MA5832 - Capstone Report

Sean O'Rourke (13984624)

Due 14 October 2021

A note on location of r scripts used for this report

- Less important, or superseded plots, code chunks and outputs are located in the appendices.
- An independent r script was used to perform the data wrangling and feature engineering steps and has been attached in appendix 2. Only relevant chunks have been included in this markdown document.
- An independent r script was used to generate the Neural Network models and has been attached in appendix 3.
- Appendix 4 contains the full r code to create this r mark down. The MARS model was developed within this markdown document, thus the full code for this model can be found within the markdown code

Abstract

This investigation aimed to use a single layer machine layer and dense neural network to predict unemployment rates in Australia using a set of 7 numerical predictors provided by the Australian Bureau of Statistics. To deal with the time series nature of the data set the predictors were lagged and summaried over various windows. These engineered features were then used to fit a MARS regression spline and a dense neural network. Interestingly the MARS model performed well in the short term however it failed to capture any of the trend around the time the corona virus pandemic began. While the absolute values found by the Neural Network were not that accurate the model predicted the trends observed in the test period well, including the pandemic conditions.

Introduction

The aim of this investigation is to predict the unemployment rate in Australia for the period March 2018 to September 2020 using the supplied "AUS_Data.xlsx" data set. The data set contains a selection of economic metric collected from June 1981 through to September 2020. The specifications of the task require the fitting of a single layer machine learning algorithm and a Neural Network to the data prior to March 2018 and then, using the predictors from March 2018 to September 2020, predict the unemployment rate for this period. The provided data set contains the following variables:

- Y : Unemployment Rate (%) (Response)
- X1 : Change in GDP (%) (Predictor)
- X2 : Change in Government final consumption expenditure (%) (Predictor)
- X3 : Percentage change in final consumption expenditure all industries (%) (Predictor)
- X4 : Term of Trade Index (%) (Predictor)
- X5 : Consumer Price Index (all groups) (Predictor)
- X6 : Number of Job vacancies (1000's) (Predictor)
- X7 : Population (estimated) (1000's) (Predictor)

Firstly it is important to understand the meaning of unemployment which can be considered as the percentage of people who are available and willing to work but currently without work (OECD, 2020). While there are many issues around unemployment and participation (making ones self willing and able to work) they are beyond the scope of this investigation. The predictors in the supplied data set consists entirely of quantitative continuous measures compiled by the Australian Bureau of Statistics. It is beyond the scope of this investigation to research the soundness of the provided predictors or extend the data set to included other predictors. It is however, important to understand some of the potential short comings of the supplied data so as they are not exacerbated in the modelling.
Firstly there are many qualitative measures of the economic conditions such as the change in expectation of long term unemployment (Claveria, 2019) which could be found to be important. There are also countless examples of economic forces such as stock market crashes, natural disasters and change in governments and their policies which is also not directly captured in this data set that may influence the unemployment rate. An example of this is the oil price shocks in the 1980's which is recognised as one of the greatest drivers of the spike in unemployment over the late 80's (Karanassou and Sala 2009). Further, it was also found that the the increase in terms of trade correlated well with the subsequent downward trend in unemployment(Karanassou and Sala 2009). A notably relevant example of a similar shock is the COVID-19 pandemic which falls in the targeted prediction period. While this brief review of the literature is by no mean exhaustive it does highlight the broad bases of factors that are commonly considered when modelling unemployment relative to the small number of provided predictors. This highlights the need to generate models which generalize well and do not learn underlying patterns that are not represented by the predictors. This will be particularity important to remember as the data set is relatively small and thus will be prone to over fitting (Geron, 2019). Unemployment is considered a negative influence on the economy as a whole, not just financially but from a well being point of view of individuals and firms. Sustained levels of unemployment also lead to loss economic advancement which can not be recovered thus it is important for policy makers to be able to preempt downturns in the employment market and implement counter measures before an actual loss is incurred (Simpson S 2020). This requirement motivates the need for acurate modelling of unemployment.
To gain an understanding of the provided data some exploratory data exploration was undertaken. Basic inspection can be found in Appendix 1, importantly it can be seen that X6 and X7 contain missing values which will need to be addressed. To gain an initial understanding of the data set and potential correlations between the predictors and response a time series plot is shown in Figure 1. Importantly it can be seen that while some predictors correlate directly with unemployment (either positively or negatively), many don't, however it can be seen that it appears for some that unemployment is high when variance is high and low when the measure is stable. Further the need to normalize the population, job vacancy and CPI predictors which simply continue on an upward trend is apparent.

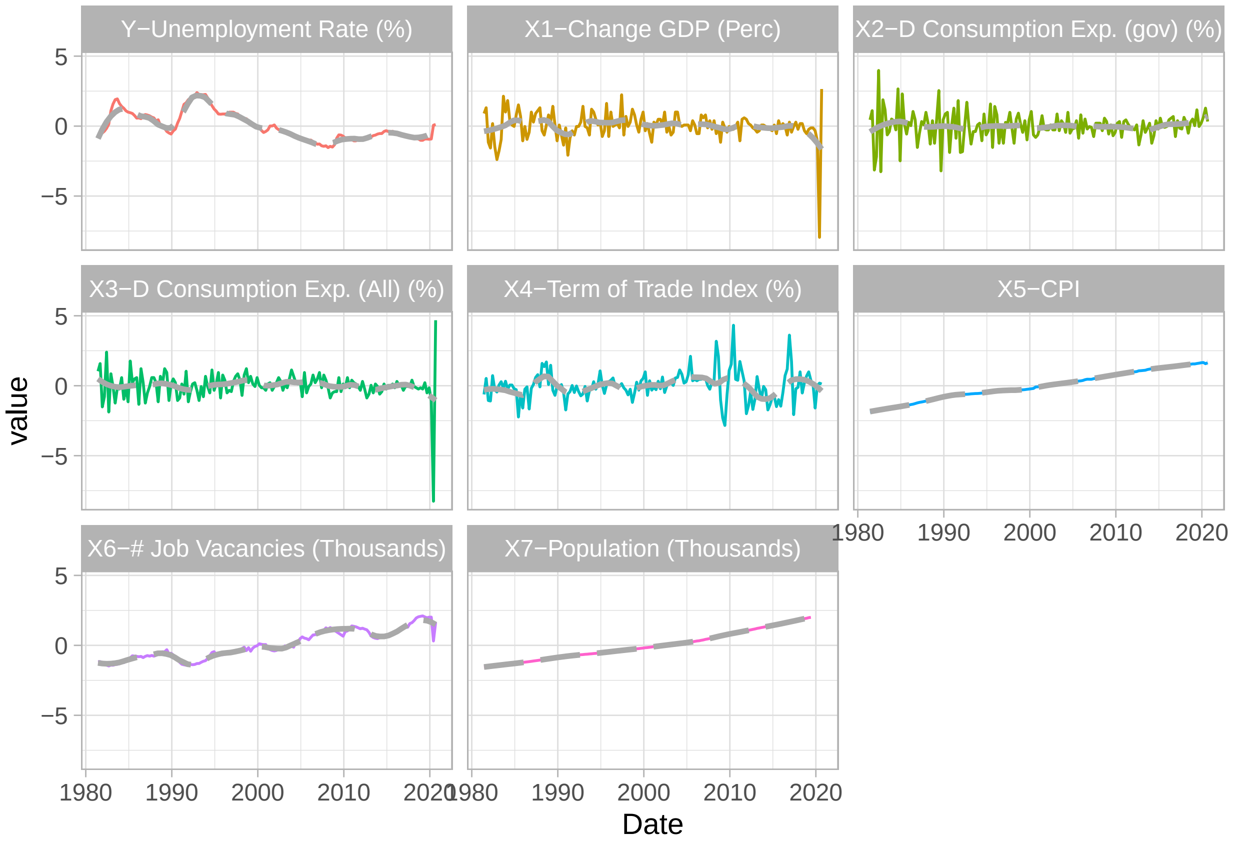

Figure 1: Time Series plots of the response and predictor variables

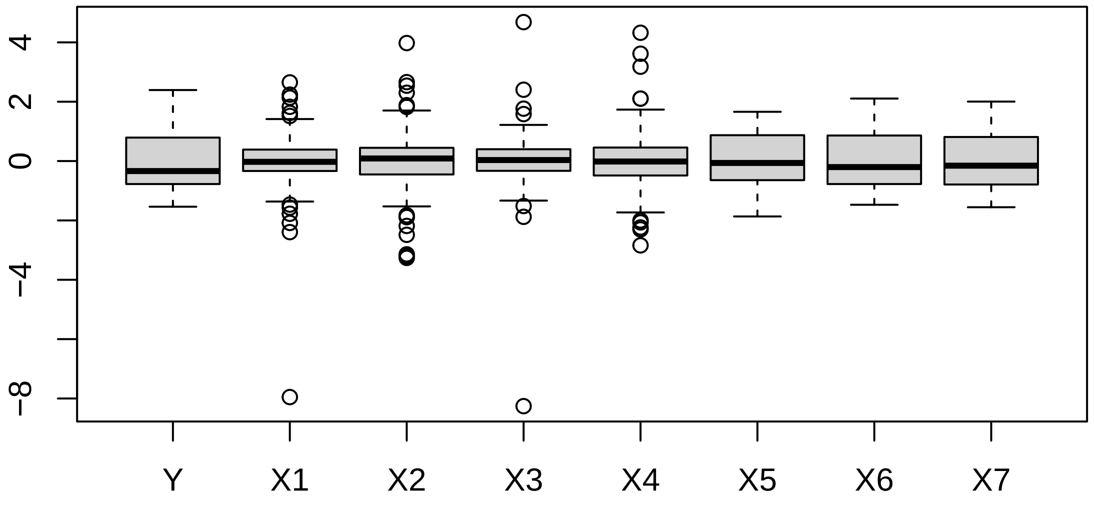

Figure 2: Boxplot of normalized variables

Data

The first step in any machine learning task it to prepare and wrangle the data into a use-able format. This investigation will undertake this in 2 steps, Firstly, simple tidying, imputation and normalization will be undertaken. Secondly, the time series nature of the data will be addressed to meet the needs of the planned algorithms. The first step involved the following tasks:

- Imputation of missing values,
- Correction for GST,
- Normalization of upward trending variables.

Imputation of missing values

It can be seen in Appendix 1 Figure 4 and Figure 5 that X6 is missing values, roughly 3/4 of the way through the series and X7 is missing values at the end of the series. Further it can be seen that X6 has no clear trend and X7 appears to be linear, especially in the second half of the series. For X6 `impute_AR1_Gaussian()` from imputeFin package (Liu & Palomar 2021) was used to estimate the values for X6 based on an auto regressive Gaussian model. The linear nature of X7 lent itself to a simple linear regression to impute the missing values. It was then assumed that these values were sound, while this is a bold assumption the data set is relatively small and thus the loss of 20 observations is not a realistic option. Where the oportunity arises through the investigation the possible side effects of this assumption will be investigated. The resulting plots of the variables can be seen in Figures 3 and 4 respectively.

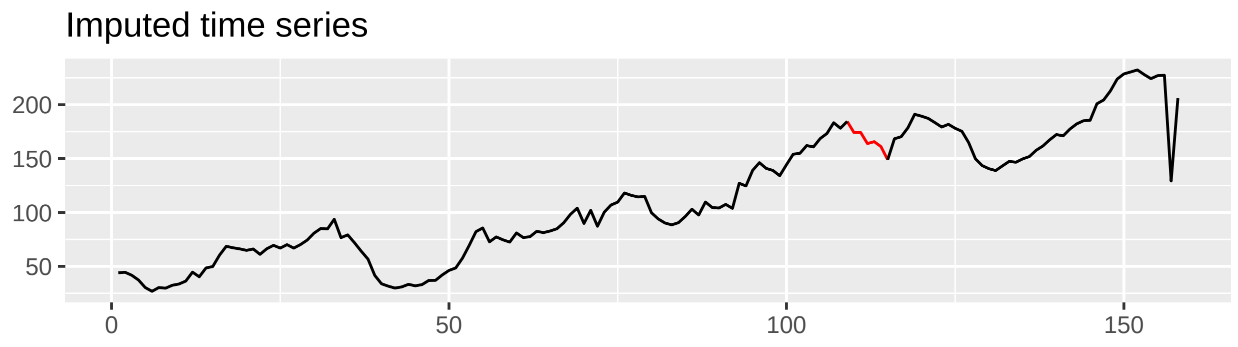

Figure 3: Imputation of missing job vacancy data

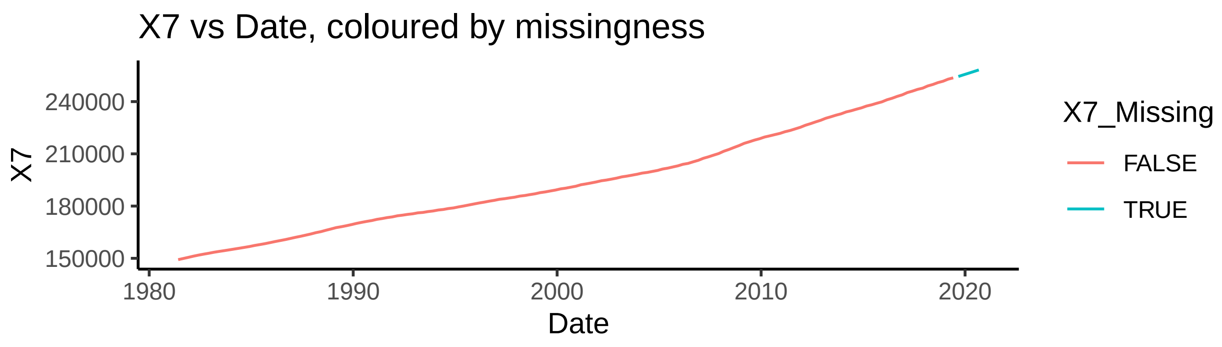

Figure 4: Imputation of missing population data

Correction for GST

In July 2000 a 10% tax on most goods and services was introduced. This can be seen as a step in X5 which should be removed as it does not correlate with the target variable and will reduce the predictive power of X5. While the absolute accuracy of this is debated in literature as at the same time several state taxes were removed it will be adopted in this investigation. To achieve this transformation all values prior to July 2000 were increased by 10% as shown in the below code chunk.

```r
rawData$T1 <- # create T1 variable % change period on period to X6
  insert(diff(rawData$X6), 1, values = NA) / rawData$X6 * 100
rawData <-
  rawData %>% mutate(T2 = X6 / X7 * 100) %>% # create T2 Variable X6 normalized for population growth
  mutate(T3 = ifelse(Date < as.Date("2000-07-01"), X5 * 1.1, X5)) # create T3 Correct CPI for GST
rawData$T4 <- # change X5 (CPI) to inflation, calculated quarterly but anualized
  insert(diff(rawData$X5), 1, values = NA) / rawData$X5 * 400
POP_TREND <- # create long term linear model of population growth
  lm(X7 ~ Date, data = rawData)
rawData$T5 <- # remove long term linear trend of population growth
  rawData$X7 - unname(predict.lm(POP_TREND, newdata = data.frame(Date = rawData$Date)))
```

Normalization of upward trending variables.

Figure 1, from the previous section, shows that the following variables all have an underlying, long term, upwards trend:

- X5 - Consumer Price Index (CPI),
- X6 - Jobs Vacancies (1000's),
- X7 - Population (1000's).

Two well recognized methods for the normalization of time series data with continual trends that only correlate with time passed are detrending and differencing (Shumway & Stoffer 2019). The first involves fitting an underlying model of the trend and subtracting this trend from the values and the second is simply calculating the period on period change of the variable (absolute or percentage). Each of the identified variables needs to addressed in a way which suits its context. The R code for implementing the changes outlined below is shown in the code chunk in the 'Correcting for GST' section above.

Firstly, CPI is the raw measure used to calculate inflation (ABS 2017) and inflation is the primary index used to normalize dollar values over time as well as a key metric in the overall condition of the economy (RBA 2021). Inflation is simply the annualized change in CPI. Since the calculation of inflation is simply a specific case of differencing it has been adopted to normalize X5. A small detail that should be noted that as the change is quarter on quarter and inflation is considered as a percentage change over a year the calculate difference needs to be multiplied by 4.

Secondly, the number of job vacancies, can be viewed to affect unemployment in two lights. Both of which have been prepared for further investigation. the affect could be seen as a function of the previous job vacancies (ie it would be harder to find a job as time goes on if vacancies are falling month on month) or as a function of the population (ie it is harder to find a job when you are 1 of 10,000 people compared to one of 100). The later makes an assumption that the participation rate discussed in the previous section correlates with population. These are both examples of differencing and detrending respectively and have both been implemented.

Finally, as previously discussed and seen in figure 1, Population has a steady long term upward trend that can be accurately modeled with a linear regression. The predicted results from this long term trend were then subtracted. This amounts to proposing that it would be harder or easier to gain a job in a period when the population growth is above or below the long term trend. This is a good example of detrending and thus has been implemented. The quality of the linear model fitted was assessed and found to be statistically significant as can be seen in the below code output. While it is confounding, the quality of this fit justifies the use of linear regression to impute the missing values for X7 in the previous section.

```
##
## Call:
## lm(formula = X7 ~ Date, data = rawData)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -6706.7 -3834.2   481.9  2758.6  8988.2
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1.139e+05  9.676e+02  117.74   <2e-16 ***
## Date        7.309e+00  8.010e-02   91.24   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4193 on 156 degrees of freedom
## Multiple R-squared:  0.9816, Adjusted R-squared:  0.9815
## F-statistic:  8325 on 1 and 156 DF,  p-value: < 2.2e-16
```

Time Series Data

The time series nature of the data will now be considered. The use of time series data to generate predictive models is not an uncommon machine learning task with examples including predicting stock market trends, modeling infection in communities, drawing correlation between treatment and outcomes in a medical domain and modelling and forecasting economic conditions all regularly use data sets which have an implied value in their sequence. The fundamental difference when using time series data to prepare a model is that adjacent observations affect the observation in question, that is each observation is not an independent sample (Shumway and Stoffer 2017).
The failure of this assumption makes a majority of single layer machine learning algorithms inadequate for use. Further, while the dense neural network to be considered in this investigation is capable of handling time series data, it is usually best practice to use recurrent neural networks (RNN) or convoluted neural networks (CNN) for time series modeling projects (Geron 2019). The training, validation and testing of models which capture the sequential value of the predictors is also more complicated as each test, training and validation split must be made such that this is taken into account. This is achieved through taking splits similar to that is specified for the test training in this investigation (ie train on before March 18, test on after). While this is not problematic if one was to make splits similar to 4 fold cross validation one would have to accept a reduction in the size of both the training and validation set to avoid repeatedly testing on the same data. The reduction is inversely proportionate, that is a 25% reduction in the training set leads to a 75% reduction in the test set. With such a small data set this would be hard to accommodate thus other methods will be investigated.
To address the failure of the independence assumption and use regular machine learning algorithms two procedures are commonly used, the frequency domain approach and lagged relationships (Shumway and Stoffer 2017). This report will focus on engineering a feature set of lagged relationships. To prepare the data in this way the following was undertaken:

- Inspect the data for seasonality,
- Visual inspection for lagged correlations with raw and simply transformed predictors,
- Visual inspection for lagged correlations with statistical measures of predictors (calculated over a given window)
- Engineering of features for use in model development
- Assessment of engineered features.

To asses if a seasonal pattern was evident the change in unemployment between periods was plotted against the seasons. As can be seen in figure 5 no clear correlation is present and thus it is assumed the data is non seasonal which validates the descion to not investigate frequency domains as predictors.

To engineer features the following was considered

- correlation (C) between Y and X at any given time (t) in the past,
- correlation (C) between Y and U at any given time (t) in the past, where u is the rolling mean of X over window w,
- correlation (C) between Y and S at any given time (t) in the past, where S is the rolling standard deviation of X over window w,
- If C was statistically significant,
- at what t was the first local maxima of C.

To achieve this efficiently an r script was prepared to iterate over each predictor, asses if it has a statistical significant correlation, at any lag, to Y and then return the lag for the first local maxima of correlation. The script then build new features by iterating over each feature and calculating the rolling mean and standard deviation for all possible time windows w. These features where then assessed as per the original features for statistically significant correlation at all possible lags and if significant return the first local maxima of correlation was returned.

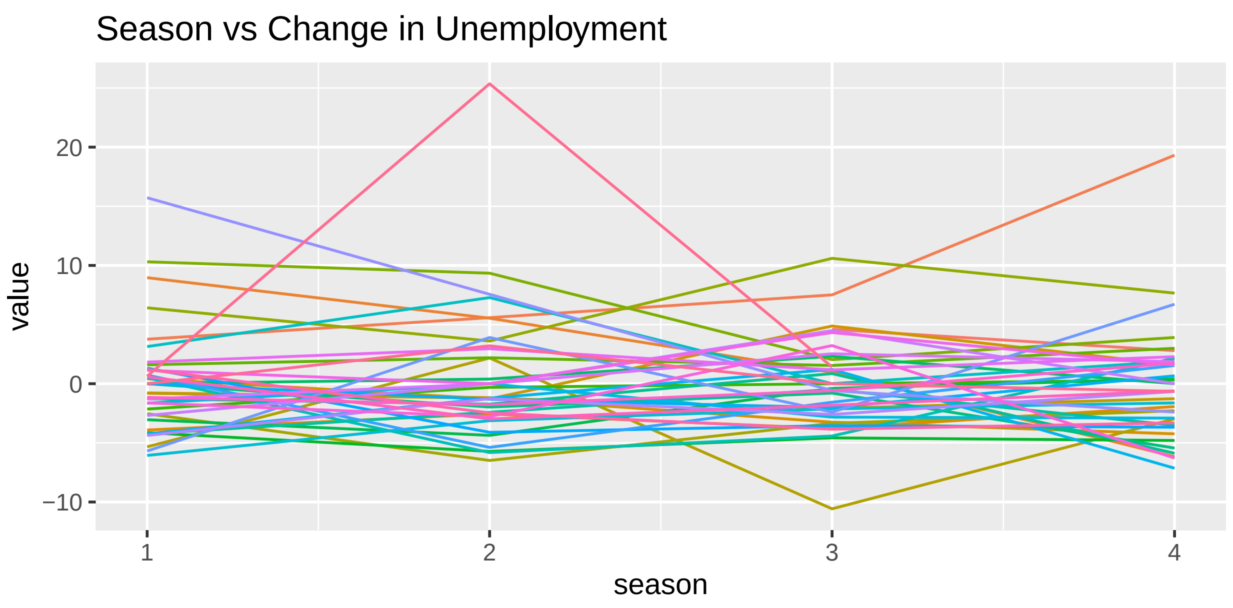

Figure 5: Plot to assess seasonality of change in unemployment

A key issue with the rolling windows used was that they reduced the size of the training set. To optimize this the number of observations removed by the window was plotted against the correlation with Y , Figure 6, where it can be observed that the predictor space gains 16 observations back for the loss of only 3 predictors if the number of lost observations is capped at 20. Unfortunately these a engineered features that correlate well with Y. The final step in the feature engineering process was to remove any poor predictors which shared a high correlation with other features or low statistical signifigance. This was performed with the below code block.

```
dataMatrix <- as.matrix(modelData[ , -1]) # convert to matrix, excluding date

pCor <- rcorr(dataMatrix, type = "pearson") # return pearson correlation and significance levels
pCor_matrix <- pCor$r # correlation matrix
pCor_sig <- pCor$P # significance levels

toRemove <-
  foreach(m = 2:ncol(pCor_matrix),  .combine = c) %do% {
    #remove junk (defined as correlation with another feature > 0.95
    #or significance of correlation with Y < .05)
  foreach(n = 2:nrow(pCor_matrix), .combine = c) %do% {
    # iterate over all rows and columns
    if (n != m){ # if not on self
      if(pCor_matrix[n,m] > 0.95 | pCor_sig[1,m] > 0.05){ # test for "junk"
        colName <- rownames(pCor_matrix)[m] #find variable name
        colName
      }
    }
  }
}
```

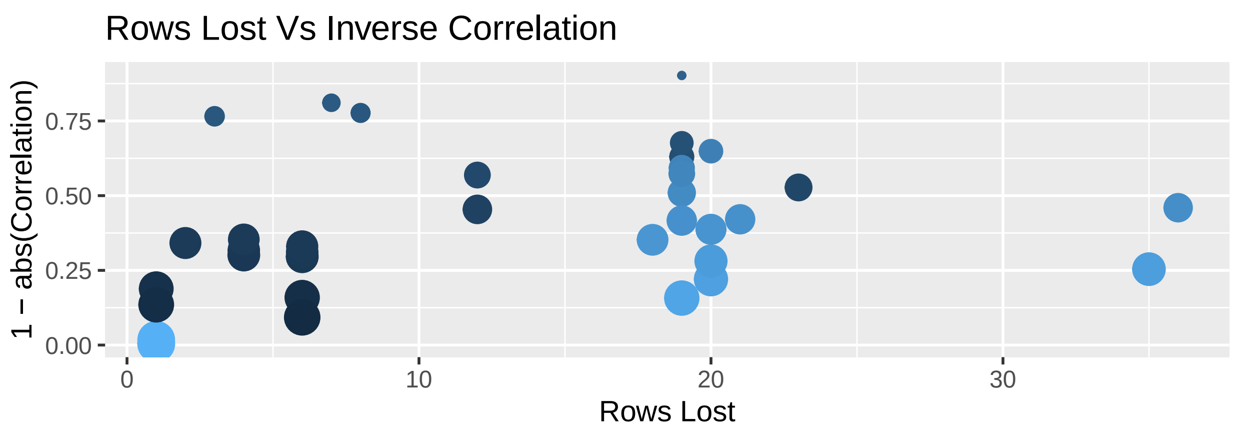

Figure 6: Assessing value of each engineered feature

Feature Normalisation and Test Training split

Since the normalization process and test and training splits is common to both models the process under taken to peform these tasks will be covered in this section.Firstly the test training split was performed with a simple filter as the specifications of the investigation provided the details for this. The mean and standard deviation for each feature over the training set only was then calculated. the `scale()` function from base r (r core team 2013) was then used to scale and center both the test and training sets prior to being exported for use in other scripts. The r code for this task is shown in the below code chunk. The full r code for the above feature engineering process can be found in Apendix 2.

```
## Build test training split, normalize data.
testData <-
  modelData[modelData$Date > as.Date("2018-02-28"), c(-1)] # test Data all observations after 28 Feb 20
trainData <- modelData[modelData$Date <= as.Date("2018-02-28"), c(-1)] # train Data all other obs
results <- modelData[modelData$Date > as.Date("2018-02-28"), c(1,2)] # test response and date (for plot
trainAss <- modelData[modelData$Date <= as.Date("2018-02-28"), c(1,2)] # training response and date

mean <- apply(trainData[ , -1], 2, mean) # calculate mean for each variable
std <- apply(trainData[ , -1], 2, sd) # calculate SD for each variable

testPred = scale(testData[ ,-1], center = mean, scale = std) # scale test data and make predictor df
testResponse = testData[,1] # make response list

trainPred = scale(trainData[ ,-1], center = mean, scale = std) # scale training data and make pred df
trainResponse = trainData[, 1] # make train response df

## check data splits make sense
nrow(modelData)
nrow(trainPred)
length(trainResponse)
nrow(testPred)
length(testResponse)
## Save objects for use in various models
saveRDS(list(rawData, modelData, testData, trainData, results, trainAss, testPred,
             testResponse, trainPred, trainResponse, mean, std), "data.RDS")
```

Machine Learning

Algorithm Selection

The first task encountered in building the single layer machine learning model was selecting the algorithm. The following were key features of the engineered feature space, developed in the previous section, that were considered influential while selecting the algorithm:

- The space is wide. ie. a large number of predictors and few observations (by machine learning standards),
- The importance, hierarchy and interaction between predictors is unknown,
- The feature space is likely missing many predictors,
- All predictors and the response are numeric (pure regression),
- The shape of the response variable is random with many sharp changes in direction and is not recognizable as any basic polynomial.

A Multivariate Adaptive Regression Spline (MARS) was selected for this task as it is generally considered to perform well in situations where complex and / or non linear responses to the predictors are present (Nisbet, Miner & Yale 2018). The strong points of this algorithm that align well with this project are as follows:

- Automatic feature selection and ranking,
- Automatic detection of feature interaction,
- No issues with entrapment in local minima,
- Good ability to fit sharp local structures,
- Computationally cheap.

The MARS algorithm is however susceptible to over fitting due to its large degree of flexibility. (Nisbet, Miner & Yale 2018)

Hyper-parameter Selection

With the algorithm and data prepared the model then needed to be trained before assessing performance on the training data and eventually using the model to make predictions on the test data set. To train the model the caret package (Kuhan 2008) was used and the hyper-parameters associated with the MARS algorithm is shown in the below code block. A 2 step training process was undertaken with an initial broad brushed approach to hyper-parameter tuning and an uninformed approach to validation and measures of accuracy. The second code block shows the `train()` (Kuhan 2008) used for the initial training run as well as a summary of the resulting model. Figures 7 and 8 show the models performance over the tuning grid and the best models performance across all validation folds. This information will be debriefed in the next section while selecting the parameters for the final tune.

```r
modelLookup("earth")
```

```
##   model parameter          label forReg forClass probModel
## 1 earth    nprune         #Terms   TRUE     TRUE      TRUE
## 2 earth    degree Product Degree   TRUE     TRUE      TRUE
```

```r
## Inital corse tune of MARS model
set.seed(123) # set seed
MARS_TUNE <- train(x = trainPredictors, # use caret train to train the model on training predictors
                   y = trainResponse, # training response
                   method = "earth", # use the MARS algorithm
                   metric = "MAE", # initially assess the model using Mean Absolute Error
                   trControl = trainControl( method = "cv", # perform 10 fold cross validation
                                             number = 10),
                   tuneGrid = expand.grid(degree = 1:4, # tune for degree 1 --> 4
                                          nprune = seq(2,20,4) # tune over the range  --> 20
                   ))

summary(MARS_TUNE)
```

```
## Call: earth(x=matrix[126,19], y=c(8.2,8.333,8.2...), keepxy=TRUE, degree=1,
##             nprune=18)
##
##                                              coefficients
## (Intercept)                                     8.8519658
## h(0.468674-Y_Variance_Window_19_lagged_0)      -0.7379847
## h(Y_Variance_Window_19_lagged_0-0.468674)       0.4870206
## h(-1.09752-X1_Variance_Window_19_lagged_0)     -1.8340922
## h(X1_Variance_Window_19_lagged_0- -1.09752)     0.3765473
## h(1.08987-X1_mean_Window_5_lagged_3)            0.1851080
## h(1.27031-X3_Variance_Window_19_lagged_2)      -0.4702777
## h(0.271051-X3_mean_Window_19_lagged_0)          0.1624032
## h(X3_mean_Window_19_lagged_0-0.271051)         -0.4804273
## h(-0.533796-X4_Variance_Window_12_lagged_0)     1.4712317
## h(0.379168-X4_mean_Window_12_lagged_0)          0.1288337
## h(X4_mean_Window_12_lagged_0-0.379168)         -0.3913241
## h(1.54278-T1_Variance_Window_19_lagged_0)      -0.5482330
## h(T1_Variance_Window_19_lagged_0-1.54278)      -0.8086632
## h(T4_mean_Window_19_lagged_0-1.92648)          -2.2708108
## h(0.738915-T5_lagged_18)                       -0.4968090
## h(T5_lagged_18-0.738915)                       -1.5875102
##
## Selected 17 of 21 terms, and 10 of 19 predictors (nprune=18)
## Termination condition: RSq changed by less than 0.001 at 21 terms
## Importance: Y_Variance_Window_19_lagged_0, T5_lagged_18, ...
## Number of terms at each degree of interaction: 1 16 (additive model)
## GCV 0.06863749    RSS 4.711473    GRSq 0.978736    RSq 0.9882296
```

```r
cat(" Hyperparameters of best Tune", "\n",
    "nprune = ", MARS_TUNE$bestTune$nprune, "\n",
    "degree = ", MARS_TUNE$bestTune$degree)
```

```
##  Hyperparameters of best Tune
##  nprune =  18
##  degree =  1
```

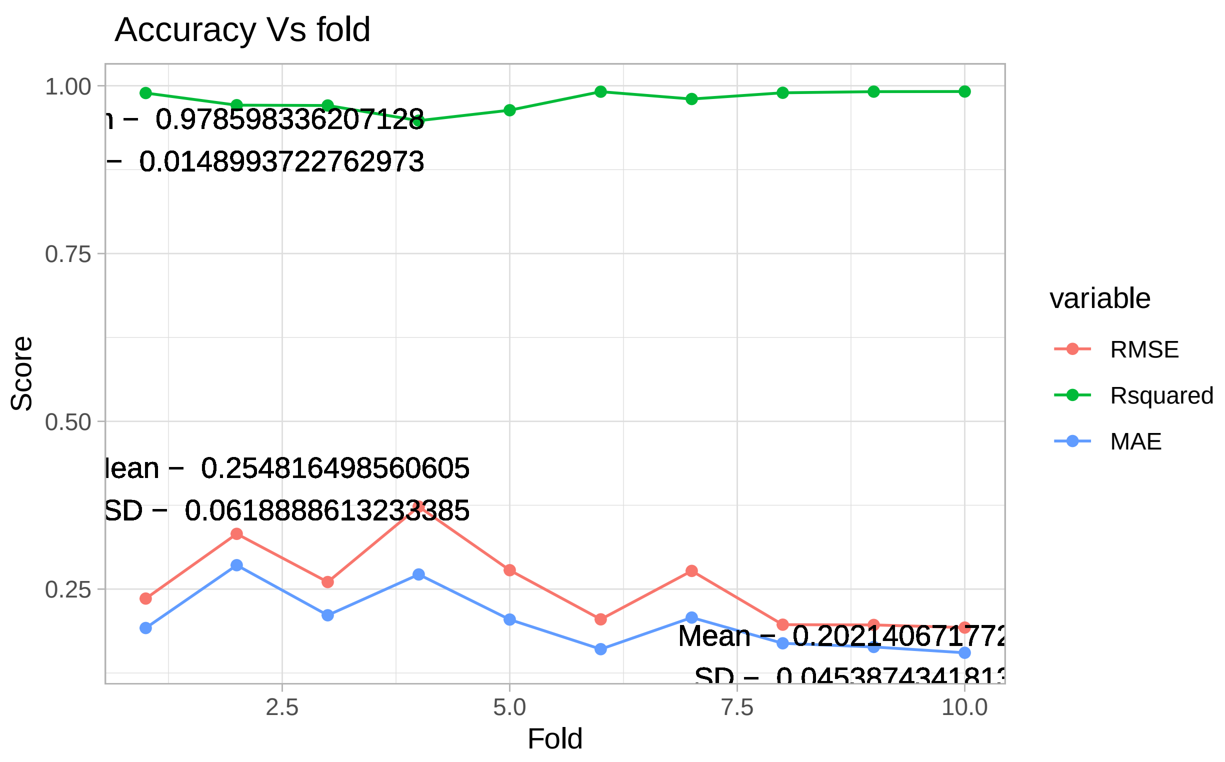

Figure 7: Measure of accuracy across all cross validation folds

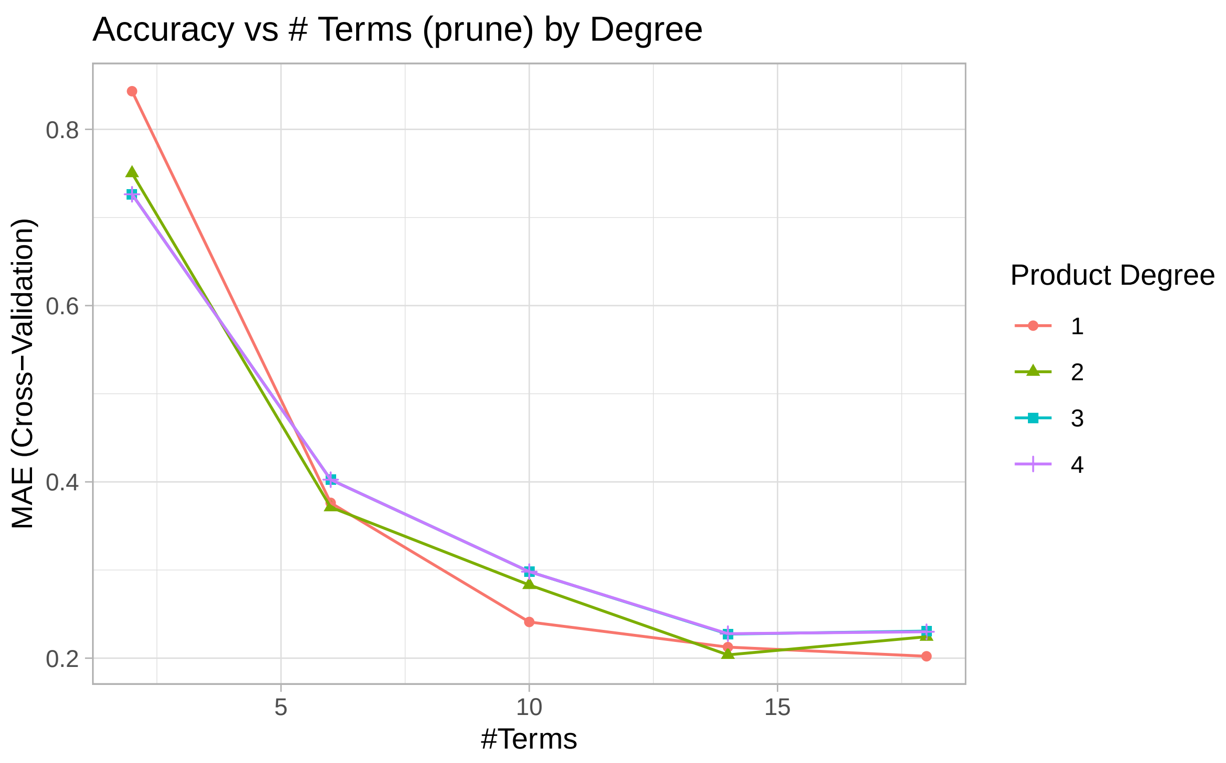

Figure 8: Average accuracy across all folds for each hyper-parameter

Model Performance - Training Data

In the calling of the second and final training run of the MARS model the following considerations where made:

- Accuracy Measure - Figure 6 shows a high degree of variation between folds. This is often due to the presence of more outlier or extreme values in some folds. To prevent over fitting to these cases the r-squared accuracy metric will be used for the final run. The r-squared metric scores the accuracy as the difference between the observation and prediction as a fraction of the distance from the mean, this should reduce the penalty for misfitting points at extreme values.
- Validation method - As over fitting was identified as a concern and a high degree of variability in the validation accuracy was observed in the initial run "leave one out cross validation" (LOOCV) was used as it can be more robust to outliers and help prevent over fitting as the model is trained on the largest possible data set. (Gupta 2017)
- Tuning Grid - as both hyper-parameters have to be integers the tuning grid refinement remains simple. The last run indicated degree = 1 and nprune = 18. Simply degree1 and 2 will be tested and nprune 3 values either side of the last training run so 15 - 21.

The final tuning call and model summary can be seen in the below code chunk and figure 9 shows the models average accuracy for each hyper-parameter combination. A increase in the accuracy was reported on the training set with a final r-squared value of 0.99 being an increase on the preliminary models 0.98. While this is a good improvement at this stage it is not known if it is at the cost of over fitting and thus poor predictive performance. This will be investigated using the test data set in the next section.
To aid in interpreting the influence of each variable the variable importance has been plotted in figure 10 it can be seen the following variables contribute to the model

the rank of variable importance is as follows with a brief interpretation:

- Unemployment - rolling Variance over previous 19 periods - the instability in unemployment over the previous ~ 5 year
- Normalised population growth from 18 periods previous - Population growth varying from the long term trend 4.5 years prior
- Normalised population growth averaged for 19 periods prior - average difference from long term average over ~ 5 years
- Percentage change in final consumption expenditure all industries (%) - averaged over ~ 5 years
- change in job vacancies - variance over previous 19 periods - the instability in job vacancies over the previous ~ 5 years
- CPI - averaged over ~ 5 years
- Change in gdp - averaged over 5 years and lagged 3
- Terms of trade index - average over previous 3 years
- Percentage change in final consumption expenditure all industries (%) variance over ~ 5 years lagged 6 months

Interestingly the lagged and rolling mean and variance variables are well represented in the list of most important variables. This suggests that the model is finding a correlation in the economic conditions in the previous 5 years and the current unemployment market. This reinforces the point raised in the introduction when motivating the model that accurate forecasting is required for governments to have policy settings appropriate to prevent future upward trends in unemployment as they clearly in to be inplace well ahead of time.

```
set.seed(123)
MARS_REFINED <- train(x = trainPredictors, # use caret train to train the model on training predictors
                      y = trainResponse, # training response
                      method = "earth", # use the MARS algorithm
                      metric = "Rsquared", # train using Rsquared
                      trControl = trainControl( method = "LOOCV", # use LOOCV to help reduce influence
                                                number = 1), # # to leave out
                      tuneGrid = expand.grid(degree = 1:2, # tune for degree 1 --> 2
                                             nprune = 15:21), # tune over the range 10 --> 25
)

summary(MARS_REFINED) # return summary of final model
```

```
## Call: earth(x=matrix[126,19], y=c(8.2,8.333,8.2...), keepxy=TRUE, degree=2,
##             nprune=20)
##
##                                                                                       coefficients
## (Intercept)                                                                              9.5761453
## h(Y_Variance_Window_19_lagged_0-0.468674)                                                0.4904748
## h(-0.41734-X1_mean_Window_5_lagged_3)                                                    0.1276987
## h(X1_mean_Window_5_lagged_3- -0.41734)                                                  -0.2546689
## h(0.379168-X4_mean_Window_12_lagged_0)                                                   0.1683073
## h(X4_mean_Window_12_lagged_0-0.379168)                                                  -0.5939741
## h(1.68025-T1_Variance_Window_19_lagged_0)                                               -0.5425903
## h(T1_Variance_Window_19_lagged_0-1.68025)                                               -0.7373306
## h(1.92648-T4_mean_Window_19_lagged_0)                                                   -0.1365954
## h(T4_mean_Window_19_lagged_0-1.92648)                                                   -2.2178036
## h(0.468402-T5_lagged_18)                                                                -1.6290390
## h(0.468674-Y_Variance_Window_19_lagged_0) * h(T5_mean_Window_19_lagged_0- -0.878242)   -0.6067798
## h(-0.268315-X3_Variance_Window_19_lagged_2) * h(1.92648-T4_mean_Window_19_lagged_0)    -0.2993573
## h(X3_mean_Window_19_lagged_0-0.271051) * h(1.68025-T1_Variance_Window_19_lagged_0)     -0.2152499
## h(X4_mean_Window_12_lagged_0- -0.617354) * h(1.92648-T4_mean_Window_19_lagged_0)        0.1609761
## h(-0.625968-T1_Variance_Window_19_lagged_0) * h(0.468402-T5_lagged_18)                  1.0043642
##
## Selected 16 of 23 terms, and 9 of 19 predictors (nprune=20)
## Termination condition: RSq changed by less than 0.001 at 23 terms
## Importance: T5_lagged_18, Y_Variance_Window_19_lagged_0, ...
## Number of terms at each degree of interaction: 1 10 5
## GCV 0.06142156    RSS 3.732213    GRSq 0.9809715    RSq 0.990676
```

```
cat(" Hyperparameters of best Tune", "\n", # text
    "nprune = ", MARS_REFINED$bestTune$nprune, "\n", # nprune hp
    "degree = ", MARS_REFINED$bestTune$degree) # degree hp
```

```
##  Hyperparameters of best Tune
##  nprune =  20
##  degree =  2
```

```
cat("Final Model Accuracy", "\n",
    "R-squared = ", MARS_TUNE$finalModel$rsq)
```

```
## Final Model Accuracy
##  R-squared =  0.9882296
```

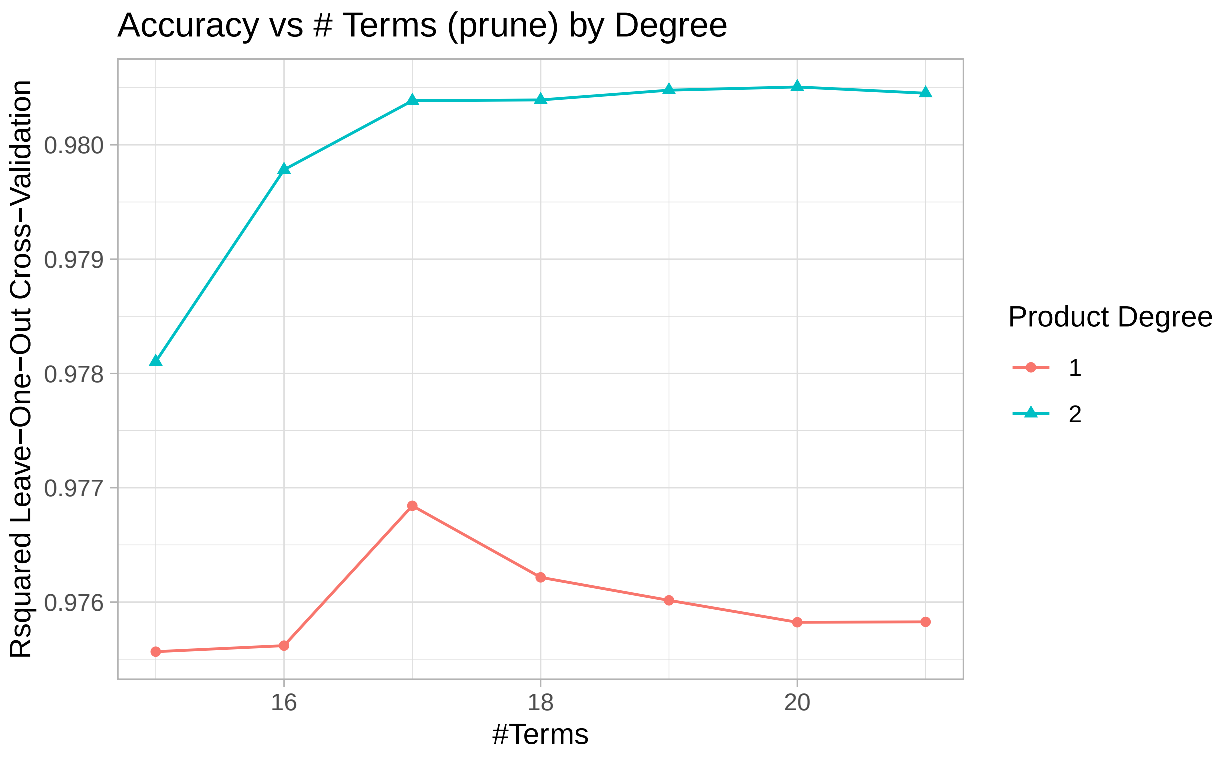

Figure 9: Average accuracy across all folds for each hyper-parameter combination showing degree = 2 and nprune = 20 as the optimal solution

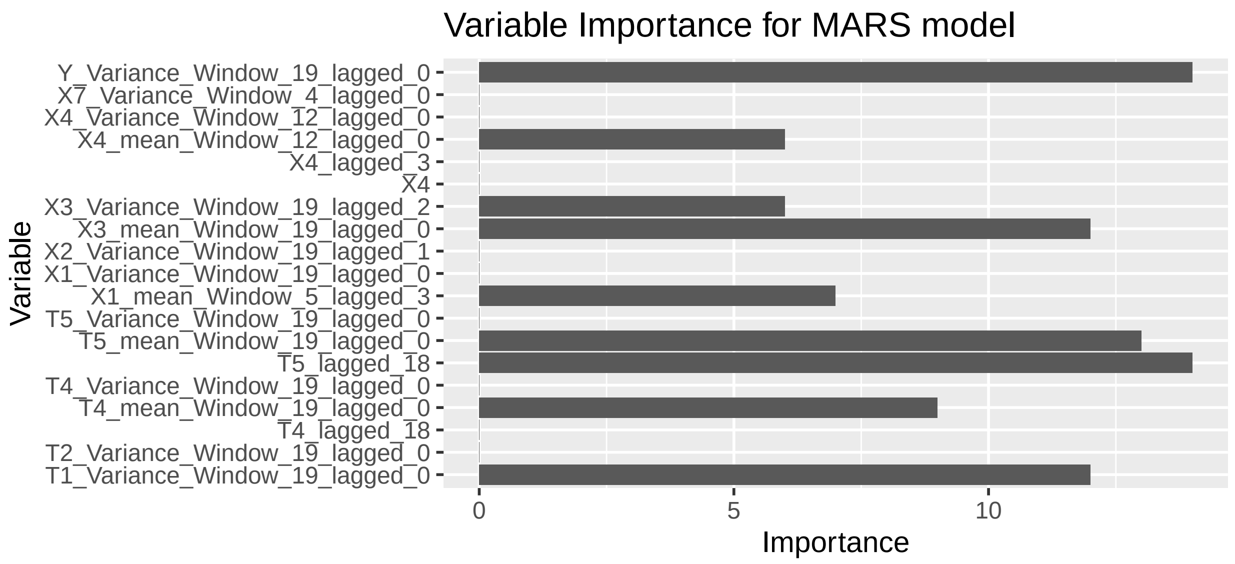

Figure 10: Plot of variable importance for final MARS model

Model Performance - Test Set

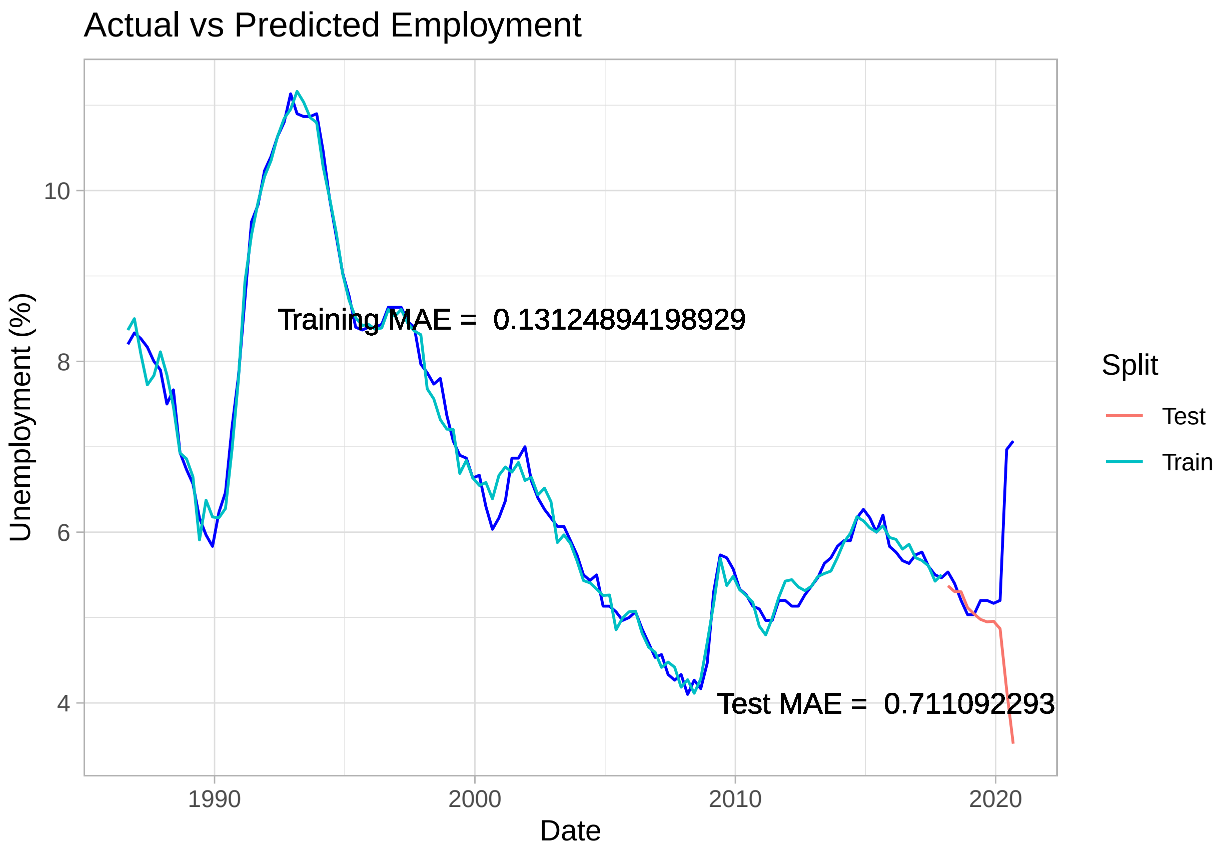

Figure 11: Predictions from MARS model and actual unemployment over both the test and training set

Finally the model was used to predict unemployment over the training period of March 2018 to September 2020. The predicted unemployment as well as the fitted values from the training data set have been plotted alongside the actual observed unemployment in figure 11. The reported performance metric is essentially meaningless as it can be seen the values after 2020 trend steeply in opposite directions. After closer inspection 3 distinct regions of performance can be seen.

- short term prediction - for the first 12 months the model performs well to accurately model the continued downward trend of unemployment.
- in the medium term - the model fails to accurately detect that shallowing and then upturn in unemployment. However this is only for a few periods and it can be seen in the training data that the model often lags changes in trends.
- the start of the corona virus pandemic - around the start of 2020 the model differs wildly from the actual observations. This correlates with the start of the corona virus pandemic. It is clear the dynamic of the employment market changed significantly at this point and these changes are not captures in the model. Factors such as lock downs, stand downs and travel restrictions would have changed the way in which the employment market worked. Further the statistics through this time would have lost meaning with issues such as how stood down workers are accounted for one such example. Finally with international boarders shut the assumptions of the long term linear population growth used to impute the missing population data would have failed to hold true. With population being the underlying statistic in 3 of the most important variables it is of little surprise the model performs so poorly.

Neural Network

Structure, assumptions and training plan

The development of Neural Networks is a much more challenging task for data scientists. The practically infinite number of possibilities to configure the network combine with longer training times leads to an extremely complex task. Since traditional machine learning algorithms require the selection of only a few known hyper parameter it is possible to scan the possible values of these and find the best option by following the direction of improvements, similar to gradient descent. When considering neural networks this space is so large it is computationally impossible to address all possible formulations. Further with so many dimension it is easy to become "trapped" in a local minima and not find the real best option. It is up-to the skill and intuition of the data scientist to overcome these challenges. To aid in simplifying the development of this neural network the following has been assumed as sufficient and not been investigated.

- The use of a fully connected (dense) neural network,
- Each hidden layer uses "relu" activation,
- The loss function used was "mse",
- The optimization function was "adam",
- the metric was "mae".

To develop the model the following general model generation and compile calls where used

```
build_model <- function(l, u, is, opt, loss, met, dr)
  { # function recieves # layers, # units, input shape, optimiser, loss and metrics
  ## define building blocks
  inputLayer <-
    layer_dense(units = u,
                activation = "relu",
                input_shape = c(is)) # define input layer with # features (columns) as the single dimen
  outputLayer <-
    layer_dense(units = 1) # define output layer with no activation function and a single output, suita
  hiddenLayer <- list( # define hidden layers as list of 16 dense layers with units u and activation "r
    #####
    layer_dense(unit = u, activation = "relu"),
    layer_dense(unit = u, activation = "relu"),
    layer_dense(unit = u, activation = "relu"),
  )
  #####
  ## build model
  model <- keras_model_sequential( # use keras_seqential to compile model
    name = paste("model_units-",u,"_layer-",l, sep = ""), # give model a meaningful name
    layers = c(inputLayer,  # conc input layer and output layer with user defined number of hidden laye
               hiddenLayer[1:l],
               outputLayer)
  )

  ## Compile Model
  model %>% compile(
    optimizer = opt,
    loss = loss,
    metrics = c(met)
  )
```

Implementation of Neural Network

The following outlines the steps undertaken to develop the neural network.

Initially a course tuning grid as shown in the below code chunk was used to perform a coarse sweep of potentially suitable models.

```
resultsGrid <-
  test_tune_grid( # call test tuen grid to build and test model with the following parameters
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(2,4,8,16), # define # layers to test
  units = c(4,8,16,20), # define # units to test
  batch = c(4,8,16,20), # batch size to trial
  e = 300, # define number of epochs (note callback is used so rarely will this number be achieved)
  delta = .001, # set delta for call back end training
  patience = 5, # set patience for call back end training
  dropout = 0, # regulations parameter, not used in this tune
  aim = 1 # set aim to return the results gris (ie train and Val MAE)
```

```
## ‘summarise()‘ has grouped output by ’Layers’, ’Units’. You can override using the ‘.groups‘ argument
```

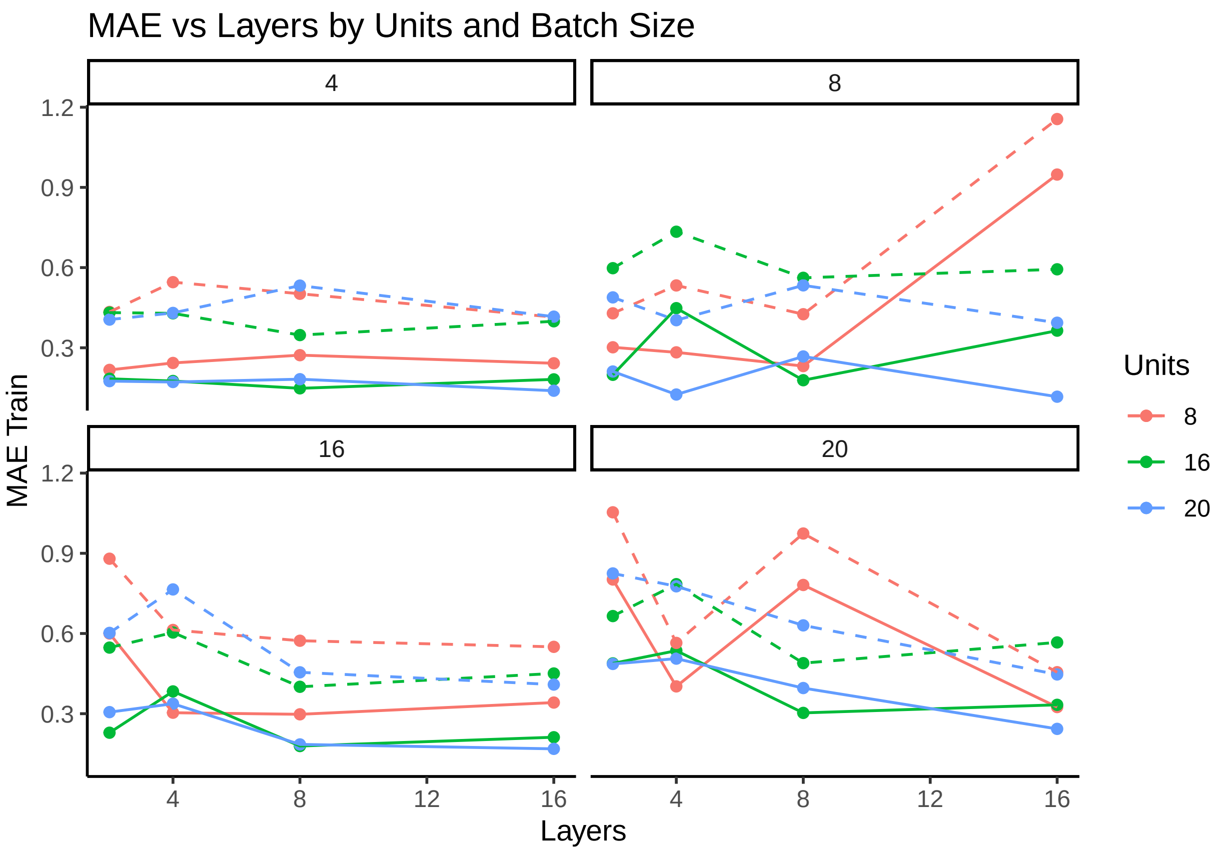

Figure 12: Results for coarse tuning grid

From figure 12 it was deduced that models trained with 4 batches out performed higher batch numbers. Further the "sweet spot" for model complexity seemed to be around 8-12 layers and 18-21 units in each layer.

The following code chunk shows the parameters used to call a finer tuning grid and the results from this grid can be seen in figure 13. It can be seen that a network of 12 layers and 21 units performed the best on the validation set and nearly as well as other configurations on the training data.

```
resultsGrid <-
  test_tune_grid( # call test tune grid to build and test model with the following parameters
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(8,9,10,11,12), # define # layers to test
  units = c(18,19,20,21), # define # units to test
  batch = c(2, 4), # batch size to trial
  e = 300, # define number of epochs (note  callback is used so rarley will this number be achieved)
  delta = .00001, # set delta for call back end training
  patience = 5, # set patience for call back end training
  dropout = 0, # regulations parameter, not used in this tune
  aim = 1 # set aim to return the results gris (ie train and Val MAE)
```

```
## 'summarise()' has grouped output by 'Layers', 'Units'. You can override using the '.groups' argument
```

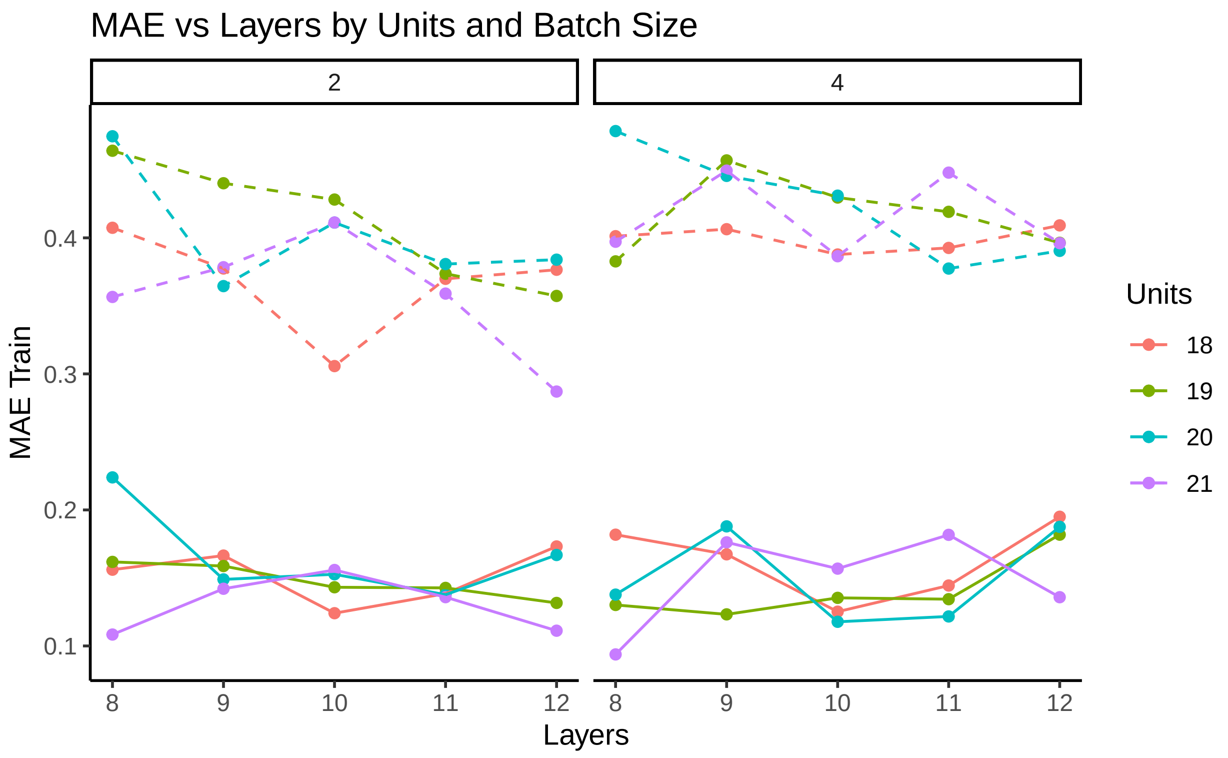

Figure 13: Results from fine tuining grid

The next step was to explore if the addition of regulisation improved the models performance. Only one form of regilzation was considered, drop out, with rates tested as shown in the call below. Drop out regulation works by preventing the model from fitting to random noise in the training set by omitting a percentage (the drop out rate) of information learnt between each layer. This leads to poorer performance on the training data but better proportionally better performance on the test data set and thus when the model is deployed on new data. As regulisation "forces" the model to "forget" some information it follows that to achieve similar performance the model may need more "capacity" which is provided by layers and units. The tuning grid was specified to explore this. The results of the regulasation tuning grid are shown in figure 14.

```
resultsGrid <-
  test_tune_grid( # call test tune grid to build and test model with the following parameters
  model_builder = build_model_reg, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12,24,36), # freeze layers from previous investigation
  units = c(20), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 500, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = .00000005, # set delta for call back end training
  patience = 10, # set patience for call back end training
  dropout = c(0.2,0.4,0.6), # regulations parameter
  aim = 1 # set aim to return the results gris (ie train and Val MAE)
)
```

```
## 'summarise()' has grouped output by 'Layers'. You can override using the '.groups' argument.
```

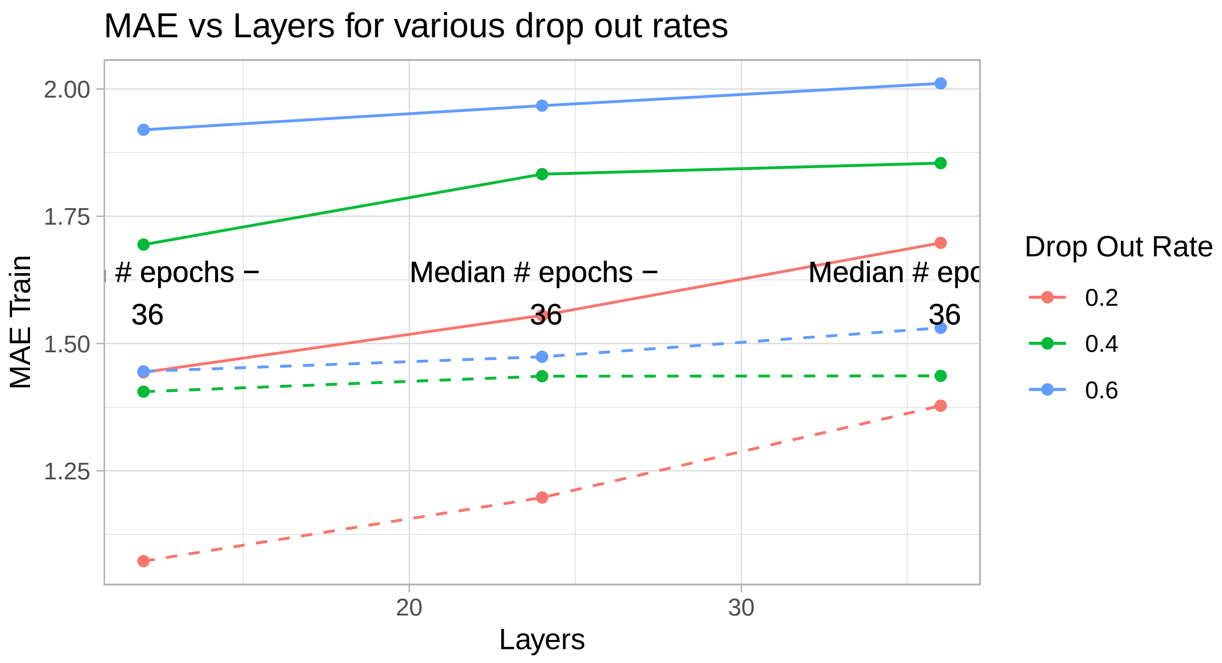

Figure 14: results from regulissation training grid.

The final step in refining the very limited option explored in this investigation is identifying the number of epoch's to train the data on. To do this 4 fold cross validation was again performed for a signifigant number of epoch's. The call for training is shown in the below code chunk and the results for both the dense and regulated network are shown in figure 12.

```r
## No reg
history_21_12 <-
  test_tune_grid( # call test tune grid to build and test model with the following parameters
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12), # freeze layers from previous investigation
  units = c(21), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 500, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = 0, # set delta for call back end training
  patience = 500, # effectively turn off stop early
  dropout = c(0), # regulations parameter
  aim = 2 # set aim to return the results gris (ie train and Val MAE)

## Reg
history_20_12_reg <-
  test_tune_grid( # call test tune grid to build and test model with the following parameters
  model_builder = build_model_reg, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12), # freeze layers from previous investigation
  units = c(20), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 150, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = 0, # set delta for call back end training
  patience = 50, # effectively turn off stop early
  dropout = c(0.2), # regulations parameter
  aim = 2 # set aim to return the results gris (ie train and Val MAE)
)
```

```
## Warning in melt(plotData, id.vars = c("epochs", "regulisation")): The melt
## generic in data.table has been passed a data.frame and will attempt to redirect
## to the relevant reshape2 method; please note that reshape2 is deprecated, and
## this redirection is now deprecated as well. To continue using melt methods from
## reshape2 while both libraries are attached, e.g. melt.list, you can prepend the
## namespace like reshape2::melt(plotData). In the next version, this warning will
## become an error.

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

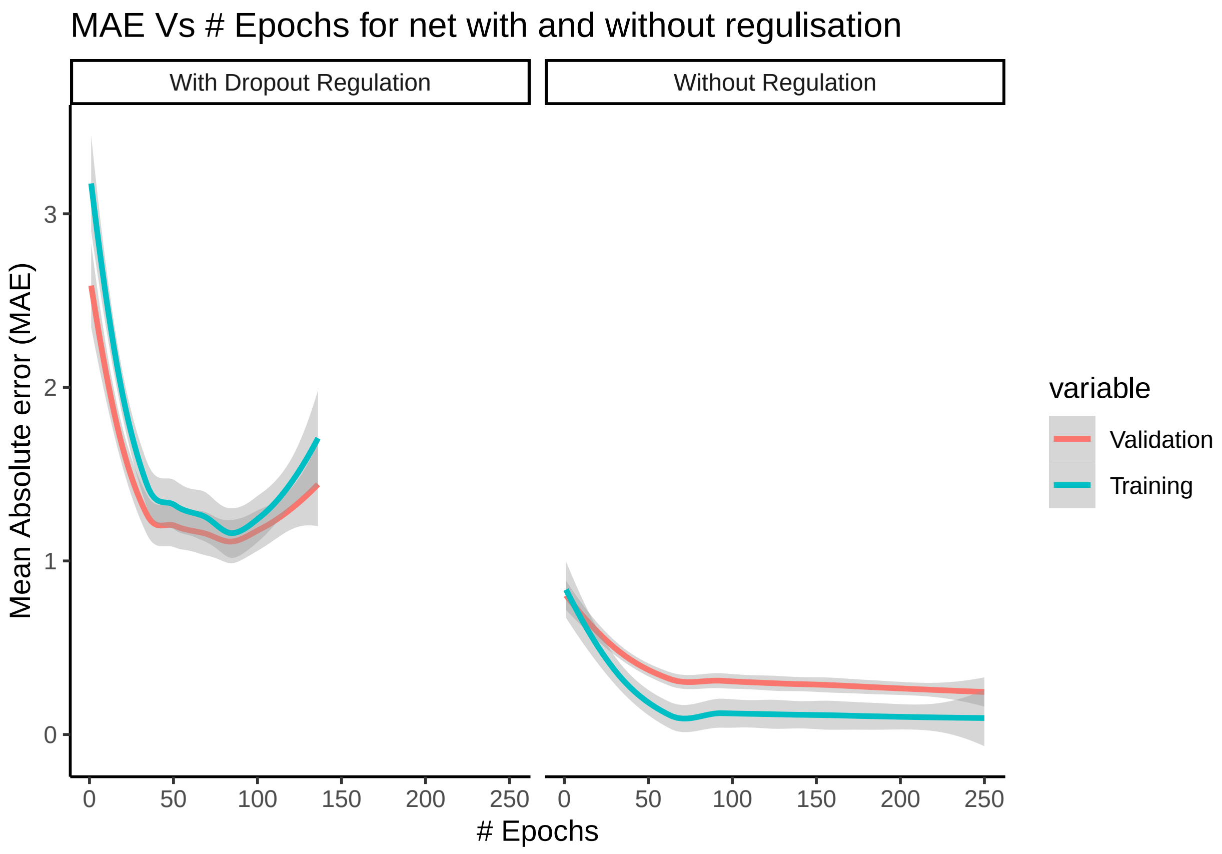

Figure 15: Traininng and Validation MAE for regulated and unregulated models

Performance on Training Data

The un-regulised model appeared to be performing best so it was selected for final assessment. The model was trained on the entire data set for a total of 75 epoch's. 75 epoch's correlates with the straining and test MAE minimum observed in figure 15. The code to train the model and make predictions on both the test and training set is shown in the code chunk in the below section.

```r
model_final <- build_model( # use the model builder without regulisation to build model
  l = 12, # build model with 12 layers
  u = 20,  # build model with 20 units
  is = ncol(trainPredictors), # input shape
  opt = "adam", # optimiser
  loss = "mse", # loss function
  met = "mae", # metrics
  dr = NA # dropout is NA as regulisation is not being used
)

## Fit Model and story history
history <- model_final %>% fit( # fit model and record results in history
  trainPredictors, # training predictors
  trainResponse, # training response
  epochs = 75, # epochs (predefined for easy adjustment)
  batch_size = 2 # batch size from tuning grid
  #callbacks = callbacks # predefined call backs
)

## return test and training data metrics
resultsTest <- model_final %>% evaluate(testPredictors, testResponse) # calculate fit metrics on test d
resultsTraining <- model_final %>% evaluate(trainPredictors, trainResponse) # calculate fit on training

## Predict unemployment using the test and training predictors
plotData_test$Predictions <- model_final %>% predict(testPredictors) %>% as.numeric() # return test pre
plotData_train$Predictions <- model_final %>% predict(trainPredictors) %>% as.numeric() # return traini
plotData_test$Split <- rep("Test", nrow(plotData_test)) # add note that these values are from test spli
plotData_train$Split <- rep("Train", nrow(plotData_train)) # add note that these values are from traini

timeSeriesData <- rbind(plotData_train, # combine all predictions into a single dataframe
                        plotData_test)

finalResults <- list(resultsTest, # list all relevant results for export
                     resultsTraining,
                     timeSeriesData)
```

Performance on Test Set

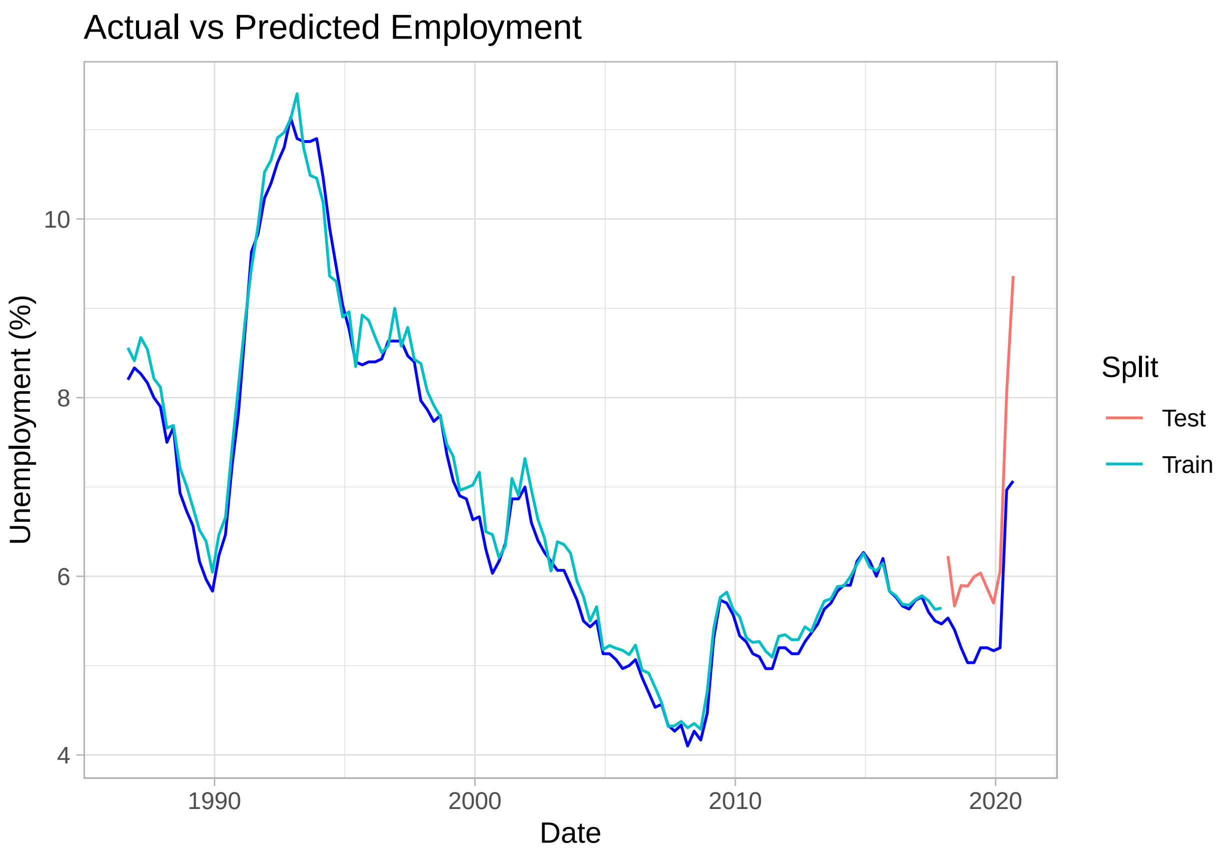

Figure 16: Time series plot of model fitted to training and test data with actual unemployment rates

It can be seen in figure 16 the neural network model fits the training set well This is also reflected in the low MAE scores shown in figure 15. However the consistently high MAE scores seen through out training on the validation sets has been reflected in the performance of the model on the test training set. It is however interesting to note how well the model follows the trend, especially after 2020 when the pandemic would of been influencing the economy.

Comparision of Models

Numerically the MARS model produced results with lower measures of error on both the test and training set. Further the Mars accurately predicted unemployment values in the short term while the neural network never accuratly predicted absolute numbers it very accuratly captured the trend, even under the extreme shock conditions of the pandemic.

The MARS model was computationally significantly cheaper this was due to two key factors

- The algorithm was computationally more efficient to solve
- the tuning grid was significantly smaller. This difference in grid size is even with the unrealistic restraints imposed on the size of the neural network to limit the size of investigation.

The MARS model provided a much more meaningful output as far as being able ti understand the value of predictors.

Conculsions

In conclusion both models have their strong points. The MARS models efficiency and short term accuracy are both impressive.

The neural network was ultimately constrained by compute time but the ability to accurate predict the trend during a shock such as the pandemic is impressive and the value of this should not be ignored.

The following should be considered to improve the prediction of unemployment

- combination of the models
- allowing the neural network to develop its own predictors and address the time series issues
- better investigation of seasonality of the data
- more accurate imputation methods of the missing values.

---

references

Claveria, O. Forecasting the unemployment rate using the degree of agreement in consumer unemployment expectations. J Labour Market Res 53, 3 (2019). https://doi.org/10.1186/s12651-019-0253-4

Karanassou, M. Sala, H. Labour Market Dynamics in Australia: What Drives Unemployment? (January 2009) (published in: Economic Record, 2010, 86 (273), 185-209) https://data.oecd.org/unemp/unemployment-rate-forecast.htm

Geron, A. (2019) Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow. Sebastopol: O'Reilly

Shumway, R & Stoffer, D (2019)Time Series Analysis and Its Applications. Cham: Springer

(ABS 2017)https://www.abs.gov.au/statistics/economy/price-indexes-and-inflation/guide-consumer-price-index-17th-series/latest-release

(RBA 2021) https://www.rba.gov.au/education/resources/explainers/pdf/australias-inflation-target.pdf?v=2021-10-13-08-05-37

Simpsons, S.(2021). The cost of unemployment to the economy https://www.investopedia.com/financial-edge/0811/the-cost-of-unemployment-to-the-economy.aspx

Liu J, Palomar DP (2021). imputeFin: Imputation of Financial Time Series with Missing Values. R package version 0.1.2, https://CRAN.R-project.org/package=imputeFin.

R Core Team (2020). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/

Nisbet, R. Miner, G. Yale, K. (2018). Handbook of Statistical Analysis and Data Mining Applications. London: Academic Press

Kuhn, M. (2008). Caret package. Journal of Statistical Software, 28(5).

GUPTA, P.(2017)cross validation in machine learning https://towardsdatascience.com/cross-validation-in-machine-learning-72924a69872f

Appendix 1 - Initial Data Exploration

```
## Set up environemnt and import data

rm(list = ls()) # removes all variables
if(!is.null(dev.list())) dev.off() # clear plots
cat("\014") # clear console

library(readxl, quietly = TRUE)
library(tidyr, quietly = TRUE)
library(dplyr, quietly = TRUE)
library(DataExplorer, quietly = TRUE)
library(reshape2, quietly = TRUE)
library(ggplot2, quietly = TRUE)
library(R.utils, quietly = TRUE)
library(data.table, quietly = TRUE)
library(tseries, quietly = TRUE)
library(roll, quietly = TRUE)
library(imputeFin, quietly = TRUE)
library(earth, quietly = TRUE)
library(caret, quietly = TRUE)

## Import Data from .xlsx
file <- "AUS_Data.xlsx" # set source data file name
dataIn <-  read_excel(file,
                      col_name = TRUE, # first row is column names
                      col_types = rep(c("numeric"), times = 9) # import all variables as numeric
                      ) # the data is imported, the .csv file has headers which will be used as the co

rawData <- data.frame(dataIn[-1 , ]) # change data structure to dataframe drop the first row as it was

rawData[ ,1] <- as.Date(as.POSIXct(rawData[ ,1]*24*60*60, origin = "1900-01-01")) # convert excel time
colnames(rawData)[1] <- c("Date") # set column name

## exploratory visulisation

introduce(rawData) # quick tables detailing data
plot_missing(rawData) # find missing values

plot(rawData$X6)
plot(rawData$X7)
```

Appendix 2 - Feature Engineering code

```r
## set up environment
rm(list = ls()) # removes all variables
if(!is.null(dev.list())) dev.off() # clear plots
cat("\014") # clear console

library(readxl, quietly = TRUE)
library(tidyr, quietly = TRUE)
library(dplyr, quietly = TRUE)
library(R.utils, quietly = TRUE)
library(data.table, quietly = TRUE)
library(tseries, quietly = TRUE)
library(roll, quietly = TRUE)
library(imputeFin, quietly = TRUE)
library(foreach, quietly = TRUE)
library(Hmisc, quietly = TRUE)

## Import Data change date into a meaningful format
file <- "AUS_Data.xlsx" # set source data file name
dataIn <-  read_excel(file, # import data from .xlsx
                      col_name = TRUE, # first row is column names
                      col_types = rep(c("numeric"), times = 9)) # import all variables as numeric

rawData <- data.frame(dataIn[-1 , ]) # change data structure to df drop the first row as it was the col

rawData[ ,1] <- as.Date(as.POSIXct(# convert excel time to POSIXct
  rawData[ ,1]*24*60*60, #excel time is days since 1990, POSIXct expects seconds from origin
  origin = "1900-01-01"))

colnames(rawData)[1] <- c("Date") # set column name

## Input missing values
rawData$X6 <- impute_AR1_Gaussian(rawData$X6) # values are in the middle and no clear trend so we will
attr(rawData$X6, "index_miss") <- NULL # remove attribute from data frame for simplicity

X7_NA <- lm(X7 ~ Date, # values are at the end and part of a stable linear trend so we will impute with
            data = rawData[year(rawData$Date) > 2010 & !is.na(rawData$X7), ]) # fit linear model on dat
for(i in 1:nrow(rawData)){ # cycle through missing values and input using linear model
  if(is.na(rawData$X7[i])){
    rawData$X7[i] = predict.lm(X7_NA, newdata = data.frame(Date = rawData$Date[i]))
  }
}

## Remove trends and correct for GST

rawData$T1 <- insert(diff(rawData$X6), 1, values = NA) / rawData$X6 * 100 # create T1 variable % change

rawData <- rawData %>% mutate(T2 = X6 / X7 * 100) %>% # create T2 Variable X6 normalized for population
  mutate(T3 = ifelse(Date < as.Date("2000-07-01"), X5 * 1.1, X5)) # create T3 Correct CPI for GST

rawData$T4 <- insert(diff(rawData$X5), 1, values = NA) / rawData$X5 * 400 # change X5 (CPI) to inflatio
```

```
POP_TREND <- lm(X7 ~ Date, data = rawData) # create long term linear model of population growth

rawData$T5 <- rawData$X7 - unname(predict.lm(POP_TREND, newdata = data.frame(Date = rawData$Date))) # r

## Calculate rolling trends (mean, standard deviation) for 2 to 20 quarters (6months to 5 years) for ea

VAR <- function(vec, w, response){
  ## Function to take time series data, calculate the rolling variance of the data over window w
  # then inspect for correlation at lags 0 --> 20 and return lagged variable that had the highest corre
  var <- roll_var(vec, width = w) # calculate rolling variance
  cor_var <- ccf(var, response, na.action = na.omit, lag.max = 20) # calculate rolling variance for 0 t
  lim <- qnorm((1 + 0.95)/2)/sqrt(cor_var$n.used) # calculate 95% confidence interval of correlation
  meaningful <- max(abs(cor_var$acf[1:20])) > lim # check if correlation is significant

  if (meaningful){ # if correlation is significant
    correlation <- abs(rev(cor_var$acf[1:20])) # vector of correlation coefficients from lag 0 to 20 (o
    lags <- which(diff(diff(correlation)>=0)<0) # identify local maxima of correlations along the lag
    if (length(lags) != 0){ # if a maxima is found
      max <- which.max(correlation[lags]) # identify which maxima is greatest
      lag <- lags[max] # set lag to the lag corresponding to the highest correlation
      var <- lag(var,lag) # apply lag to the rolling variance data
      cc <- correlation[lag]
      return(list(var,cc,lag))  # return vector and corresponding correlation
    }
    else { # if no maxima is found return the rolling data and correlation for lag = 0
      lag <- 0
      cc<- abs(cor_var$acf[21])
      return(list(var,cc,lag))
    }
  }
  else{ # if not meaningful return NA's for all values
    vec <- rep(NA, length(vec))
    cc <- NA
    lag <- NA
    return(list(vec, cc, lag))
  }
}

ME <- function(vec, w, response){
  ## Function to take time series data, calculate the rolling mean of the data over window w
  # then inspect for correlation at lags 0 --> 20 and return lagged variable that had the highest corre
  me <- roll_mean(vec, width = w) # calculate rolling variance
  cor_me <- ccf(me, response, na.action = na.omit, lag.max = 20) # calculate rolling mean for 0 to 20 t
  lim <- qnorm((1 + 0.95)/2)/sqrt(cor_me$n.used) # calculate 95% confidence interval of correlation
  meaningful <- max(abs(cor_me$acf[1:20])) > lim # check if correlation is significant

  if (meaningful){ # if correlation is significant
    correlation <- abs(rev(cor_me$acf[1:20])) # vector of correlation coefficients from lag 0 to 20
    lags <- which(diff(diff(correlation)>=0)<0) # identify local maxima of correlations along the lag
    if (length(lags) != 0){ # if a maxima is found
      max <- which.max(correlation[lags]) # identify which maxima is greatest
      lag <- lags[max] # set lag to the lag corresponding to the highest correlation
      me <- lag(me,lag) # apply lag to the rolling mean data
      cc <- correlation[lag]
      return(list(me,cc,lag))  # return vector and corresponding correlation
    }
    else{ # if no maxima is found return the rolling data and correlation for lag = 0
      lag <- 0
      cc<- abs(cor_me$acf[21])
      return(list(me,cc,lag))
    }
  }
  else{ # if not meaningful return NA's for all values
    vec <- rep(NA, length(vec))
    cc <- NA
    lag <- NA
    return(list(vec, cc, lag))
  }
}

VEC <- function(vec, response){

  ## Function to take time series data and inspect for correlation at lags 0 --> 20 and return lagged va
  cor_vec <- ccf(vec, response, na.action = na.omit, lag.max = 20) # calculate rolling mean for 0 to 20
  lim <- qnorm((1 + 0.95)/2)/sqrt(cor_vec$n.used) # calculate 95% confidence interval of correlation
  meaningful <- max(abs(cor_vec$acf[1:20])) > lim # check if correlation is significant

  if (meaningful){ # if correlation is significant
    correlation <- abs(rev(cor_vec$acf[1:20])) # vector of correlation coefficients from lag 0 to 20
    correlation[c(1,20)] <- 0 # set the first correlation to 0 so if it is an acending serris a maxima i
    lags <- which(diff(diff(correlation)>=0)<0) # identify local maxima of correlations along the lag
    if (length(lags) != 0){ # if a maxima is found
      max <- which.max(correlation[lags]) # identify which maxima is greatest
      lag <- lags[max] # set lag to the lag corresponding to the highest correlation
      vec <- lag(vec,lag) # apply lag to the rolling mean data
      cc <- correlation[lag]
      return(list(vec,cc,lag))  # return vector and corresponding correlation
    }
  }
  else{ # if not meaningful return NA's for all values
    vec <- rep(NA, length(vec))
    cc <- NA
    lag <- NA
    return(list(vec, cc, lag))
  }
}

ENG <- function(predictor, response){ # function to take response and predictor and find best rolling a
  #well as best lagging and return lagged predictors as a df

  varFeatures <- foreach(i = 2:20, .combine = cbind) %do% { # cycle over rolling window 2 --> 20 (6 mon
    VAR(predictor, i, response)
  }

  meFeatures <- foreach(i = 2:20, .combine = cbind) %do% { # cycle over rolling window 2 --> 20 (6 mont
    ME(predictor, i, response)
```

```r
  }

  lagFeatures <- VEC(predictor, response)

   n <- ifelse(length(which.max(unlist(varFeatures[seq(from = 2, to = length(varFeatures), by = 3)]))) =
              1, # if doesnt exist set to 1 and NA's will carry to next step
         which.max(unlist(varFeatures[seq(from = 2, to = length(varFeatures), by = 3)]))*3-2) # other

  varFeat_rol <- varFeatures[n] # assign the rolling feature which has highest correlation
  meFeat_rol  <- meFeatures[which.max(unlist(meFeatures[seq(from = 2, to = length(meFeatures), by = 3)]
  lagFeat <- lagFeatures[1] # assign best lagged features

  varFeat_rol_lag <- varFeatures[which.max(unlist(varFeatures[seq(from = 2, to = length(varFeatures), by
  meFeat_rol_lag <- meFeatures[which.max(unlist(meFeatures[seq(from = 2, to = length(meFeatures), by =
  lagFeat_lag <- lagFeatures[3] # record best lag

  varFeat_rol_window <- which.max(unlist(varFeatures[seq(from = 2, to = length(varFeatures), by = 3)]))
  meFeat_rol_window <- which.max(unlist(meFeatures[seq(from = 2, to = length(meFeatures), by = 3)])) #

  engFeatures <- data.frame(varFeat_rol, meFeat_rol, lagFeat) # group feature vectors together in data
  names(engFeatures) <- c( # give meaningful names
    paste("Variance_Window",unlist(varFeat_rol_window),"lagged", varFeat_rol_lag, sep  ="_"),
    paste("mean_Window",unlist(meFeat_rol_window),"lagged", meFeat_rol_lag, sep  ="_"),
    paste("lagged", lagFeat_lag, sep  ="_"))
  return(engFeatures)
}

engineeredFeatures <- foreach(i = 2:ncol(rawData), .combine = cbind) %do% { # apply ENG function to eac
  feat <- names(rawData)[i] # name of the feature currently loaded
  newFeat <- ENG(rawData[ ,i], rawData$Y) # list of new features
  label <- names(newFeat) # new feature names
  foreach(n = 1:length(names(newFeat)), .combine = c) %do% { # cycle over new feature names and add the
    label[n] <- paste(feat,label[n], sep = "_")
  }
  names(newFeat) <- label # assign new names
  newFeat # export list
}

## Inspect new feature space and evaluate which features should be kept
engineeredFeatures <- engineeredFeatures[ ,colSums(is.na(engineeredFeatures))<nrow(engineeredFeatures)]

Correlation <- apply(engineeredFeatures, 2, cor, y = rawData$Y, use = "complete.obs") # calculate corre
`# NA's` <- colSums(is.na(engineeredFeatures)) # count the number of NA's (ie how many observations wil

corPlot_df <- data.frame(`Rows Lost` = `# NA's`, Cor = Correlation)
saveRDS(corPlot_df, file = "corPlot.RDS")
corPlot<- plot(x = `# NA's`, y = 1 - abs(Correlation)) + # plot # NA's vs inverse correlation to determ
text(x = `# NA's`, y = .95 - abs(Correlation), label = `# NA's`) # add labels for # NA's for easy ident
engineeredFeatures <- engineeredFeatures[ ,colSums(is.na(engineeredFeatures))<21] # Remove features wit

modelData <- data.frame(rawData, engineeredFeatures) # combine original variables and engineered featur
modelData <- modelData[complete.cases(modelData), ] # remove observations which are incomplete ( due to
## Inspect for correlation

dataMatrix <- as.matrix(modelData[ , -1]) # convert to matrix, excluding date

pCor <- rcorr(dataMatrix, type = "pearson") # return pearson correlation and signifigance levels
pCor_matrix <- pCor$r # correlation matrix
pCor_sig <- pCor$P # signifigance levels

toRemove <- foreach(m = 2:ncol(pCor_matrix),  .combine = c) %do% { #remove junk (defined as correlation
  foreach(n = 2:nrow(pCor_matrix), .combine = c) %do% { # iterate over all rows and columns
    if (n != m){ # if not on self
      if(pCor_matrix[n,m] > 0.95 | pCor_sig[1,m] > 0.05){ # test for "junk"
        colName <- rownames(pCor_matrix)[m] #find variable name
        colName
      }
    }
  }
}

toRemove <- unique(toRemove) # unique names only
(toRemove) # check
length(toRemove) # check
modelData <-modelData[ ,names(modelData) %nin% toRemove] # remove junk features

## Build test training split, normalize data.

testData <- modelData[modelData$Date > as.Date("2018-02-28"), c(-1)] # test Data all observations after
trainData <- modelData[modelData$Date <= as.Date("2018-02-28"), c(-1)] # train Data all other obs
results <- modelData[modelData$Date > as.Date("2018-02-28"), c(1,2)] # test response and date (for plot
trainAss <- modelData[modelData$Date <= as.Date("2018-02-28"), c(1,2)] # training response and date

mean <- apply(trainData[ , -1], 2, mean) # calculate mean for each variable
std <- apply(trainData[ , -1], 2, sd) # calculate SD for each variable

testPred = scale(testData[ ,-1], center = mean, scale = std) # scale test data and make predictor df
testResponse = testData[,1] # make response list

trainPred = scale(trainData[ ,-1], center = mean, scale = std) # scale training data and make pred df
trainResponse = trainData[, 1] # make train response df

## check data splits make sense
nrow(modelData)
nrow(trainPred)
length(trainResponse)
nrow(testPred)
length(testResponse)
## Save objects for use in various models
saveRDS(list(rawData, modelData, testData, trainData, results, trainAss, testPred, testResponse, trainP
```

Appendix 3 - Neural Network code

```
## Set up Environment and Import Data
rm(list = ls()) # removes all variables
if(!is.null(dev.list())) dev.off() # clear plots
cat("\014") # clear console
.rs.restartR() # restart r session to disconnected any old python attachments

library(keras, quietly = TRUE)
library(ggplot2, quietly = TRUE)
library(tensorflow, quietly = TRUE)
library(reticulate, quietly = TRUE)
library(caret, quietly = TRUE)

## Set env variables and check Py
#Sys.setenv(RETICULATE_PYTHON = "/home/sean/anaconda3/envs/r-reticulate/bin/python") # for reticulate t
Sys.setenv(RETICULATE_PYTHON = "/home/veering/anaconda3/envs/r-reticulate/bin/python") # for reticulate
Sys.getenv("RETICULATE_PYTHON") # check Sys variable is correct
py_config() # check python config

## Set seed for keras
set_random_seed(123)

## Read data in
Time <- Sys.time() # record start time

#####
data <- readRDS(file = "data.RDS") # import model data prerpared earlier and daved in RDS file
plotData_test <- as.data.frame(data[5]) # test reponses and dates
plotData_train <- as.data.frame(data[6]) # train response and dates
testPredictors <- as.matrix(as.data.frame(data[7])) # test predictors, matrix for tf
testResponse <- as.numeric(unlist(data[8])) # test response, numeric vec for tf
trainPredictors <- as.matrix(as.data.frame(data[9])) # training predictors, as matrix for tf
trainResponse <- as.numeric(unlist(data[10])) # training response, as matrix for tf
#####

## check dimensions and type of all data
#####
dim(testPredictors)
typeof(testPredictors)

length(testResponse)
typeof(testResponse)

dim(trainPredictors)
typeof(trainPredictors)

length(trainResponse)
typeof(trainResponse)

## Define function to create sequential keras model with specified hidden layers and units.
#####
build_model <- function(l, u, is, opt, loss, met, dr){ # function recieves # layers, # units, input sha
```

```
## define building blocks
inputLayer <- layer_dense(units = u, activation = "relu", input_shape = c(is)) # define input layer w
outputLayer <- layer_dense(units = 1) # define output layer with no activation function and a single
hiddenLayer <- list( # define hidden layers as list of 16 dense layers with units u and activation "r
  #####
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu"),
  layer_dense(unit = u, activation = "relu")
  )
  #####

  ## build model
  model <- keras_model_sequential( # use keras_seqential to compile model
    name = paste("model_units-",u,"_layer-",l, sep = ""), # give model a meaningful name
    layers = c(inputLayer,  # conc input layer and output layer with user defined number of hidden laye
               hiddenLayer[1:l],
               outputLayer)
  )

  ## Compile Model
  model %>% compile(
    optimizer = opt,
    loss = loss,
    metrics = c(met)
  )
}

build_model_reg <- function(l, u, is, opt, loss, met, dr){ # function recieves # layers, # units, input

  ## define building blocks
  inputLayer <- layer_dense(units = u, activation = "relu", input_shape = c(is)) # define input layer w
  outputLayer <- layer_dense(units = 1) # define output layer with no activation function and a single
  hiddenLayer <- list( # define hidden layers as list of 16 dense layers with units u and activation "r
    #####
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr),
    layer_dense(unit = u, activation = "relu"),
    layer_dropout(rate = dr)
  )
  #####

  ## build model
  model <- keras_model_sequential( # use keras_seqential to compile model
    name = paste("model_units-",u,"_layer-",l, "_reg-", dr, sep = ""), # give model a meaningful name
    layers = c(inputLayer,  # conc input layer and output layer with user defined number of hidden laye
               hiddenLayer[1:(l*2)], # double layer as each hiden layer is now 2 elemenst long
               outputLayer)
  )

  ## Compile Model
  model %>% compile(
    optimizer = opt,
    loss = loss,
    metrics = c(met)
  )
}

test_tune_grid <- function(model_builder, trainPredictors, trainResponse, k, layers, units, batch, e, d
  ## Create 4 x splits set from training data for validation

  set.seed(123) # set seed
  trainIndex <- createDataPartition(trainResponse, p = 0.8, times = k, list = FALSE) # use caret to cre

  ## Build tuning grid
  #####
  tuneGrid <- expand.grid(
    l = layers, # layers
    u = units, # units
    b = batch, # obs in batch
    dr = dropout # dropout rate
  )
  #####

  ## build results grid to store MSE and MAE from training and validation
  #####
```

```r
resultsGrid <- expand.grid(
  k = 1:k, # the for loops will iterate over k first
  l = layers, # layers
  u = units, # units
  b = batch, # obs in batch
  dr = dropout, # dropout rate
  tg = NA, # NA to hold place for metrics
  i = NA,
  mae = NA,
  mae_val = NA,
  num_epoch = NA
)
#####

## Call tensor board and define call backs
#tensorboard("my_log_dir")
callbacks = list(
  #callback_tensorboard( # call TB
  #  log_dir = "my_log_dir",
  #  histogram_freq = 1),
  callback_early_stopping( # call early stopping
    monitor = "val_loss", # use loss on validation set as metric
    min_delta = delta, # min change to metric
    patience = patience) # epochs to tolerate < min change before stopping
)

## Build and train each model over all 4 folds

rg <- 1 # results grid counter
histVal  <- c() # reserve variable name
histTrain <- c() # reserve variable name

## Train models and record results
for(tg in (1:nrow(tuneGrid))) { # iterate over each row of the tune grid
  for(i in 1:k){ # iterate over each fold
    cat("fold # ", i," from tg # ", tg, " storing results in rg # ", rg, "\n", # tracker for console
        "Layer = ", tuneGrid[tg, 1],
        "Unit = ", tuneGrid[tg, 2],
        "Batch Size = ", tuneGrid[tg, 3],
        "Dropout Rate = ", tuneGrid[tg, 4], "\n")

    ## make training and Validation split
    x <- as.matrix(trainPredictors[trainIndex[, i], ]) # predictors as x
    y <- as.numeric(trainResponse[trainIndex[, i]]) # response as y
    x_val <- as.matrix(trainPredictors[-trainIndex[, i], ]) # validation predictors
    y_val <- as.numeric(trainResponse[-trainIndex[, i]]) # validation response

    ### Troubleshooting
    #####
    # dim(x)
    # typeof(x)
    # length(y)
    # typeof(y)
```

```r
      # dim(x_val)
      # typeof(x_val)
      # length(y_val)
      # typeof(y_val)
      #####

      ## use make function to define and compile model
      set.seed(123) # set seed
      model <- model_builder(l = tuneGrid[tg, 1], # build model this # layers from tg
                             u = tuneGrid[tg, 2],  # build model with # units from tg
                             is = ncol(trainPredictors), # input shape
                             opt = "adam", # optimiser
                             loss = "mse", # loss function
                             met = "mae", # metrics
                             dr = tuneGrid[tg, 4] # dropout rate
      )

      ## Fit model
      set.seed(123) # set seed
      history <- model %>% fit( # fit model and record results in history
        x, # training predictors
        y, # training response
        epochs = e, # epochs (predefined for easy adjustment)
        batch_size = tuneGrid[tg, 3], # batch size from tuning grid
        validation_data = list( # list validation predictors and response
          x_val,
          y_val),
        callbacks = callbacks # predefined call backs
      )

      ## return metrics of interest to resultsGrid
      resultsGrid$tg[rg] <- tg # store tg number for result traceability
      resultsGrid$i[rg] <- i # return fold number for result traceability
      resultsGrid$mae[rg] <- min(history$metrics$mae) # assume the min value is the final value (ok for
      resultsGrid$mae_val[rg] <- min(history$metrics$val_mae)
      resultsGrid$num_epoch[rg] <- length(history$metrics$mae) # count epoch's by length of metric vect

      ## return metric for each epoch
      histVal <- rbind(histVal, history$metrics$val_mae) # combine with other folds
      histTrain <- rbind(histTrain, history$metrics$mae) # combine with other folds

      rg <- rg + 1 # increase results grid counter by 1
    }
  }
  if(aim == 1)  return(resultsGrid)
  if(aim == 2)  return(list(histVal, histTrain))
}
#####

## Course Tune
#####
## record start time
## record start time
```

```
startTime <- Sys.time()

resultsGrid <- test_tune_grid( # call test tuen grid to build and test model with the following paramet
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(2,4,8,16), # define # layers to test
  units = c(4,8,16,20), # define # units to test
  batch = c(4,8,16,20), # batch size to trial
  e = 300, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = .001, # set delta for call back end training
  patience = 5, # set patience for call back end training
  dropout = 0, # regulations parameter, not used in this tune
  aim = 1 # set aim to return the results gris (ie train and Val MAE)
)

## record finish time
finishTime <- Sys.time()
(runTime_ct <- difftime(finishTime, startTime))
#####
## save resultsGrid to RDS for use later
saveRDS(resultsGrid,"resultsGrid_NN_course_gloud.RDS")

## Fine Tune
#####
## record start time
## record start time
startTime <- Sys.time()

resultsGrid <- test_tune_grid( # call test tune grid to build and test model with the following paramet
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(8,9,10,11,12), # define # layers to test
  units = c(18,19,20,21), # define # units to test
  batch = c(2, 4), # batch size to trial
  e = 300, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = .00001, # set delta for call back end training
  patience = 5, # set patience for call back end training
  dropout = 0, # regulations parameter, not used in this tune
  aim = 1 # set aim to return the results gris (ie train and Val MAE)
)

## record finish time
finishTime <- Sys.time()
(runTime_ft <- difftime(finishTime, startTime))
#####
## save resultsGrid to RDS for use later
saveRDS(resultsGrid,"resultsGrid_NN_fine_gcloud.RDS")

## Investigate regularization
```

```r
#####
## record start time
## record start time
startTime <- Sys.time()

resultsGrid <- test_tune_grid( # call test tune grid to build and test model with the following paramet
  model_builder = build_model_reg, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12,24,36), # freeze layers from previous investigation
  units = c(20), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 500, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = .00000005, # set delta for call back end training
  patience = 10, # set patience for call back end training
  dropout = c(0.2,0.4,0.6), # regulations parameter
  aim = 1 # set aim to return the results gris (ie train and Val MAE)
)

## record finish time
finishTime <- Sys.time()
(runTime_reg <- difftime(finishTime, startTime))
#####
## save resultsGrid to RDS for use later
saveRDS(resultsGrid,"resultsGrid_NN_reg_gcloud.RDS")

## Build 2 final models for comparison and assessment one with drop out one with out
#####
## record start time
startTime <- Sys.time()

## No reg
history_21_12 <- test_tune_grid( # call test tune grid to build and test model with the following param
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12), # freeze layers from previous investigation
  units = c(21), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 500, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = 0, # set delta for call back end training
  patience = 500, # effectively turn off stop early
  dropout = c(0), # regulations parameter
  aim = 2 # set aim to return the results gris (ie train and Val MAE)
)

## save history_19_11 to RDS for use later
saveRDS(history_21_12,"history_21_12.RDS")

## Reg
history_20_12_reg <- test_tune_grid( # call test tune grid to build and test model with the following p
  model_builder = build_model_reg, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12), # freeze layers from previous investigation
  units = c(20), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 150, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = 0, # set delta for call back end training
  patience = 50, # effectively turn off stop early
  dropout = c(0.2), # regulations parameter
  aim = 2 # set aim to return the results gris (ie train and Val MAE)
)

## record finish time
finishTime <- Sys.time()
(runTime_mods <- difftime(finishTime, startTime))
#####
## save history_19_11 to RDS for use later
saveRDS(history_20_12_reg,"history_20_12_reg_02.RDS")

## Build final model epochs = 75, no regulation and fit to training data, then test on test data
#####
## Build Model
startTime <- Sys.time() # set start time

#tensorboard("my_log_dir") # call tensor board

#callbacks = list( # only call back TB, no need to stop training early as we have selected the desired
  #callback_tensorboard( # call TB
   # log_dir = "my_log_dir",
    #histogram_freq = 1))

model_final <- build_model( # use the model builder without regulisation to build model
  l = 12, # build model with 12 layers
  u = 20,  # build model with 20 units
  is = ncol(trainPredictors), # input shape
  opt = "adam", # optimiser
  loss = "mse", # loss function
  met = "mae", # metrics
  dr = NA # dropout is NA as regulisation is not being used
)

## Fit Model and story history
history <- model_final %>% fit( # fit model and record results in history
  trainPredictors, # training predictors
  trainResponse, # training response
  epochs = 75, # epochs (predefined for easy adjustment)
  batch_size = 2 # batch size from tuning grid
  #callbacks = callbacks # predefined call backs
)

## return test and training data metrics
```

```
resultsTest <- model_final %>% evaluate(testPredictors, testResponse) # calculate fit metrics on test d
resultsTraining <- model_final %>% evaluate(trainPredictors, trainResponse) # calculate fit on training

## Predict unemployment using the test and training predictors
plotData_test$Predictions <- model_final %>% predict(testPredictors) %>% as.numeric() # return test pre
plotData_train$Predictions <- model_final %>% predict(trainPredictors) %>% as.numeric() # return traini
plotData_test$Split <- rep("Test", nrow(plotData_test)) # add note that these values are from test spli
plotData_train$Split <- rep("Train", nrow(plotData_train)) # add note that these values are from traini

timeSeriesData <- rbind(plotData_train, # combine all predictions into a single dataframe
                        plotData_test)

finalResults <- list(resultsTest, # list all relevant results for export
                     resultsTraining,
                     timeSeriesData)
#####
finishTime <- Sys.timme() # Set finish time
(runTime_mod <- difftime(finishTime, startTime))
(runTime_all <- difftime(finishTime, Time))
saveRDS(finalResults, file = "finalResults_gcloud.RDS")
saveRDS(list(runTime_ct, runTime_ft, runTime_reg, runTime_mods, runTime_mod, runTime_all), file = "runTi
```

Appendix 4 - r Markdown document code

```
## Set up environemnt and import data

rm(list = ls()) # removes all variables
if(!is.null(dev.list())) dev.off() # clear plots
cat("\014") # clear console

library(readxl, quietly = TRUE)
library(tidyr, quietly = TRUE)
library(dplyr, quietly = TRUE)
library(DataExplorer, quietly = TRUE)
library(reshape2, quietly = TRUE)
library(ggplot2, quietly = TRUE)
library(R.utils, quietly = TRUE)
library(data.table, quietly = TRUE)
library(tseries, quietly = TRUE)
library(roll, quietly = TRUE)
library(imputeFin, quietly = TRUE)
library(earth, quietly = TRUE)
library(caret, quietly = TRUE)

## Import Data from .xlsx
file <- "AUS_Data.xlsx" # set source data file name
dataIn <-  read_excel(file,
                      col_name = TRUE, # first row is column names
                      col_types = rep(c("numeric"), times = 9) # import all variables as numeric
                      ) # the data is imported, the .csv file has headers which will be used as the co

rawData <- data.frame(dataIn[-1 , ]) # change data structure to dataframe drop the first row as it was

rawData[ ,1] <- as.Date(as.POSIXct(rawData[ ,1]*24*60*60, origin = "1900-01-01")) # convert excel time
colnames(rawData)[1] <- c("Date") # set column name

## Create time series plot of response and predictors
plotData <- rawData # move data to new Df for manipulation for plotting
colnames(plotData) <- c("Date",
                        "Y-Unemployment Rate (%)",
                        "X1-Change GDP (Perc)",
                        "X2-D Consumption Exp. (gov) (%)",
                        "X3-D Consumption Exp. (All) (%)",
                        "X4-Term of Trade Index (%)",
                        "X5-CPI",
                        "X6-# Job Vacancies (Thousands)",
                        "X7-Population (Thousands)")

plotData[ ,-1] <- scale(plotData[ , -1], center = TRUE, scale = TRUE) # scale all values except the per
plotData <- melt( plotData, # Melt Data for plotting
  id.vars = c("Date"), # use Periods as ID's
  varnames = c("Date", "variable", "value"), # Set column names
  na.rm = TRUE) # remove missinng values for the time being

timeSeries_plot_init <- ggplot(data = plotData, aes(x = Date, y = value)) + # plot Data
```

```
  geom_line(aes(color = variable), show.legend = FALSE, size = 0.5) + # add line showing data
  geom_smooth(colour = "darkgrey", method = "loess", span = 0.2, alpha = .5, se = FALSE, linetype = "lo
  facet_wrap(plotData$variable) +
  theme_light()

timeSeries_plot_init

## quick box plot
plotData <- rawData # move data to new Df for manipulation for plotting
colnames(plotData) <- c("Date",
                        "Y",
                        "X1",
                        "X2",
                        "X3",
                        "X4",
                        "X5",
                        "X6",
                        "X7")
plotData[ ,-1] <- scale(plotData[ , -1], center = TRUE, scale = TRUE) # scale all values except the per
plotData <- melt( plotData, # Melt Data for plotting
  id.vars = c("Date"), # use Periods as ID's
  varnames = c("Date", "variable", "value"), # Set column names
  na.rm = TRUE) # remove missinng values for the time being

boxplot(plotData$value~plotData$variable, ann=FALSE)



X6_plot <- plot_imputed(impute_AR1_Gaussian(rawData$X6)) # use impute fin to calculate and plot missing
X6_plot

X7_Missing <- as.factor(is.na(dataIn$X7[-1])) # create factor if an observation of X7 was NA or not
X7_NA <- lm(X7 ~ Date,
            data = rawData[year(rawData$Date) > 2010 & !is.na(rawData$X7), ]) # fit linear model on dat
for(i in 1:nrow(rawData)){
  if(is.na(rawData$X7[i])){
    rawData$X7[i] = predict.lm(X7_NA, newdata = data.frame(Date = rawData$Date[i]))
  }
}
X7_plot <- ggplot(data = rawData) + # create ggplot
  geom_line(aes(x = Date, y = X7, colour = X7_Missing)) + # add line colourd by if the value was impute
  labs(title = "X7 vs Date, coloured by missingness")+ # add title
  theme_classic()
X7_plot

rawData$T1 <- # create T1 variable % change period on period to X6
  insert(diff(rawData$X6), 1, values = NA) / rawData$X6 * 100
rawData <-
  rawData %>% mutate(T2 = X6 / X7 * 100) %>% # create T2 Variable X6 normalized for population growth
  mutate(T3 = ifelse(Date < as.Date("2000-07-01"), X5 * 1.1, X5)) # create T3 Correct CPI for GST
rawData$T4 <- # change X5 (CPI) to inflation, calculated quarterly but anualized
  insert(diff(rawData$X5), 1, values = NA) / rawData$X5 * 400
POP_TREND <- # create long term linear model of population growth
```

```r
  lm(X7 ~ Date, data = rawData)
rawData$T5 <- # remove long term linear trend of population growth
  rawData$X7 - unname(predict.lm(POP_TREND, newdata = data.frame(Date = rawData$Date)))

POP_TREND <- # create long term linear model of population growth
  lm(X7 ~ Date, data = rawData)

summary(POP_TREND)

## Inspect Data for seasonality
plotData <- rawData %>%
  mutate(season = month(Date)/3, .after = Date) %>% # create season variable as 1 = summer 2 = autumn 3
  mutate(year = year(Date), .after = Date)

plotData <- plotData[ , c(1,2,3,4)] # select only required variables

plotData$Y <- insert(diff(plotData$Y), 1, values = NA) / plotData$Y * 100 # calculate change in unemplo
Ymin <- min(plotData$Y, na.rm = TRUE)
Ymax <- max(plotData$Y, na.rm = TRUE)
#plotData$Ystd <- (plotData$Y - Ymin) / (Ymax - Ymin)
#plotData$Ystd <- plotData$Y

plotData <- melt( plotData[ ,-1], # Melt Data for plotting
                  id.vars = c("year", "season"),
                  na.rm = TRUE) # remove missing values for the time being
plotData <- plotData[order(plotData$year, as.numeric(plotData$season)), ]

seasonal_plot <- ggplot(data = plotData, aes(x = season, y = value)) +
  geom_line(aes(color = as.factor(year))) +
  labs(title = "Season vs Change in Unemployment", ylabs = "Change in Unemployment (%)") +
  theme(legend.position = "none")
seasonal_plot


dataMatrix <- as.matrix(modelData[ , -1]) # convert to matrix, excluding date

pCor <- rcorr(dataMatrix, type = "pearson") # return pearson correlation and significance levels
pCor_matrix <- pCor$r # correlation matrix
pCor_sig <- pCor$P # significance levels

toRemove <-
  foreach(m = 2:ncol(pCor_matrix),  .combine = c) %do% {
    #remove junk (defined as correlation with another feature > 0.95
    #or significance of correlation with Y < .05)
  foreach(n = 2:nrow(pCor_matrix), .combine = c) %do% {
    # iterate over all rows and columns
    if (n != m){ # if not on self
      if(pCor_matrix[n,m] > 0.95 | pCor_sig[1,m] > 0.05){ # test for "junk"
        colName <- rownames(pCor_matrix)[m] #find variable name
        colName
      }
    }
  }
```

```
}

corPlot_data <- readRDS("corPlot.RDS") # load data exported from featureEngineering script
names(corPlot_data) <- c("Rows Lost", "Correlation")
corPlot <- ggplot(data = corPlot_data) + # initiate ggplot
  geom_point(aes(x = `Rows Lost`, y = 1 - abs(Correlation), colour = Correlation, size = abs(Correlatio
  labs(title = "Rows Lost Vs Inverse Correlation") +
  theme(legend.position = "none")

corPlot

## Build test training split, normalize data.
testData <-
  modelData[modelData$Date > as.Date("2018-02-28"), c(-1)] # test Data all observations after 28 Feb 20
trainData <- modelData[modelData$Date <= as.Date("2018-02-28"), c(-1)] # train Data all other obs
results <- modelData[modelData$Date > as.Date("2018-02-28"), c(1,2)] # test response and date (for plot
trainAss <- modelData[modelData$Date <= as.Date("2018-02-28"), c(1,2)] # training response and date

mean <- apply(trainData[ , -1], 2, mean) # calculate mean for each variable
std <- apply(trainData[ , -1], 2, sd) # calculate SD for each variable

testPred = scale(testData[ ,-1], center = mean, scale = std) # scale test data and make predictor df
testResponse = testData[,1] # make response list

trainPred = scale(trainData[ ,-1], center = mean, scale = std) # scale training data and make pred df
trainResponse = trainData[, 1] # make train response df

## check data splits make sense
nrow(modelData)
nrow(trainPred)
length(trainResponse)
nrow(testPred)
length(testResponse)
## Save objects for use in various models
saveRDS(list(rawData, modelData, testData, trainData, results, trainAss, testPred,
             testResponse, trainPred, trainResponse, mean, std), "data.RDS")


###########################
## Set up for MARS Model ##
###########################
rm(list = ls()) # removes all variables
if(!is.null(dev.list())) dev.off() # clear plots
cat("\014") # clear console

library(ggplot2, quietly = TRUE)
library(earth, quietly = TRUE)
library(caret, quietly = TRUE)
library(vip, quietly = TRUE)
library(pdp, quietly = TRUE)
library(data.table, quietly = TRUE)

data <- readRDS(file = "data.RDS") # import model data prepared earlier and saved in RDS file
```

```
plotData_test <- as.data.frame(data[5]) # test response and dates
plotData_train <- as.data.frame(data[6]) # train response and dates
testPredictors <- as.matrix(as.data.frame(data[7])) # test predictors, matrix for tf
testResponse <- as.numeric(unlist(data[8])) # test response, numeric vec for tf
trainPredictors <- as.matrix(as.data.frame(data[9])) # training predictors, as matrix for tf
trainResponse <- as.numeric(unlist(data[10])) # training response, as matrix for tf

The MARS algorithm is however susceptible to over fitting due to its large degree of flexibility. (Nisb


modelLookup("earth")

set.seed(123) # set seed
MARS_TUNE <- train(x = trainPredictors, # use caret train to train the model on training predictors
                   y = trainResponse, # training response
                   method = "earth", # use the MARS algorithm
                   metric = "MAE", # initially assess the model using Mean Absolute Error
                   trControl = trainControl( method = "cv", # perform 10 fold cross validation
                                             number = 10),
                   tuneGrid = expand.grid(degree = 1:4, # tune for degree 1 --> 4
                                          nprune = seq(2,20,4) # tune over the range  --> 20
                   ))

summary(MARS_TUNE)
cat(" Hyperparameters of best Tune", "\n",
    "nprune = ", MARS_TUNE$bestTune$nprune, "\n",
    "degree = ", MARS_TUNE$bestTune$degree)

## Plot various accuracy measures vs folds to asses metrics
accPlot_data <- as.data.frame(MARS_TUNE$resample) # return accuracy for all folds of the best tune
accPlot_data <- melt(accPlot_data, id.vars = "Resample") # melt data for plotting
accPlot_data$fold <- 1:10 # add fold variable so numeric value for fold
accPlot_stats <- accPlot_data %>% group_by(variable) %>% summarise(Mean = mean(value), SD = sd(value))

accPlot <- ggplot(data = accPlot_data) + # plot accuracy Data
  geom_point(aes(x = fold, y = value, color = variable)) + # plot fold on x axis, value on y axis, colo
  geom_line(aes(x = fold, y = value, color = variable)) + # add line
  labs(title = " Accuracy Vs fold",
       y = "Score",
       x = "Fold") +
  geom_text(aes(x = 2, y = 0.92, label = paste("Mean - ", accPlot_stats$Mean[2], "\n", # add mean and S
                                             "SD - ", accPlot_stats$SD[2]))) +
  geom_text(aes(x = 2.5, y = 0.4, label = paste("Mean - ", accPlot_stats$Mean[1], "\n",
                                             "SD - ", accPlot_stats$SD[1]))) +
  geom_text(aes(x = 9, y = 0.15, label = paste("Mean - ", accPlot_stats$Mean[3], "\n",
                                               "SD - ", accPlot_stats$SD[3]))) +
  theme_light()
accPlot

## Plot results
marsCT_plot <- ggplot(MARS_TUNE) + # quick plot of MAE vs prune and degree
  labs(title = "Accuracy vs # Terms (prune) by Degree") + # add tittle
  theme_light()
```

```r
marsCT_plot

set.seed(123)
MARS_REFINED <- train(x = trainPredictors, # use caret train to train the model on training predictors
                      y = trainResponse, # training response
                      method = "earth", # use the MARS algorithm
                      metric = "Rsquared", # train using Rsquared
                      trControl = trainControl( method = "LOOCV", # use LOOCV to help reduce influence
                                                number = 1), # # to leave out
                      tuneGrid = expand.grid(degree = 1:2, # tune for degree 1 --> 2
                                             nprune = 15:21), # tune over the range 10 --> 25
)

summary(MARS_REFINED) # return summary of final model
cat(" Hyperparameters of best Tune", "\n", # text
    "nprune = ", MARS_REFINED$bestTune$nprune, "\n", # nprune hp
    "degree = ", MARS_REFINED$bestTune$degree) # degree hp

cat("Final Model Accuracy", "\n",
    "R-squared = ", MARS_TUNE$finalModel$rsq)

## Plot results
marsCT_plot <- ggplot(MARS_REFINED) + # quick plot of MAE vs prune and degree
  labs(title = "Accuracy vs # Terms (prune) by Degree") + # add tittle
  theme_light()
marsCT_plot
`
## Plot variable importance
vi_scores <- vi(MARS_REFINED$finalModel) # use vimp to calculate variable importance scores
varImp_plot <- ggplot(data = vi_scores, aes(x = Importance, y = Variable)) + # call ggplot
  geom_col() +
  labs(title = "Variable Importance for MARS model")
varImp_plot

## Plot predicted v actual
plotData_test$Predictions <- predict(MARS_REFINED$finalModel, testPredictors) # add model predictions fo
plotData_train$Predictions <- MARS_REFINED$finalModel$fitted.values # add model predictions for training
plotData_test$Split <- rep("Test", nrow(plotData_test)) # add note that these values are from test split
plotData_train$Split <- rep("Train", nrow(plotData_train)) # add note that these values are from trainin
timeSeriesData <- rbind(plotData_train, # combine all predictions into a single data frame
                        plotData_test)
timeSeriesData$Split <- as.factor(timeSeriesData$Split) # make Split a factor

ts_plot <- ggplot(data = timeSeriesData) + # use time series data to make plot
  geom_line(aes(x = Date, y = Y), colour = "blue") + # plot actual unemployment in blue
  geom_line(aes(x = Date, y = Predictions, color = Split)) +
  labs(title = "Actual vs Predicted Employment", y = "Unemployment (%)", x = "Date") + # add tittle and
  geom_text(aes(x = Date[60], y = 8.5, label = paste("Training MAE = ", MAE(plotData_train$Y, plotData_t
  geom_text(aes(x = Date[125], y = 4, label = paste("Test MAE = ", MAE(plotData_test$Y, plotData_test$Pr
  theme_light()
  ts_plot

build_model <- function(l, u, is, opt, loss, met, dr)
```

```
  { # function recieves # layers, # units, input shape, optimiser, loss and metrics
  ## define building blocks
  inputLayer <-
    layer_dense(units = u,
                activation = "relu",
                input_shape = c(is)) # define input layer with # features (columns) as the single dimens
  outputLayer <-
    layer_dense(units = 1) # define output layer with no activation function and a single output, suital
  hiddenLayer <- list( # define hidden layers as list of 16 dense layers with units u and activation "re
    #####
    layer_dense(unit = u, activation = "relu"),
    layer_dense(unit = u, activation = "relu"),
    layer_dense(unit = u, activation = "relu"),
  )
  #####
  ## build model
  model <- keras_model_sequential( # use keras_seqential to compile model
    name = paste("model_units-",u,"_layer-",l, sep = ""), # give model a meaningful name
    layers = c(inputLayer,  # conc input layer and output layer with user defined number of hidden layer
               hiddenLayer[1:l],
               outputLayer)
  )

  ## Compile Model
  model %>% compile(
    optimizer = opt,
    loss = loss,
    metrics = c(met)
  )

resultsGrid <-
  test_tune_grid( # call test tuen grid to build and test model with the following parameters
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(2,4,8,16), # define # layers to test
  units = c(4,8,16,20), # define # units to test
  batch = c(4,8,16,20), # batch size to trial
  e = 300, # define number of epochs (note callback is used so rarely will this number be achieved)
  delta = .001, # set delta for call back end training
  patience = 5, # set patience for call back end training
  dropout = 0, # regulations parameter, not used in this tune
  aim = 1 # set aim to return the results gris (ie train and Val MAE)

## define function to plot tuining grids
GRID_PLOT <- function(resultsGrid){ # function to plot results from nn tuning grid
  names(resultsGrid) <-
    c("Fold", "Layers", "Units",
      "Batch Size", "Drop Out Rate", "tg",
      "k", "MAE Train", "MAE Val", "# Epochs") # give training grid results meaning fulname

  ## Sumarise Data
  avgMAE <- resultsGrid %>% group_by(Layers, Units, `Batch Size`) %>%  # group by layer, unit and batch
    filter(Units != 4) %>% # from inspection 4 units was woefully inaccurate so omit at this step to mal
    summarize(`MAE Train` = mean(`MAE Train`),
              `MAE Val` = mean(`MAE Val`)) # calculate average (over the k folds) MAE for training and v
  avgMAE$Units <- as.factor(avgMAE$Units) # make Units a factor for plotting
  avgMAE$`Batch Size` <- as.factor(avgMAE$`Batch Size`) # make

  ## Create Plot
  grid_plot <- ggplot(data = avgMAE) +
    geom_point(aes(x = Layers, y = `MAE Train`, colour = Units)) + # add points for each value of units
    geom_line(aes(x = Layers, y = `MAE Train`, colour = Units)) + # add lines for each value of units
    geom_point(aes(x = Layers, y = `MAE Val`, colour = Units)) +  # repeat for validation MAE
    geom_line(aes(x = Layers, y = `MAE Val`, colour = Units), linetype = "dashed") +
    facet_wrap(as.factor(avgMAE$`Batch Size`)) + # facet wrap Batch size
    labs(title = "MAE vs Layers by Units and Batch Size", Y = "Mean Absolute Error (MAE)") +
    theme_classic()

  return(grid_plot)
}

## plot the results from the corse tuning grid
gridPlot_course <- GRID_PLOT(readRDS(file = "resultsGrid_NN_course_gloud.RDS"))
gridPlot_course

resultsGrid <-
  test_tune_grid( # call test tune grid to build and test model with the following parameters
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(8,9,10,11,12), # define # layers to test
  units = c(18,19,20,21), # define # units to test
  batch = c(2, 4), # batch size to trial
  e = 300, # define number of epochs (note  callback is used so rarley will this number be achieved)
  delta = .00001, # set delta for call back end training
  patience = 5, # set patience for call back end training
  dropout = 0, # regulations parameter, not used in this tune
  aim = 1 # set aim to return the results gris (ie train and Val MAE)

## plot the results from the fine tuining grid
gridPlot_fine <- GRID_PLOT(readRDS(file = "resultsGrid_NN_fine_gcloud.RDS"))
gridPlot_fine

resultsGrid <-
  test_tune_grid( # call test tune grid to build and test model with the following parameters
  model_builder = build_model_reg, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12,24,36), # freeze layers from previous investigation
  units = c(20), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 500, # define number of epochs (note --> callback is used so rarley will this number be achieved)
```

```r
  delta = .00000005, # set delta for call back end training
  patience = 10, # set patience for call back end training
  dropout = c(0.2,0.4,0.6), # regulations parameter
  aim = 1 # set aim to return the results gris (ie train and Val MAE)
)

## plot results for regulization
resultsGrid<- as.data.frame(readRDS(file = "resultsGrid_NN_reg_gcloud.RDS")) # import results from RDS
names(resultsGrid) <- c("Fold", "Layers", "Units",
                        "Batch Size", "Drop Out Rate",
                        "tg", "k", "MAE Train", "MAE Val", "# Epochs") # give training grid results mea
resultsGrid$`Drop Out Rate` <- as.factor(resultsGrid$`Drop Out Rate`) # make dr a factor

avgMAE <- resultsGrid %>% group_by(Layers, `Drop Out Rate`) %>%  # group by layer, unit and drop out ra
  summarize(`MAE Train` = mean(`MAE Train`),
            `MAE Val` = mean(`MAE Val`), epoch = median(`# Epochs`)) # calculate average (over the k fo

reg_plot <- ggplot(data = avgMAE) + # Create Plot
  geom_point(aes(x = Layers, y = `MAE Train`, color = `Drop Out Rate`)) +
  geom_line(aes(x = Layers, y = `MAE Train`, color = `Drop Out Rate`)) +
  geom_point(aes(x = Layers, y = `MAE Val`, color = `Drop Out Rate`)) +
  geom_line(aes(x = Layers, y = `MAE Val`, color = `Drop Out Rate`), linetype = "dashed") +
  geom_text(aes(x = avgMAE$Layers, y = 1.6, label = paste("Median # epochs - ", "\n", median(avgMAE$epo
  labs(title = "MAE vs Layers for various drop out rates", Y = "Mean Absolute Error (MAE)") +

  theme_light()
reg_plot

## No reg
history_21_12 <-
  test_tune_grid( # call test tune grid to build and test model with the following parameters
  model_builder = build_model, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12), # freeze layers from previous investigation
  units = c(21), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 500, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = 0, # set delta for call back end training
  patience = 500, # effectively turn off stop early
  dropout = c(0), # regulations parameter
  aim = 2 # set aim to return the results gris (ie train and Val MAE)

## Reg
history_20_12_reg <-
  test_tune_grid( # call test tune grid to build and test model with the following parameters
  model_builder = build_model_reg, # use the model builder without regulisation
  trainPredictors = trainPredictors, # pass all training predictors
  trainResponse = trainResponse, # pass all training responses
  k = 4, # Define # folds for k folds cross validation
  layers = c(12), # freeze layers from previous investigation
  units = c(20), # freeze units from previous investigation
  batch = c(2), # freeze batch from previous investigation
  e = 150, # define number of epochs (note --> callback is used so rarley will this number be achieved)
  delta = 0, # set delta for call back end training
  patience = 50, # effectively turn off stop early
  dropout = c(0.2), # regulations parameter
  aim = 2 # set aim to return the results gris (ie train and Val MAE)
)

# plot mae vs epoch for reg and no reg
HIST_TO_PLOT <- function(history, reg){
historyVal <- history[[1]] # unlist validation and training MAE
historyTrain <- history[[2]]
plotData <- data.frame(epochs = seq(1:ncol(historyVal)), # create epochs variable as seq to end of mae
                       regulisation = rep(reg, ncol(historyVal)), # add note that these are unregulised
                       Validation = apply(historyVal, 2, mean), # average validation MAE over all folds
                       Training = apply(historyTrain, 2, mean)) # average training MAE over all folds
                        plotData <- plotData %>% filter(epochs <= 250)
}

plotData <- rbind( # Build plot Data Data frame by combining both regulated and uin regulated model his
  HIST_TO_PLOT( # call function to average valdiation and training MAE's
    readRDS(file = "history_21_12.RDS"), # import list of training and validation MAE's
    "Without Regulation"), # specify no regulation
  HIST_TO_PLOT( # call function to average over folds
    readRDS(file = "history_20_12_reg_02.RDS"), # read regulated model history
            "With Dropout Regulation") # Specify Regulated
)
plotData <- melt(plotData, id.vars = c("epochs", "regulisation")) # melt Data for plotting
plotData$regulisation <- as.factor(plotData$regulisation)


epochs_plot_comb <- ggplot(plotData, aes(x = epochs, y = value, colour = variable)) + # plot with GG
  geom_smooth() + # add smoothing
  facet_wrap(plotData$regulisation) + # facet wrap regulisation
  theme_classic() +
  labs(title = "MAE Vs # Epochs for net with and without regulisation",
       y = "Mean Absolute error (MAE)",
       x = "# Epochs")

epochs_plot_comb

model_final <- build_model( # use the model builder without regulisation to build model
  l = 12, # build model with 12 layers
  u = 20,  # build model with 20 units
  is = ncol(trainPredictors), # input shape
  opt = "adam", # optimiser
  loss = "mse", # loss function
  met = "mae", # metrics
  dr = NA # dropout is NA as regulisation is not being used
)

## Fit Model and story history
history <- model_final %>% fit( # fit model and record results in history
```

```r
  trainPredictors, # training predictors
  trainResponse, # training response
  epochs = 75, # epochs (predefined for easy adjustment)
  batch_size = 2 # batch size from tuning grid
  #callbacks = callbacks # predefined call backs
)

## return test and training data metrics
resultsTest <- model_final %>% evaluate(testPredictors, testResponse) # calculate fit metrics on test d
resultsTraining <- model_final %>% evaluate(trainPredictors, trainResponse) # calculate fit on training

## Predict unemployment using the test and training predictors
plotData_test$Predictions <- model_final %>% predict(testPredictors) %>% as.numeric() # return test pre
plotData_train$Predictions <- model_final %>% predict(trainPredictors) %>% as.numeric() # return traini
plotData_test$Split <- rep("Test", nrow(plotData_test)) # add note that these values are from test spli
plotData_train$Split <- rep("Train", nrow(plotData_train)) # add note that these values are from traini

timeSeriesData <- rbind(plotData_train, # combine all predictions into a single dataframe
                        plotData_test)

finalResults <- list(resultsTest, # list all relevant results for export
                     resultsTraining,
                     timeSeriesData)

finalResults <- readRDS(file = "finalResults_gcloud.RDS") # import final results RDS
MAEtest <- finalResults[[1]] # extract test MAE
MAEtrain <- finalResults[[2]] # extract training MAE
timeSeries <- finalResults[[3]] # extract time series predictions
timeSeries$Split <- as.factor(timeSeries$Split) # make Split a factor

ts_plot <- ggplot(data = timeSeries) + # use time series data to make plot
  geom_line(aes(x = Date, y = Y), colour = "blue") + # plot actual unemployment in blue
  geom_line(aes(x = Date, y = Predictions, color = Split)) +
  labs(title = "Actual vs Predicted Employment", y = "Unemployment (%)", x = "Date")+
  theme_light()
ts_plot
```