factor regarding prediction performance.

Missing values are especially problematic in multivariate time series analysis. As described, the format and size of big data might cause missing values as such. An additional factor is the density or framerate of time series. To analyze multiple time series' effect on each other, they must be synchronized by framerate. This means that each individual series must share timestamps. Hence, and additional preprocessing step is required for synchronizing the time series, which could be a heavy process especially with large datasets. A time series can be synchronized either by sparsening or densening it. Sparsening means discarding for example every nth instance from a series. It could lose information value, as the discarded instances could include informatically valuable ones. Densening, in contrast, means adding artificial instances between original ones. It does not lose information value, but could corrupt the data, if the artificial instances are inaccurate. A synchronization process could include applying both, to obtain a uniform framerate for every time series in a dataset.

3.2.5 Data outages

A special type of missingness are data outages. Affecting especially real time data platforms, outages refer to continuous missingness of a data source or feature for a limited period. An example outage is if a sensor or collecting service malfunctions, which causes missing values or no data until it is fixed. Outages do not only refer errors, as they could also be due to regular restart or reboot of a service. Because of this, outages are especially difficult to model, as there could be multiple origins for one, either normal or erroneous. There are very few publications on how to handle outages in multivariate time series analysis. A primitive approach like with regular missing values would be to discard periods or complete features with outages occurring. However, this would lose potential information value that defines the missingness. Because an outage could cause or be caused by an event in another series or features, identifying such correlations could be critical for example in root cause analysis of an error.

If wished to benefit from outages in analysis, the biggest concern is how to represent them in the data. As described, most models cannot function with nulls, so this requires a process of labeling the outages. The difficulty of labeling depends on feature types. For example, one can simply add a new category or label representing outages for a categorical feature. However, or an ordinal and numerical feature, it is difficult to allocate a value for representing one. Intuitively, one could label outages as zeros, but with assumption that it is not in the natural scale of the feature. If so, an alternative strategy could lie in discretization. Discretization means transforming a continuous value into discrete set of ordinal categories. An example discretization is transforming temperature degrees into classes like *freeze*, *around-zero*, and *heat*. This set of classes could then be expanded with a new class that represents outage. However, it would still not satisfy the ordinality requirement for most features, as an outage cannot be measured as high or low, for example by temperature. Especially statistical time series methods (see 5.2) require ordinality, also making discretization ineligible.

Among the few publications concerning the issue is [30]'s study of missing event prediction in sensor data streams. For an outage situation, they utilized Kalman filters, that is a recursive data approximation function for predicting the missing sensor readings. However, this approach merely predicts the missing values, rather than utilizing the informative missingness in the outages. A candidate method that could benefit from outage information is the GRU-D neural network proposed by [28], as it particularly analyzes patterns of missingness.

4 Case study – Preprocessing for time series analysis

4.1 Target & objectives

The case study is targeted on a specific client of *company A*, henceforth *client B*. *B*'s production operation runs on five host machines that contain software components such as integration engine and API manager. Their main process are *B*'s B2B-integrations that conduct the internal and outbound message traffic, including orders and other logistical messages between various physical and virtual endpoints. The integration engine's load is balanced into two hosts, namely *cores 1* and *2*. The other three hosts in the cluster perform additional supportive operations such as secure relaying. In this chapter, *client B*'s operational data is preprocessed and prepared for time series analysis in part II of the study. Both custom-made techniques, and methods reviewed in chapter 3 are experimented, to create a computationally compact and informative dataset from the raw data.

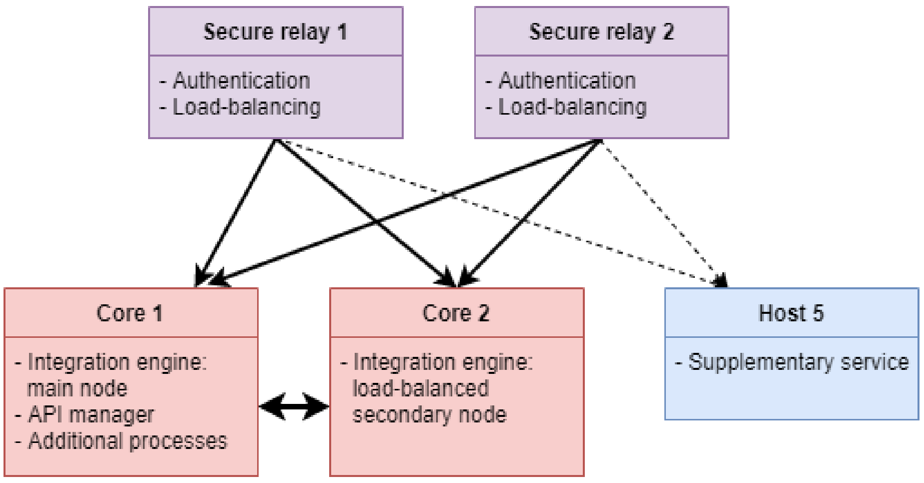


Figure 2 Client B's cluster

As for any data prediction task, the first consideration is which variables to analyze and which to predict. Since *cores 1* and *2* host the main operation, these two also produce the most impactful logs and metrics to analyze. Considering this, the preprocessed dataset should contain mostly core features, so the models do not overfit on the supplementary behavior. The target or output variable should also represent the *cores*. Although being supplementary of the main operation, the other three nodes could provide additional root causes that affect the behavior of the *cores*, and hence should not be considered as input features. The preprocessing objectives are illustrated in fig. 3:

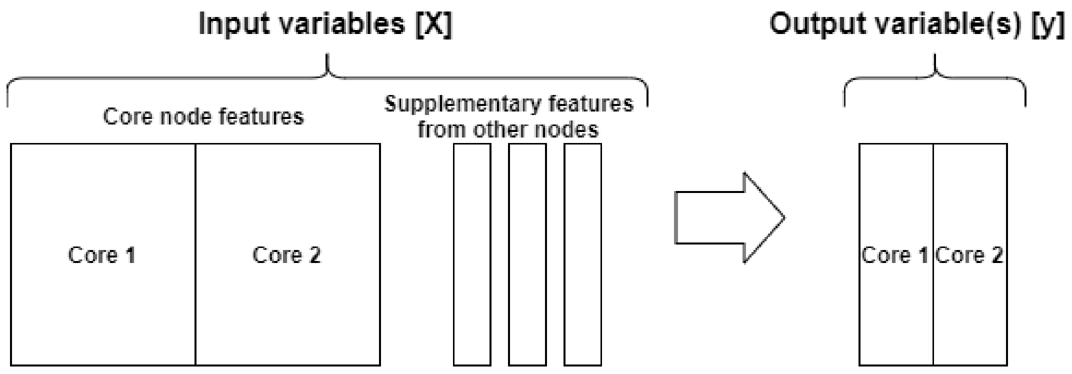


Figure 3 The forecasting methods analyze input variables to predict output variables. The boxes indicate which host the features or variables are selected from. The output variable should present the predicted state of errors in both cores.

As for the target variable, since not being predefined, one must either select a representative feature from the original features as such or extract it from the original features. Most forecasting methods cannot predict multiple targets, so if selecting multiple targets, the forecast would need to be repeated for each individual log level (see 3.2.3). Hence, the goal is to craft a single target to predict. Since *company A*'s role in *client B*'s operation is to ensure system health, the target variable should reflect the health of the operation. There are two key variables in the data, that are expected to reflect the health best; the log level features that the data contains plenty of, and outages that occur from time to time in all features. Log levels are straightforward indicators of errors as such, whereas outages could be caused by a critical error in the system making them potentially informative targets to forecast (see 3.2.5).

4.2 Dataset

The dataset of *client B* is a collection of data tables provided as csv-files. These tables were created in a separate data engineering project [31], acquiring the data from Elasticsearch, transforming it from JSON into tables, and reducing the volume by a large scale. The final tables contain multivariate time series, from the five host machines in *B*'s production environment. There are from two to three tables per each host, each table containing a specific source of data in it. The source types are described below:

1. ***Filebeat*** – Contains application log data. This includes log levels and additional status information of logs. The message contents are filtered out due to their excessive lengths and complexity to analyze. *Filebeat*-sources have a dynamic collection interval, as they are collected simultaneously when produced by services. The features of the source are mostly categorical variables.
2. ***Metricbeat*** – Contains system metrics such as CPU and memory usage of hosts. The readings are collected at fixed rates within configured intervals. All *metricbeat*-features are categorical.
3. ***Jmx-beat*** – Contains Java management extension (JMX) statistics, i.e. metrics from important java processes. These readings are also numerical and collected within configured intervals.

The table rows are ordered so that the first row of each table represents the first lag and last row the last lag in the time series. The raw data contains from 4 to 63 features per each table. The collection intervals above refer to the rate that the data is collected to *A*'s Elasticsearch cluster. The tables are synchronized to uniform framerates, so that the nth row of each represent the same moment of time. The time series span a period of originally 8 months: April 18th – November 15th. However, the provided data is divided into two subsets. *Dataset 1* contains the full period with a 5-minute framerate (60780 rows), and *dataset 2* only the last 2 months but with a 1-minute framerate (87900 rows). The advantage of *dataset 1* is that it covers the full period, including a wider history. However, the original data is generated denser than 5 minutes, so the synchronization process has decreased event accuracy. In contrast, *dataset 2* is closer to the original density but covers a shorter history. In 4.4.2, the length of *dataset 2* is further shortened to 50500 rows for reasons explained.

As a remark, both datasets contain a lot of missing values, for two reasons. First, the data is highly dimensional, and the collection rates vary between individual features and tables. Because of this, some of the features have been densened by inserting empty values between the original values to match the uniform framerates. Second, there are a lot of outages, that appear as missing values.

4.3 Missing values & outages

The foremost problem of this study to solve are missing values. This is because most forecasting methods and preprocessing techniques cannot operate with nulls and cannot be used before replacing these. A key factor affecting the replacement are outages. There are two types of them: normal outages due to service reboots or updates, and outages caused by errors. Moreover, outages can take place for a single service, or for the whole system.

A certain property of outages in is that they act differently on logs and metrics. Since logs are written based on actions of the process, a missing log could represent a healthy period, where in contrast, dense logging could indicate errors. Hence, missing values are considered natural for *filebeat*-tables. Metrics, on the other hand, are collected at fixed rates. Therefore, missing metrics can indicate either a sparser collection interval or an outage. Because of this, missingness should be treated differently on *metricbeat* and *jmx-beat* tables. Also, as shown by [28], there could be information value in patterns of missingness, so the missing values should be kept as such rather than imputing. The next proposed methods aim to classify outages either as ***a)*** *natural missing value due to sparser collection rate,* ***b)*** *normal outage due to service restart,* or ***c)*** *error-caused outage due to system malfunction*. The natural missing values could then be imputed, and the outages labelled either as normal or erroneous outages.

4.3.1 Outage detection

To address the described requirements, the first process is trying to separate natural missing values from outages. The following proposed method analyzes complete missing rows for separating the instances. As *filebeat*-outages are treated differently, this method is applied only on *metricbeat* and *jmx-beat* tables. The basic idea of it is to check whether some values of a row are missing, or all of them, as all features missing would refer to a metric-wide outage. The method is presented as algorithm below:

```
iterate table by rows:
if some, but not all features of row are 'null':
    impute missing values
else if all features of row are 'null':
    label row as outage
```

By experimenting, it is noted that with *client B*’s data, this method still yields a heavy portion of outages in *metricbeat*. Below is an illustration of the resulting outage labels on *core 1*:

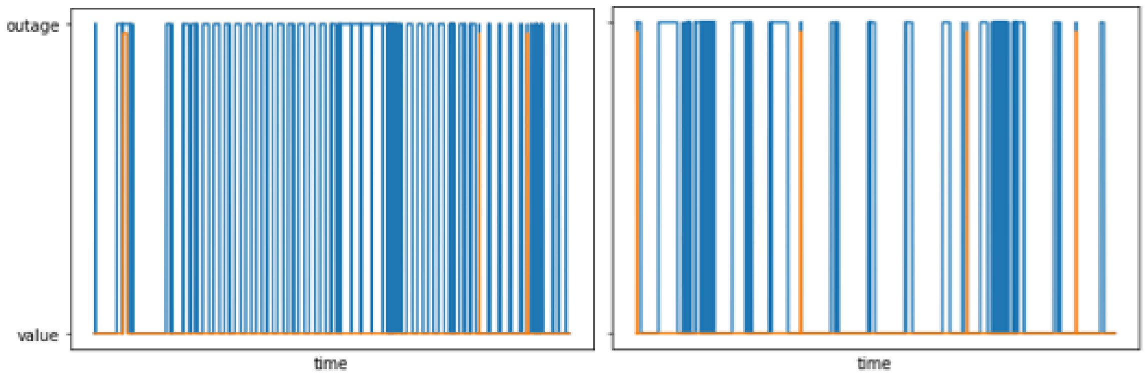


Figure 4 Dataset 1 (left), dataset 2 (right). Metricbeat outages (blue), jmx-beat outages (orange). The x-axis represents the full analysis periods on both datasets 1 and 2.

The problem of heavy missingness is that as the metrics are numerical, they would also need to be imputed with a numerical value, rather than a new category representing them (see 3.2.4). However, the regularity of the missingness in *metricbeat* suggests that they are caused by the sparser collection, rather than actual outages occurring. Unlike *metricbeat*, *jmx-beat*'s original collection intervals are denser than the synchronized interval, so there are no sparseness-caused missingness. Because of this, the outages present in *jmx-beat* (orange) are likely indicating actual system-wide outages. More evidence is given by the fact that, there are also missing values in *metricbeat* and *filebeat*, during *jmx-beat*'s outages. Since *jmx-beat*-tables represent outages so well, the approach is changed to labeling only the outages detected by *jmx-beat*. Hence, the outages detected in *metricbeat* only are handled like regular missing values, which is performed in 4.3.4.

4.3.2 Missing value imputation

After the outages have been detected, the imputation strategy for other missing values must be selected. The imputation is applied to the missing values that are not considered as outages in 4.3.1. A key factor considering the strategy is the nature of data. An expected property of server metric data such as *client B*'s is that the metrics should keep previous state if no explicit changes occur. There should also be no dirty data features (see 3.2.3), as all features are recorded by sensor rather than human. Lastly, since working with time series data, common imputation strategies such as mean, or median cannot be used (see

3.2.4). Acknowledging these, it is decided to impute missing values based on the preceding active value for each feature. If the missing value follows an outage, the next active value should be used instead, because the value is more likely to represent the state after the outage, rather than state before it. The process is also represented by following algorithm:

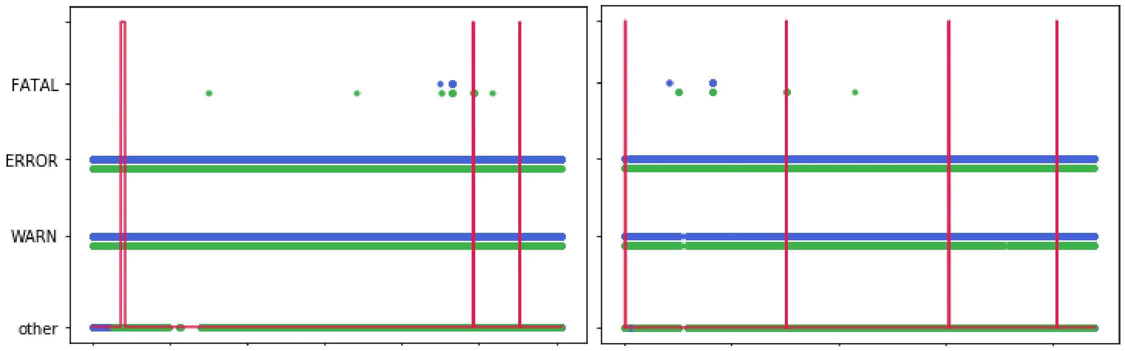
```
1. iterate table by rows (top-down):
    iterate row by values:
        if row is not outage and value is 'null':
            impute with corresponding value from preceding row
2. repeat step 1. bottom-up
```

Iteration 1. imputes missing values with the last collected value of the feature if one exists within an outage-less period. This leaves all missing values that follow an outage empty. Iteration 2. repeats the process but from bottom to top, so the missing values after outages get imputed with the next collected value of the feature.

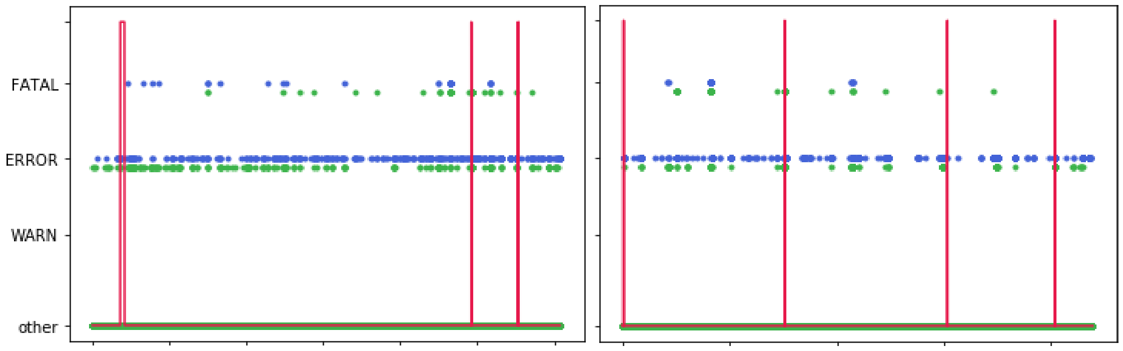
4.3.3 Outage-error correlation

After detecting outages, a consideration is whether it is possible to segregate them as normal e.g. updates and reboots, or error-caused. The distinction between the two could benefit the forecast, as one is only interested in predicting the error-caused ones. It would enable the models to specialize in detecting feature values that correlate with them. The normal outages could be imputed like regular missing values in 4.3.2. As the data contains no distinction between normal and error-caused outages, an error-correlation analysis is proposed. Since there is no metric collection during outages, one cannot calculate straight correlation between them. Instead, a visual analysis is performed, where the log levels are plotted against systemwide outages to detect if increasing errors lead to outages. This method also reveals correlation in the opposite direction, i.e. whether outages lead to increasing errors. Below is illustrated the outages of *core 1*, against the log levels of 4 corresponding *filebeat*-logs recorded in both *cores*:

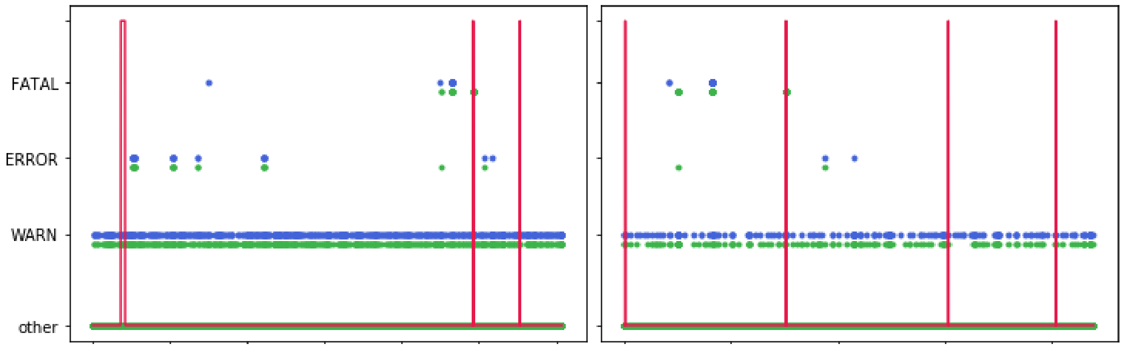
log ***a)***



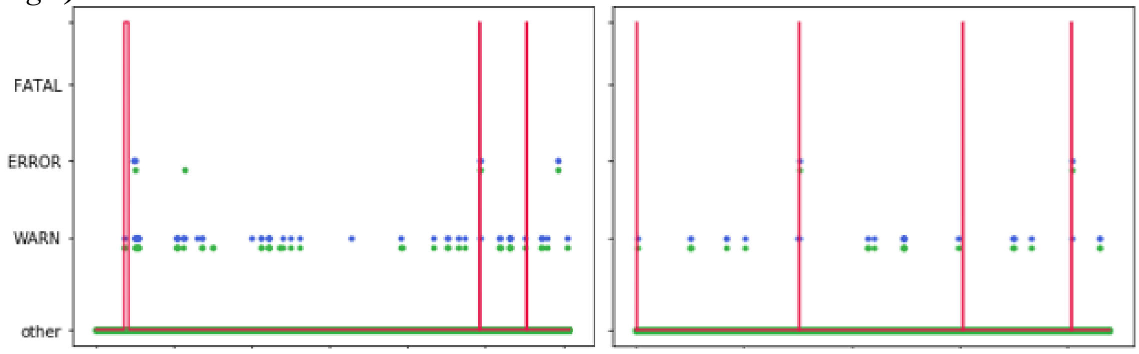
log ***b)***



log ***c)***



log **d)**



Figures 5 Outage-error correlations, dataset 1 (left), dataset 2 (right), outages of core 1 (red), log levels of core 1 (blue), core 2 (green).

It is observed that there are simultaneous fatal errors occurring in multiple logs, especially prior to index 50000 of *dataset 1* and 20000 of *dataset 2*. This could indicate that the following outage is caused by these errors. However, as there is distance between the two, the behavior resembles a system restart more than a crash. In the opposite direction, there seems only to be some correlation in logs **c)** and **d)**, where *error*-levels follow an outage straight. Nevertheless, there is still not enough visible causality in either direction so that any separation could be done between the types of outages. It is hence decided to neglect the separation. Instead, all outages detected by 4.3.1 are considered as same. Therefore, the target variable designed in 4.4.3 should focus more on log levels and less on outages, as one cannot distinct whether these are normal or error-caused.

4.3.4 Outage labeling

The one-class labeling of outages is performed followingly:

```
iterate table by rows:
    if row is outage:
        iterate row by values:
            if feature of value is categorical:
                replace value with 'null'
            else if feature of value is numerical:
                replace value with 0
```

It is acknowledged that the numerical label 0 is likely to corrupt the data. However, since there is no other plausible numerical representation for outages, the goal of the preprocessing is changed towards minimizing the usage of metric features, that depend on numerical outage representation. The labeling is also applied to *filebeat*-tables. Although being mostly categorical, they contain some numerical features that require outage labeling with 0. In 4.4.2, a visual method is proposed for eliminating problematic features, some of which are caused by this outage labeling process.

4.4 Feature selection & extraction

After treating missing values in the data, the next process is to apply FS and FE. Since the original feature-set is already filtered from knowledgably redundant features in the data engineering project, each feature left in the datasets is considered potentially valuable. However, as described, the analysis variables should focus on core features, as they host the main processes of *client B*'s operation. Therefore, a tighter approach is taken on features from the supplementary hosts.

4.4.1 Supplementary features

Among the 3 supplementary hosts (see 4.1), the *secure relays* (SRs) differ from every other host in that they lack the collection of *jmx-beat*. Because of this, one cannot use the host-wide outage detection presented in 4.3.1. Since mere *metricbeat* yields too heavy missingness, it is decided to also drop out these tables from the *SRs*. The remaining SR-features from *filebeat* are http-message-statuses and a single log level feature from each SR. The log levels are considered containing enough information of these hosts' health, so it is decided to extract these as the only features representing the SR-hosts.

Features selected from Secure relays 1 & 2:

- 2x corresponding load-balancer log levels from SR1 & SR2

The third supplementary host is *host 5*. Like with the *SRs*, only *filebeat*-features from are selected from *host 5* as well, as they are considered representative enough of the health of the host. For instance, if a metric such as memory maxes out on it, it is likely to be visible as a severe logging or outage in it. However, as *host 5* still records *jmx-beat*, the outage detection of 4.3.1 is applied on the host's *jmx-beat*, and the discovered outages are extracted as a binary feature where 1 represents outage and 0 a normal situation. The supplementary features from *host 5* are listed below. The number of all supplementary features is finally reduced to five, which accomplishes the lesser required amount, as compared to core features.

Features selected from Host 5:

- 2x supplementary process log level features
- Binary outage-labels

4.4.2 Visual feature elimination

Now that the few features from supplementary nodes are selected, the next step is deciding which features to analyze from the remaining feature set, which contains the supplementary features and all logs and metrics from core hosts. A feature elimination approach, i.e. filtering out features by condition, is chosen, rather than feature selection, i.e. selecting in features by condition.

An important factor considering ML modelling of any dataset is the static distribution of feature values. An example of this is that one must always have the same number of categories for a categorical feature. Relatedly, an image recognition model is preconfigured to analyze the same number of pixels on each training image, and to train with images of altering sizes, these must be resized to fit the model's input dimensions. This can be seen in regular dataset, for example if one-hot-encoding categorical features. The model is preconfigured for one input variable per each encoded feature in training data. If the number of categories increased in testing data, the model will not have positions for the new OHE-categories, caused by the expansion of feature space.

This issue affects especially time series modelling, where categories can expand or reduce over time due to updates in the system. In case of expansion, models cannot longer operate without either completely retraining them or neglecting the new categories. One of the occasions that cause such issues are service updates and fixes. These are especially in hand in this study, where structural changes occur regularly during the 8 months analysis period. To address this, a visual feature space analysis is proposed for detecting problematic features and discarding them from the data.

For categorical features, the data is plotted representing time in the x-axis and the number of unique feature categories in the y-axis. Each new category seen along the data is represented with a unique integer y-value. Hence, the visualization reveals the distribution of feature categories during the analysis period in x-axis. The method functions also on numerical features, for which, instead of plotting the unique categories, the actual feature values are plotted. This reveals changes in value distributions, similarly than with category distributions. Like categories, the distribution of missing values could also change during the time series. Therefore, the distributions are plotted as blue dots and the changes in outages as vertical grey lines for each lag. Below are represented some of the most significant discoveries made by running the method on *dataset 1*.

Categorical features

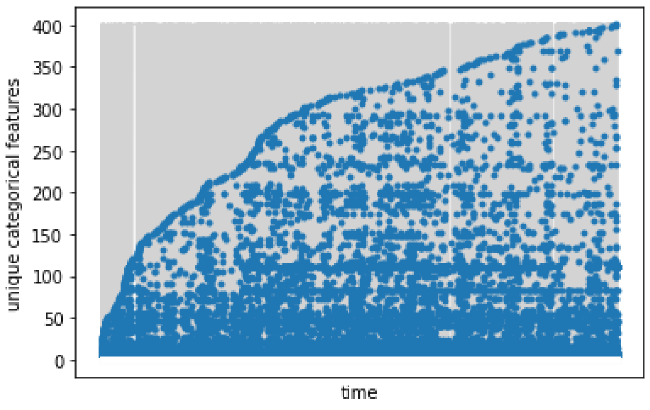


Figure 6 Feature **a)**

Feature **a)** is an example of having a problematic dynamic feature space. The mere number of categories is an issue, as for instance, applying OHE would increase the dimensionality already by 400. Another issue is the increasement of unique categories. The feature is unusable, as one cannot define the point where the increasement stops. In other words, the feature seems continuous rather than discrete. The slight slow-down after the first third of the period suggests that the most common categories have appeared in the data by that time, but the increasement still continues. The actual feature behind the figure refers to the connection points of operational messages. It is dynamic by nature, as the number of the points increases as more integrations are deployed to the operation.

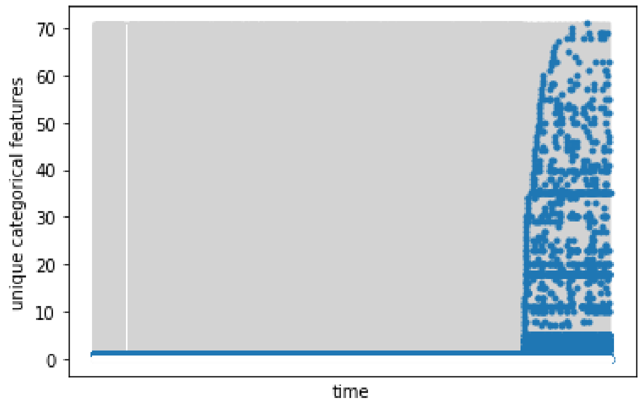


Figure 7 Feature **b)**

Another issue is seen with feature **b)**. It witnesses an update on the API-manager during the last quarter of the analysis period. The feature is not just highly dimensional, but also unevenly distributed during the period, which is expected to be problematic. If the model were trained with first three quartile of the period, it would not learn to associate the feature values to target, since the training portion would only contain value 0. If the model were trained on the whole dataset, the first 3 quartiles would likely mislead the model parameters to always expect 0 before the other values would take effect. This makes the feature likely unusable at least for *dataset 1*.

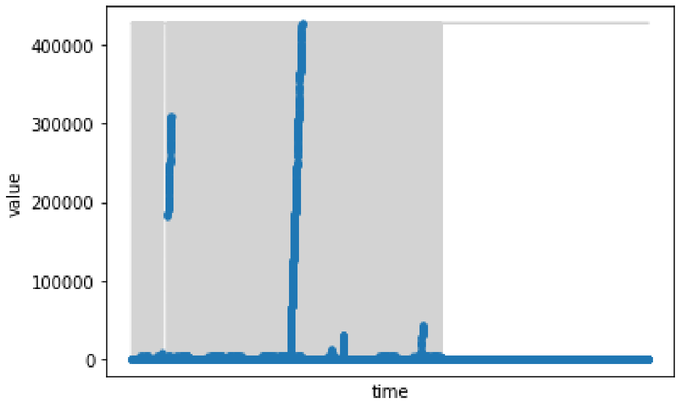


Figure 8 Feature **c)**

A similar issue than with b) is seen with feature **c)**, this being one of the few numerical features in *filebeat*. Here, a change of distribution concerns missingness, as all values after half the period are missing. This is expected to cause similar issues than with changes of value distributions with feature b). Feature c) is an example of a feature, the collection of which was stopped during the analysis period.

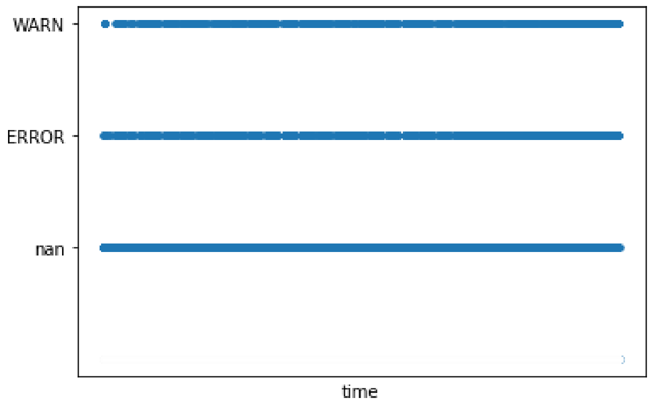


Figure 9 Feature **d)**

Example of a more stable feature is shown by feature **d)**. It is statically distributed over time, making it safe to use in the analysis. A potential problem, however, is that there it misses loggings with for example *fatal* severity. This could mean that one is yet to occur rather than the log is not generating them. To avoid conflicts later, the feature could be expanded in beforehand with a category for *fatal* errors, if deployed as part of real time warning model. Although, the model would need training to learn how to associate the fatal errors with against the rest of the data.

Numerical features

The method is also applied on the remaining numerical features from *metricbeat* and *jmx-beat* of the *cores*. It is discovered that the same issues also affect them, although with a different manner.

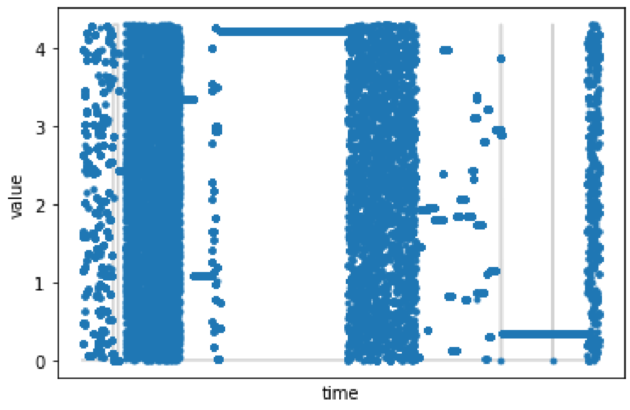


Figure 10 Feature **e)**

Feature **e)** represents the metric of outgoing network bytes, which has clearly two states of behavior: active and idle. The models are likely to have trouble learning the dynamics of both states, which makes such features expectedly unusable. This feature also shows an adverse effect of outage detection and labeling. It clearly contains outages; however, these are being treated as normal missing values, due to reasons described in 4.3.1. As outcome, e)'s outages are imputed as long-lasting static horizontal lines of the last or next active value. This is problematic as the long-lasting value is different on each outage. Hence, the models cannot associate the individual outages as an entity of same behavior, making this type of features expectedly unusable. These features would be better represented by a binary feature of *active* or *inactive* states.

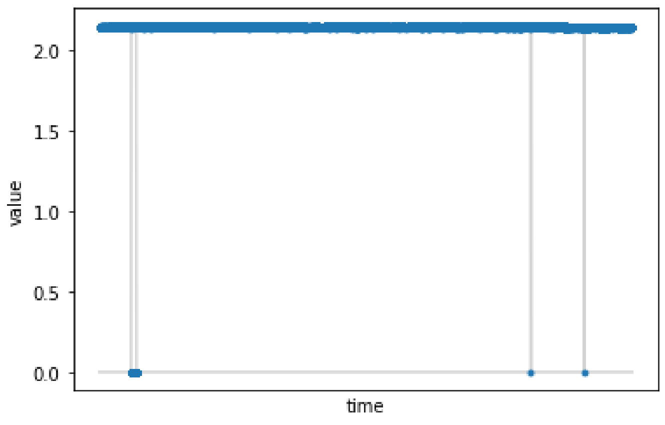


Figure 11 Feature **f)**

The final noteworthy discovery are features like **f)**. It has seemingly the same value during the whole period, if disregarding the sudden changes during labeled outages plotted as 0. It is acknowledged that the variance of f) could seem more significant if the visualization would not scale down to 0 during outages. This shows another flaw in the outage labeling process, when labeling numerical outages as zero. This type of features could be transformed to better scale, however, in this study they are discarded, since the number of features is already relatedly large. The results of the feature elimination are summarized below:

Feature elimination results:

- Both log and metric features are filtered. Each feature that has issues like **a), b), c), e)** or **f)** are discarded from analysis. The process results in 36 log and 29 metric features (total 65) for *dataset 1*. For *dataset 2,* the numbers are 35 for both logs and metrics (total 70).
- A decision is made to shorten the time series of *dataset 2*, by matching it with the update of API-manager (see fig. 7). There are multiple features of the service that would have needed to be discarded in both datasets because of the update, that are now only discarded from *dataset 1*.

4.4.3 Target variable

To apply the final step of feature selection, the target variable must be selected or extracted before. This variable should focus on the 10 main log level features produced by the integration engine in *cores 1* and *2*. The proposed solution is an overall error score variable calculated from each of the logs. This would reflect the state of the whole operation, as errors in a single log would increase the score marginally, whereas errors in all logs would increase it drastically.

As seen in 4.3.3, the individual logs have different minimum and maximum values of severity, some having the latter as *fatal* and others as *error*. Therefore, each log level is first given a numerical value representing the severity, and after that, these numbers are standardized by z-score, where each log's maximum severities are scored highest, and minimum lowest, based on the distribution of severity. It is acknowledged that for some logs, the higher severities could be yet to occur, but this issue is not focused on this study. To prevent corrupting the log level scoring, the outages are disregarded from the output variable, rather than given each log level a score like 0 during them. This is done by imputing missing values with the last occurred severity for each log, like the imputation in 4.3.2. The whole process of error score is presented as algorithm below:

```
1. iterate log level feature by values (repeat for each log level):
    if value is 'null':
        replace value with last preceding or next following value
    else if value is not 'null':
        replace:
            'WARN'  by 1
            'ERROR' by 2
            'FATAL' by 3
             others by 0
2. z-score standardize values along each feature
3. concatenate the features into table (1 column/feature).
4. initialize final error score variable as list
5. iterate the concatenated table by rows:
    count sum of of all values in the row
    append the sum to error score list
```

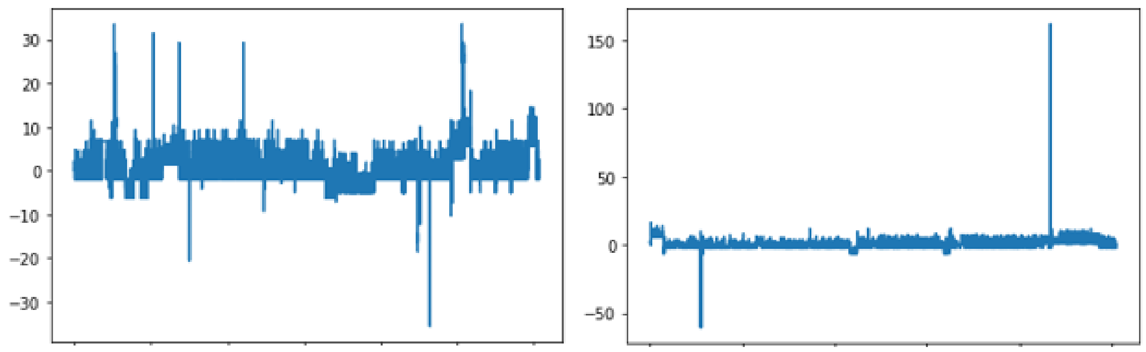


Figure 12 Initial error score visualization. Dataset 1 (left), dataset 2 (right).

The generated error scores are plotted in fig. 12. With *dataset 2*, the frequency of more severe loggings is very sparse for some logs, which makes their scores very high after standardization. With *dataset 1*, the variable seems to vary in more stable manner. To avoid the high variance, it is decided to transform all scores higher than 20 to 20 and scores lower than -20 to -20. This smoothens the plot, which makes the predicting better, especially if the forecasting method is easily biased by outliers in data. Although reducing the error scores of the few very high instances, the transformation keeps the score relatively high to trigger a warning, which is essentially the only requirement for the error score. The transformed error scores are plotted in fig. 13. A potential threshold for triggering warnings is visualized as a red line.

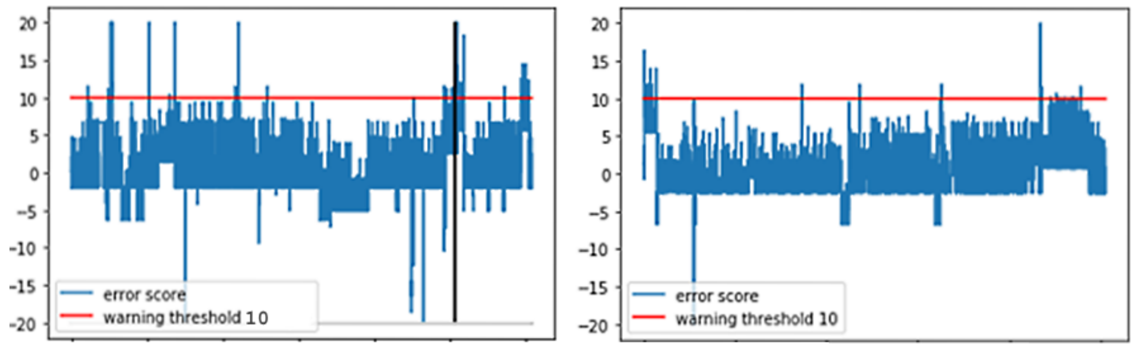


Figure 13 Transformed error score visualization. Dataset 1 (left), dataset 2 (right). The black column indicates the starting point of dataset 2 within dataset 1's span.

4.4.4 Feature subsets

After the feature elimination, some FS-procedures are still required before forecast. A reason for this is that some forecasting methods are computationally complex to apply on highly dimensional data. To experiment as many types of forecasting methods, multiple subsets are created, so that a suitable number of features are available for each candidate method. The methods that cope with higher dimensionality could also be applied on the smaller subsets to see whether more features improve or worsen the forecast. The following 3 subsets are decided to be provided.

Subset 1 **– All features, binary encoded**

As the first subset, it is decided to use all variables left, after the elimination process in 4.4.2. Some of the categorical features have a high number of categories, so applying OHE would increase the dimensionality heavily. This increase is lightened by using [25]'s binary encoding instead. The resulting numbers of features are 142 for *dataset 1* and 169 for *dataset 2*. As a comparison, the corresponding numbers with OHE would have been 234 and 341.

Subset 2 **– Expert defined (manual) features, one-hot-encoded**

The second subset of features is selected manually based on expert-knowledge of *company A*. These are expectedly the most impactful features in the dataset. Significant metrics such as CPU, memory, and network usages are selected, along with most important log sources such as each log level feature, and additional features especially produced by the integration engine. For *dataset 2*, also features from the API manager are selected, that were discarded from *dataset 1*. All supplementary features are also discarded from this subset since they are expected not to affect the health of the *cores*. OHE is applied on categorical features, which results in a total of 122 features for *dataset 1* and 146 for *dataset 2*.

Subset 3 **– Filter-reduced features, one-hot-encoded**

As the third subset, a reviewed FS algorithm is experimented. Since embedded methods restrict the models too much, and wrapper methods would be too heavy to validate, a simple filter method is experimented, based on feature correlation coefficient (see 3.2.2). Since lacking a stable implementation of [21]'s FCBF, Pearson's correlation coefficient is used for selecting *k* most positively or negatively correlated features to the target variable. Since the target is continuous rather than categorical, one cannot use the chi-squared selector for categorical variables. Instead, a custom correlation check is proposed for these, applying OHE and measuring correlation between the encoded features and the target variable. Instead of selecting *k* most correlated OHE-binary-features, each original feature is encoded individually, each encoded features' correlations are calculated, and the average of these correlations are saved as the original feature's *correlation score*. These scores are then compared, and *k* best scoring features are filtered in. Finally, the selected categorical features are applied OHE, and joined with the most correlated numerical features, selected with the regular correlation measurement.

A property of *subset 3* is that it is aimed to be smallest, with preferably around 50 features total after encoding. Since both other subsets contain over 100 features each, it is likely that *subset 3* is the only compatible one for computationally complex models other than neural networks, for reasons described in chapter 6. The portion of categorical features are selected arbitrarily as 10, and the corresponding number for numerical features as 8. The number is bigger for categorical features such as log levels, since they are expected to be more important. After encoding, their amount increases to 38, for both datasets 1 and 2. From both categories, this makes a total of 46 features for the whole subset.

As a side note, there is a potential disadvantage in measuring correlation between features and target of the same lag. Since the data is prepared for a forecast, the selection could be more efficient, if the correlation were calculated between input features of current time and target variable of the predicted time or future. However, this hypothesis is not concerned in this study, and left for future experiments.

This marks the end of part I, now that all preprocessing aspects have been considered, and experimental preprocessing subsets have been constructed.

Part II

Time series analysis

5 Background

5.1 Definition & categorization

Time series are two-dimensional datasets, where every instance i.e. point, observation has a timestamp along with a feature i.e. field, attribute that is measured against time. Time series are commonly used for visualizing the evolution of some feature of interest, such as the temperature and stock prices. Time series data can be univariate monitoring a single feature, or multivariate monitoring multiple evolving features against time. The main distinctive feature of time series compared to other data types is the temporal correlation i.e. *autocorrelation*, which means that subsequent values in the dataset are correlated. This also means that changing the order of instances in a series completely loses this correlation, making the series a regular collection of single-variable data. There are certain noteworthy attributes specific to time series type of data. Below are listed the main attributes, as described by [32]:

- **Stationarity** indicates that the probability laws within the data do not change in time. Examples of stationary time series are computer metrics like CPU or disk usage, that are always measured between 0 and 100 % despite temporal variance.
- The opposite of stationarity is **trend** i.e. non-stationarity, which defines how the probability laws change over time (fig. 14, blue). [32] divide trends into two types; **deterministic** which is expected to be constant, and **stochastic** which is expected to change over time.
- **Cyclicity** or **seasonality** refers to changes in time series that repeat regularly (fig. 14, green). An example is ice cream sales, that increase on summer and decrease on winter, unrelatedly on the development of annual sales amount.
- Like with any other data type, individual instances that relatedly differ a lot from average, are referred as **noise** (fig. 14, red).

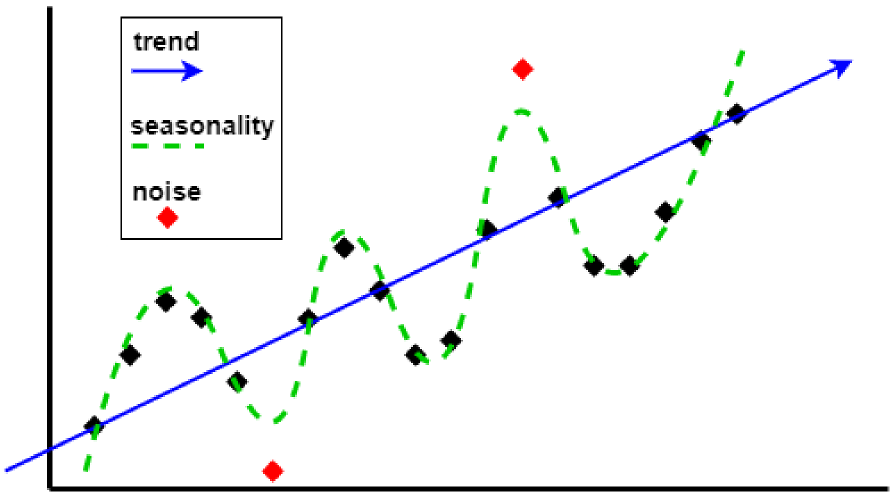


Figure 14 Basic visualization of time series is achieved by plotting the instances in chronological order in a two-dimensional graph such as line chart or scatter plot. Generally, x-axis represents time and y-axis value.

Time series analysis covers any sort of visual or statistic perception of time series. Conceptually, it can be divided into understanding, control and forecast. Understanding refers to recognizing a series' definitive attributes listed above. Control means observations that can be made of a series, after understanding its attributes. An example of this is analyzing whether a change occurring in a series indicates normal or erroneous behavior. Lastly, forecast means analyzing past instances to predict future ones. Most publications of time series analysis focus on forecast, but there are also fields more related to time series control, such as anomaly detection. This study, however, focuses on forecasting techniques.

Time series forecast can be categorized into univariate or multivariate by its analysis and forecasting types. Univariate analysis is limited to analyzing and predicting a single series, whereas multivariate analysis enables calculating multiple series and inter-series relations, that is, whether a change in one series affects another. Univariate forecast, on the other hand refers to predicting the future of a single series, whereas multivariate forecasting predicts futures of multiple series at once. Multivariate forecast can be accomplished with any multivariate analysis method by repeating the forecast for each variable of interest. However, it can be inefficient if targeting many series or complex methods, which must be considered when selecting best models for a task.

5.2 Traditional methods & terms

As described, most publications of traditional time series analysis concern predicting future values i.e. forecast. The practice of it has established into a three-phase process of ***a)*** *developing a predictive model,* ***b)*** *fitting the model on a dataset,* and ***c)*** *predicting next instances*. The term model comes from the fact that the methods aim to replicate i.e. model the dynamics of the series as well as possible. Most traditional methods utilize a set of common terms or components, based on the works of Box and Jenkins [3]. Some of the key terms from the theory are summarized below:

Lags

Lags i.e. lagged values or lag distance denote how many prior instances of a time series are calculated in a process. Lags are commonly represented with the *t-n* notation, where *t* stands for time now and *n* for the number of timesteps before *t*. For example, if a time series had the density or framerate of one instance per day, the day one week ago would be notated as *t-7*. As common in forecasting, lag *t* is the first predicted instance, whereas `t-1,t-2,…,t-n` are the lags analyzed for predicting *t*.

Autocorrelation

The basic property of any time series is autocorrelation (AC). It refers to a time series instance's correlation to each preceding instance in the series. An example of this is today's weather temperature, which is generally strongly correlated to yesterday's temperature and less correlated to the temperature of for example one month ago. This difference can be formalized as `AC(t-1) > AC(t-7)`. A common derivative of AC is *partial autocorrelation* (PAC). PAC refers to each individual lag's (`t-1,t-2,…,t-n`) correlation to *t*. Instead of calculating the straight AC for example between *t* and *t-2*, the AC of *t-1* is subtracted, leaving over just the residual correlation between *t* and *t-2* [32,3]:

```
PAC(t-n) = AC(t-n) - (AC(t-1) + AC(t-2) + ... + AC(t-(n-1))
```

Differencing

A key process of traditional time series analysis is differencing. Differencing removes trend and seasonality, transforming a non-stationary time series to stationary. The basic differencing, aka. *lag 1* difference is performed by subtracting the value of the previous instance from current instance:

```
diff(t) = value(t) - value(t-1)
```

In *lag-2* difference, instead of the preceding, the second to preceding instance is subtracted:

```
lag_2_diff(t) = value(t) - value(t-2)
```

A common approach for choosing the optimal lag distance is to match it with the length of a cycle in a time series. This leaves the series without the cycle or seasonality; however, it might not yet make the series stationary [32]. In such cases, the *order* of difference can be increased. Order denotes how many times the series is differenced. The first difference is the output of a differencing applied once, and the second difference is the output of a differencing applied to the first difference. Multiple differencing is usually applied when a trend is polynomial or exponential [32].

If a forecasting model functions with differences rather than original data, the predicted values will also be in the differenced scale. These values are straight-forward to convert back to original scale. This process is also known as *inverting* [33]. Below is an example inversion from a first order lag-1 difference:

```
value(t) = diff(t) + value(t-1)
```

Where AC is a definitive property of time series, it is also problematic concerning forecast. Since lag *t* is usually heaviest correlated to its first preceding lag *t-1*, forecasting models easily overfit to this lag, meaning that a predicted *t* is always relatively close to *t-1*, despite other lags would suggest the opposite. This can make the forecasting accuracy overly optimistic, providing relatively good accuracy, despite the only thing that the models do is replicating the last instance in the series [34]. What differencing does, it evens the relation of *t* to more distant lags than *t-1*, making the models less overfit to this.

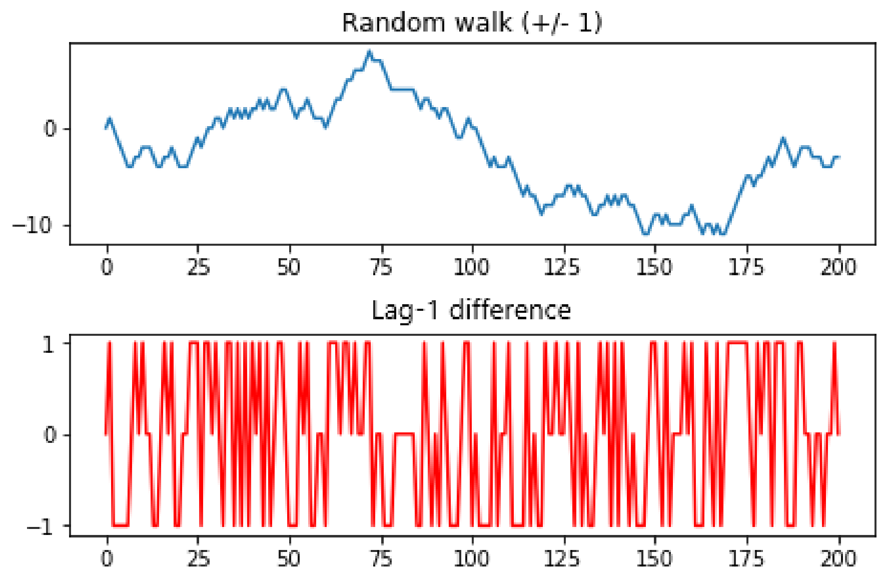


Figure 15 Illustration of a time series (top) and it differenced (bottom). Random walk refers to a series where each instance is generated randomly, either 1 higher, 1 lower, or same as the preceding instance. As term differencing implies, the y-axis in the bottom figure represents the differences between sequential instances, making the scale only vary around between -1 and 1.

Autoregression & moving average

Two of the most essential forecasting models are autoregression (AR) and moving average (MA) by [3] and [35]. In addition to being used for forecasting univariate time series themselves, these two are key components of more complex statistical models.

In AR, a prediction is calculated based on the preceding values in the series. PAC-function is used first to find the negative or positive correlation of prior values. This reveals the impact of each lag against *t*. The lags are then associated with proper weights based on how they affected negatively or positively. For example, if lag *t-5* has generally highly negative correlation to the value, it should be associated with a corresponding high negative weight. [32,36] The AR(*p*) formula is represented below, where parameter *p* denotes how many lags are weighted and used for calculating the predictions:

$$y(t) = \mu + w1 \cdot y(t-1) + w2 \cdot y(t-2) + \dots + wp \cdot y(t-p) + e(t)$$

* y = lagged values; y(t) = prediction, μ = average of series, w1-p = weights for lagged values, e(t) = random error term for prediction.

In contrast in MA, a prediction is calculated based on the errors in preceding lags, i.e. how much each value differed from the average in the series. MA utilizes AC for calculating correlation of the preceding errors. Since PAC usually cuts down quickly due to low residuals after the first lag, it misses possible long-distance correlations, whereas AC does not. Hence, bare MA is usually better than bare AR in such situations. Like AR, MA applies weights to the lagged errors, emphasizing the effect of each lag. The MA(*q*) formula is presented below, where parameter *q* denotes the number of lags similar to *p* in AR:

$$y(t) = \mu + e(t) + w1 \cdot e(t-1) + w2 \cdot e(t-2) + \dots + wq \cdot e(t-q)$$

* e = lagged errors; e(t) = randomized error term for each prediction.

ARMA-models

The most widely used statistical forecasting models are ARMA-models, which compound both AR and MA components for prediction. The basic ARMA(*p,q*) is simply a combination of the previous models:

$$\begin{aligned} y(t) = {} & \mu + W1 \cdot y(t-1) + w1 \cdot e(t-1) + W2 \cdot y(t-2) + w2 \cdot e(t-2) + \ldots + Wp \cdot y(t-p) \\ & + wq \cdot e(t-q) + e(t) \end{aligned}$$

* W1-p = weights for AR lags, w1-q = weights for MA errors.

The basic ARMA is applicable for univariate forecasting of stationary time series. However, an enhanced version of it is ARIMA (AR-integrated-MA), which adds an additional step of differencing in the model, thus capable for non-stationary time series. The formula is ARIMA(*p,d,q*), where *d* denotes the order of difference applied for the original series before forecasting. [32]

As a parametric model, the performance of ARMA-models is often dependent on selecting the correct parameter values for *p*, *d,* and *q*. A common method today is to perform the parameter selection automatically. For example [37,38] use the term auto-ARIMA for library code implementations that optimize the parameters for given data. These methods use the Akaike and Bayesian information criterions for selecting best candidate parameter set for given dataset.

Vector autoregression

Although there are many variations of ARMA and ARIMA, a downside of each of them is lacking the capability of utilizing inter-series correlations. Vector autoregression (VAR) is the most common ARMA-affiliate, applicable for multivariate statistical forecast [39]. In fact, VAR is a straight derivative of basic AR; however, instead of having a term for each lagged value, it has vectors representing the corresponding lags for each individual series analyzed [39]. The formula of VAR(*p*) model for two correlated time series is represented below. Like with AR, *p* denotes the number of lags concerned. As shown, the equations of multiple AR models can be transformed into single VAR model by converting them into matrix multiplication of weights and lagged values.

$$a(t) = \mu(a) + w11 \cdot a(t-1) + w12 \cdot a(t-2) + \ldots + w1p \cdot a(t-p) + e\mathit{a}(t)$$

$$b(t) = \mu(b) + w21 \cdot b(t-1) + w22 \cdot b(t-2) + \ldots + w2p \cdot b(t-p) + e\mathit{b}(t)$$

$$\updownarrow$$

$$\begin{array}{rl} & [\mu(a)] + [w11\ w12\ \ldots\ w1p] \cdot [a(t-1)\ a(t-2)\ \ldots\ a(t-p)] \\ f(t) = & [\mu(b)] + [w21\ w22\ \ldots\ w2p] \cdot [b(t-1)\ b(t-2)\ \ldots\ b(t-p)] + e(t) \end{array}$$

* a,b = lagged values of two individual time series. The predictions a(t) & b(t) are transformed into single vector f(t), as well as error terms e*a*(t) and e*b*(t) into single e(t). w11-1p = weights of series a, w21-2p = weights of series b.

One of the biggest advantages of VAR is the straightforward forecasting of multiple output variables. As described in 5.1, most methods can only predict one variable per model, achieving multivariate forecast only by repeating the process for each predicted variable. The base VAR is the most common traditional multivariate forecasting method. However, it is also extendable into more sophisticated methods such as VARMA or VARIMA, by applying MA components and differencing [39].

Concludingly, a property of most statistical forecasting methods is that they are mainly concerned predicting single steps into future. This length of the prediction period is also referred as forecast horizon [39,40]. However, a longer horizon is applicable in by iterating, by first predicting a single step in future, and then utilizing this prediction in next iteration as lag *t-1*. If a time series evolves gradually, one can predict a longer horizon, but in case of unstable and highly varying time series, it is difficult to predict more than one step in the future. Some scenarios enable the correction of falsely predicted instances with the real outcomes before making next predictions. [41] refers to this as multi-step forecast with re-estimation, as the method is re-estimated between predictions. This approach is desirable for any forecast if possible.

5.3 Machine learning

Like traditional methods, most ML methods concern especially forecast. Plenty of papers have been published of ML forecasting methods, which mostly represent supervised learning, by regression of time series values, or classification of behavioral classes. However, there are no well-established benchmark datasets like the MNIST [42] in computer vision field, for comparing the experiments globally. This is likely because different forecasts vary heavily by number of lags, horizon length and other related variables, whereas a similar image recognition model can be applied to basically any collection of images of desired resolution. Because of this, many proposed ML methods are compared against traditional methods like ARIMA and VAR. This is also because the statistical methods have been established as the standard of forecasting, and ML has been concerned relatively lately.

5.3.1 Neural networks

The first major publications of ML for time series date back in the 90s, when artificial neural networks (ANNs) were a public interest. The first experimented network is multi-layer perceptron (MLP), which is the most basic feed-forward neural network structure. The MLP's components are input layer, hidden layers, and output layer. Input layer consists of input nodes, which are basically the feature values of a given input sample or instance. Between input and output layers there is an arbitrary number of hidden layers, each containing an arbitrary number of hidden nodes. Each node has adjustable weights, one per connection from previous layer, plus an additional bias weight. At the end of a node is an activation function, used for *squashing* the output into a desired scale. An input vector entering a node is modified by the corresponding weights and activation, before feeding forward to the next layer. In the end of the model is the output layer, which is either a single node representing the predicted value in regression or multiple nodes representing one output class per each in classification. In classification approach, the class prediction is denoted by which output node has the highest output value i.e. activation. The MLP is trained by the combination of *forward propagation* and *backpropagation* (BP). Forward propagation refers to feeding an input through the network producing and output, whereas BP adjusts the weights of the layers, based on how correct this output is. This adjustment is propagated backwards in layers by chain rule, until finally modifying the first hidden layer's weights. The optimization process of a weight is called *gradient descent*, as the error denoted by the loss function descends towards minima.

Among the first published papers is [43]'s study comparing ARMA and MLP for forecasting monthly flour prices in three US states. They proposed a network for which the input variables are directly the lagged values of the series. For example, a lag distance of 6, would make this network's input consist of 6 nodes. They used a single hidden layer, matching the number of hidden nodes with the number of input nodes. The output is a single regression output, evaluated by mean squared error (MSE) for backpropagation. They propose a couple of network architectures, including univariate models for predicting each state's price separately and a multivariate network. The latter would use lags from every three states as input, and the output would still yield a prediction of just a single state's price. Both ANN-approaches reportedly outperformed ARMA in prediction performance, but the best result was achieved by the multivariate network. This especially indicates that ARMA-models are inadequate whenever multivariate correlations are exploitable.

Another type of ANN is presented by [44] and [45], applying the method for stock market time series forecast, formulated as classification. The ANN model is proposed by [44], whereas [45] presents comprehensive comparison of the model and other ML approaches for the task. The network takes as input the latest average stock price *t-1* alongside 4 statistical derivations, such as daily rise and fall rate, moving variance of this rate, and the ratio of moving variance. They also use a single hidden layer consisting of 7 nodes. The output layer consists of three market-behavioral classes, that are stable, unstable and crisis. The network reportedly set a benchmark for modelling the Korean stock market index from around a 1997's crisis period. Furthermore, in [45]'s comparison, the network was one of the best performing against methods like logistic discrimination, and SVM. However, since the network only utilizes the latest lag's features, it cannot explicitly model temporal correlation of multiple past values. In contrast, the network seems to learn the pattern of behaviors, i.e. which features denote stable and which features crisis period, but not the temporal evolution in the series. This assumption could be verified by testing if the performance persists even after changing the order of the instances in the dataset.

As a conclusion, it seems that basic ANNs may still be considerable methods of choice for time series forecast, even without exploiting temporal correlation.

5.3.2 Support vector machines

Another frequently experimented ML-method is SVM [46]. Originally a binary classification method; SVM forms a multidimensional border that divides input data into two classes. The border is set to maximize the margin that separates the two classes, where margin means a constant length from the border to closest instance in each side (see fig. 16 - left). This behavior is trained by penalizing on instances that fall within the margin. If the two classes are separable, one could fit a border that has a margin which only touches one point of each class. However, in most cases the two classes are inseparable. Therefore, one must allow instances to persist between the margins. The term *support vectors* refer to the instances falling within the final margin. The number of vectors is controlled by *C*, which is the penalty term. The higher *C*, the more penalty is given for multiple support vectors. However, a high *C* is more likely to overfit to training data and lose on generalization performance. In addition to *C*, another parameter of SVM is the kernel function, that defines how to project original data to the higher dimension.

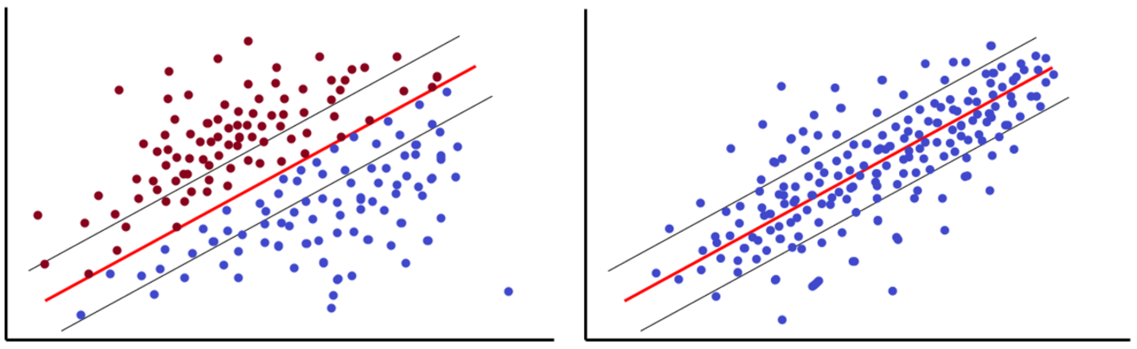


Figure 16 SVM classification (left) & regression (right). Red lines indicate the borders and black lines the margins.

The SVM-variant related to time series forecast is support vector regression (SVR). It differs from regular SVM, as the goal is not to separate classes but to fit support vectors to model the training data. Conceptually, where SVM maximizes the margin between two classes, SVR applies minimization of margin, so that it still contains most training instances (see fig. 16 right). Where in regular SVM the border defines the distribution between two classes, in SVR the border is a function of predicted output values, given some input variables. SVR has an additional $\mathcal{E}$-hyperparameter which controls a threshold of how much to fit the border on the training data. This threshold is sometimes also referred as tube size [40]. The correlation between $\mathcal{E}$ and the threshold is negative, as with higher $\mathcal{E}$, less training data falls within the margin of the border or the output function. In contrast, higher $\mathcal{E}$ usually provides better generalization and less overfit on training data for the model.

SVM-forecast has been studied by many others in addition to [45] mentioned above. For instance, [40] experimented SVR for forecasting univariate financial data, similar to [44,45]'s dataset. As input, they used lags *t-5*, *t-10*, *t-15*, and *t-20*, along with a 100-day average feature. A comparison was made between MLP, regular SVR with predefined hyperparameters and SVR with *adaptive parameters*. Accordingly, the adaption of regularization term C and $\mathcal{E}$ is especially impactful for SVR's performance in forecasting. Expectedly, the best model was achieved by the adaptive SVR, followed by the regular SVR that still reportedly outperformed MLP. This shows that SVM is an eligible model for time series forecast, at least with univariate datasets, and hence, less parameters requiring optimization. A similar experiment is presented by [47], forecasting univariate supply-chain demand time series with MLP, SVR and RNN. The performances are also compared against traditional methods, including AR and MA models. 5 lags are used as input variables, forecasting a predicted change in the demand for the next lag. The best models in this experiment are SVR and RNN; however, [47] consider their advantage to traditional methods insignificant, demonstrating traditional methods' continuing eligibility in univariate time series forecast.

SVM has also been studied for multivariate forecasts. [48] present a survey of SVM-strategies for functional magnetic resonance imaging data analysis. The general task with the dataset is to classify image samples as perceptual states; however, these datasets also form time series allowing forecast as well. [48]'s dataset differs a lot from the previously mentioned, since it is not only multivariate but also highly dimensional. Generally, SVM suffers from computational complexity and high-dimensionality issues (see 3.2.2). However, [48] report SVM as viable solution for classification, if attention is given to the feature selection process. Moreover, for regression forecast, they propose an SVR-variation originally by [49], known as the relevance vector machine (RVM). It is Bayesian learning based, sparser and more compact model, reportedly achieving still a similar performance to SVR. Another of its benefits is that it does not require validation of hyperparameters C and $\mathcal{E}$, as opposed to regular SVR.

5.3.3 Recurrent neural networks

The category of neural networks especially designed for modelling sequences are RNNs. They are trained with so-called *backpropagation through time* (BPTT) which means adjusting the network weights based on not just current, but also on preceding instances in a time series. This is implemented in network node level, by keeping a previous iteration's node output values in memory and combining them with current node inputs to calculate predictions. A practical difference between regular ANNs and RNNs can be observed when preparing their datasets. Where regular network datasets are essentially two-dimensional tables, consisting of *instances x features*, RNN-datasets are transformed into three dimensional tables aka. tensors that consist of *instances x timeframes x features*. The timeframes-dimension defines the lag distance of the analysis. For example, if the number of timeframes is 6, a single lag's features will populate 6 instances' timeframes, once as *t-1*, *t-2*, etc., and finally *t-6* for the last instance that covers this lag. Furthermore, where ANN-nodes are given input as single lag of features, RNN-nodes are given all lags in the sequence per each forward propagation. The hidden state is updated during the iteration of a single feature set sequence, but the last state can also be kept for the next feature set's first iteration. [50]

RNNs itself have originate in works of [51], but for example [43] considered training them too computationally complex during their experiments in 1990. Therefore, the RNNs are often considered deep learning i.e. deep neural networks. One of the first studies of RNN forecast is [47]'s paper in 2006, which is also discussed for their SVM approach. Like with SVM, they are unable to gain significant advantage to traditional methods with their RNN. In addition to computational complexity, a key issue with basic RNNs is vanishing and exploding gradients [50]. In practice, vanishing gradients means the inability to learn long-term dependencies, i.e. dependencies on distant lags, and exploding gradients that some node weights get adjusted overly high, making the whole nodes futile. More recently, technical improvements in parallel computation have facilitated the computational complexity problems of ANNs in general. Furthermore, improved network technologies have been developed for overcoming the gradient issues. Network architectures such as LSTM and GRU have shown more significance especially in the language processing field [52]. The pivotal difference against basic RNNs is the capability of learning long-term dependencies.

The main difference between LSTM and basic ANN and RNN is that in the place of hidden nodes there are so-called *memory cells*. They are much more complex than regular network nodes, containing for example multiple activation functions per cell, as compared to single function per regular node. In addition to keeping the outputs of previous iteration like regular RNNs, the cells maintain a *hidden state* or *cell state*, making predictions based on previous output, current input, and the cell state after previous iteration. An iteration of a cell can be divided into 3 following steps, starting from an input value vector fed to it:

1. **Forget** – Determine whether and how much to forget the cell state based on previous output and current input.
2. **Update** – Determine whether and how much to update the cell state (after forgetting), based on previous output and current input.
3. **Predict** – Combine the updated cell state with the previous output and current input to make a prediction.

Like with regular network nodes, the input of a cell is generally a vector of features of an instance. The state variable is essentially a vector of the same size. Hence, the operations between current inputs, previous outputs, cell states and cell functions are essentially vector operations like adding and multiplication. An LSTM cell has four main functions, also referred as gates [4]. Each gate is associated with adjustable weights, and hence can be considered as individual neural networks. The forget-step operates with *forget*-gate, which is a *sigmoid* activation function. The update-step operates with *update* (*sigmoid*) and *input* (nonlinear transformation function e.g. hyperbolic tangent i.e. *tanh*) gates. Finally, the predict-step uses *output*-gate (*sigmoid*) and a similar nonlinear transformation of the final cell state, for making the prediction. Below is represented a more detailed illustration of the process (fig. 17):

- `h(t) & h(t-1)`: cell state after current and previous iteration
- `y(t) & y(t-1)`: prediction of current and previous iteration
- `x(t)`: current cell input

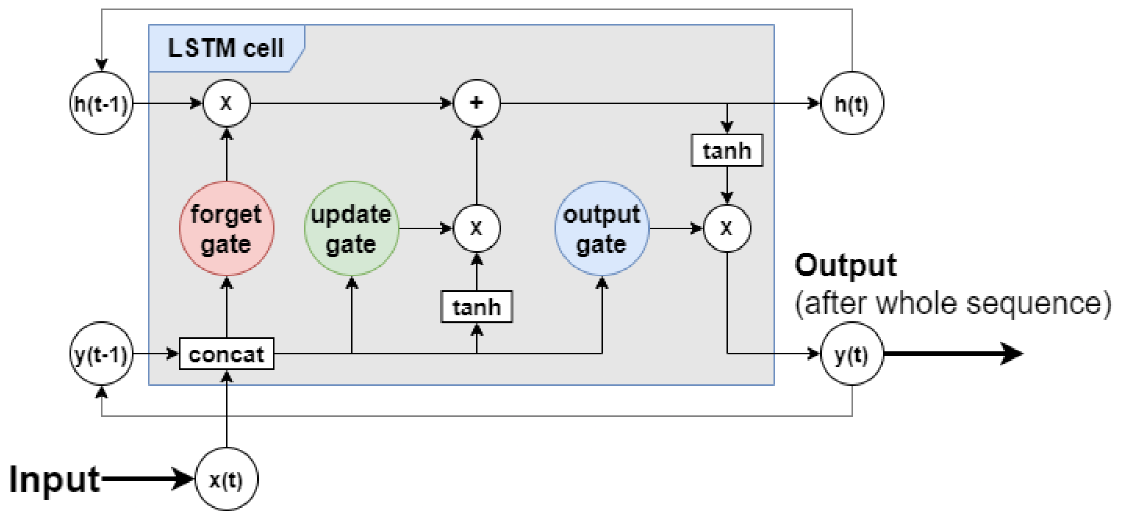


Figure 17 LSTM-cell. After producing output, the h(t) and y(t)are passed as h(t-1) and y(t-1) for the next iteration.

A single memory cell is often illustrated as a chain of cells, each representing single lag of an instance. With RNNs, this type of representation is also known as unfolding. It describes the recurrence that occurs when feeding input to a cell, as one instance always contains the desired number of lags, and the cell iteration is run once per each lag. In addition to having multiple recurrent cell iterations per instance, RNNs like LSTM can consist of multiple individual cells, such as a regular network layer consists of multiple nodes. Similarly, the individual cells are independent, each cell learning a different model of the data. Furthermore, RNNs like LSTM can be increased in depth, consisting of multiple layers of nodes or cells [50]. This is also referred as stacking [53].

Another of the common modern RNN-techniques is GRU or GRU-network. It differs from LSTM by its memory cell structure, known as the GR-unit. Instead of maintaining a separate cell state like LSTM, GRU only operates with previous output and current input. A more lightweight component than LSTM-cell; the GR-unit contains only two gates, though still achieving long-term relation learning. The GRU-iteration can be divided into following steps:

1. **Update** – Determine whether and how much to pass the previous output, based on current input.
2. **Reset** – Determine whether and how much to forget the previous output, based on current input.
3. **Predict** – Combine the gate outputs for a prediction.

The *reset* and *update* gates are essentially like LSTM's *forget* and *update*. However, the prediction step of GRU does not contain gates, but is a slightly more complex set of vector operations. The output vector of update gate is first transformed by *1– operation*, which is essentially each vector value subtracted from 1. This vector is then multiplied by previous output vector, the result being the first piece of information of output. The other piece in the combination starts also from the *update*-gate's output vector but is instead combined to nonlinear transformation of the *reset*-gate's output vector. The two pieces are combined as vector sum, which is the final output of an iteration. [50,54]

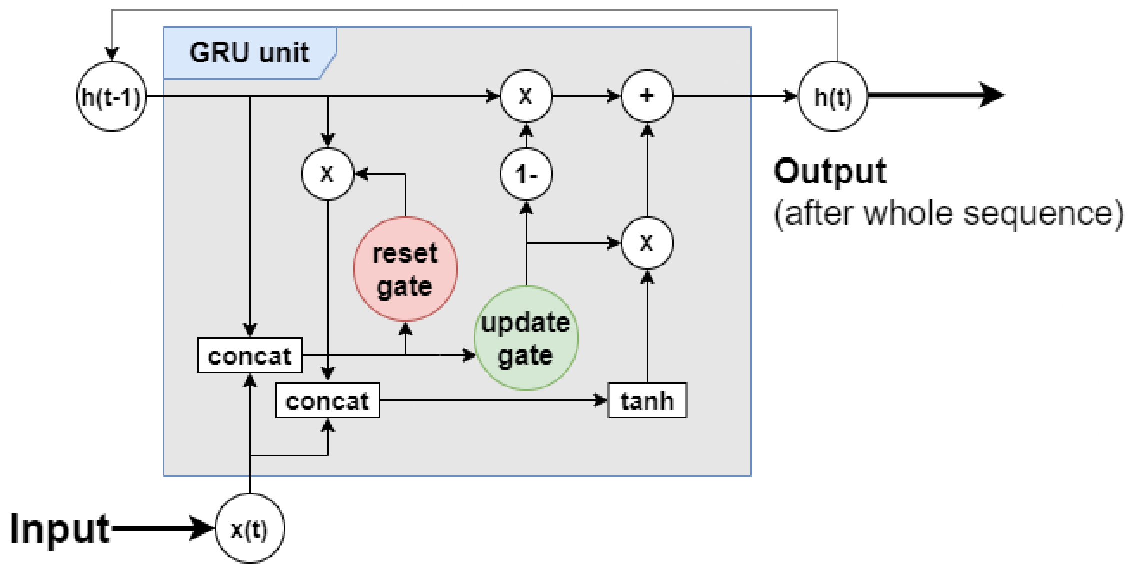


Figure 18 GRU-unit

For experiments, [29] compares both LSTM and GRU networks against ARIMA, forecasting univariate traffic flow data. They use 6 lags of 5 minutes periods, i.e. lag distance of 30 minutes to predict a regression output for 50 individual sensor values. Both of their RNN networks contain a single memory cell or unit, making the networks compact. Their LSTM predicts approximately 10% better than ARIMA, and their GRU-network even marginally better than the LSTM. More significant results are achieved by [41] in 2018. They compare LSTM and ARIMA for forecasting univariate financial and economic time series. The models are ARIMA(5,1,0) and a single-cell LSTM with 4 lags per instance. The LSTM achieves an 85 % reduction of error against ARIMA on average, making a great improvement as compared to [29].

Despite ML's capability of handling highly dimensional data such as multivariate time series, most publications concern univariate analysis. As an example of multivariate, [55] compares LSTM and VAR for forecasting multivariate aviation and climate sensor datasets. The forecast is formulated as regression of multiple features. The lag distance of VAR is chosen as the longest distance until error rate increases, and of LSTM, by a grid search of distance that minimizes error. Surprisingly, their VAR-model outperforms LSTM in the first dataset. This shows that traditional techniques are effective even in multivariate modelling tasks. Another observation of the experiment is that stacking LSTM-layers does not increase prediction accuracy.

[55] suppose the result is due to statistical methods superiority in utilizing linear relations. On the other hand, statistical methods cannot handle nonlinearities, whereas RNNs do. To address these limitations, they also experiment with a hybrid residual network model, combining the VAR-model to the LSTM. The hybrid starts with a VAR-component that outputs initial predictions that are then passed as input to the LSTM-component. Single-lag VAR(1)-model is chosen as the VAR-part. The LSTM-part is fed the VAR-output and corresponding ground truth -vectors and is trained to fix the patterns of error that VAR leaves as residual. This model shows promising results, outperforming both single VAR and LSTM models in the same experiment setup. Moreover, the residual RNN reportedly requires less memory cells per layer than basic LSTM, which makes it also faster to train.

6 Case study – Regression forecast by supervised ML

6.1 Learning task & evaluation

In part I, a client dataset of *company A* was prepared for forecast, by selecting and extracting three feature subsets and a target variable to predict. In this chapter, the prepared dataset and the reviewed forecasting techniques are experimented and evaluated. As described in chapter 2, the experiments are performed via cross-validation of forecasting methods, forecasting parameters i.e. lags and horizon, and the preprocessed subsets. This validation is performed twice, once per each *dataset 1* and *2*. There are various other model parameters to optimize, such as the number of layers in neural networks, number of neurons in a hidden layers, loss function, and regularization terms. However, it is addressed that optimizing every parameter would require excessive testing and therefore, is not focused on this study. Instead, the models are initialized with "the most standard" parameters, and further hyperparameter-optimization is left for future research. Another decision is made to not experiment with statistical forecasting techniques i.e. VAR at all. This is because of the issues shown in 3.2, as especially categorical variables and missing values would require complete omission.

The task set in chapter 2 is to predict the target variable from 15 to 30 minutes ahead. Since the point of interest is whether a warning should be triggered rather than forecasting the whole sequence, the learning task is formulated as a single-output regression of the target variable, for both 15 minutes and 30 minutes. For *dataset 1*, the corresponding lags for the minutes are *t+2* and *t+5*, and for *dataset 2*, *t+14* and *t+29*. Three ML methods are chosen as model candidates of the forecast: RNNs, basic ANN, and SVM. Although preprocessed, the data subsets are still rather complex; from 46 to 169 features per timestamp, with 60780 timestamps for *dataset 1* and 50500 for *dataset 2* (after the reduction in 4.4.2). The hardware for processing has 3,50 GHz of CPU, 12 GB of memory, and 8 GB of GPU memory. It is acknowledged that neural networks are perhaps the only processable method, especially since they can be trained utilizing GPU-processing provided by Tensorflow [56]. Therefore, mainly RNNs are focused, since they are expected to perform best. Both LSTM and GRU networks are experimented and compared to other methods.

The regression performance of the candidate models is evaluated by comparing their MSE-losses. The last 20% of both dataset periods are dedicated as the testing portions, and the evaluation is based on prediction accuracy on these. After discovering the best models for all categories, they are evaluated as a warning system. This is done by first configuring a threshold for triggering warnings, and then evaluating the correctness of warnings triggered by the models (see 6.3).

6.2 Model training & results

6.2.1 RNNs

Since capable of handling high-dimensional data, the RNNs are experimented with each of the prepared *subsets 1*, *2*, and *3*. A definitive advantage of them is that they can handle input as sequences, rather than single instances. As a comparison, a regular NN would require an input neuron for each feature of each lag. This would increase the dimensionality drastically if multiple lags were analyzed, especially if each lag contained multiple features. The RNNs are the only methods capable of sequence learning without excessive increase of dimensionality.

The sequences are prepared followingly. As described in chapter 2, one hour is considered enough lagged history for making predictions. For *dataset 1*, this results in lags from *t-1* to *t-12* for each *t*, making the data shaped *60780 x 12 x subset size*. However, to have lags *t-12* and *t+5* for all instances, the number of instances is cut to 60763 from the end, and the target variable is cut similarly from the beginning. *Dataset 2*, on the other hand, is more problematic. The full one-hour period would take 60 timestamps per instance, which makes the model heavy to train. This number is reduced by analyzing only 30 minutes or timestamps per instance, making *dataset 2* shaped *50441 x 30 x subset size*.

The same network structure is used for both LSTM and GRU, allowing a direct comparison of their performances. For simplicity, a single recurrent layer only is added. The number of memory cells is defined as the 2/3 of features in the subset, rounded to the next 10. This is because an excessive number of neurons are likely to cause overfit on training data. *Sigmoid* and *tanh* are used as the memory cell activations, like in figs. 17 and 18. The weight-update method, i.e. optimizer is chosen as Adam [57], since being a common first choice in today's practice. The networks are trained with the first 80% of the analysis periods. However, another 20% of the periods' length is removed from the training portions' ends (see fig. 19). These are used as *validation set*, for simulating prediction loss after each epoch of training. The optimal epochs count is then chosen as the point where validation loss stops decreasing, which should prevent overfit on training data. After validating the epochs, the 20% validation portions are merged into training portions, so the final models are trained with first 80% and evaluated with last 20% of the datasets.

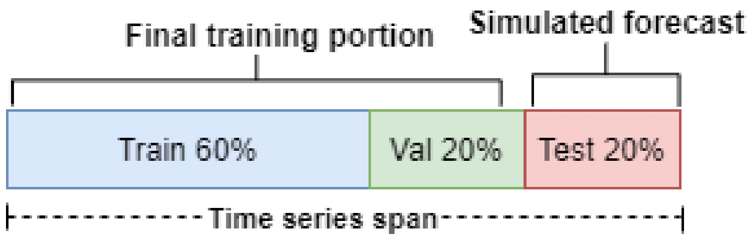


Figure 19 Training, validation, and test split of neural networks

Furthermore, before validating the epochs, a grid-search is performed on each *dataset x subset x horizon x RNN* combination, to optimize the learning rate for each. This is because the feature and lag dimensions vary per each combination. The learning rate is adjusted so that for every combination, a decreasing curve can be seen from both training and testing accuracy, before selecting the epoch count (see fig. 20) Lastly, the training is done in batches of 60 instances, which reflects 5 hours in *dataset 1* and one hour in *dataset 2*.

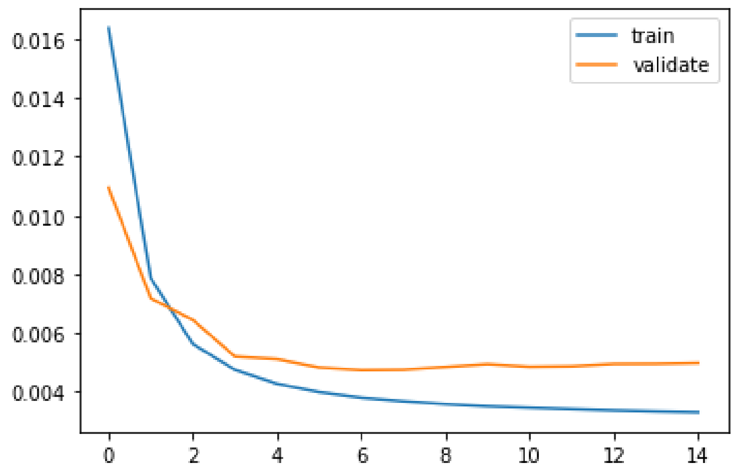


Figure 20 Example of training and validation losses (subset 1, LSTM). The optimal epoch count would be between 1 and 3 here.

Results

Already on optimization of learning rates and epoch counts, it is noticed that the validation loss decreases only little before starting to increase again. This suggests that there is not that much to learn, or no drastic increasement can be made as compared to a random guess. To find the optimal epoch counts, the learning rates are decreased by a large margin for some combinations. Another observation is that the learned patterns vary heavily for each combination, when repeating the training processes. The models might predict similar curves than the target variable on one iteration, but on second, they could predict very differently (see fig. 21). The final MSE-scores listed below are, therefore, calculated as an average of 3 iterations with the same parameters, but different random initializations:

Dataset 1

Feature subset	Rec. units	**Unit type**	L-rate *15min*	Epochs *15min*	L-rate *30min*	Epochs *30min*	**MSE *15min***	**MSE *30min***
1, all	110	LSTM	0.000001	3	0.00001	3	0.0490	0.0341
		GRU	0.00001	6	0.0001	4	0.0238	**0.0174**
2, expert defined	90	LSTM	0.00001	4	0.00001	4	**0.0211**	0.0214
		GRU	0.000001	4	0.00001	3	0.0448	0.0224
3, filter-reduced	30	LSTM	0.00001	2	0.00001	2	0.0310	0.0310
		GRU	0.00001	4	0.0001	4	0.0486	0.0176

Feature subset	Avg. MSE
1, all	0.0311
2, expert defined	**0.0274**
3, filter-reduced	0.0321

Unit type	Avg. MSE
LSTM	0.0313
GRU	**0.0219**

Horizon	Avg. MSE
15min	0.0364
30min	**0.0240**

Dataset 2

Feature subset	Rec. units	**Unit type**	L-rate *15min*	Epochs *15min*	L-rate *30min*	Epochs *30min*	**MSE *15min***	**MSE *30min***
1, all	110	LSTM	0.00001	3	0.00001	2	**0.0140**	0.0153
		GRU	0.000001	5	0.00001	4	0.0374	0.0252
2, expert defined	90	LSTM	0.000001	1	0.000001	1	0.0612	0.0611
		GRU	0.000001	4	0.000001	3	0.0384	0.0407
3, filter-reduced	30	LSTM	0.0001	3	0.0001	2	0.0149	**0.0149**
		GRU	0.0001	2	0.0001	4	0.0557	0.0460

Feature subset	Avg. MSE
1, all	**0.0230**
2, expert defined	0.0504
3, filter-reduced	0.0329

Unit type	Avg. MSE
LSTM	**0.0302**
GRU	0.0406

Horizon	Avg. MSE
15min	0.0369
30min	**0.0339**

* L-rate refers to learning rate, *15min* parameters were used when training models for predicting 15 minutes ahead and *30min* when for 30 minutes ahead.

What stands out on the results is the variance. On *dataset 1*, especially *subset 2* and GRU-network achieved the lowest losses on the testing portion. However, on *dataset 2*, their corresponding losses were highest. What is interesting, the 30 minutes ahead predictions were more accurate on average, against the 15 minutes predictions. These observations support the prior assumption that there is not much to learn in the data, and the few better results are due to occasion. An example of the variance is shown in fig 21, representing GRU-network predictions of *subset 3* on *dataset 1*.

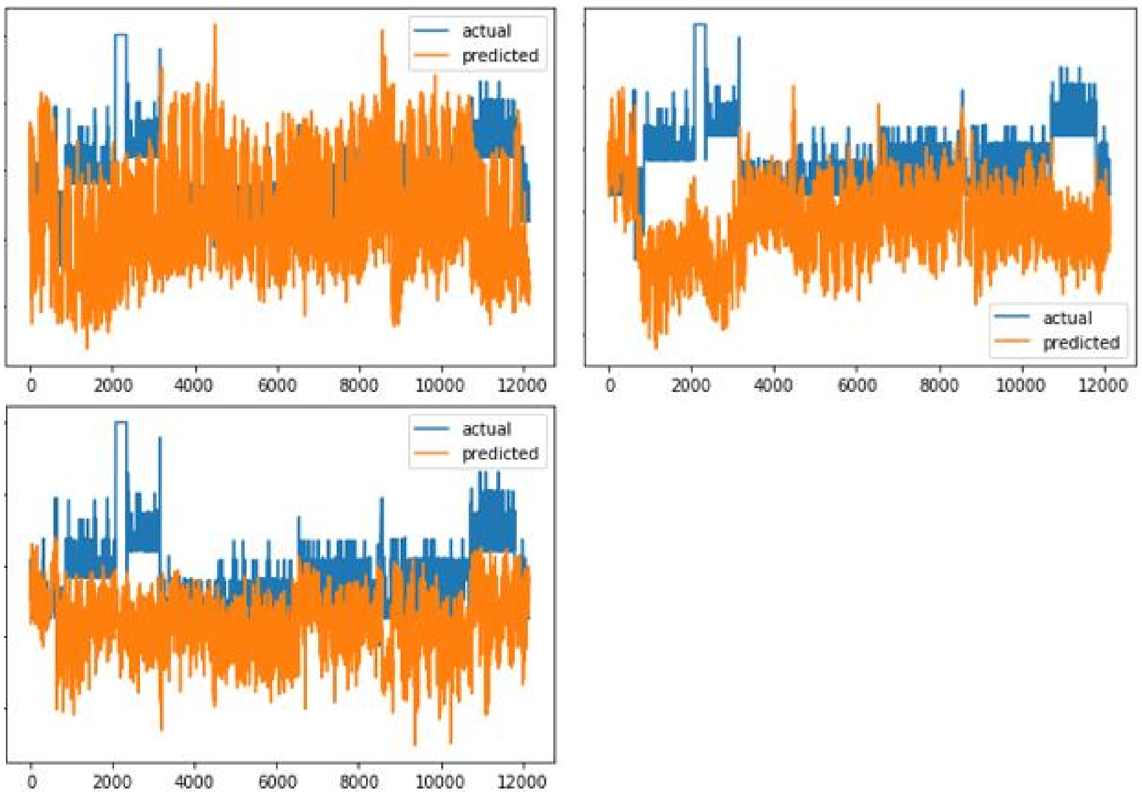


Figure 21 Example of variance in target predictions between 3 iterations.

Another observation is that the MSE of predicted target variable might not reveal the best model. In fig. 21, the lowest loss was scored by the model of iteration 3. However, almost all its predicted values are very low in contrast to actual values. Depending on the threshold for triggering a warning, this model would likely miss all occasions that would need to be warned. In other terms, the model produces purely false negatives, making it useless.

To experiment the issue further, a similar visualization is provided on the best performing combination, which is the LSTM model of *subset 1* on *dataset 2*, illustrated in fig. 22. The testing portion of *dataset 2* has one critically high error score at approximately 1/3 of the span, followed by multiple relatively high scores during the middle. The represent model would fail to all of them. To address the issue, it is decided to select the final model based on both loss and visual analysis. The optimal model should have relatively small MSE, but still follow the error score curve, at least during the critical parts that would require warning. This model selection is performed in 6.3, after experimenting with all candidate forecasting methods.

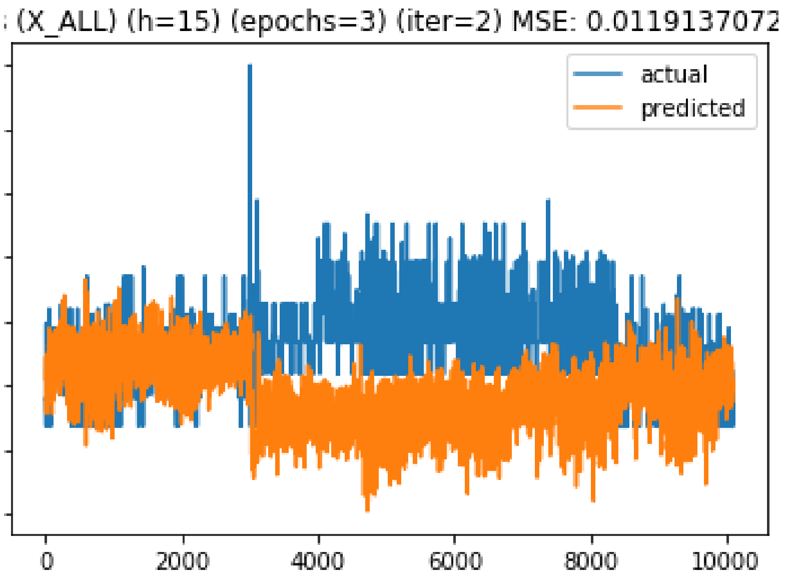


Figure 22 Forecast of best MSE-performing model iteration.

6.2.2 Basic ANN

The second experimented method is the basic ANN, without recurrent layers. It is trained with single lag per instance, which ignores the temporal correlation, focusing only on feature relations. Since the RNNs are capable of learning both, this experiment should reveal whether the prior RNNs were able to utilize the change in time. If the results with ANN are relatively similar, it can be assumed that they could not.

Since not learning sequences, the basic NN is trained faster. Therefore, an additional hidden layer is added, as depth could provide efficiency in learning complex functions like the one here. The first hidden layer is kept the same size as in RNNs i.e. 2/3 of features. The second layer is defined as 1/3 of this size. The hidden layers are activated with rectified linear units (ReLU) [58], which is a common standard choice of today, like the Adam optimizer. The network structure and training processes are kept otherwise same, also utilizing the validation portion for optimizing the learning rate and epochs count. The training is also repeated trice, calculating the average of iterations as the final loss score of the model.

Results

Dataset 1

Feature subset	H1-nodes	H2-nodes	L-rate *15min*	Epochs *15min*	L-rate *30min*	Epochs *30min*	**MSE *15min***	**MSE *30min***
1, all	110	110	0.00001	4	0.00001	4	0.0285	0.0286
2, expert defined	90	90	0.00001	3	0.00001	3	**0.0206**	**0.0208**
3, filter-reduced	30	30	0.00001	3	0.0001	2	0.0563	0.0253
							0.0351	*0.0249*

Dataset 2

Feature subset	H1-nodes	H2-nodes	L-rate *15min*	Epochs *15min*	L-rate *30min*	Epochs *30min*	**MSE *15min***	**MSE *30min***
1, all	110	110	0.0001	2	0.00001	4	0.0229	0.0412
2, expert defined	90	90	0.00001	3	0.00001	3	**0.0221**	**0.0224**
3, filter-reduced	30	30	0.00001	4	0.0001	2	0.1117	0.0583
							0.0522	*0.0406*

* Colored cells = (column) average of predicted horizon.

Based on the three-iteration average, the ANNs score higher MSE-losses on average. However, as shown in 6.2.1, this might not indicate worse generalization if considered as warning system. For ANN, especially *subset 2* seems to provide good fit. One of the best fits (*dataset 1, subset 2, 15min*) is illustrated below (fig. 23). This model seems to reproduce especially some of the relatively high error scores from the actual data's curve, that would likely be set for triggering warnings.

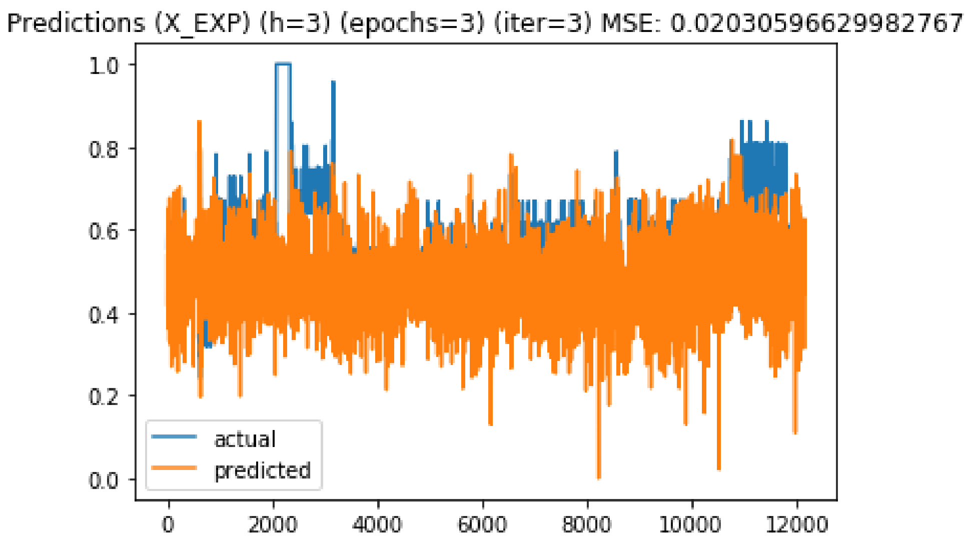


Figure 23 Example predictions from ANN

6.2.3 SVR

The final experimented method is SVM, more precisely regression by SVR. The training of SVM is slightly different from the neural network approaches. Instead of utilizing the validation portion for optimizing learning rate and epochs count, the whole 80% is used for training. This is because SVM converges to the global optimum, rather than iterating batches and epochs in search of best local optimum. The initial approach is to experiment with both regular SVR with parameter optimization and the RVM, used by [48]. However, the current available implementations of RVM are discovered to require too much memory to function even with the smallest subset. Because of this, the RVM is dropped out. Another issue is found with computation times of the regular SVR. Especially with *subset 1* i.e. all features, the training takes relatively too long to complete so this subset is discarded from the SVR experiments.

The regular SVR model is trained using radial basis function (RBF) kernel, for this being a standard choice in nonlinear regression, that this task should also represent. Due to time consumption, it is decided to optimize the two parameters, namely regularization term *C* and $\mathcal{E}$ based on couple of test iterations rather than performing a complete grid search. As the outcome of these iterations, the *C*-parameter is set to 20, and $\mathcal{E}$ to 0.0001.

Results

	Dataset 1		Dataset 2	
Feature subset	**MSE *15min***	**MSE *30min***	**MSE *15min***	**MSE *30min***
2, expert defined	0.0366	0.0433	**0.0206**	**0.0187**
3, filter-reduced	**0.0167**	**0.0170**	0.0945	0.1059

The SVR provides some better and some worse fits on the data subsets. Like basic NNs, there are great differences between the best and worst models. Especially *subset 3* of *dataset 1* provides visually one of the best results of the experiment (see fig. 24).

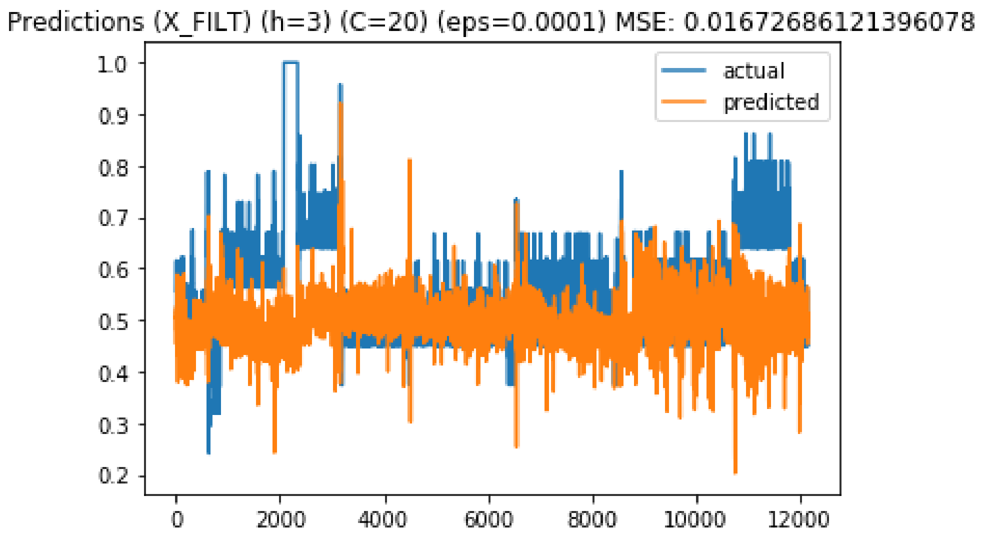


Figure 24 The best visual fit of SVR

6.3 Warning performance

The final evaluation is done on the best performing models, based on MSE scores and visual analysis of the forecast curves. Four evaluations are performed, one per each dataset and horizon combination. For each of these evaluations, the best model is chosen from all 3 forecast methods and their performances compared to each other. The neural networks are selected based on the following criteria:

1. **Best model** by MSE,
2. **If** at least one of its training iterations reproduces a similar curve to the actual target. This filters out models that have good MSE but bad generalization, caused by predicted error score staying closely around 0.5.

The average of best of three iterations are once again used as the final score per neural network, whereas the SVR converges, so only one iteration is needed. The selected models are listed below.

Task	Selected combination		
	RNN	**Basic NN**	**SVR**
Dataset 1, *15min*	GRU; subset 1, all	*2*, expert defined	*3*, filter-reduced
Dataset 1, *30min*	GRU; subset 2, expert defined	*2*, expert defined	*3*, filter-reduced
Dataset 2, *15min*	GRU; subset 2, expert defined	*2*, expert defined	*2*, expert defined
Dataset 2, 30*min*	GRU; subset 2, expert defined	*2*, expert defined	*2*, expert defined

The selection reveals that *subset 2* provided clearly the best forecasts based on visual analysis. Another noteworthy factor is that based on the selection criteria, no LSTM-networks were chosen, which implies that GRU is the more advanced RNN-type for this learning task.

The warning evaluation is performed by first transforming the forecasts into binary classes. Each timestamp is labeled as either 1, representing *positive* test or “should warn”, or 0 representing *negative* test or “should not warn”. In order to apply the binary labeling, a threshold must be selected. The threshold refers to the error score value that if surpassed should denote a warning. After setting the thresholds, instances can be labeled by whether their values surpass the threshold or not. Finally, the predictions' binary labels are compared to the binarized real values to see if the warnings are triggered correctly. The thresholds are selected based on *company A*'s knowledge of how often critical errors occur, that being approximately twice a month. Based on this, the values are set as 7.5 for *dataset 1* and 8 for *dataset 2*, illustrated in fig. 25.

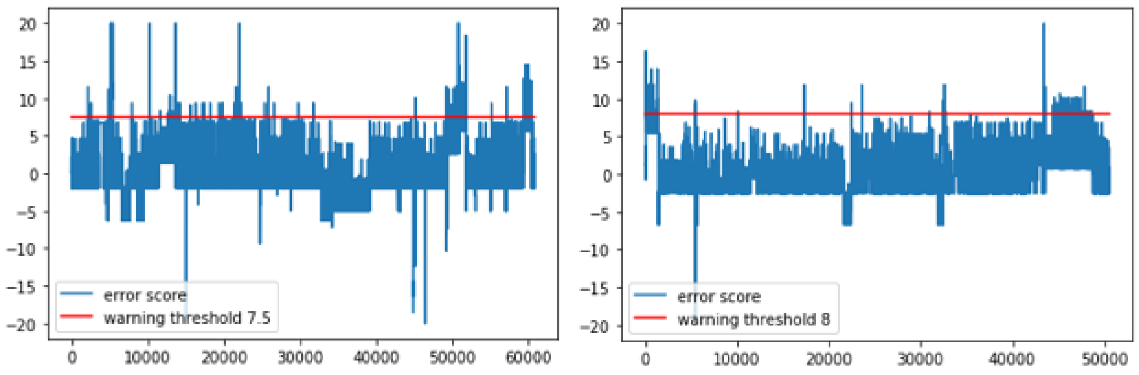


Figure 25 Configured warning thresholds. Dataset 1 (left), dataset 2 (right).

The final warning performance evaluation is measured by calculating the area under curve (AUC) scores. Originating in the receiver operating characteristic (ROC), the AUC refers to the size of the area below the ROC-curve, which is the plotting of false positivity rate (FPR) against true positivity rate (TPR). In this study, false positives denote instances where the forecasting methods would trigger false alarms, and true positives instances that they would trigger correct alarms. The ROC-curve is illustrated in fig. 26. When FPR plotted in x-axis is relatively small compared to TPR plotted in y-axis, the area is larger, which denotes good AUC-score. Score 1 denotes perfect warning accuracy, score 0.5 refers to warning completely by random, and 0 denotes reversed learning where the model always predicts the opposite of real value.

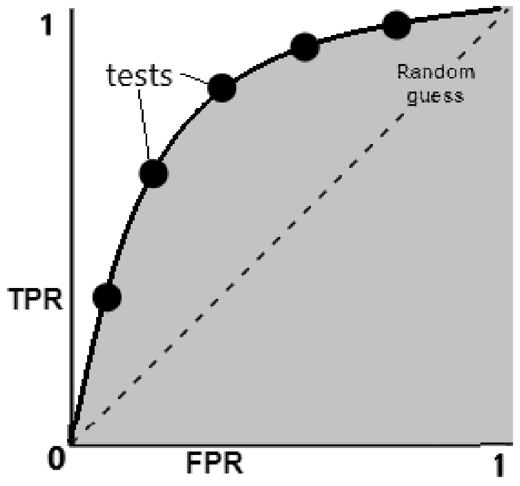


Figure 26 ROC curve and AUC-score

Results

Dataset 1

Model	FPR, *15min*	TPR, *15min*	**AUC, *15min***	FPR, *30min*	TPR, *30min*	**AUC, *30min***
GRU	0.0001	0.0014	0.5006	0.0127	0.0463	**0.5168**
NN	0.0050	0.0163	**0.5057**	0.0049	0.0143	0.5047
SVR	0.0008	0.0007	0.4999	0.0007	0.0014	0.5003

Dataset 2

Model	FPR, *15min*	TPR, *15min*	**AUC, *15min***	FPR, *30min*	TPR, *30min*	**AUC, *30min***
GRU	0.0066	0	0.4967	0.0076	0.0123	0.5024
NN	0.0007	0.0123	0.5058	0.0698	0.0864	0.5083
SVR	0.2091	0.2593	**0.5251**	0.2292	0.2963	**0.5335**

Despite the relatively good visual fits of some models, it is observed that the warning performances are poor. What's special, both FPR and TPR are very low for most models. This means that the models would trigger warnings very rarely. This is a factor that is not seen in the visual evaluation of the model performances, because the dataset is so large. An assumption is that the models cannot predict multiple simultaneous high error scores, where the high values can linger for some time, before dropping below the warning threshold.

The model that stands out from the final evaluation is the SVR of *dataset 2* (see fig. 27). It is the only model seemingly capable of simultaneous high error score predictions, and hence also achieves the best AUC-scores. However, these scores are still significantly too low for any reliable warning system to be established on it. Therefore, the implementation of this system is decided to be canceled, as for now.

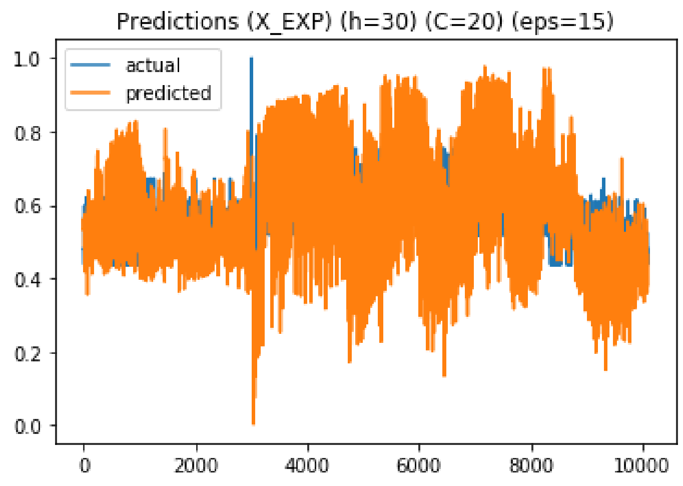


Figure 27 SVR forecast of dataset 2 (30min). The high FPR can be seen from the heavy fluctuation of the forecast-curve.

7 Discussion

The results in 6.3 provide no accurate predicting of the target variable nor triggering of warnings, and the experiments are considered unsuccessful. This supports the null hypothesis set in chapter 2, being that no effective methods are applicable on dataset of such scale. A foremost issue in assessing the experiments is the number of steps involved in the whole process. Starting from the separate data engineering project of [31], one has had to make compromises and knowledge-based decisions without validating every step, for instance, when selecting the 1-minute and 5-minutes framerates for datasets. However, not all steps of the process could have been validated at all, as for example framerates denser than 1 minute would have reduced instances by too little, considering the available resources. Because of the large number of steps and incapability to evaluate all of them, one cannot verify which steps in the process are actually successful and which of them are not.

Another issue that affects especially the preprocessing, is that one cannot explicitly evaluate the information value in a dataset. There are methods capable of explicit feature evaluation such as correlation coefficient to target variable. But, as shown, it is not straight applicable on time series data, as one should calculate the correlations between past features and future targets, instead of concurrent features and targets. Instead, the techniques and decisions are made by selecting candidates, applying the final modelling on them, and selecting best performing subsets by cross-validation. Many of the selections are also made without any validation or by not evaluating every option, since there would be greatly too many variables to assess. In addition to preprocessing parameters, there is also a heavy amount of model parameters that would need excessive testing to optimize for each dataset.

A third issue discovered is the complexity of big data for such modelling task. High volume, dimensionality, and complexity of the dataset causes computation issues, as for example the SVMs take several hours to converge on the training data. Some potential methods are completely discarded, such as statistical models since requiring strictly numerical data. Another example is the RVM that causes memory issues, at least with the available implementation and resources. This supports the assumption that neural networks are superior in big data modelling, since being the only technique capable of parallel computation by GPU-processing.

Besides these issues, there are some limitations set by the target dataset that expectedly hindered the analyzability on this specific study. Perhaps the biggest of these are missing values. They are hard to replace especially in time series, where they could denote either occasional missingness, or informatically valuable outages. It is noted that the imputation and outage labeling applied in 4.3 is likely to corrupt the data and hence impair the forecasting accuracy. Therefore, the results could be improved by applying a different missing value policy, such as completely imputing every missing value by some mean.

As a final thought, there is a certain limitation in applying regression forecast for warning logic. Since triggering warnings is essentially binary classification, one could achieve better predictions by formulating the learning task as classification of "should warn" or "should not warn". However, a similar process than with regression is still required for determining the positive and negative instances in an unlabeled dataset such as *client B*'s. Overall, the experiments indicate that the scope of such project is too wide to achieve practical and efficient results.

Summary of objectives set in chapter 2:

- Preprocessing techniques were successfully discovered for transforming multivariate big data into analyzable format. In addition to reviewed methods, custom methods such as visual feature elimination were proposed.
- A custom log-level-based target variable was crafted for unlabeled data. However, the efficiency of the variable is unclear as the final forecasts were inaccurate.
- Neural networks were discovered as best method for big data forecast, for their parallel computability. However, no advantage was found in their prediction performance, partially due to lack of testing different parameters and depths.

8 Conclusion

Part I of this thesis presents a comprehensive review on data preprocessing techniques, especially targeted on big data and time series analysis. One of the main issues is found out to be missing values and outages, common in real-time time series of certain industries. These are shown to hinder the usability of common preprocessing techniques such as missing value imputation, that work better with non-time-series data. Instead of replacing missingness by imputation, keeping them in data could provide valuable information of the nature of the time series, although they are challenging to model as variables of analysis. In addition to reviewed preprocessing techniques, some custom methods are also proposed, including a custom log-level-based target variable design, and a visual feature elimination process.

In part II, a review is made on multivariate time series analysis and forecast techniques, from statistical to ML. The theory is put to practice by experimenting forecast on a case dataset, that is first preprocessed in the first part. It is discovered that neural networks are practically the only suitable forecasting method for such datasets, due to size and dimensionality of big data, and neural networks' capability of training via parallel computation. SVMs are also successfully trained, although consuming inconveniently long training times. None of the forecasting methods perform significantly well, which could be due to unsuccessful preprocessing, too little method parameter optimization, or the mere difficulty of such learning task. This demonstrates a definitive issue of applying supervised ML on big data, which is the excessive number of factors and variables that must be validated, to ensure the success of the task. In the future, unsupervised ML methods, instead, could be experimented since they require less preprocessing.

References

[1] Apache Software Foundation. Hadoop home page. <*https://hadoop.apache.org*>

[2] Elastic NV. Elasticsearch home page. <*https://www.elastic.co*>

[3] Box, GEP., et al. Time series analysis: forecasting and control. *John Wiley & Sons, 2015.*

[4] Hochreiter S., Schmidhuber J. Long short-term memory. *Neural computation 9.8: 1735-1780, 1997.*

[5] Kyunghyun C., et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv:1406.1078, 2014.*

[6] Elastic NV. Kibana home page. <*https://www.elastic.co/kibana*>

[7] Grafana Labs. Grafana home page. <*https://grafana.com*>

[8] Ward, JS., Barker, A. Undefined by data: a survey of big data definitions. *arXiv:1309.5821, 2013.*

[9] Apache Software Foundation. Cassandra home page. <*https://cassandra.apache.org*>

[10] MongoDB Inc. MongoDB home page. <*https://www.mongodb.com*>

[11] Liu, ZH., Hammerschmidt, B., McMahon, D. JSON data management: supporting schema-less development in RDBMS. *Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 2014.*

[12] Splunk Inc. Splunk home page. <*https://www.splunk.com*>

[13] Apache Software Foundation. Sqoop home page. <*https://sqoop.apache.org*>

[14] Elastic NV. Logstash home page. <*https://www.elastic.co/logstash*>

[15] Apache Software Foundation. Mahout home page. <*https://mahout.apache.org*>

[16] García, S., et al. Big data preprocessing: methods and prospects. *Big Data Analytics 1.1: 9, 2016.*

[17] Kotsiantis, SB., Kanellopoulos, D., and Pintelas, PE. Data preprocessing for supervised leaning. *International Journal of Computer Science 1.2: 111-117, 2006.*

[18] Wilson, DR., Martinez, TR. Instance pruning techniques. *ICML. Vol. 97. No. 1997, 1997.*

[19] Bellman, RE. Adaptive control processes: a guided tour. *Princeton university press, 2015.*

[20] Saeys, Y., Iñaki I., Larrañaga, P. A review of feature selection techniques in bioinformatics. *Bioinformatics 23.19: 2507-2517, 2007.*

[21] Yu, L., Huan L. Feature selection for high-dimensional data: A fast correlation-based filter solution. *Proceedings of the 20th international conference on machine learning (ICML-03), 2003.*

[22] Tibshirani, R. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological) 58.1: 267-288, 1996.*

[23] Du, S., et al. Robust unsupervised feature selection via matrix factorization. *Neurocomputing 241: 115-127, 2017.*

[24] Cerda, P., Varoquaux G., and Kégl, B. Similarity encoding for learning with dirty categorical variables. *Machine Learning 107.8-10: 1477-1494, 2018.*

[25] Seger, C. An investigation of categorical variable encoding techniques in machine learning: binary versus one-hot and feature hashing. *2018.*

[26] Bengio, S. & Y. Taking on the curse of dimensionality in joint distributions using neural networks. *IEEE Transactions on Neural Networks 11.3: 550-557, 2000.*

[27] Honaker, J., King, G. What to do about missing values in time-series cross-section data. *American journal of political science 54.2: 561-581, 2010.*

[28] Che, Z., et al. Recurrent neural networks for multivariate time series with missing values. *Scientific reports 8.1: 1-12, 2018.*

[29] Fu, R., Zuo Z., Li L. Using LSTM and GRU neural network methods for traffic flow prediction. *31st Youth Academic Annual Conference of Chinese Association of Automation (YAC). IEEE, 2016.*

[30] Vijayakumar, N., Plale, B. Prediction of missing events in sensor data streams using kalman filters. *Proceedings of the 1st Int'l Workshop on Knowledge Discovery from Sensor Data, in conjunction with ACM 13th Int'l Conference on Knowledge Discovery and Data Mining, 2007.*

[31] Kylänpää, M. Technical report: Transforming big data time series for machine learning. *University of Turku Master's Project, 2020.*

[32] Cryer, JD., Chan, K-S. Time Series Analysis: With Applications in R. *Springer, 2008.*

[33] Brownlee, J. Multi-step Time Series Forecasting with Long Short-Term Memory Networks in Python. *<https://machinelearningmastery.com/multi-step-time-series-forecasting-long-short-term-memory-networks-python>*

[34] Flovik, V. How (not) to use Machine Learning for time series forecasting: Avoiding the pitfalls. *<https://towardsdatascience.com/how-not-to-use-machine-learning-for-time-series-forecasting-avoiding-the-pitfalls-19f9d7adf424>*

[35] Yule, GU. Why do we sometimes get nonsense-correlations between Time-Series? --a study in sampling and the nature of time-series. *Journal of the royal statistical society 89.1: 1-63, 1926.*

[36] Salvi, J. Significance of ACF and PACF Plots In Time Series Analysis. *<https://towardsdatascience.com/significance-of-acf-and-pacf-plots-in-time-series-analysis-2fa11a5d10a8>*

[37] Singh, A. Build High Performance Time Series Models using Auto ARIMA in Python and R. *<https://www.analyticsvidhya.com/blog/2018/08/auto-arima-time-series-modeling-python-r>*

[38] Portilla, JM. Using Python and Auto ARIMA to Forecast Seasonal Time Series. *<https://medium.com/@josemarcialportilla/using-python-and-auto-arima-to-forecast-seasonal-time-series-90877adff03c>*

[39] Tsay, RS. Multivariate time series analysis: with R and financial applications. *John Wiley & Sons, 2013.*

[40] Cao, L-J., Eng Hock, FT. Support vector machine with adaptive parameters in financial time series forecasting. *IEEE Transactions on neural networks 14.6: 1506-1518, 2003.*

[41] Siami-Namini, S., Siami Namin, A. Forecasting economics and financial time series: ARIMA vs. LSTM. *arXiv:1803.06386, 2018.*

[42] Yann L., Cortes C., Burges CJC. THE MNIST DATABASE of handwritten digits. *<http://yann.lecun.com/exdb/mnist>*

[43] Chakraborty, K., et al. Forecasting the behavior of multivariate time series using neural networks. *Neural networks 5.6: 961-970, 1992.*

[44] Kim, TY., Changha H., Jongkyu L. Korean economic condition indicator using a neural network trained on the 1997 crisis. *Journal of Data Science 2.4: 371-381, 2004.*

[45] Kim, TY., et al. Usefulness of artificial neural networks for early warning system of economic crisis. *Expert Systems with Applications 26.4: 583-590, 2004.*

[46] Cortes, C., Vapnik V. Support-vector networks. *Machine learning 20.3: 273-297, 1995.*

[47] Carbonneau, R., Laframboise, K., Vahidov, R. Application of machine learning techniques for supply chain demand forecasting. *European Journal of Operational Research 184.3: 1140-1154, 2008.*

[48] Formisano, E., De Martino, F., and Valente, G. Multivariate analysis of fMRI time series: classification and regression of brain responses using machine learning. *Magnetic resonance imaging 26.7: 921-934, 2008.*

[49] Tipping, ME. Sparse Bayesian learning and the relevance vector machine. *Journal of machine learning research 1.6: 211-244, 2001.*

[50] Bianchi, FM., et al. Recurrent neural networks for short-term load forecasting: an overview and comparative analysis. *Springer, 2017.*

[51] Rumelhart, DE., Hinton, GE., Williams, RJ. Learning representations by back-propagating errors. *Nature 323.6088: 533-536, 1986.*

[52] Graves, A., Schmidhuber, J. Offline handwriting recognition with multidimensional recurrent neural networks. *Advances in neural information processing systems, 2009.*

[53] Brownlee, J. Stacked Long Short-Term Memory Networks. *<https://machinelearningmastery.com/stacked-long-short-term-memory-networks>*

[54] Kostadinov, S. Understanding GRU Networks. *<https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be>*

[55] Goel, H., Melnyk, I., Banerjee, A. R2N2: residual recurrent neural networks for multivariate time series forecasting. *arXiv:1709.03159, 2017.*

[56] Google LLC, Google Brain Team. Tensorflow home page. *<https://www.tensorflow.org>*

[57] Kingma, DP., Ba, J. Adam: A method for stochastic optimization. *arXiv:1412.6980, 2014.*

[58] Hahnloser, RHR., et al. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature 405.6789: 947-951, 2000.*

All websites visited on 02.06.2020.