ETH
Swiss Federal Institute of Technology Zurich

Seminar for Statistics

Department of Mathematics

Master Thesis Summer 2019

Vladimir Fomin

The Effect of Random Rotation Upscaling on Neural Networks and Gradient Boosted Trees

Submission Date: August 12th 2019

Adviser: Dr. Markus Kalisch

To my family:
Alfira, Olga and Valentin

With special thanks to:
Carla

Abstract

Gradient boosted trees and neural networks are currently two of the most successful and used statistical methods. As a result, they consistently occupy the top leaderboards on Kaggle challenges, which is a platform where machine learning hobbyists and professionals tackle problems posed by the industry and scientific community. The goal of this thesis is to provide an overview of the theory behind these methods as well as to explore their strengths and differences in practical applications.

In an attempt to examine these differences, a simulation study was performed which yields an interesting idea. At the core of this idea are random rotations which allow the upscaling of low dimensional data into a higher dimensional space while preserving some of its properties.

This may seem counterintuitive at first since upscaling data usually leads to a bigger search space and additional noise, but the result surprisingly could suggest the opposite if certain conditions are met. We will delve deeper into this idea in the later chapters of this work.

Contents

List of Figures

Notation

- $\mathbb{N} \coloneqq \{1, 2, 3, \dots\}$.
- $\forall L \in \mathbb{N} : [L] \coloneqq \{1, \dots, L\}$.
- $\mathcal{C}^1(\mathbb{R}, \mathbb{R}) \coloneqq \{f : \mathbb{R} \to \mathbb{R} | f \text{ is continuously differentiable } \}$

Chapter 1

Introduction

The aim of this work is to explore gradient boosted trees as well as Neural Networks. We will dive into the theoretical foundation of both methods and their practical application in addition to their strengths and differences.

The last part will deal with the effect of random rotations on the training data provided to these statistical methods.

To lay the mathematical groundwork, we will cover the general setting and common machine learning algorithms such as stagewise modelling and gradient descent in this chapter. Using this foundation, the second chapter will deal with Neural Networks and gradient boosted trees in depth. To provide an additional practical view on both methods, the third chapter will be a short application of Keras and XGBoost, which implement Neural Networks and gradient boosted trees respectively. Lastly, we will perform a simulation study and explore the effect of random rotations on both these methods.

1.1 General Setting

In this section we will introduce the setting of the random variables Y, X and their sample space. Next, we will cover basic notions such as the loss, total loss and risk. The assumptions as well as the definitions in this chapter borrow from Sigrist [1].

We assume that we have a pair of random variables $(Y, X) \in \mathbb{R} \times \mathbb{R}^p$ with $p \in \mathbb{N}$, where Y is the response variable alongside its predictors X. Additionally, we assume that the marginal distribution of X and the conditional distribution of $Y|X$ are absolutely continuous with respect to either the Lebesgue measure, a counting measure, a mixture of both, or a product measure of the former measures.

This covers both regression and classification tasks as well as mixtures of the two. Furthermore, we define the sample space for a collection of realizations of these random variables.

Definition 1.1.0.1. *Let $(y, x) \in \mathcal{S}$ be a sample in the set of all realizations of (Y, X). Then we define:*

i) y is called the sample output, target or response.

ii) x is called the sample input or sample data.

iii) $\mathcal{S}$ *is called the sample space.*

Next, we define the loss and the total loss associated with a sample space $\mathcal{S}$. Furthermore, let f be some function with which a prediction is computed given a sample input.

Definition 1.1.0.2. *Let* $f : \mathbb{R}^p \to \mathbb{R}$ *be a function contained in some function space* Ω, $\mathcal{L} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}_+$ *a function and* $\mathcal{S}$ *some sample space. Then we define:*

i) f *is called a computing function.*

ii) $\mathcal{L}(y, f(x))$ *is called the loss of the sample (y,x) given the computing function* f*.*

iii) $\mathcal{L}_{\mathcal{S},f} \coloneqq \sum_{(y,x)\in\mathcal{S}} \mathcal{L}(y, f(x))$ *is called the total loss given the sample space* $\mathcal{S}$ *and computing function* f*.*

Finally, we introduce the risk and the corresponding empirical risk, since we will mostly deal with a discrete sample space.

Definition 1.1.0.3. *Let* $\mathcal{R}, \mathcal{R}_{\mathcal{S}} : \Omega \to \mathbb{R}$ *be functions, then we define the risk as:*

$$\mathcal{R}(f) \coloneqq \mathbb{E}_{Y,X}\left[\mathcal{L}(y, f(x))\right].$$

Furthermore, let $\mathcal{S}$ *be some sample space, then we call the following quantity the empirical risk:*

$$\mathcal{R}_{\mathcal{S}}(f) \coloneqq \frac{1}{|\mathcal{S}|} \sum_{(y,x)\in\mathcal{S}} \mathcal{L}(y, f(x)) = \frac{1}{|\mathcal{S}|}\mathcal{L}_{\mathcal{S},f}.$$

1.2 Stagewise Modelling

Stagewise modelling is a procedure by which we incrementally arrive at a quasi optimal computing function f^*. But what do we mean by quasi optimal?

Again, the notions in this section follow Sigrist [1]. Suppose we have some kind of statistical method defined by a function space Ω. An optimal solution given a sample space $\mathcal{S}$ and a loss $\mathcal{L}$ would be a function f^* which satisfies the following:

$$f^* \coloneqq \arg\min_{f\in\Omega} \mathcal{R}(f) = \arg\min_{f\in\Omega} \mathbb{E}_{Y,X}\left[\mathcal{L}(y, f(x))\right]. \tag{1.2.0.1}$$

The problem with this definition is that the optimizer f^* most of the time cannot be found analytically and as a result, we must resort to a numerical approximation. This is why we refer to the solution of stagewise modelling as quasi optimal.

To compute this approximation, we start off by guessing a reasonable candidate $f_0 \in \Omega$. Again, the term reasonable may differ depending on the problem. Let $I \in \mathbb{N}$ be the number of stages or iterations. Then we define:

$$\forall i \in \{0, \dots, I\} : f_{i+1} \coloneqq f_i + \tilde{f}, \tag{1.2.0.2}$$

where $\tilde{f}$ minimizes the following quantity approximately:

$$\tilde{f} \approx \arg\min_{f \in \Omega} \mathcal{R}(f_i + f). \tag{1.2.0.3}$$

What exactly is the difference between equations 1.2.0.1 and 1.2.0.3? Any optimal solution f^* may not be representable by addition of f_i and any other function $f \in \Omega$. This implies that depending on the problem f^* and $\tilde{f}$ are different.

Sadly, even the altered version 1.2.0.3 is often times not solvable analytically. Still, if we can find a good approximation of equation 1.2.0.3 and combine it with a good initial guess f_0, we can proceed in an incremental or stagewise manner to the optimal solution. Note that there may numerical problems such as getting trapped in local minima and other phenomena.

Also, we defined the stagewise method by means of the plus operator, but really it is applicable with any operator given that we can approximate equation 1.2.0.3 well enough. For instance, gradient descent which is covered in the next section is in essence a stagewise method and we will use it to adjust the weights and biases of Neural Networks. This adjustment can be thought of as a different operator than simply adding the values of computing functions.

1.3 Gradient Descent

As previously mentioned, gradient descent is a form of a stagewise modelling approach. Furthermore, it is also one of the most widely used machine learning algorithms due to its simplicity and relatively mild requirements.

The key requisite for this method is the differentiability of the loss and that the derivative is integrable with respect to the probability measure of (Y, X). This allows us to interchange the derivation operator with the integral in the expectation. Henceforth, we will assume that the loss fulfills these conditions whenever we refer to gradient descent.

Another intricacy is that we need to compute the gradient of the loss function, but what is that gradient exactly? Suppose that we have some computing function f. Usually f is described by its meta-parameters specific to its enclosing function space Ω. For example, if we are dealing with regression trees that might be the depth of the tree or in the case of lasso regression the value of the regularization parameter λ.

So in order to apply gradient descent, we have to be able to derivate the computing function f regarding its meta-parameters. In that case, the meta-parameters and the gradient ∇ are defined as follows:

$$\begin{aligned} \mathcal{I} &\coloneqq \{i : i \text{ corresponds to some meta-parameter of the computing function } f\} \\ \lambda_{i \in \mathcal{I}} &\coloneqq \text{ the value of the associated meta-parameter} \end{aligned}$$

$$\nabla \coloneqq \begin{pmatrix} \frac{\partial}{\partial \lambda_1} \\ \vdots \\ \frac{\partial}{\partial \lambda_{|\mathcal{I}|}} \end{pmatrix}$$

The gradient provides us with an abstract direction regarding the parameter space of the computing function f. It always points in the direction of the steepest ascent. Though, since we are interested in minimizing the risk, we flip the sign. Additionally, we introduce a new regularization parameter γ corresponding to the step size. Gradient descent is then described by the formula:

$$\begin{aligned} f_{n+1} &= f_n - \gamma \cdot \nabla \mathcal{R}(f) \\ &= f_n - \gamma \cdot \nabla \mathbb{E}_{Y,X} \left[\mathcal{L}(y, f(x)) \right] \\ &\overset{i)}{=} f_n \underbrace{- \gamma \cdot \mathbb{E}_{Y,X} \left[\nabla \mathcal{L}(y, f(x)) \right]}_{\star} . \end{aligned}$$

As mentioned above, ***i)*** holds due to the regularity conditions imposed on the loss $\mathcal{L}$. Notice the similarity to stagewise modeling in 1.2.0.2. Gradient descent chooses $\star$ as the approximation of the risk minimizer in 1.2.0.3. As such, $\star$ can be interpreted as a computing function in Ω itself. It is the computing function defined by parameters providing the steepest descent in risk when each parameter is added to its corresponding parameter defining the original function f.

Therefore, the plus sign in equation 1.2.0.3 corresponds to component wise addition of meta-parameters. Though, due to the flipped sign this operation manifests as a component wise subtraction.

Since gradient descent is a form of stagewise modelling and approximation is involved, numerical considerations have to be taken into account. Gradient descent may suffer from the following shortcomings:

i) The obtained minimum of the loss $\mathcal{R}(f)$ may not be the global minimum and hence could be arbitrarily far from the optimum.

ii) It is not clear if the algorithm converges at all and therefore the obtained result is not even guaranteed to be a minimum.

iii) The choice of the step size γ poses a problem in and of itself as it directly influences the convergence speed and behaviour. For instance, the algorithm may not advance at all if oscillation occurs.

Nevertheless, gradient descent's main advantage is the relatively minor condition imposed on the loss. Most loss functions are differentiable anyhow and those that are not can be approximated via splines for example.

Chapter 2

Neural Networks and Gradient Boosted Trees

The previous chapter finally allows us to introduce Neural Networks as well as gradient boosted trees. First, we will start by covering Neural Networks and the procedures to fit and train them. Then we will move on to gradient boosted trees.

2.1 Neural Networks

Neural Networks were developed as early as 1958 by Rosenblatt [2] under the name of perceptrons. Though, research stagnated soon after their inception as they were deemed impractical. Their main problem at the time was the lack of general computing power which was desperately needed in order to compute the gradient introduced in the previous section.

Later on, an algorithm named Backpropagation was conceived. This algorithm cannot be accredited to any one scientist directly as it was derived by multiple researchers in the early 60's. However, Werbos [3] was the first to realize in his doctoral thesis that Backpropagation could be used to efficiently compute the gradient of Neural Networks.

As computing power steadily rose, Neural Networks coupled with Backpropagation became increasingly practical. This lead to a downright boom when researchers realized that Neural Networks are very proficient in solving historically hard problems like image classification or speech recognition.

We will cover the most basic form of Neural Networks as well as Backpropagation in this section. For a more visual presentation of the following definitions as well as Backpropagation, I recommend the excellent three part series by Sanderson [4] on Neural Networks. This section is heavily inspired by his work.

Definition 2.1.0.1. *We call the tuple $N = (n, w, b, f)$ a Neural Network of size L . We define its contents as follows:*

i) $n \coloneqq (n(1), \ldots, n(L))$, *where* $\forall l \in [L] : n(l) \in \mathbb{N}$

We refer to $n(l)$ as the number of neurons, units or nodes in the network layer l.

ii) $w \coloneqq (w(1), \dots, w(L-1))$, *where* $\forall l \in [L-1] : w(l) \in \mathbb{R}^{n(l) \times n(l+1)}$

The $w(l)$ *are called weights or weight matrix of layer* l*.*

iii) $b \coloneqq (b(1), \dots, b(L-1))$, *where* $\forall l \in [L-1] : b(l) \in \mathbb{R}^{n(l)}$

This vector is called the bias of the layer l*.*

iv) $f \coloneqq (f_2, \dots, f_L)$, *where* $\forall l \in \{2, \dots, L\} : f_l \in \mathcal{C}^1(\mathbb{R}, \mathbb{R})$

Any of these functions is called an activation function of a given layer l*.*

Remark. The use of layer depth, number of neurons in each layer, weights as well as biases will become apparent later. But the use of different types of activation functions is a more subtle one. Having a non linear activation function is the reason why it makes sense to increase the layer depth L. If all activation functions would be linear, the whole network could be represented as a single linear transformation of input data, as combining linear operations results in a linear operation.

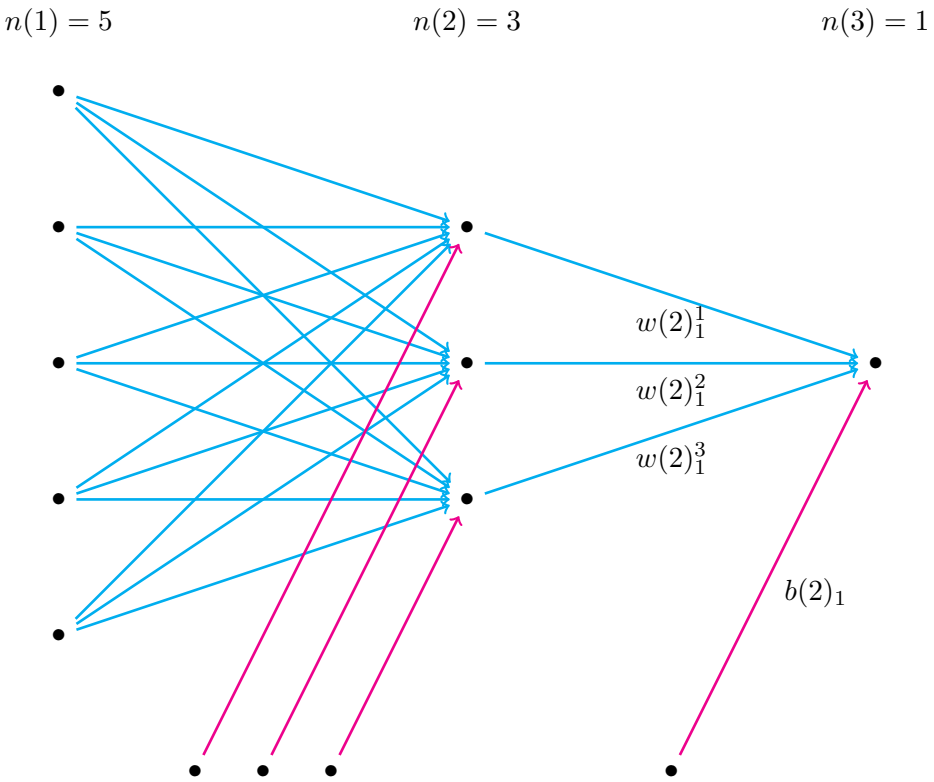

Figure 2.1.1: An example of a Neural Network of size three.

In figure 2.1.1 you can see a visualization of a Neural Network of size three. The weights are illustrated by blue and the biases by magenta arcs.

Clearly, a Neural Network induces a directed graph. This graph consists of layers where each node of a layer is fully interconnected with the adjacent layer. The weight matrix of some layer defines the edge weights of these interconnecting edges. Note that we index the weight matrix in the following way:

$$w(l)_i^j \coloneqq \begin{cases} \text{edge weight of layer } l \\ \text{connecting neuron } i \text{ of layer } l+1 \\ \text{and neuron } j \text{ of layer } l \end{cases}$$

Remember that our goal is to use these networks for tasks like regression and classification. Luckily, the directed graph structure provides a natural way to define a computing function. Though, to properly define this function we first need to define the so called *activations*.

Definition 2.1.0.2. *We define the activations $a(l)_{i\in[n(l)]}$ for some layer $l \in \{2,\dots,L\}$ as the solution of the recursion:*

$$a(l)_i \coloneqq f_l(z(l)) \coloneqq f_l(w(l-1)_i \cdot a(l-1) + b(l-1)_i)$$

Note that the $z(l)$ are merely defined to simplify the notation later on. The initial activations $a(1)$ needed to compute this recursion will be provided by the input data to the aforementioned computing function. Also, the multiplication above is shorthand notation for:

$$w(l-1)_i \cdot a(l-1) \coloneqq \sum_{j=1}^{n(l-1)} w(l-1)_i^j \cdot a(l-1)_j$$

Definition 2.1.0.3. *We call $N(x) : \mathbb{R}^{n(1)} \to \mathbb{R}^{n(L)}$ the computing function of the Neural Network N. This function is defined by the activations of the last layer with initial values $a(1) = x$ to the recursion in definition 2.1.0.2.*

$$N(x) \coloneqq a(L)$$

Hence, it makes sense to refer to the first layer as the *input layer* and to the last layer as the *output layer*. Also, sometimes we may refer to the computing function $N(x)$ of some Neural Network N directly when there is no ambiguity.

As we pass data into the Neural Network, we can see that it propagates through each layer by means of multiplication with the weight matrix, addition with the bias and finally component wise application of the activation functions. Note that the activation functions are mapping $\mathbb{R} \to \mathbb{R}$. Nevertheless we may apply them to vectors in a component wise manner when it is notationally convenient.

$$w(l) \cdot a(l) + b(l) = z(l+1)$$

$$\Updownarrow$$

$$\begin{pmatrix} w(l)_1^1 & \cdots & w(l)_1^{n(l)} \\ \vdots & \ddots & \vdots \\ w(l)_{n(l+1)}^1 & \cdots & w(l)_{n(l+1)}^{n(l)} \end{pmatrix} \cdot \begin{pmatrix} a(l)_1 \\ \vdots \\ a(l)_{n(l)} \end{pmatrix} + \begin{pmatrix} b(l)_1 \\ \vdots \\ b(l)_{n(l+1)} \end{pmatrix} = \begin{pmatrix} z(l+1)_1 \\ \vdots \\ z(l+1)_{n(l+1)} \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} f_{l+1}(z(l+1)_1) \\ \vdots \\ f_{l+1}(z(l+1)_{n(l+1)}) \end{pmatrix} =: f(z(l+1)) = a(l+1)$$

Finally, we can write the propagation of data through one layer of a Neural Network conveniently as:

$$f_{l+1}(w(l) \cdot a(l) + b(l)) = f_{l+1}(z(l+1)) = a(l+1).$$

2.1.1 Training Procedure and Backpropagation

In the previous section we covered what Neural Networks are, but it is not really clear how one would choose the number of layers and neurons in each layer as well as the weights, biases and activation functions.

The choice of layer depth, number of neurons in each layer and type of activation function differs widely for each task and each dataset. Therefore, these parameters are usually hand picked.

On the other hand, Gradient Descent offers a viable method to approximate the optimal values for the weights and biases. As mentioned in a previous section, to use Gradient Descent we need to define a set of meta-parameters on which to apply the gradient on as well as an initial computing function $f_0 \in \Omega$. In the case of Neural Networks, these meta-parameters are precisely the weights and biases. The initial computing function is usually chosen by assigning random values to the weights and biases.

$$\begin{aligned} \mathcal{I} &\coloneqq \{i : i \text{ corresponds to some edge weight or bias of the network } N\} \\ \lambda_{i \in \mathcal{I}} &\coloneqq \text{ the value of the associated weight or bias} \end{aligned}$$

Now, computing the gradient of some Neural Network N is extremely hard. Remember that the data flows through the network layer by layer and thus every edge affects all subsequent layers. Therefore, computing $\frac{\partial}{\partial \lambda_i} \mathcal{R}(N)$ for some i directly, quickly becomes impractical with growing network size. This difficulty is solved by Backpropagation. Though, to introduce Backpropagation we need a key abstraction first.

It may seem counterintuitive, but suppose that we are not interested in the influence of the edges, but in the influence of the neurons. That is:

$$\begin{aligned}\mathcal{J} &\coloneqq \{j : j \text{ corresponds to some neuron of the network } N\}\\ \varphi_{j\in\mathcal{J}} &\coloneqq \text{ the value of the associated neuron}\end{aligned}$$

Of course we cannot influence the activations of neurons directly and therefore we will substitute φ_j with λ_i later on.

Remark. An important thing to mention here is that the set $\mathcal{I}$ and $\mathcal{J}$ are arbitrarily ordered, while the neurons and edges in the Neural Network are split up into different layers. To simplify notation, we will assume that the index of some neuron or weight in $\mathcal{I}$ and $\mathcal{J}$ coincides with its index regarding the layer.

The first major component of Backpropagation is the following recursion. Let $l_0 \in \{3, \dots, L\}$ be some arbitrary layer and suppose that we want to find the influence of some neuron j^* located in some layer $l \in [l_0 - 2]$ on some other neuron m in the layer l_0.

$$\begin{aligned}\frac{\partial}{\partial\varphi_{j^*}} a(l_0)_m &= \frac{\partial}{\partial\varphi_{j^*}} f_{l_0}(z(l_0)_m)\\ &\overset{i)}{=} \dot{f}_{l_0}(z(l_0)_m) \cdot \frac{\partial}{\partial\varphi_{j^*}} [z(l_0)_m]\\ &= \dot{f}_{l_0}(z(l_0)_m) \cdot \frac{\partial}{\partial\varphi_{j^*}} [w(l_0-1)_m \cdot a(l_0-1) + b(l_0-1)_m]\\ &= \dot{f}_{l_0}(z(l_0)_m) \cdot \frac{\partial}{\partial\varphi_{j^*}} \left[\sum_{k=1}^{n(l_0-1)} w(l_0-1)^k_m \cdot a(l_0-1)_k + b(l_0-1)_m\right]\\ &\overset{ii)}{=} \dot{f}_{l_0}(z(l_0)_m) \cdot \left[\sum_{k=1}^{n(l_0-1)} w(l_0-1)^k_m \cdot \frac{\partial}{\partial\varphi_{j^*}} a(l_0-1)_k + \underbrace{\frac{\partial}{\partial\varphi_{j^*}} b(l_0-1)_{j^*}}_{=0}\right]\\ &= \dot{f}_{l_0}(z(l_0)_m) \cdot \left[\sum_{k=1}^{n(l_0-1)} w(l_0-1)^k_m \cdot \underbrace{\frac{\partial}{\partial\varphi_{j^*}} a(l_0-1)_k}_{\star}\right]\end{aligned}$$

Here, step ***i)*** follows from the chain rule and ***ii)*** since the bias is not influenced by neurons in preceding layers.

We may not have found the influence of the neuron j^* on the neuron m, but we reduced the problem to finding the influence of j^* on all neurons located in the layer $l_0 - 1$ due to the recursion in $\star$.

This calculation is visualized in figure 2.1.2. Again, as you can see the neuron j^* influences all subsequent neurons. Therefore, we need to sum up the influence of the neuron j^* regarding all neurons in the layer $l_0 - 1$.

Note that we only covered the case where the neuron j^* lies in the layers $\{1, \dots, l_0 - 2\}$, which means that it lies at least two layers apart from neuron m. If we continue with the

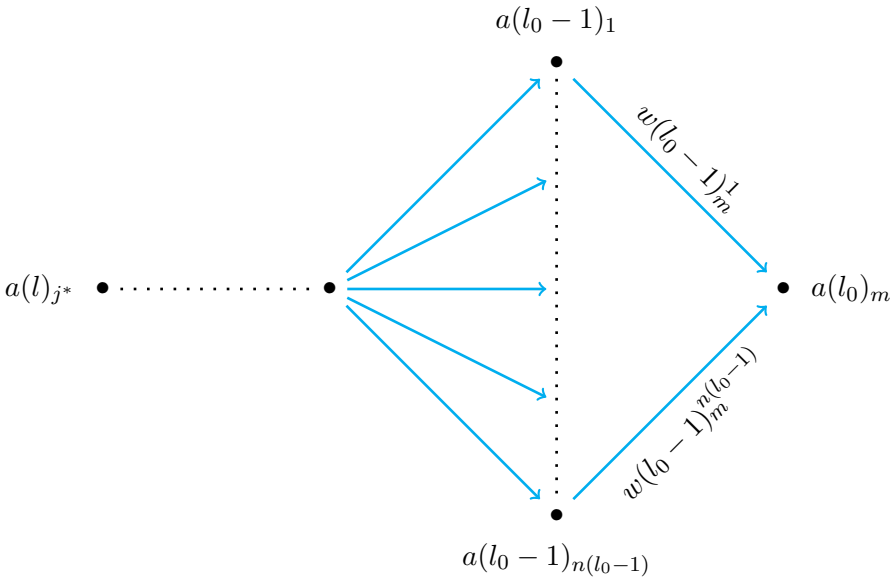

Figure 2.1.2: A visualization of the Backpropagation recursion (excluding biases).

recursion, we will arrive at the case where the given neuron m lies exactly one layer apart from j^*. We call this case the base case. Hence, suppose the influencer neuron j^* sits right behind the neuron m as visualized in figure 2.1.3. It follows:

$$\begin{aligned}
\frac{\partial}{\partial \varphi_{j^*}} a(l_0)_m &= \frac{\partial}{\partial \varphi_{j^*}} f_{l_0}(z(l_0)_m) \\
&= \dot{f}_{l_0}(z(l_0)_m) \cdot \left[\sum_{k=1}^{n(l_0-1)} w(l_0-1)^k_m \cdot \frac{\partial}{\partial \varphi_{j^*}} a(l_0-1)_k + \frac{\partial}{\partial \varphi_{j^*}} b(l_0-1)_{j^*} \right] \\
&= \dot{f}_{l_0}(z(l_0)_m) \cdot \begin{cases} w(l_0-1)^{j^*}_m \cdot \frac{\partial}{\partial \varphi_{j^*}} a(l_0-1)_{j^*} = w(l_0-1)^{j^*}_m, & \text{if influencer is a neuron} \\ \frac{\partial}{\partial \varphi_{j^*}} b(l_0-1)_{j^*} = 1, & \text{if influencer is a bias} \end{cases}
\end{aligned}$$

Using the recursion as well as the base case, we are now able to compute the influence of any neuron or bias on the outcome of the network. As such, we can also compute the influence of some neuron regarding the total loss of the network. Now, it only remains to substitute φ_j with λ_i.

For this, consider the base case again. We associate the neuron j^* with the edge $w(l_0-1)^{j^*}_m$ connecting it to the neuron m. This edge will be referred to as edge i^* in the following calculations:

$$\begin{aligned}
&i) \quad \frac{\partial}{\partial \varphi_{j^*}} a(l_0)_m = \dot{f}_{l_0}(z(l_0)_m) \cdot w(l_0-1)^{j^*}_m \\
&ii) \quad \frac{\partial}{\partial \lambda_{i^*}} a(l_0)_m = \dot{f}_{l_0}(z(l_0)_m) \cdot \left[\sum_{k=1}^{n(l_0-1)} \frac{\partial}{\partial \lambda_{i^*}} w(l_0-1)^k_m \cdot a(l_0-1)_k + \frac{\partial}{\partial \lambda_{i^*}} b(l_0-1)_{j^*} \right] \\
&\qquad\qquad = \dot{f}_{l_0}(z(l_0)_m) \cdot a(l_0-1)_{i^*}
\end{aligned}$$

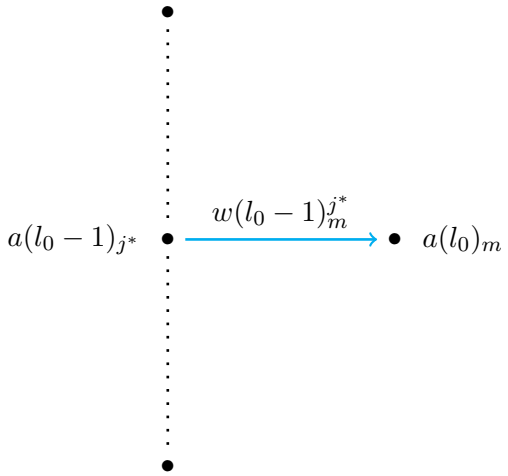

Figure 2.1.3: A visualization of the Backpropagation base case.

Solving for $\dot{f}_{l_0}(z(l_0)_m)$ and combining both equations yields the wanted substitution.

$$\frac{\partial}{\partial \lambda_{i^*}} a(l_0)_m = \frac{a(l_0-1)_{i^*}}{w(l_0-1)^{j^*}_m} \cdot \frac{\partial}{\partial \varphi_{j^*}} a(l_0)_m$$

Using this substitution and previous calculations, we are finally able to compute the influence of some weight on the risk of the network N.

$$\begin{aligned}
\nabla \mathcal{R}(N) &= \nabla \mathbb{E}_{Y,X}\left[\mathcal{L}(y, N(x))\right] \\
&\overset{i)}{=} \mathbb{E}_{Y,X}\left[\nabla \mathcal{L}(y, N(x))\right] \\
&= \mathbb{E}_{Y,X}\left[\left(\frac{\partial}{\partial \lambda_j}\mathcal{L}(y, N(x))\right)_{j\in\mathcal{I}}\right] \\
&\overset{ii)}{=} \mathbb{E}_{Y,X}\left[\left(\frac{\partial}{\partial f}\mathcal{L}(y, f)\Big|_{f=N(x)} \cdot \frac{\partial}{\partial \lambda_j} N(x)\right)_{j\in\mathcal{I}}\right] \\
&\overset{iii)}{=} \mathbb{E}_{Y,X}\left[\left(\frac{\partial}{\partial f}\mathcal{L}(y, f)\Big|_{f=N(x)} \cdot \frac{\partial}{\partial \lambda_j} a_x(L)\right)_{j\in\mathcal{I}}\right] \\
&\overset{iv)}{=} \mathbb{E}_{Y,X}\left[\left(\frac{\partial}{\partial f}\mathcal{L}(y, f)\Big|_{f=N(x)} \cdot \left(\frac{\partial}{\partial \lambda_j} a_x(L)_k\right)_{k\in[n(L)]}\right)_{j\in\mathcal{I}}\right] \\
&\overset{v)}{=} \mathbb{E}_{Y,X}\left[\left(\frac{\partial}{\partial f}\mathcal{L}(y, f)\Big|_{f=N(x)} \cdot \left(\frac{a(L-1)_j}{w(L-1)^s_k} \cdot \underbrace{\frac{\partial}{\partial \varphi_s} a_x(L)_k}_{\text{plug into recursion}}\right)_{k\in[n(L)]}\right)_{j\in\mathcal{I}}\right]
\end{aligned}$$

In step ***i)*** we assume that the loss fulfills the regularity conditions proposed in the section

about Gradient Descent. Then we use the chain rule in step ***ii)***. Also, here we refer to the activations regarding the input x as a_x in step ***iii)*** and honor the dimensionality of $a_x(L)$ in step ***iv)***. Finally, we use the aforementioned substitution in step ***v)*** and plug the resulting term in the described recursion. With all of these steps combined we are finished in describing the Backpropagation algorithm, but there are a few things left to mention.

In practice, we are not able to compute the risk directly and must settle with the empirical risk. The calculation above can be adjusted accordingly by replacing the expectation with the sample mean.

Also, we might notice that not only does the data propagate layer by layer, but also the influence of each edge. Coupled with the fact that we compute this influence in a recursive manner, the name Backpropagation is indeed fitting.

Furthermore, it may be apparent that due to the structure of the recursion we will reuse a lot of the partial derivatives. An efficient implementation might therefore analytically derivate the individual components and use them as placeholders. After doing the first pass through the network in this way, such an implementation could compute the related derivative for each equivalence class of placeholders exactly once.

Additionally, one does not usually compute the gradient for each training sample. In practice, the sample space $\mathcal{S}$ is split up into random batches similar to folds in Cross Validation. Then the gradient of the risk is only computed on some small batch of these samples. The network parameters are updated for each of these batches. Once all batches have been processed, a so called *epoch* has passed.

2.2 Gradient Boosted Trees

The aim of this section is to introduce gradient boosted trees. Though, first we will look into tree based methods in general. This section is based on the book *The Elements of Statistical Learning* by Hastie, Tibshirani and Friedman [5, pp. 266-279].

Trees are commonly used due to their simplicity. They partition the parameter space into rectangles and assign a value to each one. This value depends on whether we are dealing with regression or classification and the specific loss function used.

Definition 2.2.0.1. *We call the tuple $T = (\mathcal{P}, \Gamma)$ a tree of size M on p predictors. We define its contents as follows:*

i) $\mathcal{P} \coloneqq (R_1, \dots, R_M)$, *where* $\forall i \in [M] : R_i \subset \mathbb{R}^p$, $\forall j \neq i : R_i \cap R_j = \varnothing$ *and* $\bigcup_{i=1}^{M} R_i = \mathbb{R}^p$

We refer to the $\mathcal{P}$ as the regions or partition of the tree.

ii) $\Gamma \coloneqq (\gamma_1, \dots, \gamma_M)$, *where* $\forall i \in [M] : \gamma_i \in \mathbb{R}$

The individual γ_i is called the value of the region R_i.

Additionally, we call the function $T(x) : \mathbb{R}^p \to \mathbb{R}$ the computing function of the tree T. It is defined by:

$$T(x) \coloneqq \sum_{i=1}^{M} \gamma_i \cdot \mathbb{1}_{R_i}(x) = \sum_{i=1}^{M} \gamma_i \cdot \begin{cases} 1, & \text{if } x \in R_i \\ 0, & \text{if } x \notin R_i \end{cases}$$

Let us visualize the definition with the help of figure 2.2.1. We can see a partition of the space $\mathbb{R}^2$ which means we are dealing with two predictors. Each region contains samples which in turn define its assigned value. Here, the values are color coded and each color represents its own value γ_i.

If we are given new input data x, we can compute its corresponding value by returning the value of the region which contains x. In the figure that would be value γ_3 since x is located in R_3.

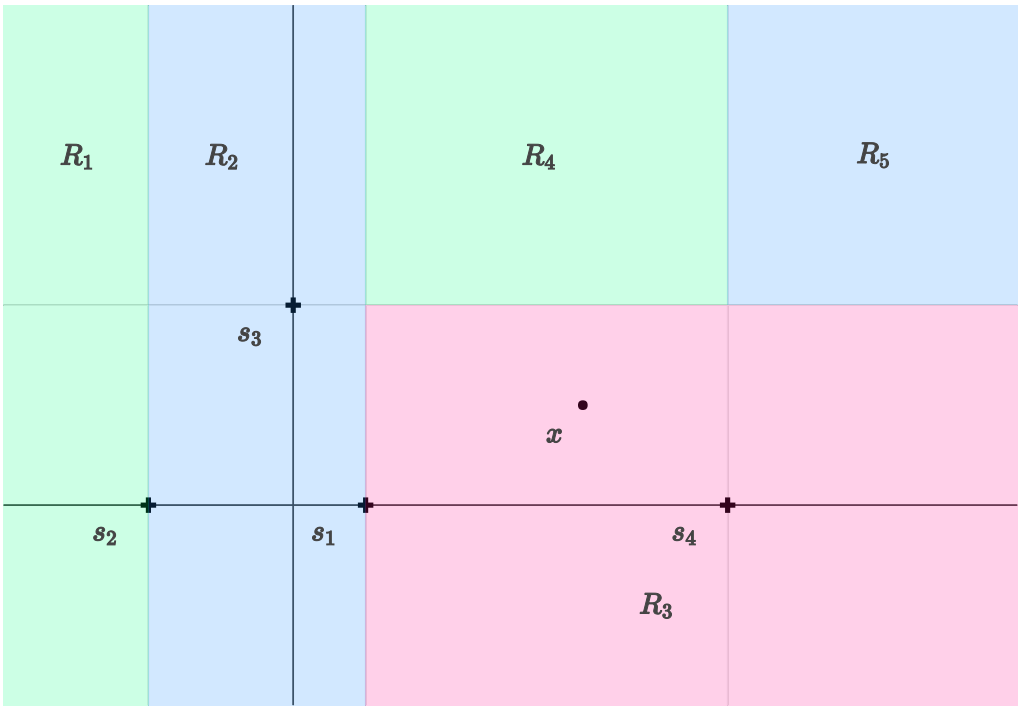

Figure 2.2.1: An example of tree of size 5 on 2 predictors.

Again, we assume the general setting. That is, we have some probability distribution for the variables (Y, X) and some loss function $\mathcal{L}$ which is differentiable and integrable regarding the probability measure.

How can we find an optimal tree or rather find values Γ^* and a partition $\mathcal{P}^*$ such that the following holds?

$$(\mathcal{P}^*, \Gamma^*) \coloneqq \arg\min_{(\mathcal{P},\Gamma)} \mathcal{R}(T) \tag{2.2.0.1}$$

As always, finding the optimal partition or solution to equation 2.2.0.1 is often impossible. If the amount of predictors p increases, the task gets substantially harder as we try to find the corresponding partition in p dimensions. As a result, we have to settle with a greedy approach.

2.2.1 Binary Splitting

Binary splitting offers a simple and greedy approach to find some viable partition. As the name suggests, the method splits the covariate space $\mathbb{R}^p$ in half, doing so in a greedy and iterative manner. A single iteration is then referred to as a binary split.

Such a binary split is performed by deciding on some predictor variable indexed by j and some split point $s \in \mathbb{R}$. These variables j and s induce a partition of the space $\mathbb{R}^p$ in the following way:

i) $R_{\leq}(j, s) \coloneqq \{x \in \mathbb{R}^p : x^j \leq s\}$

ii) $R_{>}(j, s) \coloneqq \{x \in \mathbb{R}^p : x^j > s\}$

Note that x^j corresponds to the j-th component of a sample while x_j would refer to the j-th sample.

It is clear that both sets form a partition of $\mathbb{R}^p$. Also, we can proceed by applying the same idea on the induced sets $R_{\leq}$ and $R_{>}$. This will create an increasingly fine grid.

One such grid can be seen in figure 2.2.1. The corresponding splitting procedure can then be visualized by a tree which is presented in figure 2.2.2. Each non leaf node of this tree represents an inequality which we determined during splitting. If we follow the arcs down from the root, we eventually arrive some partition region R_i. Coincidentally, this allows us to calculate the value of new data based on the individual inequalities present at each root.

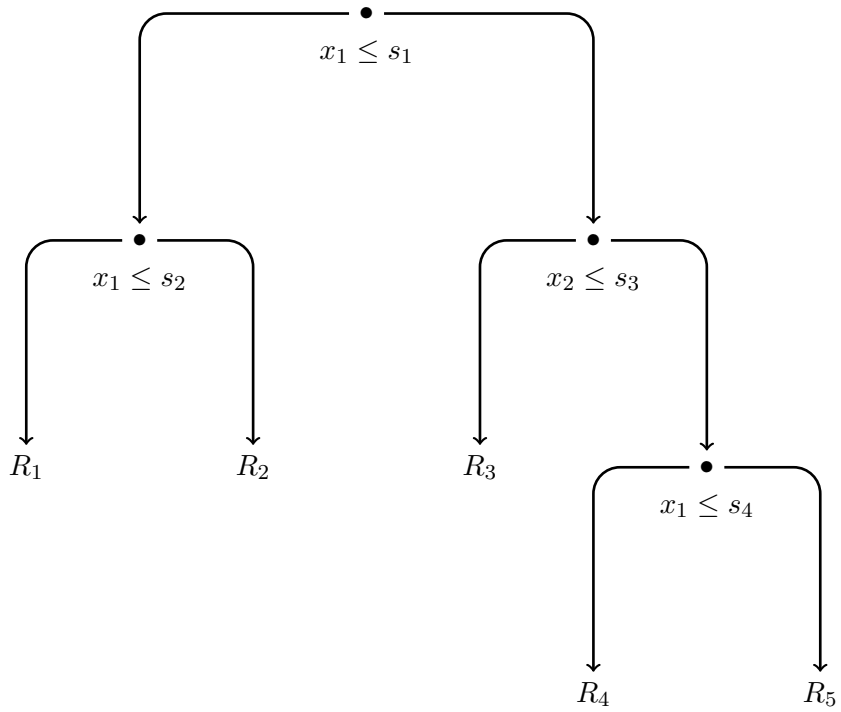

Figure 2.2.2: A tree induced by binary splitting.

We say that binary splitting is greedy, but what exactly is so greedy about it? The greediness comes from choosing the split variable j, split point s as well as split values

γ_1 and γ_2 such that the immediate risk $\mathcal{R}(T)$ after the split is minimized, i.e we try to minimize the following quantity:

$$\begin{aligned}\mathcal{R}(T) \overset{i)}{=} \mathbb{E}_{Y,X}\Big[\ \mathcal{L}(y,\gamma_1)\ \Big|\ x \in R_{\leq}(j,s)\ \Big] \cdot \mathbb{P}\Big[x \in R_{\leq}(j,s)\Big] \\ + \mathbb{E}_{Y,X}\Big[\ \mathcal{L}(y,\gamma_2)\ \Big|\ x \in R_{>}(j,s)\ \Big] \cdot \mathbb{P}\Big[x \in R_{>}(j,s)\Big].\end{aligned}$$

Here, step ***i)*** follows from the law of iterated expectation.

Usually, we do not know the probability distribution of the underlying data. Therefore, we settle for the sample distribution and empirical risk. This means that we also have to approximate $\mathbb{P}[x \in R_{\leq}(j,s)]$ and $\mathbb{P}[x \in R_{>}(j,s)]$ by their sample counterparts. This results in the following equation:

$$\mathcal{R}_{\mathcal{S}}(T) = \frac{1}{|\mathcal{S}|}\left[\sum_{x_i \in R_{\leq}(j,s)} \mathcal{L}(y,\gamma_1) + \sum_{x_i \in R_{>}(j,s)} \mathcal{L}(y,\gamma_2)\right].$$

Consequently, when performing a binary split, we are looking for the following variables:

$$(j^*, s^*, \gamma_1^*, \gamma_2^*) \coloneqq \underset{(j,s,\gamma_1,\gamma_2)}{\arg\min}\left[\sum_{x_i \in R_{\leq}(j,s)} \mathcal{L}(y,\gamma_1) + \sum_{x_i \in R_{>}(j,s)} \mathcal{L}(y,\gamma_2)\right] \tag{2.2.1.1}$$

Notice that the minimization regarding the region values γ_1 and γ_2 is fully dependent on the split variable and split point. This implies that we must solve the whole problem sequentially.

For each splitting variable or predictor we can perform a line search and find the split point s and the corresponding γ_1 and γ_2 very quickly. This leaves us with p number of tuples of the form $(j, s, \gamma_1, \gamma_2)$ corresponding to each predictor. Scanning through all these pairs we arrive at the minimum.

Though, an important requisite for this method is that we are able to compute the values γ_1 and γ_2 quickly if we are given the splitting variable and split point. For example, in the case of quadratic loss computing these values is trivial. The task is analogous to finding the intercept in simple linear regression and hence the solutions simplifies to the sample mean of each region.

Now, we are able to perform a binary split given some sample data and grow a tree recursively. Doing so until each sample is contained in its own partition region would lead to overfitting as the tree would have just memorized the data leaving no room for interaction or dependencies. Clearly, we need some kind of stopping criteria or other method to prevent this. This is where cost complexity pruning comes in.

2.2.2 Cost Complexity Pruning

An interesting and counterintuitive solution to overfitting is the following. We do not concern ourselves with stopping the procedure in time, but rather work backwards after every sample is contained in its own region and slowly reverse the fitting process.

For this, we need to define the proper terminology:

Definition 2.2.2.1. *Let T be a tree of size M, which was grown via recursive binary splitting.*

i) We define graph(T) as the graph induced by binary splitting as in figure 2.2.2. Furthermore, we say that a node or vertex of graph(T) is non-terminal if it is a non-leaf vertex.

ii) By pruning or collapsing the tree at some non-terminal node, we mean reversing all binary splits done to that non-terminal node.

iii) We call a tree $T' \subset T$ a subtree of T if it can be obtained by performing any number of pruning operations on T.

iv) We define the cost of the tree regarding some $\alpha \geq 0$ as the following function:

$$\mathcal{C}(T, \alpha) = \sum_{m=1}^{M} \sum_{x_i \in R_m} \mathcal{L}(y_i, \gamma_m) + \alpha \cdot M = \mathcal{L}_{\mathcal{S},T} + \alpha \cdot M \tag{2.2.2.1}$$

Now, the idea to reduce overfitting is to find a subtree $T_\alpha \subset T$ given any $\alpha \geq 0$. Additionally, by varying α and finding the corresponding subtree T_α, we can adjust the flexibility of the model and hence α serves as a regularization parameter.

For instance, if we choose α as 0, the cost equals the total loss $\mathcal{L}$, which results in no pruning at all. On the other hand, if we choose α too large, then we may have to prune all nodes resulting in a single partition region. Choosing α is usually done with some form of Cross-validation.

Finding the subtree T_α involves two main issues. First, we have to ascertain its uniqueness and second, the computation must not be too tedious since we have to vary α over a wide range.

Luckily for us, both issues are not too much of a problem. As we will see, not only is there a unique smallest subtree T_α for each α, but also for $\alpha_1 \leq \alpha_2$ it holds that $T_{\alpha_2} \subset T_{\alpha_1}$. This implies that we can search for these subtrees in an iterative manner by steadily increasing α and computing the related T_α. This will yield a chain $T \supset T_{\alpha_1} \supset T_{\alpha_2} \supset \dots$.

The following lemma is needed to perform these tasks.

Lemma 2.2.2.2. *Let T be any tree, $\mathcal{S}$ a sample space, $\mathcal{L}$ a loss function, v_c any non-terminal as well as non-root vertex of graph(T) and v_p the parent of v_c. Then, pruning the tree at v_p results in a total loss which is bigger than or equal to the total loss of the tree pruned at v_c.*

That is, the following holds if we define the resulting subtress as T_p and T_c respectively:

$$\mathcal{L}_{\mathcal{S},T_p} \geq \mathcal{L}_{\mathcal{S},T_c}$$

Proof. First, note that we do not need to consider all partition regions of T, but rather only R_p which is induced by the resulting terminal node when pruning T at v_p. The reason for this is that all other partition regions and their values stay invariant when pruning at v_p

or its child v_c and because R_p is the union of all following partition regions corresponding to terminal nodes after v_p.

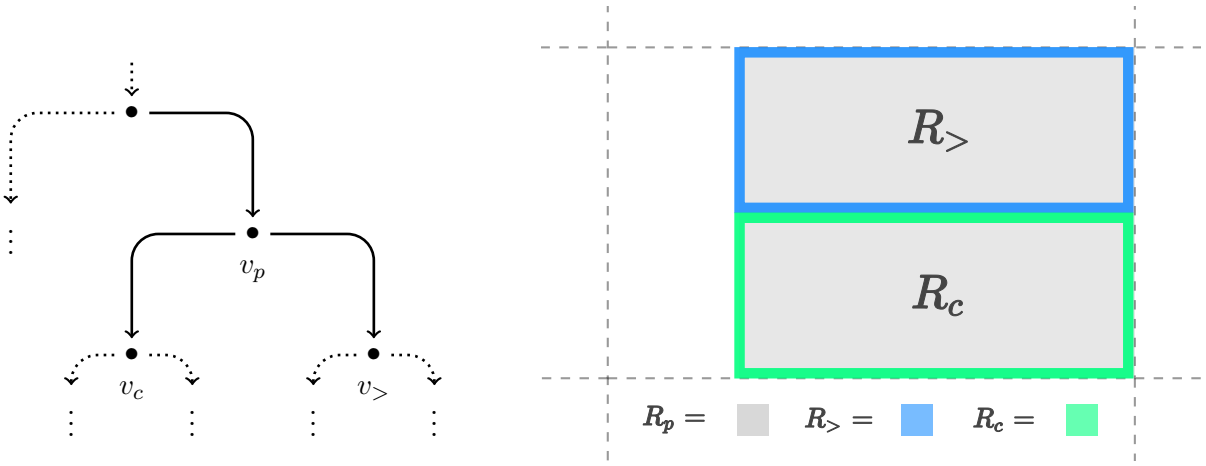

Figure 2.2.3: Induced partition regions of v_p, v_c and $v_>$.

Next, let us define a few quantities:

$$R_p \coloneqq \text{region induced when pruning at } v_p \qquad R_c \coloneqq \text{region induced when pruning at } v_p$$

$$\gamma_p \coloneqq \text{solution to loss minimization of } R_p \qquad \gamma_c \coloneqq \text{solution to loss minimization of } R_c$$

As mentioned above, only the samples contained in R_p contribute to the change in total loss when pruning either v_p or v_c. Therefore, let us consider the loss regarding R_p after pruning at v_p and apply binary splitting equation 2.2.1.1 once more:

$$\begin{aligned}
\underbrace{\sum_{x_i \in R_p} \mathcal{L}(y_i, \gamma_p)}_{\star} &\overset{i)}{\geq} \min_{j,s} \left[\min_{\gamma_1} \sum_{x_i \in R_{\leq}(j,s)} \mathcal{L}(y_i, \gamma_1) + \min_{\gamma_2} \sum_{x_i \in R_{>}(j,s)} \mathcal{L}(y_i, \gamma_2) \right] \\
&\overset{ii)}{=} \min_{\gamma_1} \sum_{x_i \in R_{\leq}(j^*,s^*)} \mathcal{L}(y_i, \gamma_1) + \min_{\gamma_2} \sum_{x_i \in R_{>}(j^*,s^*)} \mathcal{L}(y_i, \gamma_2) \\
&\overset{iii)}{=} \underbrace{\sum_{x_i \in R_c} \mathcal{L}(y_i, \gamma_c)}_{\star\star} + \underbrace{\min_{\gamma_2} \sum_{x_i \in R_{>}(j^*,s^*)} \mathcal{L}(y_i, \gamma_2)}_{\geq 0}
\end{aligned}$$

Here, ***i)*** uses the binary splitting procedure. The inequality holds since we minimize over a partition of R_p. Next, in ***ii)*** we define j^* and s^* as the variables where the minumum was

obtained. Lastly, without loss of generality, we can choose in ***iii)*** one of the summands to be the child vertex v_c.

We can see that, if we prune at v_p, the total loss regarding R_p is $\star$. But if we prune at v_c, then R_p remains partitioned in R_c and $R_>$ and the loss is influenced by $\star\star$ and the other summand. Though, due to the inequality the total loss increases by less if we prune at v_c. Thus, the claim follows.

□

This lemma has a key implication. When deciding which vertex to prune while trying to keep the cost $\mathcal{C}$ low, one only needs to consider the parents of the leafs as every other vertex would increase the loss at least as much as every child further down its branch.

So suppose we have a binary splitted tree T and choose $\alpha > 0$. There are two things which can happen:

i) There is no parent v_p of two leafs such that when pruned, the resulting subtree T_p increases the loss more than α, i.e:

$$\nexists v_p : \mathcal{L}_{\mathcal{S},T_p} - \mathcal{L}_{\mathcal{S},T} > \alpha$$

From this, it follows that T already minimizes the cost 2.2.2.1 regarding this α as pruning any vertex would increase it. This is because pruning any vertex decreases the size of the tree by one and hence also the cost. But, since we increase the cost by pruning by more than α, we conclude that T indeed is a minimizer.

ii) There is a parent v_p of two leafs such that when pruned, the resulting subtree T_p increases the loss at most as much as α, i.e:

$$\exists v_p : \mathcal{L}_{\mathcal{S},T_p} - \mathcal{L}_{\mathcal{S},T} \leq \alpha$$

We can prune all the vertices which fulfill this condition. The cost may decrease or remain the same depending on equality or strict inequality. But remember, we are looking for the *smallest* tree which minimizes the cost. Next, we need to check for the existence of such vertices again as the parents of the pruned vertices may fulfill it once more. Thus, we can continue until we encounter the first case.

Applying this type of procedure for increasing values of α yields the trees that minimize the cost for each α. The beauty of this method however, lies in the fact the we do not need to re-prune the whole tree when incrementing α but rather continue where we left of.

Finally, combining both binary splitting and cost complexity pruning, we are able to fit a tree which is much more robust in regards to overfitting. With these techniques in mind, we are able to introduce gradient boosted trees in the next section.

2.2.3 Gradient and Newton Boosting

Can a set of weak computing functions be combined into a single strong computing function? This was a question posed by Kearns and Valiant [6] in 1989. They referred to this

as Boosting since learners which are barely better than random guessing are combined into a single strong learner.

That question was positively answered by Schapire and Freund [7] in 1990 by presenting an algorithm called AdaBoost. AdaBoost starts off by fitting a shallow tree on some dataset. Of course, we do not expect a high accuracy from such a shallow tree, but we also do not need to tune individual meta-parameters like tree depth. Next, AdaBoost increases the relative weight of samples on which the shallow tree performed bad in and it decreases the relative weight of samples on which the tree performed good in. Then, AdaBoost fits a new tree on the altered dataset. Finally, this new tree is then combined with the previous one into a new single tree which is slightly better than both of them individually, since the second tree compensates for the shortcomings of the first one. Repeating this process enough times yields a computing function which can achieve very competitive results.

AdaBoost operates in an explicit manner by altering the dataset in each iteration. On the other hand, Gradient and Newton Boosting, which we will take a look at, use a stagewise modelling approach. Again, the ideas and calculations in this section are based on Sigrist [1].

First, let us assume that the risk $\mathcal{R}(f)$ is Gâteaux differentiable for all functions f in the space Ω. We denote the Gâteaux derivative by:

$$\partial\mathcal{R}(f,g) \coloneqq \frac{\partial}{\partial\epsilon}\mathcal{R}(f+\epsilon\cdot g)\Big|_{\epsilon=0}$$

Essentially, the Gâteaux derivative of the risk describes the change in risk if we add another function g to the original function f. Note that in this case the plus sign is to be taken literally unlike in the Neural Network case where it was understood as a componentwise change in meta-parameters.

Next, we use the first order Taylor approximation for Gâteaux differentiable functions and add the remainder $\star$. For a detailed explanation refer to Sigrist [1, p. 7].

$$\mathcal{R}(f+g) \approx \mathcal{R}(f) + \partial\mathcal{R}(f,g) + \underbrace{\frac{1}{2}\mathbb{E}_X[g(x)^2]}_{\star}$$

Just as in equation 1.2.0.3, our goal is to minimize the risk $\mathcal{R}$ using stagewise modelling. Suppose that we are in some iteration step i. What if we just minimize the Taylor approximation of the risk and simplify the resulting equation? This will lead to a surprising insight.

$$\begin{aligned}
\partial\mathcal{R}(f,g) &= \frac{\partial}{\partial\epsilon}\mathcal{R}(f+\epsilon\cdot g)\Big|_{\epsilon=0} \\
&= \frac{\partial}{\partial\epsilon}\mathbb{E}_{Y,X}\left[\mathcal{L}(y, f_i(x)+\epsilon\cdot g(x))\right]\Big|_{\epsilon=0} \\
&\overset{i)}{=} \mathbb{E}_{Y,X}\left[\frac{\partial}{\partial\epsilon}\mathcal{L}(y, f_i(x)+\epsilon\cdot g(x))\Big|_{\epsilon=0}\right] \\
&\overset{ii)}{=} \mathbb{E}_{Y,X}\left[\left(\frac{\partial}{\partial h}\mathcal{L}(y,h)\Big|_{h=f_i(x)+\epsilon\cdot g(x)}\frac{\partial}{\partial\epsilon}(f_i(x)+\epsilon\cdot g(x))\right)\Big|_{\epsilon=0}\right]
\end{aligned}$$

$$
\begin{aligned}
&= \mathbb{E}_{Y,X}\left[\underbrace{\frac{\partial}{\partial h}\mathcal{L}(y,h)\Big|_{h=f_i(x)}}_{=:\alpha(y,f_i(x))} \cdot g(x)\right] \\
&= \mathbb{E}_{Y,X}\left[\alpha(y, f_i(x)) \cdot g(x)\right]
\end{aligned}
$$

Here, ***i)*** holds due to the imposed regularity conditions and ***ii)*** is an application of the chain rule. Using these steps, we can drastically simplify the first order Taylor approximation and define the function update f_{i+1}.

$$
\begin{aligned}
f_{i+1} &\coloneqq \arg\min_{g\in\Omega} \mathcal{R}(f_i + g) \\
&\approx \arg\min_{g\in\Omega} \left[\mathcal{R}(f_i) + \partial\mathcal{R}(f_i, g) + \frac{1}{2}\mathbb{E}_X[g(x)^2]\right] \\
&\overset{i)}{=} \arg\min_{g\in\Omega} \left[\partial\mathcal{R}(f_i, g) + \frac{1}{2}\mathbb{E}_X[g(x)^2]\right] \\
&= \arg\min_{g\in\Omega} \left[\mathbb{E}_{Y,X}\left[\alpha(y, f_i(x)) \cdot g(x)\right] + \frac{1}{2}\mathbb{E}_X[g(x)^2]\right] \\
&\overset{ii)}{=} \arg\min_{g\in\Omega} \mathbb{E}_{Y,X}\left[2 \cdot \alpha(y, f_i(x)) \cdot g(x) + g(x)^2 + \alpha(y, f_i(x))^2\right] \\
&= \arg\min_{g\in\Omega} \mathbb{E}_{Y,X}\left[\left(-\alpha(y, f_i(x)) - g(x)\right)^2\right]
\end{aligned}
$$

Here, step ***i)*** holds since we can include or exclude additive and multiplicative terms that do not depend on the minimizer. Using the same idea and the linearity of expectation, we conclude step ***ii)***.

This leads us to the aforementioned surprising connection. Remember that $\alpha(y, f_i(x))$ is the gradient of the loss and hence the optimal $g(x)$ is nothing more than the squared error minimizer of the negative gradient. Defining the function update f_{i+1} in this way results in Gradient Boosting.

Next, let us introduce another method called Newton Boosting. Newton Boosting goes a step further and considers the second order Taylor approximation for Gâteaux differentiable functions and as such demands further regularity conditions. Hence, let us assume that the loss is twice differentiable regarding the computing function f and the first derivative integrable regarding the probability measure. This yields the following approximation.

$$
\mathcal{R}(f + g) \approx \mathcal{R}(f) + \partial\mathcal{R}(f, g) + \frac{1}{2}\partial^2\mathcal{R}(f, g)
$$

Again, to arrive at a meaningful result, we need to simplify the term $\frac{1}{2}\partial^2\mathcal{R}(f, g)$ by using the same considerations as above.

$$
\partial^2\mathcal{R}(f, g) = \frac{\partial^2}{\partial\epsilon^2}\mathcal{R}(f + \epsilon \cdot g)\Big|_{\epsilon=0}
$$

$$
\begin{aligned}
&= \frac{\partial^2}{\partial \epsilon^2} \mathbb{E}_{Y,X} \left[\mathcal{L}(y, f_i(x) + \epsilon \cdot g(x)) \right] \Big|_{\epsilon=0} \\
&= \mathbb{E}_{Y,X} \left[\frac{\partial^2}{\partial \epsilon^2} \mathcal{L}(y, f_i(x) + \epsilon \cdot g(x)) \Big|_{\epsilon=0} \right] \\
&= \mathbb{E}_{Y,X} \left[\frac{\partial}{\partial \epsilon} \left(\frac{\partial}{\partial h} \mathcal{L}(y, h) \Big|_{h=f_i(x)+\epsilon \cdot g(x)} \frac{\partial}{\partial \epsilon} (f_i(x) + \epsilon \cdot g(x)) \right) \Big|_{\epsilon=0} \right] \\
&= \mathbb{E}_{Y,X} \left[\left(\frac{\partial^2}{\partial h^2} \mathcal{L}(y, h) \Big|_{h=f_i(x)+\epsilon \cdot g(x)} \cdot g(x) \cdot \frac{\partial}{\partial \epsilon} (f_i(x) + \epsilon \cdot g(x)) \right) \Big|_{\epsilon=0} \right] \\
&= \mathbb{E}_{Y,X} \left[\underbrace{\frac{\partial^2}{\partial h^2} \mathcal{L}(y, h) \Big|_{h=f_i(x)}}_{=:\beta(y, f_i(x))} \cdot g(x)^2 \right] \\
&= \mathbb{E}_{Y,X} \left[\beta(y, f_i(x)) \cdot g(x)^2 \right]
\end{aligned}
$$

By applying the above calculation to the second order Taylor approximation in a similar manner as before, we arrive at the function update f_{i+1} corresponding to Newton Boosting.

$$
\begin{aligned}
f_{i+1} &\coloneqq \arg\min_{g \in \Omega} \mathcal{R}(f_i + g) \\
&\approx \arg\min_{g \in \Omega} \left[\mathcal{R}(f) + \partial \mathcal{R}(f, g) + \frac{1}{2} \partial^2 \mathcal{R}(f, g) \right] \\
&= \arg\min_{g \in \Omega} \left[\partial \mathcal{R}(f, g) + \frac{1}{2} \partial^2 \mathcal{R}(f, g) \right] \\
&= \arg\min_{g \in \Omega} \left[\mathbb{E}_{Y,X} \left[\alpha(y, f_i(x)) \cdot g(x) \right] + \frac{1}{2} \mathbb{E}_{Y,X} \left[\beta(y, f_i(x)) \cdot g(x)^2 \right] \right] \\
&= \arg\min_{g \in \Omega} \mathbb{E}_{Y,X} \left[2 \cdot \alpha(y, f_i(x)) \cdot g(x) + \beta(y, f_i(x)) \cdot g(x)^2 + \frac{\alpha(y, f_i(x))^2}{\beta(y, f_i(x))^2} \right] \\
&= \arg\min_{g \in \Omega} \mathbb{E}_{Y,X} \left[\beta(y, f_i(x)) \cdot \left(-\frac{\alpha(y, f_i(x))}{\beta(y, f_i(x))} - g(x) \right)^2 \right]
\end{aligned}
$$

As we can see, the expressions regarding Gradient and Newton Boosting are similar and only differ in the factor $\beta(y, f_i(x))$. By using the second derivative of the loss $\beta(y, f_i(x))$, we incorporate more information into the function update f_{i+1}. As such, one might also expect a better overall result. This is further examined by Sigrist [1].

Finally, by using the function updates f_{i+1} above corresponding to either Gradient or Newton Boosting, we are able to perform a stagewise approach as seen in the equations 1.2.0.2 and 1.2.0.3. Additionally, we can introduce a regularization parameter similar to the step size of Gradient Descent. This parameter ν, which is known as the learning rate, may or may not depend on the current iteration step. All of this combined yields the following final computing function f when using M as the number of iterations.

$$f \coloneqq \sum_{i=1}^{M} \nu_i \cdot f_i$$

2.3 Summary

Let us briefly summarize what we covered in the chapter. First, we introduced Neural Networks. Neural Networks are composed of multiple layers containing neurons. Input data propagates through each layer until the output layer is reached. The output layer can contain any number of neurons depending on the task.

For example, in a binary classification or a simple regression task a single neuron in the output layer is sufficient, but in a multi-classification or multivariate regression additional neurons are used. For each of these cases, a different loss function is required. Preferably, this loss function is differentiable so that we can use Gradient Descent by means of Backpropagation to fit the network efficiently.

Hence, Neural Networks can be used for most statistical tasks. The regularization and meta-parameter are the following:

meta-parameters:

- layer depth
- number of neurons in each layer
- weights in each layer
- biases in each layer
- type of activation functions

regularization parameters:

- number of epochs
- step size for Gradient Descent
- batch size for computing the gradient

Note that meta-parameters can also be treated as regularization parameters, since reducing the network flexibility results in a lower tendency to overfit. The whole fitting process of Neural Networks can be summarized by algorithm 2.3.1.

Algorithm 2.3.1 Training a Neural Network.

Input: Sample space $\mathcal{S}$, loss $\mathcal{L}$, *step_size*, *epochs*, *batch_size* and *network_structure*.

Initialize $N(x)$ with random weights according to *network_structure*.
for $j \in [epochs]$ **do**
    Randomly divide $\mathcal{S}$ into batches $(b_i)_{i \in \mathcal{I}}$ of *size* $\approx$ *batch_size*.
    for $i \in \mathcal{I}$ **do**
        $\nabla \mathcal{R}(N) \leftarrow$ Compute gradient of $N(x)$ regarding b_i and $\mathcal{L}$.
        $N(x) \leftarrow N(x) - \text{step\_size} \cdot \nabla \mathcal{R}(N)$
    end for
end for

Output: Neural Network computing function $N(x) : \mathbb{R}^{n(1)} \to \mathbb{R}^{n(L)}$.

Then, we covered tree based methods. A tree consists of a partition of the parameter space alongside values assigned to each individual partition region. It induces a graph which visualizes the way the computation is performed. They are fit using recursive binary splitting and then pruned back. Their accuracy can be greatly improved by using Gradient or Newton Boosting. Usually, when using these methods, the pruning step is skipped and the tree size is bounded by some value.

In contrast to Neural Networks, trees are a lot easier to handle. They do not require extensive meta-parameter tweaking and can be used out of the box for both regression and classification depending on the used loss function. One of their downsides is that they cannot really be used for multivariate regression, since they assign one dimensional values to each partition region.

If we are not using Boosting, the only relevant regularization parameter is α which tunes cost complexity pruning. However, if we are using gradient or Newton Boosting, the regularization and meta-parameters are the following:

meta-parameter:

- tree depth

regularization parameters:

- learning rate of Boosting
- number of Boosting iterations

An interesting thing to point out is that both, Neural Networks and trees, use a stagewise modelling approach in some way. Neural Networks use it to train themselves creating a *single* and increasingly accurate model by adjusting a few meta-parameters individually.

On the other hand, trees use stagewise modeling in conjunction with Gradient or Newton Boosting. This results in *multiple* models which are combined to arrive at a new single computing function. The way this combination is performed usually depends on the given task. For example, in the case of regression, the models can be summed up while in the case of classification, we output the class which was most prevalent. We can summarize Gradient Boosting by algorithm 2.3.2 if the task is regression.

Algorithm 2.3.2 Training a gradient boosted tree.

Input: Sample space $\mathcal{S}$, loss $\mathcal{L}$, learning rate ν, *tree_depth* and *boosting_steps*.

1: Fit tree f_0 of depth *tree_depth*.
2: **for** $j \in [boosting\_steps]$ **do**
3: Compute f_j according to Gradient Boosting.
4: $f_j \leftarrow f_{j-1} + \nu \cdot f_j$
5: **end for**
6: $T(x) \leftarrow f_j$

Output: Gradient boosted tree $T(x) : \mathbb{R}^p \to \mathbb{R}$.

Chapter 3

Computing Libraries

In this chapter we will introduce the currently widely used libraries Keras and XGBoost. Previously, we covered Neural Networks and gradient boosted trees theoretically. To illustrate the theory in practice, we will use Keras for Neural Networks and XGBoost for gradient boosted trees.

The task will be to classify handwritten digits which are provided in the MNIST dataset. This problem and namely this dataset is one of the classical difficult problems for traditional algorithms. It became practically solvable with the emergence of modern techniques such as back propagation and gradient boosting in addition to growing computing power. In figure 3.0.1, we can see some of the handwritten digits included in the dataset.

Figure 3.0.1: A sample of MNIST digits.

3.1 Keras

Keras is an open source library which was developed by Chollet [8] and released in 2015. It is written in Python but offers support for a variety of programming languages including R. One of its main strengths is the use of a functional api which allows for complex models in very few lines of code and no extensive knowledge about low level programming routines.

Say for example we want to define a fully connected Neural Network which expects an input dimension of 784 and 2 subsequent layers with 512 and 256 neurons respectively. Also, as our goal is multi-class classification of handwritten digits, we require 10 output neurons. The following code does the trick:

```
# Import the library and set the seed.
library(keras)

use_session_with_seed(444)

# Define the model.
model <- keras_model_sequential() %>%
    layer_dense(units = 512, activation = "relu", input_shape = 784) %>%
    layer_dense(units = 256, activation = "relu") %>%
    layer_dense(units = 10, activation = "softmax")
```

Note that such a network would have 4 layers as described in section 2.1, but Keras does not consider the input layer as a true layer. In contrast to the other layers, we need to define the input layer via the `input_shape` parameter. Additionally, we defined the activation functions present in each layer by setting the value of the `activation` parameter. Here, `"relu"` is short for rectified linear unit which essentially filters out negative contributions and `"softmax"` transforms the output of the previous layer into probabilities.

There are still a few things missing, though. We still need to define a loss function to be applied on the output and the true label. A suitable loss function in this case is the categorical cross-entropy. This yields the following code:

```
# Compile the model.
model %>% compile(
    optimizer = "rmsprop",
    loss = "categorical_crossentropy",
    metrics = ("accuracy"))
```

Here the additional parameter `rmsprop` describes an adaptive variant of the gradient descent. The `metrics` parameter allows to define metrics which will be kept track of while the model trains itself on the data. In this case `accuracy` refers to the proportion of correctly classified samples.

As our Neural Network requires the input shape of data to be a vector of length 784 and the MNIST data is provided as a two dimensional matrix with pixel values ranging from 0 to 255, we need to transform the individual digits in the following way. Additionally, we need to transform the labels such that they are one hot encoded.

```
# Load the dataset.
mnist <- dataset_mnist()
```

```
3
4  # Define images and labels.
5  train_images <- mnist$train$x
6  train_labels <- mnist$train$y
7  val_images <- mnist$test$x
8  val_labels <- mnist$test$y
9
10 # Encode pixel values in [0,1].
11 train_images <- array_reshape(train_images, c(60000, 28 * 28)) / 255
12 val_images <- array_reshape(val_images, c(10000, 28 * 28)) / 255
13
14 # Use one hot encoding on the labels.
15 one_hot_encoding <- function(labels){
16     result <- matrix(0, nrow = length(labels), ncol = length(levels(as.factor(labels))))
17
18     for(i in 1:length(labels)){
19         result[i,labels[i] + 1] <- 1
20     }
21
22     return(result)
23 }
24
25 train_labels <- one_hot_encoding(train_labels)
26 val_labels <- one_hot_encoding(val_labels)
```

All we need to do now is to start the training process with the following code. The `batch_size` parameter determines the amount of training samples to be used when computing the gradient for gradient descent. The whole training set is then divided into batches of size `batch_size` and a gradient descent step is performed for each of them. When all training samples have been processed in this way, an `epoch` is finished. The whole process is analogous to algorithm 2.3.1. The number of such `epochs` can be controlled with the same-named parameter. Lastly, the parameter `validation_data` allows us to monitor the test loss and accuracy while training.

```
1  history <- model %>% fit(train_images,
2                           train_labels,
3                           epochs = 20,
4                           batch_size = 128,
5                           validation_data = list(val_images, val_labels))
```

How good does our model perform? As we can see in figure 3.1.1, the validation accuracy fluctuates after the eighth epoch and does not improve substantially afterwards. This suggests that the highest test accuracy may be achieved with only 8 epochs and training the network further could lead to overfitting.

An accuracy of around 98% is certainly an astonishing result even though the used model is relatively simple and some of the digits in figure 3.0.1 look horrific.

3.2 XGBoost

XGBoost is also an open source library which was initially started as a research project by Chen [9]. The library was first released in 2014 and quickly gained popularity due to

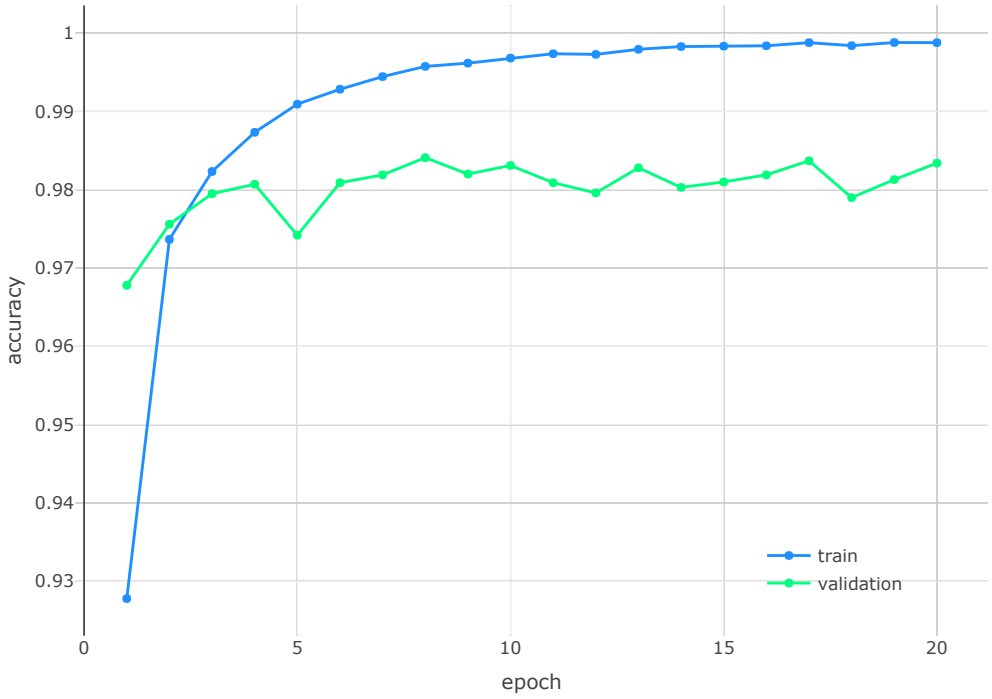

Figure 3.1.1: Training history of the Neural Network.

its success in the Kaggle Higgs Machine Learning Challenge. In regards to ease of use, it even beats out Neural Networks since there is no need to define a network structure. Let us load the data set similarly to the Neural Network example.

```
# Import necessary libraries and set the seed.
library(xgboost)
library(keras)
library(Matrix)

use_session_with_seed(444)

# Load the dataset.
mnist <- dataset_mnist()

# Define images and labels.
train_images <- mnist$train$x
train_labels <- mnist$train$y
val_images <- mnist$test$x
val_labels <- mnist$test$y

# Encode pixel values in [0,1].
train_images <- array_reshape(train_images, c(60000, 28 * 28)) / 255
train_images <- Matrix(train_images, sparse = TRUE)

val_images <- array_reshape(val_images, c(10000, 28 * 28)) / 255
val_images <- Matrix(val_images, sparse = TRUE)
```

In contrast to Neural Networks, we additionally transform the data to a sparse matrix as this speeds up the computation time. To monitor validation accuracy during training,

XGBoost requires the use of its own `xgb.DMatrix` matrix class.

```
# Preparing data for XGBoost.
dtrain <- xgb.DMatrix(data = train_images, label = train_labels)
dval <- xgb.DMatrix(data = val_images, label = val_labels)

watchlist <- list(train=dtrain, test=dval)
```

Now, all we need to do is to define the model parameters and start the training process. We use a learning rate of 0.1 which corresponds to the `eta` parameter. Also, we fix the depth of the grown trees to 8 by setting the `max.depth` parameter. The number of boosting rounds is set to 20 via the `nrounds` parameter. Lastly, the parameters `num_class` and `objective` correspond to the multi-class classification task.

```
# Train the model.
bst <- xgb.train(data = dtrain,
                 max.depth = 8,
                 eta = 0.1,
                 nthread = 2,
                 nrounds = 20,
                 watchlist = watchlist,
                 num_class = 10,
                 subsample = 0.8,
                 objective = "multi:softmax")
```

As we can see in figure 3.2.1, the gradient boosted model steadily increases with each boosting step. Unlike the Neural Network, it does not seem to suffer from overfitting. Hence, we could increase the boosting steeps and achieve an even higher accuracy. Nevertheless, we refrain from doing so as the computational time was already longer than in the Neural Network example.

Nevertheless, implementing the model in XGBoost was even simpler than in Keras as no complex network architecture had to be defined. Even though the validation accuracy of around 95% somewhat pales in comparison to the Neural Network, it still is very impressive.

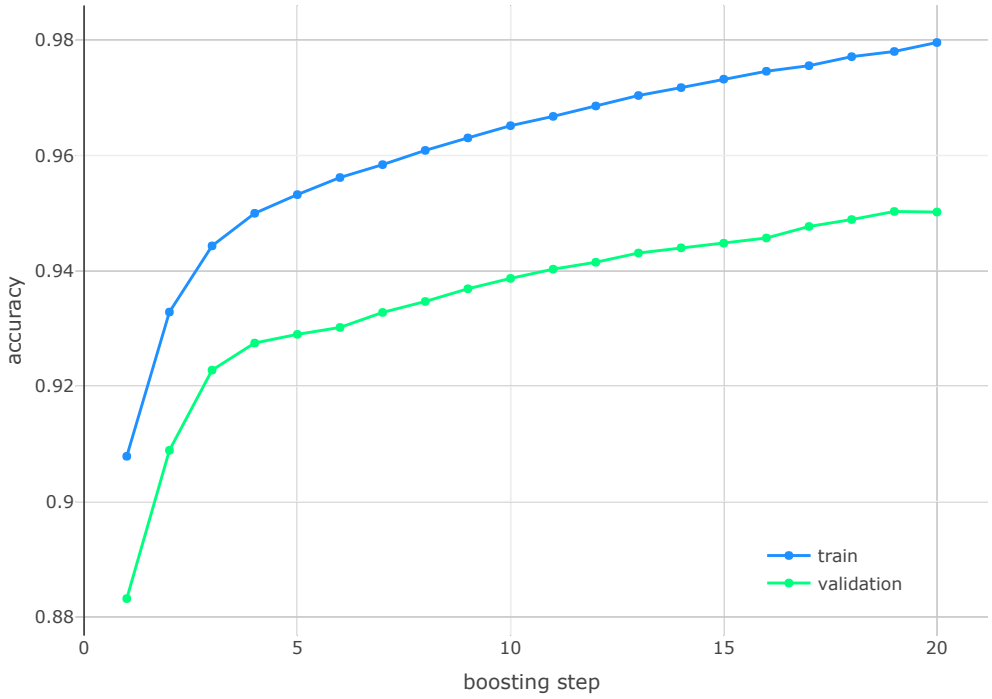

Figure 3.2.1: Training history of XGBoost.

Chapter 4

Random Rotation Upscaling

In the previous chapters, we learned in depth about Neural Networks and gradient boosted trees from a theoretical and practical viewpoint. During the making of these chapters and in an effort to distinguish both methods, the idea of *Random Rotation Upscaling* emerged.

The original intent was to study the change in accuracy of Neural Networks and gradient boosted trees after applying a random rotation on the predictor space $\mathbb{R}^p$ of dimension p. However, what if instead we sample a random rotation in regards to a higher dimension $q > p$ and apply it to the predictor space $\mathbb{R}^p$? Intuitively, we would expect the accuracy of both methods to fall as the data is upscaled into a higher dimensional space which introduces a more extensive search space and increased noise. But is that really the case?

4.1 Random Rotations

To explore this idea further, we need to introduce the notion of random rotations first. We are looking for a rotation matrix Q in arbitrary dimensions, i.e a matrix that fulfills the following properties:

i) $Q \in \mathbb{R}^{q \times q}$

ii) $Q^t \cdot Q = Q \cdot Q^t = \mathbb{1}_{q \times q}$

iii) $\det(Q) = 1$

The last condition *iii)* ensures that Q is a member of the special orthogonal group $SO(q)$. Members of this group are generally called rotations since for $q \in \{2, 3\}$ they represent the usual rotations of 2 or 3 dimensional euclidean space.

Additionally, we would like to sample these rotations, preferably in a uniform manner. Luckily for us, this is a topic in a paper by Srestasathiern et al. [10] (algorithm 2). To derive the following algorithm 4.1.1, Srestasathiern et al. combined ideas from Stewart [11] and Kanatani [12]. Stewart proposed a uniform sampling method for matrices in the orthogonal group $O(q)$ while Kanatani provided a way to project these matrices onto the $SO(q)$ manifold.

Algorithm 4.1.1 Sampling of q dimensional rotation.

Input: $q \in \mathbb{N}$ and $\sigma > 0$

1: **for** $j \in \{q, \dots, 2\}$ **do**
2: Sample $\alpha_j \in \mathbb{R}^j$ from normal distribution $\mathcal{N}(0_j, \sigma^2 \mathbb{1}_{j\times j})$.
3: Compute Householder transformation $\hat{H}_j$ reducing α_j to $r_j \cdot e_j$ where $r_j \in \mathbb{R}$ and $e_j \coloneqq (1, 0, \dots, 0)^t \in \mathbb{R}^j$.
4: $H_j \leftarrow \operatorname{diag}(\mathbb{1}_{q-j\times q-j}, \hat{H}_j)$
5: **end for**
6: Sample r_1 from $\mathcal{N}(0, \sigma^2)$.
7: $H_1 \leftarrow -1$
8: $D \leftarrow \operatorname{diag}(sign(r_q), \dots, sign(r_1))$
9: $WDV^t \leftarrow \operatorname{SVD}(D \cdot H_q \cdot \ldots \cdot H_1)$
10: $Q \leftarrow W \cdot \operatorname{diag}(\underbrace{1, \dots, 1, \det(WV^t)}_{q \text{ entries}})\, V^t$

Output: $Q \in SO(q)$

Remark.

i) A variable σ is needed as an input to the algorithm 4.1.1. However, the output matrix D of step 8 is an orthogonal matrix according to Stewart [11] and as such does not rely on the σ.

ii) Steps 9 and 10 apply the idea of projecting the orthogonal matrix D onto the $SO(q)$ manifold. But as Kanatani [12] never mentions the preservation of uniformity under this operation, we can only call the resulting rotation matrix Q quasi uniformly sampled.

iii) Algorithm 4.1.1 contains the lines 6 and 7 representing a Householder transformation in the one dimensional case. In the original paper by Srestasathiern et al. [10], these lines were missing. Though, this had to be a mistake since the matrix dimensions do not coincide in step 9 if we leave out lines 6 and 7.

As we can see in the algorithm, it works by computing Householder transformations. This can be done as follows. Suppose we have a fixed j, $\alpha \sim \mathcal{N}(0_j, \sigma^2 \mathbb{1}_{j\times j})$ and $e \coloneqq (1, 0, \dots, 0)^t \in \mathbb{R}^j$. Then a Householder transformation is a matrix $\hat{H} \in \mathbb{R}^{j\times j}$ that fulfills the equation:

$$\hat{H}\alpha = re \tag{4.1.0.1}$$

The explicit form of $\hat{H}$ can be obtained with the following three equations:

i) $\hat{H} = \mathbb{1}_{j\times j} - 2\frac{uu^t}{\|u\|_2^2}$

ii) $u \coloneqq \operatorname{sign}(\alpha_1)\|\alpha\|_2 \cdot e + \alpha$

iii) $r \coloneqq \pm\|\alpha\|_2^2$ s.t. equation 4.1.0.1 is satisfied.

Note that α_1 in equation *ii)* describes the first entry of $\alpha \in \mathbb{R}^j$. Using the above considerations, we can compute a rotation matrix in arbitrary dimensions. However, there are two things to consider.

First, the runtime of this algorithm is bound to suffer for very high q. In step 9, we multiplicate q Householder matrices which are of dimension $q \times q$. Thus, even without considering the singular value decomposition nor calculation of the determinant, the runtime is at least $\Omega(q^3)$.

Second, if we increase the upscaling dimension q, the individual entries of Q can become very small as each column is normalized. This results in numerical instability for large q.

4.2 Synthetic Dataset

To use random rotations in an upscaling scenario, we also need a data set. Let us create a simple binary classification task. The idea is to sample a grid on the predictor space $[-1, 1]^p$ and then assign a label uniformly at random in $\{0, 1\}$ to each grid region. This procedure can be summarized by algorithm 4.2.1.

Algorithm 4.2.1 Uniform sampling of a grid on $[-1, 1]^p$.

Input: $p \in \mathbb{N}$ and number of splits C.

1: $\mathcal{J} \leftarrow \{\varnothing\}$
2: $\mathcal{G} \leftarrow \{\varnothing\} \times \cdots \times \{\varnothing\} = \{\varnothing\}^p$
3: **for** $j \in [C]$ **do**
4: Sample split variable $v_j \sim \text{Unif}(\{1, \ldots, p\})$ and split point $s_j \sim \text{Unif}([-1, 1])$.
5: $\mathcal{G}_{v_j} \leftarrow \mathcal{G}_{v_j} \cup s_j$
6: **end for**
7: **for** $t = (s_i)_{i \in [p]} \in \mathcal{G}$ **do**
8: Sample $l \sim \text{Unif}(\{0, 1\})$.
9: $\mathcal{J} \leftarrow \mathcal{J} \cup (t, l)$
10: **end for**

Output: Grid $\mathcal{G}$ and labels $\mathcal{J}$.

As we can see, $\mathcal{G}$ is a p dimensional set of sets which contain individual split points. This induces a grid on $[-1, 1]^p$ and each unique tupel $t = (s_i)_{i \in [p]} \in \mathcal{G}$ represents an individual region in that grid. Therefore, we can characterize the labels $\mathcal{J}$ by those tuples alongside a label l, as done in step 9.

In the case of $q = 2$, we can visualize the resulting grid $\mathcal{G}$ and labels $\mathcal{J}$ with figure 4.2.1. The green rectangles represent grid regions which were assigned label 0 and the blue ones correspond to label 1.

Algorithm 4.2.1 generates the grid and labels, but we are still missing the data. All we need to do is to sample values from $[-1, 1]^p$ and check in which grid region they fall. As can be seen in figure 4.2.1, this determines their corresponding label. The following algorithm 4.2.2 performs exactly this task.

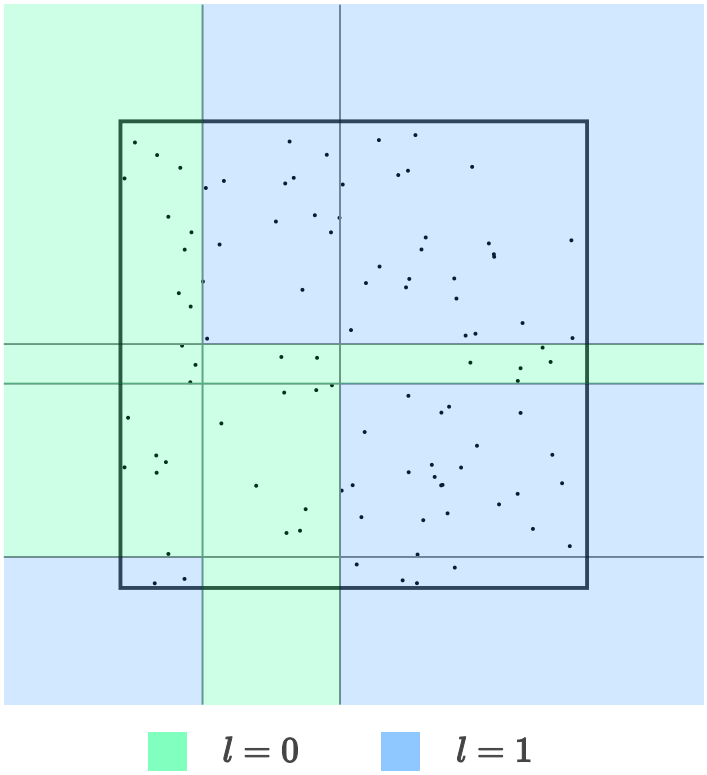

Figure 4.2.1: An example of a two dimensional grid with binary labels.

Next, let us combine the sampled data from algorithm 4.2.2 and a random rotation matrix Q from algorithm 4.1.1 with dimensions $q = p$. We can apply the matrix on the data and obtain figure 4.2.2 which unsurprisingly is just a rotation of figure 4.2.1. A more exciting use of random rotations will be presented in the next section.

4.3 Upscaling

Upscaling is a term mostly used in computer graphics and video processing. It stands for the practice of increasing the resolution of images. Similarly, we will refer to the practice of embedding a data set into a higher dimensional space as upscaling.

The main idea is to pad our synthetic data from algorithm 4.2.2 with zeros to increase its dimensionality. This way, we embed the predictor space $\mathbb{R}^p$ onto a p-dimensional hyperplane in the space $\mathbb{R}^q$ with $q > p$. Next, we can sample a q-dimensional rotation matrix Q and apply it on our embedded data. As a result, the hyperplane gets randomly rotated in q dimensions and the individual samples x_j become true inhabitants of $\mathbb{R}^q$. Looking at each individual sample x_j, we cannot conclude that they were once upscaled from a lower dimension, since the padded zeros disappear and with high probability, every

Algorithm 4.2.2 Uniform sampling of data with labels.

Input: $p \in \mathbb{N}$, number of splits C and number of samples n.

1: $(\mathcal{G}, \mathcal{J}) \leftarrow$ **Algorithm 4.2.1** (p, C)
2: $\mathcal{S} \leftarrow \{\varnothing\}$
3: **for** $j \in [n]$ **do**
4: Sample $x_j \sim \text{Unif}([-1, 1]^p)$.
5: Determine grid region $t_j \in \mathcal{G}$ of x_j.
6: Determine label y_j of t_j from $\mathcal{J}$.
7: $\mathcal{S} \leftarrow \mathcal{S} \cup \{(x_j, y_j)\}$
8: **end for**

Output: Sample space $\mathcal{S}$, grid $\mathcal{G}$ and labels $\mathcal{J}$.

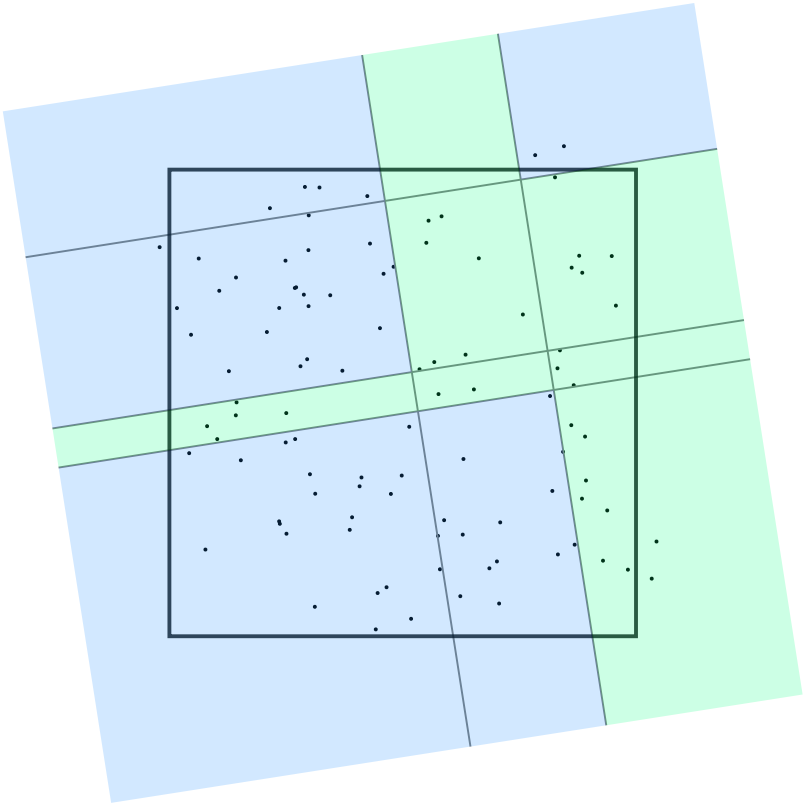

Figure 4.2.2: Two dimensional rotation of figure 4.2.1.

entry is non zero. These steps are formalized in the final algorithm 4.3.1.

To visualize this whole process, let us upscale figure 4.2.1 into dimension 3, which results in figure 4.3.1. As we can see, the internal structure is preserved while the upscaled data set itself can be thought of as 3 dimensional.

Algorithm 4.3.1 Upscaling of a data set.

Input: Sample space $\mathcal{S}$ (of dimension p) and upscaling dimension $q > p$.

1: $Q \leftarrow$ **Algorithm 4.1.1** $(q, 1)$
2: $\tilde{\mathcal{S}} \leftarrow \{\varnothing\}$
3: **for** $j \in [|\mathcal{S}|]$ **do**
4: $\quad \tilde{x}_j \leftarrow x_j \times \{0\} \times \cdots \times \{0\} \in \mathbb{R}^q$
5: $\quad \tilde{x}_j \leftarrow Q \cdot \tilde{x}_j$
6: $\quad \tilde{\mathcal{S}} \leftarrow \tilde{\mathcal{S}} \cup (\tilde{x}_j, y_j)$
7: **end for**

Output: Upscaled sample space $\tilde{\mathcal{S}}$.

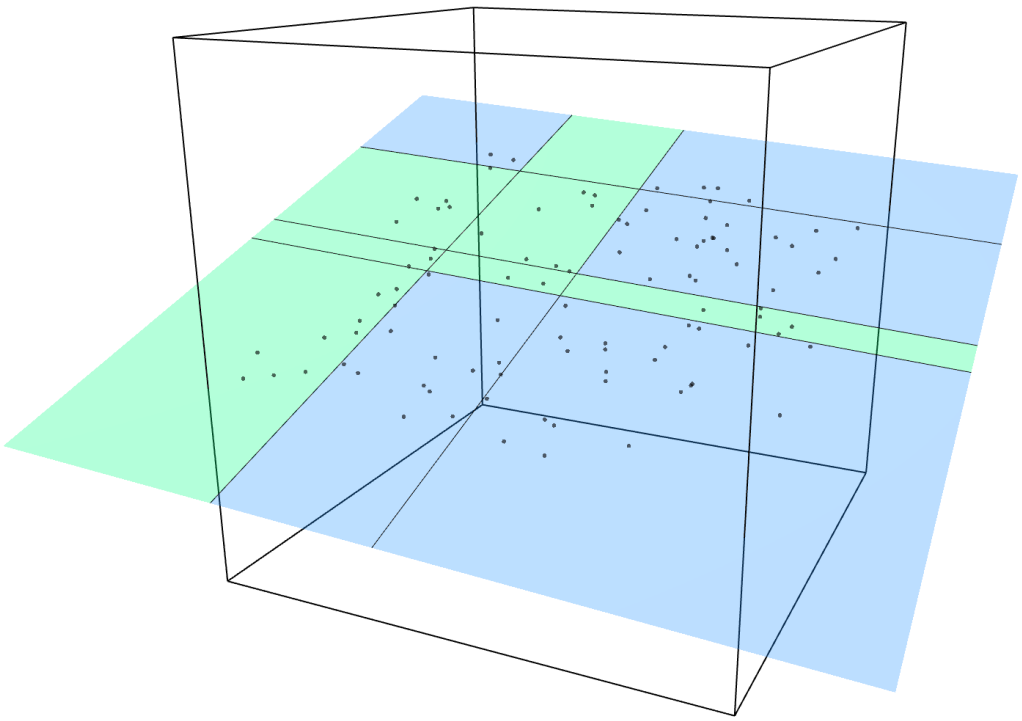

Figure 4.3.1: Three dimensional upscaling of figure 4.2.1.

4.4 Simulation Study

We have seen that upscaling produces an alternate representation of data in a higher dimension. So, how does a Neural Network or gradient boosted tree perform on the new representation? To explore this question, we will perform a simulation study. In this study, we will only examine random rotation upscaling as in figure 4.3.1 and not random

rotation on its own as seen in figure 4.2.2.

First, we will generate several synthetic data sets of dimension 10 with 10000 samples and different split numbers C in the range of $\{5, \dots, 16\}$. Then, all of them will be upscaled to dimensions ranging in $\{50, 100, 250, 500\}$. These steps will be repeated 500 times yielding a total of 30000 data sets.

$$30000 = 500 \cdot \#\text{splits} \cdot \#(\text{upscaling dimensions} + \text{original dimension})$$

Remark. As mentioned in section 4.1, algorithm 4.1.1 which samples a random rotation matrix Q may suffer from numerical instability for higher values of q. This resulted in several failed singular value decompositions reducing the number of data sets from 30000 to 27180. The loss of 2820 data sets may seem extensive, but in truth only around 4 in 500 decompositions fail. However, each failure affected a different seed of the random number generator in R. For easier reproducibility of the study, all successful seeds were intersected resulting in this severe loss of data sets. Refer to appendix A for more information.

Fitting both a Neural Network as well as gradient boosted trees on each data set while tracking their performance will yield an empirical distribution from which we can infer results. Note that the meta-parameters regarding each model will be fixed for all iterations. As such, all fitted Neural Networks and all gradient boosted trees will have the same degree of flexibility in their class. The detailed structure as well as meta parameters regarding both methods are presented in appendix A.

Due to the scale of the simulation, it was performed on the ETH Zurich high performance computing cluster Leonhard [13]. It took 24 graphical processors running in parallel for around 18 hours to finish the whole simulation.

The two relevant metrics that we kept track of during the simulation are **classification accuracy** and **fitting index**. In the case of binary classification, the accuracy is the amount of correctly classified samples divided by the total amount of samples. The fitting index specifies the number of boosting steps or epochs after which the models did not significantly improve in accuracy. An excerpt of the simulation results can be seen in figure 4.4.1.

Let us examine a part of the data. In figure 4.4.2, we can see the empirical distributions regarding the accuracy based on 11 splits and the upscaling dimensions {"not upscaled", 100, 500}. Unsurprisingly, gradient boosted trees perform exceptionally well on the original data set. As we have seen, trees split the predictor space parallel to the axes and as a result are optimally suited for the original data. This advantage seems to carry over to Gradient Boosting.

However, the story changes as soon as we apply upscaling. The accuracy of gradient boosted trees degrade drastically while Neural Networks seem unperturbed. Moreover, in the case of Neural Networks, the outliers, which are marked in pink, even seem to regress to the mean. This might indicate an increase in accuracy regarding worst cases as well as an improvement in numerical stability.

Next, let us take a look at figure 4.4.3. Instead of plotting the accuracy, we plot the fitting index regarding the same data. Again, gradient boosted trees perform best on the non upscaled samples, but have a higher fitting index after upscaling. On the other hand,

Accuracy	Index	Type	Dimension	Splits
0.874	172	GB	100	6
0.929	201	NN	10	8
0.942	183	NN	250	7
0.938	138	NN	500	8
0.827	218	NN	50	12
0.828	169	GB	50	5
0.638	169	GB	500	16
0.882	162	NN	100	12
0.723	160	GB	50	13
0.948	127	NN	50	9
0.925	133	NN	50	7
0.98	25	GB	10	8
0.971	104	NN	500	6
0.752	124	GB	50	11
0.863	122	NN	250	10
⋮	⋮	⋮	⋮	⋮

Figure 4.4.1: Excerpt of simulation results.

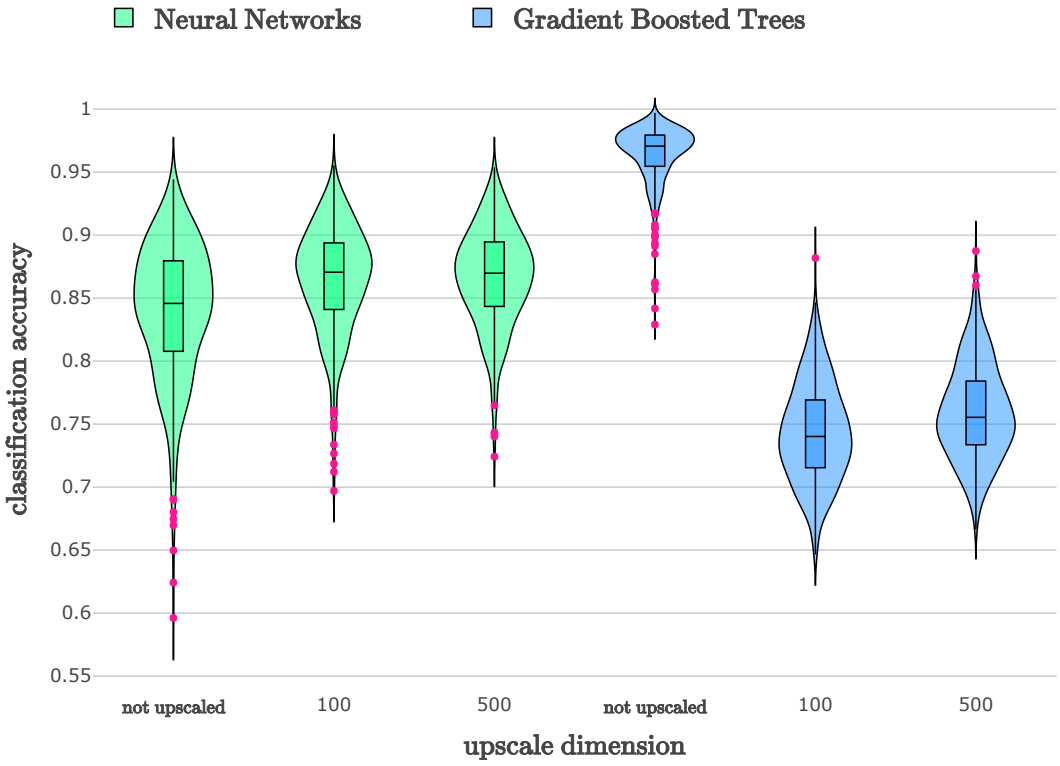

Figure 4.4.2: Accuracy of data with 11 splits.

Neural Networks seem to benefit from upscaling as the fitting index decreases in higher dimensions.

These figures provide us with valuable intuition and possible conclusions which we can

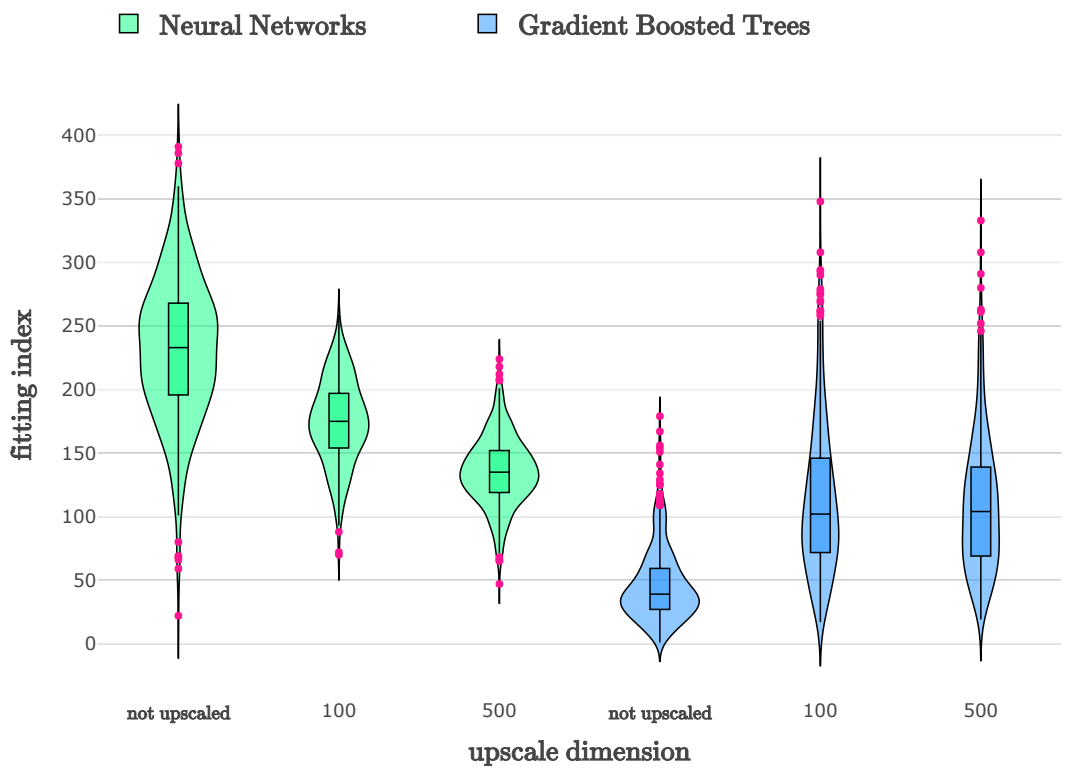

Figure 4.4.3: Fitting index of data with 11 splits.

further investigate. As such, we will focus on the following hypotheses:

Hypothesis 1: *Upscaling increases the classification accuracy of Neural Networks.*

Hypothesis 2: *Upscaling reduces the fitting index of Neural Networks.*

Hypothesis 3: *Upscaling increases the classification accuracy of gradient boosted trees if one excludes the non upscaled samples.*

Hypothesis 4: *Upscaling reduces the fitting index of gradient boosted trees if one excludes the non upscaled samples.*

Why are we excluding the non upscaled samples from gradient boosted trees in the above hypotheses? As mentioned above, gradient boosted trees have an unfair advantage when it comes to the non upscaled samples. Including them would introduce a negative bias against Random Rotation Upscaling. Furthermore, in practice data sets are seldom in the perfect orientation or form.

To test these hypotheses, we fit two multiple linear regression models on the data seen in figure 4.4.1, which can be downloaded from Github [14]. Also, we allow for interaction in these models by multiplying all coefficients by the `Type` parameter. The result is the following R code:

```r
# Load the data and clear gradient boosted trees from non upscaled samples.
load("./data.Rdata")

data_clear <- data[which(data$Type == "NN"),]
data_clear <- rbind(data[intersect(which(data$Type == "GB"),
                                   which(data$Dimension > 10)),],
                    data_clear)

# Fit the two multiple linear regression models.
model_acc <- lm(Accuracy ~ Type + Dimension * Type + Splits * Type,
                data = data_clear)

model_index <- lm(sqrt(Index) ~ Type + Dimension * Type + Splits * Type,
                  data = data_clear)

summary(model_acc)
summary(model_index)
```

Note that we had to adjust the model regarding the fitting index. Instead of using $Index$, we used $\sqrt{Index}$ as the assumptions of multiple linear regression were more accurately satisfied. Also, we could have used the *Logit* transformation in the accuracy model, but for easier interpretation of the coefficients we refrained from doing so. Nevertheless, the relationship regarding the accuracy cannot be linear as that would allow for accuracies exceeding 100% when making predictions. As such, we have to be careful.

To check how well both models satisfy these assumptions, please refer to appendix B where the output of `plot(model_acc)` and `plot(model_index)` can be seen.

The last two lines of the previous script provide the following summaries as output.

```
Call:
lm(formula = Accuracy ~ Type + Dimension * Type + Splits * Type,
    data = data_clear)

Residuals:
      Min        1Q    Median        3Q       Max
-0.247694 -0.023209  0.000622  0.025788  0.188898

Coefficients:
                   Estimate Std. Error  t value Pr(>|t|)
(Intercept)       9.967e-01  9.915e-04 1005.236  < 2e-16 ***
TypeNN            1.063e-01  1.315e-03   80.838  < 2e-16 ***
Dimension         4.698e-05  1.642e-06   28.610  < 2e-16 ***
Splits           -2.294e-02  8.324e-05 -275.625  < 2e-16 ***
TypeNN:Dimension -1.145e-05  2.183e-06   -5.243 1.59e-07 ***
TypeNN:Splits    -6.589e-04  1.117e-04   -5.900 3.65e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.04237 on 48918 degrees of freedom
Multiple R-squared:  0.8301,        Adjusted R-squared:  0.8301
F-statistic: 4.779e+04 on 5 and 48918 DF,  p-value: < 2.2e-16
```

```
Call:
lm(formula = sqrt(Index) ~ Type + Dimension * Type + Splits *
```

```
    Type, data = data_clear)

Residuals:
     Min       1Q   Median       3Q      Max
-12.5535  -1.3351   0.0137   1.3266  11.5638

Coefficients:
                   Estimate Std. Error t value Pr(>|t|)
(Intercept)       1.340e+01  5.152e-02  260.00   <2e-16 ***
TypeNN           -1.487e+00  6.832e-02  -21.76   <2e-16 ***
Dimension        -9.089e-04  8.533e-05  -10.65   <2e-16 ***
Splits           -2.862e-01  4.326e-03  -66.17   <2e-16 ***
TypeNN:Dimension -4.311e-03  1.135e-04  -38.00   <2e-16 ***
TypeNN:Splits     4.270e-01  5.803e-03   73.58   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.202 on 48918 degrees of freedom
Multiple R-squared:  0.3228,        Adjusted R-squared:  0.3227
F-statistic:  4663 on 5 and 48918 DF,  p-value: < 2.2e-16
```

As we can see, all p-values regarding the coefficients are highly significant. Additionally, the models seem to account for most of the variance in the data as the `Adjusted R-squared` value is 0.83 and 0.32 respectively. The value regarding the fitting index seems especially low. However, this only matters if one is interested in the predictive accuracy of a model and not so much regarding general trends, which are relevant in our case.

Let us examine the 95% confidence intervals of the coefficients regarding both models. Similarly to the p-values, they are minuscule.

```
confint(model_acc)

                          2.5 %        97.5 %
(Intercept)       9.947348e-01  9.986215e-01
TypeNN            1.037147e-01  1.088690e-01
Dimension         4.375991e-05  5.019681e-05
Splits           -2.310703e-02 -2.278071e-02
TypeNN:Dimension -1.572645e-05 -7.167267e-06
TypeNN:Splits    -8.778454e-04 -4.400480e-04
```

```
confint(model_index)

                         2.5 %       97.5 %
(Intercept)      13.294294651 13.496254437
TypeNN           -1.620759802 -1.352928711
Dimension        -0.001076105 -0.000741629
Splits           -0.294715062 -0.277758950
TypeNN:Dimension -0.004533573 -0.004088818
TypeNN:Splits     0.415624974  0.438373985
```

So is there any interaction present in the linear models or are gradient boosted trees and Neural Networks equivalent in regards to the training data? If we look at the `summary(model_acc)` output above, we can see that the coefficient `TypeNN:Dimension` with value `-1.145e-05` is highly significant but tiny. As such, gradient boosted trees and Neural Net-

works differ in the change of accuracy per 100 dimensions of upscaling only by about 0.11% (±0.04% regarding a 95% confidence interval).

Nevertheless, if we look at the individual increase in accuracy which is given by the parameters `Dimension` for gradient boosted trees and `Dimension + TypeNN:Dimension` for Neural Networks, we obtain the following result. Upscaling the data by 100 dimensions yields an increase in accuracy by $\approx 0.47\% \pm 0.03\%$ for gradient boosted trees and by $\approx 0.35\% \pm 0.03\%$ for Neural Networks. Hence, we can positively conclude hypotheses 1 and 3.

What about the fitting index? This time, the interaction between the models is high enough to visualize it in figure 4.4.4. Note that we plotted the squared response as the model was fitted on $\sqrt{Index}$.

As we can see, by upscaling samples with 6 splits to dimension 500, we can expect a decrease in the fitting index of $\approx 58 \pm 2.5$ for Neural Networks and $\approx 10 \pm 2.8$ for gradient boosted trees. Note that since we square the response $\sqrt{Index}$ this interaction is not independent of the split number. However, the differences in fitting index after upscaling remain around those same numbers as we can see in figure 4.4.5.

Finally, by considering each model individually and the relevant p-values regarding parameters `Dimension` and `TypeNN:Dimension` in `summary(model_index)`, we can also positively conclude hypotheses 2 and 4.

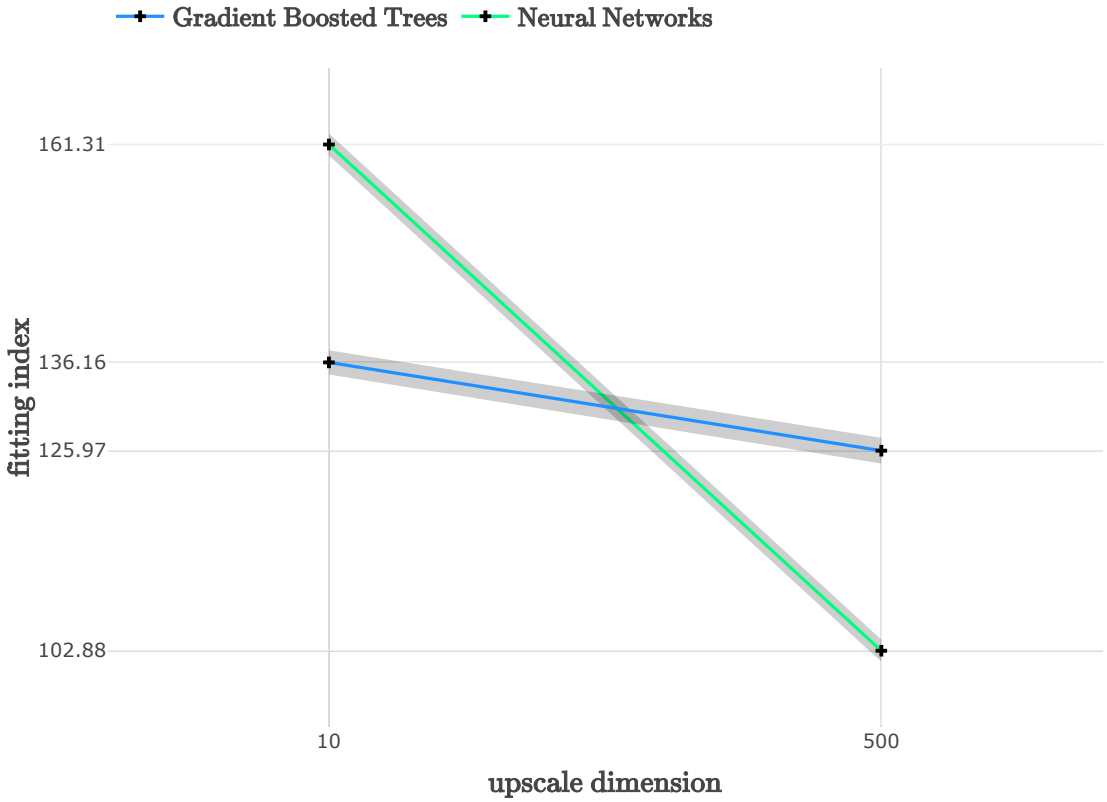

Figure 4.4.4: Interaction regarding data with 6 splits with 95% confidence intervals.

	5	6	7	8	9	10	11	12	13	14	15	16
NN	-58	-58	-59	-60	-61	-61	-62	-63	-63	-64	-65	-66
GB	-10	-10	-10	-10	-9	-9	-9	-9	-8	-8	-8	-8

Figure 4.4.5: Difference in fitting index for each split number.

Chapter 5

Summary and Conclusion

In the first two chapters, we introduced the general setting, Neural Networks and gradient boosted trees. The third chapter dealt with a simple practical application of these two methods regarding the classification of handwritten digits. The fourth dealt with Random Rotation Upscaling and the simulation study.

We have seen that Random Rotation Upscaling indeed does have an effect on the classification accuracy and fitting index of Neural Networks as well as gradient boosted trees. Surprisingly though, this effect is positive in nature. Namely, the classification accuracy increases and the fitting index decreases when upscaling to a higher dimension.

Though, an important thing to keep in mind is that we used a particular synthetic data set. It is not clear at all if these results carry over to other data sets and/or tasks like regression.

Nevertheless, Random Rotation Upscaling seems promising in regards to classification tasks with low dimensional but highly convoluted predictor space $\mathbb{R}^p$. Intuitively speaking, upscaling allows statistical methods to "see" the data from different perspectives, which may or may not influence them positively.

To examine these phenomenons further, we could look at the effect of Random Rotation Upscaling on different statistical methods and other data sets regarding tasks like regression. Presumably, real significance can only be established if the presented results prove themselves to be reproducible with real world data sets which are regretfully missing from this work.

Bibliography

[1] Sigrist, F. (2018, 08). Gradient and Newton Boosting for Classification and Regression. *arXiv e-prints*, arXiv:1808.03064.

[2] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65–386.

[3] Werbos, P. and P. John (1974, 01). Beyond regression: New tools for prediction and analysis in the behavioral sciences.

[4] Sanderson, G. aka. BlueBrown (2017). Youtube series on Neural Networks:
Part 1: *youtube.com/watch?v=aircAruvnKk*,
Part 2: *youtube.com/watch?v=IHZwWFHWa-w* and
Part 3: *youtube.com/watch?v=Ilg3gGewQ5U*.

[5] Hastie, T., R. Tibshirani, and J. Friedman (2001). *The Elements of Statistical Learning*. Springer.

[6] Kearns, M. and L. G. Valiant (1989). Crytographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, New York, NY, USA, pp. 433–444. ACM.

[7] Freund, Y. and R. E. Schapire (1997, 08). A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci. 55*(1), 119–139.

[8] Chollet, F. (2015). Keras: *keras.rstudio.com*.

[9] Chen, T. (2019). XGBoost: *xgboost.readthedocs.io*.

[10] Srestasathiern, P., S. Lawawirojwong, R. Suwantong, and P. Phuthong (2016, 06). Rotation matrix sampling scheme for multidimensional probability distribution transfer. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences III-7*, 103–110.

[11] Stewart, G. W. (1980). The efficient generation of random orthogonal matrices with an application to condition estimators. *SIAM Journal on Numerical Analysis 17*(3), 403–409.

[12] Kanatani, K. (1994, 06). Analysis of 3-d rotation fitting. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 16*, 543–549.

[13] ETH Zurich (2019). High Performance Computing Cluster Leonhard: *scicomp.ethz.ch/wiki/Leonhard*.

[14] Fomin, V. (2014). Github Repository: *github.com/fominv/Random-Rotation-Upscaling*.

Appendix A

R Code

The purpose of this appendix is to provide all R code necessary to reproduce the simulation study performed in section 4.4. Additionally, we present the used Neural Network architecture and Gradient Boosting parameters at the end.

To perform the aforementioned study, we can copy all the following code snippets in an empty directory and run the `simulation_study.R` script. To obtain the same data set as illustrated in figure 4.4.1, we can additionally run `construct_data_set.R`. Alternatively, the scripts can be downloaded from a Github repository [14].

We start off with the R equivalent of algorithm 4.1.1 which samples a random rotation matrix.

```r
# Save this script as sample_rotation.R!

library(keras)
library(Matrix)

sample_rotation <- function(n,sigma=1){
    # Sample a rotation matrix.

    # Define data structures.
    signums <-  list()
    H <-  list()

    for (i in n:2){
        # Define unit vector.
        e <- c(1,rep(0,i-1))

        # Sample from multivariate normal distribution.
        alpha <- rnorm(i, mean=0, sd=sigma)
        sign <- sign(alpha[1])

        # Compute Householder Transformation of a.
        v <- sign * sqrt(sum(alpha^2)) * e + alpha
        H_hat <- diag(i) - 2 * v %*% t(v) / sum(v^2)

        # Save signums.
        signums[[i]] <- sign((H_hat %*% alpha)[1])

        # Due to optimized multiplication save H_hat.
        H[[i]] <- H_hat
```

```
    }

    # Compute missing 1 dimensional householder transformation.
    signums[[1]] <- sign(rnorm(1,mean=0,sd=sigma))
    H[[1]] <- as.matrix(-1)

    # Sampled last diagonal element in contrast to original paper!
    D <- bdiag(rev(signums))

    # Modified multiplication to speed up computation time.
    mod_mult <- function(H_2, H_1){
        result <- H_2 %*% bdiag(1, H_1)
        return(result)
    }

    # Compute singular value decomposition.
    c(D,U,V) %<-% svd(D %*% Reduce(f = mod_mult, rev(H), right = TRUE))

    # Define rotation matrix.
    R <- as.matrix(U %*% bdiag(diag(n-1), det(U %*% t(V))) %*% t(V))

    return(R)
}

rotate <- function(points, rotation=NULL){
    # Rotate points or rbinded matrices of points.

    # Return originial point if no rotation is given.
    if(is.null(rotation)){return(point)}

    # Define rotation function regarding one point.
    rotate_point <- function(point){
        return(rotation %*% point)
    }

    # Transform to one dimenstionl matrix if input is only a sequence.
    if(is.null(dim(points))){
        points <- matrix(data = points, nrow=1)
    }

    # Transpose before return to output the same orientation of samples and dimensions.
    return(t(apply(points, 1,rotate_point)))
}
```

The next script represents algorithm 4.2.1 and provides a function to sample a grid.

```
# Save this script as sample_grid.R!

sample_grid <- function(layout){
    # Samples a grid according to layout sequence. Layout sets number of splits
    # regarding each dimension.

    # Sample the splits.
    grid <- lapply(layout-1, runif, min=-1,max=1)
    grid <- lapply(grid, sort)

    # Clear numeric(0) if no split was given to a dimension.
    for(i in 1:length(layout)){
        if(length(grid[[i]]) == 0){
            grid[[i]] <- NA
```

```
        }
    }

    return(grid)
}

assign_partition <- function(grid, point){
    # Assigns the corresponding partition region in a grid given a point in the
    # form of a sequence.

    # Define dimension and initialize data structure.
    n <- length(grid)
    partition_region <- numeric(n)

    for(i in 1:n){
    # Find the split point.
    which <- which(grid[[i]] < point[i])

    # Define the index of the split.
    if (length(which) == 0){
        index <- 1                          # 1 as numbering should start with 1.
    }else{
        index <- tail(which, n=1) + 1       # +1 as numbering should start with 1.
    }
    partition_region[i] <- index
    }

    return(partition_region)
}

sample_binary_labels <- function(layout){
    # Sample binary labels for each partition region. Expects a sequence.

    # Stop condition for recursion
    if(length(layout) == 0){
        return(sample(c(0,1),1))
    }

    # Define data structures.
    tree <- list()
    number_of_children <- layout[1]

    for(i in 1:number_of_children){
        # Recursive call to go to next dimension.
        tree[[i]] <- sample_binary_labels(layout[-1])
    }

    return(tree)
}
```

Next, is the equivalent of algorithm 4.2.2 by which we can obtain the initial data set. The mentioned variable `layout` specifies the number of partition regions (which are equal to splits plus one) in each dimension. However, we sample this layout in the simulation study instead of providing it explicitly.

```
# Save this script as sample_data.R!

source("./sample_grid.R")
source("./sample_rotation.R")
```

```
5
6   split_by_frac <- function(matrix, fractions){
7       # Split matrices row wise according to fractions.
8
9       # Values in fractions should sum up to one.
10      if(sum(fractions) != 1){
11          stop("Error: fractions need to sum up to one!")
12      }
13
14      # Check if all fractions provided
15      if(length(fractions) != 3){
16          stop("Error: You need to provide train, test and validation proportions.")
17      }
18
19      # Check if proportions in [0,1].
20      if({fractions[1] < 0 || fractions[2] < 0 || fractions[2] < 0 ||
21          fractions[1] > 1 || fractions[2] > 1 || fractions[2] > 1}){
22          stop("Error: Proportions must be in [0,1].")
23      }
24
25      # Check if resulting subrows when rounded are > 1.
26      if({ceiling(dim(matrix)[1] * fractions[1]) < 2 ||
27          ceiling(dim(matrix)[1] * fractions[2]) < 2 ||
28          ceiling(dim(matrix)[1] * fractions[3]) < 2}){
29          stop("Error: You need a higher sample size or bigger fractions.")
30      }
31
32      result <- list()
33      index_low <- 1
34      index_high <- 0
35
36      # Divide matrix into submatrices.
37      for(i in 1:length(fractions)){
38          index_high <- index_high + ceiling(dim(matrix)[1] * fractions[i])
39          result[[i]] <- matrix[index_low:index_high,]
40          index_low <- index_high + 1
41      }
42
43      return(result)
44  }
45
46  sample_samples <- function(dimension,
47                             size,
48                             train_prop,
49                             test_prop,
50                             val_prop){
51      # Sample data in [-1,1]^dimension
52      data <- matrix(nrow = size, ncol = dimension)
53
54      for(i in 1:size){
55        data[i,] <- sapply(rep(1,dimension), runif, min=-1, max=1)
56      }
57
58      # Split the result by fractions.
59      return(split_by_frac(data, c(train_prop, test_prop, val_prop)))
60  }
61
62  sample_data <- function(dimension,
63                             size,
64                             layout,
```

```
                        train_prop,
                        test_prop,
                        val_prop){
    # Sample data according to split numbers defined in layout.

    # Sample data.
    samples <- sample_samples(dimension, size, train_prop, test_prop, val_prop)

    # Sample the grid and binary labels.
    grid <- sample_grid(layout)
    labels <- sample_binary_labels(layout)

    # Assign labels to train/test/val data.
    sample_labels <- list(numeric(dim(samples[[1]])[1]),
                          numeric(dim(samples[[2]])[1]),
                          numeric(dim(samples[[3]])[1]))

    for(i in 1:3){
        for(j in 1:dim(samples[[i]])[1]){
            partition <- assign_partition(grid, samples[[i]][j,])
            sample_labels[[i]][j] <- labels[[partition]]
        }
    }

    # Construct nested data list.
    data <- list(train = list(data = samples[[1]], labels = sample_labels[[1]]),
                 test = list(data = samples[[2]], labels = sample_labels[[2]]),
                 val = list(data = samples[[3]], labels = sample_labels[[3]]))

  return(data)
}

sample_data_capped <- function(dimension,
                               size,
                               cap,
                               train_prop,
                               test_prop,
                               val_prop){
    # Sample data but instead of providing a split layout, we provide a cap.

    # Sample a layout.
    splits <- c(1, sample(1:cap+1, dimension - 1, replace = TRUE), cap+1)
    splits <- sort(splits)
    layout <- diff(splits) + 1

    return(sample_data(dimension,
                       size,
                       layout,
                       train_prop,
                       test_prop,
                       val_prop))
}
```

As mentioned, we face a numerical problem when sampling a rotation matrix. Namely the singular value decomposition may fail in rare cases (around 4 in 500). During the study we recorded the intersection of all seeds where the decomposition succeeded. These seeds are provided below.

```
# Save this script as seeds.R!
```

```

get_seeds <- function(){
    failed_seeds <- c( 2, 5, 8, 9, 32, 58, 84, 94, 96, 118, 125, 128, 146, 149, 160,
    170, 172, 176, 186, 189, 208, 211, 214, 221, 231, 237, 250, 251, 268, 270, 309, 319,
    346, 353, 355, 360, 366, 388, 410, 415, 423, 431, 456, 460, 469, 487, 494, 496)
    return(setdiff(1:501, failed_seeds))
}
```

Finally, the simulation study can be performed with the following script. To run a quick test of the simulation, refer to the comment regarding dimensions, cap and seeds.

```
# Save this script as simulation_study.R!

library(keras)
library(xgboost)

source("./sample_data.R")
source("./seeds.R")

# Define dimensions, cap and seeds.
dimensions <- c(10,50,100,250,500)
cap <- 5:16
seeds <- get_seeds()

# Test Run dimensions, cap and seeds. Uncomment to the test the code!
# dimensions <- c(10,50)
# cap <- 10:11
#Âãseeds <- get_seeds()[c(1,2)]

# Define sample parameters.
size = 12500
train_prop = 0.8
val_prop = 0.1
test_prop = 0.1

# Define relevant data structures.
simulation_output <- list()

gb_result <- list(test = list(),
                  accuracy = list(),
                  max_accuracy = list(),
                  max_index = list())
nn_result <- list(test = list(),
                  accuracy = list(),
                  max_accuracy = list(),
                  max_index = list())

for(i in 1:length(dimensions)){
    gb_result$test[[i]] <- list()
    gb_result$accuracy[[i]] <- list()
    gb_result$max_accuracy[[i]] <- numeric(length(seeds))
    gb_result$max_index[[i]] <- numeric(length(seeds))
    nn_result$test[[i]] <- list()
    nn_result$accuracy[[i]] <- list()
    nn_result$max_accuracy[[i]] <- numeric(length(seeds))
    nn_result$max_index[[i]] <- numeric(length(seeds))
}

# Define accuracy of test data for xgboost.
calculate_accuracy <- function(model, new_data){
```

```
    prediction <- predict(model, new_data$data)
    prediction <- as.numeric(prediction > 0.5)
    error <- mean(prediction != new_data$label)
    return(1-error)
}

# Define xgboost model parameters.
boosting_rounds <- 2000

args <- list()

args$max.depth <- 8
args$eta <- 0.1
args$nthread <- 2
args$nrounds <- boosting_rounds
args$subsample <- 0.8
args$verbose <- 0
args$objective <- "binary:logistic"

# For gpu support uncomment
# args£tree_method <- "gpu_hist"

args$early_stopping_rounds <- 50
args$xgb_model <- NULL

# Function to initialize the keras model.
init_nn_model <- function(dimension){
    model <- keras_model_sequential() %>%
        layer_dense(units = 500, activation = "sigmoid",
                    input_shape = c(dimension)) %>%
        layer_dense(units = 250, activation = "relu") %>%
        layer_dense(units = 100, activation = "relu") %>%
        layer_dense(units = 50, activation = "relu") %>%
        layer_dense(units = 1, activation = "sigmoid")

    model %>% compile(
        optimizer = "rmsprop",
        loss = "binary_crossentropy",
        metrics = c("accuracy")
    )

    return(model)
}

# Define additional keras parameters.
epochs = 2000
batch_size = 500

# Loop of over all configurations.
for(c in cap){
for(j in 1:length(seeds)){
    # Set the appropriate seed.
    use_session_with_seed(seeds[j],
                          disable_gpu = FALSE,
                          disable_parallel_cpu = FALSE)

    # Sample the data and rotated data.
    data <- list()

    # dimensions[[1]] data.
```

```
    c(train, test, val) %<-% sample_data_capped(dimensions[1], size, c,
                                                train_prop, test_prop, val_prop)

    data[[1]] <- list(train = train, test = test, val = val)

    # Prepare rotated data.
    data_index <- 2
    for(dim in dimensions[-1]){
        train$data <- cbind(
            data[[1]]$train$data,
            matrix(0, nrow = dim(data[[1]]$train$data)[1],
                   ncol = dim - dimensions[1]))
        test$data <- cbind(
            data[[1]]$test$data,
            matrix(0, nrow = dim(data[[1]]$test$data)[1],
                   ncol = dim - dimensions[1]))
        val$data <- cbind(
            data[[1]]$val$data,
            matrix(0, nrow = dim(data[[1]]$val$data)[1],
                   ncol = dim - dimensions[1]))

        ### Sample the rotation.
        rotation <- sample_rotation(dim)

        train$data <- t(apply(train$data, 1,rotate, rotation=rotation))
        test$data <- t(apply(test$data, 1,rotate, rotation=rotation))
        val$data <- t(apply(val$data, 1,rotate, rotation=rotation))

        data[[data_index]] <- list(train = train, test = test, val = val)
        data_index <- data_index + 1
    }

    # Train xgboost and keras.
    # Loop over the different dimensions.
    for(i in 1:length(data)){
        # XGboost.
        # Define missing model parameters.
        args$data <- xgb.DMatrix(data = data[[i]]$train$data,
                                 label = data[[i]]$train$labels)

        val <- xgb.DMatrix(data = data[[i]]$val$data,
                           label = data[[i]]$val$labels)

        args$watchlist <- list(train=args$data, test=val)

        # Fit the model.
        gb_model <- do.call(xgb.train, args)

        # Exctract relevant data.
        gb_result$accuracy[[i]][[j]] <- 1 - gb_model$evaluation_log$test_error
        gb_result$test[[i]][[j]] <- calculate_accuracy(gb_model, test)

        gb_result$max_accuracy[[i]][j] <- max(
            1 - gb_model$evaluation_log$test_error)
        gb_result$max_index[[i]][j] <- which.max(
            1 - gb_model$evaluation_log$test_error)

        # Keras.
        # Initialize Neural Network.
        nn_model <- init_nn_model(dimensions[i])
```

```
170
171         # Fit the model.
172         nn_model %>% fit(data[[i]]$train$data,
173                          data[[i]]$train$labels,
174                          epochs = epochs,
175                          batch_size = batch_size,
176                          validation_data = list(data[[i]]$val$data,
177                                                 data[[i]]$val$labels),
178                          verbose = 0,
179                          callbacks = list(callback_early_stopping(
180                              monitor = "val_acc",
181                              min_delta = 0.01,
182                              patience = 50,
183                              verbose = 1)))
184
185         # Exctract relevant data.
186         nn_result$accuracy[[i]][[j]] <- nn_model$history$history$val_acc
187         nn_result$test[[i]][[j]] <- evaluate(nn_model,
188                                              data[[i]]$test$data,
189                                              data[[i]]$test$labels)
190
191         nn_result$max_accuracy[[i]][j] <- max(
192             as.numeric(nn_model$history$history$val_acc))
193         nn_result$max_index[[i]][j] <- which.max(
194             as.numeric(nn_model$history$history$val_acc))
195     }
196 }
197
198 # Combine all results into final output.
199 simulation_output[[c - cap[1] + 1]] <- list(NN = nn_result, GB = gb_result)
200 }
201
202 # Save relevant variables for further processing.
203 save(list = c("simulation_output",
204               "seeds",
205               "cap",
206               "dimensions"),
207      file = "./simulation_output.Rdata")
```

To produce the data set seen in figure 4.4.1, run the following final script.

```
1  # Save this script as simulation_study.R!
2
3  library(keras)
4  library(xgboost)
5
6  source("./sample_data.R")
7  source("./seeds.R")
8
9  # Define dimensions, cap and seeds.
10 dimensions <- c(10,50,100,250,500)
11 cap <- 5:16
12 seeds <- get_seeds()
13
14 # Test Run dimensions, cap and seeds. Uncomment to the test the code!
15 # dimensions <- c(10,50)
16 # cap <- 10:11
17 #Âãseeds <- get_seeds()[c(1,2)]
18
19 # Define sample parameters.
```

```
size = 12500
train_prop = 0.8
val_prop = 0.1
test_prop = 0.1

# Define relevant data structures.
simulation_output <- list()

gb_result <- list(test = list(),
                  accuracy = list(),
                  max_accuracy = list(),
                  max_index = list())
nn_result <- list(test = list(),
                  accuracy = list(),
                  max_accuracy = list(),
                  max_index = list())

for(i in 1:length(dimensions)){
    gb_result$test[[i]] <- list()
    gb_result$accuracy[[i]] <- list()
    gb_result$max_accuracy[[i]] <- numeric(length(seeds))
    gb_result$max_index[[i]] <- numeric(length(seeds))
    nn_result$test[[i]] <- list()
    nn_result$accuracy[[i]] <- list()
    nn_result$max_accuracy[[i]] <- numeric(length(seeds))
    nn_result$max_index[[i]] <- numeric(length(seeds))
}

# Define accuracy of test data for xgboost.
calculate_accuracy <- function(model, new_data){
    prediction <- predict(model, new_data$data)
    prediction <- as.numeric(prediction > 0.5)
    error <- mean(prediction != new_data$label)
    return(1-error)
}

# Define xgboost model parameters.
boosting_rounds <- 2000

args <- list()

args$max.depth <- 8
args$eta <- 0.1
args$nthread <- 2
args$nrounds <- boosting_rounds
args$subsample <- 0.8
args$verbose <- 0
args$objective <- "binary:logistic"

# For gpu support uncomment
# args£tree_method <- "gpu_hist"

args$early_stopping_rounds <- 50
args$xgb_model <- NULL

# Function to initialize the keras model.
init_nn_model <- function(dimension){
    model <- keras_model_sequential() %>%
        layer_dense(units = 500, activation = "sigmoid",
                    input_shape = c(dimension)) %>%
```

```
        layer_dense(units = 250, activation = "relu") %>%
        layer_dense(units = 100, activation = "relu") %>%
        layer_dense(units = 50, activation = "relu") %>%
        layer_dense(units = 1, activation = "sigmoid")

    model %>% compile(
        optimizer = "rmsprop",
        loss = "binary_crossentropy",
        metrics = c("accuracy")
    )

    return(model)
}

# Define additional keras parameters.
epochs = 2000
batch_size = 500

# Loop of over all configurations.
for(c in cap){
for(j in 1:length(seeds)){
    # Set the appropriate seed.
    use_session_with_seed(seeds[j],
                          disable_gpu = FALSE,
                          disable_parallel_cpu = FALSE)

    # Sample the data and rotated data.
    data <- list()

    # dimensions[[1]] data.
    c(train, test, val) %<-% sample_data_capped(dimensions[1], size, c,
                                                train_prop, test_prop, val_prop)

    data[[1]] <- list(train = train, test = test, val = val)

    # Prepare rotated data.
    data_index <- 2
    for(dim in dimensions[-1]){
        train$data <- cbind(
            data[[1]]$train$data,
            matrix(0, nrow = dim(data[[1]]$train$data)[1],
                   ncol = dim - dimensions[1]))
        test$data <- cbind(
            data[[1]]$test$data,
            matrix(0, nrow = dim(data[[1]]$test$data)[1],
                   ncol = dim - dimensions[1]))
        val$data <- cbind(
            data[[1]]$val$data,
            matrix(0, nrow = dim(data[[1]]$val$data)[1],
                   ncol = dim - dimensions[1]))

        ### Sample the rotation.
        rotation <- sample_rotation(dim)

        train$data <- t(apply(train$data, 1,rotate, rotation=rotation))
        test$data <- t(apply(test$data, 1,rotate, rotation=rotation))
        val$data <- t(apply(val$data, 1,rotate, rotation=rotation))

        data[[data_index]] <- list(train = train, test = test, val = val)
        data_index <- data_index + 1
```

```
    }

    # Train xgboost and keras.
    # Loop over the different dimensions.
    for(i in 1:length(data)){
        # XGboost.
        # Define missing model parameters.
        args$data <- xgb.DMatrix(data = data[[i]]$train$data,
                                 label = data[[i]]$train$labels)

        val <- xgb.DMatrix(data = data[[i]]$val$data,
                           label = data[[i]]$val$labels)

        args$watchlist <- list(train=args$data, test=val)

        # Fit the model.
        gb_model <- do.call(xgb.train, args)

        # Exctract relevant data.
        gb_result$accuracy[[i]][[j]] <- 1 - gb_model$evaluation_log$test_error
        gb_result$test[[i]][[j]] <- calculate_accuracy(gb_model, test)

        gb_result$max_accuracy[[i]][j] <- max(
            1 - gb_model$evaluation_log$test_error)
        gb_result$max_index[[i]][j] <- which.max(
            1 - gb_model$evaluation_log$test_error)

        # Keras.
        # Initialize Neural Network.
        nn_model <- init_nn_model(dimensions[i])

        # Fit the model.
        nn_model %>% fit(data[[i]]$train$data,
                         data[[i]]$train$labels,
                         epochs = epochs,
                         batch_size = batch_size,
                         validation_data = list(data[[i]]$val$data,
                                                data[[i]]$val$labels),
                         verbose = 0,
                         callbacks = list(callback_early_stopping(
                             monitor = "val_acc",
                             min_delta = 0.01,
                             patience = 50,
                             verbose = 1)))

        # Exctract relevant data.
        nn_result$accuracy[[i]][[j]] <- nn_model$history$history$val_acc
        nn_result$test[[i]][[j]] <- evaluate(nn_model,
                                             data[[i]]$test$data,
                                             data[[i]]$test$labels)

        nn_result$max_accuracy[[i]][j] <- max(
            as.numeric(nn_model$history$history$val_acc))
        nn_result$max_index[[i]][j] <- which.max(
            as.numeric(nn_model$history$history$val_acc))
    }
}

# Combine all results into final output.
simulation_output[[c - cap[1] + 1]] <- list(NN = nn_result, GB = gb_result)
```

```
}

# Save relevant variables for further processing.
save(list = c("simulation_output",
              "seeds",
              "cap",
              "dimensions"),
     file = "./simulation_output.Rdata")
```

In the previous scripts, we can see the used Neural Network structure and Gradient Boosting parameters. For easier view, we provide these parameters once more in distinct scripts.

```
# Function to initialize the keras model.
init_nn_model <- function(dimension){
    model <- keras_model_sequential() %>%
        layer_dense(units = 500, activation = "sigmoid",
                    input_shape = c(dimension)) %>%
        layer_dense(units = 250, activation = "relu") %>%
        layer_dense(units = 100, activation = "relu") %>%
        layer_dense(units = 50, activation = "relu") %>%
        layer_dense(units = 1, activation = "sigmoid")

    model %>% compile(
        optimizer = "rmsprop",
        loss = "binary_crossentropy",
        metrics = c("accuracy")
    )

    return(model)
}

# Initialize Neural Network.
nn_model <- init_nn_model(dimensions[i])

# Fit the model.
nn_model %>% fit(data[[i]]$train$data,
                 data[[i]]$train$labels,
                 epochs = 2000,
                 batch_size = 500,
                 validation_data = list(data[[i]]$val$data,
                                        data[[i]]$val$labels),
                 verbose = 0,
                 callbacks = list(callback_early_stopping(
                     monitor = "val_acc",
                     min_delta = 0.01,
                     patience = 50,
                     verbose = 1)))
```

```
# Define xgboost model parameters.
boosting_rounds <- 2000

args <- list()

args$max.depth <- 8
args$eta <- 0.1
args$nthread <- 2
args$nrounds <- boosting_rounds
args$subsample <- 0.8
```

```
args$verbose <- 0
args$objective <- "binary:logistic"
args$early_stopping_rounds <- 50
args$xgb_model <- NULL

# Define missing model parameters.
args$data <- xgb.DMatrix(data = data[[i]]$train$data,
                         label = data[[i]]$train$labels)

val <- xgb.DMatrix(data = data[[i]]$val$data,
                   label = data[[i]]$val$labels)

args$watchlist <- list(train=args$data, test=val)

# Fit the model.
gb_model <- do.call(xgb.train, args)
```

Appendix B

Linear Model Assumptions

This appendix aims to provide context to the models used in the simulation study. It consists of two sections which correspond to `model_acc` and `model_index` respectively. Each section contains plots which can be produced by the command `plot.lm(model)`. These plots show if the usual preconditions to apply multiple linear regression are satisfied.

B.1 Plots of Classification Accuracy Model

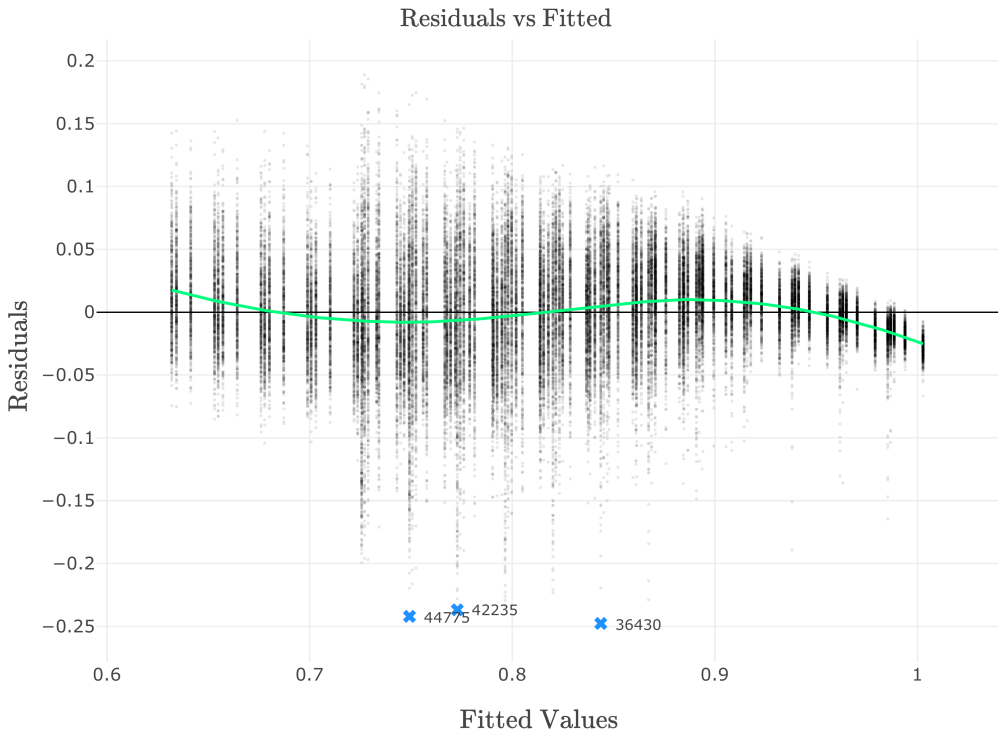

Figure B.1.1: Residuals agains fitted values of `model_acc`.

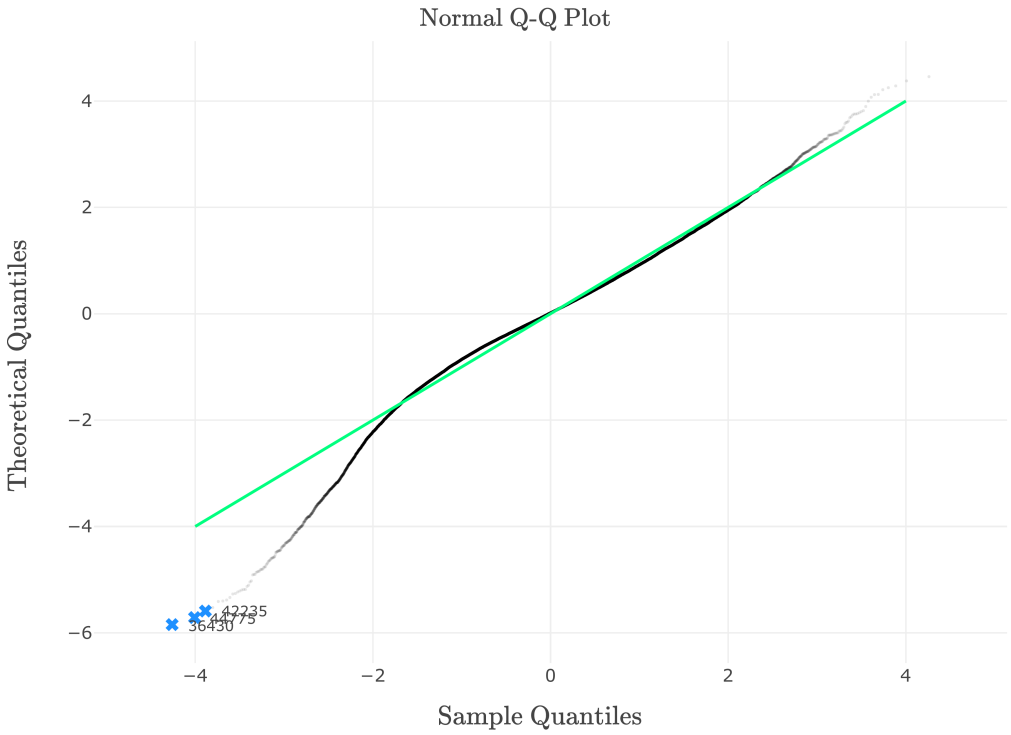

Figure B.1.2: Q-Q plot of `model_acc`.

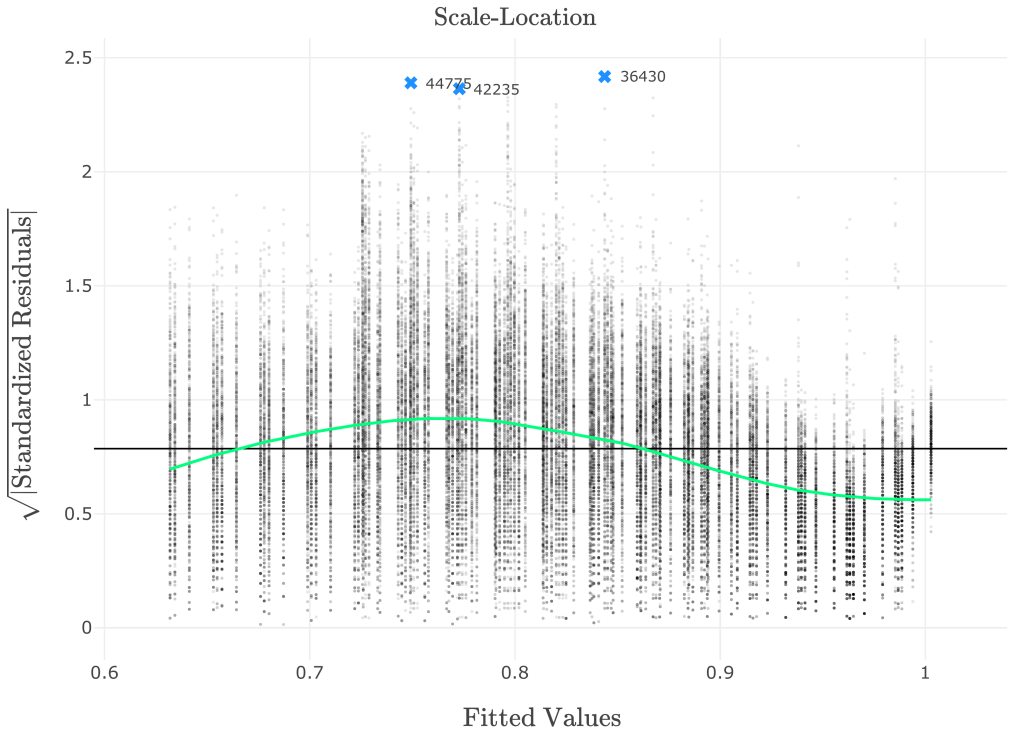

Figure B.1.3: Standardized residuals plot of `model_acc`.

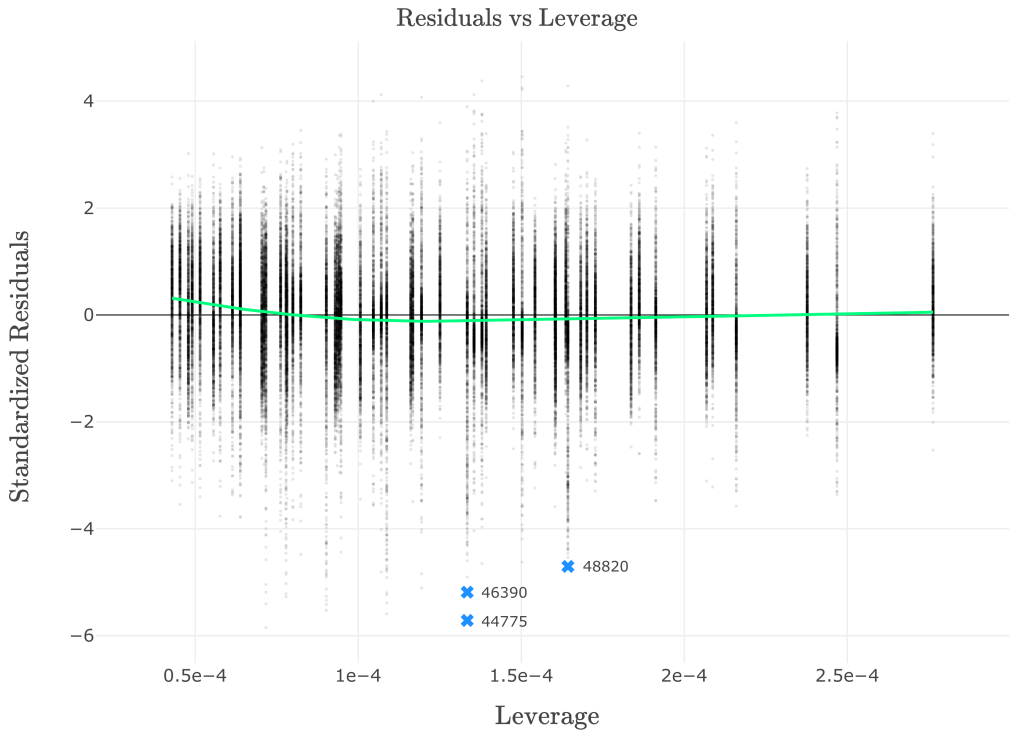

Figure B.1.4: Residuals against leverage of `model_acc`.

B.2 Plots of Fitting Index Model

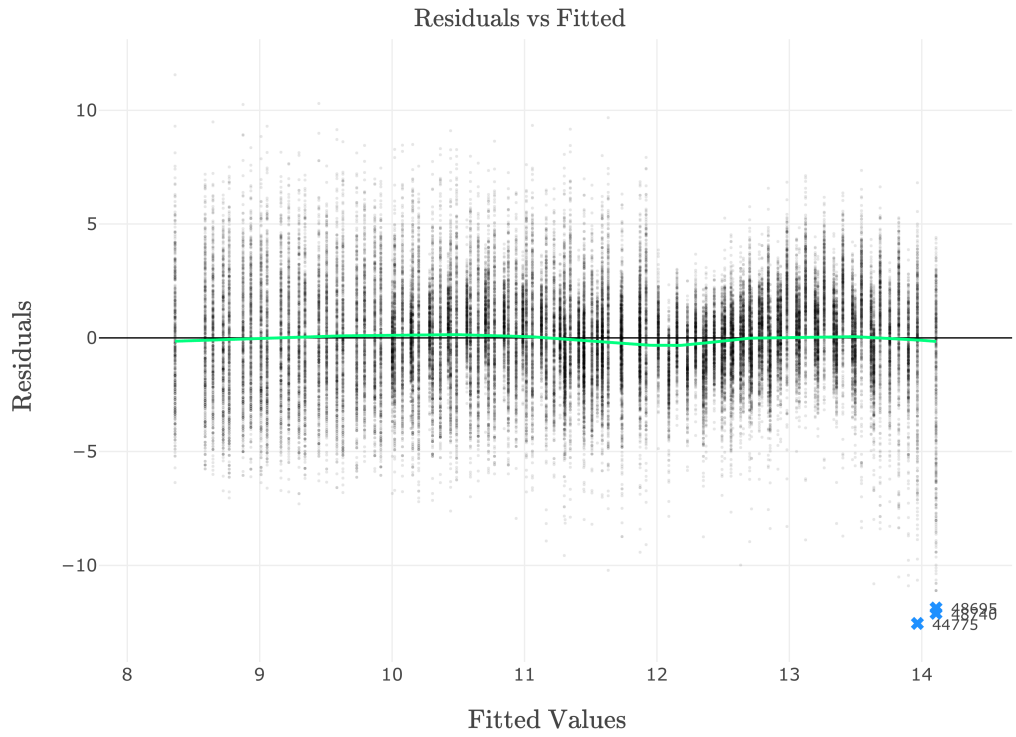

Figure B.2.1: Residuals agains fitted values of `model_index`.

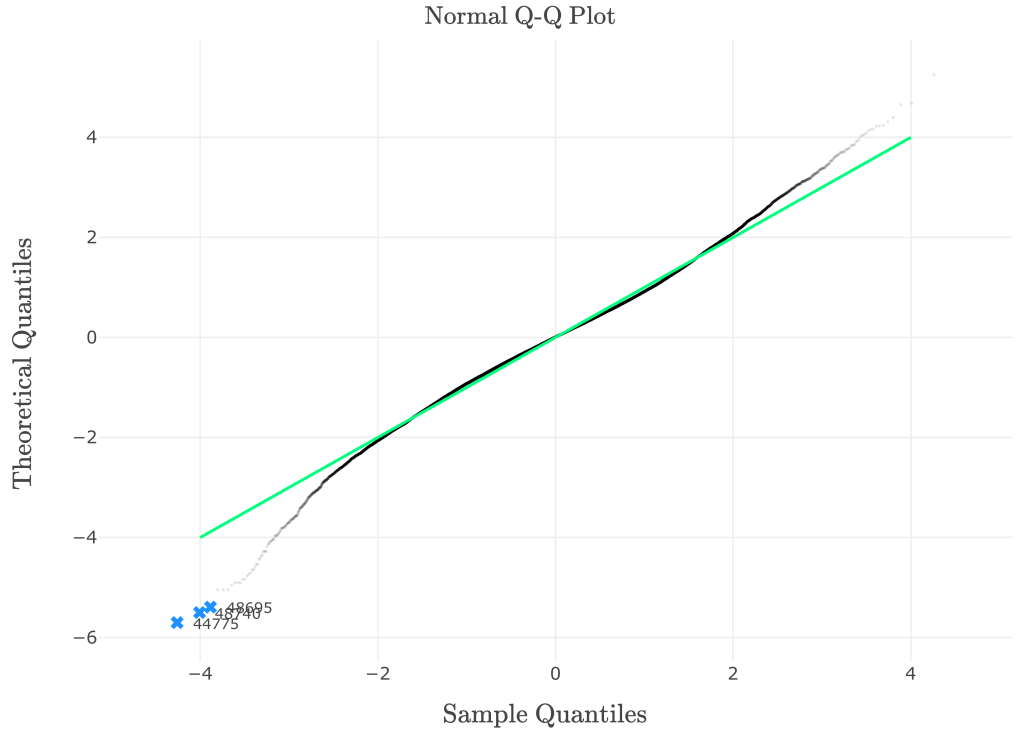

Figure B.2.2: Q-Q plot of `model_index`.

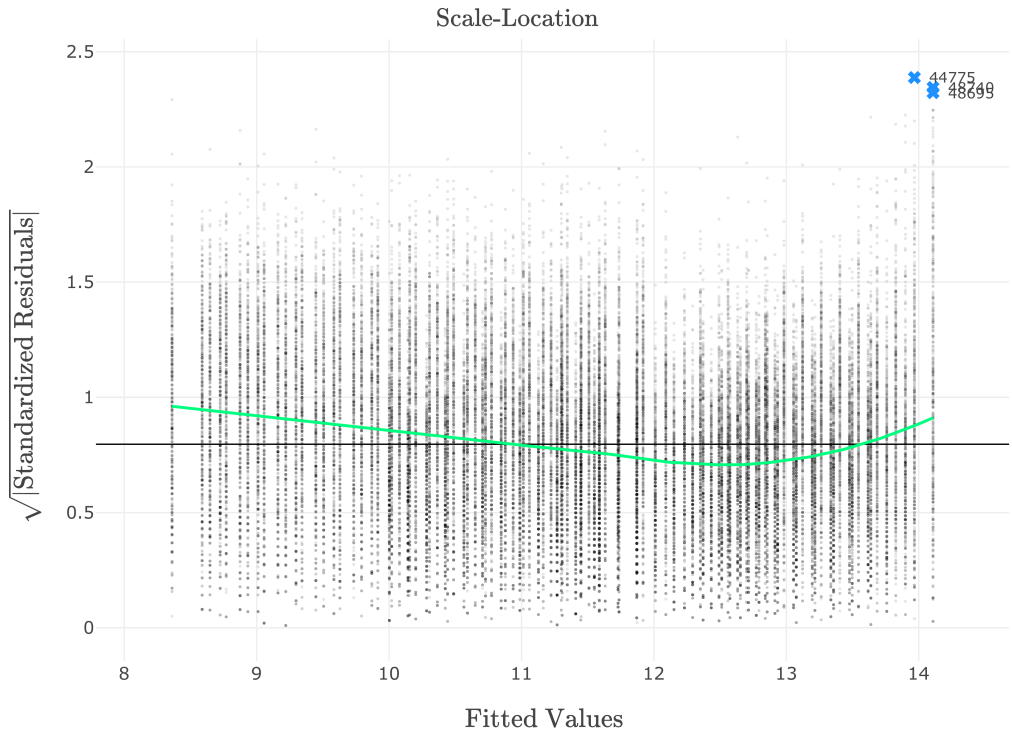

Figure B.2.3: Standardized residuals plot of `model_index`.

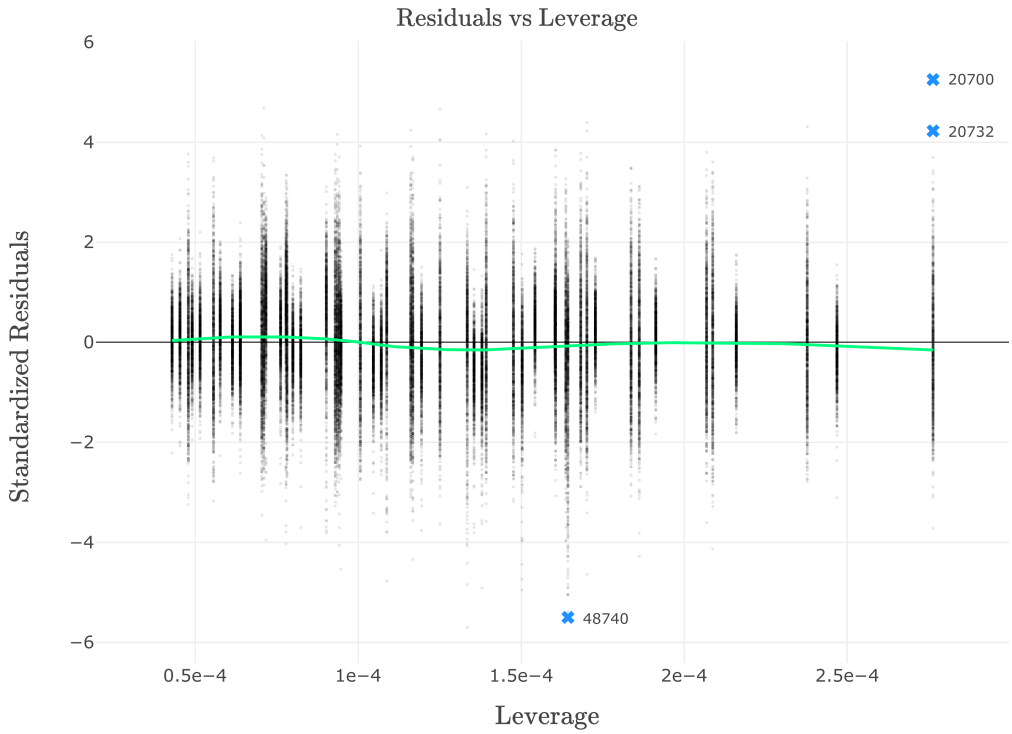

Figure B.2.4: Residuals against leverage of `model_index`.

Declaration of Originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor .

Title of work (in block letters):

THE EFFECT OF RANDOM ROTATION UPSCALING ON NEURAL NETWORKS AND GRADIENT BOOSTED TREES

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
FOMIN	VLADIMIR

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the Citation etiquette information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work .
- I am aware that the work may be screened electronically for plagiarism.
- I have understood and followed the guidelines in the document *Scientific Works in Mathematics.*

Place, date:	**Signature(s):**
8048 Zürich, 5.8.19	Vladimir Fomin

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper