Event Detection and Predictive Maintenance using Component Echo State Networks

Jonatan Westholm

FMS820

Master's Thesis in Mathematical Statistics

Contents

1 Abstract

With a growing number of sensors collecting information about systems in industry and infrastructure, one wants to extract useful information from this data. The goal of this project is to investigate the applicability of Echo State Network techniques to time-varying classification of multivariate time series from primarily mechanical and electrical systems. Two relevant technical problems are predicting impending failure of systems (predictive maintenance), and classifying a common event related to the system (event detection). In this project, they are formulated as a supervised machine learning problem on a multivariate time series. For this problem, Echo State Networks (ESN) have proven effective. However, applying these algorithms to new data sets involves a lot of guesswork as to how the algorithm should be configured to model the data effectively. In this work, a modification of the Echo State Network (ESN) model is presented, that helps to remove some of this guesswork. The new algorithm uses specifically structured components in order to facilitate the generation of relevant features by the ESN. The algorithm is tested on two easy event detection data sets, and one hard predictive maintenance data set. The results are compared to Support Vector Machine and Multilayer Perceptron classifiers, as well as to a basic ESN, which is also implemented as a reference. The component ESN successfully generates promising features, and outperforms the minimum complexity ESN as well as the standard classifiers.

2 Introduction

More and more companies with equipment in industry and infrastructure choose to collect and store sensor data from their systems. These sensors typically measure physical quantities such as temperature, pressure, resistance, rotational speed, or more high level features such as specific errors occurring. Thanks to cheap digital technologies, a relatively small investment can be potentially very valuable. The crucial step of actually extracting usable knowledge from this data, however, is rarely a straight path. This is the subject of the growing field of data mining for industry. A lot of this sensor data comes in the form of time series, with new measurements being taken and stored regularly. Additionally, one often wants to take many sensors into account when studying the system, since interesting phenomena can potentially be found in investigating the dependence between different sensors. For example, normally insignificant values from one sensor might be very important conditional on another sensor's value. Also the order in which events happen can be important. Therefore, it is interesting to conceive good algorithms for modeling multivariate time series. There is a number of technical problems which could be solved using such models. Among them are optimizing process settings, detecting and possibly adjusting for common events which may be loosely connected to the sensors, and to predict rarely occurring failures of a system. The two latter are commonly called event detection and predictive maintenance, and in this project a new method for solving them is presented and evaluated on real data sets.

Both event detection and predictive maintenance can be formulated as a supervised machine learning problem on (multivariate) time series. The method is based on Echo State Networks (ESN) as they are put forward by [Rodan2011]. An ESN consists of a large number of nodes which are sparsely but recurrently connected. These nodes are referred to as the reservoir, and ESN belongs to the class of reservoir computing models, the other large model type being the Liquid State Machine [Maass2002]. The reservoir nodes are connected to output nodes, which are supposed to approximate a given target. Only these output connections are trained, which makes training mathematically and technically easier than other alternatives for training time-recurrent models. However, ESN has other difficulties. For example, in [Jaeger2005] Herbert Jaeger, one of the pioneers of ESN research, addresses the problem that it is hard to determine what class of processes which can effectively be modeled using a particular ESN.

The original approach in this project is to take a practical approach to this problem: the ESN should be synthesized so as to be able to capture the patterns which can be observed when looking at the data. This is accomplished with the use of components, which are sub-reservoirs with a specific structure. The design of the ESN includes deciding which components to include, and how they should be connected to the input, the output, and to each other. In addition to the component approach, some experimentation is done with dimensionality reduction of the generated features, and different techniques for learning the target from the reduced features are tried. The performance of component ESN is compared to a "basic" ESN, as it is formulated in [Rodan2011]. The ESN

classifiers are also compared to static time classifiers Support Vector Machine (SVM) and Multilayer Perceptron (MLP).

Two event detection data sets are tested. One is EEG signals from a person, where the label is whether the person's eyes are closed, see section 5.2. The other is data from sensors in an office room, measuring among other things temperature and CO_2. The label is whether there is a person in the room. For these two data sets, there are associated articles which registered performance figures. It is shown that both ESN models can match the results in these publications. For the predictive maintenance data set, open access data of so called SMART values from hard drives are used. The SMART values monitor mechanical running conditions and errors in a hard drive. The label is whether the drive breaks. In this case, no comparable study has been found. However, a study made at Google [Pinheiro2007] claims that "...it is unlikely that an accurate predictive failure model can be built using [SMART values] alone." (direct quote). In this report, it is shown that a component ESN can be used to make better than random predictions, when given samples which are not currently identified before failure by the company which supplies the data [Backblaze2017c]. The other algorithms, including basic ESN, do not perform significantly better than random guessing.

This report is structured as follows. In section 3, the research background of ESN is laid out, and the reason for framing predictive maintenance as a supervised learning problem on multivariate time series is explained. In section 4, the learning problem is formally stated, and the used test metric is defined. Also, the strategy for a proof of concept is laid out. In section 5, the data sets are presented. Their properties and the applied preprocessing is listed. Additionally, relevant qualitative questions of what the data sets can be used to show about the component ESN are posed. In section 6 and 7, the used algorithms are presented. In section 7, the component ESN is defined, as well as some component types, and feature reduction techniques. An overview of the implementation is presented in section 8. The results of testing are presented in section 9. In section 10, some theoretical properties of the modeling ability of ESN is derived, and the results are analyzed in the light of this. In section 13, some exciting ideas for further development of component ESN are presented.

3 Background

The reader of this master's thesis is assumed to be familiar with introductory mathematical statistics and stochastic processes, as well as to be acquainted with the most important machine learning methods.

3.1 Research Background

3.1.1 Markov Process

A very common modeling tool for stochastic processes is to make a Markov assumption. A stochastic process has a Markov property if:

$$f(Y_t = y_t | Y_{t_1} = y_1, Y_{t_2} = y_2, ...) = f(Y_t = y_t | Y_{t_1} = y_1) \tag{1}$$

if $t > t_i$ for $i = 1, 2, ...$ and $t_1 > t_i$ for $i = 2, 3, ...$

If we call t the *future* and t_1 the *present*, this means that the future of the process is conditionally independent of the past, given the present. So, all information that is useful for predicting the future of the process is contained in the most recent known state. The fact that we only have to account for a finite amount of information makes designing a prediction or classification algorithm much easier. Therefore, this is the basis of many time series algorithms. A naive but sometimes accurate assumption is that the process is itself a Markov process. A more powerful approach is to assume that there is a *hidden* process which has a Markov property. The process that we can observe is some function of the states of the hidden process. The models described in the sections below all work like this.

3.1.2 Linear State Space

Given a process U_t, which may be multivariate, with an associated label process Y_t, a linear state space model from U to Y is defined as:

$$X_t = f(AX_{t-1} + BU_t) \tag{2}$$
$$Y_t = CX_t \tag{3}$$

where X_t is called the state process. A is typically called a system matrix, B is called an input matrix, and C is called an output matrix.

3.1.3 Recurrent Neural Networks

With the same assumptions on the processes as in the previous section, the basic recurrent neural network (RNN) can be expressed in a very similar way to the linear state space models:

$$X_t = f(AX_{t-1} + BU_t) \tag{4}$$
$$Y_t = CX_t \tag{5}$$

where A, B, C are linear transforms, and $f\dot{()}$ is an element-wise nonlinear function.

RNNs are potentially very powerful at modelling stochastic processes, but they are also difficult to train from data. When training an RNN, one attempts to adjust the weights of A,B, and C, so as to approximate the function $U_t \to Y_t$

implied by the data. In 1994, it was shown that training an RNN using backpropagation is *difficult*, since the error derivatives will either decay or grow exponentially [Bengio1994]. Backpropagation is the standard method for training feed-forward neural networks, where the derivative of the error with respect to the network weights are calculated recursively backwards in the network. For this reason, the focus of the research has been to make certain restrictions on the structure of RNNs so as to make training easier and safer without sacrificing too much modeling capacity. One successful approach to this is the Long-Short Term Memory network introduced in [Schmidhuber1997]. The Echo State Network, which is the topic of this paper, is another one.

3.1.4 Echo State Networks

The Echo State Network (ESN) was described by [Jaeger2001a]. The ESN relies on the same state description as the full RNN, with A,B,C being general-case matrices. However, A and B are *not trained*, only C. Consequently, supervised learning uses no backpropagation in time, only a direct mapping from extended features to output. In the words of Ali Rodan and Peter Tiňo [Rodan2011]:

> Roughly speaking, ESN is a recurrent neural network with a non-trainable sparse recurrent part (reservoir) and a simple linear read-out.

The downside is that the model has less ability to adapt to the data, and modelling a complex function requires very many nodes [Jaeger2001a]. ESNs have been used to solve many different time series problems (see next section). In [Rodan2011], it is found that a simple yet randomized structure of the ESN is just as good as "trimmed" ESNs. In 2001, Jaeger proved that there is a theoretical limit to the storage capacity of the nodes of an ESN [Jaeger2001b].

3.1.5 Projects Related to the Chosen Method

Prediction is probably the most natural inference problem for time series. Given that we know (some of) the historical values of a process, what will the future values be? Prediction can be done as supervised learning: the past values is the data and the present values is the label. In [Busseti2012], deep neural networks (both recurrent and forward-only) are used to predict electricity demand. Echo State Networks are also frequently used for prediction. [Rodan2011] presents a comparative study of different ESN implementations for a number of prediction tasks. [Bianchi2015] and [Han2015] employ linear subspace-techniques to make ESN training and predictions safer and less noisy. In the area of Natural Language Processing, ESNs have been shown to outperform trigram models in predicting grammatically correct words to follow a given word sequence [Tong2007]. In [Lin2011], ESNs are used to predict future stock values, and are shown to outperform buy-and-hold strategies, especially in bear markets.

3.2 Technical Background

The data type in focus in this project is multivariate numeric time series. Thanks to the miniaturization revolution in the last decades, this is a data type which is becoming more and more common in industry and infrastructure. Obtaining, transferring and storing data from different types of sensors has become cheaper and easier. At the same time, the interest in extracting useful information from these time series has also increased [Keogh2004]. This master's thesis and the associated software project was created in house at Sentiantechnologies AB (also called Sentian.AI), in Malmö, Sweden [Sentian2017]. Sentian is an AI solutions company whose business goal is to make industries smarter, using advanced data mining and processing methods. The goal of this project is to investigate the applicability of Echo State Networks techniques to time-varying classification of multivariate time series from primarily mechanical and electrical systems. Specifically, the problems that were identified as interesting when scoping this project, was predictive maintenance and event detection.

3.3 Problem Background

Both predictive maintenance and event detection are essentially about detecting changes in the characteristics of a time series, and there exists many methods to do it that does not involve machine learning. Typically, to solve these problems without machine learning would require a model of the process being studied. The parameters of this model could then be recursively over segments of the data. To test whether a significant change has occurred at time T, one can estimate the parameters of the process for some time before T, and perform a test whether these are significantly different from the parameters estimated for the segment just after T. The model of the process can be very simple. For example, the model can be that the process is a constant plus noise. A common method in traditional time series analysis is to model the process as an ARMA process. There are known methods for efficient and effective estimation for ARMA-coefficients, as well as uncertainty estimates [Jakobsson2015], which makes it very suitable for this approach.

3.3.1 Predictive Maintenance

Predictive maintenance of a system means identifying and remedying risky behaviour, before the system is irrecoverably broken or requires expensive *reactive maintenance*. A machine learning approach to this can be to produce an algorithm that organizes or visualizes the sensor data so as to make it more understandable to a human operator. This would typically be formulated as an unsupervised problem. A more strictly automated approach is to make a system that learns to identify risky behaviour itself. This would be formulated as a supervised problem. This project uses the latter approach. One good reason for this is that the supervised learning problem is more easily evaluated objectively, independent of a human operator. A successful system can also reduce

the cost of monitoring substantially more than one which requires an operator. An argument against the supervised approach is that the resulting algorithm is not very transparent. This may lead to missing out on valuable insight into the process that might have been acquired with human supervision. Such insight could even be more valuable than the predictive maintenance system itself.

Given that supervised learning should be used for the predictive maintenance system, it is often advantageous to formulate the problem as a *time series* problem. That is, the chronological order of the time-samples should be respected so that past values of the process and not just the current ones, can influence the algorithm's prediction. Even if the modeled system is often restarted with the same settings so that its future should be independent of its history, an error is by definition the system not working as intended, so the "fresh restart" property may very well be violated in the presence of an error. So what one wants to do is to learn a model that differentiates "normal" from "abnormal" system behaviour.

The best data for this problem, is to have long time series covering a lot of fault-free running, as well as many error incidents with context. This type of data is notoriously hard to come by [Saxena2008]. This is because it is typically collected by private companies, and the data is potentially very valuable and could also reveal dangerous shortcomings of the system. An open collection of predictive maintenance data made by NASA can be found at [PCoE2017].

4 Problem Formulation and Methodology

Below is the data representation and terminology used in the project.

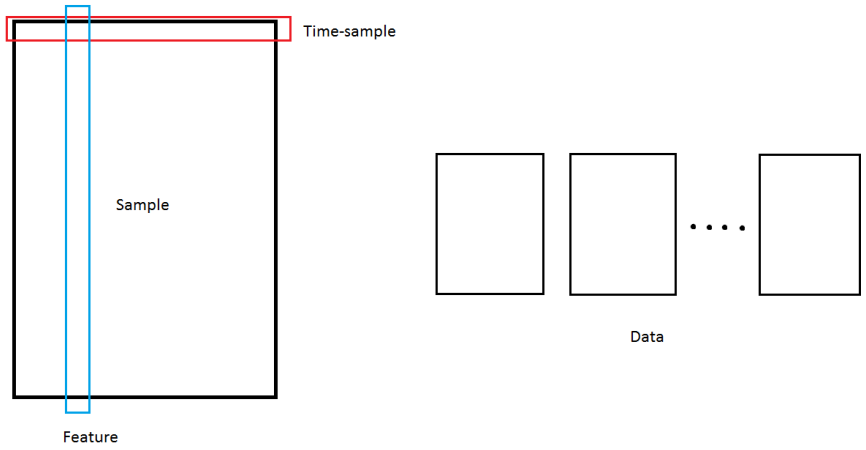

Figure 1: Data representation used in the project.

4.1 Formal Problem Statement

Call the given data $X_1, X_2, ..., X_n$ with samples X_i. We also have label arrays $Y_1, Y_2, ..., Y_N$ where Y_i has one label for each time sample in X_i. The problem is this. Given:

$$X_i(\tau), 0 \leq \tau \leq t \tag{6}$$

Predict:

$$Y_i(t) \in \{0, 1\}. \tag{7}$$

4.2 Test Metric

For classification and cegression tasks, the *precision* and *recall* measures are easily interpretable, and give a good idea of how useful the predictions are.

- **Recall** is the probability that the prediction is positive, given that the label is positive.
- **Precision** is the probability that the label is positive, given that the prediction is positive.

It is easy to make a predictor with perfect recall: simply predict always positive. This will give a bad precision, however. A popular way of obtaining a combined performance measure from precision and recall is to take their harmonic mean F:

$$\frac{1}{F} = \frac{1}{2}(\frac{1}{P} + \frac{1}{R}). \tag{8}$$

This is also known as the F-measure. The above makes it possible to compare the performance of the model between different data sets.

4.2.1 Accuracy Test Metric

In [Candanedo2016], the test metric used is called *accuracy* and is calculated as:

$$acc = \frac{A + D}{A + B + C + D} \tag{9}$$

where A is true positives, B is false positives, C is false negatives, and D is true negatives. In short, the accuracy is the ratio of time instances when the predictor is correct. This is a reasonable measure for data sets with roughly an equal number of positive and negative examples, and when making a false positive is as bad as a false negative. This is not the case for a predictive maintenance system however, since it has many more negative examples, and false negatives are worse than false positives.

4.2.2 Lost Days Estimation

For an imperfect predictive maintenance system, one has to make a trade-off between missing some failures, and reporting some failures too early. Where this trade-off should be made is very contextual, and depends on economic factors which are outside the scope of this paper. However, we can estimate the expected number of lost days due to premature warnings. When running on known data, simply take the average of the number of days that the algorithm warns before the unit actually did break down.

4.3 Performance Measuring Procedure

All the tested algorithms use user-defined hyper-parameters. To make a fair comparison, one should use find suitable hyper-parameters for each new data set. The data set was split into three parts: *Training*, *Validation*, and *Test*. To find good hyper-parameters, the model was iteratively trained on the *Training* part and its performance tested on *Validation*. The parameters that gave the best result on *Validation* (with some checking that the result was robust for similar settings) was used to train the model on both *Training* and *Validation* and tested on *Test*. The performance on *Test* was reported as the final score.

4.4 Proof of Concept Strategy

The strategy for evaluating whether component ESN is a valuable modification of the ESN method, is to measure performance on different real data sets. The standard minimum complexity ESN described in [Rodan2011] is referred to as *basic ESN*. The two ESN methods are also compared to two standard classification algorithms: the SVM and the MLP. If the component ESN can be shown to:

- Solve the formal problem significantly better than the basic ESN, SVM, and MLP,
- Solve a reasonable practical problem significantly better than the reference algorithms,
- Be shown to generate apparently valuable features in the reservoir, which are not generated by the basic ESN

Then, it will be considered that the component ESN has potential as a relevant improving modification of the basic ESN, since it outperforms the basic ESN as well as standard classifiers.

5 Data sets

5.1 General Properties

The data type of interest in this thesis is multivariate time series. These can be represented as matrices, with each time instance occupying one row. The elements of the matrices are numeric. Furthermore, the data comes from many different units, so in fact what is being worked on is sets of multivariate time series. It is assumed that all units have the same internal dynamics.

5.2 Eye Data

Source: [Eye2013]. This data consists of 14 time series of Electroencephalogram (EEG) measurements from a person sitting in a quiet room [Suendermann2013]. The time series is annotated with a 0 is the person's eye is open, and a 1 if the person's eye is closed. Note that this is measurements from a single person, at a single occasion. The time series cover 117 seconds and 14980 time-samples, during which the person blinks 11 times. The data has some properties which makes it interesting to try to model with a Component ESN:

- It contains a number of features which all have a *similar* appearance. Compare Fig 3 with, for example, the plots in Section 15.1. Therefore, a homogeneous reservoir should be efficient.
- Looking at Fig 3, it seems as though every start and end of a positive section (the person having eyes closed), is marked with a spike in some of the features. So, it is interesting to see whether the ESN will adapt to *persistent* generated features.
- The signal has a high frequency variability on small scales. Will the ESNs feature generation and feature reduction have a *smoothing* effect?
- There are many performance figures in the associated article [Suendermann2013], with which the performance of a classifier can be compared.
- The authors of the associated article [Suendermann2013] mention in the section "future work" whether one could obtain the same results with a fewer number of sensors. This will also be investigated in the Results section.

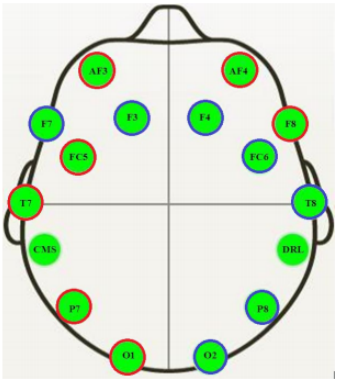

Figure 2: The placement of EEG sensors in the Eye data set.

5.2.1 Preprocessing

See Fig 3 for results of the preprocessing.

- The data set contains a few outliers in each time series. According to the article [Suendermann2013], these were added for robustness testing at a late stage in the project. Since these would ruin the normalization, they were removed and replaced by the feature mean.
- The EEG signals were normalized to mean value 0 and standard deviation 1. This means that the previously replaced outliers end up at a value of 0.

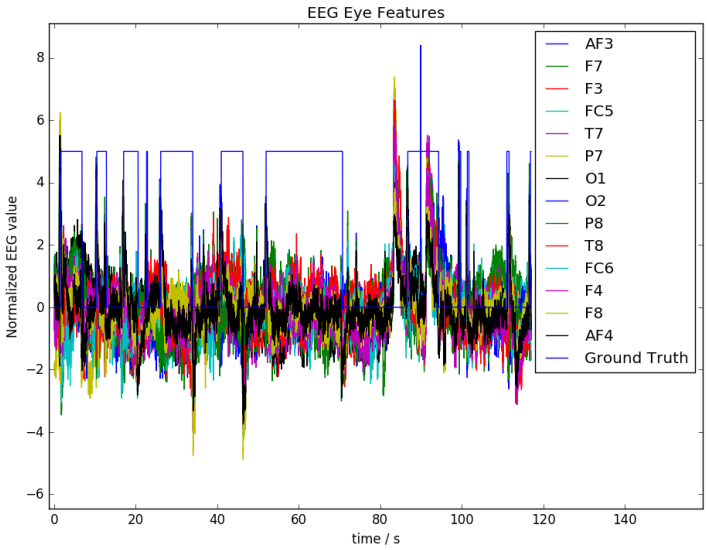

Figure 3: The preprocessed features of the Eye data set. Note that the "Ground Truth" has been scaled for visibility. It is actually a binary variable.

5.2.2 Related Research

This data set was published in conjunction with an article [Suendermann2013]. The authors' use 48 different non-time recurrent machine learning methods for classifying the eye as being open or closed. The Weka toolkit is used. Naive Bayes inference and the Multilayer Perceptron are found to perform badly. Random Forest and KStar are found to perform well. Since the data is so small, the results are evaluated on the training set. See Section 9.1 for detailed results.

5.3 Occupancy Data

Source: [Occupancy2016]. This data is from an article that presents a method to detect whether a room is occupied by a person by measuring light, CO_2, humidity, and temperature [Candanedo2016]. The thought end goal was to save energy by regulating heating to only times when rooms are being used. The measurements were made every minute during 16 days. The equipment was set up in an office, so people typically were there during the day and not on weekends, leaving two days without positive label. See Fig 4. The data set is labeled with a ground truth. The Occupancy data set has a couple of properties whose effects are interesting to study on the Component ESN:

- The features have a large scale varying mean. Particularly Temperature, Humidity, and HumidityRatio are at different levels for different days.

However, the change profile during the workdays seems to be the same. Will some reduced features be activated by these patterns, independent of the day average?

- There are performance figures in the associated article [Candanedo2016], which can be compared to the performance of the implemented classifiers. Will it be useful to exploit the time-aspect of the data, or will it just introduce unnecessary complexity and over-fitting?

5.3.1 Preprocessing

The data was normalized feature-wise. This was done by taking the mean and standard deviation over the training data, and then removing these from all available data.

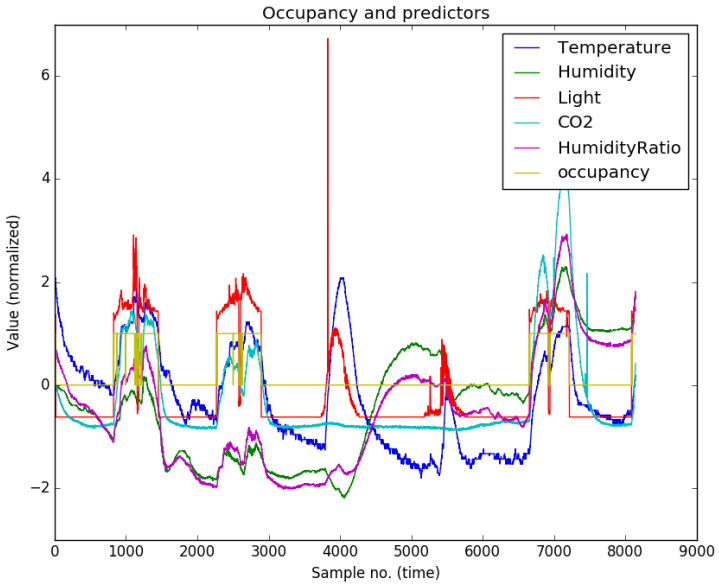

Figure 4: The training data for the Occupancy data. The data has been normalized for the purpose of modelling it. Note that the label ('occupancy') was not normalized. This signal covers 5 days. The blocks where the label is positive are working days.

5.3.2 Related Research

This data set comes with an article [Candanedo2016]. The authors pose the problem as a supervised learning task, and solve it using non-time recurrent machine learning methods, including Random Forest, Gradient Boosting Machines, Linear Discriminant Analysis, and Classification and Regression Trees.

Different combinations of features are investigated to see which can be omitted. It was found that using only the features *Light* and *Humidity Ratio*, almost the same accuracy (see Section 4.2.1 for authors' definition of accuracy) could be achieved as with all features. See section 9.2 for details and exact figures.

5.4 Hard Disk Drive Data

This data comes from a company in the backup storage business called Backblaze [Backblaze2017a], and is available to the public for free. As a step in their maintenance work, they monitor a number of so called *SMART* stats that are reported by the different hard drives that they use. SMART stands for Self-Monitoring And Reporting Technology. Its purpose is to warn the owner of a hard drive about errors which may render the drive unusable.

The data has a number of properties

- Backblaze has made the job of normalizing all values that reported so that a value of 100 is "normal", and higher is "better".
- The drives come from different producers. The SMART stats are not strictly standardized, so different stats are reported from different models.
- Different producers can have different ways of measuring the stats. Therefore, one prediction model has to be made per product model.
- The data is run-to-failure. The failure is reported explicitly, so it is possible to determine between units that broke, and those that were for some taken out of operation.
- The failure is reported either when a drive is not contactable or syncable, i.e, when it cannot be used anymore. A failure is also reported when one of five SMART values exceed a predefined threshold. These are typically serious errors, after which the hard drive is not considered reliable. See 5.4.3 - Related Research for more details.

5.4.1 Preprocessing

In addition to the basic normalization mentioned above, some extra preprocessing was done.

- Many of the features never change. The data was scanned for unchanging features, and these were removed.
- The data starts in the middle of operation, so some units naturally break very shortly after the logging begins. Units that broke less than 100 days after starting logging were disregarded.
- A very small share of the units had missing values. Those units were disregarded.

- The data was normalized feature-wise. All features were scaled so that their standard deviation on the training set was 1. No mean was removed. The choice not to remove means is motivated by a property of the data: for most features a value of 0 means 'normal', while a larger value means 'abnormal'. By not removing the mean, algorithms do not have to compensate for the normal state.
- The company has active scanning of five SMART values. According to the website [Backblaze2017b], when two of these reach a non-normal value then the drive is taken out of order. See Section 5.4.3 (Related Research) below for further details. One single deviation in these features is also indicative of a drive not functioning normally. At this point, the drive is inspected manually. Since these errors are apparently caught already, a model that can detect failure in the *other* units would be of most value. Therefore, units that have a deviation of any of these SMART features on their final day of functioning, were removed from the training and testing.

Below is a list of the features that were used. Note that this varies between product models, since manufacturers have a choice in which SMART values to report. There is also some freedom in how to report the values. For this reason, a single product model was chosen to model. The product model is *Seagate Desktop ST4000DM000*. Almost half of Backblaze's drives are of this type (36281 out of 86059 drives). Manual inspection was initially done on several product types, to check that this particular model was not an exceptional case in terms of what the SMART values seem to represent. Below a list of the remaining SMART values that were used (15 features). As noted above, values which never change in the data are removed, also from this list.

- 1: Raw Read Error Rate
- 3: Spin Up Time
- 5: Reallocated Sector Count
- 7: Seek Error Rate
- 9: Power On Hours
- 12: Power Cycle Count
- 183: Runtime Bad Block
- 184: End-to-End Error
- 187: Reported Uncorrect Error
- 188: Command Timeout
- 189: High Fly Writes
- 190: Airflow Temperature Celsius

- 193: Load Cycle Count
- 197: Current Pending Sector
- 198: Uncorrectable Sector Count

See the Appendix section 15.1 for some graphs over what the features look like for drives of the *Seagate Desktop ST400DM000* type. For these plots, about 10% of the data from the drives were used (3009 out of 36281 drives).

5.4.2 Target Generation

The hard disk drive data contains a feature that says whether the unit broke down that day, such that it had to be discarded. In this project, focus was on predicting impending failure, so the target was set to 1 for 20 days before failure, and 0 otherwise. An example: suppose that the data log begins on 2016-01-01 and that the unit breaks down on 2016-07-01, 182 days later (counting the leap day). Then, the target is 0 for the first 162 days, and 1 for the remaining 20 days. For units that did not break down during the logging time, 20 days were removed from the end. That way, one can be certain that failure is more than 20 days away for these units. This target should not be seen as a ground truth. Many errors make the hard drive break down faster than 20 days. Other errors can occur in the unit and make it break down hundreds of days later. A linear target was initially used, to set it as the number of days until failure. For time to failure larger than 20 days, the target was 20. However, this proved to be too difficult to get the models to learn. The binary target used is a compromise between a target that is reasonable to learn, evaluate, and also provides valuable information if it can be learned accurately.

5.4.3 Related Research

Since the introduction of SMART, several attempts have been made to use the system to predict imminent failure in drives. In [Hughes2002] a linear, instantaneous method is used. A large study at Google [Pinheiro2007] reached three key findings:

- A few SMART values have a large impact on impending failure.
- A large share of drives (36%) show no indication on these SMART values before failing.
- Operating temperature and disk activity does not have a large impact on drive lifetime.

According to the company's website [Backblaze2017b] (spring 2017), Backblaze uses five SMART values to predict when a drive is about to fail. These are:

- SMART 5: Reallocated Sectors Count
- SMART 187: Reported Uncorrectable Errors

- SMART 188: Command Timeout
- SMART 197: Current Pending Sector Count
- SMART 198: Uncorrectable Sector Count

The procedure at the site is to do manual investigation when one of these errors are reported. On the company website [Backblaze2017b], there is collected statistics about how helpful these values are on their own, and in combination.

Looking at plots of the features in Section 15.1, there is a number of questions that can be asked about what can be inferred from the data:

- Does any of the features except the five critical ones mentioned above, have any single-handed large impact on impending failure?
- Just as with the Occupancy data set, the features exhibit a great variation in their appearance. Will this be reflected in the extracted features of the ESN?
- Is the imposed learning problem even sensible? If it can be solved perfectly then it is of course good enough, but analyzing the usability of a less-than-perfect system requires defining some rule for when to advise the user to replace or repair the unit, given the output of the classifier. The effects of different warning rules are shown in the Results section.

6 Reference Algorithms

To get an idea of the performance of the algorithm that was developed, it was compared to other machine learning algorithms. Two of these - SVM and MLP - are very well tested and proven effective to a wide range of learning tasks. Most researchers and developers who have worked with a machine learning project are familiar with the capabilities of SVM and MLP, so performance relative to these algorithms should give an idea of how good the proposed algorithm is. These algorithms are however, in their basic formulation, not time-recursive. In this project, they have therefore been learning *static classification*. That is, without hidden states as defined in 3.1.1. Since SVM and MLP as they have been used in this project do not take into account the time aspect, the rules that they can learn are much less complex. These two are then used as a reference against the two ESN-algorithms. When attempting to prove that a modification of the ESN is an improvement for some data sets, one wants to at least make sure that ESN is better than standard static classification algorithms for those data sets. The main reference algorithm is Basic ESN, which is based on the minimum complexity ESN described in [Rodan2011].

6.1 Support Vector Machine

The Support Vector Machine (SVM) is widely used for binary classification tasks. It can easily be modified to prioritize few false positives or few false

negatives, by setting weights to samples. In this project, Scikitlearn's implementation was used [Scikitlearn2017a]. The default settings were used, except weighting of samples. For the Backblaze data set [Backblaze2017a], since the events that were to be detected were typically rare, positive examples got a higher weight. This weight was set to give the best result for the training set.

6.2 Multi Layer Perceptron

Multi Layer Perceptrons, also called Feed-forward Neural Networks, is widely used for classification and regression tasks. In this project, Scikitlearn's implementation was used [Scikitlearn2017b].

6.3 Basic ESN

This algorithm, as implemented in this project, is largely based on the minimum complexity ESN described in [Rodan2011]. One important difference from the algorithm described in the article is that it deals with univariate time series. See section 7.3.3 for how the basic ESN is modified to take multivariate time series as input. Furthermore, the algorithm implemented here has an additional feature reduction step, as shown in Fig 5. In the original minimum complexity ESN, the target is learned directly with a linear transformation of the node activations. This was not done here for two reasons. Firstly, [Rodan2011] is concerned with a numeric target, while this project is concerned with a class target. It was discovered that using a special classification algorithm such as SVM and MLP was superior to a linear method for this task. But these learners are only effective for up to about 20 input features with the data used. So using all 100s of node activations directly as features will not work. Secondly, since this project deals with multivariate time series, the reservoir needs to be bigger in proportion to how many input features there are. This further increases the need for feature reduction. See more in section 7.4.

7 Component ESN

7.1 Overview

The Component ESN is a state space model, as described in (4). The transfer function from input U to output Y is:

$$X(t) = f(AX(t-1) + BU(t)) \tag{10}$$
$$X_r(t) = R(X(t)) \tag{11}$$
$$Y(t) = C(X_r(t)) \tag{12}$$

Here, A and B are matrices, and A is sparse. Furthermore, $f(\dot{})$ is a nonlinear, elementwise function. The feature generation module takes in a multivariate

time series and outputs, for each time point, an aggregate of the time serie's history. The synthesis and properties of the including elements A, B, f are described in section 7.3. The feature reduction module is a static function of the generated features, to a vector space with a lower number of dimensions. Two methods for feature reductions are given in section 7.4. Finally, the classification module is also a static function that takes in reduced features and return a classification of the current state of the input process. Three methods for the classification are given in section 7.5.

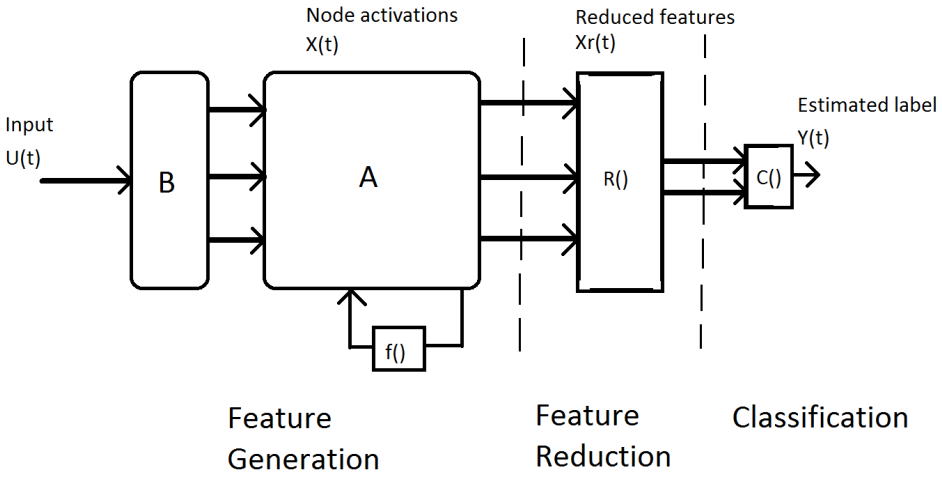

Figure 5: Flowchart of ESN. Blocks A and B are matrices, and f is an elementwise function. With some of the methods used in this project, functions R and C are linear mappings, but not all.

7.2 Training

The training data consists of a set of row matrices, where each row is a time-sample of the process being modelled. Each such matrix is called a sample. For each sample, the node activations are initialized to $X(0) = 0$. Some initial time-samples are fed into the system, as a burn in. In this project, 10 burn in time-samples were used. After the burn in, the node activations are saved in a new row matrix. This will be the input to training the feature reduction. For each new sample, the nodes are reset. The feature reduction $R()$ is trained on node activations from the whole training data. The dimensionality-reduced data, $X_r(t)$, is used to train the classifier $C()$.

7.2.1 Online training

The classifiers used in this project can be updated with new labeled samples after training. This has not been done, however. For the feature reduction techniques used in this project, they do not lend themselves to online training. The reservoir itself is not trained at all, as is the central idea of ESN.

7.3 Feature Generation

The original approach in this project is to construct an Echo State Network (ESN) model using components. This is meant to make it easier to configure the model to what is known about the data set, compared to a standard ESN. In the description below, M denotes the number of inputs to the model in each time step, or in other words the number of features of the data.

7.3.1 Direct

The DIRECT component simply takes the latest process values and stores them in nodes. This is redundant if one already has a Linear Delay Line component in the ESN. Below the update equation for Direct component. Could have been written $x(t) = u(t)$ for short. It is included here as a demonstration of the format.

$$\begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_M(t) \end{bmatrix} = \begin{bmatrix} 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1(t-1) \\ x_2(t-1) \\ \vdots \\ x_M(t-1) \end{bmatrix} + \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \cdot \begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_M(t) \end{bmatrix}$$

Figure 6: The general form of the Direct component with M inputs.

7.3.2 Linear Delay Line

The Linear Delay Line component consists of $M * (p + 1)$ nodes, where p is a user defined parameter. M head nodes take in the inputs, which are then shifted down p steps. With a Linear Delay Line, the Component ESN can represent a p-th order VAR model. Below the update equation for the Linear Delay Line. For compactness, it is here assumed that $M = 2$ and $p = 1$.

$$\begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1(t-1) \\ x_2(t-1) \\ x_3(t-1) \\ x_4(t-1) \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix}$$

Figure 7: An example of the Linear Delay Line component with 2 inputs and 1 delay. Here, x_1 and x_3 are called head nodes, and take in external input. This is then sent in successive iterations to subsequent nodes, x_2 and x_4 in this case. Notably, x_2 does not send any activation to x_3, so the delay lines are separate.

7.3.3 Rodan

This style of nodes was proposed in [Rodan2011]. Specifically, the one that was used corresponds to the Delay Line Reservoir (DLR) described in the article. For this project, the structure was changed to accomodate for multivariate time series. The structure used is in effect M separare DLR reservoirs, one for each feature. This structure does not generate any combinations of features, but this can be arranged by using Mixing: see section 7.3.6.

Below, $0 < r, b < 1$ are two parameters that determine how much activation is transferred between two subsequent nodes. In [Rodan2011], $r = 0.5$ and $b = 0$ are used for the Delay Line Reservoir. The same values were used for this project. Furthermore, $0 < v$ is a parameter that determines how much input values effect the activation of nodes. Note that scale is not unimportant, since the activation function is nonlinear. The value used in this project is $v = 0.5$. In the below example, $M = 2, N = 4$ is used for compactness. The sign on the input weights are randomly generated (with same probability for positive as for negative). The activation function f below is the same as used in [Rodan2011].

$$\begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \end{bmatrix} = f(\begin{bmatrix} 0 & b & 0 & 0 \\ r & 0 & 0 & 0 \\ 0 & 0 & 0 & b \\ 0 & 0 & r & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1(t-1) \\ x_2(t-1) \\ x_3(t-1) \\ x_4(t-1) \end{bmatrix} + \begin{bmatrix} v & 0 \\ -v & 0 \\ 0 & -v \\ 0 & v \end{bmatrix} \cdot \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix})$$

Figure 8: An example of a Rodan component with 2 inputs and 2 nodes per time series. This is a multivariate adaption of the reservoir used in [Rodan2011]. The input values all have the same absolute value, but with a random sign. In this project, the parameters used were $r = 0.5$, $b = 0$, $v = 0.5$.

$$f_i(x_i) = tanh(x_i) = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}}, i = 1, 2, ..., N \tag{13}$$

7.3.4 Thres

The Thres component is inspired by the transistor. It consists of two nodes. The first is activated when its input signal surpasses a given threshold. The second lets its input signal go through only when the first node is activated. This is supposed to model conditional importance between variables. Below is an example of a single Thres component. Here, $TL \sim N(0,1)$ denotes the signalling threshold. The parameter H is chosen as a very large number, so that the x_2 cannot be activated without the x_1 being activated. Furthermore $TH >> H$, so that x_2 can have a negative activation. The input weights v_1, v_2 are sampled from $N(0,1)$. The random input weighting is so that one can generate many Thres components with different properties, with input from the same two features.

$$\begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = f(\begin{bmatrix} 0 & 0 \\ TH & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1(t-1) \\ x_2(t-1) \end{bmatrix} + \begin{bmatrix} v_1 & 0 \\ 0 & v_2 \end{bmatrix} \cdot \begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix})$$

Figure 9: Transfer function of a single Thres component. The inspiration is the transistor. Here, x_2 is only nonzero when $x_1 > TL$.

$$f_1(x_1) = \begin{cases} 0 & x_1 \leq TL \\ 1 & x_1 > TL \end{cases} \tag{14}$$

$$f_2(x_2) = \begin{cases} 0 & x_2 \leq H \\ x_2 - TH & x_2 > H \end{cases} \tag{15}$$

7.3.5 Exponential Delay Line

The goal of the Exponential Delay Line is to store information about the average value of the input feature, at time scales which can grow exponentially with the amount of used memory. The principle is best illustrated with a figure, see Fig 10.

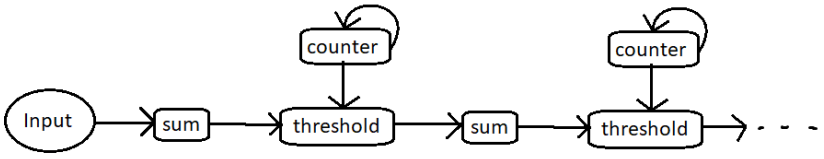

Figure 10: Exponential Delay Line structure. Input is led into a "sum" node, which has just a linear response, and is self-preserving. When the "counter" node is reaches 0, the "sum" node activates the "threshold" node, which activates the next "sum" node, and empties the first "sum" node. The weight from the "threshold" to the next "sum" is $\frac{1}{2}$. The counters have cyclic activation, reaching 0 every T iterations. The trick is that

$$T_{i+1} = 2 * T_i$$

so each "sum" node gets incremented exactly twice in each cycle. In the second half of its cycle, the "sum" will contain the average of the 2^i last samples before its last activation, if the "sum" node is i sections from the input.

Below is the state space model. For compactness, it is assumed that there is just one input feature, and that the *order* of the exponential delay line is 3. In figure 12 is shown the impulse response of an Exponential Delay Line.

$$\begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \\ x_4(t) \\ x_5(t) \\ x_6(t) \\ x_7(t) \\ x_8(t) \\ x_9(t) \end{bmatrix} = f(\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0.5 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0.5 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & -2H & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & -2H & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -2H \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_1(t-1) \\ x_2(t-1) \\ x_3(t-1) \\ x_4(t-1) \\ x_5(t-1) \\ x_6(t-1) \\ x_7(t-1) \\ x_8(t-1) \\ x_9(t-1) \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \cdot u_1(t))$$

Figure 11: A single Exponential Delay Line. The order is 3, meaning that there are three of the sections sketched in figure 10, and the max memory is 2^3 time steps. Node x_1, x_2, x_3 will contain the average of the $2^1, 2^2, 2^3$ latest input values, respectively. H is a large constant, so that the sections will not leak.

$$f_i(x_i) = x_i, i = 1, 2, 3 \tag{16}$$

$$f_i(x) = \begin{cases} 0 & x_i \leq -H \\ x_i & x_i > -H \end{cases}, i = 4, 5, 6 \tag{17}$$

$$f_7(x_7) = x_7 - 1 \mod 2 \tag{18}$$

$$f_8(x_8) = x_8 - 1 \mod 4 \tag{19}$$

$$f_9(x_9) = x_9 - 1 \mod 8 \tag{20}$$

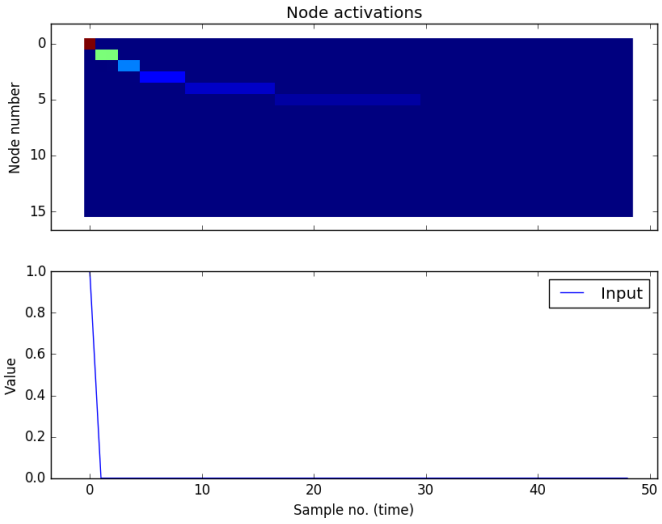

Figure 12: Response of a single order 5 Exponential Delay Line, when given an impulse signal. On each column, node activations for a certain time sample. On each row, activations over time for a certain node. The nodes with visible activation are the "sum" nodes. The other nodes have been muted by rescaling parameters in Eq. (7.3.5). When using an Exponential Delay Line in an ESN, also the sum nodes were scaled down, so that the response would be smoothed in for example a Rodan component.

7.3.6 Mixing Components

To model dependencies between variables, values from outputs of components are led into inputs of other components. These links are specified according to a scheme ["Source Component", "Target Component", k,replace,w], where k is an integer determining the number of links to make, and w is a positive real value determining the weight of the links. With this, k nodes of Source type are selected at random and led into k nodes of Target type (1-to-1). The user

can determine whether the selection will be with replacement or not, by editing a configuration file. The weight of all links is the same, and can be determined by the user. The default value is $w = 0.5$. The sign of the weight is randomly generated.

It is a bit crude to treat all components of the same type as the same. A future improvement to this specification of the mixing could be to let the different components have ID's, and the mixing specifies a source and target ID instead of a source and target component type.

7.4 Feature Reduction

The networks that were created in this project consist of 200-3000 nodes. This is purposefully many; since the weights in an ESN reservoir are not trained, most nodes will not contribute to predicting the class. With many 100s of nodes, there is a clear risk of over-fitting. Furthermore, many of the nodes may be noisy, or be very similar to each other. Therefore, it is necessary to extract some few features from these nodes. After feature reduction, the number of features is $10 - 40$. The output dimensionality of the feature reduction is a trade-off between not losing any important information (using many features), and avoiding over-fitting in the final classifier (using few features). The loss of information can be estimated with average projection errors. A reasonable amount of inputs for the final classifier can be estimated from the amount of data that is available. More data and more varied data would mean that more features can be used without over-fitting. However, one needs to do some tests to see which settings actually give a good result.

7.4.1 Principal Components

This feature reduction is done by taking a Singular Value Decomposition of the node activations. The principal components corresponding to the k largest singular values are saved as a linear transformation. Here, k is a user defined parameter. In this project, k values of $10-40$ were used. Lower limits for k were found by looking at the associated singular values of the principal components.

From some testing, it was found that a lower k is appropriate for SVM and MLP final classifiers, while a linear classifier could use a larger number of inputs without over-fitting. The principal components method for feature reduction is used by [Rodan2011]. See figures 13 and 14 for examples of what data looks like when projected onto principal components.

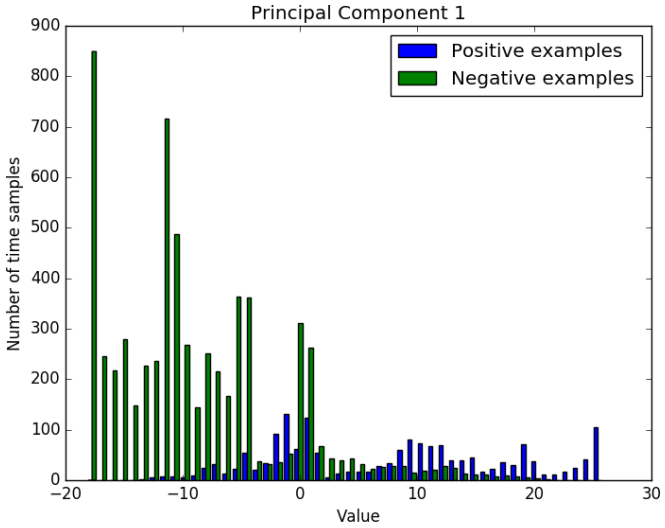

Figure 13: This histogram was constructed by collecting the activations generated in the ESN from the Occupancy data set, and then projecting them on their first principal component. As we can see, the positive examples and the negative examples are already quite well separated with only this feature.

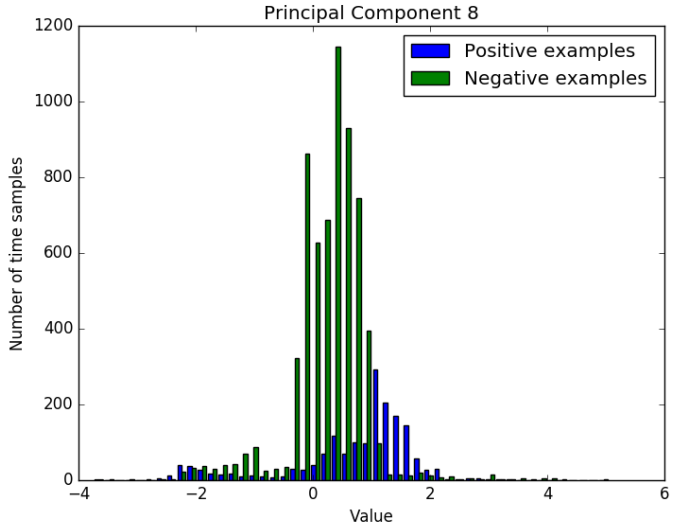

Figure 14: This histogram was constructed by collecting the activations generated in the ESN from the Occupancy data set, and then projecting them on their eighth principal component. Note the change in scale from Fig 13.

7.4.2 Class Principal Components

One problem with the principal component reduction is that if the positive events are rare, then there is a risk that the principal components will only model negative events. With class principal components, the node activations is first split into classes by the label. Then $\frac{k}{2}$ principal components are taken from each class. Here, k is a user defined parameter. In this project, a value of $10 - 40$ was used for k. This was found to be reasonable numbers for the trade-off mentioned in section 7.4 after some testing. In most cases, it was found that class principal components are a better feature reduction than standard principal components.

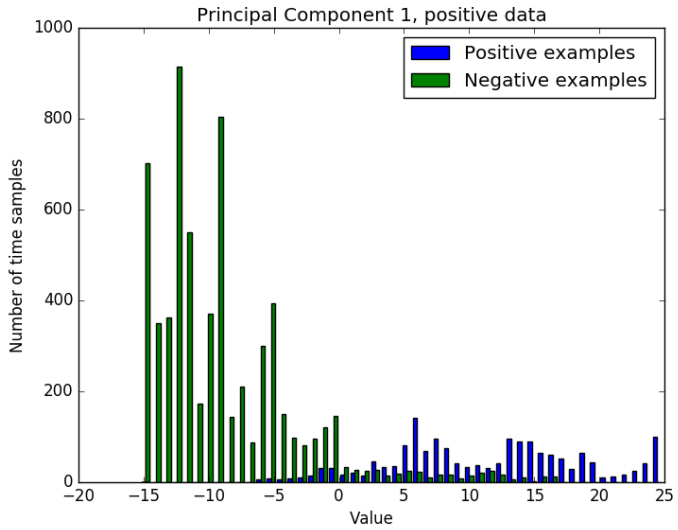

Figure 15: This histogram was constructed by collecting the activations generated in the ESN from the positive examples of the Occupancy data set, and then projecting them on their first principal component. Compare this to figure 13, which was generated with the same activations. The two look similar, but the classes are a bit more well separated in this example.

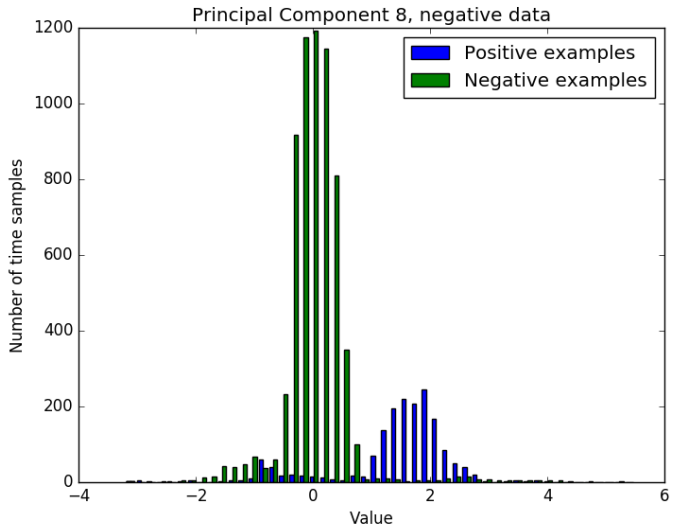

Figure 16: This histogram was constructed by collecting the activations generated in the ESN from the negative examples of the Occupancy data set, and then projecting them on their eighth principal component. Compare this to figure 14, which was generated from the same activations. The two look similar, but this projection clearly gives a better separation of the classes.

7.5 Final Learning

The final step of the model is to fit the reduced features to the label. This is a step that was not subject to a lot of development during the project. Rather, standard algorithms were used.

7.5.1 Linear

Linear learning refers simply to solving a least squares problem from reduced features to label. A constant is added to the feature representation, to account for constant terms. To prevent over-fitting of parameters, Tikhonov regularization is used. To compensate for a majority of positive or negative examples, sample weighting is used. The minimization in equation form:

$$\theta_C = \min_{\theta} ||FX_r\theta - FY||^2 + ||\Gamma X||^2 \tag{21}$$

X_r is a matrix of reduced features from the node activations from the training. Every row corresponds to one time-sample. Y is an array of labels to the training data. Here, F is a diagonal matrix with

$$F_{i,i} = \begin{cases} 1 & Y_i = 0 \\ w_+ & Y_i = 1. \end{cases} \tag{22}$$

The factor Γ is called the Tikhonov Matrix. In this project, $\Gamma = 10I$ was used. This method of regularization is also called Ridge Regression. This regularization method is encouraged by [Rodan2011], where it is used for regression. This minimization has an obvious downside for classification: the least squares solution minimizes the distance to the label, interpreted as a number, and not the miss-classification rate. Nevertheless, the linear classifier is competitive with the other classifiers which is why it's included here. The final classification is done by:

$$\hat{Y}(t) = X_r(t)\theta_C \geq 0.5. \tag{23}$$

7.5.2 Support Vector Machine

The second type of final classifier that was used in this project, was a Support Vector Machine. The settings used were default settings from Scikitlearn's implementation [Scikitlean2017a]. Some tests were done with other kernels, but did not yield good results. It should be said that not much was done to optimize the settings, as the focus was on the feature generation and reduction mostly.

Setting	Value
Kernel	Radial Basis Function
σ^2	$\frac{1}{2}$ number of features

Table 1: The settings used for the SVM classifier throughout the project. Both as a standalone reference algorithm and as a final classifier for ESN. The settings used are default settings from Scikitlearn's implementation [Scikitlean2017a].

7.5.3 Multi Layer Perceptron

The third type of final classifier that was used, was a Multilayer Perceptron. This was applied with little changes from the default settings in the Scikitlearn implementation [Scikitlearn2017b], see figure 2. The only change that was made was to have only one hidden layer, as more layers were found to cause overfitting. As for the number of nodes in the hidden layer, 5 was found to be a suitable number. It should be said that not much was done with the settings on the MLP beyond a working classifier, as the focus of the project was to investigate the generated features in the ESN.

Setting	Value
Hidden layer size	5
Activation	ReLU
L2 penalty	10^{-5}
Optimization algorithm	L-BFGS
Learning rate	10^{-3}
Momentum	0.9
Max iterations	200

Table 2: The settings used for the MLP classifier throughout the project. Both as a standalone reference algorithm and as a final classifier for ESN. All settings except the hidden layer size are default settings according to [Scikitlearn2017b]. For insight into the L-BFGS optimization, see [Liu1989].

8 Implementation

The project was developed in Python 3. The *sparse* subpackage from *Scipy* was used to represent the reservoir [ScipySparse2017]. This is a reasonable choice, since the reservoir matrix is very sparse, with less than one percent of elements being nonzero. For the included learning algorithms, *Scikitlearn* was used. It is an open and free package with a very simple setup. Since the final learning problems in this project were not very large (typically 20 input features with 10 000 samples), an open source implementation like this should be fine. The Support Vector Machine implementation used was *Scikitlearn*'s *SVC* and *SVR* for classification and regression respectively [Scikitlearn2017a]. Similarly, *Scikitlearn*'s *MLPClassifier* and *MLPRegressor* were used for the feed forward neural networks [Scikitlearn2017b]. These implementations were used both for standalone classification, and as the final step in Component ESN. For *K-means*, *Scikitlearn*'s *KMeans* was used.

8.1 Using the System for New Data

In order to comply with the testing system, only one function needs to be implemented: one that a list of takes file names as input, and outputs a list of data and labels in matrix format. The design of the reservoir is defined in a settings file separate from the rest of the code. The rest of the runtime parameters are defined by flags in the terminal. Such as which model to use, and whether to plot all graphs.

8.2 Overall Structure

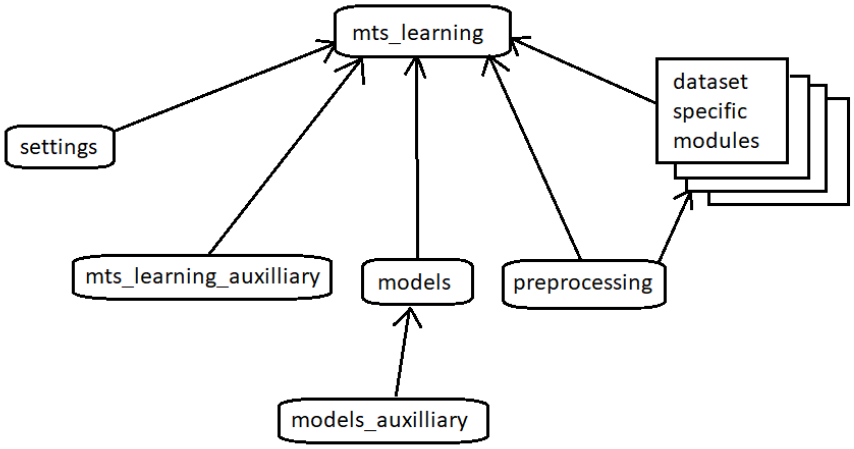

Figure 17: The structure of the software that was developed for preprocessing, modelling, and evaluating the data sets. Each box is a Python module. An arrow from A to B means that B imports from A.

Some small descriptions of the different modules.

- mts_learning. The main script. Organizes the pipeline from fetching data to evaluating results.
- mts_learning_auxilliary. Unloads code-heavy tasks from mts_learning. For example plotting.
- models. Contains objects that describe the different classifiers used in the project: SVM, MLP, and ESN. The SVM and MLP classes are just wrapper classes that allows mts_learning to treat them as if they were time series models. Performs the calculations for learning, and later, predictions.
- models_auxilliary. A module for synthesizing Component ESN, and for doing the heavy lifting in the feature reduction.
- preprocessing. Collects some methods that are often used for preprocessing, such as filtering out missing values and normalization.
- settings. Stores the configurations for synthesizing the Component ESN.

Also, every data set has its own script, since the preprocessing is very data specific, and the code for doing it should not be mixed with the general learner system.

9 Results

9.1 Eye

The data source [Eye2013] contains a single file of (roughly) 15000 samples taken during 2 minutes. The whole data set was used for training and testing. For the reference SVM, default settings were used. The ratio of weights between positive and negative examples was 1. For the reference MLP, 5 hidden nodes were used. For the Basic ESN, 500 nodes were used for the reservoir. The feature reduction for Basic ESN was Class PCA with 20 outputs, and the final learner was a linear classifier with same weight for positive and negative examples. As noted in Section 5.2, looking at the EEG signals gives us two hints about how to design the reservoir. Firstly, the features are similar in appearance, so it is logical to choose a reservoir which is symmetrical with respect to different features. With the randomization, the reservoir will not be perfectly symmetric of course, but the randomization is symmetric. Second, since the start and ends of the sections where the person has his eyes closed are marked by a spike in some features. Since there can be thousands of samples between spikes, a reservoir which has a good long term memory is reasonable. The chosen architecture can be seen in Fig 18. Class Principal Components with 20 reduced features was the feature reduction that worked best. An SVM was used as final learner. When training the SVM, positive and negative examples were given the same weight, as there about as many samples of each. A comparison of the results on the test set is shown in Table 3. Sample outputs are shown in Fig 19-23.

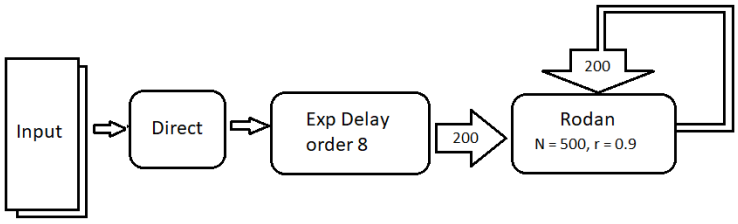

Figure 18: The architecture for the Component ESN used for the Eye data set. The order of the Exponential Delay line is set quite high since it was expected that the model needs to be persistent. The used feature reduction was Class Principal Components with 20 reduced features (10 per class). An SVM was used as the final learner.

Training and testing on were both done on the whole data set. Since the Eye data set is so small, the goal was to answer the questions in Section 5.2, rather than to compare the generalization performance of the algorithms.

	Recall	Precision	F-measure	Classification accuracy
KStar	-	-	-	0.974
SVM	0.867	0.919	0.892	0.906
MLP	0.819	0.777	0.797	0.813
Basic ESN	0.993	0.993	0.993	0.994
Component ESN	0.985	0.992	0.988	0.990
Component ESN, reduced features	0.994	0.991	0.993	0.993

Table 3: Results obtained for the Eye data set. Note that this is *training performance*, because the data is so small. *KStar* is the best algorithm reported in [Suendermann2013]. The reduced features are the four: AF3,AF4,F3,F4 (see Fig 2).

Below are classification outputs on the training/test set for the Eye data set, for different models. See Figures 19 - 22. For the Component ESN, learning on fewer features was also tried, see Fig 23.

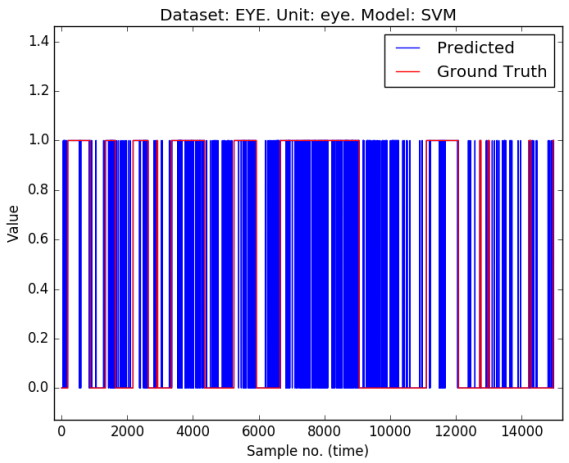

Figure 19: Training output of SVM classifier for Eye data set. This shows that the SVM clearly has not learned to separate the classes, as there are many false positives and negatives.

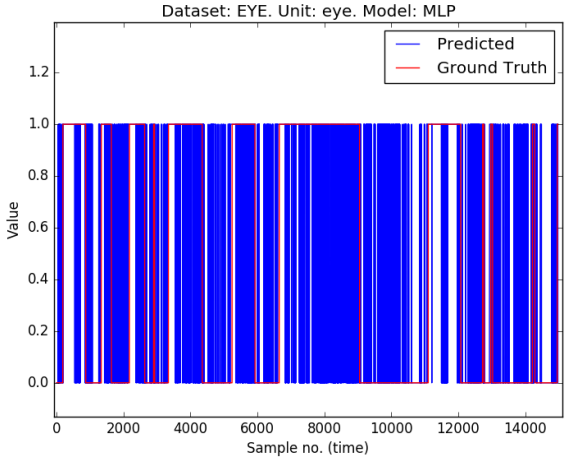

Figure 20: Training output of MLP classifier for Eye data set. The results is similar to what was seen with the SVM - the classifier has not learned to separate the classes.

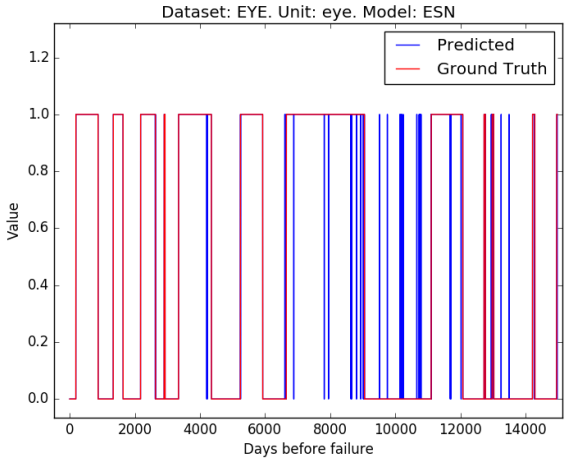

Figure 21: Training output of Basic ESN for Eye data set. The difference from the baseline algorithms is noticeable as the predictions are almost precisely as the ground truth.

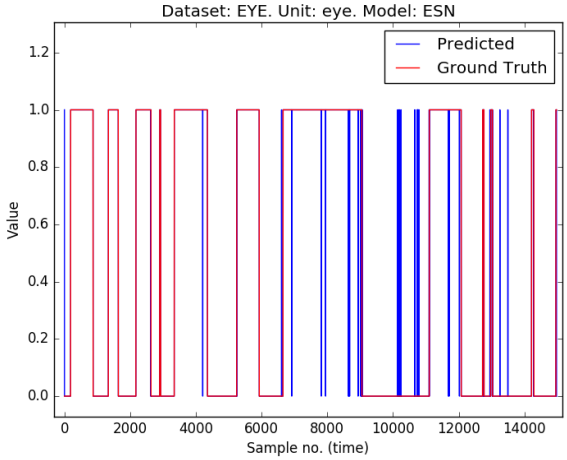

Figure 22: Training output of Component ESN for Eye data set. The prediction is slightly worse than for the basic ESN, but still clearly better than the standard algorithms.

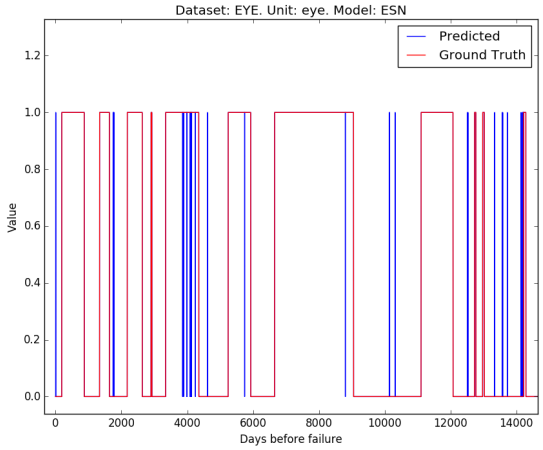

Figure 23: Training output of Component ESN for Eye data set, using only 4 out of 14 features. The features were chosen as the sensors on the front of the head. This actually improves the accuracy of the algorithm, so it seems that there is a problem with noisy features in the data set.

9.2 Occupancy

In the [Occupancy2016] folder, there are three files. The file *datatraining.txt* (8143 samples) was used for training, and *datatest.txt* and *datatest2.txt* (2804 and 9752 samples respectively) are used for validation and testing. Since the sample time is 60 seconds, that comes down to about 134 hours for training and 209 hours for testing. For the reference SVM, default setting from [Scikitlearn2017a] were used. The ratio of weights between positive and negative examples was 1. For the reference MLP, 5 hidden nodes were used. For the Basic ESN, 500 nodes were used for the reservoir. The feature reduction for Basic ESN was Class PCA with 20 outputs, and the final learner was a linear classifier with same weight for positive and negative examples. The Component ESN architecture is shown in Fig 24. A linear delay line component is added to give the reservoir some safer short term memory. A threshold component is added to accentuate strong derivatives of the time series, since it seems that derivative can be a useful feature. The used feature reduction was Class Principal Components with 20 reduced features (10 per class). A Linear classifier was used as the final learner. A comparison of the results on the test set is shown in Table 4. Sample outputs are shown in Fig 25-28.

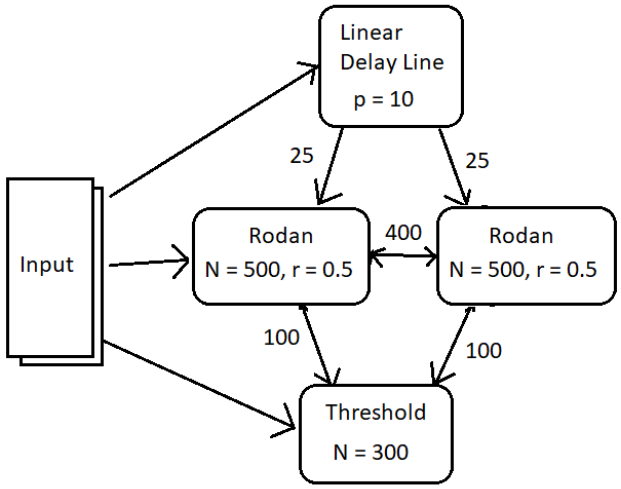

Figure 24: The architecture used for the Component ESN for the Occupancy data set. The linear delay line component is added to give the reservoir some safer short term memory. The threshold component is added to accentuate strong derivatives of the time series, since it seems that derivative can be a useful feature. The used feature reduction was Class Principal Components with 20 reduced features (10 per class). A linear classifier was used as the final learner.

	Recall	Precision	F-measure	Classification accuracy
LDA	-	-	-	0.988
SVM	0.979	0.859	0.915	0.956
MLP	0.906	0.833	0.868	0.933
Basic ESN	0.997	0.954	0.975	0.988
Component ESN	0.992	0.965	0.978	0.989

Table 4: Results obtained for the Occupancy data set. The data set was already split into training and testing with training consisting of 28% of the time-samples. *LDA* (Linear Discriminant Analysis) is the best algorithm reported in [Candanedo2016].

Below are example outputs from the classifiers. The total duration of the sample is six days. Since the equipment was set up in an office, it is a reasonable interpretation that each block is a working day with a "lunch" in the middle. The two days without occupancy is the weekend.

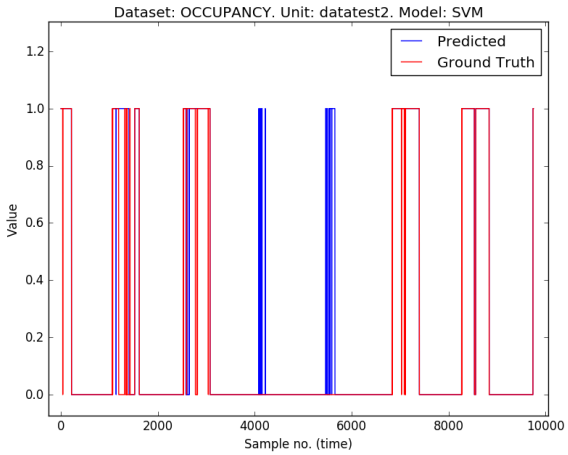

Figure 25: Test output of SVM classifier for Occupancy data set. The predictions are mostly correct for the weekdays (the periods of positive labels), however the algorithm makes a lot of false positives during the weekend (the middle of the graph).

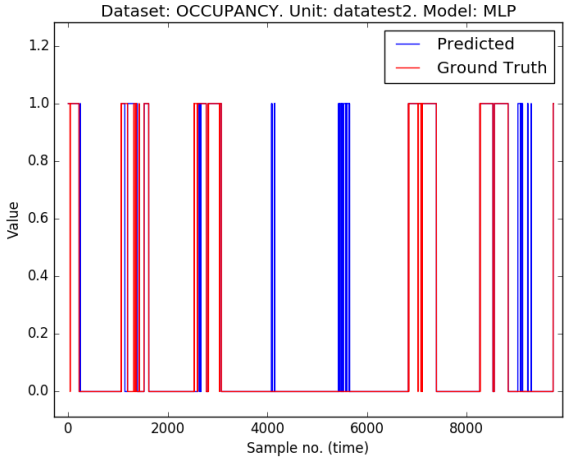

Figure 26: Test output of MLP classifier for Occupancy data set. the Results are similar to SVM: mostly correct during weekdays, but makes false positives during weekend. Additionally makes false positives between two days in the right of the graph.

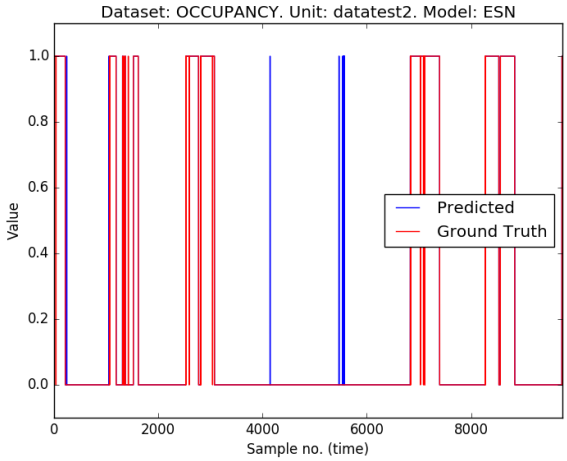

Figure 27: Test output of basic ESN for Occupancy data set. Even if the false positives are fewer during the weekend, they are still there. What is hard to see in this graph is the breaks of a few minutes during the workdays. Typically, these are false positives for the classifier.

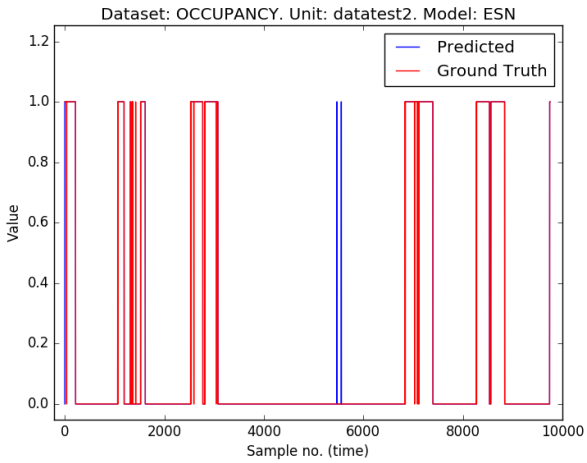

Figure 28: Test output of component ESN for Occupancy data set. The component ESN does not make any false positives for the Saturday (the weekend is in the middle of the graph). For the Sunday, there is only a small number of false positives. One could likely eliminate the false detections by some rule that the detection must be positive for some consequetive time. What is hard to see in this graph is the breaks of a few minutes during the workdays. Typically, these are false positives for the classifier.

9.3 Hard Disk

For the below results, hard drives of product model *ST4000DM000* and with a serial number starting with *Z300* were used for tinkering with the model parameters. For final testing, 30 units with a serial number starting with *Z301* were used for training and 71 for testing. Only units that broke during the data capture were used. For the reference SVM, positive examples were weighted 5 times higher than negative examples. For the reference MLP, 7 hidden nodes were used. For the Basic ESN, 1000 nodes were used for the reservoir. For both Basic and Component ESN, the feature reduction used was Class PCA with 20 outputs. An MLP with 5 hidden nodes was used as final classifier for the ESNs. The architecture for the Component ESN can be seen in Fig 29. A comparison of the results on the test set is shown in Table 5. With the Hard Disk data, the relation between the computed performance and the practical usefulness is not so clear as with the previous two data sets. Figures 30-34 demonstrate this. Graphs for estimating expected lost days are shown in Figures 36-35.

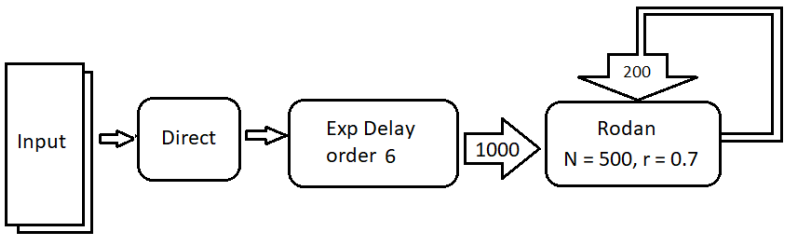

Figure 29: The architecture for the Component ESN used for the Hard Drive data set. The exponential delay line is included for persistence. With 13 input features, the 1000 links sends the activation of every "sum" node in the exponential delay line to about 13 nodes in the remaining reservoir. The used feature reduction was Class Principal Components with 20 reduced features (10 per class). An MLP with 5 hidden nodes was used as the final learner.

	Recall	Precision	F-measure
SVM	0.35	0.19	0.25
MLP	0.24	0.17	0.20
Basic ESN	0.23	0.13	0.16
Component ESN	0.27	0.14	0.18

Table 5: Results obtained for testing on 70 drives from the Hard Drive data set.

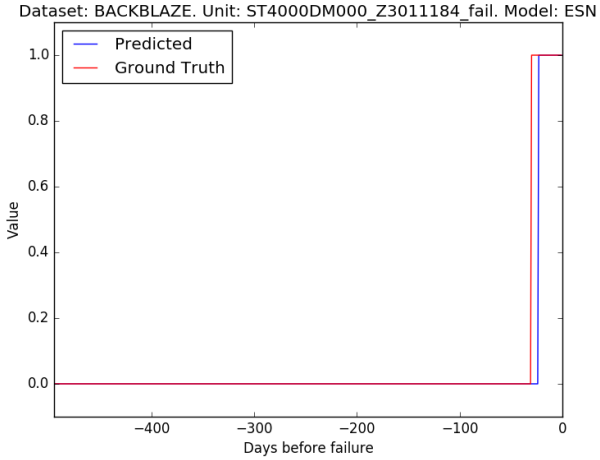

Figure 30: Example of output where the classifier has made a near perfect classification. The prediction will get a good F-score, and is also practically useful.

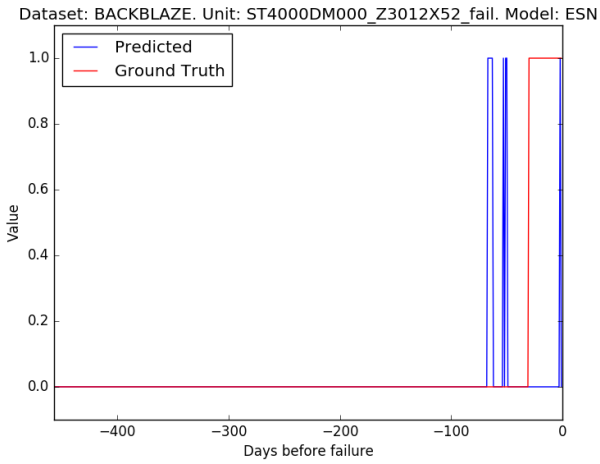

Figure 31: Example of a classifier output which will get a bad F-score but which is still valuable.

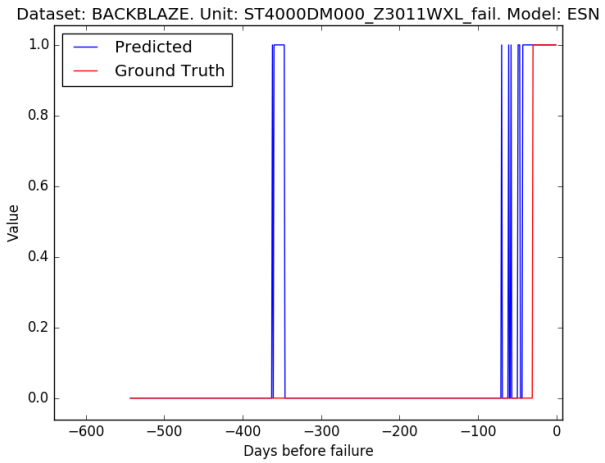

Figure 32: Example of a classifier output which will get a good F-score but which will lead to the unit being taken out of service prematurely.

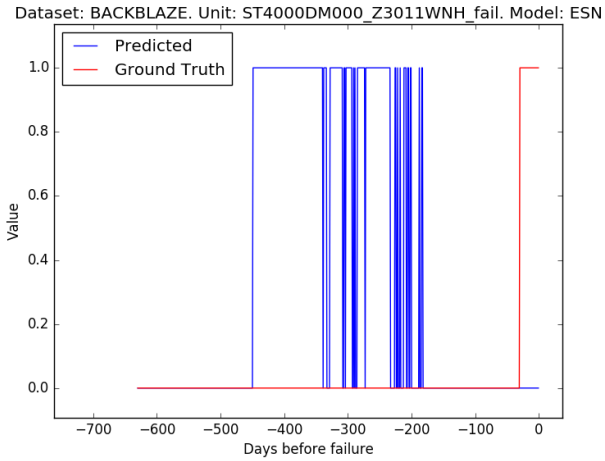

Figure 33: Example of a bad classifier output. The posed learning problem is not solved, nor is the prediction valuable.

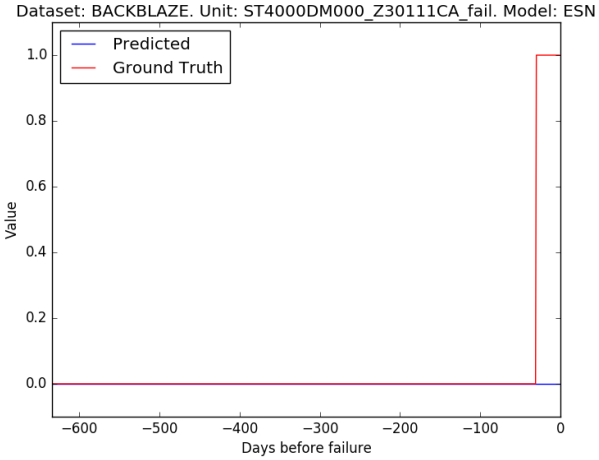

Figure 34: Example of the majority of classifier outputs: no positive predictions whatsoever.

The below images show how many hard drives that have received a *warning* from the system, as a function of time before their failure. Different warning rules are applied. The warning rule *2 in a row*, for example, warns for impending failure when the classifier has made a positive prediction for two consequetive days. The units in this test are the "lost cases", i.e. the hard drives that did not exhibit any critical error before failing, see section 5.4.3. A perfect warning system here would be a flat line at 0, and would go up to 1 just before failure ($x = 0$). The *Data availability* line marks the cumulative distribution of the lifetimes of the hard drives in the test set. Note that the data capture does not necessarily start when the drives are newly installed.

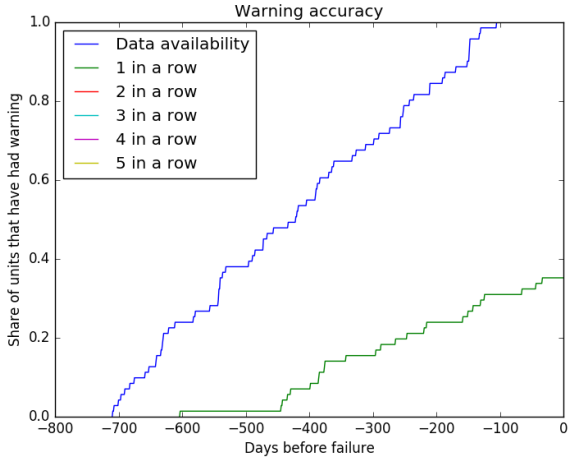

Figure 35: Warning accuracy of a "random" warning system, which warns the user each day with a probability of $\frac{1}{800}$. This probability was chosen so that the final recall would be about 0.35, which is was the other algorithms had.

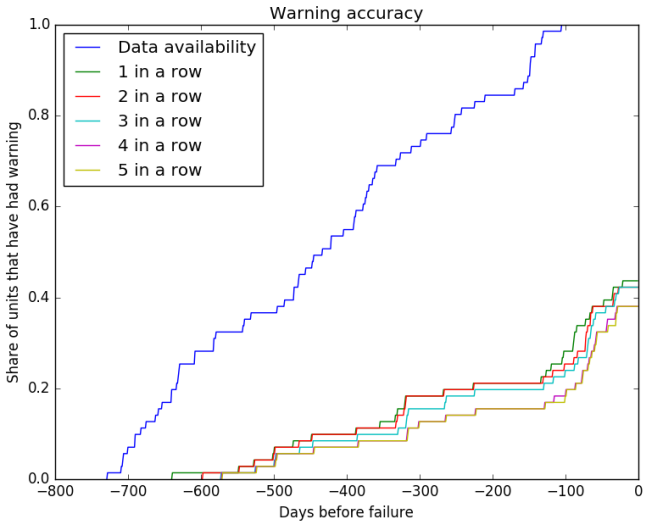

Figure 36: SVM classifier warning times according to different warning rules. Starts making a lot of warnings at about 400 days before failure, which would cause a lot of lost day if it was used as a reason to exchange the drive. Would cause roughly the same amount of lost days as a "random" warning system, so does not do anything useful.

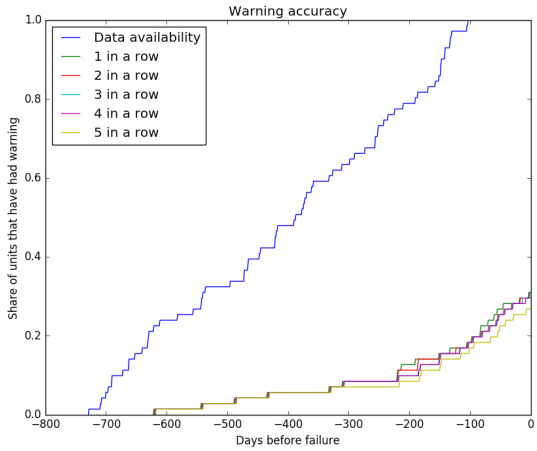

Figure 37: MLP classifier warning times according to different warning rules. Just as SVM, starts returning premature warnings at 400 days before failure. However, the number of lost days here is clearly better than with SVM. May very well model some relevant indicators.

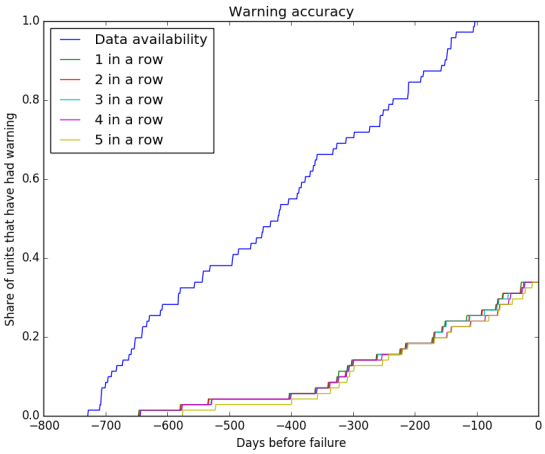

Figure 38: Basic ESN classifier warning times according to different warning rules. Basic ESN starts giving premature warnings a bit later than MLP, but the total number of lost days is about the same. Just as with SVM and MLP, it is hard to say whether this is actually better than the random classifier.

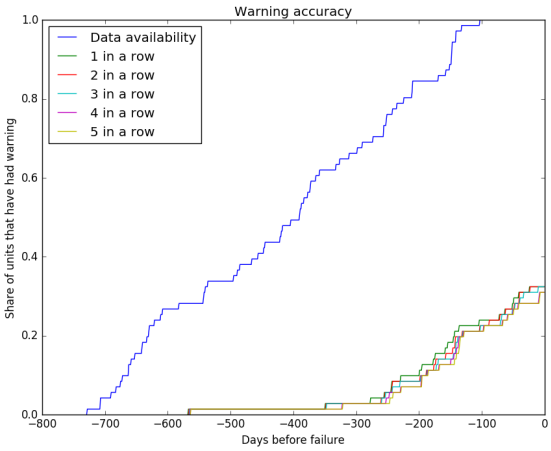

Figure 39: Component ESN classifier warning times according to different warning rules. Gives almost no premature warnings until 300 days before failure. The component ESN does seem to be better than a random classifier. Note that this test was done on the hard drives which did not display any of the critical errors known to the storage industry, and mentioned in section 5.4.3. So this is predictions on the "lost cases".

10 Analysis

In this section, observations made in the previous section are combined with what can be said theoretically about the Component ESN, in order to answer the qualitative questions made about the data sets in section 5. For this, we need to know something about the hypothesis space which is generated by the ESN.

10.1 Hypothesis Space of Basic ESN

Suppose that we have a univariate process which is fed into a Basic ESN, as described in section 4. Call the input weight v, the delay weight r, and the input signs s_k. The activation of node k at time t is then:

$$\begin{aligned} x_{k,t} = \phi(vs_k u_t + rx_{k-1,t-1}) = \\ \phi(vs_k u_t + r\phi(vs_{k-1}u_{t-1} + rx_{k-2,t-2})) = \\ \phi(vs_k u_t + r\phi(vs_{k-1}u_{t-1} + r\phi(vs_{k-2}u_{t-2} + rx_{k-3,t-3})) = ... \end{aligned}$$

where $\phi = \tanh()\dot{}$. A natural question about a recurrent network is: how long is its memory? That is, how much can past values of the process influence the

current state? Call the "unactivated" nodes $z_{k,t}$, i.e.

$$z_{k,t} = \phi^{-}1(x_{k,t}).$$

A derivative expansion gives:

$$\frac{\partial x_{k,t}}{\partial u_{t-\tau}} = r\frac{\partial x_{k-1,t-1}}{\partial u_{t-\tau}}\phi'(z_{k,t}) =$$
$$r^2\frac{\partial x_{k-2,t-2}}{\partial u_{t-\tau}}\phi'(z_{k,t})\phi'(z_{k-1,t-1}) = ...$$
$$r^\tau v s_{k-\tau}\phi'(z_{k,t})...\phi'(z_{k-\tau,t-\tau}).$$

The absolute value of this derivative is then bounded from above by $O(r^\tau)$, since $0 < \phi'(z) \leq 1$, and v is a constant. Since this is true for any values of the nodes, if $u_{t-\tau}$ is perturbed by a finite value Δ then the future value of $x_{k,t}$ cannot change by more than $O(\Delta r^\tau)$. So, the effect of past values is necessarily exponentially decaying in the Basic ESN!

Suppose now that $v = 1$, and that we consider changes less than δ to be insignificant for the activation of a node. The effective memory of a node is then less than $\frac{\log(\delta)}{\log(r)}$, see Table 6.

r	δ	effective memory T
0.5	0.05	4
0.5	0.01	6
0.7	0.05	8
0.9	0.05	28

Table 6: The effective memory of nodes in a Basic ESN, given values of the delay weight r, and the smallest significant change δ. The values are rounded down to the nearest integer.

Since the only difference between node topologies in the Basic ESN is the input sign, the total number of significantly different features which can be generated by the Basic ESN is 2^T. This is then the hypothesis space that is available to the feature reduction and final learning.

10.2 Analyzing Generated Features

Looking at the graphs of the reduced features alongside the input features, makes it possible to say something about which qualities do get extracted from the reservoir.

10.2.1 Eye

See Fig 40. We are now ready to answer the questions posed in Sections 5.2. Firstly, a homogeneous and persistent reservoir worked well. The generated features are apparently not smoother than the input features. This could perhaps

be done with preprocessing instead, if smoothing is important. However, we can see that the generated features are clearly more persistent than the input features. For this data set, this is a good quality since we have typical "event starting indicators" that should not be forgotten quickly. Furthermore, Fig 23 shows that using just 4 of the 14 features is sufficient to represent the data even better than in the original article [Suendermann2013].

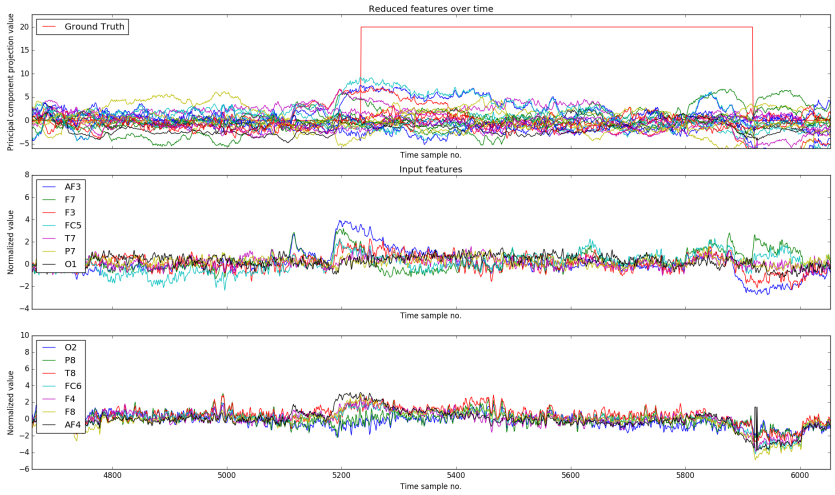

Figure 40: A section of the Eye data set, aligned with the corresponding reduced features generated by the Component ESN. We can see that a short-time spike in the input features results in a long "echo" in the generated features. This seems to be a good rule, if one looks at the input data.

10.2.2 Occupancy

See Fig 41. There we can see that the features which are marked by blue lines, have basically singlehandedly solved the trickiest part of the modelling: the case when light and temperature is high, but when the room is unoccupied. But as can be seen, the blue lines are have pretty much the same sign as the label (albeit reversed). So, this was a successful feature generation.

Now see Fig 42. The question was asked whether the algorithm would be able to detect small breaks of a few samples when the occupant leaves the room for a few minutes (and leaves the light switch on). It seems as though the answer is no. Looking at the input features, they do not change noticeably during the small breaks. And the generated features have not done any "magical" thing and picked it up, either. However for the end goal of saving energy on heating, these small breaks are not important to classify correctly since heating is such a slow process compared to the length of these intervals.

Lastly, see Fig 43. It was asked whether some generated features would be invariant to large-scale daily changes, but still sensitive to changes over minutes and hours. A sort of high-pass quality, that is. The feature in mind particularly is Temperature. Also here, it seems that the answer is no. Compare the left and right blocks. The input features have a similar shape in both, but also a different mean. The same seems to apply for the reduced features.

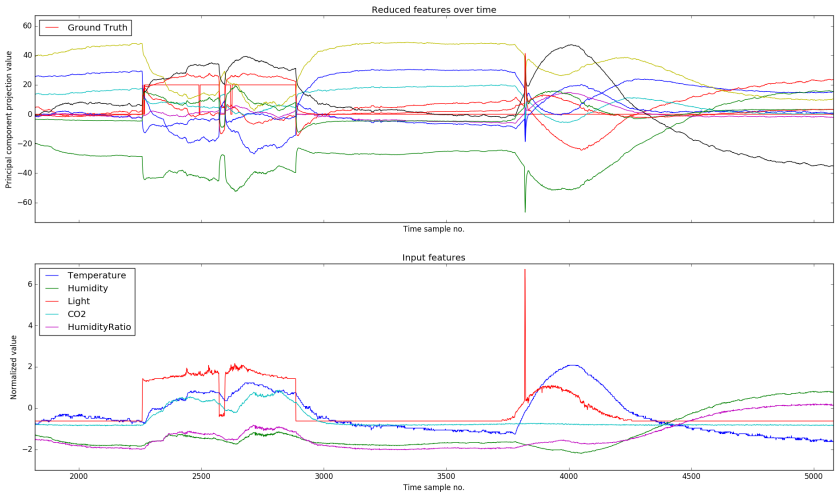

Figure 41: A section of the Occupancy data set, aligned with the corresponding reduced features generated by the Component ESN.

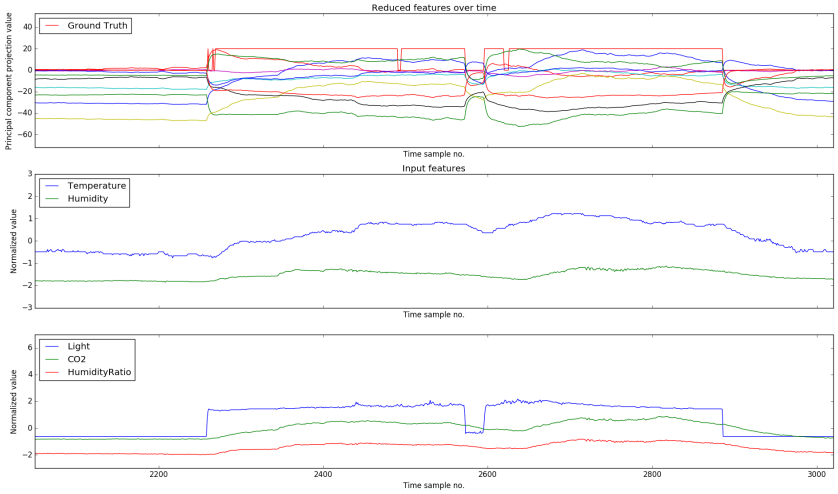

Figure 42: A section of the Occupancy data set, aligned with the corresponding reduced features generated by the Component ESN.

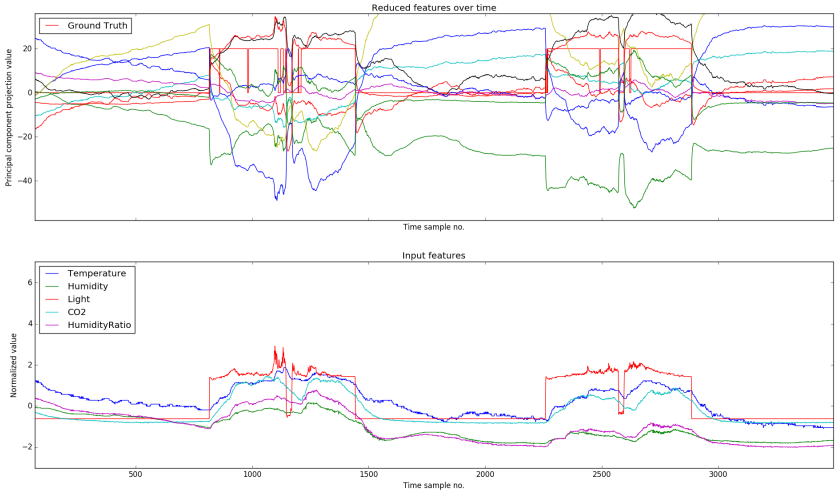

Figure 43: A section of the Occupancy data set, aligned with the corresponding reduced features generated by the Component ESN.

10.2.3 Hard Drive

Reduced feature graphs can be seen in Figures 60-67 in Section 15.2 in the Appendix. We can now answer the questions posed in Section 5.4.3. Looking at Figs 60 and comparing it to 61, we see that the Component ESN successfully picks out SMART 193: Load Cycle count as a potential indicator of impending failure, while the Basic ESN does not. The same can be said about SMART 184: End-to-End Error, with Figs 66 and 67. This pattern can be checked with the feature plots for SMART 184 and SMART 193, see Figs 50 and 54 respectively. In general, we can see from Figures 60 - 67 that patterns from the input reoccur in the generated features. So, the reservoirs mainly pass through the input unchanged in detail, but makes some linear combinations and amplifies some patterns. Furthermore, comparing Fig 64 to Fig 65, we see that it happens that the Basic ESN accentuates an interesting pattern, while the Component ESN does not.

10.3 Analyzing the Classifier Outputs

Looking at the prediction graphs and test metrics, we can identify performance differences between the tested algorithms.

10.3.1 Eye

We can see that the ESN models give a consistently better prediction that the static time classifiers. There is no great difference between the Basic and the Component ESN, but using only a few select input features improves the result. Both according to the test metrics, and ostensibly in the prediction graphs. However, since the test was done on the same data as the training, one should be very careful to draw any conclusions from what is likely an overfit model. Especially since complex models are more liable to overfit than simpler ones. It is worth noting that the classification time (i.e. not learning time) reported by [Suendermann2013] is *38 minutes*, a considerable time for classifying a sample taken from 2 minutes real time data. With the algorithms used here, classification time is only a few seconds.

10.3.2 Occupancy

Also for Occupancy, the ESN models outperform the SVM and the MLP. The Component ESN performs slightly better than Basic ESN, as it only gets one false positive day, see Fig 28. However, considering the end goal of strategic heating, the false positives made by the ESNs are probably not a problem since they are isolated spikes. Thanks to this, it should be possible to make a "safe" decision rule based on filtering the classifier output. Even though testing was not done on the training data here, the two samples share some characteristics. When looking at the reduced features in for example Fig 43, it is clear that the "light" input feature has a great impact. Looking at that feature, it is apparent that it depends mostly on the electric lighting in the room. What would be

the interesting caveat case for the classification is when someone has gone home for the day and forgot to turn of the light switch. Without such cases in the data, it is hard to evaluate the real-world performance of the classifiers for this problem.

10.3.3 Hard Drive

First of all in Table 9.3, we see that the formal learning problem has not been solved very well by any of the algorithms. Calculations on the data give that an algorithm which always makes a positive prediction would get an F-score of 0.132. So the classifiers are not remarkably better than the trivial predictor, as far as the formal problem goes. However, Figures 30 to 34 demonstrate that the test metrics are problematic in this case. This is because the used test metric doesn't take into particular account the time for the *first* warning. In practice, a guarantee on a short time (but not too short) from the first warning to failure is necessary to avoid many prematurely discarded units. And looking at the warning accuracy graphs, we can see that the Component ESN outperforms the other models. The rate of units which have been warned before failure is about the same for all classifiers, but Component ESN has a lower rate of premature warnings. It also has fewer premature warnings than a random warning system with similar recall, see Fig 35. Furthermore, we can see that using different thresholds for warning, given the classifier output, does not show potential as a cheap filtering.

11 Conclusion

An initial conclusion based on the results from Occupancy, is that when the Basic ESN already has a good result, it is hard to improve it with a Component ESN. Since in the Eye case, testing was done on the same data as training, it cannot be used to say anything about learning performance. For the Eye and Occupancy data sets, however, the Component ESN is able to generate relevant features with the help of its components. These may help with the generalization of the classifier, but one cannot say that for certain without testing it one more varied data. For the Hard Drive data, the Component ESN is shown to outperform the Basic ESN and the reference algorithms, which are not much better than a random warning system. So, it can be said that Component ESN has potential as a new modeling tool for classifying multivariate time series. The best feature reduction technique was Class PCA, and regular PCA is also valuable. As for the final classifiers, the implemented ones were good for different data sets. So, it's worth it to try out some different classifiers when working with new data.

12 Discussion

In this project, only minimal preprocessing was done with the data. A 1-day limit was used as a goal as for how long time it was allowed to work on extracting, preprocessing, and finding a suitable architecture. For the Hard Disk data set, this limit was surpassed, as the data had to be "transposed" (it was originally one file per day, and not one file per unit) and it also had to be inspected for a while to be understood. There were also missing values which had to be handled. For the Eye data set however, it only took 2 hours from downloading the data to having a Component ESN model that improved on the results in the original article. The Occupancy data set did not take much time to preprocess either. This is seen as a clear positive, that the system is quick and easy to try out on a new data set (at least for the creator). Only one function needs to be implemented to comply with the testing system: one that takes file names as input, and outputs data and labels in matrix format. The design of the Component ESN is defined in a text file.

It also turned out that a simple design of the Component reservoir was hard to improve. Partly because there are a lot of design parameters, and it's hard to know what will actually affect the result positively. Overall, it is hard to test whether the *paradigm* of Component ESNs is a good idea, since this depends on many practical and contextual factors. For example, how much time, data, and computational resources are available. Above, it is concluded that the system can be deployed quickly for new data. It can also be scaled up or down, to suit the amount of available data. With Occupancy, we have seen that a small amount of data is sufficient. The calculations are computationally expensive, however. Based on measurements on a Intel Core i5-4300U CPU at 1 core on 1.90GHz, training an ESN with the architecture in Fig 24 takes about 20 seconds, while training the SVM takes less than 2 seconds. This is with the numpy C backend and sparse matrices, that optimize the matrix multiplications. However, only one processor core is used at a time. A possibility of mitigating this is that some things can be done in parallel: collecting activations for different units can be completely parallelized. Calculating the activations for a single unit is also just a matrix multiplication and an elementwise activation, so it can be entirely parallelized also. One bottleneck is to compute Principal Components of a large number of large vectors, as is done in the feature reduction. If there was a way to (approximately) calculate the combined principal component of different sets of vectors, this could speed up computations many times over.

13 Future Work

13.1 Scaling up

As noted in the Discussion, it would be possible to parallelize many of the computations. With the help of this, training and test time classification could be made much faster, and tackle larger data sets in reasonable time.

13.2 User Interface

One development which could help users get a better overview of their reservoirs, is to have a graphical interface where the reservoir is shown in graph form, as in Figures 18, 24, 29. One way of implementing this could be to "piggyback" on TensorFlow's built-in graph visualization tool [TFGraphs2017]. Implementing the Component ESN in TensorFlow or Theano or similar, could also be a simple and accessible way of making parallelization possible. All parts of the Component ESN take in a tensor (a vector) and outputs another tensor (also vector), so it fits the computational model of TensorFlow perfectly well.

13.3 Relationship between Architecture and Features

The large research question that comes out of this project is: given a data set with some known properties, how does one design a reservoir which will generate good features? In order to understand the components' effects, they could be studied in isolation. With the analysis in Section 10.1 and the result in [Jaeger2001b], we see that the "memory" of ESNs is limited, although it is a time-recurrent model. It would be enlightening to make similar analysis of how many units of a certain component, and/or how much linking is needed between certain components, for it to be likely to capture a given pattern in the data with the reservoir. For testing the components in this project, "perfect" data was simulated. Considering the apparent difficulty of with real data in order to learn about a model, it could be useful to simulate more complex but less noisy data, and see if the Component ESN can capture this. And in that case, what is the smallest possible ESN architecture that is likely to model the simulated data?

13.4 New Components

In this paper we have seen example of a few components being used and combined, but of course different components could be made. One interesting development could be to have a semi-trained reservoir, where some components are first trained in an unsupervised fashion. This could solve normalization problems, which a completely untrained reservoir is vulnerable to.

14 References

14.1 Company

Sentian2017 http://www.sentian.ai

14.2 data sets

PCoE2017 https://ti.arc.nasa.gov/tech/dash/pcoe/prognostic-data-repository/ . Viewed on 2017-02-01.

Backblaze2017a https://www.backblaze.com/b2/hard-drive-test-data.html . Data downloaded on 2017-02-16.

Occupancy2016 https://archive.ics.uci.edu/ml/machine-learning-databases/00357/ . Data downloaded on 2017-04-25.

Eye2013 https://archive.ics.uci.edu/ml/machine-learning-databases/00264/ . Data downloaded on 2017-06-04.

Suendermann2013 Rösler, Oliver, and David Suendermann. "A first step towards eye state prediction using eeg." Proc. of the AIHLS (2013).

Candanedo2016 Accurate occupancy detection of an office room from light, temperature, humidity and CO2 measurements using statistical learning models. Luis M. Candanedo, Véronique Feldheim. Energy and Buildings. Volume 112, 15 January 2016, Pages 28-39.

Softraid2017 https://www.softraid.com/pages/features/softraid_monitor.html. Viewed on 2017-05-22.

Hughes2002 Hughes, Gordon F., et al. "Improved disk-drive failure warnings." IEEE Transactions on Reliability 51.3 (2002): 350-357.

Backblaze2017b https://www.backblaze.com/blog/what-smart-stats-indicate-hard-drive-failures/ . Viewed on 2017-02-16.

Backblaze2017c https://www.backblaze.com/blog/hard-drive-failure-rates-q1-2017/ . Viewed on 2017-05-23.

Pinheiro2007 Pinheiro, Eduardo, Wolf-Dietrich Weber, and Luiz André Barroso. "Failure Trends in a Large Disk Drive Population." FAST. Vol. 7. 2007.

14.3 Code packages

ScipySparse2017 https://docs.scipy.org/doc/scipy-0.18.1/reference/sparse.html. Viewed on 2017-03-28.

Scikitlearn2017a http://scikit-learn.org/stable/modules/svm.html. Viewed on 2017-04-20.

Scikitlearn2017b http://scikit-learn.org/stable/modules/neural_networks_supervised.html. Viewed on 2017-04-20.

Scikitlearn2017c http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html. Viewed on 2017-04-27.

TFGraphs2017 https://www.tensorflow.org/get_started/graph_viz . Viewed on 2017-07-23.

14.4 Articles

Liu1989 Liu, Dong C., and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization." Mathematical programming 45.1 (1989): 503-528.

Bengio1994 Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." IEEE transactions on neural networks 5.2 (1994): 157-166.

Schmidhuber1997 Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

Jaeger2001a Jaeger, Herbert. "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note." Bonn, Germany: German National Research Center for Information Technology GMD Technical Report 148.34 (2001): 13.

Jaeger2001b Jaeger, Herbert. Short term memory in echo state networks. Vol. 5. GMD-Forschungszentrum Informationstechnik, 2001.

Maass2002 W. Maass, T. Natschläger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. Neural Computation, 14(11):2531–2560, 2002.

Keogh2004 Keogh, Eamonn, et al. "Segmenting time series: A survey and novel approach." Data mining in time series databases 57 (2004): 1-22.

Jaeger2005 Jaeger, Herbert. "Reservoir riddles: Suggestions for echo state network research." Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on. Vol. 3. IEEE, 2005.

Jaeger2007 Jaeger, Herbert, et al. "Optimization and applications of echo state networks with leaky-integrator neurons." Neural networks 20.3 (2007): 335-352.

Tong2007 Tong, Matthew H., et al. "Learning grammatical structure with echo state networks." Neural networks 20.3 (2007): 424-432.

Saxena2008 Saxena, Abhinav, et al. "Damage propagation modeling for aircraft engine run-to-failure simulation." Prognostics and Health Management, 2008. PHM 2008. International Conference on. IEEE, 2008.

Rodan2011 Rodan, Ali, and Peter Tino. "Minimum complexity echo state network." IEEE transactions on neural networks 22.1 (2011): 131-144.

Lin2011 Lin, Xiaowei, Zehong Yang, and Yixu Song. "Intelligent stock trading system based on improved technical analysis and Echo State Network." Expert systems with Applications 38.9 (2011): 11347-11354.

Busseti2012 Busseti, Enzo, Ian Osband, and Scott Wong. "Deep learning for time series modeling." Technical report, Stanford University (2012).

Bengio2013 Bengio, Yoshua, Aaron Courville, and Pascal Vincent. "Representation learning: A review and new perspectives." IEEE transactions on pattern analysis and machine intelligence 35.8 (2013): 1798-1828.

Bianchi2015 Bianchi, Filippo Maria, et al. "Short-term electric load forecasting using echo state networks and PCA decomposition." Ieee Access 3 (2015): 1931-1943.

Han2015 Han, Min, and Meiling Xu. "Predicting multivariate time series using subspace echo state network." Neural Processing Letters 41.2 (2015): 201-209.

15 Appendix

15.1 Plots of SMART Features

SMART features plotted for 3009 drives of type *Seagate Desktop ST4000DM000*. The graphs have been right-aligned so that they all end at the same time.

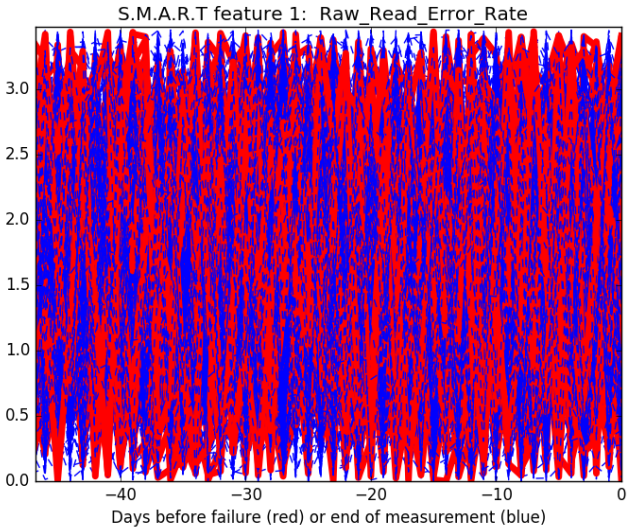

Figure 44: Note that in this graph, the x-axis has been rescaled so as to show some more patterns in the graphs. It is not possible to see any interesting patterns in this feature, based on this graph.

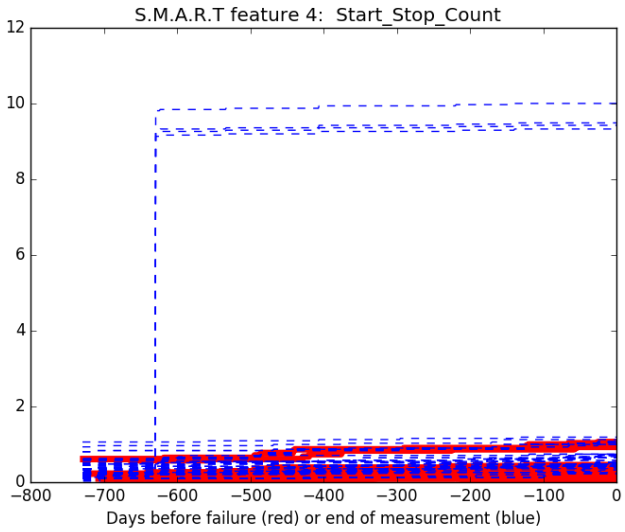

Figure 45: Does not seem to have any relevance for predicting impending failure on its own.

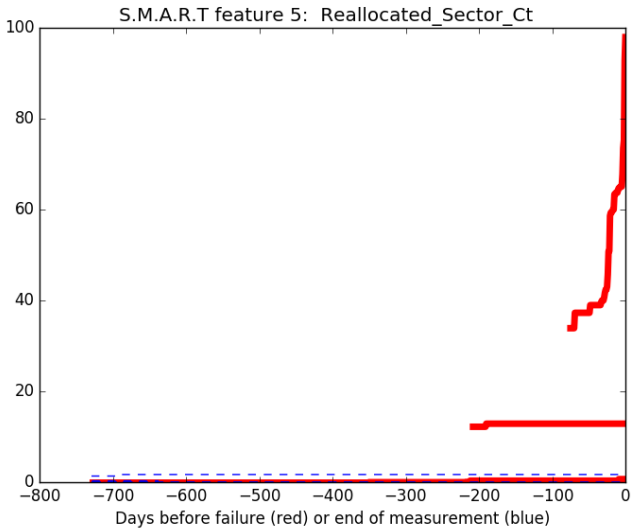

Figure 46: Clear indication that a large value means that drive is about to break. Is used by storage industry [Backblaze2017b].

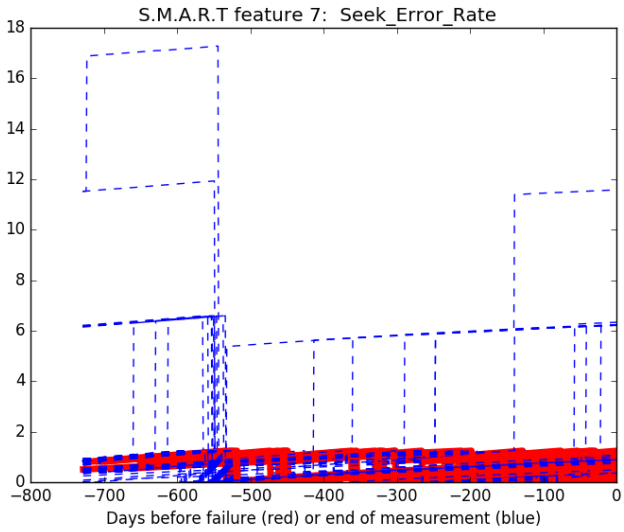

Figure 47: No indication that this has any relevance on its own.

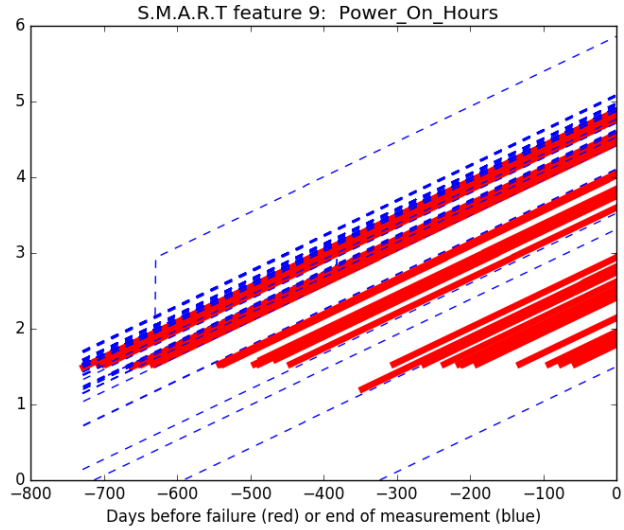

Figure 48: Not really a useful value.

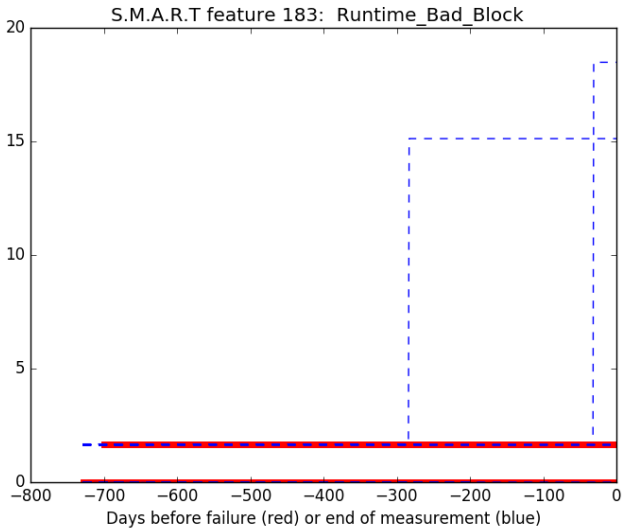

Figure 49: No indication that this has relevance on its own.

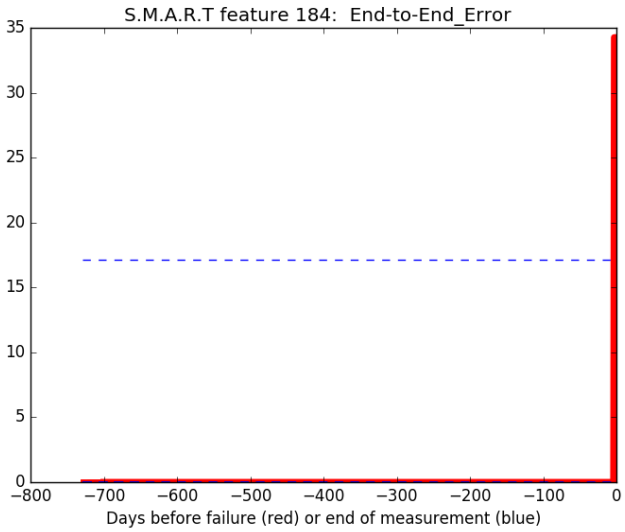

Figure 50: Clear indication that a spike means impending failure. Only one false positive with this rule (out of over 3000 drives). Is not used by storage industry [Backblaze2017b], yet. However, may be strongly correlated to SMART values that are already used.

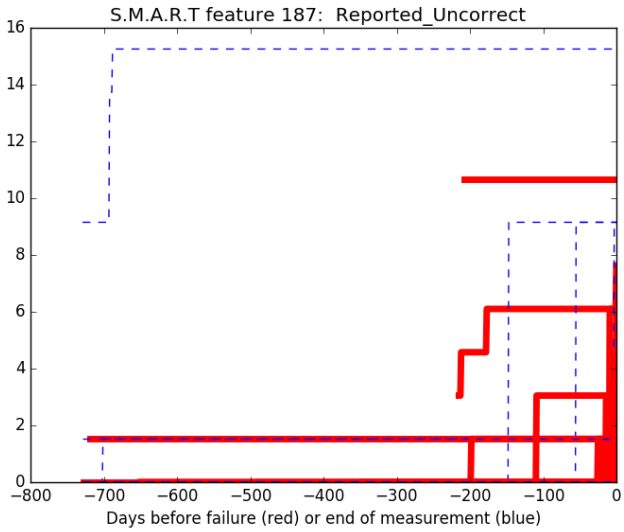

Figure 51: Clear indication that a drive will not function for long with a positive value. Only a few false positives with this rule (out of over 3000 drives). Is used by storage industry [Backblaze2017b].

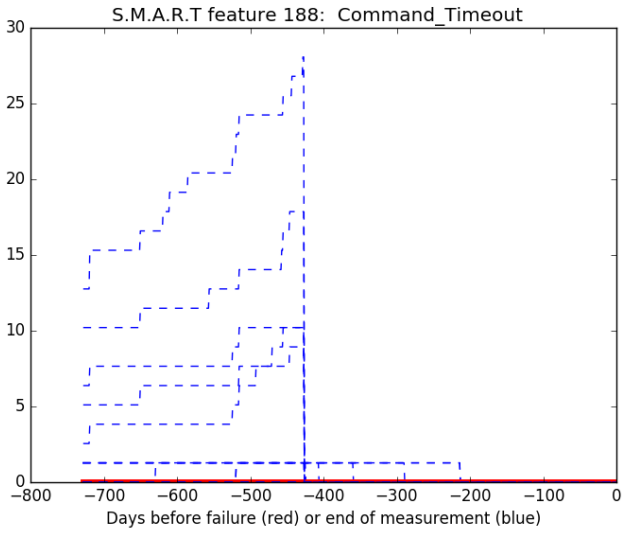

Figure 52: No indication that nonzero value means impending failure. Is however used by storage industry [Backblaze2017b]. Perhaps there is a point in making different rules for different models of hard drives?

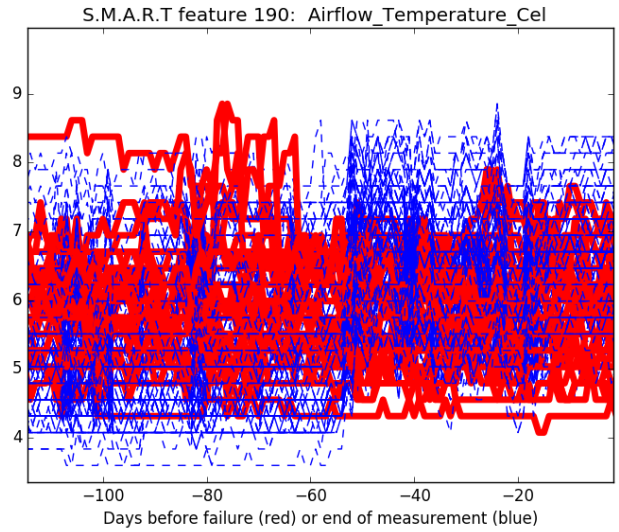

Figure 53: Does not seem to be a useful value on its own. The resolution is likely too low: just one temperature measurement per day.

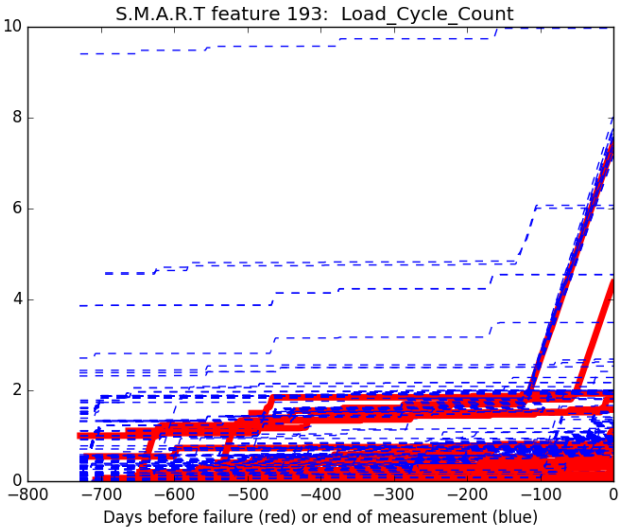

Figure 54: Looks like an interesting feature, but a rule based on this only would give a lot of false positives.

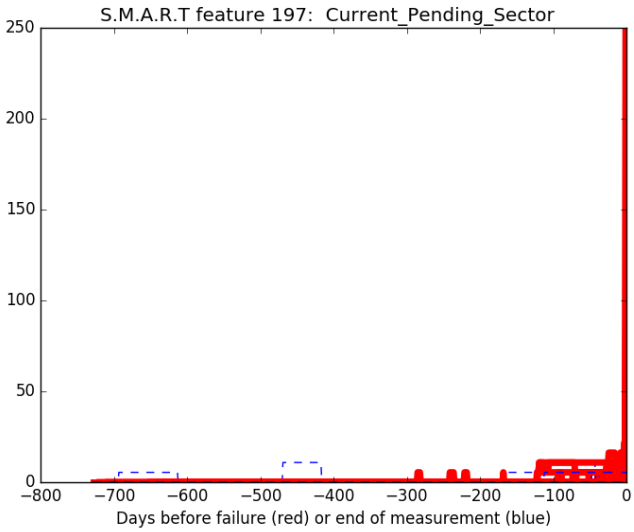

Figure 55: Clear indication that nonzero value is critical. Is used by storage industry [Backblaze2017b].

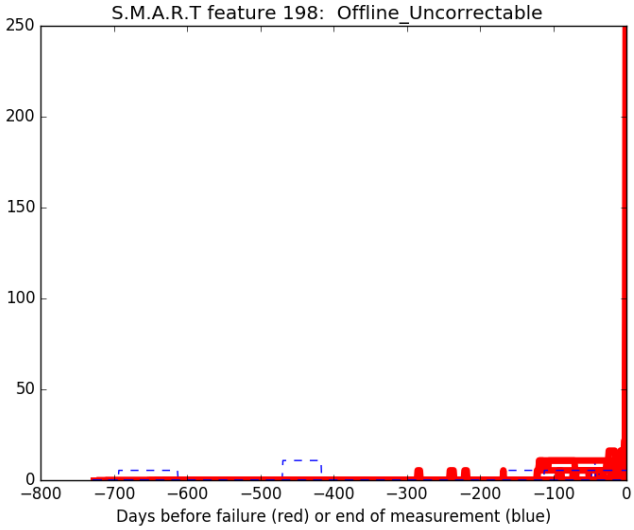

Figure 56: Clear indication that nonzero value is critical. Is used by storage industry. Is strongly correlated with SMART 197 above [Backblaze2017b].

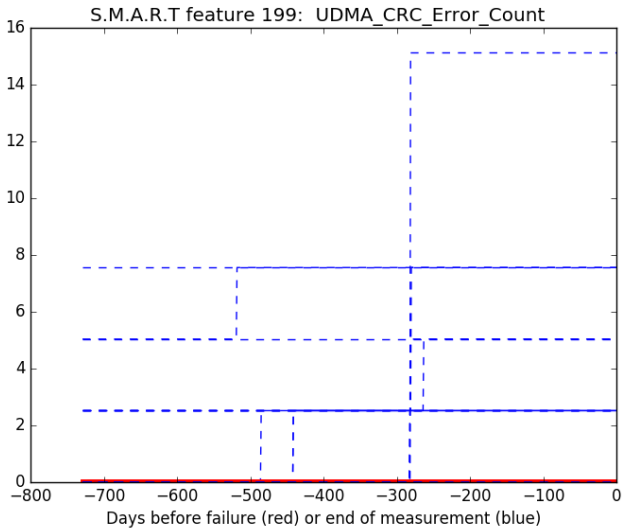

Figure 57: Does not seem to be a useful value on its own.

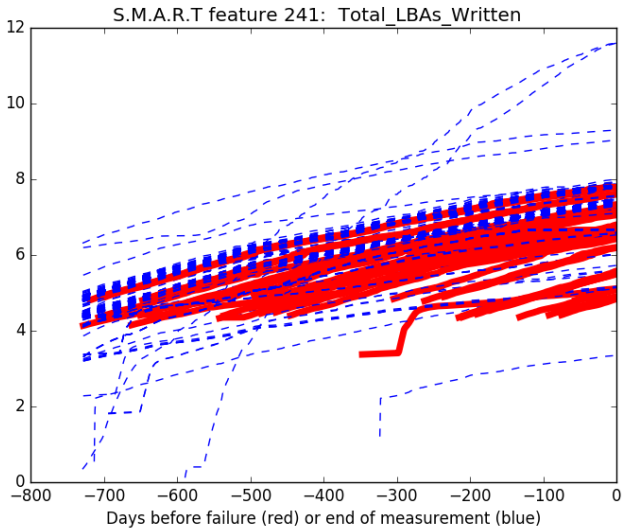

Figure 58: Interesting value, since it has some distinct reoccurring patterns. Does not seem to be relevant to predicting impending failure on its own, however.

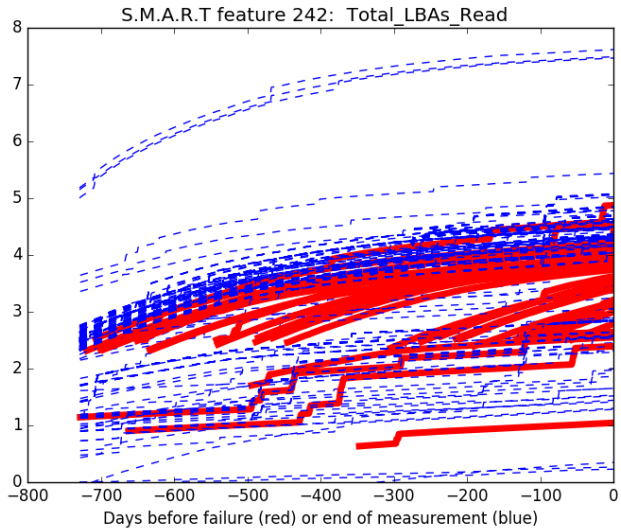

Figure 59: Just as with SMART 241 above, this is an interesting value with distinct reoccurring patterns. But does not seem to indicate impending failure.

15.2 Reduced Features for Hard Drive Data

Below are some examples of where the reduced features are noticeably different for the Component ESN and the Basic ESN. Comments on these are in section 10.2.3.

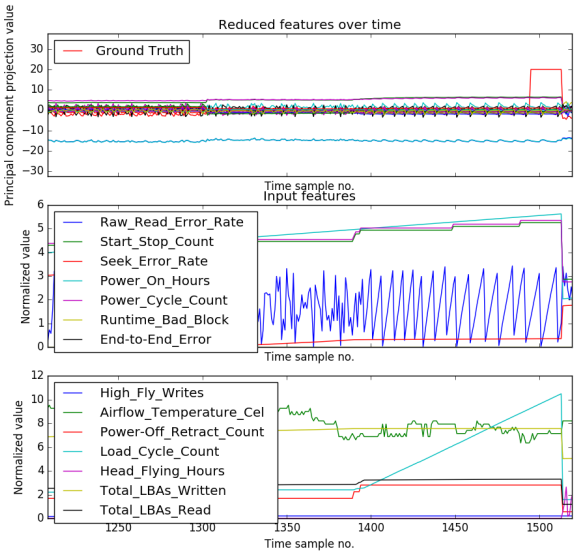

Figure 60: Reduced features for Basic ESN on unit 1.

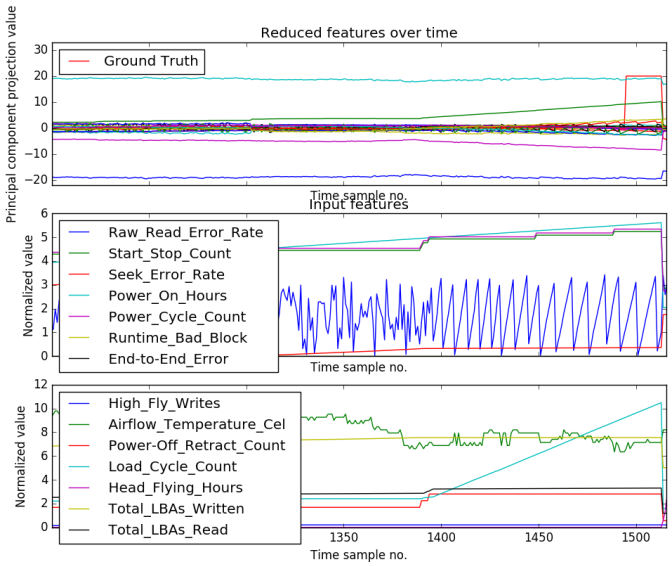

Figure 61: Reduced features for Component ESN on unit 1.

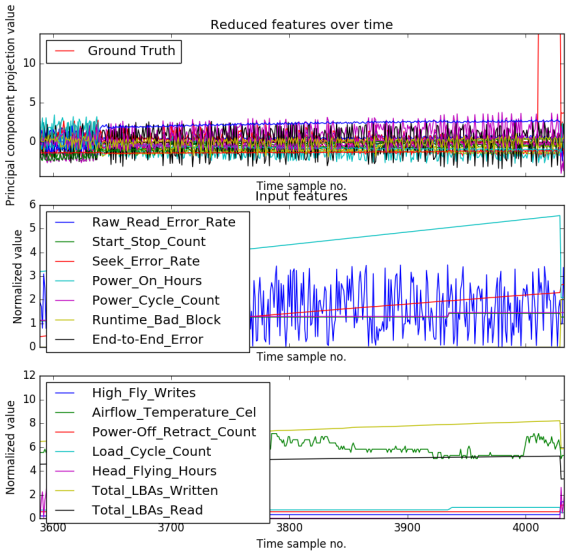

Figure 62: Reduced features for Basic ESN on unit 2.

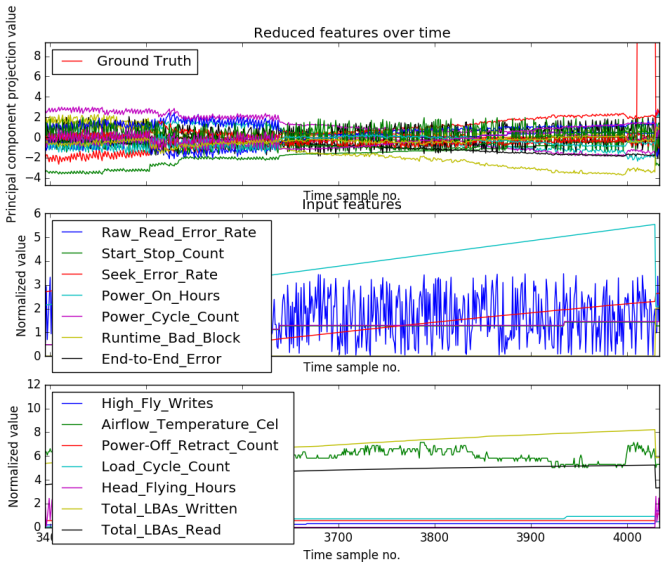

Figure 63: Reduced features for Component ESN on unit 2.

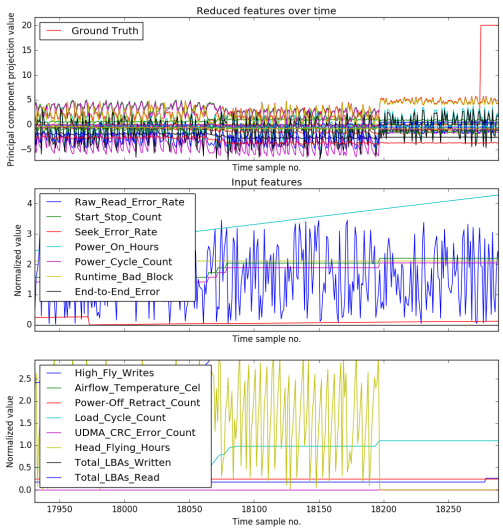

Figure 64: Reduced features for Basic ESN on unit 3.

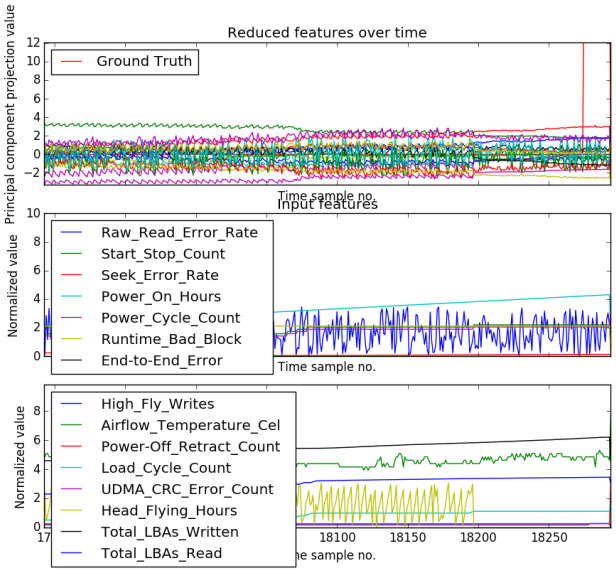

Figure 65: Reduced features for Component ESN on unit 3.

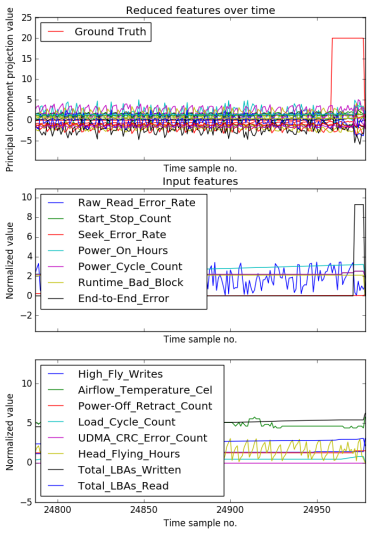

Figure 66: Reduced features for Basic ESN on unit 4.

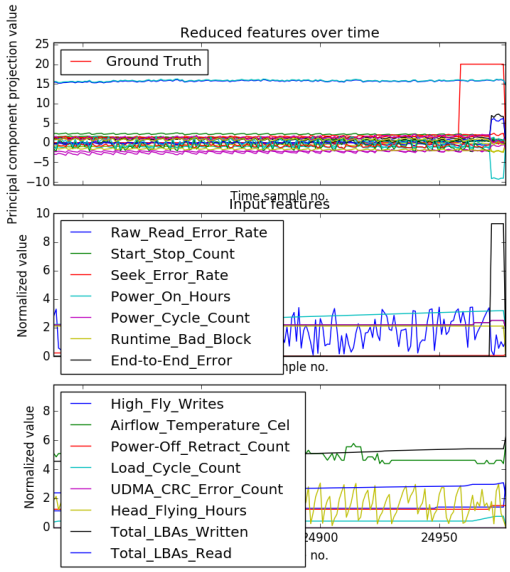

Figure 67: Reduced features for Component ESN on unit 4.