Lehrstuhl das Fach Verteilte Informations und Multimedia-Systeme

Health Status for Hard Drive Failure Detection

Masterarbeit von

Miguel Vieira Pereira

1. Prüfer	2. Prüfer
Prof. Dr. Harald Kosch	Prof. Dr. Michael Granitzer

August 21, 2025

Contents

Abstract

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The amount of information produced and processed in the world has seen a steady increase over the past decades. The world internet traffic is increasing exponentially since a few decades. Edholm's Law[1] predicts that this behaviour should continue until at least 2030. It shows that telecommunications data rates are rising in a manner analogous to the one predicted by Moore's Law[2]: doubling every 18 months. Even 20 years after this prediction was done, it is still used by organizations responsible by developing the infrastructure to transmit huge amount of data [3].

As the amount of data generated by users all around the world increases, the need to store and access this data increases accordingly. This evolution is matched by an expansion on the number of digital services offered to users: musics, videos, books and other types of files and media that can be accessed from anywhere. The solution companies found to these problems was Cloud Computing. Even though Cloud Computing is a vague term, one way to define it is "a technique where IT services are provided by massive low-cost computing units connected by IP networks"[4].

However, even though Cloud Computing is sold as a way for users to access computing resources they don't have physical access to, the hardware responsible for this computing power must be placed somewhere. This is achieved by the establishment of multiple data centers all around the world to which devices from anywhere can connect in order to get access to the desired services. Microsoft, for example, has 300 data centers worldwide to provide their services to their clients[5].

A data center is a complex installation that consists of thousands of hard drives connect by kilometers of optical fiber cables. Therefore, there are massive amounts of investment done in order to create and maintain these facilities. Microsoft is investing $80 billion

in order to improve their data centers[5]. So there is a demand for services related to the maintenance and improvement of data centers.

In this context, one of the main aspects of Cloud Computing in general is the strong virtualization and reliability of the system[4], meaning that even if some of the hardware fails the service should still be provided without interruption. So, as hardware components fail, they need to be replaced to ensure that the system keeps working for a long time.

Moreover, in order to maintain a reliable system it is not enough to only replace the hardware after it fails. In order to prevent some serious issues such as data loss without having to permanently have a copy of all the data it is necessary to predict the hardware failures.

Therefore, one of the main problems faced by data centers is the need to detect which pieces of hardware are going to fail before a fatal crash occurs to allow it to be replaced. This allows both the prevention of data loss.

Out of all the components that are part of a computer, 80% of the failures occur due to the problems on the hard drives, as indicated by Table 1.1 Therefore, there is a special focus on predicting failures on disks.

Hardware vendors are well aware of this situation and the difficulties related to managing thousands of hard drives. So, in order to allow their users to better tackle these problems, they added a monitoring system to their drives. This technology is called SMART (Self-Monitoring, Analysis, and Reporting Technology).

Using this protocol, vendors can provide to users indicators of the disk status to the user. Some of these attributes are Power-On Hours, Air Flow Temperature and Reallocated Sector Count (number of sectors that had to be copied elsewhere on the disk after a failed read or write operation in order to prevent data loss)[7].

The SMART system can also include status of different attributes. However, these are limited to threshold not exceeded or threshold exceeded for each attribute, where the threshold is a reference value set by the vendor[7].

Moreover, even when there is an indication that a threshold has been exceeded, it does not mean that the drive will suffer an unrecoverable failure. Sometimes it can be a sign that the drive will work more slowly than its specifications indicate.

Device	Proportion
HDD	81.84 %
Miscellaneous	10.20 %
Memory	3.06 %
Power	1.74 %
RAID card	1.23 %
Flash card	0.67 %
Motherboard	0.57 %
SSD	0.31 %
Fan	0.19 %
HDD backboard	0.14 %
CPU	0.04 %

Table 1.1: Data center failure percentage by component - [6]

In addition to that, these status provided by the vendors consider each attribute independently. They do not take into account the fact that some problems in the hard drive can cause different indicators to increase simultaneously, so by considering the relationship between different attributes would allow problems to be more accurately detected. For example, having a huge number of Power-On cycles when compared to Power-On hours may indicate a problem in the disk that causes it to crash and restart constantly.

Finally, this ad-hoc approach uses static values and do not consider the evolution of the attributes over time. Imagine the situation of a disk that has not reallocated any disks so far, and suddenly rate at which sectors have to be reallocated increases. By taking into account how this attribute evolves over time, this problem can be more quickly detected before the threshold value is reached.

However, even though the built-in failure detection system is not very effective, it doesn't mean that the data itself cannot be used to obtain more interesting results, after all the problem of failure detection must still be solved. Current research uses mostly machine learning approaches such as SVMs and Neural Networks to tackle this problem. Further details of how these methods work will be detailed in Chapter 2.

This project has three main objectives. The first one is to create a library that implement

some of the current approaches used in research. The idea is to also made it in such a way that it can be extended with new methods in order to allow them to be easily tested and compared to other ones.

After creating the library, further tests can be performed on existing methods to understand how they evolve as their parameters change. For example, the research of Zhu et al. on backpropagation neural networks for disk failure[8] uses a neural network with one single hidden layer with a fixed amount of 30 nodes on it. With our library, we can change the number of nodes of a neural network analogous to the one presented by them and study how the results change when the number of layer or nodes is altered.

The third objective is to extend existing methods to also make use of the concept of Health Status introduced by Xu et al.[9]. Their idea is to give a score from 1 to n to each sample depending on how close it is to a failure, instead of using only a pass/fail status.

So, the goal is to extend other methods to support more than two classes and study their reaction to this change. Different models can be extended this way, whether they are other neural network based methods such as [8] or even if they use other approaches such as the Random Forest one presented by Shen et al.[10].

1.2 Problem

Let $x_{i,t}$ be the vector in which the j^{th} component designs the measure of the j^{th} SMART attribute for disk i at time t. As input, we also have a value y_i that is either 0, if this disk has eventually failed or 1 if, until the end of the observation period, the disk kept working properly. Also, let the number of different smart attributes be equal to m.

Then given a series of vectors with the SMART attributes for an unseen disk, $\hat{x}_t$, we want to output a value $\hat{y}_t$. Here, $\hat{y}_t$ should be equal to 1 if the algorithm predicts that the samples $(\hat{x}_1 \dots \hat{x}_t)$ are closer to the samples in the input that did not fail than to the ones that malfunctioned, and 0 otherwise.

In a real world scenario, if the value $\hat{y}_t$ is equal to 1, then a warning should be sent to the people responsible for maintaining the data center in order to replace the concerned disk.

This corresponds to a classical classification problem. The numerical vectors as input and a binary output suggest that machine learning methods such as Classification Trees[11], Support Vector Machines[8] and Neural Networks[9] are well adapted to tackle this problem.

However, this problem has a few specific characteristics. First of all, the samples are not completely independent, since the values $(x_{i,1}, x_{i,t})$ are a time series that describe the status of disk i through time. As we will see on Chapter 3, this can be a motivation to use methods such as LSTM (Long Short-Term Memory) networks that can encode and work with time dependencies.

Moreover, even other methods such as Random Forests that can't easily deal with time dependencies can be extended to make use of the time series aspect of the data. This can be one by appending new components to $x_{i,t}$ corresponding to the value $x_{i,t}[j] - x_{i,t-\delta}[j], j \in \{1, \ldots, m\}$. In this case, δ is a constant that allows the algorithm to detect sudden changes in the value of an attribute[10].

A second point that needs to be mentioned is that the amount of disks in each class is not similar at all. Since a hard drive has a service life of around 3 to 5 years[12], and observations are performed for a few weeks, no failure will be observed for most of the disks. The ratio of failing disks over the total amount is between 0.4% and 1.9%[9]. So, any approach to this problem has to handle this imbalance in the input data.

Thirdly, the meaning of each SMART attribute is not the same for different vendors[7]. The protocol is standardized, but what it reports is not. So, an algorithm for failure prediction cannot be trained on data from different vendor disks. Moreover, in general, it is not a good idea to mix data from different disks on the same dataset, since they can have different failure causes profiles.

Most of the time, this does not pose a problem to the datasets generated by the data centers. This is due to the fact that, in practice, a data center uses hundreds of copies of the hardware of the same model and has at most a couple of different models. This allows for the replacement if failing disks do be done in a cheaper and more swiftly manner, since it simplifies stock management.

When evaluating an algorithm there are two main metrics that must be taken into account. The first one is the FDR (Failure Detection Rate). This is equal to the ratio

between the number of disks that are correctly predicted to be failing and the total amount of failing disks. Its value should be as close to 1 as possible

The second metric is the FAR (False Alarm Rate). This is equal to the ratio between the number of disks that are wrongly predicted to be failing and the total amount of disks that do not fail. Its value should be as close to 0 as possible.

The challenge is to find algorithms that find an equilibrium between this values. If it predicts every $\hat{x}_t$ to give an output $\hat{y} = 1$, the FAR will be equal to 0, but the FDR too. On the other hand, if it predicts every $\hat{x}_t$ to give an output $\hat{y} = 0$, the FDR will be equal to 1, but the FDR too. So, we notice that a trade off must be done between the FDR and the FAR.

It does not mean, however, that both metrics should be treated identically. Suppose that during a period of one month, 2% of the hard drives in a data center fail. This corresponds to a lifetime of about 4 years, which may represent a real world scenario. If the FDR decreases by 1%, then only 0.02% of the disks of the data center will fail without a warning during one month.

In contrast, if the FAR increases by the same amount, then 0.98% of the disks of the whole data center will be unnecessarily replaced during the same period. This is close to 50% of the disks that actually need to be replaced and can incur a substantial cost. So, most algorithms in the literature prefer to have a bias towards classifying a disk as working rather than failing.

Aditionnaly, it is impossible to reach perfect values for both indicators. This is due to the fact that hard drives are subject to real world conditions. For example, a short circuit on a disk may be caused by an electrical surge, but it does not depend on the internal state of the disk and thus cannot be predicted by the approach using SMART attributes.

Some additional steps can be made after executing the machine learning algorithm. A common one is the use a voting algorithm. The simplest way this can be done is as follows[10]: each sample $\hat{x}_t, t \in \{1, \ldots, T\}$, often including the change rate components, is evaluated independently. A sequence $\hat{y}_t, \{1, \ldots, T\}$ is obtained. Then, a fraction f is chosen. If more than $f \cdot T$ of $\hat{y}_t$ is equal to 0 then the output of the system is 0, otherwise it is 1.

The main advantage of this method is that it allows to easily control the trade off between the FDR and the FAR which can be desired as we have seen above. It suffices to increase f in order to decrease the FAR at a cost of also decreasing the FDR.

2 Background

Given the way the problem can be formulated according to Section 1.2, the failure detection problem can be viewed as a classification problem.

In this case the SMART attributes are the input variables. We have two classes: **good**, which corresponds to the disks that did not fail during the period that they were observed, and **bad**, which are the ones that did fail.

Therefore, machine learning approaches are the most appropriate to tackle the problem. In the literature, some of the machne learning methods used are Decision Trees, Random Forests, Recurrent Neural Network and Long Short-Term Memroy Networks.

In the next few sections we will discuss how these approaches were implemented and the results that they obtained.

2.1 Decision Tree

Decision Trees represent a flowchart in the form of if-else statements. Because of this, one of their main advantages is the fact that it is fairly easy to interpret a decision tree and to understand how it works and how it arrives at its conclusions.

Formally, a Decision Tree is a rooted tree $G = (V, E), E \subseteq V^2$. The leaves of the tree are labeled with one of the possible results of the decision problem. In our case, it will be either **good** or **bad**.

The other nodes are called decision nodes. When evaluating an unseen sample, consists in traversing the tree starting from the root and taking a left or right child of the current node depending on the conditions in it until a leaf is reached, which corresponds to an output.

For the following discussion, it may be useful to have at least a superficial understanding of the training process of a Decision Tree. The tree starts with a single node, the root, and all the training samples are placed there. The root is then added to a queue.

While the queue is not empty, the first node in it is processed. The processing step starts by choosing a splitting value c for each attribute.

Then, for each attribute i, the samples are divided in two sets depending on wheter its value is bigger or smaller than a threshold value c_i. One way to obtain the threshold value is to compute the average of the attribute i over the samples assigned to the node being processed.

Then the algorithm computes which of the partitions maximize a certain criterion function. A common criterion to use here is the information gain, which is the opposite of the Shannon Entropy [13].

Then, two new nodes and edges are created. The original node is then labeled by (i, c_i) where i is the index of the attribute that maximizes the criterion function. The training samples that were assigned to the node are they split between its two children according to the value of their attribute i.

If some condition for one of the children is met such as a certain depth or all samples are in the same node, then the is not added to the queue. Instead, it is kept as a leaf with a label that corresponds to the class that appears most frequently in the samples assigned to it.

Otherwise the node is added to the queue to be processed later.

After being trained, when the tree needs to predict the class of a certain sample x it will start traversing the tree from its root. Then, at each non-leaf node, it will read the label (i, c) and if x_i is smaller than c then the next node in the traversal is the left child, else it is the right child.

The training and the evaluation processes are illustrated by **Algorithm 1** and **Algorithm 2** respectively.

Algorithm 1: TrainDecisionTree(T: train set)

```
1  r ← Node(), q ← Queue();
2  r.samples ← T;
3  q.push(r);
4  while not q.empty() do
5      n ← q.pop(), maxi ← ∞, att ← −1;
6      c ← −1;
7      for i ← 1 to m do
8          c′ ← avg(n.samples, i);
9          s1 ← {x ∈ n.samples | x_i ≤ c′};
10         s2 ← {x ∈ n.samples | x_i > c′};
11         gain = criterion(s1, s2);
12         if gain > maxi then
13             mini ← gain;
14             att ← i;
15             c ← c′;
16             n.left.samples ← s1, n.right.samples ← s2;
17         end
18     end
19     n.attribute ← att;
20     n.threshold ← c;
21     for child of n do
22         if shouldProcess(child) then
23             q.push(child);
24         end
25         else
26             child.isLeaf ← true;
27             child.result = mostCommon(child.samples);
28         end
29     end
30 end
```

Algorithm 2: EvaluateDecisionTree(*tree*: Decision Tree, x: sample)

```
cur ← tree.root;
while not cur.isLeaf do
    if x[cur.attribute] ≤ cur.threshold then
        cur ← cur.left;
    end
    else
        cur ← cur.right;
    end
end
return cur.result;
```

The explicability of this model comes from the fact that when a sample is evaluated, the decision steps that resulted in the corresponding output can be followed, as show in **Algorithm 2**. So, it is possible to verify which parameters are taken into account and to predict what would happen if one of then was changed. Therefore the impact of each attribute can be interpreted.

Decision Trees come in two flavors: Classification and Regression Trees. The former corresponds to classifying samples in discrete, independent classes. So, Classification Trees will treat **good** and **bad** samples as different entities.

Regression Trees are concerned in predicting a continuous value. So, they allow us to use the concept of health status.

Suppose we try to give an integer score from 1 to m to each drive. The ones that do not fail correspond to a value of m while the samples of the failing drive are given values from 1 to $m - 1$ depending on how close they are to the moment of the breakdown. Then a Regression Tree will not try to simply predict an integer from 1 to m, intead it will try to find an exact value to it.

We can explain the difference between the two approaches when using multiple classes as follows: imagine we have a training sample whose desired output value is 1. Then, for the Classification Tree the error is the same when it puts it in the class 2 and in the class m.

On the other hand, a Regression Tree uses a mean squared error funcion, so even if it doesn't put the sample in the class 1 it will consider it worse the cases in which it is put

in class m compared to when it is put in class 2. It may even predict a value of 1.5 to the sample, even though there is no sample in the training set with this value for the dependent variable.

Compare this to the task of classifying a shape as a square, a triangle or a circle. Then, unless there is some additional issue specific to the context of the application, misclassifying a circle as a square is not worse than classifying it as a triangle . This fundamentally differs from the task of predicting a health status in which it is better to predict a 1.5 to a sample whose expected output is 1 than to predict a 5 for example.

Decision Trees were used by Li et al. [11] in order to obtain an algorithm capable of predicting hard drive failure. They tested both Classification and Regression trees.

In their approach, they were able to obtain an FDR of more than 95% while keeping the FAR below 1%.

Apart from the result in itself this work introduce some additional concepts that are useful for the implementation of other methods and is actually used by later researchers. First of all, they add a feature selection step that keeps only a subset of the SMART features passed as input.

More interesting though is the fact that they add columns to their table. This corresponds to the variation of the value of certain features over an interval. It allows the algorithm to take into account the fact that the training samples for a specific hard drive represent a time series.

More precisely, they choose a value for T. Let the samples for the i^{th} hard drive be the list x_i in which each entry corresponds to a sample and they are ordered in the order they were taken. Let $x_{i,j}[t]$ be the j^{th} feature of the sample taken at instant t. Then the value of the new column for the vector $x_i[t]$ is:

$$x_{i,n+j}[t] = x_{i,j}[t] - x_{i,j}[t-T], j \in \{0, \ldots, n-1\}$$

Where n is the original number of features in the vector.

This is an elegant way to include the time dependence aspect of the problem. The main advantage is that it can be applied to any method to try to improve methods that are not designed to work with time series.

In their work, they also use a voting algorithm. This way, in order to classify an unseen hard drive, they take the last N samples and evaluate each of them using the Decision Tree. If more than $\frac{N}{2}$ of them are put in the **bad** class, then the hard drive is classified as failing.

However, their research on Decision Trees does not implement some other techniques used by other research projects and that could be included. For instance, they do not test different thresholds for the voting algorithm. They always use a ratio of 0.5.

In addition to that, they always train the model with a constant ratio between good and bad disks in the training set. For each 3 **bad** ones, they include 7 **good** ones. They do not study what happens if, for example, for each **bad** HD there are 10 **good** ones, which more closely represents the real world scenario if no data is filtered.

In the end, the results they obtained are promising. They obtained FARs below 0.5% while keeping the FDR around 95% for the Classification Tress.

The Regression Tree model used a continuous value between -1 and 0 as the health status value, linearly dependent on how much time before the breakdown of the drive is. This approach was able to increase the FDR by 1% while also decreasing de FAR by around 0.2% when compared to the Classification Tree implemented by them.

2.2 Backpropagation Neural Network

A Neural Network (NN) is a Machine Learning model that can be used to perform both classification and regression tasks. Its name comes from the fact that its development was inspired by how a brain work: with neurons that can be interpreted as nodes and synapses that connect them. The first implementation of this approach is credited to Rosenblatt with his Perceptron architecture [14].

In general, this architecture can be represented as a sequence of layer of nodes (the neurons) and a set of directed edges from every node in layer l to every node in layer $l+1$. Moreover, each node has an output value o_j and each edge (u, v) has a weight w_{uv}.

The first and last layers are special. The output values of the nodes of the first one correspond to the input values to the network. The values in these nodes represent the

sample being evaluated. The last layer, in contrast, represents the output values of the network.

From the values on the input layer we can compute the output values of every node in the network. In order to do that, we need to introduce the concept of activation function φ.

The activation function is a non-linear function that will map the input given to a node to its output. The input to a node v is the sum of the ouput values of u such that (u, v) is in the network times the weight of the edge (u, v). Formally:

$$o_v = \varphi\left(\sum_{u=1}^{n} w_{uv} o_u\right)$$

Where w_{uv} is taken to be 0 when the edge (u, v) does not belong to the graph.

Therefore, from an input to the first layer, we can compute the values of the outputs of the second layer. This process can be repeated until the output value for the last layer is calculated.

The fact that the activation function takes as input a linear combination of the ouputs of the nodes of the previous layer explains why it shouldn't be linear. If φ were linear, then the ouput of each neuron would be a linear with respect to the ones of the last layer. So, the output of the last layer would be a linear function of the inputs.

Having a non-linear activation function is desired, therefore, not because otherwise the math would not work. Instead it is due to the fact that there are other, simpler methods to learn linear patterns such as SVM or simply the minimum squares method. The power of a neural network is exactly that it is able to recognize non-linear patterns, which can only be done when using non-linear activation functions.

Here we note that the activation function does not need to be the same on every layer, but it does not change the analysis we are performing here. Some of the most commonly used activation functions are the ReLU $ReLU(x) = \max(0, x)$ and the logistic function $\phi(x) = \dfrac{1}{1 + e^{-z}}$.

A classic problem that can illustrate how useful a non-linear function can be is the binary classification of samples in two classes 0 and 1. Suppose that on the last layer a function

that maps $(-\infty, \infty)$ to $[0, 1]$, such as the logistic function, is used. Then, the output of the network will be a real number between 0 and 1.

A classic interpretation of this value is the probability p of the sample belonging to class 1. Trivially, the probability of the sample belonging to class 0 is $1 - p$. So, the model is able to not only classify the sample, but also to indicate how confident it is that this is the correct result.

The output value of every neuron, including the output ones, can be computed from the input values and the weights. So, in order to fully describe a network, the only attributes that we need are the weights. The challenge is, therefore, how to determine the weights that are appropriate for the problem at hand.

The method used by the Backpropagation Neural Networks to compute the weights of the network is called backpropagation. It is inspired by the Gradient Descent approach first developed by Cauchy [15] long before the advent of computers.

The intuition behind the approach proposed by Cauchy is as follows: imagine there is a person on a region that is full of hills. When the person is at point (x, y), the height relative to the sea level is $h(x, y)$. Now suppose that the person is trying to find the lowest point in the plane.

This can be easily mapped to the problem of trying to minimize a function. Moreover, even though our discussion uses as example a function with two independent variables, it can be extended for functions with more variables without loss of generality.

The gradient descent technique uses the fact that the gradient of a function $\nabla h(x, y)$ always points in the direction in which the incline is non-negative and the steepest at (x, y). Therefore, $-\nabla h(x, y)$ is the direction in which the value of h decreases as fast as possible at (x, y). So, the idea is to take a small step in that direction since the change of height when traveling in that direction in the neighborhood of (x, y) is non-negative.

From the above description it is possible to see that this method tends to arrive at a local minimum. The most notable exception is when the function has some discontinuities or some points in which its values decreases very rapidly. In this case, if the "length" of the step (called learning rate) is too big, it may happen that we jump over the region that would yield a smaller value.

The solution can be to decrease the learning rate, but this causes the algorithm to take longer to converge. So, there is a tradeoff between the time required to train the model and its capacity to find the best values.

In the context of neural networks, the function that the model tries to minimize is the loss function over the training samples and its parameters are the weights of the connections of the network.

It can be proven [16] that at each step, each weight of the network should be update by:

$$\Delta w_{ij}^{l} = -\eta e_i^{(l)} o_j^{(l-1)}$$

Where η is the learning rate $e_i^{(l)}$ is called the error signal of the i^{th} node at layer l. The error signal can be computed recursively as follows:

$$\begin{cases} e_i^{(k)} & = \varphi'(u_i^{(k)}) \\ e_i^{(l)} & = \varphi'(u_i^{(l)}) \sum_j w_{ij}^{(l)} e_j^{(l+1)}, l \in \{1, \ldots, k-1\} \end{cases} \tag{2.1}$$

One notable aspect of Equation 2.1 is that the error signal of neurons on layer l depend only on the error signals on layer $l+1$. So, the errors for each node of a given layer can be computed in parallel.

This property is so important that the capacity of computing such values in parallel made it possible to introduce neural network training in GPUs [17] which was one of the main causes of the AI spring that has been happening on the last 20 years.

Zhu et al. [8] applied the backpropagation neural network approach presented above to the problem of hard drive failure detection. The lowest FAR they obtained was 0.48% while keeping an FDR of 94.6%. They were even able to obtain a model that had and FDR of 100% on their test set, but with a relatively higher FAR of 2.26%.

However, their work has some limitations. Specially due to the fact that they only varied one parameter. The only variable they changed on their experiment was the number of samples per failing hard drive to be used in the training and test processes. So, if the time window was 12 hours, for example, only the samples taken at most 12 hours before a hard drive failed were included in the **bad** class.

No results were presented about what happens with a bigger or smaller network or with more or less input parameters, for instance.

This highlights the importance of the current work that intends to test how different parameters influence the results for a variety of models. Moreover, the objective of creating a piece of software that can be easily configured to train a model with different parameters will allow the study of the impact of different parameters easier, even when the results are not readily available.

2.3 Recurrent Neural Network

A Backpropagation neural network is a relatively simple yet powerful model capable of solving multiple regressions and classification problems. However, due to its simplicity it can have some limitations.

One of the most notable ones is how it evaluates one sample at a time. It does not care about previous samples and doesn't have any type of memory.

However, in many problems the concept of time or, more generally, of sequences is present. Due to how they work, backpropagation neural networks are not capable of efficiently working with sequences.

This limitation was found early on and there are works from the 1980s already trying to figure out how a neural network could store a state depending on what the previous samples were [18].

However, long before computer science tried to tackle the problem of modeling sequences, neuroscience had already tried to find a method that allowed the brain to do so. The question that they were trying to answer was how does memory in the human brain. More specifically, how does neurons and synapses allow the brain to react to stimuli it received in the past.

The first clues to an explanation were found at the end of the 19^{th} century, when researchers were suggesting that the brain had "recurrent semicircles" [19] implying that the signal of a neuron at a certain time t could influence its value at a time $t' > t$.

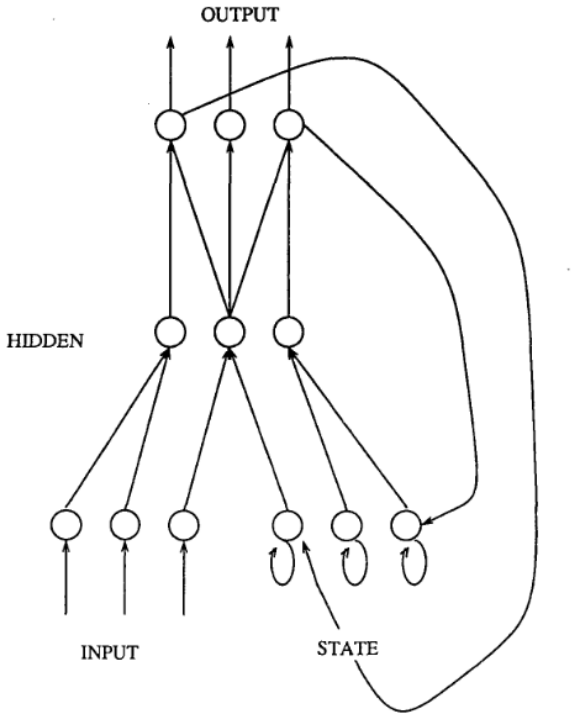

Figure 2.1: Schema of a Jordan Network - [21]

As a consequence, later on, one of the first mathematical models of a neuron that was developed to try and explain brain activity was the MCP [20], which allowed loops between neurons in the network. In their paper presenting the MCP, McCulloch and Pitts were able to show that their architecture allowed the activity of the network to be influenced by activity indefinitely far in the past [20].

And, as was common at the advent of the development of neural networks, researchers used the model of the brain as an inspiration. This culminated on the work of Jordan [18] and Elman [21]. They proposed a model that is nowadays labeled as Simple Recurrent Neural Networks (SRNN).

In an SRNN, the first layer of the network doesn't have only input nodes, it also has state nodes. These state nodes are connected to the output of the network. Figure 2.1 shows a schema of how such a netwrok is connected.

So, when evaluating a sample, the input layer receives not exclusively the values of the independent variables of the sample, but also some values that depend on its state at previous times.

The ability to modify the output depending on the previous sample evaluated is what allows the to have some type of memory. Models that are able to take don't evaluate each sample independently but are instead able to take into account a context, are useful for applications such as Natural Language Processing (NLP).

This is due to the fact that when trying to predict the next word in a sentence for example, it is important to take into account evey previous word [22]. A backpropagation neural network, due to its nature, would only be able to take into account a fixed number of words.

Nowadays, other methods beyond SRNNs are used to do this task. Most notably, in the last few years, Transformers[23] became very popular. Nevertheless, models that are able to take into account a sequence of samples are still very much needed.

In the context of hard drive failure prediction, our samples are not words, but they do consist on a time series and thus form a well defined sequence. This indicates that SRNNs may be a good model to tackle the problem.

Indeed, Xu et al [9] made use of Recurrent Neural Networks to the problem at hand. Their setup for the neural network is quite standard, up until the point in which they introduce the concept of health status.

So, differently from every other approach in the litterature they do not divide the samples on good or failing. They actually use a more general method in which samples are divided in N classes.

The ones from hard drives that do not fail during the observation period are given the label $N-1$. The samples from failing disks are divided in the other $N-1$ classes, labeled from 0 to $N-2$. The algorithm they use to assign the failing samples divides them depending on how close they are to the failure by using fixed-length intervals.

We can formally define the approach that they explained in natural language in their paper as follows: suppose that the last n samples from each failing disk are kept and need to be divided in $N-1$ bins numbered from 1 to $N-1$. Let the last sample from a specific hard drive before it fails be taken at time T_i. Then, supposing that a sample is taken every 1 unit of time (usually once per hour) the class $c_i(t)$ of the sample of the disk i at time t is given by:

$$c_i(t) = \begin{cases} N-1, \text{ if } i \in \textbf{good} \\ \left\lfloor (T_i - t)\dfrac{(N-1)}{n} \right\rfloor, 0 \leq T_i - t < n, \text{ if } i \in \textbf{bad} \end{cases} \tag{2.2}$$

Here, $\lfloor x \rfloor$ is the floor of x, meaning the largest integer smaller than or equal to x. For example, $\lfloor 1.8 \rfloor = 1$, $\lfloor 3 \rfloor = 3$.

By analizing this formula, we see that the $\frac{n}{N}$ samples closest to the moment of failure are assigned to class 1, the next $\frac{n}{N}$ are assigned to class 2 and so on.

We can also prove that the formula indeed does what is desired, that is, it assigns a number from 0 to $N-2$ to each sample. In order to do so, we notice that since we have n samples taken every 1 unit of time, and the last one is taken at time T_i, the first one is taken at $T_i - n + 1$.

We can compute $c_i(T_i) = 0$ and $c_i(T_i - n + 1) = \left\lfloor (N-1)\left(1 - \frac{1}{n}\right) \right\rfloor + 1$. Since $\left(1 - \frac{1}{n}\right) < 1$, $\left\lfloor (N-1)\left(1 - \frac{1}{n}\right) \right\rfloor < N - 2$. Moreover, $c_i(t)$ is clearly monotonic, since it is a composition of the floor function and elementary mathematical operations.

So, we can guarantee that the formula given above for $c_i(t)$ correctly assigns each sample to a class from 0 to $N-2$ in which the closer the sample is to the moment the drive fails, the smaller is the value of its class label.

Here it is important to stress that the implementation of health status on the study by Xu et al [9] is different from the one by Li et al. [11] that used this concept to train their decision trees. The main distinction is that the later used continuous values in order to train a regression tree. The former, on the other hand, divides the samples in discrete classes.

So, when training the recurrent neural network model, the N classes are completely independent. Concretely, it does not use the fact that when the correct output for a certain sample is 1, it is better to classify it as belonging to class 2 than to class N.

Even though the health status is a useful abstraction, in the end there is a need to output a prediction of whether the sample belongs to the **good** or **bad** hard drive. In other words, the model needs to state if the hard drive is going to fail soon or not which is the actual problem we are trying to solve.

In order to do that, when they want to predict the status of a certain drive, they take its last N samples. Then, in order from the oldest to the most recent they put each sample as input to the network. The order in which they are evaluated is important, since the model being used is an RNN and therefore has memory.

For each sample, they classify it as belonging to the class whose associated output neuron value is maximum. By doing this, it is obtained a sequence of integers $c = (c_1, \ldots, c_n), 1 \leq c_i \leq N$.

For each $j \in \{1, \ldots, N\}$, let C_j be the cardinality of j in c. It is clear that $\sum_{j=1}^{N} C_i = n$. Then, they define two algorithms, the Voting Algorithm which Tends to Health (VAT2H) and the Voting Algorithm which Tends to Failure (VAT2F):

$$\text{VAT2H}(C) = \begin{cases} \text{Healthy, if } \sum_{j=1}^{N-2} C_j \leq C_N \\ \text{Failure, otherwise} \end{cases} \tag{2.3}$$

$$\text{VAT2F}(C) = \begin{cases} \text{Healthy, if } \sum_{j=1}^{N-2} C_j < C_N \\ \text{Failure, otherwise} \end{cases} \tag{2.4}$$

Notice that the contribution of the class $N - 1$ is ignored in both algorithms. This implies that these algorithms only work for when $N \geq 3$.

So, in order to tackle the problem they start by expanding the number of classes from 2 to N using equation 2.2. Then, after the network classifies a sequence of samples of the same disk in the classes from 1 to N they use equation 2.3 or 2.4 to obtain the final binary classification output.

They perform their tests on three different datasets corresponding to different models of hard drives. The neural network for each model is trained independently since, as discussed, not only the behavior of hard drives of different models can be different, but also, for different vendors, the values of the SMART attributes are not guaranteed to be compatible.

They obtained promising results of their approach. The FDR of their models was around 97% while the FAR was almost always kept below 0.1%. As was expected, the VAT2H algorithm resulted in a lower FDR and a lower FAR than the VAT2F algorithm.

Nevertheless, their study admitted to having some limitation such as not studying the impact of other health status algorithms different from the one in equation 2.2. Moreover, they used a fixed amount of classes with $N = 6$ all over their research as well as a constant number of nodes in the hidden layer of their network.

2.4 Long Short-Term Memory Network

Time has proved that RNNs are an effective method to tackle multiple sequence-based problems such as NLP [22] and video processing [24].

However, RNNs suffer from a limtation called the vanishing gradient problem. This limits their efficiency when they need to learn long term dependencies.

This has been proven theoretically by Hochreiter [25] as well as by Bengio using different approaches [26]. The intuition behind this result can be explained using the concept error signal defined in Equation 2.1.

If we expand the bottom expression to explicitly write $e_i^{(l)}$ in terms of the error signals of layer $l + 2$ instead of $l + 1$, we obtain:

$$e_i^{(l)} = \varphi'(u_i^{(l)}) \sum_j w_{ij}^{(l)} \varphi'(u_j^{l+1}) \sum_k w_{jk}^{(l+1)} e_k^{l+2}$$

From this, we can see that $\frac{e_i^{(l)}}{e_i^{(l+2)}} \propto \varphi'(u_i^{(l)})\varphi'(u_j^{l+1})$. So, it is proportional to the product of two derivatives. We can exend this process to show that the impact induced by layer $l + m$ on layer l must be scaled by the product of m derivatives.

But, if we redraw our RNN to be represented by Figure , we see that the input of sample t can be interpreted as being connected to the imput of sample t' by $(t' - t + 1)k$ layers. So the impact of the sample t on sample t' is proportional to $\prod_{i=1} (t' - t + 1)k\varphi'(x_i)$.

But, if the values of $\varphi'(x_i)$ are all smaller than 1, not only the scaling term will approach 0, but it will do so exponentially fast with respect to the distance between the two samples.

What Bengio showed on his paper [26] is that, as a network learns, these derivatives decrease and become smaller than 1. From this result, he was able to prove that as the distance between two samples increase, the impact of the earlier one on the other on the RNN tends to zero, as long as the network has enough time to learn.

The only assumption he made was that the model was not sensitive to noisy perturbations. This is equivalent to stating that if two data sets are similar, then the model is able to learn approximately the same pattern, which is coherent to the problem at hand.

So, there is a theoretical demonstration that for long enough sequences, the RNN architecture will be subjected to the vanishing gradient problem. However, theory alone is not enough to define what is a long enough sequence. In order to state that the vanishing gradient problem has an impact on RNNs it is needed to observe such phenomenon in real-world scenarios.

And, indeed, experiments show that problems as diverse as sentiment analysis ([27]) and greenhouse gas predictions ([28]) can be better learned by networks that try to solve the vanishing gradient problem such as LSTMs.

Other research projects show that applying techniques explicitly developed to curb the vanishing gradient problem can result in better results even without needing to change the architecture at all. For instance, in [29] they adapted the activation functions that increase the value of their derivatives.

With this evidence, both theoretical and experimental, it is clear the need to develop networks that are able to better handle or completely avoid the vanishing gradient problem.

So, after proving the vanishing grdient problem in RNNs, Hochreiter developed a network that was not subjected to the same problem. His solution was the Long Short-Term Memory architecture (LSTM). A schematic view of such network is presented on Figure 2.2.

Simillarly to the RNN architecture, we combine the output of the previous sample (represented by $h^{(n)}$ on the image) with the input data for the current sample ($\chi^{(n+1)}$). However, the main innovation is the presence of the cell state, the channel represented by the horizontal lane in the bottom half of the image and dented by $c^{(n)}$.

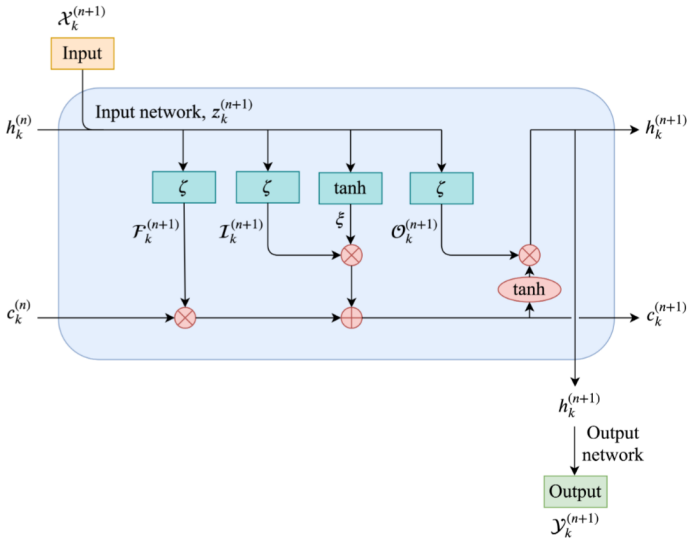

Figure 2.2: Schema of an LSTM - [30]

It is not updated using the same logic of the neurons used by BPNNs and RNNs so it is not subject to the vanishing gradient problem. The operations performed by an LSTM can be explained in three steps, each one called a gate.

The first one is called the forget gate and is represented in the image by the leftmost ζ node. It takes the input of the current sample $(\chi^{(n+1)})$ and the last output $(h^{(n)})$ of the network and generates a forget factor vector $P^{(n+1)}$ which is then multiplied component by component with the current value of the state $c^{(n)}$.

The name from this gate comes from the fact that it controls how much the network remembers from the past samples. In the limit case in which $P^{(n+1)} = \mathbf{0}$, the value of c is reset and no information is kept from previous interations.

The next step is the input gate, which in the image corresponds to the central ζ and tanh nodes. If the forget gate decides how much from previous iterations should be kept in the cell state, the input gate controls which data from the current one should be added to the cell state.

The input gate is divided in two steps: the first is to generate the candidate values ξ and the second is to compute the weights $I^{(n+1)}$. Then each component of the candidate values vector is multiplied by its corresponding weight and added to the cell state.

Finally, there is the output gate that will actually compute the predicted ouput value for the sample. In the image it is represented by the rightmost ζ and tanh gates.

This is the step that uses the cell state as input. It combines the cell state value as well as the sample input $x^{(n+1)}$ and the previous sample output $h^{(n)}$ to compute a prediction $h^{(n+1)}$ for the current sample.

In the previous paragraphs, we used generic terms such as compute and generate. However, for an LSTM, the equations to calculate each term presented above is well defined and are given by:

$$\begin{cases} P^{(n+1)} &= \zeta(W_f \cdot [h^{(n)}, x^{(n+1)}] + b_f) \\ I^{(n+1)} &= \zeta(W_i \cdot [h^{(n)}, x^{(n+1)}] + b_i) \\ \xi &= \tanh(W_\xi \cdot [h^{(n)}, x^{(n+1)}] + b_\xi) \\ C^{(n+1)} &= P^{(n+1)} \odot C^{(n)} + \xi \odot I^{(n+1)} \\ O^{(n+1)} &= \zeta(W_o \cdot [h^{(n)}, x^{(n+1)}] + b_o) \\ h^{(n+1)} &= O^{(n+1)} \odot \tanh(C^{n+1}) \end{cases} \tag{2.5}$$

In the equation above, $[x, y]$ is the concatenation of vectors x and y and $x \odot y$ is the Hadamard or element-wise product. Also, W_x and b_x represent, respectively, the weights and bias of the network when computing variable x. These are the values that need to be learned during the training process.

When it comes to applying this architecture to solving the problem of prediciting hard drive failure in data centers, there are not many research projects that focused in this problem that have been published. Nevertheless, there are some related works.

The first of them was done by Zhang et al. ([31]). The main objective of their work was to develop a symbolization-based feature extraction algorithm to improve event detection in LSTMs.

To show the impact of their approach they did perform an experiment on hardware failure detection. However, they were presenting a more general method. Consequently, they used different metrics such as balanced accuracy instead of FAR and FDR that would allow us to directly compare with other methods.

Despite us not having access do the desired metrics we can deduce some aspects of their results that allow us to compare to other works. In order to do that, let positive be the event of the hard drive failing and negative the event of it not failing.

Then the true positives TP are the hard drives whose failure was correctly predicted. The true negative TN are the hard drives that did not fail and that were not flagged as going to fail. The false negatives FN are the hard drives that failed but were not flagged as so. And, the false positives FP are the hard drives that did not fail but that were predicted as going to fail.

This allows us to rewrite the FDR and the FAR as follows:

$$\begin{cases} FDR = \dfrac{TP}{TP + FN} \\ FAR = \dfrac{FP}{FP + TN} \end{cases} \tag{2.6}$$

We then introduce the definition of Balanced Accuracy(BA), which was the metric used in [31]:

$$BA \equiv \frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right) \tag{2.7}$$

But by manipulating 2.7, it is possible to rewrite it in terms of the FDR and the FAR:

$$BA = \frac{FDR + (1 - FAR)}{2} \tag{2.8}$$

So, even though it is not possible to retrieve the FDR and the FAR by themselves, it is possible to find a range of possible values for them from the balanced accuracy value.

In [31], the best BA they achieved over all the experiments with the LSTMs, with or without their feature extraction algorithm, was 85.2%. But, since BA is the average of FDR and $(1 - FAR)$, it implies that one of these values is at most 85.2%, else their average would be bigger.

So, either the FDR they achieved is under 85.2% or their FAR is above 14.8% which is much worse than the one achieved with any other methods we have discussed.

This indicates that their research cannot be used as a reference for the hard disk failure detection problem.

There is an additional paper by Das et al. ([32]) that also applies LSTM to hardware failure detection. However, it is not for hard drives in data centers, but rather to nodes in a supercomputer which leads to a profile much different than the papers discussed so far.

The first main difference is that the time scales in which it operates is much smaller. The TIA they achieve is of the order of 100 seconds, which indicates a very different behavior than the one found in data centers in which it is possible to predict a failure more than 100 hours in advance ([11], [8]).

But the most important difference is that in [32] they try to predict failures that happen in components other than the hard drive. So, they do not make use of SMART attributes, they instead analyze system logs.

Therefore, their motivation to use LSTMs is to detect patterns in text rather than to study time-series.

The FDR they achieve with their LSTM-based approach is between 85.1% and 87.5%. However it is not possible to compare it to the other results presented above since their problem is quite different from the detection of hard drive failures in data centers.

2.5 Other Techniques

There are three additional methods that have been used to tackle the hard drive failure prediction problem and that are worth mentioning. Since these approaches will not be implemented by our current work, we will limit ourselves to a briefer explanation of the theory behind them.

2.6 Random Forest

One of the problems faced by Decision Trees, specially when they become large is over-fitting [33]. This is due to the fact that the split values for each attribute used by the

tree are directy taken from the values on the training set. So, when the tree is deep and there are only a few training samples in a node, the splitting values will sharply follow the ones in the training set.

Another way to explain this is that since a Decision Tree can closely follow the patterns observed in the training set because it can choose the splitting threshold independently of the ancestor nodes in the tree, it presents a small bias. But in Machine Learning there is a principle known as the Bias-Variance tradeoff that states that a model with low bias will present a high variance and vice-versa [34].

More concretely, for our problem at hand, suppose that there are only a few samples in the training set with a certain cause of failure. During training, if they are in a node **N** and are mixed with other samples, we may be able to correctly put them in a leaf that is a child of **N**.

So, the model will have a good performance on the training set which corresponds to a small bias, since the expected and predicted value will be the same. However, imagine that when evaluating an unseen sample there is an attribute, that is not important, to identify this cause of failure, is slightly different from the values observed in the training set.

Then, the path taken will not go through **N** and thus it won't go through the same evaluation process as samples that are simillar. As a result, a small change in a random attribute can cause a huge differnce on the path traversed and thus on the result. Having an algorithm that can behave very differently for slightly different inputs corresponds to a model that has a high variance.

In order to tackle this, the concept of Random Forests has been introduced [35]. The idea is to train multiple, independent Decision Trees at once, each one trained with a different subset of the training data. Notice that these subsets do not have to be disjoint, specially when the train set is relatively small.

Random Forests have performed well for a variety of tasks, ranging from image recognition to Alzheimer's disease detection and prediction [36].

The set of trees has, therefore, a smaller bias when compared to the one trained using all the data at once. This is due to the fact that we can reduce the probability of overfitting, since there will have a larger varierty of decision nodes.

If we return to the example of a specific cause of failure, the training samples corresponding to it will not even necessarily be on the same tree anymore. So, there is a larger set of nodes that have been trained with samples corresponding to this failure. Therefore, the probability of going through one of them when traversing the tree is higher even if the unseen sample does not correspond perfectly.

The drawback is that since the training samples with the same failure cause will be spread, the probability of having a node that had multiple ones when training is smaller, which can increase the error on the training set and on samples very simillar to the ones in the training set.

Once the set of trees is created and trained, it remains to decide how to regroup all of them.

The most common approach when asking for the model to predict the result for a sample is to take the value predicted by each tree and then combine them. For a classification problem, it can be returning the most common predicted class, while for a regression problem it can be done by taking the average of the outputs of the trees.

This can be done quite efficiently, since, at each node, it suffices to compare the value of a specific attribute with a threshold and go to the corresponding child. Moreover, since the traversal processes are independent, they can even be done in parallel.

A research by Shen et al. [10] used Random Forests for hard drive failure prediction. One of the most interesting aspects of their research is how they combined the results of the different tree of the forests.

Instead of simply checking whether most trees had an output of **good** or **bad** for a specific sample, they giving different trees different weights based on a clustering algorithm.

More specifically, they performed the clustering process for the good and bad samples independently. Then, when predicting the outcome for an unseen sample, the clusters c_1 and c_2 to which it would belong if it were a good or bad one are computed.

Since it is known which samples of the training set were used to train each tree, it is possible to evaluate the accuracy of each tree for the training samples in c_1 and in c_2. Based on these accuracies, it is possible to give a bigger weight to trees that better learned the samples of the clusters to which the sample being evaluate belongs.

They used a simple algorithm in which the weight of a tree is either 0 or 1 depending on whether its accuracy for its training sample belonging to either c_1 or c_2 is at least 0.5. However, it is not difficult to imagine a slightly more complex algorithm that results in a weight $w \in [0, 1]$ which is directly proportional to the accuracy.

Despite this simplicity, they were able to achieve a great performance. The FDR obtained was above 98% while the FDR was kept under 0.1% for multiple scenarios.

What allowed them to get these excellent outcomes, besides of course the algorithm, was their dataset that contained more than 2 million samples of SMART attribute snapshots.

2.6.1 Support-Vector Machine

A Support-Vector Machine (SVM) is a non-probabilistic classifier [37]. The idea behind it is to plot the samples in an n-dimensional space, in which n corresponds to the number of features in the vector.

Then, the algorithm finds the hyperplane that minimizes its loss function. The points on one side of the hyperplane correspond to the positive class and the other to the negative class.

The main drawback of SVMs is that drawing a hyperplane means that it can only detect linear dependencies. So, if there are points in a 2D-plane and the boundaries between the good and bad class is the circumference centered at the origin with radius 1, there is no way that a line can be drawn to correctly divide both regions.

The solution to this is to use the kernel-method, in which dimensions of the original vector are combined using non-linear function and the output of the function is assigned to a new component of the sample vector. This allows the SVM to correctly learn non-linear dependencies between the variables.

In the example above, we can transform the vector $v = (x, y)$ into $v' = (x, y, x^2 + y^2)$ and then and apply the SVM to the set of v'. Then the SVM can correctly generate the plane $x_3 = 1$ and thus solve the classification problem without needing to learn explicitly non-linear dependencies.

When it comes to applying SVMs to predicting hard drive failure, the best results were achieved in [8]. They used hyperparameters on their model in order to prioritize either a small FAR or a big FDR.

When optimizing for the smallest FAR possible, Zhu et al. obtained a FAR of 0.03% and an FDR of 68.5%. When priority was giving to increasing the FDR, they obtained a FAR of 0.3% but the FDR was increased to 80.0%.

Support-Vector Machines, when compared to other approaches, present, therefore, a method that is able to achieve a smaller FAR. However, in contrast, it is not able to attain FDRs as good as other methods while keeping an acceptable FAR.

In the research by Zhu et al. they kept results that had an FAR smaller than 5%, so it implies that improving the FDR above 80% requires increasing the FAR to unacceptable levels.

2.6.2 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a deep learning machine learning model. It is usually applied to computer vision tasks. Just like BPNNs discussed in Section 2.2, it is not recurrent meaning that the output for a certain sample is not propagated to following ones.

The main aspect that differs a CNN from a BPNN is the presence of convolutional layers [38]. In a convolutional layer, a neuron in layer l is not connected to every neuron in layer $l-1$. Instead, it is connected to a few consecutive nodes of layer $l-1$.

This is useful for handling data with many input parameters, such as images, in which each pixel corresponds to an input. For a 100x100 image, fully connecting two layers would require 10,000 weights. By using a 5x5 convolution window with shared weights, it is possible to reduce this value to only 25 weights, or 400 times less. This allows the network to be much deeper.

A relevant attribute of a CNN is its translation invariance [39]. The use of shared weights implies that shifting some values by a certain constant value does not hinder the network's ability to recognize a certain pattern. This is because the convolutions will be performed with the same weights no matter in which part of the image a certain sequence of values is.

It was exactly this aspect of the CNN that motivated Sun et al. ([40]) to adapt CNNs to handle time series in order to tackle the hard drive failure prediction problem. For a time-series, translation invariance implies that a pattern due to a sequence of values or events can be detected indepndently of where in the sequence it occurs.

In order to obtain, they also had to adapt the loss function to handle the imbalanced dataset that contained many more good samples than failing ones, which is an intrisic aspect of the problem at hand.

So, they gave a bigger weight to failing samples that were misclassified, else the model would be prone to classify almost every disk as a good one. This is due to the fact that, with a classic loss function, even if the model misclassified every failing disk, its total loss would be much smaller than if it misclassified even a small percentage of the good disks.

In their results, the metric they presented that we can use to compare with other methods was the precision, which corresponds to the FDR. They achieved a FDR of 75% which is considerably smaller than the one attained by other models.

But this is mainly due to the fact that they used an heterogeneous dataset, meaning that they did not train different networks for different hard drive models. So, it is difficult to compare this results with the ones presented so far.

3 Methods

The objective of this thesis is to compare different approaches to tackle the hard drive failure prediction formally introduced in Section 1.2 problem using SMART attributes. However, we do not limit ourselves to already published results and we implement a software capable of executing the training and evaluation processes for different models.

This allows us to not only to more objectively compare the different methods by running them on the same dataset, but also to more deeply explore the influence of the hyper-parameters on the performance of the models. For instance, current work on neural networks, such as [8] and [9], for this problem do not explore the impacts of the number of nodes in the hidden layer on their performance metrics.

Finally, the current work aims to more thoroughly test the impact of the Health Status approach proposed in [9]. This will be done by firstly implementing it for neural networks other than the RNN one used in the original work.

Secondly, we will change the total number of distinct health status and observe how the performance of the models evolve. More precisely, we will change the value of N in equation 2.2 since the original research in [9] uses a constant value of $N = 6$.

3.1 Health Status

The current work aims to apply the concept of Health Status introduced in [9] and [11]. We intend to find how powerful this concept can be to improve the performance of different types of models.

The ideas is to, instead of mapping all good samples to a class labeled 1 and all bad samples to a class labeled 0. Instead, we can extend the range of different labels beyond these two values.

The rationale behind it is that samples of a failing disk that were taken closer to the moment of failure probably better describe a failing disk than the ones from the same disk that were taken much before the critical moment.

Now, the challenge is to determine how to map different samples of the same disk to distinct values. Let n be the number of samples from a bad disk that we consider as belonging to the critical window.

The first health status algorithm we have is the one we will call the Discrete Health Status Algorithm. It is the one used by Xu et al. [9] and is described by Equation 2.2.

One of the main advantages of using it is that if we set $N = 2$ in Equation 2.2, we will get label 0 for the bad disks and 1 for the good disks. So, we can consider this method a generalization of the ad-hoc labeling that is used in most binary classification problems.

The second approach we can use for an algorithm that computes the health status of a certain sample is to drop the constraint that it must be an integer, while still keeping it linear and mapping every value to the interval $[0, N-1]$.

So, we propose the following algorithm that we will refer as the Continuous Health Status Algorithm:

$$c_i(t) = \begin{cases} N-1, \text{ if } i \in \textbf{good} \\ (T_i - t)\dfrac{(N-1)}{n}, 0 \leq T_i - t < n, , \text{ if } i \in \textbf{bad} \end{cases} \tag{3.1}$$

The Continuous Health Status Algorithm completely abandons the concept of classes and instead uses a score system.

3.2 Models

In total, we have implemented five distinct models to tackle the hard drive failure prediction problem. Namely, they are Classification Trees, Regression Trees, Backpropagation Neural Networks (BPNN), Recurrent Neural Networks (RNN) and Long Short-Term Memory Networks (LSTM).

The theory behind the models we implemented was explained in Chapter 2. So, in the following section, we present the motivation behind each model as well as the input expected and the ouput provided by each of them.

3.2.1 Decision Tree

Classification and Regression Trees can be grouped under the umbrella term Decision Tree. Despite this, their application to the problem at hand is not done the same way.

Classification Tree is a model designed to tackle classification problems as its name suggests. Therefore, once we are trying to determine if a hard drive is going to fail soon or not and we have a set of SMART attributes, the idea to try this model follows trivially

However, the main drawback of a Classification Tree is that it cannot easily assign a degree of confidence to its result, since it is limited to outputting an integer corresponding to a class.

The Regression Tree allows us to tackle this limitation by not looking at the problem from a classification point of view. Instead, it aims to assign a score to each sample, which can be a real number.

For the Regression Tree, the concept of classes does not even exist. It is up to us to map the classes to scores that are fed to the model and then interpret its output to map it to a class of our problem.

So, it can be an extremely powerful tool as long as we can assign scores instead of labels to our samples. Therefore, the approach of using Health Status matches perfectly with this use case.

In order to train a Classification Tree and a Regression Tree, the training set is formatted the same way: a set of samples of SMART attribute values labeled with their corresponding Health Status. The only limitation is that the Classification Tree is limited to integer values of Health Status.

On the other hand, the output produced by both models is quite different. When an unseen sample is given to the Classification Tree, it will simply output an integer value between 0 and $N - 1$.

In contrast, the Regression Tree can produce a real score and can even take values that were not present in the training set. This values can be bigger than $N - 1$ or even smaller than 0.

3.2.2 Neural Networks

We use three different neural network architectures: BPNN, RNN and LSTM.

The idea behind using a BPNN is similar to the one to use Decision Trees: it can work well for problems in which we have to classify samples based on real valued feature vectors.

On the other hand, RNNs and LSTMs work quite differently. As discussed in sections 2.3 and 2.4, these networks are adapted to handle sequences of values instead of processing different samples independently. This lends itself perfectly to our use case in which we have in our dataset different time series, which is a type of sequence.

This difference allows us to divide our models into two different groups. We will call one the time insensitive class and the other the time sensitive class.

Time insensitive models are the ones that treats each sample independently even though some come from the same hard drive. So, these are the ones that do not have a direct way to model time dependencies. In our experiments, these are Decision Tree and BPNN.

In contrast, time sensitive models expect to receive a sequence of samples of the same hard drive in chronological order, since they are able to model how the values of a sample at a previous time impact the current one. In our experiments, these are RNN and LSTM.

The input data expected by the BPNN is therefore similar to the one for Decision Trees discussed in Subsection 3.2.1: a set of samples and health status values that are independent. This is the case for both the training and testing processes.

The time sensitive models, however, add some additional constraints to the input data. The samples from the same disk must be fed in chronological order in order for them to be able to perform their time propagation operations correctly. Moreover, the networks need to be know where the data from a hard disk ends and data from a new one begins so it can erase its memory.

Even though all data from a disk is fed sequentially, it is important to notice that the time sensitive models will also generate a different output for each sample instead of classifying the entire sequence at once.

When it comes to the ouput, we can adapt any of the three neural network architectures we have to behave in two ways. The first is the classic approach to a binary classification problem in which there is a single output neuron.

As a consequence of its architecture, can only handle cases in which $N = 2$, so we call them Binary Neural Network models.

The output value then is constrained to the interval of $[0, 1]$. In practice, this is done by using a sigmoid as the activation function of the last neuron, since it maps values in $]-\infty, \infty[$ to $]0, 1[$

The advantage of this approach is that it provides an easy way to interpret its output: the value of the output node can be seen as the degree of confidence that the model has that the given sample belong to the class labeled as 1 (in our case this corresponds to a good disk).

The second way to organize the ouput of the neural network is to use a different output node for each of the N classes being predicted. This time, the constraint of $N = 2$ can be dropped. Because of this, we call the models following this architecture as Multi-Level Neural Networks.

Each of the N output nodes produces a value that is interpreted as a score assigned to each class. The bigger the value of the i^{th} output node, the higher the confidence of the model that the sample belongs to the class i.

Moreover, when training such a model, there is the need to translate the Health Status value, which will be an integer number, into an N-dimensional vector since this is the format expected by the model.

We use the one-hot encoding method, in which the vector corresponding to a sample whose Health Status is h has all its components equal to 0 except for the h^{th} one, assuming 0-based indexing. So, if $N = 4$ and $h = 1$, the corresponding vector is $(0, 1, 0, 0)$.

The output provided by each model that we have explained in this section refers to how it handles a single sample. However, the problem we are trying to solve is to classify a disk as about to fail instead and not a single sample.

As a result, the output of the models are not enough to detect failing drives. Some additional steps are needed to allow us to make predictions about a disk. The operations we perform to take the results for samples of a disk into a prediction about its state are detailed in Subsection 3.3.6.

3.3 Setup

In this section we describe the components of the experiment beside the model such as the feature selection and the voting algorithm. These are steps that are common to all models and therefore allow us to more objectively compare the different methods than by comparing previous work since the preprocessing steps can be made exactly the same and the models can be trained and evaluated on the same dataset.

3.3.1 Change Rate

The first step is to compute the change rate of the attributes for each sample. This is done due to the fact that we are using models as Decision Trees and BPNNs presented in sections 2.1 and 2.2. These models evaluate each sample independently and therefore cannot make use of the fact that we are working with a time series unless we explicitly add new features to each sample.

More formally, we choose a positive integer Δt that is the same for every sample. Then for the hard drive i we will have a sequence $(x_{i,t}), t \in \{1, \ldots, T\}$ where $x_{i,t}$ is a vector with the SMART values for the disk at time t. We then compute the sequence:

$$(x'_{i,t}), t > \Delta t, \text{with } x'_{i,t} = x_{i,t} \oplus (x_{i,t} - x_{i,t-\Delta t}) \tag{3.2}$$

Here $\oplus$ denote vector concatenation meaning that we insert additional components to the vector. It is this sequence $(x'_{i,t})$ that is then passed to the next steps of the experiment.

Notice that the first Δt samples need to be discarded, since the change rate is not well defined for them. This fact, as well as the observation that two samples too far apart should not be as strongly correlated than two closer ones suggest that the value of Δt in practice should be kept small. As a result, for most experiments, we keep Δt equal to 1.

This step is completely optional and the next steps continue behaving correctly if the change rates are not inserted, they just works with smaller vectors. This allows us to do experiments without the change rate computation to observe its impact on the performance of the models.

3.3.2 Feature Selection

The next stage of the preprocessing procedure is to perform the feature selection. Once we have multiple sequences of vectors with or without change rates, we need to decide which features are actually going to be fed to the model.

The feature selection process has become a key step in real-world scenarios in order to remove irrelevant features and reduce the dimensionality of the data. Models working with more relevant features can more easily learn the problem at hand since a smaller number of relationships between the different parameters need to be found [41].

For the failure prediction problem we have, the challenge is to remove the features that behave similarly in the good and bad disks. In order to do that we compare 3 different methods: z-score, reverse arrangement and rank-sum.

The three of them compare the distribution of the same feature on the good and bad disks. A feature can be either a SMART attribute or the change rate of a SMART attribute computed in the previous step, but all features are evaluated independently and in the same manner. If these distributions are very different, the score assigned to the feature is higher than if the distributions are more similar.

The approach used in our experiments is to decide a number F of features to keep. Then, the F features with the highest score are kept and the others are discarded.

This is a less biased approach than using a threshold value for the score of a feature in order for it to be kept. This is due to the fact that these thresholds would be need to be different for each function used since the score distribution depends on the function.

Moreover, this allows for the approach to be more easily generalized for different datasets that can present different behaviors.

The main drawback of this approach is that multiple experiments need to be performed in order to find the value of F taht maximizes the performance. Nevertheless, the advantages easily outweight this limitation.

The z-score compares the difference of the means of an attribute over the two classes [42]. It is defined as follows:

$$z \equiv \frac{m_f - m_g}{\sqrt{\frac{\sigma_f^2}{n_f} + \frac{\sigma_g^2}{n_g}}} \tag{3.3}$$

Where m_f, σ_f and n_f are, respectively, the average of the feature over the failed samples, its standard deviation and the number of failed samples respectively. The values m_g, σ_g and n_g represent the same quantities but for the good samples.

If the average of the feature in both classes are relatively small when compared to their variances, then the probability that they come from the same distribution is bigger and the value of z will be closer to 0.

Since for the feature selection step we are interested only on whether they are similar or not and not which the distribution is bigger, we use $|z|$ as the score that the feature selection algorithm actually outputs.

The reverse arrangement test indicates whether a given sequence has a tendency to increase or decrease or not [42]. Let $(f_{i,t}), t \in \{1, \ldots, T\}$ be the sequence representing the values of a certain feature for disk i. Then, we define the test statistic A as:

$$A \equiv \sum A_t, \text{with } A_t \equiv \sum_{j=t+1}^{T} I(f_{i,t} > f_{i,j}) \tag{3.4}$$

Here, I is the identity function and is equal to 1 if the condition passed as argument is verified and 0 otherwise.

If the values of $f_{i,t}$ come from the same distribution, then there is no temporal tendency and the expected value of A can be computed as well as its standard deviation.

By analyzing the above formula we notice that if $f_{i,t}$ tends to decrease as times passes, the value of A will be large. On the other hand, if its tendency is to increase, then A will be large.

So, if the value of A is too far away from the expected average when compared to the expected standard deviation of a non-time dependent distribution, it allows us to affirm, up to some degree of confidence, that there actually is a time-dependency of $f_{i,t}$.

For a given value of T and a certain degree of confidence, the lower and upper bounds of A for a time-independent distribution can be computed. These values are available in tables such as Appendix A.6 of [43].

In our experiment, for each disk we keep its 100 last samples (the ones that fail earlier are discarded). Then we compute the value of A for each disk. We then count the number of good and bad disks in which a time-dependency can be observed with a high degree of confidence.

In order to do this we take into account the upper and lower bounds of [43] with $\alpha = 2\%$ meaning that in order for a feature of a disk to be flagged as time dependent there is a probability 98% that a time-independent distribution would not yield a value of A so far from the expected value.

Finally, the score assigned to a feature is the absolute value of the difference between the number of bad and good samples that were flagged as time dependent. In mathematical notation, for a given feature f:

$$ra(f) \equiv \left| \sum_{d \in \mathbf{good}} I(A(d,f) > U \vee A(d,f) < L) - \sum_{d \in \mathbf{bad}} I(A(d,f) > U \vee A(d,f) < L) \right| \tag{3.5}$$

Where $A(d, f)$ denotes the test static defined in Equation 3.4 for disk d and feature f. U and L are, respectively, the upper bound and the lower bound of the test statistic for the confidence value we are using.

Taking the difference between the two classes allows to handle features that are trivially always increasing or decreasing on both sets. For example, if we only computed A for the bad samples, some features such as the number of hours the disk is in power on state would always have a high score since it is monotonically increasing for every sample.

The third feature selection algorithm we have used is the rank-sum test, also known as the Mann-Whitney U test. It works in a similar way as the other methods by trying to gauge the probability that two sequences of values come from the same distribution.

In order to explain the idea behind this method we take a generic example in which we have two sequences $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$. The challenge is to determine with which degree of confidence we can affirm that X and Y were obtained by sampling the same distribution.

The rank-sum test achieves this by first merging the two sequences into a new sequence Z and sorting it. It uses the fact that if X and Y come from the same distribution, then the values of X will probably be well distributed on the sorted sequence Z instead of being concentrated at the beginning or at the end.

So, let the elements of X be at positions $(p_1, \ldots, p_n)$ in Z and the ones of Y be at positions $(q_1, \ldots, p_m)$. We notice that $(p_1, \ldots, p_n) \oplus (q_1, \ldots, q_m)$ is a permutation of $(1, \ldots, n+m)$. Then, the test statistic U is given by [44]:

$$U = \min\left(nm + \frac{n(n+1)}{2} - R_1, nm + \frac{m(m+1)}{2} - R_2\right) \tag{3.6}$$

With:

$$\begin{cases} R_1 &= \sum_{i=1}^{n} p_i \\ R_2 &= \sum_{i=1}^{m} q_i \end{cases} \tag{3.7}$$

From the values of U, n, m it is possible then to calculate the p-value of the distribution. The p-value corresponds the degree of confidence that our hypothesis that both sequences come from the same distribution is satisfied.

So, if, for a certain feature, the p-value obtained is small, we want to keep it since its value is much different on the good and bad samples. Therefore, the score returned by our algorithm is the opposite of the p-value.

3.3.3 Train-Test Split

In order to correctly train and evaluate our models, we need to split our data into training and testing sets.

This is a delicate process due to the fact that the data is not balanced, meaning that we have many more good than bad hard drives, which is a characteristic of the hard drive failure prediction problem. Also, we deal with distinct models that process data differently and therefore introduce the need for different operations to be performed at this step depending on the model at hand.

As discussed previously, time sensitive and time insensitive models expect the training and test data to be organized differently.

For the two classes of model, the first step is to choose a value n of samples of the bad hard drives that we will consider as being in the critical window. This means that we only consider the last n samples before failure as describing a failing hard drive.

As we have discussed in Section 3.1 not all of these n samples are treated equally, our experiment takes into account how close to the failure point each sample is. However, at this point we can already discard every sample from a bad disk that is not one of the last n taken for the corresponding hard drive. Also, we completely discard every disk that has less n samples.

The next step is to do a 70/30 split in the training and data sets respectively. Since we have a limited amount of bad hard drives, we want to use all of them. So, we assign 70% of them to the training set and the other 30% to the testing set.

Also, we need to choose a good-bad ratio r, meaning that for each sample from a bad disk in the training set, we will have r from good hard drives in it.

For the time insensitive models, we split the good disks in training and testing sets using a 70/30 split as well. Then, if the training set has b bad disks, each contributing with n samples, then we choose at random $b \cdot n \cdot r$ samples from the good training set to include in the final training set.

The 30% of the good and bad sets form our testing set. The split of the good disks is necessary before the sampling in order to ensure that disks in the testing set have not been seen by the model while training it.

For the time sensitive models, the sampling step of the good disks to include in the training set is a little more delicate. This is due to the fact that we need to obtain sequences of samples to train the models, so the samples can't be sampled independently.

So, we use the fact that a good disk, by definition, did not fail during the observation period. As a consequence, the same number m of samples have been measured for each of them.

Therefore, to form the training set, we choose $\frac{b \cdot n \cdot r}{m}$ distinct hard drives and include all of their samples. In the end, we will have $\frac{b \cdot n \cdot r}{m} \cdot m = b \cdot n \cdot r$ samples from good disks. This is the same number that we have for the time insensitive models, just organized in a different way.

This allows us to directly compare the results for different values of n and r without depending on other characteristics of the dataset.

3.3.4 Health Status

Before training our models, we need to perform a last preprocessing step that corresponds to computing the Health Status value corresponding to each hard drive sample.

We then can use either the Continuous or Discrete Health Status Algorithms described in Section 3.1.

The Multi-Level Neural Networks and the Classification Tree models only support the Discrete version of the algorithm. This is due to the fact that they work directly with the idea of classes.

The Binary Neural Network models also have been designed to handle the discrete health status algorithm. However, they have the additional constraint of $N = 2$.

The Regression Tree can make use of both algorithms, even though it is more adapted to the continuous version, since the models mentioned in the previous paragraph are specialized to handle a small set of discrete output values.

3.3.5 Training

After performing the train-test split and computing the Health Status for the training samples, we can feed the training set to our models in order to train them. The specifics of how it works for each model will be discussed in more details in Section 3.2.

3.3.6 Voting

The final step is to actually apply the models to the samples on the test set. This allows us to evaluate if the model has learned to predict hard drives failures for samples it has not yet seem.

In a real world scenario we will have, for each disk, a sequence $(x_1, \ldots, x_n)$ of its SMART attribute values taken at nearly constant intervals. The objective is to determine if these samples allows us to say that the disk is going to fail in the near future.

At this point, we assume that we have a model that is able to take a sample (or a sequence of them in the case of time sensitive models) and generate a prediction for it. The format of this output can be slightly different depending on the model's architecture.

The ad-hoc approach is to only consider the output of the model for the sample x_n. If it corresponds to a failing sample, we flag the disk as going to fail soon.

However, a more robust approach is to take into account an interval of v consecutive values $(x_{n-v+1}, \ldots, x_n)$. Imagine the case in which there is a single sample representing a failing state surrounded by tens of good samples. Then, probably, the hard drive is still working as intended.

The approach we have here reduces the FDR or at least the TIA, since a single failing sample is necessary but not sufficient to flag a disk as failing. However, for the same reason, we also reduce the FAR. As discussed previously, this is can be an interesting trade off.

In order to make the final decision of whether a disk is going to fail or not, we choose two values: the number of votes v and the voting threshold τ, which is a number between 0 and 1. Then we give the samples $(x_{n-v+1}, \ldots, x_n)$ to our model and receive the outputs $(y_{n-v+1}, \ldots, y_n)$.

We then propose to combine the values of the ouputs two different ways in order to obtain the final prediction of the model. The first we will call Class Based Voting algorithm and is inspired by [9].

We first convert each y_i into a class C_i. This is done differently according to the model at hand. For a Classification Tree, it is straightforward: $C_i = y_i$, since the output is already an integer.

For a Regression Tree, we do $C_i = \min(\max(0, \lfloor y_i \rceil), N-1)$. Where $\lfloor z \rceil$ denotes the closest integer to z. The min and max functions allows us to handle cases in which $y_i < 0$ or $y_i > N-1$.

For a Binary Neural Network, we have a similar expression: $C_i = \lfloor y_i \rceil$, since y_i is constrained between 0 and 1.

For a Multi-Level Neural Network, we use the class with the largest score in the output vector. Mathematically, $C_i = \text{argmax}_{j \in \{0,\dots,N-1\}}(y_i[j])$.

Once C_i is defined for each sample, we can obtain a histogram H with the number of samples classified in each class. We can then use H to determine if the disk is in a healthy or failing state as follows:

$$\text{CB}(H) = \begin{cases} \text{Healthy, if } (1-\tau)\sum_{j=0}^{\max(0,N-3)} H_j \leq \tau H_{N-1} \\ \text{Failure, otherwise} \end{cases} \tag{3.8}$$

This is similar to Equations 2.3. However, we adapt it to be able to handle the case of $N = 2$.

Also, we introduce a parameter τ. This denotes that a ratio strictly bigger than τ of the samples being considered (meaning those with $C_i \neq N-2$) have to be in classes other than $N-1$.

The second voting method is specific for Multi-Level Neural Networks. It is simillar to the Class Based Voting, but with a modified value of C_i.

We make use of the fact that we get a vector with N scores that can be interpreted as the degree of confidence of the model that the sample belongs to each of the N classes.

When we increase the value of N, the number of samples of each class from 0 to $N-2$ seen during the training decreases, even though the sum over all of this classes remains constant. So, the model may be more biased towards the class $N-1$ since it has seen more samples with it.

In order to reduce the impact of this bias, we propose a method that recombines the values in classes from 0 to $N-2$. In this case we get:

$$C_i = \begin{cases} 0, \text{ if } \sum_{j=0}^{N-3} y_i[j] > y_{N-1} \\ 1, \text{ otherwise} \end{cases} \tag{3.9}$$

We then make a histogram as before, but with only two components. Finally, we apply the following formula that we will call the Score Based Voting Algorithm:

$$\text{SB}(H) = \begin{cases} \text{Healthy, if} (1-\tau)H_0 \leq \tau H_1 \\ \text{Failure, otherwise} \end{cases} \tag{3.10}$$

3.3.7 Evaluation

In order to evaluate the model, we use the test set we described in subsection 3.3.3. It has m distinct disks with the expected classifcation result for each of them: Healthy or Failing.

In order to simulate what happens in a real-world scenario, for a disk with samples $(x_1, \ldots, x_n)$, for each $i \geq v$, we take a slice $(x_{i-v+1}, \ldots, x_i)$ of the samples. We then perform the evaluation and voting processes described in subsection 3.3.6 using our previously trained model.

If for a value of i we get a diagnosis of a healthy disk, then we continue the process for $i+1$. If we get to $i > n$ then the verdict of our experiment is that the disk is healthy.

If the expected diagnosis for this disk was to be healthy, then we add 1 to our count of true negatives. Otherwise, we add 1 to the number of false negatives.

If for some value of i the voting algorithm indicates that the disk is failing, then the verdict for this disk is that it is a failing one.

If the expected diagnosis for this disk was to be healthy, then we add 1 to our count of false positives. Otherwise, we add 1 to the number of true positives and store the time in advance with which the failure was detected, that is we store the value of $n - i$.

In the end, this experiments allows us to easily compute the FAR and the FDR for our testing set. We also compute the average time in advance as well as its standard deviation.

The average TIA indicates how much time, on average, there is to back up the data on the disk and replace it after our system flags a disk as going to fail in order to prevent data loss and disruption of service.

The standard deviation of the TIA is a measure of how much its value varies from one disk to another. Ideally, this value should be small to indicate that almost all failures are detected with approximately the same TIA.

3.4 Software

In order to perform our experiments and objectively compare the different models, we have developed a program capable of training and evaluating any of the models described above. In order to allow others to verify our results, we have made the code open source[1].

We already had in mind the need to test multiple methods for different steps of the experiment, from the feature selection algorithm to the models and the voting algorithm. So, from the start, the architecture of the program was planned so that it would be highly extensible.

This should making developing new methods and testing them with our library straightforward. There is also the added benefit that it also allows to compare the results on the same datasets with different state of the art methods.

The models we used were the implementations from PyTorch (for the neural networks) and scikit-learn (for the decision trees).

We also used an approach of decoupling the declaration of a model from the methods used to train and evaluate it. Since our models can be split into time sensitive and time

[1]Code available at `https://github.com/Miguel0312/health-status-experiment`.

insensitive as well as binary or multi-level, for example, different models have different parts of the pipeline in common.

So, a class representing a model declares its network as well as a description, which is a series of flags indicating if it is time sensitive or not, for example. We then use a dynamic dispatching approach in which a method will read the description of the model and then call the appropriate algorithm.

This can be more concretely observed on the `train_model` method of the `FailureDetectionNN` class. Instead of overriding this method for each class, it simply calls `utils.trainNN` which will read the values of the description flags and call the correct method.

This makes the code a lot more robust by not repeating code and making sure that a bug that is solved or a modifcation thatt is made for one model is propagated to all of them.

We also made the decision of using, for example, two distinct flags `BINARY` and `MULTILEVEL` instead of checking for the presence or not of the `BINARY` flag. This is the allow for future models that are neither binary nor multi-level without making it necessary to modify previously existing code.

Finally, one of the most important features of such kind of library is the ability to train multiple models and modify their parameters without touching the source code. Therefore, the configurations to the experiments of our libray is not done through code but instead through human readable text files.

Our code is able to read a file and instantiate all the steps needed to train and evaluate using the given parameters.

The file format we chose was TOML (Tom's Obvious Minimal Language). This language allows us to separate the attributes into different tables such as dataset, preprocessing, model and vote which makes it easier to read, interpret an modify.

Moreover, TOML supports arrays out of the gate. We used that to be able to instantiate multiple experiments from a single configuration file.

Suppose that a parameter is an array instead of an integer or a string. Then our code will extend the other parameters to also be an array. For those that are single values, it suffices to create a list with multiple copies of the same elements.

This make it much easier to perform multiple experiments at once and, more importantly, to analyze the impact of each of the parameters on the performance of the model.

Two examples of what the configuration files used in our experiments look like are available on Appendix A.

3.5 Parameters

In this section we describe the parameters that we were able to control during our experiments. Most of them correspond to some values that have been introduced on previous sections, but have a common vocabulary will be useful later on to discuss the results.

- **Number of Failing Samples**: the number of failing samples that we consider as belonging to the critical window of a failing disk. It is the value of n on Equations 2.2 and 3.1.
- **Change Rate Interval**: the delta we use to compute the change rates. It corresponds to the value of Δt on Equation 3.2.
- **Feature Count**: the number of features (may it be SMART values or its change rates) to be kept by the selection feature algorithm. It is the value F discussed in subsection 3.3.2.
- **Feature Selection Algorithm**: the algorithm used to rank the features in order to decide which ones to keep and which to discard. There are 3 of them: z-score, rank-sum and reverse arrangement.
- **Health Status Algorithm**: how the program should compute the health status values for the training set. There are two possible algorithms: discrete (corresponding to Equation 2.2) and continuous (described by equation 3.1).
- **Good-Bad Ratio**: the number of good samples to be included in the training set for each bad sample. The number of bad samples is constant because we have a limited amount of them and therefore we choose to use as many as possible. It is the value of r on Subsection 3.3.3.

- **Health Status Count**: the number of distinct health status classes for the discrete case or the maximum value of the health status for the continuous case. It corresponds to the value of N on Equations 2.2 and 3.1.
- **Hidden Nodes**: the number of nodes on the hidden layer of the neural network models. We mimic what is done in every state of the art paper on the problem in which only a single hidden layer is used instead of relying on deep learning.
- **Learning Rate Decay Interval**: indicates the interval (in number of epochs) in which the learning rate of the neural network models should be halved. We perform, therefore, a simulated annealing process in which in the beginning the model can explore the search space more freely without getting stuck in local minima and then later on it can fine-tune its parameters.
- **Lookback**: when training a time sensitive model, the samples from the good samples are kept, while the ones from the bad ones are limited by the Number of Failing Samples parameter. To prevent differences in behavior due to the different time intervals, we instead divide the sequences from all samples into sequences of length l, where l is the lookback. So, a sequence of length m is divided into $m - l$ sequences of length l, each starting at a point $i \in \{1, \ldots, m - l + 1\}$.
- **Vote Count**: the number of consecutive samples of a disk that should be evaluated when deciding if it is in a soon to fail state or not. It corresponds to the value v in Subsection 3.3.6.
- **Vote Threshold**: the minimum ratio of samples in an interval that need to be classified by a model as failing before the disk is flagged as in a soon to fail state. It corresponds to the value τ in Subsection 3.3.6.
- **Voting Algorithm**: which of the two voting algorithms to use. It can be either the Class-Based Voting Algorithm or the Score-Based Voting Algorithm.

4 Results

4.1 Dataset

We use a dataset from a datacenter of Baidu Inc. It is a subset of the dataset used in [8].

The reason for using only a subset is that due to GDPR limitations we do not have access to the Chinese website where the dataset is hosted. So, we use a sample from it made available on Kaggle [1].

Despite only having access to a subset, our dataset includes all the 433 failing disks on the original dataset, which is the most important aspect since in general this is the class with a much smaller amount of members. It also includes 5317 good disks.

In the dataset, for each disk a sample is taken every hour. The observation period is of 7 days for good disks and 20 days for bad ones. In total there are more than a million SMART attribute samples.

The model of every disk is the same. This model makes available 12 different SMART attributes. In the dataset they are already normalized to be between -1 and 1. When including the change rates we compute we can therefore have up to 24 different features for each sample.

4.2 Binary Models

We first perform the experiments using the classical of binary classification, that is of using only two health status levels. The parameters for these experiments are specified

[1] `https://www.kaggle.com/datasets/drtycoon/hdds-dataset-baidu-inc`

Model	FAR(%)	FDR(%)	TIA(h)	TIA SD(h)
BPNN	1.69	98.46	368.5	145.8
RNN	1.57	98.46	356.4	145.8
LSTM	1.50	98.46	356.7	145.6
Classification Tree	1.13	99.23	353.7	147.8
Regression Tree	0.63	99.23	348.3	147.3

Table 4.1: Results for binary models with standard parameters

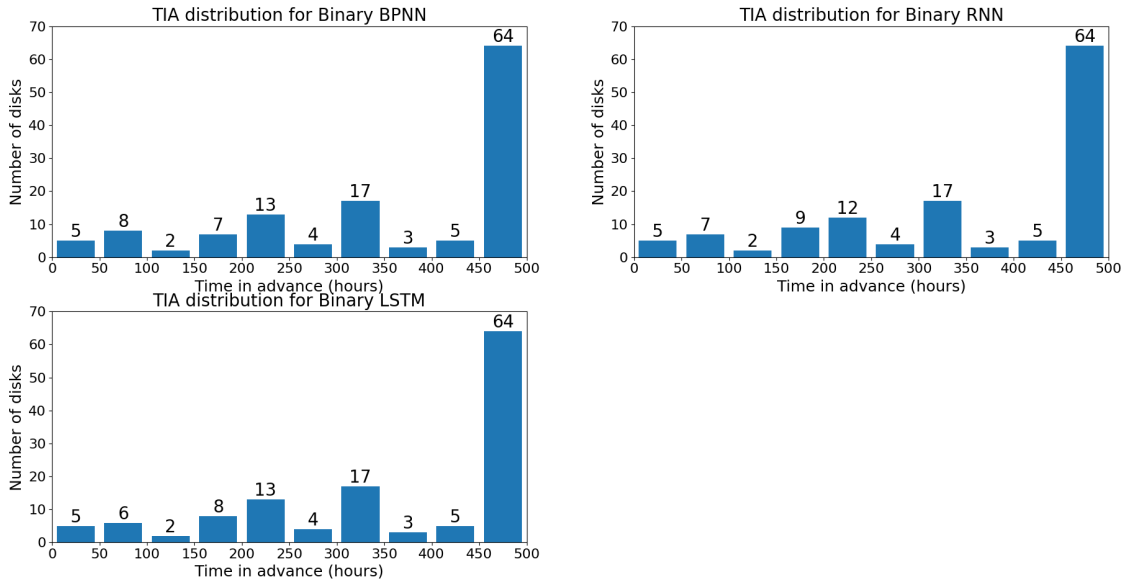

Figure 4.1: Time In Advance distribution for binary networks

on Appendix A. The results for each model is listed on Table 4.1.

We also show the distribution of the time in advance with which the failing disks were correctly detected in the histograms of Figures 4.1 and 4.2.

The analysis we can make of these results is firstly that the three network models behave similarly. LSTM is a little better than the RNN, which is a little better than BPNN.

However, the fact that the three achieve the same FDR indicates that the disks they cannot correctly classify is due more to the nature of the problem than to incorrect hyperparameters. The cause of these failures that none of the models could predict may be some events that cannot be monitored using SMART attributes such as current surges.

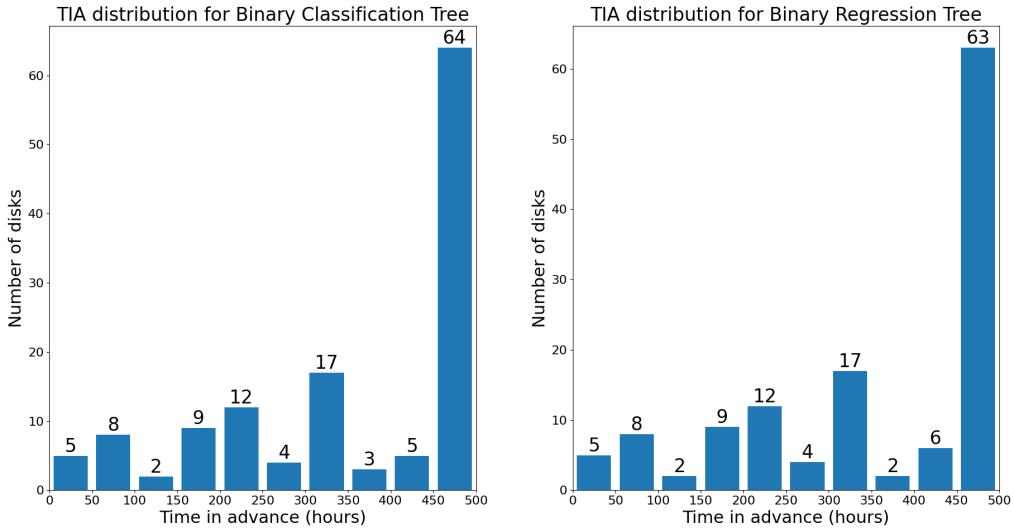

Figure 4.2: Time In Advance distribution for binary decision trees

The classification tree performs better than any of the network based models and is only beaten by the regression tree. This indicates that these older, simpler models are still very powerful and should not be overlooked.

The distribution of the TIA for the different models signals that all of them are capable of detecting signals of problems hundreds of hours in advance In practical terms, this means that the people responsible for managing the datacenters will have plenty of time to swap the failing disks and prevent any data loss or disruption of service.

The smallest value of the TIA for every model was 19 hours. This implies that no rushing is needed when a failure is predicted. The main risk lies instead on the disks that have not been detected.

The capacity of such models to be highly sensitive to failing disks in order to detect problems much before they occur while keeping an FAR below 2% proves that all of them are powerful methods to tackle the problem at hand.

We can compare our results to other experiments performed on the same dataset in [8]. Our results are similar to theirs, who obtained a better FAR of 1.14% but a slightly worse FDR of 97.69%.

The advantage of our approach is that we can directly compare the performance of different models without introducing bias due to different datasets or different preprocessing

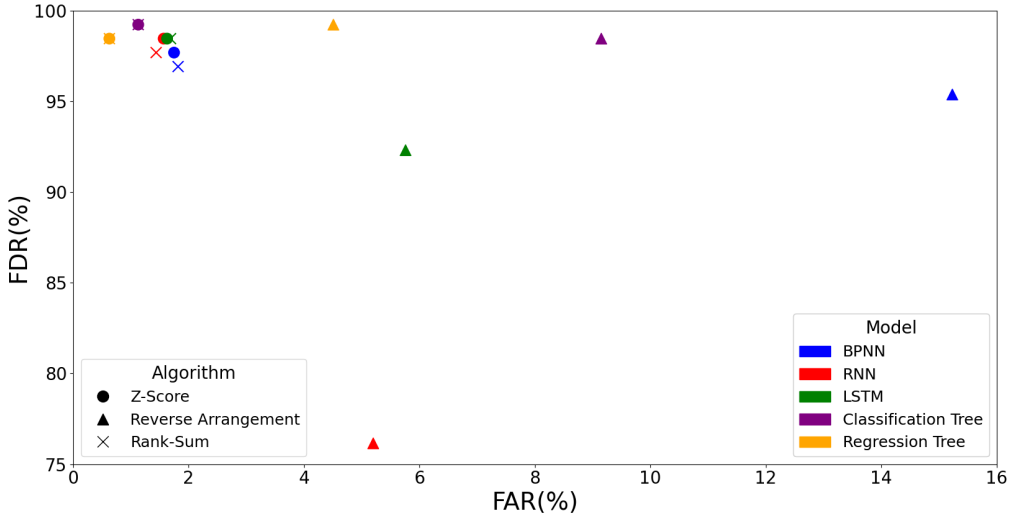

Figure 4.3: Results for different feature selection algorithms for binary models for Feature Count = 15

steps.

On the next few pages we will describe the impact of the feature selection algorithms as well as how the voting process can improve the results. Finally, we will show that the inclusion of the change rates of the SMART attributes improves the performance of every model.

4.2.1 Feature Selection Algorithms

We have trained our different models using the different feature selection algorithms described on Subsection 3.3.2. We still used the default parameters presented on Appendix A, with the only modifications being on the `feature_count` that was set to 15 and, of course, on `feature_selection_algorithm`. The results are displayed on Figure 4.3.

The most striking pattern we can observe is that the Reverse Arrangement algorithm had a worse performance than the other two on all scenarios we have tested.

Moreover, the Z-Score algorithm is slightly better than the Rank-Sum test for the network-based models. However, the result of these two algorithms is the same for our decision trees.

Model	FAR(%)	FDR(%)	TIA(h)	TIA SD(h)
BPNN	1.69	98.46	362.7	145.6
RNN	1.50	98.46	362.5	145.8
LSTM	1.50	98.46	361.3	146.2
Classification Tree	0.94	99.23	353.7	147.8
Regression Tree	0.63	98.46	353.3	148.2

Table 4.2: Results for binary models with Vote Count set to 7

If we observe which features are kept by the three algorithms, we see that 11 of them are common to all of them. These are features such as Reallocated Sector Count, Temperature and Seek Error Rate, which is to be expected since these can easily indicate failure.

Surprisingly, between the Rank-Sum and the Reverse Arrangement algorithms, only 3 of the 15 features change, but this is already enough to generate completely different results. Two of the features that appear in the selection done by the Reverse Arrangement but do not appear on Rank-Sum one are change rate features: the Current Pending Sector Change Rate and the Seek Error Rate Change Rate. The third one, High Fly Writes, also do not appear in the Z-Score selection.

This indicates that the change rate features allow the models to learn the classification problem more efficiently. We will explore this further on Subsection 4.2.3.

4.2.2 Voting

To observe how the voting process can influence the performance of the models, we ran our experiments with the default configuration but using 7 votes instead of only 1. We still kept the vote threshold at 0.5. As discussed on Subsection 3.3.6, for the binary models we are presenting, only the Class Based Voting Algorithm can be used.

The results of this experiment are displayed on Table 4.2.

As we can see, the results are very similar to the original ones. For the RNN model, there was a slight decrease on the FAR. For the Regression Tree, the FDR decreased a bit and became equal to the ones for the network based models.

Model	FAR(%)	FDR(%)	TIA(h)	TIA SD(h)
BPNN	1.69	98.46	362.5	145.5
RNN	1.38	98.46	362.3	145.4
LSTM	0.75	98.46	362.7	145.6
Classification Tree	0.94	99.23	352.3	147.6
Regression Tree	0.56	97.69	351.1	147.9

Table 4.3: Results for binary models with Vote Count set to 7

Moreover, on all models the TIA decreased a little bit. This was expected due to the fact that now more than one sample is not enough to classify a disk as failing.

The fact that the FDR didn't change a lot was to be expected given the results displayed on Figures 4.1 and 4.2. Usually, a failing disk will start producing samples indicating it much before it crashes.

Therefore, the probability that a sequence with many samples that can be predicted as failing is high. Thus, the decrease of the FDR is expected to be small, which is the result we obtained.

Now, if we keep the vote count to 7 but increase the vote threshold to 0.8, we obtain the results shown on Table 4.3.

Again we observe a further decrease on the value of the FAR, the FDR and the TIA for some models. The Regression Tree is able to achieve an FAR as small as 0.56% which is the smallest value we found for it.

The most interesting approach of using a voting algorithm is that the value of the vote count and threshold can be modified without needing to retrain the model, which is not the case when changing the feature count or the feature selection algorithm. Therefore, voting is a powerful and fast to use tool to fine tune the FAR-FDR trade off.

4.2.3 Change Rate Impact

The next step was to observe how including the change rate of the original SMART attributes affects the performance of our models. So, we trained our models without computing the change rates. In order to remove any impact due to feature selection algorithms, we used all 12 SMART attributes.

Model	FAR(%)	FDR(%)	TIA(h)	TIA SD(h)
BPNN	2.13	98.46	363.7	145.1
RNN	1.57	98.46	357.4	145.9
LSTM	2.07	95.38	361.8	143.6
Classification Tree	1.07	99.23	354.7	147.8
Regression Tree	1.07	97.69	354.7	147.8

Table 4.4: Results for binary models with Vote Count set to 7

The results are displayed on table 4.4

Most notably, every model had a worse performance compared to the original version using the change rates. This shows that the change rate we computed carries information than can be used by the models to better learn the problem.

Moreover, it is interesting to see that the models that had the best performance were the two Decision Tree ones with information about the change rates. The fact that they both Classification and Regression Trees are time insensitive implies that we can encode useful information about the problem in the form of change rates.

5 Discussion

Discuss the results. What is the outcome of your experimetns?

6 Conclusion

Summarize the thesis and provide a outlook on future work.

A Configuration Files

Below we have the standard configurations for each of the models we trained. When mentioning the results, if the value of a parameter is not mentioned, then it is because the value indicated below is used.

Listing A.1: Default configuration file for a Binary BPNN

```
[dataset]
seed = 0
data_file = "data/baidu-dataset.csv"
number_of_failing_samples = 12

[preprocessing]
change_rate_interval = 1
feature_count = 19
feature_selection_algorithm = "z_score"
health_status_algorithm = "discrete"
good_bad_ratio = 0.6

[model]
model_type = "NN"
model = "BinaryBPNN"
health_status_count = 2
hidden_nodes = 30
epoch_count = 1500
learning_rate = 0.1
evaluate_interval = 1500
lr_decay_interval = 100000

[vote]
vote_count = 1
vote_threshold = 0.5
```

Listing A.2: Default configuration file for a Binary RNN

```
[dataset]
seed = 0
data_file = "data/baidu-dataset.csv"
number_of_failing_samples = 12

[preprocessing]
change_rate_interval = 1
feature_count = 19
feature_selection_algorithm = "z_score"
health_status_algorithm = "discrete"
good_bad_ratio = 20.0

[model]
model_type = "NN"
model = "BinaryRNN"
health_status_count = 2
hidden_nodes = 10
epoch_count = 700
learning_rate = 0.1
evaluate_interval = 700
lr_decay_interval = 100
lookback = 6

[vote]
vote_count = 1
vote_threshold = 0.5
```

Listing A.3: Default configuration file for a Binary LSTM

```
[dataset]
seed = 0
data_file = "data/baidu-dataset.csv"
number_of_failing_samples = 12

[preprocessing]
change_rate_interval = 1
feature_count = 19
feature_selection_algorithm = "z_score"
health_status_algorithm = "discrete"
good_bad_ratio = 15.0

```

```
[model]
model_type = "NN"
model = "BinaryLSTM"
health_status_count = 2
hidden_nodes = 10
epoch_count = 700
learning_rate = 0.1
evaluate_interval = 700
lr_decay_interval = 100
lookback = 6

[vote]
vote_count = 1
vote_threshold = 0.5
```

Listing A.4: Default configuration file for a Multi-Level BPNN

```
[dataset]
seed = 0
data_file = "data/baidu-dataset.csv"
number_of_failing_samples = 12

[preprocessing]
change_rate_interval = 1
feature_count = 19
feature_selection_algorithm = "z_score"
health_status_algorithm = "discrete"
good_bad_ratio = 0.6

[model]
model_type = "NN"
model = "MultiLevelBPNN"
health_status_count = 4
hidden_nodes = 30
epoch_count = 1500
learning_rate = 0.1
evaluate_interval = 1500
lr_decay_interval = 100000

[vote]
vote_count = 1
vote_threshold = 0.5
```

Listing A.5: Default configuration file for a Multi-Level RNN

```
[dataset]
seed = 0
data_file = "data/baidu-dataset.csv"
number_of_failing_samples = 12

[preprocessing]
change_rate_interval = 1
feature_count = 19
feature_selection_algorithm = "z_score"
health_status_algorithm = "discrete"
good_bad_ratio = 20.0

[model]
model_type = "NN"
model = "MultiLevelRNN"
health_status_count = 4
hidden_nodes = 10
epoch_count = 700
learning_rate = 0.1
evaluate_interval = 700
lr_decay_interval = 100
lookback = 6

[vote]
vote_count = 1
vote_threshold = 0.5
```

Listing A.6: Default configuration file for a Multi-Level LSTM

```
[dataset]
seed = 0
data_file = "data/baidu-dataset.csv"
number_of_failing_samples = 12

[preprocessing]
change_rate_interval = 1
feature_count = 19
feature_selection_algorithm = "z_score"
health_status_algorithm = "discrete"
good_bad_ratio = 10.0

```

```
[model]
model_type = "NN"
model = "MultiLevelLSTM"
health_status_count = 4
hidden_nodes = 10
epoch_count = 700
learning_rate = 0.1
evaluate_interval = 700
lr_decay_interval = 100
lookback = 6

[vote]
vote_count = 1
vote_threshold = 0.5
```

Listing A.7: Default configuration file for a Classification Tree

```
[dataset]
seed = 0
data_file = "data/baidu-dataset.csv"
number_of_failing_samples = 12

[preprocessing]
change_rate_interval = 1
feature_count = 19
feature_selection_algorithm = "z_score"
health_status_algorithm = "discrete"
good_bad_ratio = 0.6

[model]
model_type = "TREE"
tree_type = "CLASSIFICATION"
health_status_count = 2
criterion = "GINI"
max_depth = 5
min_samples_leaf = 5

[vote]
vote_count = 1
vote_threshold = 0.5
```

Listing A.8: Default configuration file for a Regression Tree

```
[dataset]
seed = 0
data_file = "data/baidu-dataset.csv"
number_of_failing_samples = 12

[preprocessing]
change_rate_interval = 1
feature_count = 19
feature_selection_algorithm = "z_score"
health_status_algorithm = "continuous"
good_bad_ratio = 1.0

[model]
model_type = "TREE"
tree_type = "REGRESSION"
health_status_count = 2
criterion = "SQUARED_ERROR"
max_depth = 5
min_samples_leaf = 5

[vote]
vote_count = 1
vote_threshold = 0.5
```

Bibliography

[1] Steven Cherry. "Edholm's law of bandwidth". In: *IEEE spectrum* 41.7 (2004), pp. 58–60 (cit. on p. 1).

[2] Gordon E Moore. "Cramming more components onto integrated circuits". In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85 (cit. on p. 1).

[3] Aarne Mammela and Antti Anttonen. "Why Will Computing Power Need Particular Attention in Future Wireless Devices?" In: *IEEE Circuits and Systems Magazine* 17.1 (2017), pp. 12–26. DOI: 10.1109/MCAS.2016.2642679 (cit. on p. 1).

[4] Ling Qian et al. "Cloud computing: An overview". In: *IEEE international conference on cloud computing*. Springer. 2009, pp. 626–631 (cit. on pp. 1, 2).

[5] *Microsoft Datacenters*. https://datacenters.microsoft.com/. Accessed: 2025-05-14. Microsoft (cit. on pp. 1, 2).

[6] Guosai Wang, Lifei Zhang, and Wei Xu. "What Can We Learn from Four Years of Data Center Hardware Failures?" In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2017, pp. 25–36 (cit. on p. 3).

[7] *Communicating With Your SSD - Understanding SMART Attributes*. Tech. rep. Accessed: 2025-05-16. 2013, pp. 24–25 (cit. on pp. 2, 5).

[8] Bingpeng Zhu et al. "Proactive drive failure prediction for large scale storage systems". In: *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*. IEEE. 2013, pp. 1–5 (cit. on pp. 4, 5, 16, 27, 31, 33, 52, 54).

[9] Chang Xu et al. "Health status assessment and failure prediction for hard drives with recurrent neural networks". In: *IEEE Transactions on Computers* 65.11 (2016), pp. 3502–3508 (cit. on pp. 4, 5, 19, 20, 33, 34, 46).

[10] Jing Shen et al. “Random-forest-based failure prediction for hard disk drives”. In: *International Journal of Distributed Sensor Networks* 14.11 (2018) (cit. on pp. 4–6, 29).

[11] Jing Li et al. “Hard drive failure prediction using classification and regression trees”. In: *2014 44th annual ieee/ifip international conference on dependable systems and networks*. IEEE. 2014, pp. 383–394 (cit. on pp. 5, 12, 20, 27, 33).

[12] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. “Characterizing cloud computing hardware reliability”. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC ’10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 193–204 (cit. on p. 5).

[13] Claude E Shannon. “A mathematical theory of communication”. In: *The Bell system technical journal* 27.3 (1948), pp. 379–423 (cit. on p. 9).

[14] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton:(Project Para)*. Cornell Aeronautical Laboratory, 1957 (cit. on p. 13).

[15] Claude Lemaréchal. “Cauchy and the gradient method”. In: *Doc Math Extra* 251.254 (2012), p. 10 (cit. on p. 15).

[16] Shun-ichi Amari. “Backpropagation and stochastic gradient descent method”. In: *Neurocomputing* 5.4-5 (1993), pp. 185–196 (cit. on p. 16).

[17] Dave Steinkraus, Ian Buck, and Patrice Y Simard. “Using GPUs for machine learning algorithms”. In: *Eighth International Conference on Document Analysis and Recognition (ICDAR’05)*. IEEE. 2005, pp. 1115–1120 (cit. on p. 16).

[18] MI Jordan. *Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986*. Tech. rep. California Univ., San Diego, La Jolla (USA). Inst. for Cognitive Science, 1986 (cit. on pp. 17, 18).

[19] Juan Manuel Espinosa-Sanchez, Alex Gomez-Marin, and Fernando de Castro. “The Importance of Cajal’s and Lorente de Nó’s Neuroscience to the Birth of Cybernetics”. In: *The Neuroscientist* 31.1 (2025), pp. 14–30 (cit. on p. 17).

[20] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133 (cit. on p. 18).

[21] Jeffrey L Elman. “Finding structure in time”. In: *Cognitive science* 14.2 (1990), pp. 179–211 (cit. on p. 18).

[22] Kanchan M Tarwani and Swathi Edem. "Survey on recurrent neural network in natural language processing". In: *Int. J. Eng. Trends Technol* 48.6 (2017), pp. 301–304 (cit. on pp. 19, 22).

[23] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 19).

[24] Satya Prakash Yadav et al. "Survey on machine learning in speech emotion recognition and vision systems using a recurrent neural network (RNN)". In: *Archives of Computational Methods in Engineering* 29.3 (2022), pp. 1753–1770 (cit. on p. 22).

[25] Sepp Hochreiter. "The vanishing gradient problem during learning recurrent neural nets and problem solutions". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116 (cit. on p. 22).

[26] Yoshua Bengio, Paolo Frasconi, and Patrice Simard. "The problem of learning long-term dependencies in recurrent networks". In: *IEEE international conference on neural networks*. IEEE. 1993, pp. 1183–1188 (cit. on pp. 22, 23).

[27] Muhammad Raheel Raza, Walayat Hussain, and José Maria Merigó. "Cloud sentiment accuracy comparison using RNN, LSTM and GRU". In: *2021 Innovations in intelligent systems and applications conference (ASYU)*. IEEE. 2021, pp. 1–5 (cit. on p. 23).

[28] Simone A Ludwig. "Comparison of time series approaches applied to greenhouse gas analysis: ANFIS, RNN, and LSTM". In: *2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE. 2019, pp. 1–6 (cit. on p. 23).

[29] Zheng Hu, Jiaojiao Zhang, and Yun Ge. "Handling vanishing gradient problem using artificial derivative". In: *IEEE Access* 9 (2021), pp. 22371–22377 (cit. on p. 23).

[30] Sk M Rahman et al. "Nonintrusive reduced order modeling framework for quasigeostrophic turbulence". In: *Physical Review E* 100.5 (2019), p. 053306 (cit. on p. 24).

[31] Shengdong Zhang et al. "Deep learning on symbolic representations for large-scale heterogeneous time-series event prediction". In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2017, pp. 5970–5974 (cit. on pp. 25, 26).

[32] Anwesha Das et al. "Desh: deep learning for system health prediction of lead times to failure in hpc". In: *Proceedings of the 27th international symposium on high-performance parallel and distributed computing*. 2018, pp. 40–51 (cit. on p. 27).

[33] Xue Ying. "An overview of overfitting and its solutions". In: *Journal of physics: Conference series*. Vol. 1168. IOP Publishing. 2019 (cit. on p. 27).

[34] Erica Briscoe and Jacob Feldman. "Conceptual complexity and the bias/variance tradeoff". In: *Cognition* 118.1 (2011), pp. 2–16 (cit. on p. 28).

[35] Tin Kam Ho. "Random decision forests". In: *Proceedings of 3rd international conference on document analysis and recognition*. Vol. 1. IEEE. 1995, pp. 278–282 (cit. on p. 28).

[36] Anjaneyulu Babu Shaik and Sujatha Srinivasan. "A brief survey on random forest ensembles in classification model". In: *International Conference on Innovative Computing and Communications: Proceedings of ICICC 2018, Volume 2*. Springer. 2019, pp. 253–260 (cit. on p. 28).

[37] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Machine Learning* 20 (1995) (cit. on p. 30).

[38] Keiron O'shea and Ryan Nash. "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458* (2015) (cit. on p. 31).

[39] Osman Semih Kayhan and Jan C van Gemert. "On translation invariance in cnns: Convolutional layers can exploit absolute spatial location". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 14274–14285 (cit. on p. 31).

[40] Xiaoyi Sun et al. "System-level hardware failure prediction using deep learning". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6 (cit. on p. 32).

[41] Vipin Kumar and Sonajharia Minz. "Feature selection". In: *SmartCR* 4.3 (2014), pp. 211–229 (cit. on p. 39).

[42] Joseph F Murray et al. "Machine Learning Methods for Predicting Failures in Hard Drives: A Multiple-Instance Application." In: *Journal of Machine Learning Research* 6.5 (2005) (cit. on p. 40).

[43] Julius S Bendat and Allan G Piersol. *Random data: analysis and measurement procedures*. John Wiley & Sons, 2011 (cit. on p. 41).

[44] Thomas W MacFarland and Jan M Yates. "Mann–whitney u test". In: *Introduction to nonparametric statistics for the biological sciences using R*. Springer, 2016, pp. 103–132 (cit. on p. 42).

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, August 21, 2025

Miguel Vieira Pereira