CHALMERS
UNIVERSITY OF TECHNOLOGY

Predicting Antibiotic Resistance Phenotypes using Neural Networks

Master’s thesis in engineering mathematics and computational science

HAMPUS LANE

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020:NN

Predicting antibiotic resistance phenotypes using neural networks

HAMPUS LANE

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2020

Predicting antibiotic resistance phenotypes using neural networks
HAMPUS LANE

Supervisor & Examiner: Erik Kristiansson, Department of Mathematical Sciences
Supervisor: Anna Johnning, Department of Mathematical Sciences

Master's Thesis 2020:NN
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Predicting antibiotic resistance phenotypes using neural networks
HAMPUS LANE
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

Antibiotic resistance is one of the biggest threats to human health. Today it causes 700 000 deaths per year, a number that is estimated to rise to 10 million by 2050. Measuring antibiotic resistance is time consuming and while waiting patients may receive ineffective treatment. It is thus interesting to see what previous measurements can tell us about antibiotics that have not been measured. In this thesis neural network based imputation is investigated as a suitable method for prediction of antibiotic resistant phenotypes. This was done using comprehensive data collected by the EARS-Net. The data contains bacteria from 330000 patients that have been tested for resistance against a range of different antibiotics. In this thesis neural networks were trained to predict missing resistant and susceptible phenotypes based on available resistance data. In addition the networks had access to information about the pathogen and the reporting country. After tuning the number of nodes and training the networks on 85% of the available data we received an average error rate of 5% when testing on the remaining data. The average very major error rate, i.e. the error rate on measurements whose true value were resistant, was 11.8%. The error rates varied for different antibiotics and some had very major error rates below 5%. Neural network based imputation shows promise as a method for predicting antibiotic phenotypes and with further research into what affects the neural networks performance it could be a useful tool when measuring antibiotic resistance.

Keywords: Antibiotic Resistance, Neural Network, Imputation, Categorical Variables.

Contents

List of Figures

1 Introduction

Antibiotic resistance is one of the top ten threats to human health according to the World Health Organisation [1]. Bacteria become resistant to antibiotics when they acquire genes that encode for mechanisms that prevents the antibiotic from functioning, for example enzymes that degrade the antibiotic or prevents it from binding. Each such gene typically gives resistance to one group of antibiotics without affecting resistance towards other groups. Several resistance genes can often be locate on the same plasmid, which is shared between different bacteria. This way resistance to one type of antibiotic can influence the probability of being resistance to other antibiotics.

To treat patients infected with antibiotic resistant bacteria and to surveil and research antibiotic resistance it is important to measure antibiotic resistance in bacteria. This measurement is typically done by cultivating the bacteria in the presence of an antibiotic and seeing how well it grows [2]. If the bacteria grows well it is classified as resistant, if it exhibits reduced growth it is classified as intermediate and if there is no or sufficiently reduced growth it is classified as susceptible. The growing process can be very time consuming especially if a large number of antibiotics have to be tested. It is thus of interest to use these dependencies between antibiotic resistance to try to predict the resistance status against other antibiotics, in order to make a more informed choice of which antibiotics to test next or, if the prediction is strong enough, skip some measurements all together.

In this thesis we set out to solve the problem of predicting a bacteria's resistance against an antibiotic given previously measured resistance/susceptibilities to other antibiotics. To tackle this the problem will be treated as an imputation problem. Imputation is the process of filling in missing data. In this case the available data will be the results of the performed tests and the missing data is whether the bacteria is resistant against the antibiotics that have not been tested. This thesis specifically focuses on a neural network based method of imputation. The goal of this thesis is to answer the question "How well can we predict resistance against an antibiotic using previously measured resistance/susceptibilities to other antibiotics using neural network based imputation?"

To train the neural networks to predicting resistance it is necessary to have data. This data is provided by the European Centre for Disease Control (ECDC) and comes from the European Antimicrobial Resistance Surveillance Network (EARS-Net). The data consists of measurements on bacteria from around 330 000 patients. Each measurement consists of checking one bacterial strains for resistance against one antibiotic. Each bacteria was tested against on average 7.6 antibiotics giving a total of 2.5 million measurements. More information about the data can be found in Section 3.1.

2
Theory and Method

The goal of this thesis is to predict resistance to some antibiotics given information about other antibiotics. The available data is going to contain measurement on a large number of bacteria. Each bacteria have been tested against approximately 8 antibiotics out of 44. If this data is stored in matrix with antibiotics as columns and bacteria as rows, each row is going to contain a handful of entries and the rest of the data will be missing. Trying to predict missing data like this is called imputation and it is one way to tackle the problem at hand. Some imputation methods will be reviewed in section 2.2, especially an imputation technique using neural networks.

As the main imputation method used in this thesis is based on neural networks, section 2.1 will discuss the theory behind neural networks as well as the specific implementation used.

2.1 Neural Networks

An artificial neural networks is a computational model that is based on the neurons in the brain. A biological neuron basically works as follows [3]: Every neuron is connected to a number of other neurons via synapses. When the electrical potential in a neuron gets high enough the neuron sends out a signal in the form of an electrical current to the neurons its connected to. For some types of synapses this current increases the potential of the connected neuron and make it more likely to emit a signal. For other types of synapses the current will decrease the potential of the connected neuron and suppress a signal.

In an artificial neural network we instead have nodes connected by weights. The nodes have a value that is updated in discrete time steps. Each node then gets an input from the nodes that are connected to it. This input is the value of the node multiplied by a weight. The input is then summed, adjusted by a node specific threshold and put into an activation function. This activation function has traditionally been sigmoid making the result similar to on-off nature of the neuron while remaining smooth, but functions similar to the rectifier, $f(x) = x^+ = \max(0, x)$, have also become heavily used. [4]

A neural network can be used as a function with an input and and an output. This is typically done with a feed forward network. In section 2.1.1 one can read about what such a network looks like and how it calculates the output from the input. A

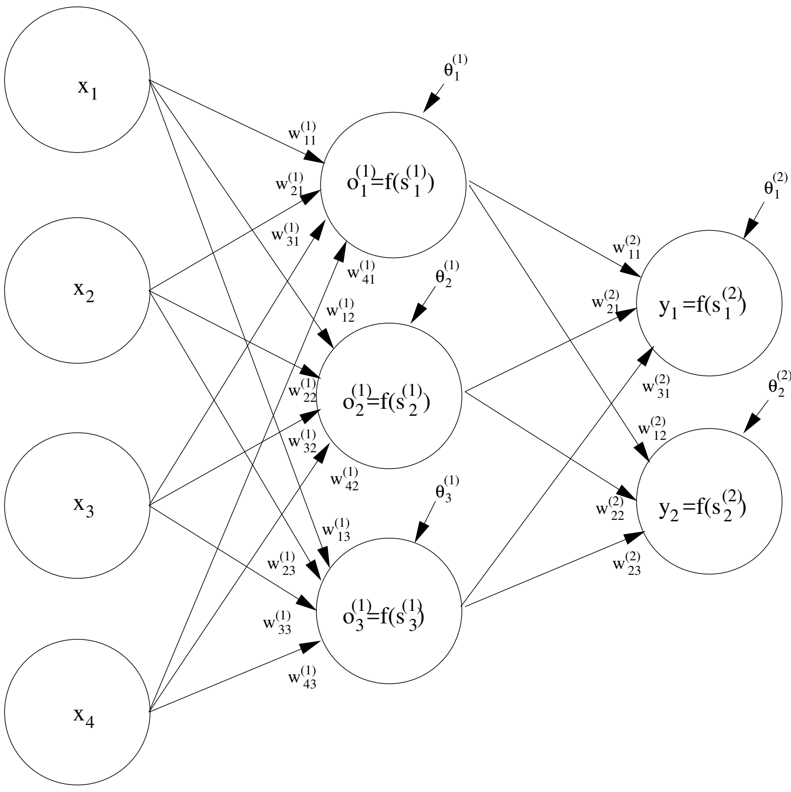


Figure 2.1: Example of a feedforward network with one hidden layer.

neural network can also be "trained" by being given sets of inputs and outputs, so that it will give similar output when given similar input. How this training works is discussed in section 2.1.2.

2.1.1 Feedforward Neural Network

In a feedforward neural network the nodes are organised in layers. A node is connected to every node in the next layer and no other nodes. So if we put values in the first layer of nodes, the first update step will fill the second layer with values, without changing any other values. The second update step will only fill the third layer and so on. This means we can put our function variables as values in the nodes in the first layer and get the values in the final layer as a function output. For this reason the first layer is called the input layer the final layer is called the output layer and layers in between are called hidden layers a. An example of a feedforward network with one hidden layer can be seen in Figure 2.1. In the input layer the value of node i is set to the value of input variable i, this value is called x_i. The value of node i in layer l is denoted $o_i^{(l)}$. The input layer is denoted as the 0:th layer thus x_i can also be denoted $o_i^{(0)}$. The weight from node i in layer $l-1$ to node j in layer l is denoted $w_{ij}^{(l)}$. The input from the previous layer is thus $\sum_k o_k^{(l-1)} w_{ki}^{(l)}$ and is known as the local field. The threshold, or bias, of node i in layer l is denoted $\theta_i^{(l)}$ and the number of nodes in layer l is denoted $n^{(l)}$. We can now describe the input to node i of layer l, $s_i^{(l)}$ as

$$s_i^{(l)} = -\theta_i^{(l)} + \sum_{k=1}^{n^{(l-1)}} o_k^{(l-1)} w_{ki}^{(l)}. \tag{2.1}$$

The value of that node is then calculated by applying the activation function. For the output node one often uses a different or no activation based on what kind of output is desired.

$$o_i^{(l)} = f\left(s_i^{(l)}\right) = f\left(-\theta_i^{(l)} + \sum_{k=1}^{n^{(l-1)}} o_k^{(l-1)} w_{ki}^{(l)}\right). \tag{2.2}$$

Using those one can assign the value of a node based only on the values in the nodes in the previous layer. As a result the layers can be filled one at a time, getting the values in the output layer to use as our function values.

This shows how to use neural networks as functions with numerical input and outputs, but in this thesis the main focus is categorical inputs and outputs. One common way to encode a categorical variable for a neural network is to have one node for each category. If, for example, a variable takes the values red, blue and green one can use three nodes to encode this. To encode that something is red the first node will take the value 1 and the other nodes will take the value 0. This works well for the input nodes where one can decide what one puts in, for the output nodes the values will still be continuous, but one can force them closer to zero or one by applying a sigmoid activation function to the nodes. To choose which output the network actually gives one can see which node has the largest value.

2.1.2 Training a neural network

We now know how a feed forward neural network can be used as a function, but to get it to be the function we want it has to be trained. To do this we need training data consisting of several input vectors $\mathbf{x}$, as well as a target output vector $\mathbf{y}$ for every input. Using the network one can get an output $\hat{\mathbf{y}}$ for every input. This is compared to the target output using an error function $E(\mathbf{y}, \hat{\mathbf{y}})$. If $\mathbf{y} = \hat{\mathbf{y}}$, this should give the value zero and it should also increase with how different they are, but exactly what this error function looks like depends on the problem. One common example error function is the squared euclidean norm $||\mathbf{y} - \hat{\mathbf{y}}||_\mathbf{2}^\mathbf{2}$, i.e. entry wise sum of squares $\sum_i (y_i - \hat{y}_i)^2$. We can then define a loss function that consists of the sum of the errors for each pair $\mathbf{y}$ and $\hat{\mathbf{y}}$ in the training data. When training the networks one views the training data as fixed and instead sees the loss function, L as a function of the weights and thresholds in the neural network. One then aims to minimise the loss function with respect to weights and threshold.

When training neural networks it is common to use variations of steepest descent. Steepest descent is one of the simplest optimisation methods, where one starts at some point and always moves in the direction which locally decreases the function the most. When using steepest descent one starts at a point x_0 and then moves to a new point according to $x_{t+1} = x_t - \eta \nabla f(x_t)$, where η is the step length. This is done until a stopping criterion is met. The stopping criterion can be based on when the change in function value, the norm of the gradient or the step size gets small enough. Note that for normal gradient descent step size and gradient norm is equivalent. To do this we need to calculate the gradient of the loss function. Remember that we are viewing it as a function of weights and thresholds, so its the partial derivatives $\frac{\partial L}{\partial w_{ij}^{(l)}}$ we're looking for. As L is just a sum of E for different values it is sufficient to find $\frac{\partial E}{\partial w_{ij}^{(l)}}$. Calculating this derivative is done by whats known as backpropagation and a thorough review of which can be found in for example [5].

Gradient descent is not a particularly sophisticated optimisation algorithm and thus carries several problems with it. One problem is that it gets stuck in local minima, ridges and on plateaus, i.e. wherever some partial derivative gets small. This problem is known as the vanishing gradient problem. Another problem specific to neural networks is that the partial derivatives with respect to weights in earlier layers tends to get small. One such method that solves the second problem and mitigates the first is resilient backpropagation [6], which still uses the partial derivatives to choose which direction to change the weights, but not how much. As a result all weights will update at a similar pace even if the gradient is smaller for the ones in the inner layers. As the step size is not proportional to the size of the derivative it will also does not get stuck as easily if the partial derivative is small. If $\Delta_{ij}^{(l)}(t)$ constitutes the step length for weight $_{ij}^{(l)}$ at time step t then an update looks like this:

$$w_{ij}^{(l)}(t+1) := w_{ij}^{(l)}(t) - \operatorname{sgn}\left(\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\right)\Delta_{ij}^{(l)}(t) \tag{2.3}$$

The step length will instead be modified so that it increases if the change keeps going in the same direction, i.e. when $\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1) > 0$, and decreases if it changes direction, i.e.$\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1) < 0$. The increase of step length will be done by multiplying with a factor $\eta^+ > 1$ and the decrease will be done by multiplying with a factor $\eta^- < 1$. The step length wont increase beyond a maximum of $\Delta_{\max}$ or below a minimum of $\Delta_{\min}$ Thus a update of the steplength looks like this:

$$\Delta_{ij}^{(l)}(t+1) := \begin{cases} \min\left(\eta^+\Delta_{ij}^{(l)}(t), \Delta_{\max}\right), & \text{if } \frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1) > 0 \\ \max\left(\eta^-\Delta_{ij}^{(l)}(t), \Delta_{\min}\right), & \text{if } \frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1) < 0 \\ \Delta_{ij}^{(l)}(t), & \text{if } \frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1) = 0 \end{cases} \tag{2.4}$$

One addition that is common is weight backtracking. In weight backtracking when the partial derivative of a weight changes sign one undoes the last step for that weight instead of taking a new step. After the step has been reverted the derivative will most likely have the same sign again and thus the step length will increase again. To avoid this one usually sets $\frac{\partial E}{\partial w_{ij}^{(l)}}(t)$ to 0. Thus in the next step $\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1)$ will be 0 and the step length wont update. The resulting algorithm looks as follows:

```
while not converged do
    Compute $\frac{\partial E}{\partial w_{ij}^{(l)}}(t)$
    if $\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1) > 0$ then
        $\Delta_{ij}^{(l)}(t+1) := \min\left(\eta^{+}\Delta_{ij}^{(l)}(t), \Delta_{\max}\right)$
        $w_{ij}^{(l)}(t+1) := w_{ij}^{(l)}(t) - \operatorname{sgn}\left(\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\right)\Delta_{ij}^{(l)}(t)$
    else if $\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1) < 0$ then
        $\Delta_{ij}^{(l)}(t+1) := \max\left(\eta^{-}\Delta_{ij}^{(l)}(t), \Delta_{\min}\right)$
        $w_{ij}^{(l)}(t+1) := w_{ij}^{(l)}(t) + \operatorname{sgn}\left(\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1)\right)\Delta_{ij}^{(l)}(t-1)$
    else if $\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\frac{\partial E}{\partial w_{ij}^{(l)}}(t-1) = 0$ then
        $w_{ij}^{(l)}(t+1) := w_{ij}^{(l)}(t) - \operatorname{sgn}\left(\frac{\partial E}{\partial w_{ij}^{(l)}}(t)\right)\Delta_{ij}^{(l)}(t)$
    end if
    t:=t+1
end while
```

2.2 Imputation

Imputation is the process of replacing missing data with substitute values. Data often consists of a set of different variables being measured from several different sources. A source can be for example an individual or geographical area from which the data is gathered. In this chapter it is assumed that the data is stored in a matrix where values from the same source are stored in rows, whereas the values of the same variable are stored in the same column. As a result a different values on the same column will come from similar distributions but not have very high dependency. Thus we should look at the columns to see in which realm the value should lie, i.e. which scale it's on for numerical variables or which categories are possible categorical variables. Other values on the same row on the other hand typically are more dependant on each other but can take completely different values. Thus an imputation method typically looks to the columns to see which values are possible but looks to the rows to see which of these possible values are more probable.

The simplest methods of imputation ignore the rows and simply look at the columns. These methods will be referred to as univariate imputation and will frequently be a step in more advanced imputation methods. These imputation methods can either be based on taking the average of the column, that is the mean or median for numerical values and the mode for categorical variables. Other methods are based around pulling a random value from a distribution based on the values in the column. This is typically done by just pulling another random value, but can also be done by fitting a distribution to the data and taking a random value from this distribution.

Another category of methods are those that fill in entries from similar observations.

These are called hot-deck imputations [7]. This can be done by just taking the entry of the observation that is closest in the other variables or one can pick out the k closest and take either a random one of them or take the mode of them [8] [9].

A bigger category of imputation methods are those that try to make a model that predicts a variable value using the other variables. These models are typically fitted to the data where the variable has been measured ad then applied to predict the missing data. This thesis will mainly focus on one such method which uses artificial neural networks for the model. It will also use another such method called multiple imputation by chained equations, or MICE, as a benchmark. MICE uses a simple imputation method like modal to fill the data, then iteratively uses regression to improve this estimate. It also does this multiple times to try to simulate a distribution of the missing data [10].

2.2.1 Benchmark imputation: MICE

MICE is an imputation method that uses three steps: initial imputation, iterative regression and multiple imputation. In this thesis the implementation used is the `mice` package in R [11].

During the initial imputation each unobserved variable is imputated using a univariate imputation method. Typical examples are using the mean or mode of that variable or drawing an observed value at random. The latter is the default of the `mice` package in R, although one may do the initial imputation oneself and provide mice with a full imputed data matrix as well as the matrix with only the observed values.

When the matrix is filled one can perform the regression. This is done by choosing one variable at the time to update. To update the variables one uses all the rows with an observed value in the relevant variable to fit a regression model. The regression model thus uses both the observed and the imputated data for predictors but only observed values for the dependent variable while fitting. What regression model to use depends on what kind of variables you have. In this thesis the data used is categorical with 2 categories and thus the default regression model of the mice package is logistic. This model is then used on each row without an observed value in the update variable, using all the other variables, both observed and imputated, in those rows to predict a new value imputated value. All variables are updated this way in a random order. Then this is repeated n times.Finally the entire imputation process is repeated m times leaving us with m suggested imputations for each missing value. Together these values can be viewed as an approximation of a predicted distribution for the missing value, thus giving us more information than just a predicted value.

Both m and n have been set to 15 giving MICE a similar run time to the neural networks.

2.2.2 Neural Network Imputation

Neural network based imputation uses neural networks as their model to predict one variable from the others. As a result one needs to train one network per variable, where the network uses one variable as output and the other variables as input. So for a variable a, the rows with an observed value in a will be used to train the network, where the other variables in the row are used as input and the values of variable a are used as the target output. The network is then used with the rows where variable a is missing as input to predict the missing values of a.

One problem is that rows are often missing more than one variable and one has to have a value for every input node both when training and using a network. One common solution is to train the networks only on the rows which do not have any missing variables, then using a univariate imputation method on the missing variables when they are treated as input variables [12]. This solution does not work in cases like ours when there are no rows without missing variables. Another possible solution available for categorical variables is to treat missing value as another category of the variable when used as input. So if a variable typically takes the values R and S one will have three input nodes for that variable, one for R one for S and one for NA if the data is missing. For the output nodes in the network that predicts this variable there will still only be two nodes, one for R and one for S. This allows us to train a network on every row where the corresponding variable has a recorded value, as every row now has an eligible input, but only the rows with a recorded value have a eligible target output. It also lets us use the network on any row without having to employ other imputation techniques. In Figure 2.2 one can see an example network for predicting ceftiofur in data that contains five variables, chloroamphenicol, daptomycin, teicoplanin, streptomycin and ceftiofur. Each variable can take the values R and S. There are three input nodes for each variable other than ceftiofur, one which will be one if the variable has the value R and zero otherwise, and likewise one for S and one for missing value. This network has been trained on every row where ceftiofur has been measured, and can then be used to predict the value of ceftiofur where it hasn't.

2.3 Implementation

The neural networks used on this project have been implemented using R and the neuralnet package. The implementation method was chosen somewhat arbitrarily based on both previous experience and ease of use. Comparison of different implementation was deamed outside the scope of this thesis.

The main function in the neuralnet package is called neuralnet. This is the function that performs the training and it takes the following arguments with the following defaults:

```
    neuralnet(formula, data, hidden = 1, threshold = 0.01,
stepmax = 1e+05, rep = 1, startweights = NULL,
learningrate.limit = NULL, learningrate.factor = list(minus = 0.5,
plus = 1.2), learningrate = NULL, lifesign = "none",
lifesign.step = 1000, algorithm = "rprop+", err.fct = "sse",
act.fct = "logistic", linear.output = TRUE, exclude = NULL,
constant.weights = NULL, likelihood = FALSE)
```

The `data` variable contains the data in a data frame. In this case the data frame will have one row per bacteria. The named columns will correspond to the antibiotics. There will be one variable per antibiotic that can take the values R for resitant, S for susceptible and NA for missing data. So this is a categorical variable with three different values. As the package will not accept categorical variables they have to be encoded differently. One common method for encoding categorical variable is to have one dummy variable per category, which takes the value when the value belongs to the category and zero otherwise. So instead of having one variable named `AntibioticA` with a value of R we will have the variables `AntibiotcA_R` with a value of 1 and `AntibioticA_S` and `Antibiotic_NA` with a value of 0. So the data frame has three columns per antibiotic.

The `formula` argument specifies which columns will be used as input and which will be used as output. This is specified as an R formula. When training the network that is meant to predict the resistance towards an antibiotic named antibioticA, the output will be `AntibioticA_S` and `Antibiotic_R`, while the input should be all other columns except for `Antibiotic_NA`. This is done with the formula `· - AntibioticA_NA ~ AntibioticA_S+AntibioticA_R`.

The `hidden` argument gives the number of hidden layers as well as the number of nodes in the hidden layers. It is given as a vector where each entry corresponds to the number of nodes in a layer. The first entry specifies the number of nodes in the hidden layer closest to the input neurons. See section 3.2.1 for which values were used.

The `threshold` argument sets the stopping criteria for the training. The neuralnet algorithm stops when the norm of the gradient goes below this given threshold. Note that this derivative will scale with the amount of indata and thus has to be tuned. See section 3.2.2 for analysis of this parameter.

The `stepmax` argument sets a maximum number of steps for updating the weights before aborting the training. It was never changed from the default `1e+05` and was never reached with reasonable values for `threshold`.

The `rep` argument sets how many neural networks are to be trained. The default 1 was used throughout.

The `startweights` argument lets you set the startweights, the default randomises them and was used throughout the project.

The arguments `learningrate.limit` and `learningrate.factor` sets Δ_{max}, Δ_{min}, η^+ and η^-. The defaults of $\Delta_{\text{max}} = \infty$, $\Delta_{\text{min}} = -\infty$, $\eta^+ = 1.2$ and $\eta^- = 0.5$ were used throughout.

`learningrate` is only used for normal backpropagation.

`lifesign` and `lifesign.step` specifies what will be printed during training and how often it will be printed. they are is immaterial to the result.

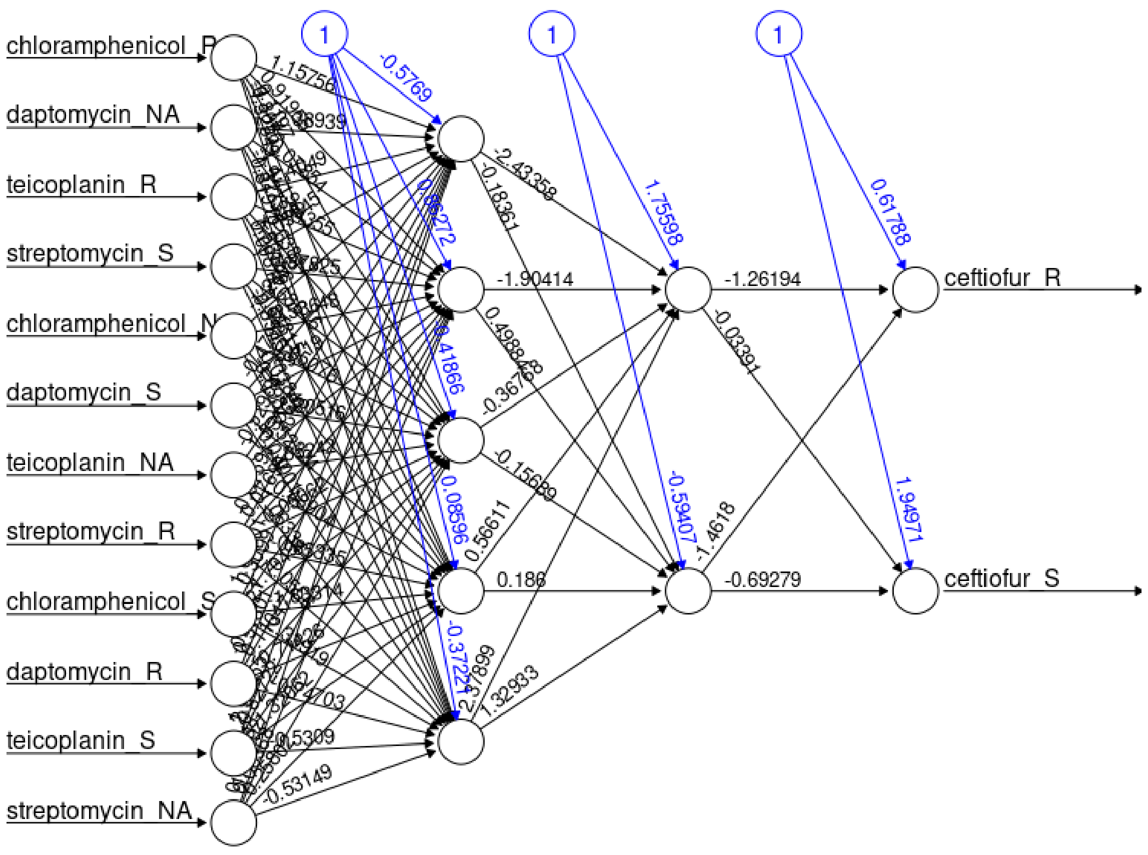


Figure 2.2: A simplified network for predicting resistance against ceftiofur. For ease of view this network only uses four antibiotics as input and two hidden layers with four and two nodes respectively. The plot was made by the plot function in the neuralnet-package.

`algorithm` chooses the algorithm used. Only the default resilient backpropagation with weight backtracking was used.

`err.fct` and `act.fct` sets the error and activation function. Only the default sum of square errors and logistic activation were used.

The `linear.output` argument changes whether the activation function is used on the output neurons. It is not used if set to the default `TRUE`. As output values of 1 and 0 are desired, this is set to `FALSE` throughout.

`exclude` and `constant.weights` can set weights to 0 or other constant values respectively. For this to be meaningful some things about the underlying structures must be known, which they aren't so these are left `NULL`.

Finally `likelihood` was kept to its default `FALSE`. If set to true one can obtain a confidence interval on the weights.

3
Result

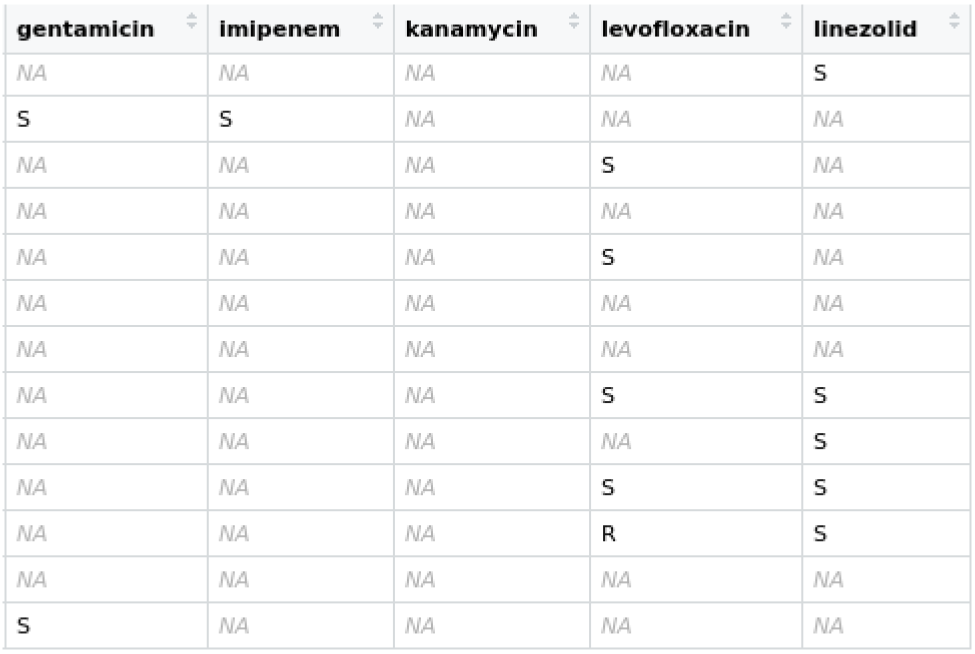
In this project neural networks are trained to predict antibiotic resistance phenotype according to the method described in section 2.2.2. To train the networks as well as test their results it is necessary to have data. In this project the data comes from The European Surveillance System – TESSy, provided by all EU countries as well as Norway and Iceland and is released by the European Centre for Disease Control, also known as ECDC. Note that the views and opinions of the authors expressed herein do not necessarily state or reflect those of ECDC. The accuracy of the authors' statistical analysis and the findings they report are not the responsibility of ECDC. ECDC is not responsible for conclusions or opinions drawn from the data provided. ECDC is not responsible for the correctness of the data and for data management, data merging and data collation after provision of the data. ECDC shall not be held liable for improper or incorrect use of the data.

In this chapter Section 3.1 will go over the data, while section 3.2 contains information about some ways to improve the performance of the neural networks, while section 3.3 compares the result with that of other imputation methods.

3.1 Exploring the data

The data provided by ECDC has been collected by the European Antimicrobial Resistance Surveillance Network (EARS-Net). EARS-Net is the EUs surveillance system of antibiotic resistance in bacteria that cause infections. They collect data from all EU countries as well as Iceland and Norway. The provided data from 2017 consists of data on bacteria from 330 000 different patients. Each strain was tested for resistance against some antibiotics. On average around seven and half antibiotics were tested on a strain resulting in 2,5 million measurements. With every measurement there is also a lot of other information. Some of it is used solely for identifying which observations came from the same bacterial strand. The following variables are the only ones that will be used in thesis for something more than identifying the measurements coming from the same patients:

- **Antibiotic:** which antibiotic was tested.
- **Country:** where was the patient tested.
- **Pathogen:** which species of bacteria it was.
- **SIR-value:** The result of the test. Takes the values R for resistant, S for



gentamicin	imipenem	kanamycin	levofloxacin	linezolid
NA	NA	NA	NA	S
S	S	NA	NA	NA
NA	NA	NA	S	NA
NA	NA	NA	NA	NA
NA	NA	NA	S	NA
NA	NA	NA	NA	NA
NA	NA	NA	NA	NA
NA	NA	NA	S	S
NA	NA	NA	NA	S
NA	NA	NA	S	S
NA	NA	NA	R	S
NA	NA	NA	NA	NA
S	NA	NA	NA	NA

Figure 3.1: Small excerpt from the data converted to matrix form.

susceptible and I for intermediate. For simplicity S and I will be treated as one category.

The data is then transformed into matrix form, with one row for each patient and one column for country, one for species and one column per antibiotic. In Figure 3.1 we can see a small exert from this data table, with some observed values of S and R and a lot of missing values. This is the matrix for which we will be performing imputation.

To get the numerical values necessary for the neural networks each column representing an antibiotic is turned into three columns. For a antibiotic named antibioticA, these would be named the following: `antibioticA_R`, `antibioticA_S` and `antibioticA_NA`. If a bacterial strain was measured resistant against antibioticA its row will have a one in column `antibioticA_R` and zeroes in `antibioticA_S` and `antibioticA_NA`, if it was susceptible or intermediate `antibioticA_S` will have the one instead and if antibioticA wasn't tested on a strain `antibioticA_NA` will have a one in that row. The column representing country is also turned into thirty columns, one of each country in the data, and the columns representing species of bacteria is similarly turned into 8 columns.

In Figure 3.2 you can see how many observations were made for each antibiotic as well as how many of the observations showed resistance. For most antibiotics a vast majority of the observations showed susceptibility, with some exceptions like amoxicillin, amoxicillin with clavulanic acid and ampicillin. In Figures 3.3 & 3.4 we can see the observations broken down by reporting country and species of bacteria respectively. A more detailed break down of the observations can be found in Appendix A.1. One notable take away from the figures in the appendix are that most species have different antibiotics tested on them with two exceptions. The

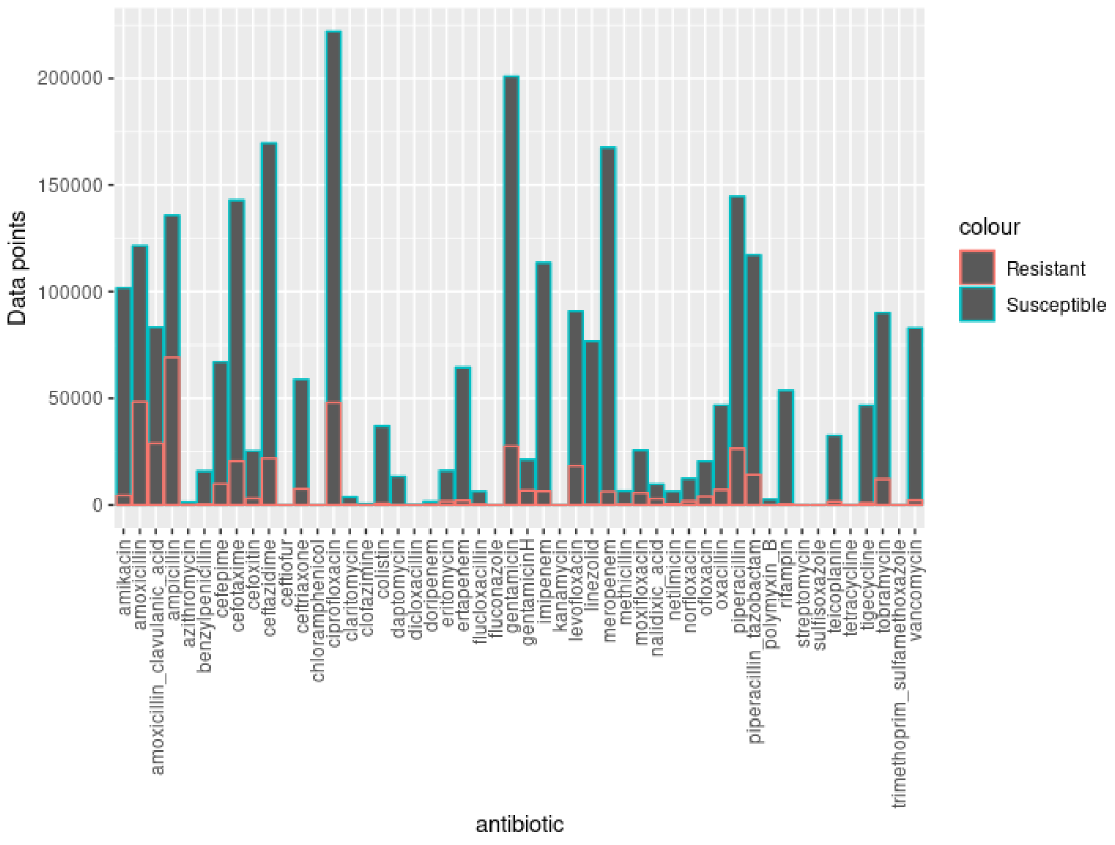


Figure 3.2: The number of observations for each antibiotic. The red part shows how many of them were the resistant.

bacterial species *Enterococcus faecium* and *Enterococcus faecalis* have the same antibiotics tested on them but with wildly different resistance rates. *Escherichia coli* and *Klebsiella pneumonia* also have similar antibiotics being tested. We can also note that different countries vary vastly in which and how many different antibiotics were measured.

3.2 Tuning the network

When setting up the neural networks used in the imputation there are some hyperparameters to be selected. In this section we will look at the number of layers and the number of nodes in them, as well as the stopping criterion and adding additional indata parameters. To be able to compare the results we use cross validation error. When one does cross validation one takes some rows of the data and set them apart as a test set, while the rest is used as training data. Then we see how well they can predict the data in the test set. This is done by letting the appropriate networks predict each value from the test set using the other observation in the same row and comparing it to the value in the test set. This error rate thus shows how well it would perform on antibiotics typically tested together if one less measurement was taken. The error rate does not show how well the network would perform on

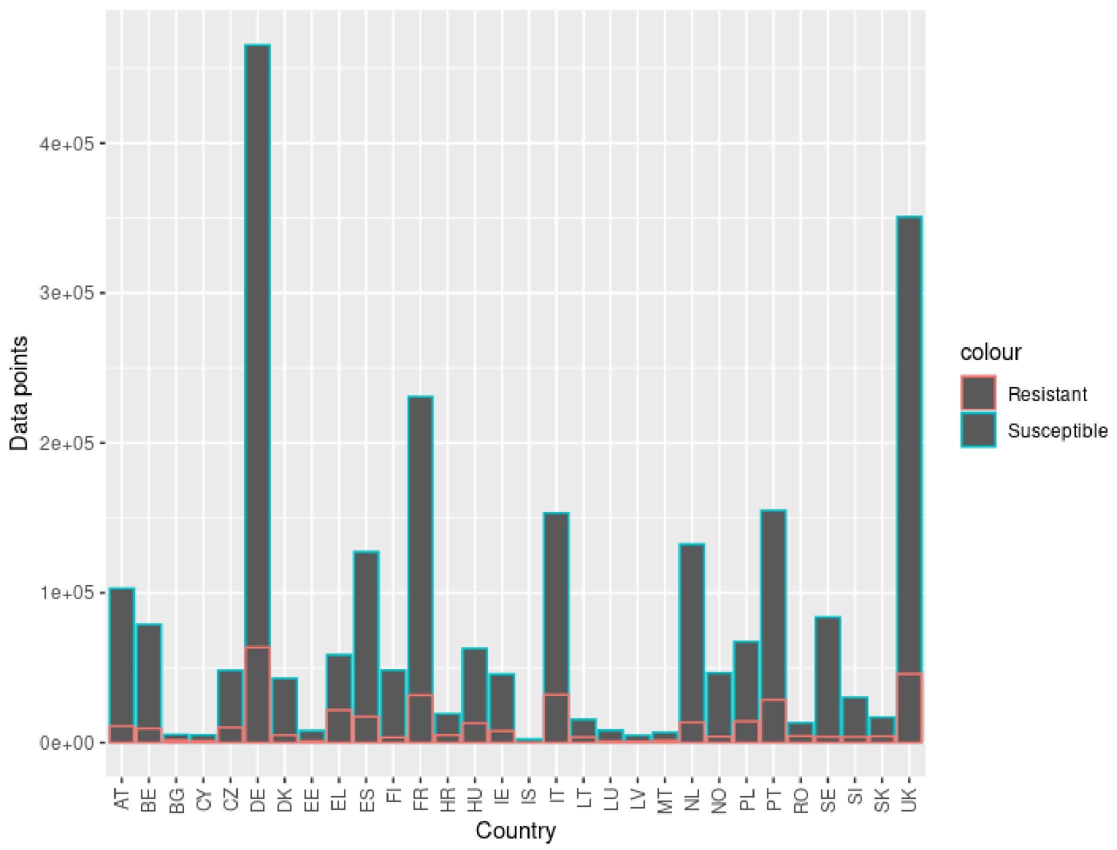


Figure 3.3: The number of observations for each country. The red part shows how many of them were the resistant.

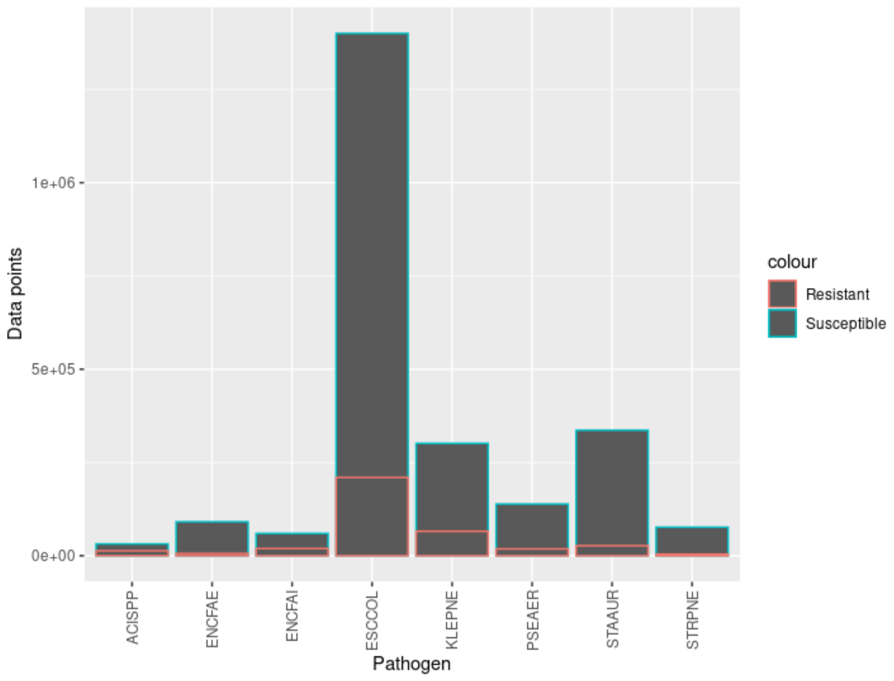


Figure 3.4: The number of observations for each bacterial species. The red part shows how many of them were the resistant.

a random antibiotic. This might seem like a weakness of the error rate, but this is actually similar to the conditions under which there might be an interest of using this kind of tool, i.e. to see which antibiotic one would test next.

3.2.1 Number of layers and nodes

First we have to decide how many layers our neural net is going to have. There is no intrinsic correlation between our variables that we know beforehand so we only need a function estimator. Two hidden layers is sufficient to to get any continuous function [13]. Despite this one might reduce the number of necessary neurons by increasing the number of layers. Thus one can increase the number of layers if the necessary number of neurons seems high.

Then there is the question of choosing the number of nodes in each layer. Typically one keeps the number of nodes in a hidden layer between the number of input and output nodes [13]. To test for the optimal number of nodes we tested using every combination of n nodes in layer one an m nodes in layer two, where m and n take one of the values 10, 25, 50 and 150 and where $m < n$. The networks were then trained on 10000 rows of test data each. The networks performance can be seen for some configurations of nodes Figure 3.5. As can be seen the performance is improved greatly by having less than 50 nodes in the hidden layer closest to the output layer. Another noticeable trend is a slight improvement with larger number of nodes in the first hidden layer, as well as a similar increase in a first layer with many nodes. In Figure 3.6 the number of nodes in the first hidden layer has been fixed to 150. A clear decrease in performance can be seen when the number of nodes increase above 35, but before that the difference seems to be random. 20 was thus arbitrarily chosen as the number of nodes in the second hidden layer. When number of nodes in the second hidden layer was fixed to 20 and the number of nodes in the first hidden layer we see no discernible nonrandom effect from the change in node number and it was thus arbitrarily left at 150. This can be seen in Figure A.2 in Appendix A.

3.2.2 Stopping criterion

When using the default stopping criterion the run time increases more than polynomially with the size of the training data, see Figure 3.7.

This has to do with the stopping criterion used in neuralnet. It stops when the norm of the gradient of the loss function gets low enough. Recall that the loss function was the sum of the errors over the entire training set. As a result both the loss function and its gradient will scale with the amount of training data. To find an appropriate value we let the threshold vary for a fixed sample of 1000 rows in the training data. The result can be seen in Figure 3.8. There is a notable improvement in performance when going down to 0.1, but then the improvement stops. When taking this value and scaling it linearly with the sample size we get the result in Figure 3.9. The run time still increases more than linearly, but in a more manageable fashion, while still

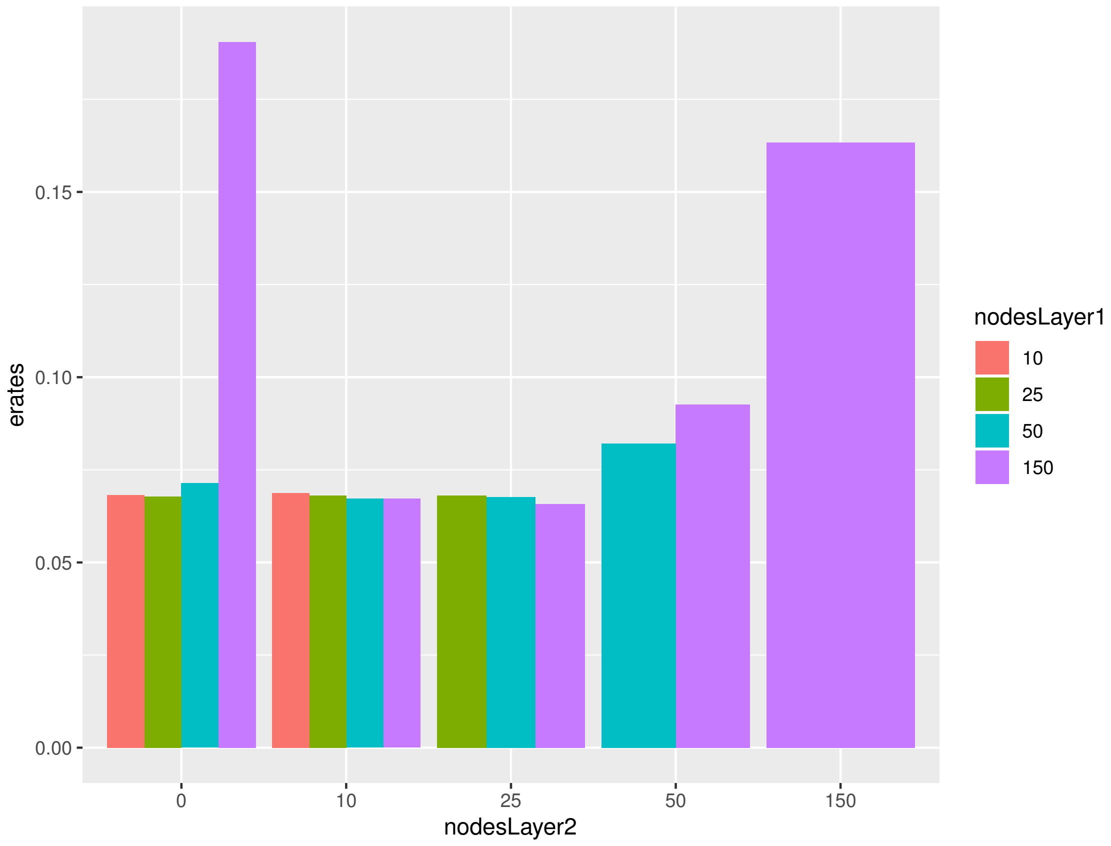


Figure 3.5: This figure shows performance of the neural network when the number of nodes were varied. Both the training and the test set consisted of 10000 patients selected at random from the data. The x-axis shows the number of nodes in the hidden layer closest to the output, with 0 nodes denoting we only have one layer. The colours show the number of nodes int the hidden layers closest to the input layer.

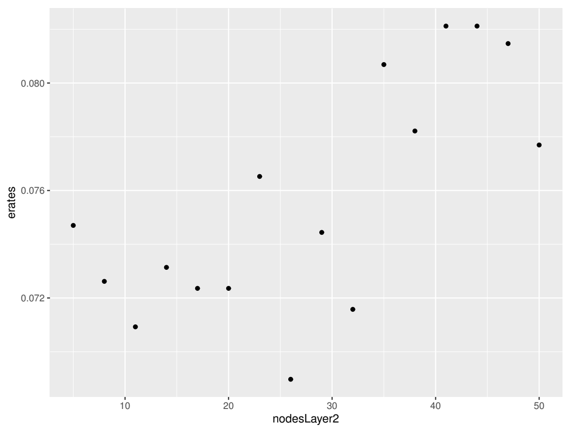


Figure 3.6: Error rate for varying number of nodes in the second hidden layer while the number of nodes in the first hidden layer was fixed to 150.

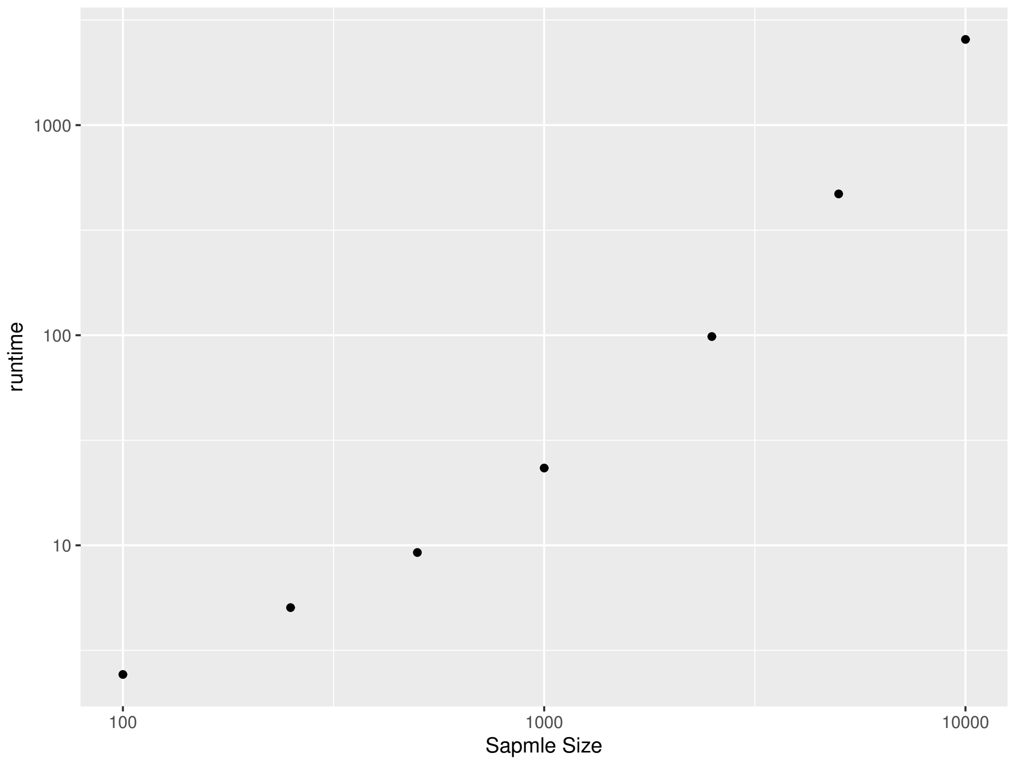


Figure 3.7: This figure contains a loglog plot of run time versus sample size with constant threshold. As it's nonlinear in the loglog plot, the proper relationship is most likely more than polynomial.

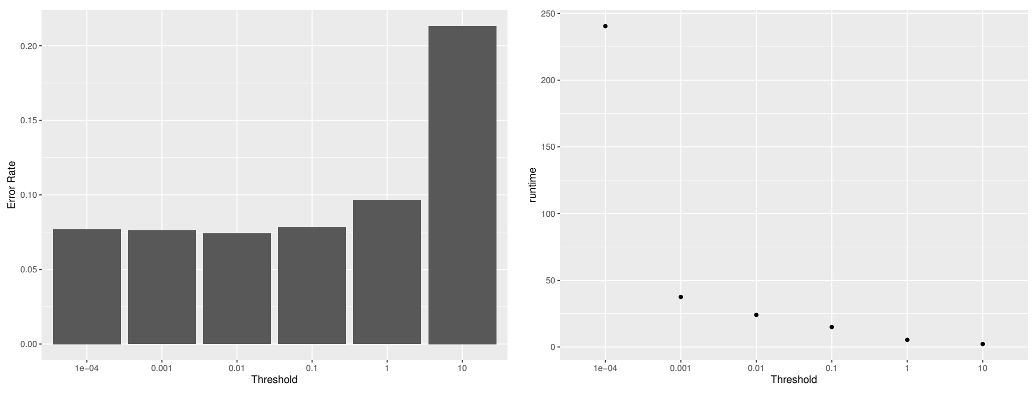


Figure 3.8: Error rate (left) and run time (right) with different thresholds in the stopping criterion and a sample size fixed to 1000. Note that the performance only improves until 0.1

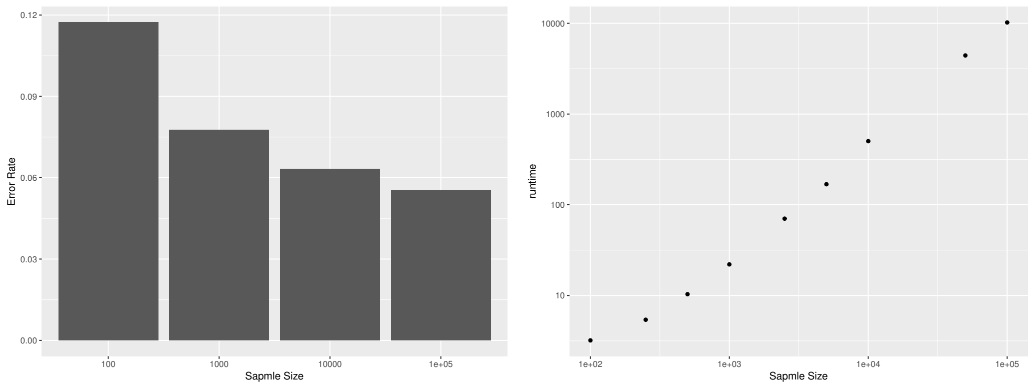


Figure 3.9: Error rate and run time with different sample sizes when scaling the threshold linearly. Note the log scale on both sample size and run time. As the slope is close to one complexity is close to linear.

retaining major performance improvements with increasing amounts of data.

3.2.3 Additional input variables

The data also includes a considerate amount of metadata. Some of this data might improve the performance of the network. The two most suitable contenders are country and species of bacteria. These variables are also categorical and will thus require an additional node in the input layer per category. In Figure 3.10 one can see the performance of four different networks trained on all the 281236 observations of the training data. One network has the reporting country as indata in addition to the resistance values, one has species of the bacteria, one has both and one has neither. Each addition of indata shows a small but noticeable increase in performance. It is also worth noting that the addition of country has a smaller improvement and the improvement of country is not found when training on a smaller data set.

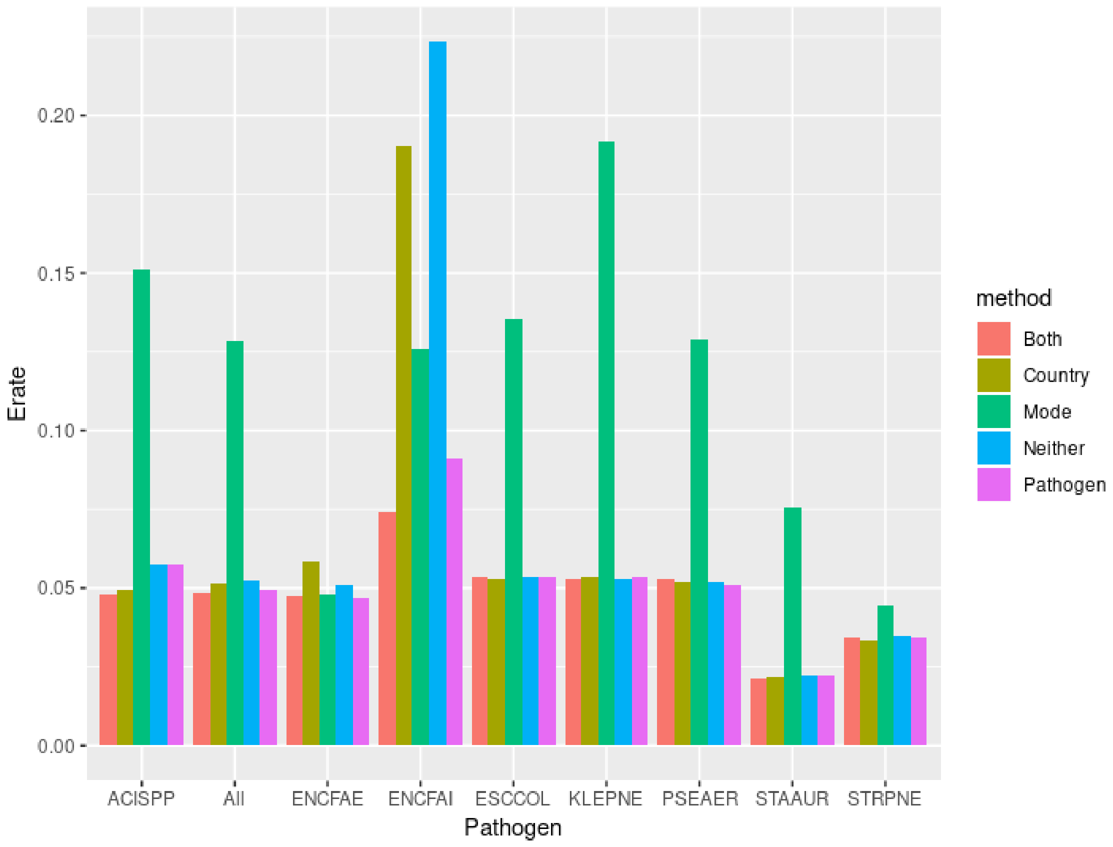


Figure 3.10: The result of neural networks trained with various amounts of extra indata applied to data from different bacteria. The bars label ed "All" shows the result on the entire test data. There is a small but notable increase to the addition of either, with most of the increase from adding Pathogen as indata coming from ENCFAI (*Enterococcus faecium*) whereas most of the improvement from using country came from ACISPP (*Acinetobacter* spp.).

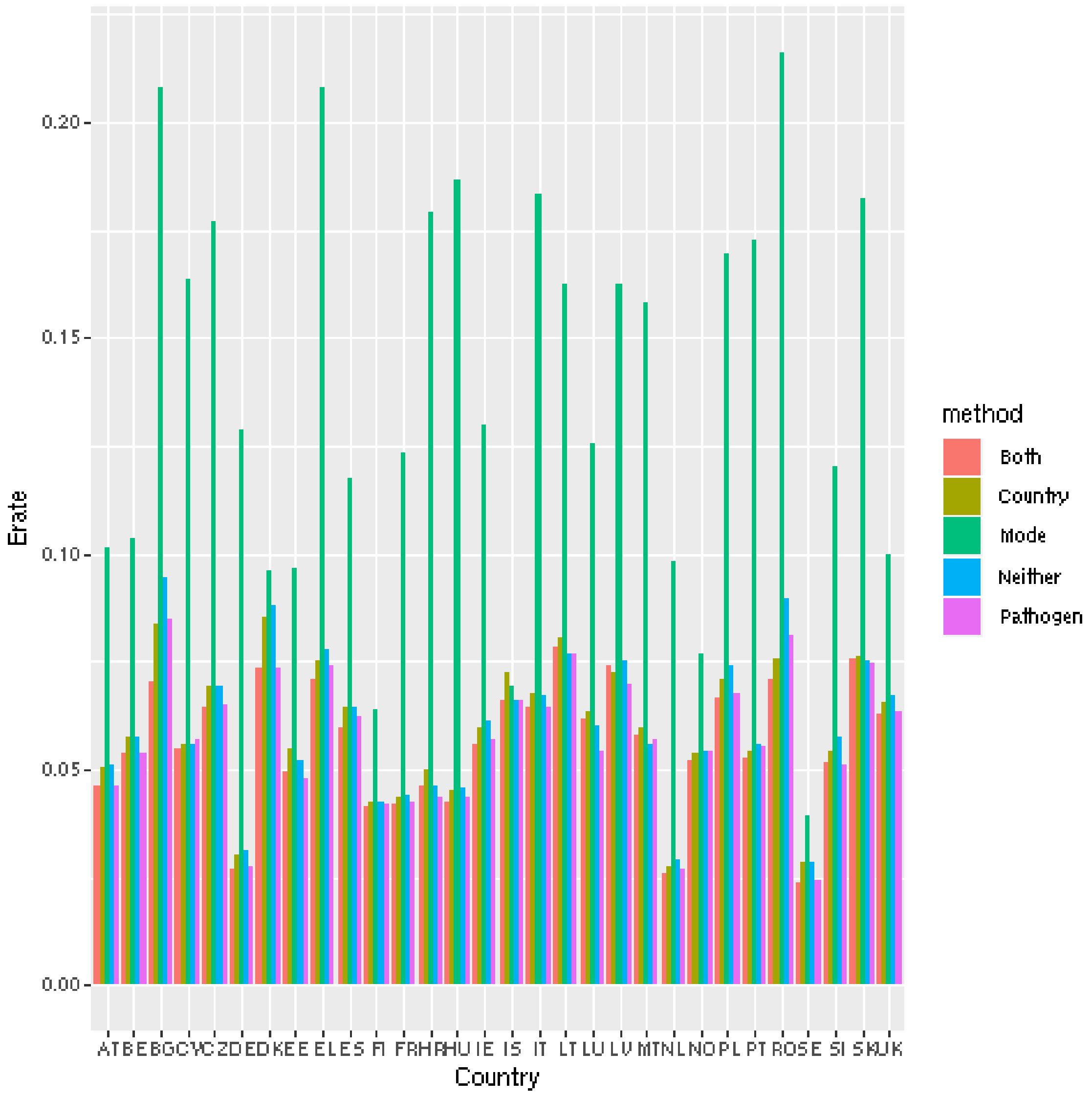


Figure 3.11: The error rates of neural networks trained with various amounts of extra indata applied to data from different countries.

3.3 Reviewing the network

After tuning the network and adding country and species as indata we get the performance seen in Figure 3.12. Here we can also see a comparison to a modal imputator and MICE with 15 regression steps and repetitions. There is also a comparison of their major and very major errors. A major error is when a susceptible bacteria is diagnosed as resistant, whereas a very major error is when a resistant bacteria is diagnosed as susceptible. As Figure 3.12 shows the very major errors are more common than the major errors. This can be ascribed to the fact that most bacteria are not resistant to most things. In Figure 3.13 we can see that for amoxicillin and ampicillin where the rates were approximately even the major and very major errors were also approximately equal.

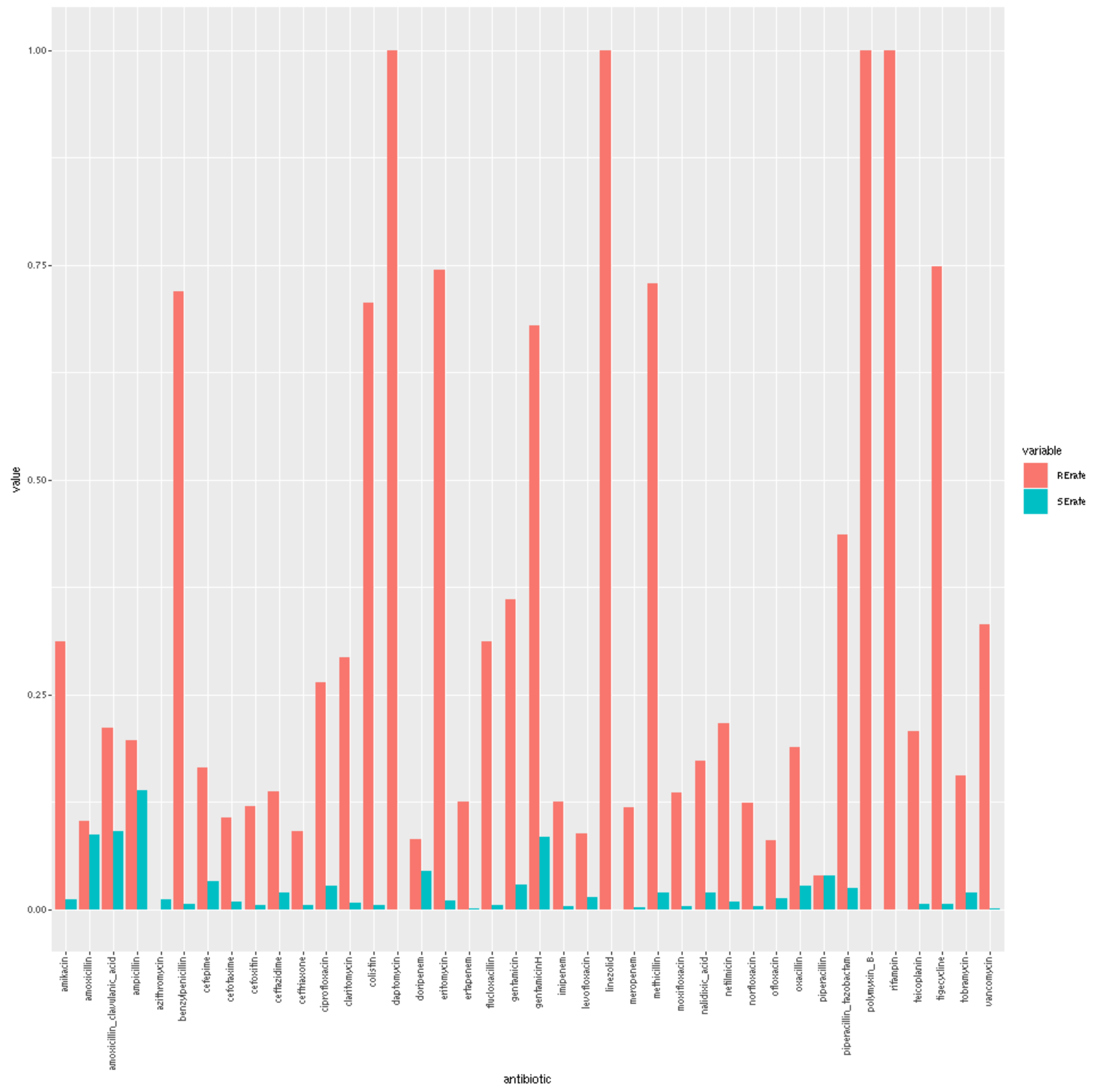


Figure 3.13: This figure shows the major and very major error rate of the final network for different antibiotics. The red shows the rate of very major errors, while the blue shows the error rate of major errors.

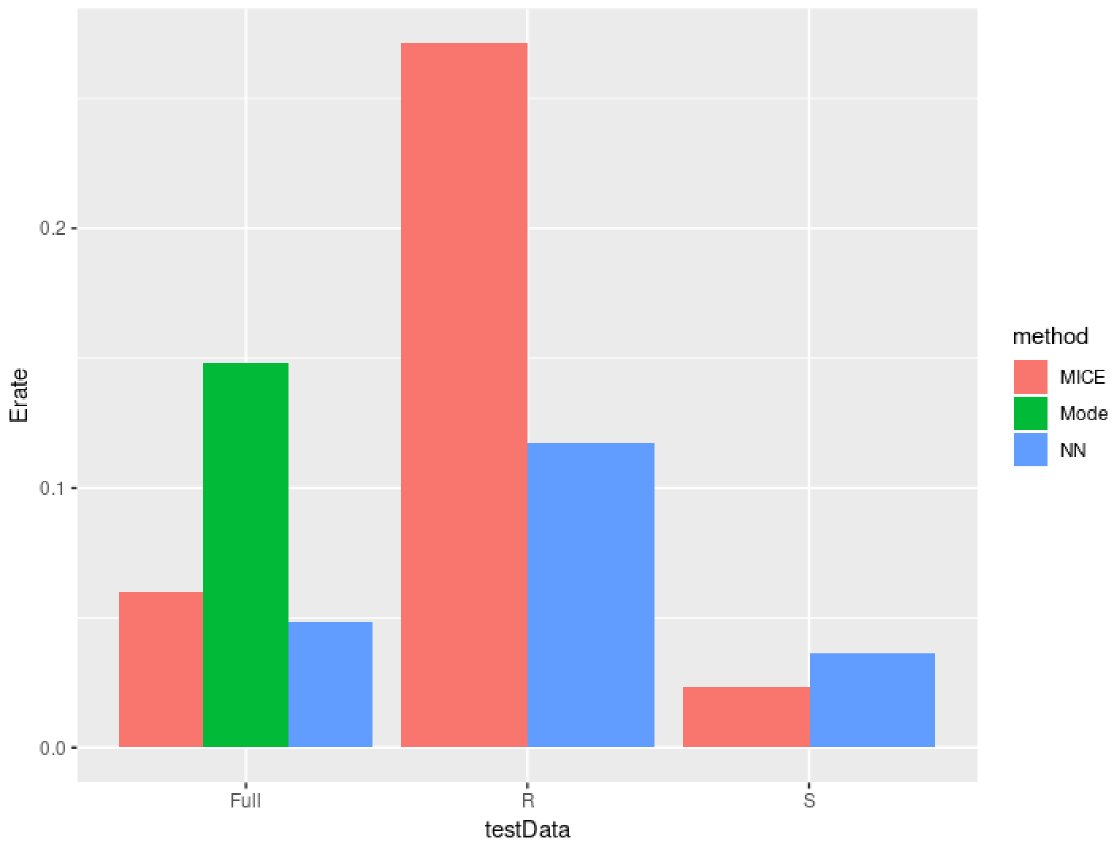


Figure 3.12: This figure shows the error rate on the set for three different methods: modal imputation (green), MICE (red) and neural network imputation (red). The neural network has been trained on the entire training data and uses both country and species as indata. The left bars shows the errors on the full test set. The middle part shows the very major error rate, i.e. how often the prediction was correct when the true value was R. The right most bars show the major error rate i.e. the error rate on the measurements where the true value was S. The NN performed notably better. We can also see that the neural net has a more even distribution of major and very major errors than MICE, which experience similar problems with very major errors being more common.

4
Discussion

The goal of this project was to predict antibiotic resistance in bacteria using previous measurements of other antibiotic resistance. As we can see in figure 3.12 we reached an average error rate of 5% and a very major error rate of 10%. For reference a clinical method of diagnosing resistance needs a major error rate below 4% and a very major error rate below 1% [14], thus more work is needed before clinicla tests can be replaced by predictions from this neural network, but it is still a very good supplement. There is still a threefold improvement compared to using the modal imputation, which is the best guess that does not take previous measurements into account. One problem with the network right now is it's tendency to make very major errors. There are numerous ways thhis can be improved. One way is to tune the loss function to punish very major errors more than major errors. In the current structure this would require different error functions on the two out nodes, which the `neuralnet` package does not allow. But it is possible to do in many other implementation. It is also possible to change the loss function by switching the output to a single node, where 1 represents resistance and 0 susceptibility and using an asymmetric loss function. Another possible solution is to actually include the intermediate category. This could potentially move some of the errors to minor, i.e. where the network incorrectly predicts intermediate resistance or predicts R or S when it is actually intermediate. One could also both add intermediacy as an outdata option and change the error function to make a network that is leaning towards intermediacy. The goal of this would be to have the networks make fewer actual predictions but make the predictions usable. This way one might be able to approach clinically low levels of major and very major at the expense of a large number of minor errors. This seems more useful as it might actually give us conditions under which one would be able to skip some time consuming clinical measurements. Another method one could use is inspired by MICE where one makes multiple imputations. In this case this would mean training multiple networks per antibiotic and have each of them make a prediction and then only give predictions if they are largely in agreement. The exact number of networks and the portion of required predictions of course have to be tuned to achieve a desirable result.

It is also interesting to look at where network performs well and where it performs poorly. One thing that can help is to look at the country wise error rates in figure 3.11. Instead of looking at the error rate of the network one can look at how much better it performs than the modal imputator. In figure 4.1 we see how lower the error rate of the networks is in terms of modal error rate, i.e. how well we are actually predicting the resistance rather than it being easy to guess. We can also see in the same figure that this is low for the countries that have few tested antibiotics per patient. This is no surprise as the neural network is gonna have less data to work with for the prediction. One of thing that would be interesting to investigate further is how the number of measurements affects the prediction rate.

It is also nice to note that the neural network method performed better than MICE, a method of similar sophistication and computation time. One advantage of neural networks is that they naturally include interactions between multiple different variables, whereas regression models will need very many terms to include interactions between more than two variables. It is also advantageous to be able to be able to do regression without having any idea what kind of dependencies there might be. The disadvantage is that it is harder to look at the neural network to obtain information about why it's predicting as it does, where as the coefficients in a typical regression model gives more information. Another advantage of the neural network based method is that it, once trained, is easier to use on a new set of variables. It is possible to adapt MICE to save all regression models and run them only on a new measurement, but it is not supported in the package originally unlike the neural network based method. In the actual result MICE gave a very high very major error rate, as discussed above one can remedy this by not having it predict according to a majority of the imputations, but require a higher portion to predict susceptibility before predictting susceptibility. This change would decrease the very major error rate, but is likely to increase both the major error rate and the overall error rate.

The test deciding the number of nodes in each layer was quite rudimentary, and just insured that we weren't missing out on big performance improvements by choosing different values rather than actually choosing optimal values. One often used principle is to keep the node numbers as low as possible without hurting the cross validation error. This is done for two reasons, the first is to avoid overfitting. Overfitting is when you train the network too much and rather than learn the underlying correlations it just remembers the data from the training set exactly. When the data is noisy this can lead to poor performance [15]. Because of the discrete nature of our data coupled with our abundant an rather unnoisy data our problem is not at that big a risk from overfitting as can be seen in figure 3.8 the performance does not become worse with excessive training, but rather stays constant. The second reason to keep the node numbers down is to simply reduce the run time, which is linearly increasing with the number of redundant nodes. With too few nodes it will also take longer as the network finds it harder to converge. A more sophisticated method of choosing the number of nodes is to start with an abundance of nodes than sequentially remove nodes that don't contribute to lowering the error rate. This process is known as pruning.

The `neuralnet` package was certainly adequate for this problem, but it also had some limitations. First the activation and error function has to be the same on all nodes, a package such as tensorflow/keras lets you set these separately per node. Secondly the stopping criterion used is unconventional and requires tuning. A more typical stopping criterion looks at the cross validation error and trains until it ceases to go down to decrease the risk of overfitting. Thirdly the algorithm used, while

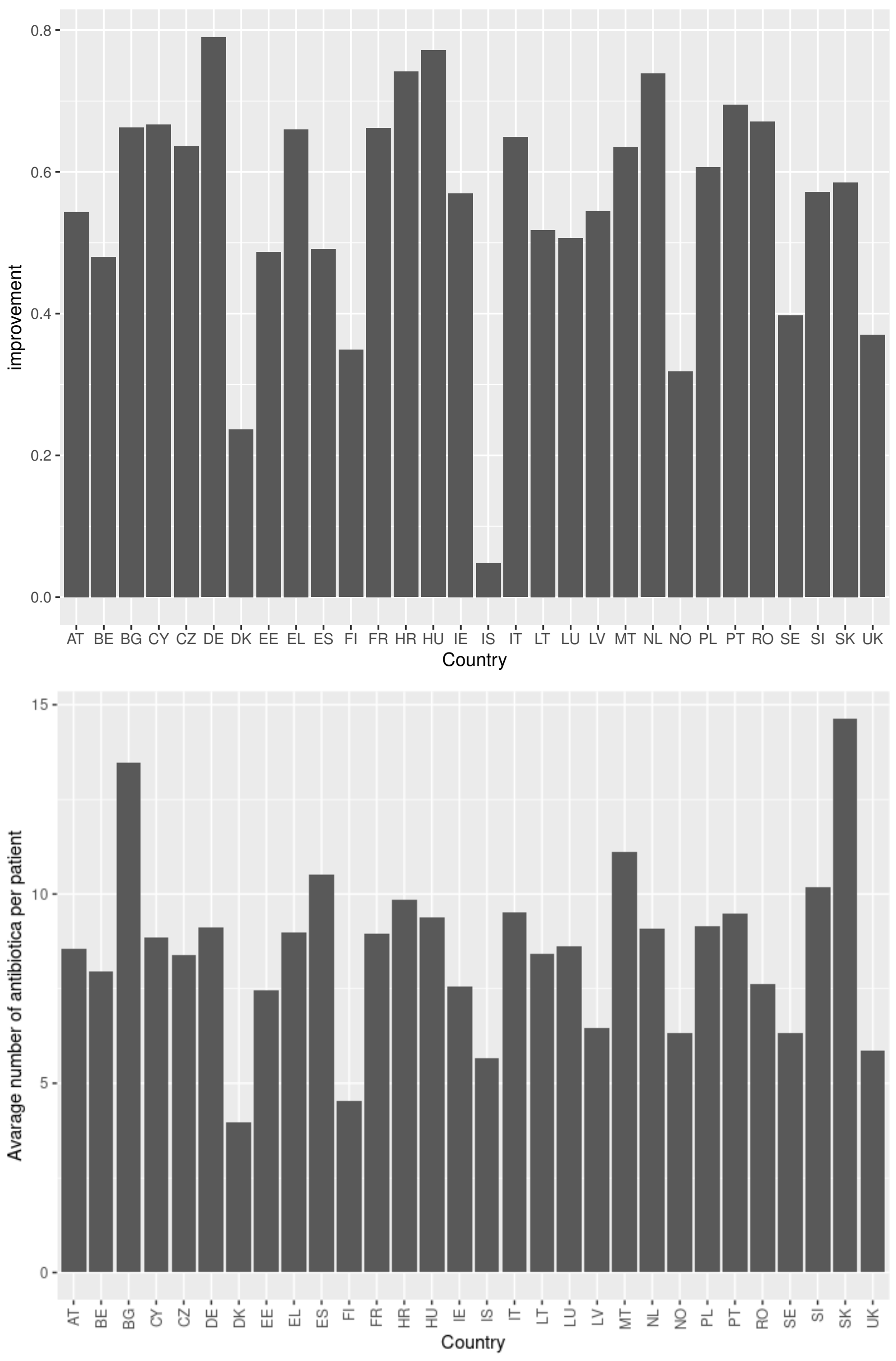


Figure 4.1: In the left graph we see how much the neural network improved on the modal imputators performance on data separated by country. In the right graph we see the average number of measurements per patient. Note how both graphs share the nordic countries and UK as low points.

adequate, is not up to industry standard. For these reason the choice of implementation would likely had it been done later in the process. With that said the package worked pretty well and its ease of use and not having to learn about multiple packages makes the decision very reasonable.

In conclusion neural network based imputation can predict resistance significantly better than the guesswork of modal imputation, but is not yet in a state where it can reduce the number of necessary measurements and maintain a low very major error rate. But with further research into when the network performs well, and with improvements such as non-binary output we might reach a point where we can reduce the number of necessary measurements.

Bibliography

[1] M.J. Friedrich. WHO's Top Health Threats for 2019. *JAMA*, 321(11):1041–1041, 03 2019.

[2] The European Committee on Antimicrobial Susceptibility Testing. Antimicrobial susceptibility testing, eucast disk diffusion method, version 7.0. 2019.

[3] Paul A. Rutecki. Neuronal excitability: voltage-dependent currents and synaptic transmission. *Journal of clinical neurophysiology : official publication of the American Electroencephalographic Society*, 9 2:195–211, 1992.

[4] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[5] Brilliant.org. Backpropagation. `https://brilliant.org/wiki/backpropagation/`. [Online; accessed 2020-01-14].

[6] Martin Riedmiller and I. Rprop. Rprop - description and implementation details. 04 2004.

[7] Roger A. Herriot. Collecting income data on sample surveys: Evidence from split-panel studies. *Journal of Marketing Research (JMR)*, 14(3):322 – 329, 1977.

[8] P. Jonsson and C. Wohlin. An evaluation of k-nearest neighbour imputation using likert data. In *10th International Symposium on Software Metrics, 2004. Proceedings.*, pages 108–118, Sep. 2004.

[9] Wieger Coutinho, Ton de Waal, and Natalie Shlomo. Calibrated hot-deck donor imputation subject to edit restrictions. *Journal of Official Statistics*, 29(2):299 – 321, 2013.

[10] Melissa J Azur, Elizabeth A Stuart, Constantine Frangakis, and Philip J Leaf. Multiple imputation by chained equations: what is it and how does it work?. *International Journal Of Methods In Psychiatric Research*, 20(1):40 – 49, 2011.

[11] Stef van Buuren and Karin Groothuis-Oudshoorn. mice: Multivariate imputation by chained equations in r. *Journal of Statistical Software, Articles*, 45(3):1–67, 2011.

[12] Kancherla Jonah Nishanth and Vadlamani Ravi. Probabilistic neural network

based categorical data imputation. *Neurocomputing*, 218:17 – 25, 2016.

[13] J. Heaton. *Introduction to Neural Networks with Java.* Heaton Research, 2008.

[14] James A. Poupard, Lori R. Walsh, and Bruce Kleger. Antimicrobial susceptibility testing : critical issues for the 90s. 1994.

[15] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

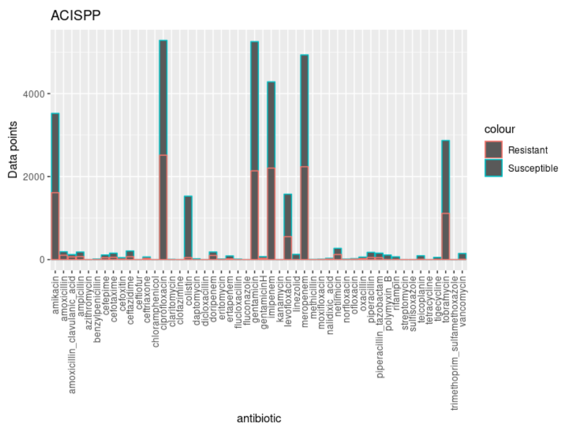
A

Appendix A: Additional Figures

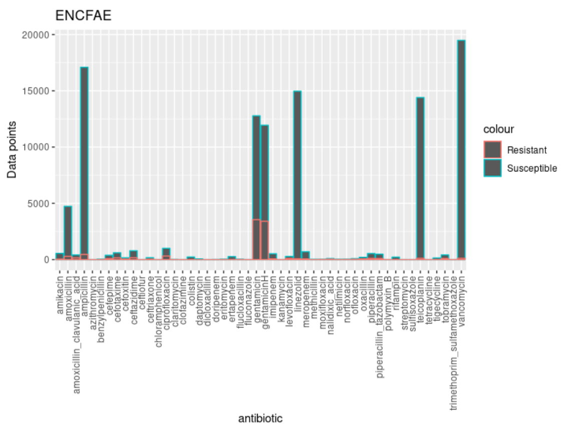
In this appendix one can find some additional figures that did not make it into the main report.

A.1 Additional figures about the data

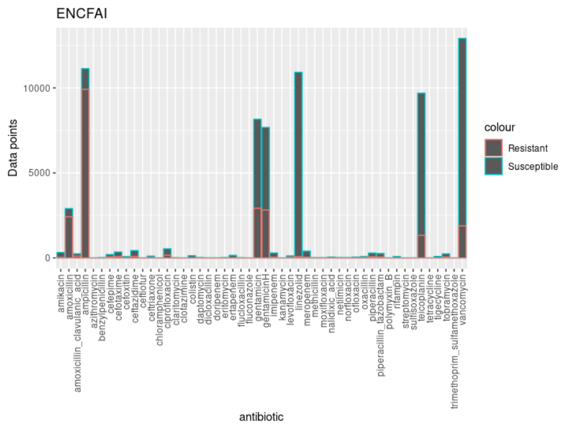
In figure A.1 one can see the usage of different antibiotics broken down by species of the bacteria.



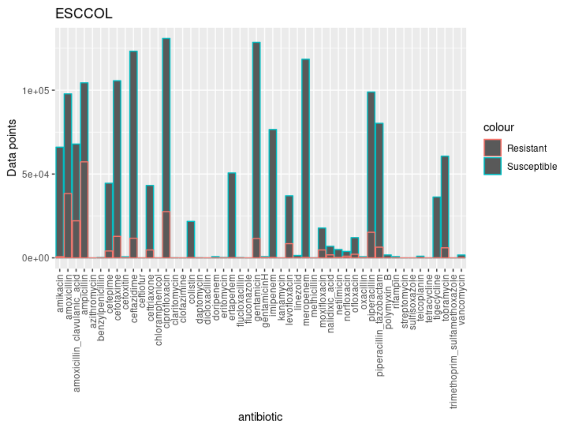
(a) Acinetobacter



(b) Enterococcus faecalis

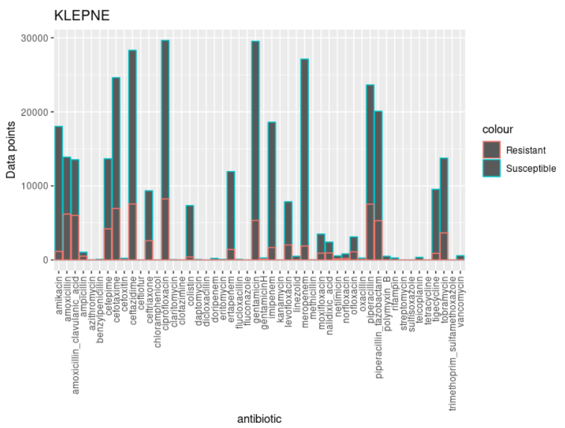


(c) Enterococcus faecium

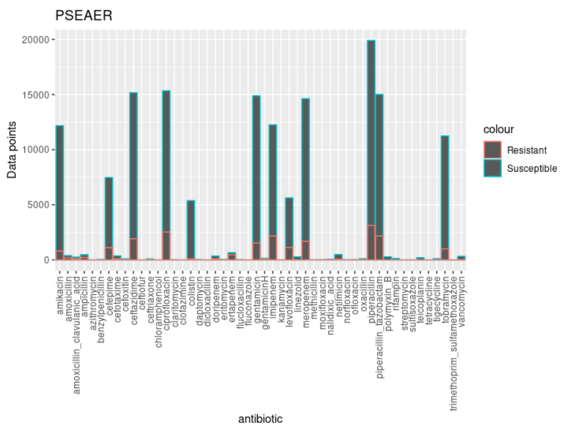


(d) Escherichia coli

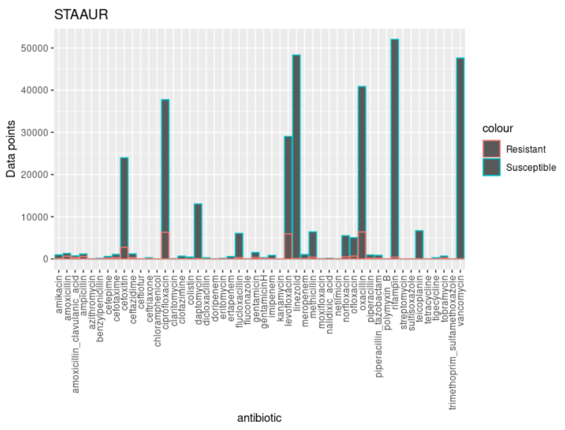
Figure A.1: Number of observations with different antibiotics for different species of bacteria. Note how Enterococcus faecium Enterococcus faecalis have the same antibiotics tested on them with radically different commonality of resistance.



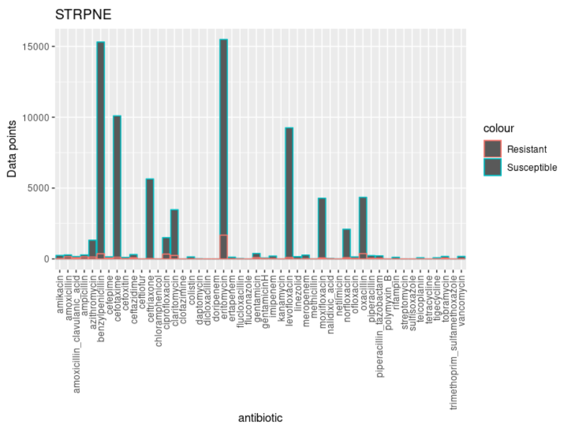
(e) Klebsiella pneumoniae



(f) Pseudomonas aeruginosa



(g) Staphylococcus aureus



(h) Streptococcus pneumoniae

Figure A.1: Continuation of figure A.1

A.2 Additional figures about the result

This section contains some additional figures concerning the performance of the neural networks. In Figure A.2 one can see the performance over varying number of nodes in the first layer.

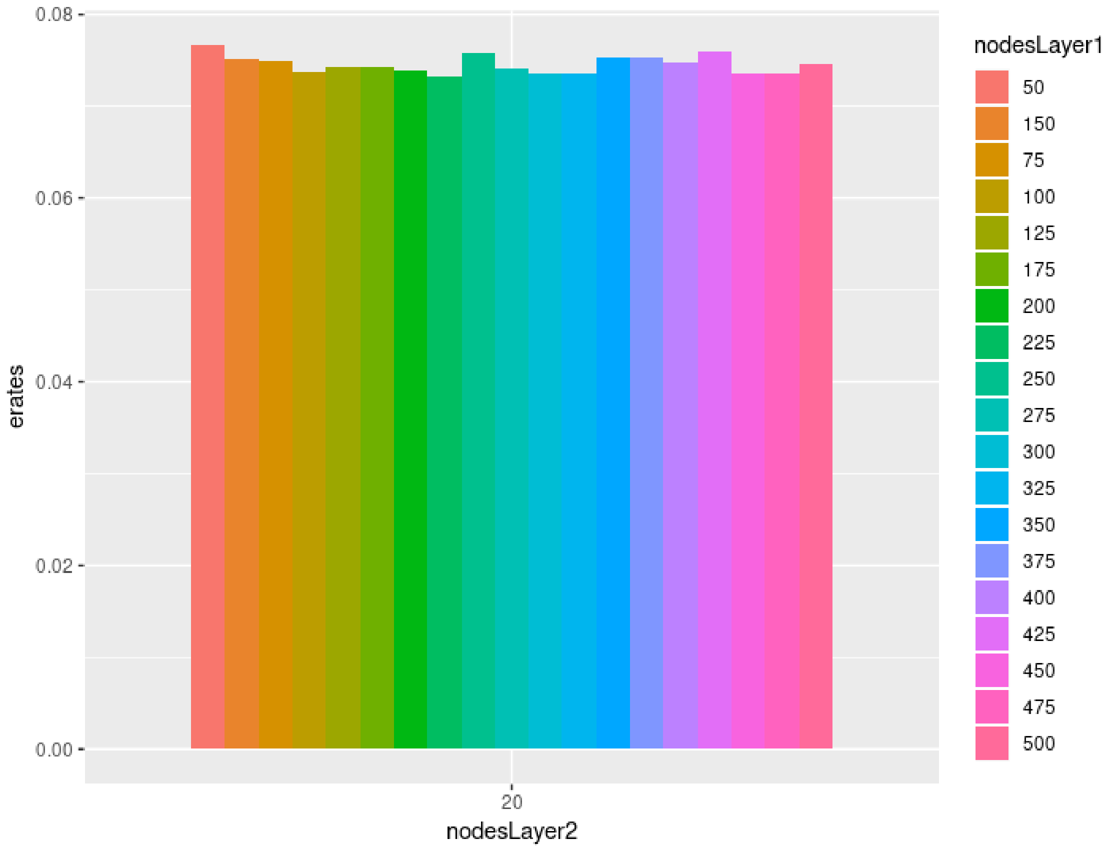


Figure A.2: Error rate for varying number of nodes in the first hidden layer while the number of nodes in the second hidden layer was fixed to 20.

B

Appendix B: Code

```
library(readr)
library(tidyverse)
library(reshape2)
library(dplyr)
library(magrittr)
library(neuralnet)
library(mice)
library(xtable)


# Read the data
    ———————————————————————————————————————————————————————————————
#Reads the data from the file provided by ECDC
#It will come in a dataframe with one row per observation.
#The following variables will be of interest:
#    Antibiotic, specifying which antibiotic was tested
#    ReportingCountry, showing which country the measurement was made
#    Pathogen, showing the species of the bacteria
#    SIR, Showing the result of the test as "R" for resistant, "I" for
    intermediate or "S" for susceptible
data <- read_csv("extracted_data_2017.csv")
nObs<-dim(data)[1]



# Make key for translating three letter short hand to full name
    ————————————
#This key was initially made to compare results from two different data
     sets.
#It is still used to show the full name in graphs and such.

antibioticKey<-tibble(full_name="amikacin",threeLetterAbv="AMK")
i=2
antibioticKey[i,]<-c("amoxicillin","AMX")
i=i+1
antibioticKey[i,]<-c("amoxicillin_clavulanic_acid","AMC")
i=i+1
antibioticKey[i,]<-c("ampicillin","AMP")
i=i+1
antibioticKey[i,]<-c("azithromycin","AZM")
i=i+1
antibioticKey[i,]<-c("benzylpenicillin","PEN")
i=i+1
```

```
antibioticKey[i,]<-c("cefepime","FEP")
i=i+1
antibioticKey[i,]<-c("cefotaxime","CTX")
i=i+1
antibioticKey[i,]<-c("cefoxitin","FOX")
i=i+1
antibioticKey[i,]<-c("ceftazidime","CAZ")
i=i+1
antibioticKey[i,]<-c("ceftriaxone","CRO")
i=i+1
antibioticKey[i,]<-c("ciprofloxacin", "CIP")
i=i+1
antibioticKey[i,]<-c("claritomycin", "CLR")
i=i+1
antibioticKey[i,]<-c("colistin", "COL")
i=i+1
antibioticKey[i,]<-c("daptomycin", "DAP")
i=i+1
antibioticKey[i,]<-c("doripenem", "DOR")
i=i+1
antibioticKey[i,]<-c("eritomycin", "ERY")
i=i+1
antibioticKey[i,]<-c("ertapenem", "ETP")
i=i+1
antibioticKey[i,]<-c("gentamicin", "GEN")
i=i+1
antibioticKey[i,]<-c("imipenem", "IPM")
i=i+1
antibioticKey[i,]<-c("levofloxacin", "LVX")
i=i+1
antibioticKey[i,]<-c("linezolid", "LNZ")
i=i+1
antibioticKey[i,]<-c("meropenem","MEM")
i=i+1
antibioticKey[i,]<-c("moxifloxacin", "MFX")
i=i+1
antibioticKey[i,]<-c("nalidixic_acid", "NAL")
i=i+1
antibioticKey[i,]<-c("netilmicin", "NET")
i=i+1
antibioticKey[i,]<-c("norfloxacin", "NOR")
i=i+1
antibioticKey[i,]<-c("ofloxacin", "OFX")
i=i+1
antibioticKey[i,]<-c("piperacillin_tazobactam", "TZP")
i=i+1
antibioticKey[i,]<-c("piperacillin", "PIP")
i=i+1
antibioticKey[i,]<-c("polymyxin_B", "POL")
i=i+1
antibioticKey[i,]<-c("rifampin", "RIF")
i=i+1
antibioticKey[i,]<-c("teicoplanin","TEC")
i=i+1
antibioticKey[i,]<-c("tigecycline", "TGC")
i=i+1
```

```
antibioticKey[i,]<-c("tobramycin", "TOB")
i=i+1
antibioticKey[i,]<-c("vancomycin", "VAN")
i=i+1
antibioticKey[i,]<-c("clofazimine","CLO")
i=i+1
antibioticKey[i,]<-c("fluconazole","FLU")
i=i+1
antibioticKey[i,]<-c("dicloxacillin","DIC")
i=i+1
antibioticKey[i,]<-c("flucloxacillin", "FLC")
i=i+1
antibioticKey[i,]<-c("methicillin", "MET")
i=i+1
antibioticKey[i,]<-c("oxacillin", "OXA")
i=i+1
antibioticKey[i,]<-c("gentamicinH", "GEH")
nAntibiotics<-dim(antibioticKey)[1]


# Tranform the data
    ------------------------------------------------------------

# A lot of entries have missing data in some of the identifying fields,
     so to identify which row belongs to which patient all the
    identifying variables are combined.
data$UniqueID<-paste(data$Pathogen,data$ReportingCountry,
                     data$Age,
                     data$HospitalId,
                     data$Gender,
                     data$IsolateId,
                     data$PatientCounter,
                     sep=".")
nPatients<-length(unique(data$UniqueID))

#Initialize the new data_frame and set the names of the variabbles.
    Three variables for every antibiotic as well as ID, Country
    Pathogen as Age
NNdata2017<-as.tibble(matrix(NA,nPatients,3*nAntibiotics+4))
colnames(NNdata2017)<-c("ID","Country","Age","Pathogen",paste0(
    antibioticKey$full_name,"_R"),paste0(antibioticKey$full_name,"_S"),
    paste0(antibioticKey$full_name,"_NA"))
NNdata2017$ID=unique(data$UniqueID)

#Initialize the matrix as no data on all antibiotics for every sample
for(s in antibioticKey$full_name){
  NNdata2017[,paste0(s,"_NA")]<-rep(1,nPatients)
  NNdata2017[,paste0(s,"_R")]<-rep(0,nPatients)
  NNdata2017[,paste0(s,"_S")]<-rep(0,nPatients)
}

#Go over the observations
for (i in 1:nObs2017){
  if(mod(i,100000)==0){
    print(i)
  }
  #Check which sample the observation belongs to.
  ind<-NNdata2017$ID==data$UniqueID[i]
  #If this is the first observation of the sample update Age, Country
      and Species
  if (is.na(NNdata2017$Country[ind])){
    NNdata2017$Country[ind]<-data$ReportingCountry[i]
    NNdata2017$Age[ind]<-data$Age[i]
    NNdata2017$Pathogen[ind]<-data$Pathogen[i]
  }
  #Set the value of the corresponding variable according to SIR value
      of the measurement
  antibiotic<-filter(antibioticKey,threeLetterAbv==data$Antibiotic[i])$
      full_name
  NNdata2017[ind,paste(antibiotic,"NA",sep="_")]=0
  if(data[i,"SIR"]=="R"){
    NNdata2017[ind,paste(antibiotic,"R",sep="_")]=1
  }else{
    NNdata2017[ind,paste(antibiotic,"S",sep="_")]=1
  }
}

NNdata2017$Country<-as.factor(NNdata2017$Country)
NNdata2017$Age<-as.factor(NNdata2017$Age)
NNdata2017$Pathogen<-as.factor(NNdata2017$Pathogen)



# Exploring data
    ——————————————————————————————————————————————

#In this section the plots used in 3.1 are generated.

#First we look at the number of resitant and susceptible observations
    per
#antibiotic.

#Initialise the tibble
nObsAnti<-tibble(antibiotic="",ndat=rep(0,0),nr=rep(0,0))
for (antibiotic in antibioticKey$full_name) {
  #Add the number of observations and how many of them are resistant.
  ndat=NNdata2017%>%select(starts_with(antibiotic),-ends_with("NA"))%>%
      sum
  nr=NNdata2017%>%select(starts_with(antibiotic))%>%select(ends_with("R
      "))%>%sum
  nObsAnti<-add_row(nObsAnti,antibiotic=antibiotic,ndat=ndat,nr=nr)
}

#And plot the result.
#The resistant columns will be in front of the total column making it
    easy
#to see how large a portion are susceptible
ggplot(data=nObsAnti,aes(x=antibiotic)) +
  geom_col(aes(y=ndat,colour="Susceptible")) +
  geom_col(aes(y=nr,colour="Resistant")) +
  ylab("Data points") +
  theme(axis.text.x = element_text(angle=90, vjust = 0.5,hjust = 1))
```

```


#Then we separate the data by pathogen and make one plot like the
    previous one
#for every species of bacteria in the data
for(p in levels(NNdata2017$Pathogen)){
  nObsAnti<-tibble(antibiotic="",ndat=rep(0,0),nr=rep(0,0))

  for (antibiotic in antibioticKey$full_name) {
    ndat=NNdata2017%>%filter(Pathogen==p)%>%select(starts_with(
        antibiotic),-ends_with("NA"))%>%sum
    nr=NNdata2017%>%filter(Pathogen==p)%>%select(starts_with(antibiotic
        ))%>%select(ends_with("R"))%>%sum
    nObsAnti<-add_row(nObsAnti,antibiotic=antibiotic,ndat=ndat,nr=nr)
  }
  print(ggplot(data=nObsAnti,aes(x=antibiotic)) +
          geom_col(aes(y=ndat,colour="Susceptible")) +
          geom_col(aes(y=nr,colour="Resistant")) +
          ylab("Data points") +
          theme(axis.text.x = element_text(angle=90, vjust = 0.5,hjust
              = 1))+
          ggtitle(p))
}


#Then the data is separated by country and the same thing is done
for(c in levels(NNdata2017$Country)){
  nObsAnti<-tibble(antibiotic="",ndat=rep(0,0),nr=rep(0,0))

  for (antibiotic in antibioticKey$full_name) {
    ndat=NNdata2017%>%filter(Country==c)%>%select(starts_with(
        antibiotic),-ends_with("NA"))%>%sum
    nr=NNdata2017%>%filter(Country==c)%>%select(paste0(antibiotic,"_R")
        )%>%sum
    nObsAnti<-add_row(nObsAnti,antibiotic=antibiotic,ndat=ndat,nr=nr)
  }
  print(ggplot(data=nObsAnti,aes(x=antibiotic)) +
          geom_col(aes(y=ndat,colour="Susceptible")) +
          geom_col(aes(y=nr,colour="Resistant")) +
          ylab("Data points") +
          theme(axis.text.x = element_text(angle=90, vjust = 0.5,hjust
              = 1))+
          ggtitle(c))
}

#Now we plot the number of observations by country
nObsAnti<-tibble(Country=rep("",0),ndat=rep(0,0),nr=rep(0,0))
for(c in levels(NNdata2017$Country)){

  ndat=NNdata2017%>%filter(Country==c)%>%select(-ends_with("NA"),-
      Country,-Pathogen,-ID,-Age)%>%sum
  nr=NNdata2017%>%filter(Country==c)%>%select(ends_with("_R"))%>%sum
  nObsAnti<-add_row(nObsAnti,Country=c,ndat=ndat,nr=nr)


}
```

```
print(ggplot(data=nObsAnti,aes(x=Country)) +
        geom_col(aes(y=ndat,colour="Susceptible")) +
        geom_col(aes(y=nr,colour="Resistant")) +
        ylab("Data points") +
        theme(axis.text.x = element_text(angle=90, vjust = 0.5,hjust = 1))+
        ggtitle(c))



#Finally we plot the average number of observations on each patient for every country.
#This plot is used in the discussion
nObsAnti<-tibble(Country=rep("",0),meantests=rep(0,0))
for(c in levels(NNdata2017$Country)){

  ndat=NNdata2017%>%filter(Country==c)%>%select(-ends_with("NA"),-Country,-Pathogen,-ID,-Age)%>%apply(1,sum)%>%mean
  nObsAnti<-add_row(nObsAnti,Country=c,meantests=ndat)


}
print(ggplot(data=nObsAnti,aes(x=Country)) +
        geom_col(aes(y=meantests)) +
        ylab("Avarage number of antibiotica per patient") +
        theme(axis.text.x = element_text(angle=90, vjust = 0.5,hjust = 1)))

# Set a testset apart -----------------------------------------------------
nTrain<-floor(nPatients*0.85)
trainInd<-sample(1:nPatients,nTrain)
trainData<-NNdata2017[trainInd,]
testData<-NNdata2017[!(1:nPatients%in%trainInd),]
#save(trainData,testData,antibioticKey,file="TrainAndTestData.Rdata")






# Functions for training the data, and extracting the error rate ----------

#Trains one network for each antibiotic in antibiotics.
#Uses all the other columns in data as indata.
#The vector layer contains the number of nodes in each hidden layer.
#Returns a list with the neural networks
train<-function(data,antibiotics,layers){
  nAntibiotics<-length(antibiotics)
  nns<-rep(list(),nAntibiotics)
  for (i in 1:nAntibiotics){
    s<-antibiotics[i]
    print(paste(i,s))
    trainind<-data[,paste0(s,"_NA")]==0
    #print(sum(trainind))
    formula<-as.formula(paste0(s,"_R + ",s, "_S ~ . -",s,"_NA"))
    traindat<-data[trainind ,]
    done=0
    while(!done){
      nn<-neuralnet(formula,traindat,hidden = layers,linear.output =
          FALSE,threshold = sum(trainind)*5e-5,lifesign = "full",
          lifesign.step = 100)
      nns[[i]]<-nn

      if(!is_null(nn$weights)){
        done=1
      }
    }
  }
  return(nns)
}

#Takes a list of neuralnetworks from the training functions
#as well as the testData and the list of antibiotics.
#Returns the average error of these netowrks on the testdata
getErrors<-function(nns,testData,antibiotics){
  nAntibiotics<-length(antibiotics)
  nError<-0
  modeError<-0
  nTest<-0
  for (i in 1:nAntibiotics) {
    antibiotic<-antibiotics[i]
    testInd<-select(testData,paste0(antibiotic,"_NA"))==0
    tD<-testData[testInd ,]
    if(dim(tD)[1]){
      result<-predict(nns[[i]],tD)
      nTest<-nTest+sum(testInd)
      nR<-sum(tD[,paste0(antibiotic,"_R")])
      nS<-sum(tD[,paste0(antibiotic,"_S")])
      if(nR>nS){
        mode<-"R"
      }else{
        mode<-"S"
      }
      modeError<-modeError+sum(tD[,paste(antibiotic,mode,sep="_")]==0)
      nError<-nError+sum(tD[,paste0(antibiotic,"_R")]!=(result[,1]>
          result[,2]))
    }
  }
  return(tibble(nTest=nTest,nError=nError,modeError=modeError))
}

#Takes one neuralnetwork as well as the testdata for it and returns
    more detailed errors
GetERates<-function(nn,testData,antibiotic){
  if(dim(testData)[1]){
    result<-predict(nn,testData)
    nTest<-dim(testData)[1]
    nR<-sum(testData[,paste0(antibiotic,"_R")])
    nS<-sum(testData[,paste0(antibiotic,"_S")])
    if(nR>nS){
```

```r
      mode<-"R"
    }else{
      mode<-"S"
    }
    modeError<-sum(testData[,paste(antibiotic,mode,sep="_")]==0)
    modeErate<-modeError/nTest
    nError<-sum(testData[,paste0(antibiotic,"_R")]!=(result[,1]>result[,2]))
    Rind<-testData[,paste0(antibiotic,"_R")]==1
    Sind<-!Rind
    nRErr<-sum(result[Rind,1]<=result[Rind,2])
    RErate<-nRErr/nR
    nSErr<-sum(result[Sind,1]>=result[Sind,2])
    SErate<-nSErr/nS
  } else {
    nTest<-0
    nR<-0
    nS<-0
    modeError<-0
    modeErate<-0
    nError<-0
    nRErr<-0
    RErate<-1
    nSErr<-0
    SErate<-1
  }
  ERates<-tibble(antibiotic=antibiotic,nTest=nTest, nR=nR, nS=nS, modeErrors=modeError, Errors=nError, RErrors=nRErr, SErrors=nSErr, modeErate=modeErate, Erate=nError/nTest, RErate=RErate, SErate=SErate)

}






# Check for optimal parameters -------------------------------------------------

#Here we check for a suitable number of nodes in the network.
#First one hidden layer is used and the number of node tested are in nodesPerLayer
#Subsequently we test all combinations of number of nodes in two hidden layers
#from nodesPerLayer, where the later layer has fewer or the same number of nodes.

nodesPerLayer<-c(nAntibiotics/5,nAntibiotics/2,nAntibiotics,nAntibiotics*3)
nodeNs<-length(nodesPerLayer)


#One Layer

```

```r
nTest<-rep(0,nodeNs)
eTest<-rep(0,nodeNs)

for (i in 1:nodeNs) {
  writeLines(paste("\nNumber of nodes:", nodesPerLayer[i],"\n"))
  nns<-train(data,antibioticKey$full_name,c(nodesPerLayer[i]))

  for (j in 1:nAntibiotics){
    antibiotic<-antibioticKey$full_name[j]
    ind<-testData[,paste0(antibiotic,"_NA")]==0
    e<-GetERates(nns[[j]],testData[ind,],antibiotic)
    nTest[i]<-nTest[i]+e$nTest
    eTest[i]<-eTest[i]+e$Errors
  }
}

#2 layers

nTest2<-matrix(0,nodeNs,nodeNs)
eTest2<-matrix(0,nodeNs,nodeNs)

for (i in nodeNs:1) {
  for (j in i:1){
    writeLines(paste("\nNumber of nodes:", nodesPerLayer[i], " and ",
        nodesPerLayer[j],"\n"))
    nns<-train(data,antibioticKey$full_name,c(nodesPerLayer[i],
        nodesPerLayer[j]))

    for (k in 1:nAntibiotics){
      antibiotic<-antibioticKey$full_name[k]
      ind<-testData[,paste0(antibiotic,"_NA")]==0
      e<-GetERates(nns[[k]],testData[ind,],antibiotic)
      nTest2[j,i]<-nTest2[j,i]+e$nTest
      eTest2[j,i]<-eTest2[j,i]+e$Errors
    }
  }
}






#The error rates are calculated and saved in nodeData
nodeData<-tibble(erates=c(eTest/nTest, eTest2[1,1]/nTest2[1,1], eTest2
    [1:2,2]/nTest2[1:2,2], eTest2[1:3,3]/nTest2[1:3,3],eTest2[1:4,4]/
    nTest2[1:4,4]),
                     nodesLayer1=as.factor(c(nodesPerLayer,nodesPerLayer
                         [1],rep(nodesPerLayer[2],2),rep(nodesPerLayer
                         [3],3),rep(nodesPerLayer[4],4))),
                     nodesLayer2=as.factor(c(rep(0,nodeNs),nodesPerLayer
                         [1],nodesPerLayer[1:2],nodesPerLayer[1:3],
                         nodesPerLayer[1:4])))

nodeData$nodesLayer3=rep(0,dim(nodeData)[1])

```

```
#The error rate is plotted
ggplot(nodeData,aes(x=nodesLayer2,fill=nodesLayer1)) +
  geom_col(aes(y=erates),position = "dodge")

# Manually try some node configs
    ____________________________________________

#Below one can manually try some other combinations of node numbers.
#Ln represents the number of nodes in layer n
for(L2 in seq(5,50,3)){
  L1<-150
  #L2<-20
  L3<-0

  if(L3!=0){
    layers<-c(L1,L2,L3)
  }else if(L2!=0){
    layers<-c(L1,L2)
  }else {
    layers<-L1
  }

  nns<-train(data,antibioticKey$full_name,layers)

  n<-0
  ne<-0

  for (k in 1:nAntibiotics){
    antibiotic<-antibioticKey$full_name[k]
    ind<-testData[,paste0(antibiotic,"_NA")]==0
    e<-GetERates(nns[[k]],testData[ind,],antibiotic)
    n<-n+e$nTest
    ne<-ne+e$Errors
  }


  if(!(L1%in%levels(nodeData$nodesLayer1))){
    levels(nodeData$nodesLayer1)<-c(levels(nodeData$nodesLayer1),L1)
  }
  if(!(L2%in%levels(nodeData$nodesLayer2))){
    levels(nodeData$nodesLayer2)<-c(levels(nodeData$nodesLayer2),L2)
  }
  if(!(L3%in%levels(nodeData$nodesLayer3))){
    levels(nodeData$nodesLayer3)<-c(levels(nodeData$nodesLayer3),L3)
  }

  nodeData<-add_case(nodeData,erates=ne/n, nodesLayer1=L1, nodesLayer2=
      L2, nodesLayer3=L3)

}

#The error rate by number of nodes in each layer is replotted
ggplot(nodeData,aes(x=nodesLayer2,fill=nodesLayer1)) +
  geom_col(aes(y=erates),position = "dodge")

#Somewhat arbitrarily chosen...
```

```r
optimalLayer=c(150,20)


# Explore The Stopping Criterion ------------------------------------------

# To explore the stopping criterion threshold, we train the network on a
# fix number of observations with varying threshold
# To accomodate for varying threshold, trainThresh is used.
trainThresh<-function(data,antibiotics,layers,threshold){
  nAntibiotics<-length(antibiotics)
  nns<-rep(list(),nAntibiotics)
  for (i in 1:nAntibiotics){
    s<-antibiotics[i]
    print(paste(i,s))
    trainind<-data[,paste0(s,"_NA")]==0
    formula<-as.formula(paste0(s,"_R + ",s, "_S ~ . -",s,"_NA"))
    traindat<-data[trainind,]
    nn<-neuralnet(formula,traindat,hidden = layers,linear.output = FALSE, #stepmax=stepmax,
                  threshold =  threshold,
                  lifesign = "full",lifesign.step = 100
                  #,err.fct = ef
    )
    nns[[i]]<-nn
  }
  return(nns)
}

#Make a subsample to reduce runtime
nonDataVariables<-c("Age","ID","Country","Pathogen")
NNsample<-trainData[sample.int(dim(trainData)[1],1000),!(names(trainData) %in% nonDataVariables)]

#Try different thresholds and train and test nns on them
for(threshold in c(10^((-1:-7)))) {
  nns<-trainN(data = NNsample, antibiotics = antibioticKey$full_name, layers=optimalLayer, threshold = nSample*5*10^(-5))
  Testsample<-testData#[sample.int(dim(testData)[1],nSample),]

  e<-getErrors(nns,Testsample,antibioticKey$full_name)
  e$testData<-nSample
  e$Erates<-e$nError/e$nTest
  Errors<-rbind(Errors,e)
}

Errors$testData<-as.factor(Errors$testData)
ggplot(Errors,aes(x=testData)) +
  geom_col(aes(y=Erates))


#To check the result of the chosen threshold (included in train) we try some different samplesizes

for (nSample in c(100,1000,10000,100000)) {
  nonDataVariables<-c("Age","ID","Country","Pathogen")
  NNsample<-trainData[sample.int(dim(trainData)[1],nSample),!(names(
      trainData) %in% nonDataVariables)]
  nns<-train(data = NNsample, antibiotics = antibioticKey$full_name,
      layers=optimalLayer)
  Testsample<-testData#[sample.int(dim(testData)[1],nSample),]

  e<-getErrors(nns,Testsample,antibioticKey$full_name)
  e$testData<-nSample
  e$Erates<-e$nError/e$nTest
  Errors<-rbind(Errors,e)


}
Errors$testData<-as.factor(Errors$testData)
ggplot(Errors,aes(x=testData)) +
  geom_col(aes(y=Erates))



# Add Pathogen and Country as indata
    ____________________________________________

nonDataVariables<-c("Age","ID","Country","Pathogen")

#First we reformat the Pathogen and Country test data to suit the
     neuralnet function
# i.e. one binary column per possible value
# We make one dataset where the pathogen data is transformed,
# one where the coutry data is and one where both are
PathNNdata<-trainData
PathTestData<-testData
for(p in levels(trainData$Pathogen)){
  PathNNdata$tmp<-(trainData$Pathogen==p)+0
  PathTestData$tmp<-(testData$Pathogen==p)+0
  colnames(PathNNdata)[dim(PathNNdata)[2]]<-p
  colnames(PathTestData)[dim(PathTestData)[2]]<-p
}
PathNNdata<-select(PathNNdata,-nonDataVariables)
PathTestData<-select(PathTestData,-nonDataVariables)

CountNNdata<-trainData
CountTestData<-testData
for(c in levels(trainData$Country)){
  CountNNdata$tmp<-(trainData$Country==c)+0
  colnames(CountNNdata)[dim(CountNNdata)[2]]<-c
  CountTestData$tmp<-(testData$Country==c)+0
  colnames(CountTestData)[dim(CountTestData)[2]]<-c
}
CountNNdata<-select(CountNNdata,-nonDataVariables)
CountTestData<-select(CountTestData,-nonDataVariables)


PathCountNNdata<-CountNNdata
PathCountTestData<-CountTestData
for(p in levels(trainData$Pathogen)){
  PathCountNNdata$tmp<-(trainData$Pathogen==p)+0
  colnames(PathCountNNdata)[dim(PathCountNNdata)[2]]<-p
  PathCountTestData$tmp<-(testData$Pathogen==p)+0
  colnames(PathCountTestData)[dim(PathCountTestData)[2]]<-p
}

#Then we train the corresponding networks
NNs<-train(data = select(trainData,-nonDataVariables),antibiotics =
    antibioticKey$full_name,layers = c(150,20))
Error<-getErrors(nns = NNs,testData = testData,antibiotics =
    antibioticKey$full_name)
print(Error)

PathNNs<-train(data = PathNNdata,antibiotics = antibioticKey$full_name,
    layers = c(150,20))
Error<-rbind(Error,getErrors(nns = PathNNs,testData = PathTestData,
    antibiotics = antibioticKey$full_name))
print(Error)

CountNNs<-train(data = CountNNdata,antibiotics = antibioticKey$full_
    name,layers = c(150,20))
Error<-rbind(Error,getErrors(nns = CountNNs,testData = CountTestData,
    antibiotics = antibioticKey$full_name))
print(Error)

PathCountNNs<-train(data = PathCountNNdata,antibiotics = antibioticKey$
    full_name,layers = c(150,20))
Error<-rbind(Error,getErrors(nns = PathCountNNs,testData =
    PathCountTestData,antibiotics = antibioticKey$full_name))
print(Error)


# Check the performance
    ———————————————————————————————————————————————

# Now we check the performance of these networks.
# The performance is checked both pathogen wise and country wise.

Errors<-tibble(nError=rep(0,0),nTest=rep(0,0),Country=rep("",0),
    Pathogen=rep("",0),method=rep("",0))

Countries<-levels(testData$Country)
Pathogens<-levels(testData$Pathogen)

#First get country wise error
Errors$method<-as.factor(Errors$method)
for (c in Countries) {
  for (m in levels(Errors$method)) {
    e<-Errors%>%filter(Country==c,method==m)%>%select(nError,nTest)%>%
        apply(2,sum)
    e$Country=c
    e$Pathogen="All"
    e$method=m
    Errors<-rbind(Errors,e)
  }
}
```

```r

#Then pathogen wise
Errors$method<-as.factor(Errors$method)
for (p in Pathogens) {
  for (m in levels(Errors$method)) {
    e<-Errors%>%filter(Pathogen==p,method==m)%>%select(nError,nTest)%>%
        apply(2,sum)
    e$Country="All"
    e$Pathogen=p
    e$method=m
    Errors<-rbind(Errors,e)
  }
}
Errors$Erate<-Errors$nError/Errors$nTest

#Then get total
for (m in levels(Errors$method)) {
  e<-Errors%>%filter(method==m)%>%select(nError,nTest)%>%apply(2,sum)
  e$Country="All"
  e$Pathogen="All"
  e$method=m
  e$Erate<-e$nError/e$nTest
  Errors<-rbind(Errors,e)
}

#And plot the result
png(file = "PathogenErates.png")
ggplot(filter(Errors,Country=="All"),aes(x=Pathogen,fill=method))+geom_
    col(aes(y=Erate),position = "dodge")
dev.off()
png(file = "CountryErates.png")
ggplot(filter(Errors,Pathogen=="All"),aes(x=Country,fill=method))+geom_
    col(aes(y=Erate),position = "dodge")
dev.off()


#Finally we want the to look at major and very major error rate

Errors2<-GetERates(0,tibble(x=rep(0,0)),0)
Errors2<-Errors2[c(),]
Errors2$method<-rep("",0)

for(i in 1:nAntibiotics){
  antibiotic<-antibioticKey$full_name[i]
  print(antibiotic)
  testind<-PathCountTestData[,paste0(antibiotic,"_NA")]==0
  e<-GetERates(NNs[[i]],PathCountTestData[testind,],antibiotic)
  e$method<-"Neither"
  Errors2<-rbind(Errors2,e)
  e<-GetERates(PathNNs[[i]],PathCountTestData[testind,],antibiotic)
  e$method<-"Pathogen"
  Errors2<-rbind(Errors2,e)
  e<-GetERates(PathCountNNs[[i]],PathCountTestData[testind,],antibiotic
      )
  e$method<-"Both"
  Errors2<-rbind(Errors2,e)
  e<-GetERates(CountNNs[[i]],PathCountTestData[testind,],antibiotic)
  em<-e
  e$method<-"Country"
  Errors2<-rbind(Errors2,e)
  em$Errors<-em$modeErrors
  em$Erate<-em$modeErate
  em$method<-"Mode"
  if(em$nR>em$nS){
    em$RErrors<-em$modeErrors
    em$SErrors<-0
    em$RErate<-0
    em$SErate<-1
  } else {
    em$SErrors<-em$modeErrors
    em$RErrors<-0
    em$SErate<-0
    em$RErate<-1
  }
  Errors2<-rbind(Errors2,em)
}

#Plot the very major and major error rate with different colors
E2Melt<-melt(filter(Errors2,method=="Both"),measure.vars = c("RErate","SErate"))
ggplot(E2Melt,aes(x=antibiotic,fill=variable))+geom_col(aes(y=value),position="dodge")+
  theme(axis.text.x = element_text(angle=90, vjust = 0.5,hjust = 1))



# Perform MICE --------------------------------------------------------------

#Transform the data to a suitable form for mice
miceDat=tibble(ID=NNdata2017$ID,Country=NNdata2017$Country,Age=NNdata2017$Age,Pathogen=NNdata2017$Pathogen)
for(antibiotic in antibioticKey$full_name){
  newCol=tibble(rep(NA,nPatients))
  colnames(newCol)<-antibiotic
  rind=NNdata2017[,paste0(antibiotic,"_R")]
  newCol[rind==1]="R"
  sind=NNdata2017[,paste0(antibiotic,"_S")]
  newCol[rind==1]="S"
  miceDat<-cbind(miceDat,as.factor(newCol))
}

#Remove a test set from this data. Because of how mice works one cant remove entire rows.
ptest<-0.15
testindices<-rep(list(),nAntibiotics)
for (i in 1:nAntibiotics){
  a<-antibioticKey$full_name
  nNA<-sum(is.na(miceDat[,a]))
  nTest<-floor((nPatients-nNA)*ptest)
  testInd<-sample((1:n)[!is.na(miceDat[,a])],nTest,replace = FALSE)
  testindices[[i]]<-testInd
  miceDat[testInd,a]=NA
}

miceImp<-mice(miceDat, m=15,maxit=15)

Mode <- function(v) {
  uniqv <- unique(na.omit(v))
  uniqv[which.max(tabulate(match(v, uniqv)))]
}

MiceErrors<-tibble(antibiotic=rep("",0),nTest=rep(0,0),nR=rep(0,0),nS=
    rep(0,0),nErrors=rep(0,0),PredRCorrS=rep(0,0),PredSCorrR=rep(0,0))
for (i in 1:nAntibiotics){
  a<-antibioticKey$full_name[i]
  print(a)
  if(!is_empty(testindices[[i]])){
    pred<-apply(MiceResult$imp[[a]][sapply(testindices[[i]],toString)
        ,],1,Mode)#function(v){WeigtedMode(v,sum(MiceResult$data[[a
        ]]=="R",na.rm=TRUE)/sum(MiceResult$data[[a]]%in%c("R","S"),na.
        rm = TRUE))})
    e<-tibble(antibiotic=a)
    e$nTest<-length(testindices[[i]])
    e$nR<-sum(MiceTest[testindices[[i]],a]=="R")
    e$nS<-sum(MiceTest[testindices[[i]],a]=="S")
    e$nErrors<-sum(MiceTest[testindices[[i]],a]!=pred)
    e$PredRCorrS<-sum(((MiceTest[testindices[[i]],a]=="S")+(pred=="R"))
        ==2)
    e$PredSCorrR<-sum(((MiceTest[testindices[[i]],a]=="R")+(pred=="S"))
        ==2)
    MiceErrors<-rbind(MiceErrors,e)
  }
}








# Set apart error rates on full set to compare the methods
    ________________

MiceErrors$Erate<-MiceErrors$nErrors/MiceErrors$nTest
MiceErrors$SErate<-MiceErrors$PredRCorrS/MiceErrors$nS
MiceErrors$RErate<-MiceErrors$PredSCorrR/MiceErrors$nR

ers<-c(sum(MiceErrors$nErrors)/sum(MiceErrors$nTest),
         sum(MiceErrors$predSCorrR)/sum(MiceErrors$nR),
         sum(MiceErrors$predRCorrS)/sum(MiceErrors$nS)
         )
dataSets=c("Full","R","S")
Erates<-tibble(Method=rep("MICE",3),testData=dataSets,Erate=ers)
modeError<-MiceErrors%>%select("nR","nS")%<%apply(1,max)%>%sum
modeError<-modeError/sum(MiceErrors$nTest)
Erates<-add_case(Erates,Method="Mode",testData="Full",Erate=modeError)

```

```r
e<-filter(Errors2,method=="Both")
ers<-c(sum(e$nErrors)/sum(e$nTest),
         sum(e$predSCorrR)/sum(e$nR),
         sum(e$predRCorrS)/sum(e$nS)
)
er<-tibble(Method=rep("NN",3),testData=dataSets,Erate=ers)

Erates<-rbind(Erates,er)

ggplot(Erates,aes(x=testData,fill=method))+geom_col(aes(y=Erate),
    position="dodge")
```

Appendix/CleanedReading.R