Assignment 3: Forecasting and Predicting
Liam Devlin - bwb18180

Forecasting describes the act of training a system on some known data, using this to build a model and then using this model to predict future values. The challenge for this assignment then, was to select a time-series dataset and through the use of a neural network, evaluate its ability to make predictions on unseen data.

Overview of the problem:

Neural Networks are systems capable of performing all manners of machine learning tasks, namely regression, classification, dimensionality reduction etc. A simple ANN (Artificial Neural Network) consists of an input layer, which is the data we feed to the neural network to make predictions, an output layer, which gives us the predictions, and a hidden layer. The hidden layer in reality usually consists of a number of layers of artificial 'neurons', which take a weighted input and produce an output based on an activation method.

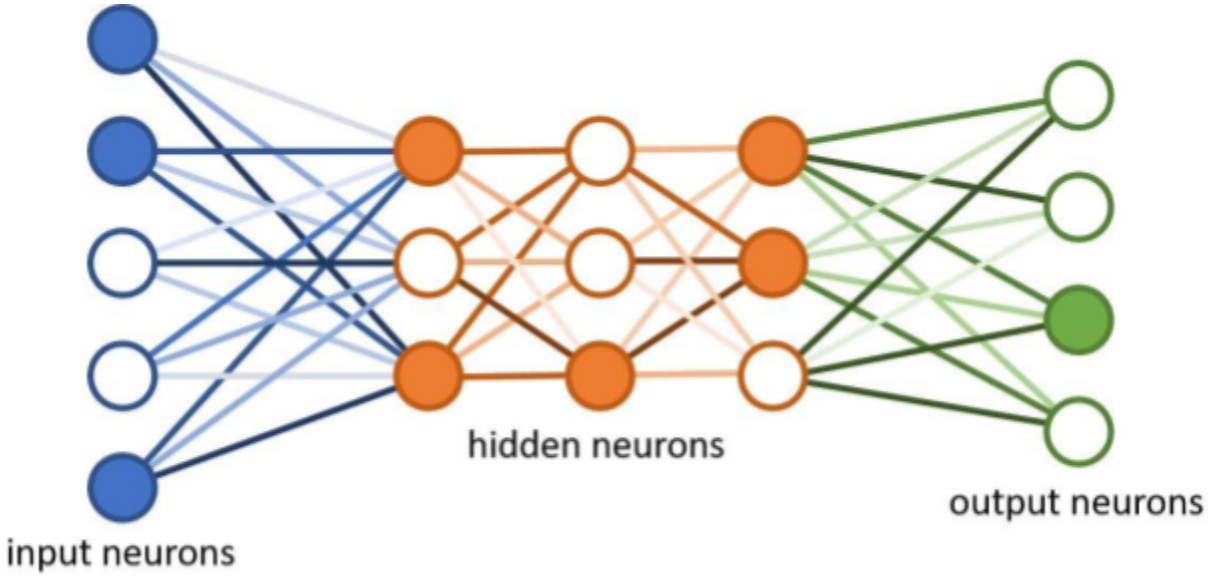

Figure 1.1 Feed Forward Neural Network Topology

Activation method described a neuron's behavior (in the hidden layer). There are many different types of activation function, linear neurons being the most simple where the output of the neuron is directly proportionate to the weighted-sum input. These step functions may still have some use but are not appropriate for non-linear problems, as stacking these neurons with n layers will still produce an output which is directly proportional to the input. Sigmoid activation functions solve this problem by introducing a sinusoidal shape to the activation function, meaning the output of this neuron is no longer linearly related to the weighted inputs. Finally, the most common activation function, ReLu, works in a similar way to linear activation methods, with one slight difference being that the output of the neuron is always 0 if the weighted sum input is less than 0, otherwise the ReLu outputs the weighted sum input.

This makes the output of a ReLu non-linear and therefore stackable. As well as this, the computational cost of a ReLu is relatively low compared to a sigmoid or tanh function, making it a great choice for the early stages of Neural Network development.

Generally, a neural network is composed of its input and output layers, as well as multiple hidden layers of a single neuron type. Multiple types of neuron *can* be used even within the same layer, however it is usually considered poor practice, and often complicates the system needlessly.

The Data:
The dataset that was selected for this assignment looks at yearly suicide and homicide rates in Australia over the period 1915-2004. There are a total of 4 metrics we wish to predict using Neural Network(s); Firearm Homicides, Firearm Suicides, Non-firearm homicides and Non-firearm Suicides. The data was sampled yearly and deaths are recorded per 100,000 people.

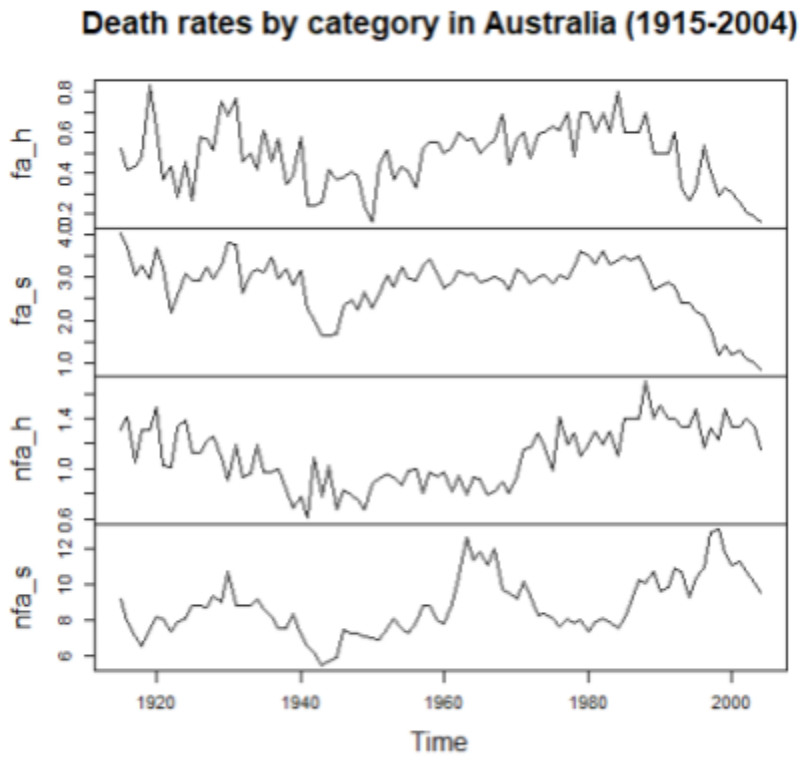

Figure 1.2 : Overall time-series plot

With a few significant exceptions, the overall trend between all the attributes are quite similar, with a small dip in the 1940s, followed by a gradual incline until we reach the 2000's, where both homicide and suicide fall slightly. In terms of prediction, the attributes the neural network(s) are likely to struggle with the most are the first two attributes (firearm homicide rate and firearm suicide rate) due in part to the high variance and lack of an easily definable trend.

Implementation:

To begin with, we must first load the data into R studio, and perform some basic data-cleaning to convert the values in each cell from strings to numeric values.

```
# Load in the dataset
aus <- read.csv("aus_homicide_suicide.csv")  # read csv file
aus <- aus[1:90,]

# convert to workable format (why is csv being a wee bam)
datanames <- c('fa_h', 'fa_s', 'nfa_h', 'nfa_s')
titlenames <- c('Firearm Homicide', 'Firearm Suicide','Non-Firearm Homicide', 'Non-Firearm Suicide')

# Scaling needs to be changed to Min Max so we can rescale it after the NN
# Smoothing also applied here
years <- as.numeric(smooth(levels(aus[,1])))[1:90]

fa_h <- scale(as.numeric(smooth(aus[,2])))
fa_s <- scale(as.numeric(smooth(aus[,3])))
nfa_h <- scale(as.numeric(smooth(aus[,4])))
nfa_s <- scale(as.numeric(smooth(aus[,5])))

init_data <- data.frame(as.numeric(aus[,2]), as.numeric(aus[,3]), as.numeric(aus[,4]), as.numeric(aus[,5]))
names(init_data) <- paste(datanames)
init_data <- ts(init_data, start=c(1915), end=c(2004), frequency =1)
plot(init_data, main = "Death rates by category in Australia (1915-2004)")
```

Figure 2.1 : Loading in the data + processing

Here, we read the CSV file into a dataframe object ‘aus’, which we use as the basis for the creation of the other dataframes, the init_data which is used for the initial plot of the time series and the lagged dataframe. We also create separate vectors of the individual time-series, which we will use to construct the lag set (discussed further shortly), and as these vectors are what we will be training our Neural network with, we must first scale and smooth the data.

Scaling is performed as neural networks are extremely sensitive to variations in scale between attributes, and as such, the scales should be normalized within the range 0-1. If scaling is not performed, the output of the neural network is almost always identical regardless of input. Smoothing helps to generalise the data slightly and reduces the impact of outliers on the overall time-series, which also generally helped improve the accuracy of the model on both training and testing. The next step is to construct the aforementioned lagged data-set.

```
# Create a lagged data frame for all the variables
LAG_1 <- 1
LAG_2 <- 2

lagged_data <- data.frame(years,fa_h,fa_s,nfa_h,nfa_s,
                  fa_h_l1=Lag(fa_h, LAG_1), fa_s_l1 = Lag(fa_s, LAG_1), nfa_h_l1 = Lag(nfa_h, LAG_1), nfa_s_l1 = Lag(nfa_s, LAG_1),
                  fa_h_l2=Lag(fa_h, LAG_2), fa_s_l2 = Lag(fa_s, LAG_2), nfa_h_l2 = Lag(nfa_h, LAG_2), nfa_s_l2 = Lag(nfa_s, LAG_2))

lagged_data <- lagged_data[complete.cases(lagged_data),]
lagged_data <- ts(lagged_data, start = c(1917), end=c(2004), frequency=1)

# create a test/train split
n_instances <- dim(lagged_data)[1]
trainIndex <- (n_instances / 10) * 8

train <- as.data.frame(lagged_data[1:trainIndex-1,])
test <- as.data.frame(lagged_data[trainIndex:n_instances,])
```

Figure 2.2 : Creating the lagged data set and performing test / train split

Before discussing how the lagged dataset was created, we must first describe why this process is applied to the data. For a given time series, a lag will push the values of each row down by a given index, 1 for example, meaning that each value in the lagged dataset would be 1 time-step behind the original time series. This is extremely useful as it allows us to feed information into the neural network about what happened at the last time step of the training data. For this dataset, two lags were created with lag values of 1 and 2, which were used as the input of the neural network with the idea that using previous values during training would result in a loss of some accuracy (especially during the training stage); However, the overall trend of the data would be far better captured, which is in stark contrast to using exact values during training which would lead to overfitting of the model.

The final dataset then is composed of 13 columns: the years of which these statistics pertain to, the firearm and non-firearm suicide and homicide rates for that year, as well as the two lagged sets of these statistics. Finally, a train and test split is performed before the training of the neural networks begins.

```
train_errors = list()
test_errors = list()

for (i in (1:4)){
  # what column are we currently working on?
  print(titlenames[i])

  # Train split
  train_y <- subset(train, select=c(i +1, i+5, i+9))
  names(train_y) <- c(datanames[i], 'a', 'b')
  train_years <- train[,1]

  # create column formula for NN
  f = as.formula(paste(datanames[i], " ~ a + b"))

  #..then create the NN
  current_nn <- neuralnet(f, data = as.data.frame(train_y), hidden=c(6, 4), threshold = 0.01, stepmax=9000000)

  # Calculate NN predictions
  nn_train_predictions <- compute(current_nn, train_y)
  nn_train_results = nn_train_predictions$net.result

  # Get Training Error
  nn_train_rmse <- rmse(train[,i+1], nn_train_results)
  train_errors <- c(train_errors, nn_train_rmse)

  # Moving Average
  train_ma <- rowMeans(train_y[c(datanames[i], 'a', 'b')], na.rm=TRUE)
  print(train_ma)

  # Plot what we got
  plot(train_years, train[,i+1], type='l', col=2, lwd=5.0, main=paste(titlenames[i],"Training Set"), xlab="Time", ylab="Value")
  lines(train_years, nn_train_results, col=3, lwd = 3.0)
  lines(train_years, train_ma, col=4, lwd=2.0)
  legend("bottomleft", c("Recorded Values", "NN Predicted Values", "Moving Average"), cex=1.0, fill=2:4)

  print("Training Complete")
```

Figure 2.3 : Training stage

Training the Neural Network

Firstly, the training and testing are performed in a loop as we are dealing with 4 different statistics to predict and this allows us to repeat the same method of prediction for each attribute easily. Also note that a list is created to store the error of each attribute's neural network at both the training and test stage. The first step of training is to create a subset of the overall data which holds only the attribute we are currently predicting, as well as its lagged value columns.

The lag columns as previously stated will be used as the input for our neural network with the recorded value of a given time-step is the output we wish to predict, for the neural network to understand this, we must feed it a formula, which dictates how the data is processed. For example, the formula in figure 2.3 would produce a formula such as “fa_h ~ a + b”, where fa_h is the rate of firearm homicides, with a and b being the respective lag columns. This formula tells the neural network that we are trying to predict the value fa_h by combining (in some way) the variables a and b. With the subset and formula prepared, we are now ready to create a neural network, we do so by creating a neuralnet object from the neuralnet package, passing through our formula, training data as well as three variables, hidden, threshold and stepmax. Hidden refers to the number of hidden layers in the neural network and can accept single value (e.g. a single layer of neurons) or a vector of numbers describing the number of neurons in each layer.

Threshold is an argument that helps define the termination point for a neural network, if the error delta over a certain number of generations is less than or equal to the threshold, the neural network will stop taking steps and the neural network will finish training, else it keeps going. Stepmax is exactly what it sounds like and dictates the maximum number of steps a neural network can take. After creating the neural network, we compute the training predictions of the neural network and calculate the RMSE and moving average. The moving average is found by taking the the recorded value as well as all lagged values for the current datapoint in the set, which results in a more generalised line of how the data is moving.

Finally we plot the recorded training results alongside the neural network predictions and the moving average.

Testing the Neural Network

```
# Test split
test_y <- subset(test, select=c(i +1, i+5, i+9))
names(test_y) <- c(datanames[i], 'a', 'b')
test_years <- test[,1]

nn_test_predictions <- compute(current_nn, test_y)
nn_test_results = nn_test_predictions$net.result

# Test RMSE
nn_test_rmse <- rmse(test[,i+1], nn_test_results)

# Moving Average
test_ma <- rowMeans(test_y[c(datanames[i], 'a', 'b')], na.rm=TRUE)

plot(test_years, test[,i+1], type='l', col=2, lwd=5.0, main=paste(titlenames[i],"Test Set"), xlab="Time", ylab="Value")
lines(test_years, nn_test_results, col=3, lwd = 3.0)
lines(test_years, test_ma, col=4, lwd=2.0)
legend("bottomleft", c("Recorded Values", "NN Predicted Values", "Moving Average"), cex=1.0, fill=2:4)

# Holt Winters
current_hw <- HoltWinters(train_y[1], beta=FALSE, gamma=FALSE)
hw_test_results <- predict(current_hw, n.ahead = (length(test_years) + 1), prediction.interval = T, level = 0.95)
plot(current_hw, hw_test_results, main = paste(titlenames[i], "Holt Winter Forecast"), xlab="Time", ylab="Value")
lines(test[,1], test_ma, col=4, lwd=2.0)

print("Testing Complete")
test_errors <- c(test_errors, nn_test_rmse)
```

Figure 2.4 : Testing stage

We again start by taking a subset of the test data which contains only the attribute (and associated lag columns) we are looking to predict. We also create a variable to hold the years associated with the test set data to make plotting slightly easier later.

We next collect the predictions for the neural network and calculate the RMSE with recorded test set, here the moving average is also calculated. The data is then plotted in an identical fashion to the training set.

Finally, a Holt-Winters model is created to compare with the results of the neural network. Creating the model is extremely simple, we simply pass through the training data and two arguments, beta and gamma. These two arguments indicate to the holt-winters algorithm the presence of seasonality and trend, which our dataset does not have as it was sampled yearly.

To make predictions with our Holt-Winters algorithm, we do not even need to pass through the test data, we simply ask it to produce a set number of predictions and the algorithm does it's best to predict the next values. With this in mind, we set the number of future predictions to be made by the holt-winters algorithm to be the same as the size of the test-set, as to make comparison simple. Finally, we plot the holt-winters data on a separate plot.

The very final stage of the program is to plot the neural network error of the train and test set for each attribute of the dataset.

```
# Error plot
error_frame <- do.call(rbind, Map(data.frame, A=train_errors, B=test_errors))
colnames(error_frame) <- paste(c("Train", "Test"))
rownames(error_frame) <- paste(datanames)
barplot(t(as.matrix(error_frame)), main="Error of Neural Network By Data Column",
        xlab = "Death Category", ylab = "NN RMSE", beside=TRUE, legend=colnames(error_frame))
```

Figure 2.5 : Plotting the error

As we have already collected the errors in two vectors, we are easily able to combine these into a dataframe for easy plotting. We simply add the column and row names, and use the barplot function to effectively visualise the difference in error between the training and test stage.

Results

For each attribute of the dataset, the parameters of the Neural Networks created was kept identical, Stepmax was always kept at a very high value of 9,000,000 to give the algorithm the best possible chance of convergence, no matter how long it takes. Threshold was also kept at 0.01 for each neural network as reducing below 0.01 did not lead to an appreciable improvement to the accuracy of the system and did in fact lead to drastic slow down in convergence time.

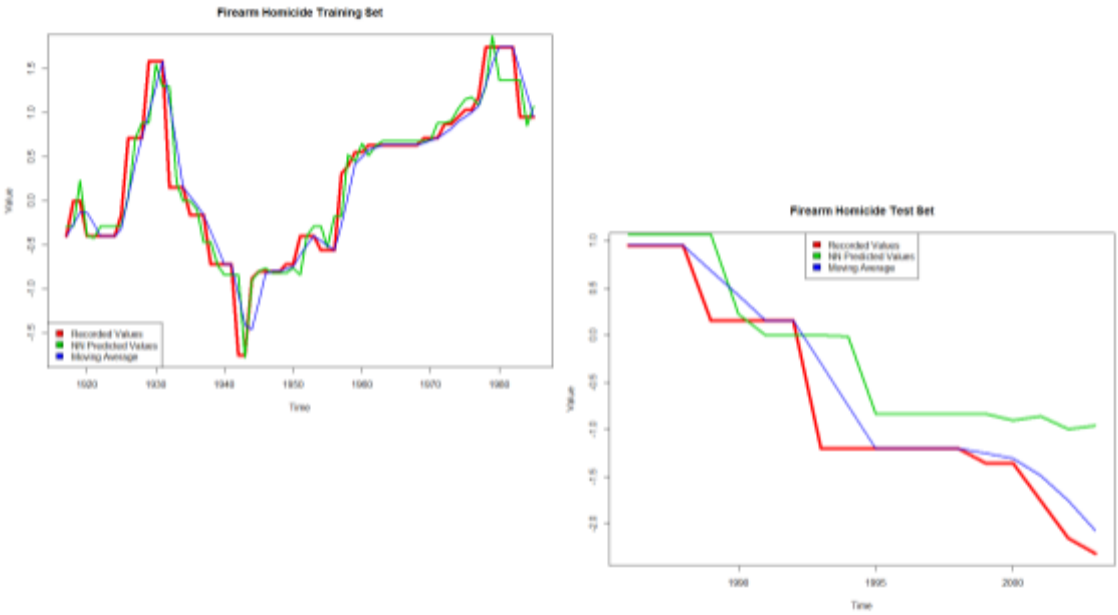

Figure 3.1 - Firearm Homicide Train and Test set results

For the firearm-homicides attribute, the neural network performed quite well, and captured the trend of the data well in both the training and test set. As the test set progresses, the accuracy of the model begins to fall apart as we see a decline in the recorded values as well as the moving average, where the neural network begins to flat line.

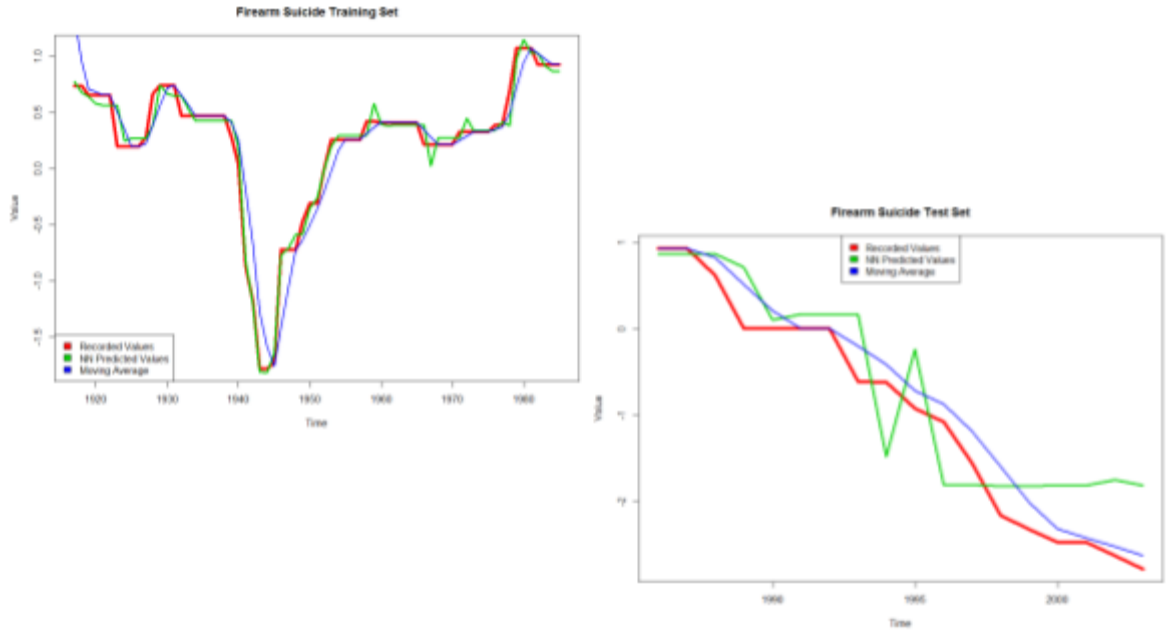

Figure 3.2 : Firearm Homicide and Suicide Train and Test set results

The firearm suicide results are interesting as, again, the neural network captures the training trend quite well (with the exception of the small spike in the mid 50's) however the test set is slightly less well fitted. The large spike in the middle of the graph is slightly puzzling as neither the recorded values or the moving average exhibit any values which would reasonably lead to this kind of behaviour. This is likely due to overfitting the model on the training data, as it does appear to have many sharp ascensions and descents during the training period.

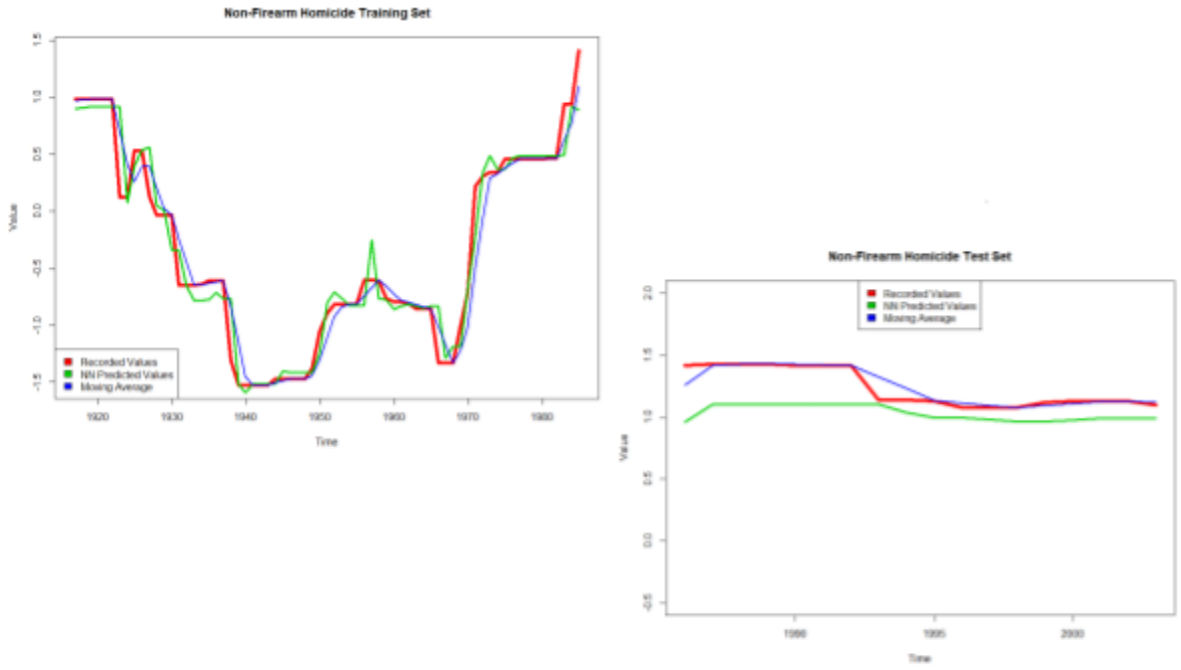

Figure 3.3 : Non-Firearm Homicides Train and Test set results

The non-firearm homicide attribute was the best performer from the data set and this is evidenced by the generally accurate test results. The neural network captured the trend and did not introduce any spikes or unexpected jumps, which is extremely surprising given the poor quantity of training data, as well as the volatility of the training set being vastly different to that of the test set.

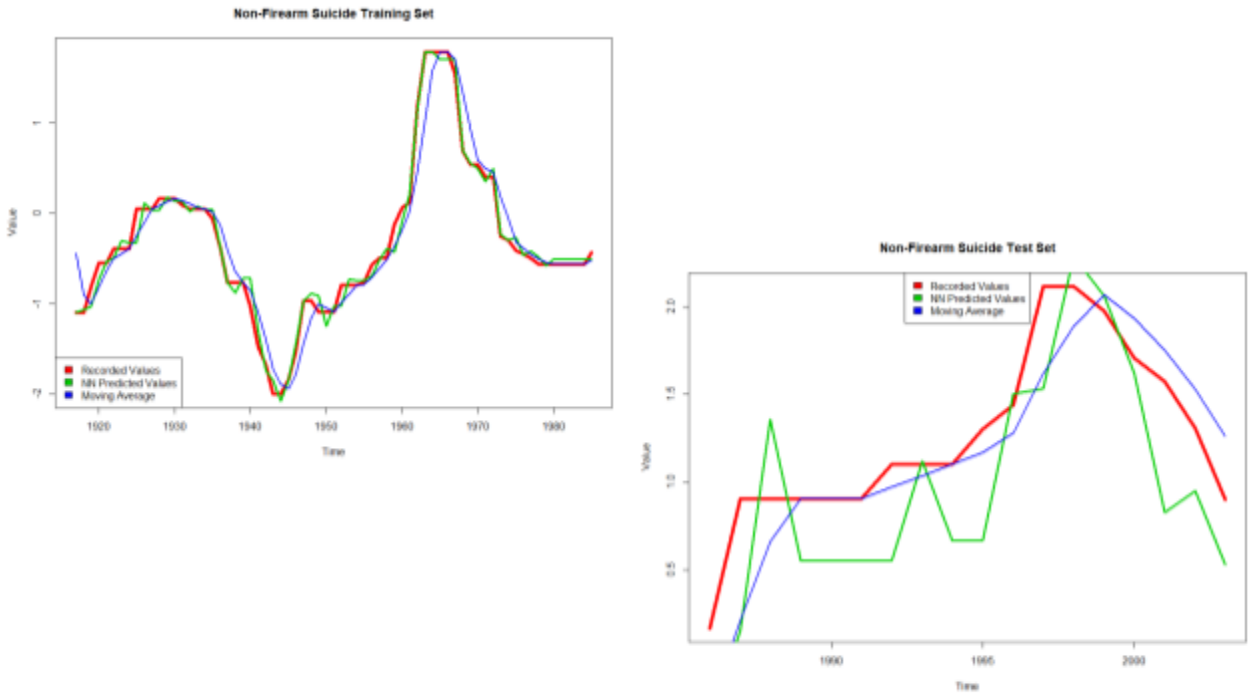

Figure 3.4 : Non-Firearm Suicide Train and Test set results

The final attribute performed decently with the general trend being mostly captured in the test set, and fully captured in the training set. With this being said, the test shows some abnormal spike in areas which the recorded and moving average values did not. Again, this is likely due to the low number of samples and generally an overfitting model.

As can be seen from figure 3.5, the errors of the neural networks are quite low, and this is what we expected to see given that, generally, the models were able to capture the way the time-series moved. The worst performing neural network was the predictor for firearm homicides, and as previously mentioned, as the test set progresses, the accuracy suffers likely due to an overfitting of the highly volatile training data.

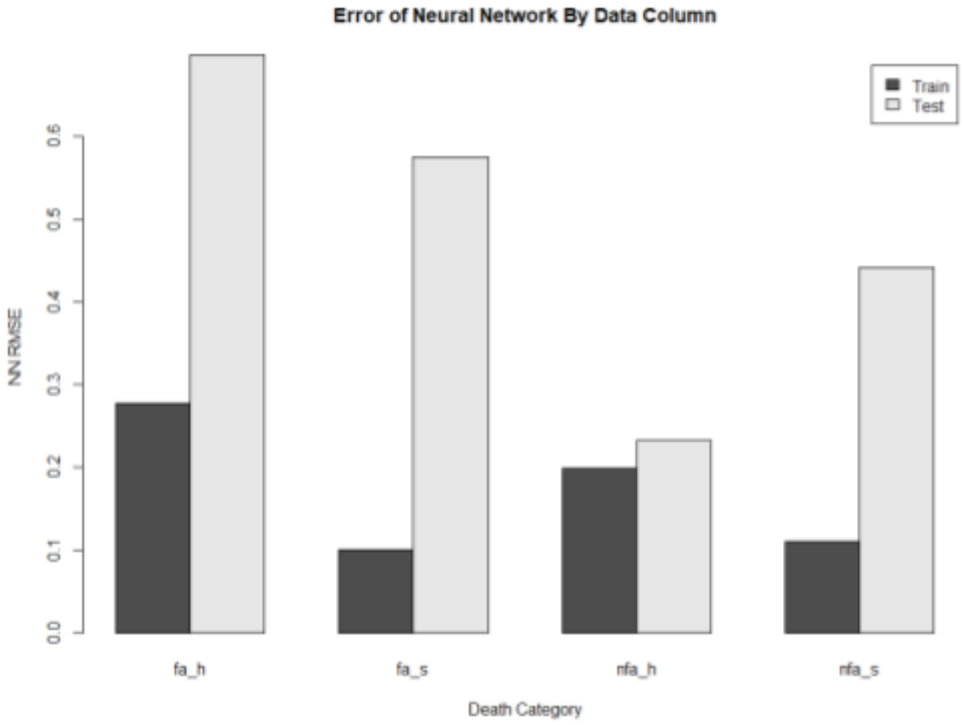

Figure 3.5 : RMSE error by data-set attribute

To compare the results of the neural network(s) further, each attribute was also predicted using a Holt-Winters algorithm, a well-known standard in the financial sector.

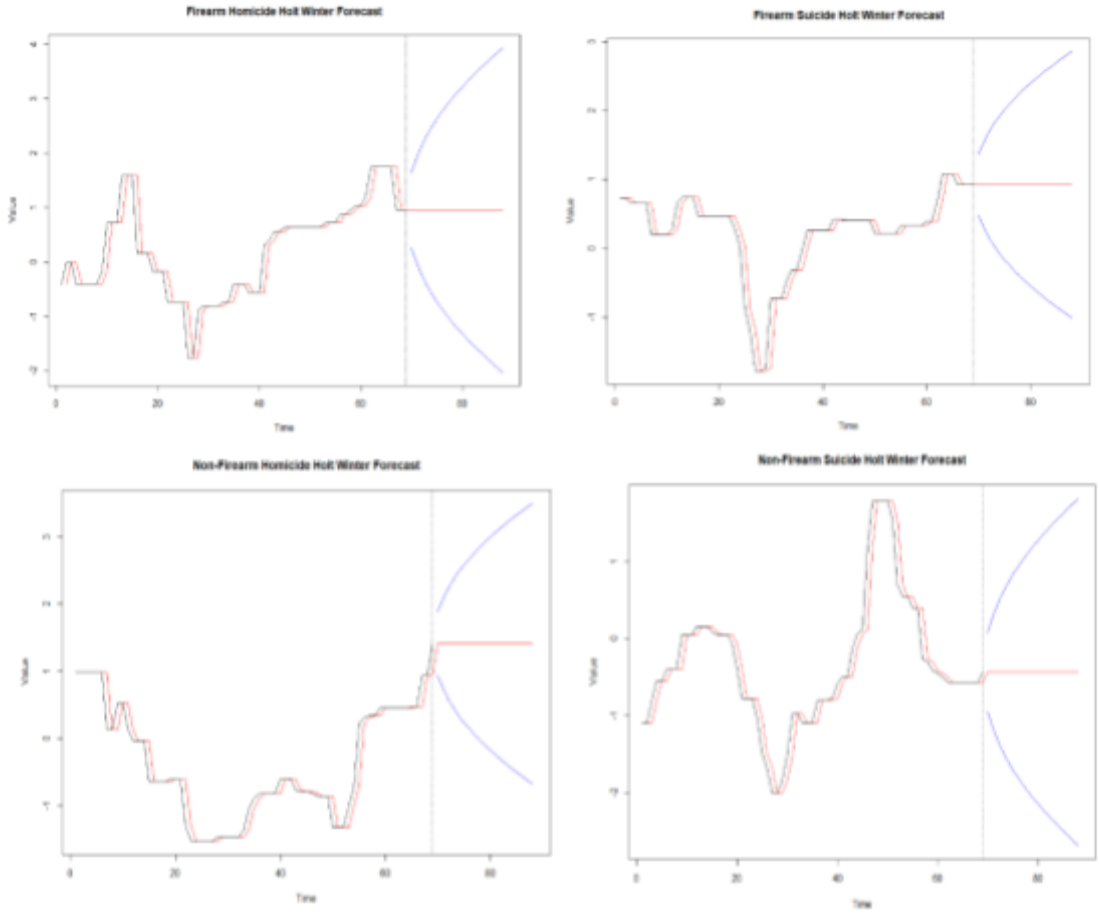

Figure 3.6 : Holt-Winters predictions

Unfortunately, the holt-winters algorithms were not able to make any meaningful predictions, with a massive spread for predictions which flatline immediately after the training data. One of the reasons this may have occurred is due to the lack of seasonality and trend in the data, as previously mentioned. Holt-Winters algorithms use these values to aid the accuracy of these predictions, and there absence was surely felt. Again, the lack of instances was more than likely a significant factor which contributed to the poor performance.