

School of Computer Science | MSc Computer Science

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Designing and creating an artificial neural network in Java to predict the outcome of professional tennis matches**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Author:

Alexander Kelly | 1868842

Tutor:

Martin Escardo

Table of Contents

[Abstract 3](#_Toc523917836)

[Introduction 5](#_Toc523917837)

[Previous Work 5](#_Toc523917838)

[Further Background 6](#_Toc523917839)

[Scope 6](#_Toc523917840)

[Data 6](#_Toc523917841)

[Analysis and Specification 8](#_Toc523917842)

[Solution 8](#_Toc523917843)

[Design 10](#_Toc523917844)

[Key Decisions 11](#_Toc523917845)

[Database 12](#_Toc523917846)

[Network 14](#_Toc523917847)

[Implementation 16](#_Toc523917848)

[Data Structures 17](#_Toc523917849)

[Algorithms 18](#_Toc523917850)

[Testing 20](#_Toc523917851)

[Unit Testing 20](#_Toc523917852)

[Integration Testing 20](#_Toc523917853)

[Black Box Testing 21](#_Toc523917854)

[Evaluation 22](#_Toc523917855)

[Results 22](#_Toc523917856)

[Project Management 22](#_Toc523917857)

[Conclusion 23](#_Toc523917858)

[References 24](#_Toc523917859)

[Appendix 25](#_Toc523917860)

[Functional Requirements 25](#_Toc523917861)

[Non-functional Requirements 26](#_Toc523917862)

[Software Instructions 27](#_Toc523917863)

[Project Proposal 28](#_Toc523917864)

# Abstract

*Predictive modelling is widely used to great effect in areas including weather forecasting, business analytics and commercial sports markets. In this report, I present the results from a project that forecasts the outcome of ATP tennis matches between 2017/18, derived from historic ATP tennis match data. It has been found that deep learning, specifically the neural network model, is able to produce a consistently high rates of prediction accuracy.*

*Keywords: predictive modelling, tennis, deep learning, neural network model.*

*All software from this project can be found at the following remote Git repository*:

https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2017/ajk742.git

# Introduction

In this project, the machine learning method of artificial neural networks has been explored in forecasting sports results. The aim was to design and implement an artificial neural network, for the purpose of predicting the outcome of ATP tennis matches. Previous tennis match data, including the court surface, round of the match, both players height, nationality and many more categories, were used to train and test the network.

There have been numerous attempts at using machine learning to predict the outcome of sports matches. However, in the majority of cases, these are less accurate than human prediction (assuming the predictor has an extensive knowledge of the particular sport).

Commercial businesses, most notably bookmakers, create their entire services based on sports prediction. At the heart of their methods are complex statistical algorithms and machine learning, but they also rely heavily on humans with specialist knowledge analysing data and submitting manual predictions.

## Previous Work

Several research papers on the subject of forecasting tennis matches specifically have been released in the past few decades.

A study from Barnet & Clarke (2005) used a combination of standard ATP tennis statistics to predict outcomes, ranging from the probability of players winning a single point, to the probable length of the match, and probability of each player winning.

In a similar approach to the project detailed in this report, Somboonphokkaphan & Phimoltares (2009) used the artificial neural network, and a dataset consisting of multiple attributes from previous matches, including the player’s first serve percentage and number of points won behind that serve, to predict match outcomes. The model was used with great effect on Grand Slam tournaments (The Australian Open, Roland Garros, Wimbledon, and the US Open) between 2007-08, and predicted with an accuracy of 75%.

# Further Background

## Scope

This application creates an artificial neural network that can be trained on a dataset using backpropagation. The training is supervised as inputs to the network have a corresponding output which is used in calculating the loss function during the backpropagation procedure. The network processes data in the form of a comma separated value (CSV) file, and transforms it into normalised numerical data that can be used as input and expected output for the network. Following training, the network is able to make predictions on data sets not previously seen.

## Data

This section provides a brief summary of the data used to train and test the network.

The data comprises details of previous tennis matches. One row within a CSV file contains the details for one individual match, and one file contains all matches from one year. The data from the files is read into database tables and separated into training and test data.

The input column headings for the network used in training and testing the network are as follows:

* tourney\_id (unique identifier for each particular tournament)
* surface (the court surface, for example grass, clay, hard or carpet)
* draw\_size (the number of players contesting that particular tournament)
* tourney\_level (the standard of the tournament)
* winner\_id (unique identifier for the winning player)
* winner\_seed (the seed of the winning player for this particular tournament, number between 1-33, if the player is unseeded in this tournament, this entry will be null)
* winner\_hand (whether the winning player is left or right-handed)
* winner\_ht (the height of the winning player)
* winner\_ioc (the nationality of the winning player)
* winner\_rank (the winning players world ranking)
* loser\_id (unique identifier for the losing player)
* loser\_seed (the seed of the losing player for this particular tournament, number between 1-33, if the player is unseeded in this tournament, this entry will be null)
* loser\_hand (whether the losing player is left or right-handed)
* loser\_ht (the height of the losing player)
* loser\_ioc (the nationality of the losing player)
* loser\_rank (the losing players world ranking)
* best\_of (the number of sets either 3 or 5)
* round (in which round of the tournament the match is, for example semi-final)
* year (the year in which the match took place)
* match\_id (unique identifier for the row in the table)

The output column headings for the network used in training and testing the network are as follows:

* better\_rank\_won (whether the better ranked player won the match)
* worse\_rank\_won (whether the worse ranked player won the match)

The composition of the data sees the players differentiated by which player won, and which player lost the match.

The network is to be trained on data from the beginning of 2008 up until the end of 2016, and tested on data from 2017 and the beginning of 2018.

# Analysis and Specification

Prior to writing any code, the functional and non-functional requirements for the system were drafted in order to understand the services that should be provided. The requirements were used as a form of guidance to ensure that the system would meet its purpose, as well as adhere to any relevant constraints on its operation. Due to this being a completely unfamiliar topic, the requirements were updated throughout. They can be found in the appendix of this report.

## Solution

The solution to the problem posed by this project is best described by analysing it from a broader perspective. In most circumstances that require prediction, there is no well-defined algorithm that exists which is able to solve the problem. Given input data and an expected output, a neural network is able to find hidden dependencies in the data, allowing it to make an accurate prediction on what output is likely, given a different set of inputs. The problem can therefore be broken down into a form of pattern recognition. The architecture of a neural network makes them specifically suited to solving this class of problem.

The network is composed of layers of neurons, starting with an input layer, one or more hidden layers, and an output layer (the presence of at least one hidden layer is what makes a neural network a multi-layer perceptron). In a fully connected network, each neuron at a given layer is connected to every neuron in the next layer, and each connection has a weight associated with it. The changeability of the neuron to neuron connection weights is what allows for the patterns in the data to be learned.

A brief overview of specifically how the network solves the problem will now be discussed. This consists of three steps:

The first step of the process involves feeding the input data (the details from one specific tennis match) forward through the network, starting at the input layer. As the input layer has no previous layers, it is used as a buffer, and the input data becomes each neuron in that layer’s output. The role of the neurons in all layer’s excluding the input layer can be separated into two parts. Firstly, they sum the output from all neurons in the previous layer, and secondly, apply an activation function to the total and feed the value forward to each neuron in the next layer. For each individual neuron in the first hidden layer (the layer directly after the input layer), the output from each input neuron is summed, and multiplied by the weight which connects the neuron in the previous layer (input layer) to the current neuron in the first hidden layer. This new value goes through an activation function, and the value obtained is the output of the particular neuron in the first hidden layer. If the output value is above a certain threshold, the neuron sends its output signal (number between -1 and 1) to all the neurons in the next layer. If it is below the threshold, no output is sent. This process continues until the output layer is reached and the final value becomes the output of the network.

The second step involves calculating the error at each neuron. The goal is to make the output of the network as close as possible to the target output. The input to the network cannot be changed as this is fixed, so the network needs to update its neuron to neuron connection weights in order to converge towards the target output. The process begins by calculating the loss function of the network, which is the difference between the actual output (the output from the output layer), and the target output. This value will be multiplied by the derivative of the chosen activation function of that particular neurons output (the reason behind this is explained in more detail below).

In order to minimise the loss function, it is necessary to propagate the error back through the neurons in each layer of the network, and update their weights. This process is called backpropagation of the error, and is a fundamental part of the network training process. As suggested previously, for backpropagation to be possible, the derivative of the activation function needs to be calculated. The derivative of any function gives the rate at which the function is increasing or decreasing values at that specific point. It further tells the network whether increasing or decreasing a particular weight will make the network minimise the loss function, which is the goal of training the network. It is essential that the network is aware of the rate at which the error changes, relative to the change in weight, so that the weights can be updated accordingly.

The next part of the error calculation stage involves propagating the error, calculated at the output layer, back through the network. The algorithm for this is as follows: the error at a hidden layer neuron is equal to the sum of the weight connecting this neuron, and the neurons in the next layer, multiplied by the next neurons error (already calculated as it is being back-propagated), multiplied by the derivative activation function value of this neurons output. This process stops at the input layer as no calculations have taken place before this point.

Updating the weights is the third and final training step. The underlying purpose of backpropagation is to optimize the connection weights to allow the network to correctly predict outputs based on inputs. Optimising the weights at each neuron decreases each neurons error, and consequently the overall error of the network.

To update the weight at a given neuron, the change in weight is calculated first, using the following algorithm: the negative value of the learning rate (small constant which decides the speed of the networks learning) multiplied by the output from the neurons connected in the previous layer, multiplied by this given neurons error (calculated in the previous step).

During each training epoch, the mean square error (MSE) of the network’s actual output is calculated. This is used as a performance indicator for the network. The lower the MSE, the more accurate the predictions that are being made by the network.

# Design

The structure of the system is separated into two fundamental components, the database, and the neural network.

The role of the database code is to prepare the data for the network. This constitutes a number of steps as follows.

1. Establishing a connection to the relevant PostgreSQL database. This proves vital as the connection is used at all other steps of the database component.
2. Creating separate tables within the database for the training and test data.
3. Reading in all CSV files into the training data table.
4. Deleting the unneeded columns from the table.
5. Remove players with a ranking below fifty in the world from the data set.
6. Add a primary key to the table so each entry can be uniquely identified.
7. Create output columns.
8. Standardise and normalise the non-numeric data.
9. Normalise the numeric data.
10. Transfer the data that will be used to test the network over to the test data table.

The role of the network is to create, train and test the neural network. This encompasses the following steps.

1. Create a neural network with any number of layers, with a specified number of neurons in each layer. During the creation process the weights connecting each neuron, and the biases are randomly initialised between 0-1.
2. Create a set of data to train the network. This involves reading the normalised training data from the training table and storing it.
3. Train the network on the data set created in the previous step. This involves the three systematic stages described in the previous chapter. The following will briefly describe how these stages are carried out in the software:
   1. Calculate the output of the network. This simply involves passing the input data set forward through the network, beginning from the input layer. The method loops through the layers, and neurons in each layer accessing a specific neuron at a given layer starting from the first hidden layer. It takes the sum of the output from all connected neurons in the previous layer, multiplied by the connection weight, adds the current bias, and applies the sigmoid activation function. The output from each neuron is temporarily stored. This process continues until the loop has reached the final hidden layer and this output is returned.
   2. Calculate the error at each neuron. The loss function is the actual output, which can be accessed from training set of data, minus the output from the network, which has been calculated in step a. Specifically, the error at the output neuron is the actual output minus the expected output multiplied by the derivative sigmoid of that neurons output (calculated during the feed forward process in step a). The error from the output neurons are stored. The method then loops through all other layers starting from the last hidden layer, where, for each neuron in the next layer (the output layer in this example), the connecting weight is multiplied by that neuron’s error, and summed. Once this has been completed for each neuron in the next layer, the total is applied to the derivative sigmoid value of the current neurons output (in the last hidden layer in this example) and this value is its error, which is stored for the process to continue propagating back the error through the network. This process gives the gradient of the error with respect to each weight, which effectively demonstrates whether the network needs to increase or decrease the weight to minimise the loss function.
   3. Finally, the weights are updated. The method starts looping at the first hidden layer, for each neuron the change of weight must be calculated. The formula for this is the negative learning rate multiplied by the output from the previous neuron (calculated in step a) multiplied by the error at the current neuron (calculated at step b). This value is summed for all the neurons connected in the previous layer, and the weight is updated accordingly. Updating the bias follows the same steps, but as it is not connected to any neurons in the previous layer, the algorithm for updating is simply the negative learning rate multiplied by the error at the current neuron (calculated at step b).
4. The MSE is calculated. This involves calculating the target output for each entry (which can be accessed from the training set created) minus the actual output (which is the output at the output layer), and this value is squared, divided by two and multiplied by the length of the output. As this gives the MSE for each individual entry in the set, the MSE for the entire set are summed, and divided by the length of the set, in order to give the average MSE over all data, for each individual training epoch. This is the final part of training the network.
5. Once the above training steps are completed, the network is tested on an unseen data set. The set is created using the same step as number 2, the only difference being that the data is taken from the test data table.
6. Testing the network involves the network running through the test data once, and storing its predictions. The predictions are compared to the target output using a single method that does the following: takes the actual output at a given entry (starting from the first entry in the test set and looping through the entire set), this will be an array containing two values (either [0,1] or [1,0] depending on whether the higher or lower ranked player won). The method checks at which index the highest value is for the actual and target output (either index 0 or 1) and compares whether these are the same. If they are at the same index, that entry goes down as a correct prediction. All the correct predictions are summed, and the value is divided by the size of the training set and multiplied by one hundred; the value left is the percentage of correct predictions.
7. The network can then be saved to a file using an object output stream.
8. Saved networks can be loaded using an object input stream.

## Key Decisions

This section will explore some of the key decisions including algorithms used, the reasons behind using these, alternatives that were considered, and the reasons why they were not selected. These are separated into the database and network decisions.

### Database

The first decision involved selecting, out of the 50 categories of data available in the CSV files, which should be used to make the network predictions most accurate. The initial intention was to use my own intuition, and knowledge of the sport of tennis, to decide which categories would have the most influence on match results. I wanted to explore correlations between certain categories. For example, the players height often determines the speed of the serve, and the players with faster serves are more likely to have success on the quicker court surfaces such as grass. Similarly, the players nationality often influences the type of court they grew up playing on, which will be their favoured surface, and consequently the surface they are more likely to have success on. I then intended to record the results based on different combinations of categories, and compile the most successful network. Unfortunately, due to unforeseen challenges and issues, I was able only to train and test the network using the original categories chosen.

In total, sixteen of the original categories were selected to train and test the network. This lead to the network requiring sixteen input neurons, one for each category, in order to feed all of the relevant data through the network.

All data is read into the training data table in preparation for the normalisation step. This is to ensure that all data of the same type will be normalised on the same scale. For example, the arbitrary normalised values for the non-numeric data is required to be the same value for both training and test data. If this precaution was not taken, the network would not recognise the same data as being equivalent between the two tables, making the dependencies learned during training irrelevant for that data.

The winner and loser seeds were the only columns allowed to contain blank values, because not all players in a tournament are seeded. Only the top thirty-two players, or thirty-three in rare circumstances, are awarded a seed. The unseeded players needed to be attributed a value to demonstrate to the network that they were less favoured than the seeded players. As the seeds show that the lower the number, the more favoured the player, it made sense to attribute a number greater than the highest available seed to all the unseeded players. I therefore decided to assign all unseeded players a value of forty, and this would serve as the upper bound for normalisation of this column.

During the normalisation process, although the players data is separated into winner and loser columns, all *like variables* within those columns were required to be normalised on the same scale. For example, it is necessary for the columns containing the winner and loser heights to be scaled together as they contain the exact same type information, just separated by who won and who lost that specific match. Given this example of player height, the network is interested in finding correlations between the height of the player and their likelihood of winning certain matches. The players height is the same regardless of whether they won or lost the match, it is an unchanging variable, and is the same type of data for both players. Therefore, it makes sense for both winner and loser height to be normalised together, on the same scale. This is the same for the players rank, seed, hand and nationality. This process involves comparing the lowest winner and loser values for each column, the lowest value becomes the lower bound, and then repeating the process for the highest value, which becomes the upper bound (full description of the algorithm can be found in the implementation section of this report).

The reason behind the decision to remove lower ranked players was derived from the intention of the network. It is specifically designed to make predictions on players who, at the time of the match, are ranked inside the top 50, and to make predictions on how they fair against other players, also ranked inside the top 50 at the time of the match. If a player slips outside of the top 50, their results during that period are ignored. The results of such matches (with the exception of the top four players) are often quite unpredictable. The network is designed to attempt discover, in what rare set of circumstances the top four players lose matches, as well as dependencies between the remaining unpredictable results.

Formulating a sufficient output that could be learned by the network proved a cumbersome task. This was primarily because the data downloaded came with no *natural* output as to who won or lost the match. The players in each entry are already differentiated as the “winner” and “loser” rather than player one and player two for example. This meant that it was not as simple as creating an output column for which player won, as it would always be the “winner”, and if the network was to predict future matches, this information would not be available. I decided to find a unique input for each player, that could be compared, to show the network which player was victorious. The most suitable input was the players ranking, as one player will always be ranked higher, and the other ranked lower. The method involved creating two output columns “better\_rank\_won” and “worse\_rank\_won”, and if the lower ranked player won, “better\_rank\_won” is assigned a value of one, and “worse\_rank\_won” a value of zero, and vice versa. The network will therefore have two output neurons, and its predictions will be either [1,0] or [0,1], or as close to those values as possible depending on the success of the training process.

In preparing the data for the network, it has already been mentioned that the numeric data was normalised. The upper and lower bounds of the normalisation were 0 and 1 respectively. The central reason behind choosing to use normalisation is that all the raw data that the network is to be trained on is not initially equal from a numerical perspective. For example, the column draw\_size has a lower bound of 4 and an upper bound of 128, whereas the winner\_seed and loser\_seed columns have a lower bound of 1 and an upper bound of 33. Feeding this raw data into the network would result in these two categories of data carrying different weight in terms of their relative effect on the network, leading to skewed results.

The Z-score normalisation technique, which normalises a value based on the mean and standard deviation of the data, was considered as a viable option for normalising the data. However, this method does not assure that all the values will be normalised to a common numerical range, which I deemed to be of paramount importance.

The median absolute deviation was considered due to its robustness; however, this technique risks losing the original input distribution, and, similar to the Z-score technique, does not convert the values into a common numerical range.

The min-max algorithm was selected because it transforms the values into a common scaled range, while retaining the original distribution of the data.

### Network

It should be mentioned that there are other methods available in the field of computer science which offer an alternative solution to this problem in question. Some of the most popular of which being linear classifiers and support vector machines. However, neural networks were the obvious choice for this particular project as they are able to outperform these alternatives in terms of performance.

The weights and biases are initialised at random and not to a set value for the simple reason that the learning algorithm is greedy in nature. A random initialisation would therefore break the symmetry. If the weights were always initialised to the same values, the algorithm would always converge towards one particular set of weights. This could be a local optimum, which would result in the network always missing the global optimum.

Neural networks can perform better or worse depending on the number of layers and the number of neurons in each layer. This is a decision that requires experimentation, primarily trial and error to discover the best performing network. From my research, I concluded that for a network being trained and tested on a similar number of inputs as this one, the number of hidden layers should be in the range of one to three. In general, the more neurons in a network implies that a more complex function can be learned, however this leads to an increased risk of overfitting. I originally tested the network with two hidden layers, experimenting slightly with the number of neurons in each layer, but found no great difference in performance, including training time and accuracy. I intended to test the network on different number of layers and neurons per layer, record the results, and save the best performing network. However, due to the unforeseen delays mentioned previously, I was unable to do so.

The sigmoid activation function was selected as the transfer function for each neuron for the simple fact that it takes all numbers and squashes them between the values zero (if the value is a higher negative number), and one (for larger positive values). It can therefore clearly represent the firing rate of a neuron: given a higher negative number, converging towards zero and not firing, to a higher positive value, leaving the function fully-saturated, closer to one, and firing. Furthermore, the main issue with the sigmoid function in terms of the potentially vanishing gradient (arising when the derivative terms less than one are multiplied by each other multiple times, such that the values become increasingly smaller until the gradient becomes increasingly closer to zero) only occurs when the number of hidden layers increases past a certain point. This is not an issue with this network because, as mentioned previously, the number of hidden layers should not surpass three.

In simple terms, the learning rate is a constant value that specifies how quickly the network will learn the desired output. If it is set too high, this will result in faster convergence toward the desired output, but runs the risk of missing the optimum. If set too low, the learning process could be significantly slower, although increasingly likely to become more accurate. The number decided upon for this network was 0.3, as much of the research on this topic concludes that this value is neither too large or small.

The amount of training iterations was not specifically a decision to be made as (up to a certain point) the more iterations, the more accurate the predictions will become. The network was tested on several iteration amounts starting with zero, which gave a prediction accuracy of 34.69% (although this is seemingly more luck than judgement), one iteration, with an accuracy of 65.31%, up to one hundred-thousand iterations, which gave an accuracy of 98.89%.

The MSE and mean absolute error (MAE) were the two options considered as a metric to test the accuracy of the networks predictions. Although both are popular choices in predictive modelling such as this, the MSE is the easier of the two with relation to computing the gradient. Another desirable property of the MSE is that it punishes higher errors significantly more than lower ones, by taking the square value.

As mentioned, the data is divided into two subsets, one to train the network and the other to test it. The training data is required to be a sufficiently larger sample so that the network can successfully learn from it. Similarly, the test data must be such that it yields statistically meaningful results, as well as also being representative of the data set as a whole. In this project, the data as a whole spanned ten years. The percentage split between training and test data recommended by the majority of machine learning experts is usually in the range of eighty to twenty respectively. Therefore, it made sense to train the network on the first eight years of match results, and test it on the final one and a half years. This should allow the network to generalise well to unseen data, and avoid the problem of overfitting (learning the training data as opposed to the hidden dependencies).

Finding a technique to test the networks accuracy seemed initially as though it would prove a cumbersome task. Notably, because the goal of testing is to verify whether the output of the network matches the expected output. It became apparent from the predictions during the training process that the actual output never exactly matched the expected output. However, the testing method was implemented as it neutralised this potential issue by checking the index of the highest value, rather than whether the values matched.

# Implementation

This section will cover the execution of the software, and elaborate on the some of the more important algorithms mentioned in previous sections of this report. The most significant decisions relating to the data structures used will also be discussed.

Initially, only one database table was created, called training data, and all information from the CSV files was read in, including the data to test the network. This was because all of the training and test data needed to be normalised on the same scale, so having them present in a single table meant that the normalisation operation could be performed simultaneously on both, while also ensuring the same scale for all data.

Reading in the data proved a slightly more cumbersome task than anticipated due to the presence of *bad data* within the files. For example, in the majority of columns that contained numeric data such as the player’s height, rank or seed, if the actual value was not available, the particular entry would contain character values rather than being left empty. This lead to the database not allowing the columns to be initialised to the correct data types, but instead all of them being initialised to type varchar so as to accommodate all potential types of data. This lead to additional steps in preparing the data for the network. Most notably, for columns that contain unchanging, constant values that can be taken from another row in the database, a number of methods were created within the Database package, in the CleanDatabase class that performs the following function: checks whether the column in question has any rows which are empty. If it does, the method takes the unique loser or winner ID from that entry (depending on whether it is the loser or winner who has the blank entry), and checks whether that player has any other entries within the table where the particular blank entry is filled in. If it does, the method fills in the value in the initially empty entry and moves on to subsequent rows. If not, the entire row is deemed unusable and is therefore deleted.

The operation of reading in the data from the CSV files took advantage of batch processing as this lead to greater efficiency. This was explicitly stated in the non-functional requirements, and successfully implemented within the software with the desired effect.

This application utilised several resources a lot during the database segment of the code. Once opened, these resources must be handled with a great deal of care. In order to ensure that the resource was closed or released even in the event of an error, the program initialised all auto closable resources within the parentheses of the try statement. This ensured that even in the event of an unexpected error, the resources were closed, and also maximised readability of the code. From a performance perspective, this prevented the possibility of resource exhaustion, and the need for application servers to be restarted unnecessarily often.

Any attempt to query or update the data within the database was carried out using the PreparedStatement object. The primary reason behind this was performance, as the SQL statement is precompiled and stored in a PreparedStatement object. This object can then be used to efficiently execute this statement multiple times. The statement is therefore compiled just once and reused, which negates the need to compile the same statement each time the query or update is run, which would consequently decrease performance. Secondly, this was taken as a security measure to guard against SQL injection attacks. This is achieved by separating the query string and the data, and sending them individually.

As an additional security measure, all methods which require input are protected against bad parameters by throwing an IllegalArgumentException. If the method is unable to perform its desired task due to the parameters passed to it, an informative error message will be displayed clearly detailing what brought about the error, and how to rectify it.

## Data Structures

The following will include a brief summary of the main data structures incorporated throughout the software, and the reason for their inclusion. In some circumstances, it will also discuss alternatives considered.

Within the database package, the majority of classes contain methods that read in data from the tables and perform some kind of operation. The two main operations performed are firstly, adding data which meets a certain criterion to a data structure, and secondly, looking up or selecting the data by its index. The data structure chosen in all such cases is the ArrayList as it has a time complexity of O(1) for both of these operations. The most notable uses of this can in fact be found in the NeuralNetwork package, specifically the NetworkData class, where an ArrayList of type two-dimensional float array is created to store the input data, at position zero, and the output data, at position one. This particular data structure is used when creating the network data set for training and testing the network, which involves adding all of the data to a network set (looping through and adding items to the end of the list), it is also used during the training process to look up the values at each index. Using this particular data structure primarily helps make training a large data set as efficient as possible.

As an alternative, the LinkedHashMap data structure was considered to map the input data to its corresponding output, while crucially maintaining the order of the data. However, as there is no intuitive way to look up data by its index using a map, it was concluded that this did not offer a better solution.

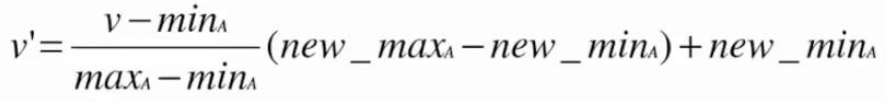
As mentioned in the design section of this report, during the normalisation process, the winner and loser column values of the same type are normalised on the same scale. Once the minimum and maximum values have been obtained, the next step involves selecting the distinct values from both the winner and loser columns to be normalised. However, this causes an issue with potentially duplicate data as the same player will both win and lose matches, therefore their distinct value will be present in both columns at some point. In order to counteract this problem, the HashSet data structure is used, as duplicate values are not allowed. Within this same method, the HashSet is transferred into an ArrayList, as the values later need to be accessed by index.

The majority of the remaining software uses simple arrays as the data structure of choice, due to better speed and performance when compared to Collections, as well as more efficient memory allocation.

## Algorithms

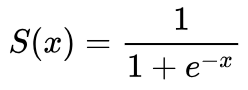
This section will explain the most fundamental algorithms used in the software, as seen in their order of execution in the program.

Firstly, the min/max normalisation algorithm is denoted as follows:



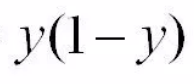
If min A is the minimum value in the particular column being normalised, and max A is the corresponding largest value, the algorithm maps a value v of A, to its normalised value (v prime) in the range between new\_max A and new\_min A as the new upper and lower bounds respectively. As mentioned previously, this is performed with the purpose of retaining the original distribution of the data.

The sigmoid activation function can be denoted as follows:



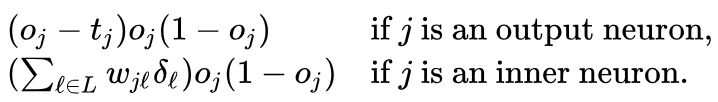
The function works by making the denominator progressively larger for higher negative values of x, which leads to the value of the function gradually approaching zero. In exactly the same way, for higher positive values of x, the exponential term vanishes, and the value of the function will converge towards one. This works adequately as the networks activation function to ascertain whether a neuron fires or not.

If the derivative of the sigmoid function of S(x) is written as y, the algorithm for calculating the derivative can be denoted simply as:



As mentioned previously, this is used crucially in the backpropagation stage to give the rate at which the function is increasing or decreasing values at a specific point.

The backpropagation algorithm, used in aiding the network to learn the given input data by calculating the error at a given neuron, can be denoted as follows:

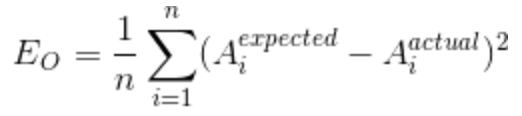


The algorithm is slightly different for output neurons compared to hidden layer neurons as the error must be propagated back through the network. For output neurons, the first part of the algorithm (oj - tj) denotes the actual output at a given output neuron (oj) minus the expected output (tj). The second part is calculating the derivative sigmoid function of the output at this output layer neuron, j.

For hidden layer neurons (inner neuron), the first part denotes the sum of the errors in the next layer, multiplied by the connected weights. The second part is the same as for output neurons; the derivative sigmoid function of the output at this hidden layer neuron, j.

The change in weights is very easily calculated once the error at each neuron has been calculated. The formula is the negative value of the chosen learning rate, multiplied by the error. As the learning rate is a static number, and the error has already been determined, calculating this just involves very simple multiplication of the two.

The MSE, used for calculating the accuracy of the network, can be denoted as follows:



“Given n output neurons, for each output neuron's weighted sum A, the training program computes the difference between the value from the training data and the value from the network, squares it, sums those values for all output neurons, and divides by the number of output neurons (n) to arrive at the total output error E” Perry (2018). This process is completed following each training epoch to depict how accurately the networks predictions are evolving.

# Testing

This section details the testing strategy incorporated within the project to verify that the program logic was correct, and to ensure that the software was reliable, and free from errors.

## Unit Testing

Where possible, unit testing was implemented to guarantee that running specific methods with valid inputs resulted in the expected outcome. The two most notable unit tests involved ensuring that the sigmoid activation function was implemented correctly, and also the method used in testing the networks predictions matched the expected output worked without any errors.

## Integration Testing

The unit test coverage was only very minor due to the majority of the code involving a connection to the PostgreSQL database. As a result of this, integration testing was used extensively to ensure these systems interacted in the manner expected. In conjunction with this, the database code was also tested in terms of performance, as some queries are considerably more efficient in MySQL compared with PostgreSQL, for example.

Many techniques were considered as an option for testing the database code, due to the inadequacies of unit testing for this purpose. The most viable of which involved creating a mock database to simulate the integration of the system. However, research concluded that creating a mock database involves an abundance of time and effort, and the end result would have seemingly deviated largely from the implementation of the current software. Therefore, the tests created would provide very limited meaning or validation. Fortunately, as the database and its contents were available throughout the development process of this software, this relinquished the need to use a mock database. Testing could take place on real data that would be encountered during the actual execution of the program. This proved very useful in ironing out minor errors, and also for observing data integrity issues that required attention.

Each component of the database was tested individually in order to certify that it was manipulating the data as intended. Initially, to simulate the program, a very small amount of training data (five rows) was read into the database. The methods were tested on the data, in the order of execution, and print line statements were used in the console to demonstrate the correct behaviour. For example, when standardising and normalising the non-numeric data, the program contained print line statements that displayed the distinct values of the particular column, and the arbitrary float value that it was being updated with in the training data table alongside each other. Similarly, for the numeric data, which was to be normalised using the minimum and maximum values for that particular column, the minimum and maximum values were printed to the console, followed by the distinct column values in ascending order, alongside the new normalised float value which the old value replacing the original value in the database table. As the data was tested on such a small sample data set initially, it was a simple task to keep track of the actual and expected values. To ensure that the update statements had worked correctly, simple SQL queries were used to retrieve and check the new data.

## Black Box Testing

Testing of the neural network was a meticulous and relatively cumbersome process. The technique utilised was black box testing. A simple two-layered network was initialised, and the biases and weights were printed to the console. A single line of input data was fed into the network, and the calculate output method was run, with the expected output being recorded before-hand and compared with the actual output. Once the two matched, it was clear that there were no errors portion of the network, and that it was functioning correctly. The same steps were followed for the backpropagation of error and updating of the weights until all methods operated correctly individually. The methods were then tested together. Following this, the network was tested on a two-dimensional input, if any modifications were required to the code, the training process would start over, and the network would be reinitialised and tested on a single line of input.

# Evaluation

This section will evaluate the success of the software project as a whole, as well as the successfulness of the management and organisation of the project throughout. Crucially, the functionality of the software will be compared with the original specification.

## Results

## Project Management

The management and organisation of the project was crucial in ensuring that the software was produced within the time-scale, and to the desired standard. Although no single strategy was followed stringently, the technique adopted mirrored a hybrid of the Agile and Kanban approaches in managing the project.

The agile methodology was selected primarily due to the unfamiliarity with neural networks. This meant that the requirements were unlikely to be fully understood during the planning stage of the project, and would need to be responsive to change. The requirements were drafted, and the priorities of the project were established, as opposed to extensive planning. Similarly, due to the short time scale of the project, creating comprehensive documentation would have wasted the limited time available.

A brief plan was drawn up, in the form of the project proposal, and from this the project was divided into four major milestones, and consequently broken down into cycles. These cycles represented sprints used in the Agile methodology, however differed slightly as following each sprint there would not be an individual deliverable. The idea of sprints in this instance was primarily to divide the project up, and understand what needed to be completed. The four milestones were as follows:

* Producing the database code
* Research on the topic of neural networks
* Producing the neural network code
* Producing the project report

Each milestone comprised several individual steps (detailed in this report), and keeping track of these involved utilising the Kanaban approach. For all tasks identified, it was recorded what had been completed, what was in progress, and fundamentally what was in the queue to be completed, and the order which steps needed to be executed.

This approach to managing the project aided greatly in producing the software on time, and further helped with keeping track of work that needed completion.

# Conclusion

# References

Barnett, T & Clarke, S (2005) “Combining player statistics to predict outcomes of tennis matches.” http://strategicgames.com.au/article2.pdf [accessed 5 September 2018]

Somboonphokkaphan, A & Phimoltares, S & Lursinsap, C (2009) “Tennis Winner Prediction based on Time Series History with Neural Modelling.” http://www.iaeng.org/publication/IMECS2009/IMECS2009\_pp127-132.pdf [accessed 5 September 2018]

Notes on PreparedStatement Object adapted from oracle docs: https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html

Perry, J (2018) “Create an artificial neural network using the Neuroph Java framework.”

https://www.ibm.com/developerworks/library/cc-artificial-neural-networks-neuroph-machine-learning/index.html [accessed 24 June 2018]

Algorithm depictions taken from Wikipedia:

https://en.wikipedia.org/wiki/Backpropagation

# Appendix

## Functional Requirements

|  |  |  |
| --- | --- | --- |
| **Number** | **Description** | **Priority** |
| 1.1 | The system must provide a secure connection to the relevant database | High |
| 1.2 | The system must create two tables within the database, one for the training data, and one for the test data | High |
| 1.3 | The system must read in the data from all the relevant CSV files, into the training table | High |
| 1.4 | The system should allow for a user to remove specified columns/rows, from the database table | Low |
| 1.5 | The system must remove entire rows containing bad data | High |
| 1.6 | The system must create appropriate output columns for the network to be trained on | High |
| 1.7 | The system must be able to normalise all numeric data in the training table using the min-max normalisation algorithm | High |
| 1.8 | The system must be able to standardise all non-numeric data to arbitrary normalised values (same scale as the numeric data) | High |
| 1.9 | The system must be able to transfer normalised test data from the training table to the test data table | High |
| 2.1 | The system must allow a user to create a neural network by specifying the number of neurons in each layer | High |
| 3.1 | The system must be able to train a neural network using backpropagation | High |
| 3.2 | The system must be able to calculate the mean square error of its predictions | Medium |
| 3.3 | The system must allow a user to set the number of training epochs to run | Medium |
| 3.4 | During training, the system should print the following information to the console: the current epoch number, and the current mean square error | Medium |
| 3.5 | During training, the system should be able to print the actual output of the network, and the expected output in a simple, readable format | Medium |
| 4.1 | The system should be able to save a trained neural network to a file | Low |
| 4.2 | The system should be able to load a previously saved neural network from a given file | Low |
| 5.1 | The system must be able to calculate the accuracy of the networks predictions (correct predictions / amount of inputs) | High |
| 5.2 | The system must be able to print the network accuracy of predictions on a previously unseen test data set | Medium |

## Non-functional Requirements

|  |  |  |
| --- | --- | --- |
| **Number** | **Description** | **Priority** |
| 1.1 | The system should display informative error messages when incorrect input is received (*Usability*) | Low |
| 2.1 | The amount of system crashes must be minimal (*Reliability*) | High |
| 2.2 | The system must be robust, demonstrating no noticeable decrease in performance over time (*Reliability*) | High |
| 2.3 | The system must close all resources taking advantage of the auto-closable interface (Reliability) | Medium |
| 3.1 | The system should be designed and implemented in a manner that allows for additional functionality to be added with minimal effort (*Modifiability*) | Medium |
| 4.1 | The system must be protected against SQL injections (*Security*) | High |
| 5.1 | The system must be coded to an efficient standard, and well documented to allow for any future maintenance to be carried out without unnecessary complications (*Maintainability*) | Medium |
| 6.1 | Where large amounts of data are involved, the system should take advantage of batch processing (*Efficiency*) | Low |
| 7.1 | The system should be written in Java, where possible making the most of object orientation (*Implementation*) | Medium |

## Software Instructions

The following contains step by step instructions for running the software.

1. Download the CSV files from the Git repository (https://git-teaching.cs.bham.ac.uk/mod-msc-proj-2017/ajk742.git) and save to a specified file, as “atp\_matches\_” followed by the year of data contained in the file “.csv”. for example, when saving the results from 2008, save as “atp\_matches\_2008.csv”
2. Within the Database package, locate the class ReadFiles, and the method readCSVTraining(). This must be updated to mirror the path to the downloaded CSV files. On line twenty-eight, edit the csvFile String to exactly match the location of the files (if files have been saved following the convention in number one, the String will only need to be changed between “/Users/” and “/training/”)
3. Ensure that the run configuration is set to Driver.Main (as this is where the main method which drives the program is located).
4. Create a PostgreSQL database called “data” and within the Database package locate the class ConnectDB and edit the following variables to match those from the database created: URL, USER and PASSWORD on lines fourteen, fifteen and sixteen respectively.
5. Ensure that all code within the main method is uncommented, and run the program.

## Project Proposal

*Using artificial neural networks in Java to predict the outcome of professional tennis matches.*

Author: Alexander Kelly

Student ID: 1868842

I will create and train a neural network, using historic tennis match result data, in order to build a model that will discover underlying hidden dependencies, and predict future match results with a degree of accuracy.

I intend to use Java for all aspects of the neural network as well as connection to any databases or files, and SQL to query the data.

Plan:

Firstly, I will obtain the relevant match result information in the form of a CSV file. I will separate these results *naturally* into a training data set, and test data set which will account for an approximate 90%-10% split respectively. To illustrate this, if all the results range from 2000 – 2017, the model will be trained on the data between 2000 – 2015 and tested on 2016 – 2017. Once the data has been acquired and separated, I will write the program to access it.

The next step will involve writing the code to create an artificial neural network that can be trained on the data. Training the data will consist of the following 5 steps on the entire training data set:

1. Randomly assign weights to the inputs
2. Normalise the outputs using an activation function
3. Compare the actual output with the expected output – this will give the error (difference between the two)
4. Back propagation by using the error to adjust the weights – this will be achieved using an appropriate formula to ensure that the adjustment is proportional to the size of the error
5. Repeat steps 2-4 for all inputs until the error is 0

Finally, I will review the neural network on the test data to ensure that the model meets the desired accuracy.

Time table:

(This may be revised once more is known about the project)

|  |  |
| --- | --- |
| Week | Task |
| 1 | Obtain match result data.  Sort into training and test data set.  Write program to access data. |
| 2 | Research techniques, options and best practises for creating neural network prior to writing any code. At the end of this week I endeavour to have a full understanding of which tools to use, and be ready to begin writing the code.  Note: research will be ongoing throughout the project, this week is reserved for gaining the required practical knowledge. |
| 3-6 | Create the neural network.  Begin writing dissertation. |
| 6 | Begin process of training the neural network. |
| 7-10 | Continue training the neural network systematically following the steps listed in the plan.  Continue writing dissertation (this will be ongoing until completion in week 14). |
| 11-13 | Once training is fully complete, the refined model shall be tested on the test data set. Any errors in the model will be corrected. |
| 13 | Make any final changes to code.  Ensure Javadoc and comments make code understandable.  Begin preparing the presentation. |
| 14 | Finalise presentation.  Finalise dissertation. |

Proposal Bibliography:

<https://www.udemy.com/neural-networks-from-scratch-in-java/>

<https://www.ibm.com/developerworks/library/cc-artificial-neural-networks-neuroph-machine-learning/index.html>