Mahmoud Abu Ghzalah
Machine Learning in Economics Project

# Bank Marketing Data

## Introduction:

This paper will investigate the features that affect a consumer's response to bank telemarketing campaigns. The results of such research can inform bank decision making regarding how best to operate marketing campaigns by targeting the demographic and socioeconomic groups with the highest predicted positive response to the campaign. Inference based on the day of the week and month in which contact was made in addition to how contact was made (eg. by telephone) can provide banks and other telemarketers with information regarding how best to structure their telemarketing operations. Furthermore, there are potential public policy implications given the inclusion of macroeconomic variables. Results may provide some information that can be generalized to the broader economy regarding consumer propensities to save in response to variations in the macroeconomy. The results of the study may provide some insight into how consumers of different demographics make their savings decisions. Moreover, it may provide some indication of how best to target less advantaged groups that are not well integrated in the financial system.

## Description of Data:

The data used in this paper was collected over a period of 5 years, from 2008 to 2013 by a Portuguese retail bank that conducted a telemarketing campaign that attempted to get customers to subscribe to a term deposit. Because some of the data was collected during and immediately after the Great Financial Crisis, there may be structural differences in the data as time progressed. However, I was unable to find a version of the dataset that included time as a variable, although one presumably does exist. Although, I do not make use of the time component in this paper, it could prove to be a fruitful avenue for further analysis.

The response variable in this study is a binary yes/no. Yes if a customer makes a bank term deposit and no if a customer declines to make a bank term deposit. Various individual specific features and macroeconomic variables are included in the dataset. Information about customer's existing bank balances, demographic information, employment information, basic information about existing debt obligations, etc. was collected. Some customers were contacted multiple times by the bank before a response. The dataset is unbalanced with only 5289 positive responses (11.7%) and 39921 negative responses. Many of the features are categorical variables and one hot encoding was used to incorporate them into the models. Portuguese macroeconomic data from the period is also used, with the model incorporating the level of the macroeconomic variables at the point in time each observation was recorded. The macroeconomic variables include the consumer price index, consumer confidence index, the daily Euribor three-month rate, and a quarterly measure of employment.

Mahmoud Abu Ghzalah
Machine Learning in Economics Project

The variable 'duration' which refers to the duration in seconds of the last contact one of the bank's telemarketers had with a customer is highly problematic. If the duration for an observation is zero then the response variable will necessarily be a no because the customer would not have been offered the term deposit to begin with. Duration also cannot be known before a call is made, so using it in a predictive model may be suboptimal, as it puts the cart before the horse in a sense. Consequently, for each of the approaches that I will consider, I will consider two variations of the dataset, one that includes duration and one that omits it. Because the response variable is binary, the models I use will be binary classification models, which conditional on observed features assign a probability that the response variable will be positive.

## Approach 1:

### Neural Network:

The first approach I used to predict the probability of a customer subscribing to a term deposit was a neural network. Given the large number of variables and large dataset using a neural network allows for automated extraction of relevant information from the data, and potentially enables modeling of nonlinear relationships among the features. From a business standpoint, the predictive power of neural networks may make this type of model the best choice to use if one were only concerned with accuracy of predictions, however, because the model is somewhat of a black box, inference cannot really be done using this type of model. Therefore, if a bank is trying to better understand its customer base, using a neural network may not be the best approach.

Francois Chollet in his book Deep Learning with R suggests that stacked densely connected layers perform well for classification. I opted to use two densely connected hidden layers each with 100 hidden units and a final densely connected layer with two hidden units. The loss function that I specified was binary cross entropy because the problem is one of binary classification. I standardized the test and training data sets by normalizing each feature using the mean and standard deviation of that feature in the training data set. This normalization ensures that inputs are treated equally by the optimization algorithm and make the learning process smoother as the data does not need to adapt to large heterogeneous differences. Also, data taking on a large range of values across different features may lead to large gradient changes in the optimization routine that would prevent convergence.

I decided to use a relatively state of the art optimization algorithm, RMSProp. RMSProp improves upon stochastic gradient descent by allowing for the learning rate of the model to be dynamically updated. RMSProp does this by keeping a moving average of the squared gradients for each weight. Because it is a moving average, the distant past is 'discarded' and prevents RMSProp from getting stuck at saddle points in the error surface. For the activation functions, which add non-linearity to the model, I opted to use rectified linear units (ReLUs). ReLUs preserve some linearity because they are piecewise linear, and do not suffer from disappearing gradients which happens with deep networks using sigmoid activation functions. The final layer is a densely connected layer with 2 units one corresponding to each class and a softmax activation function which for each observation and every class attaches a probability to an observation being in a specific class.

Mahmoud Abu Ghzalah
Machine Learning in Economics Project

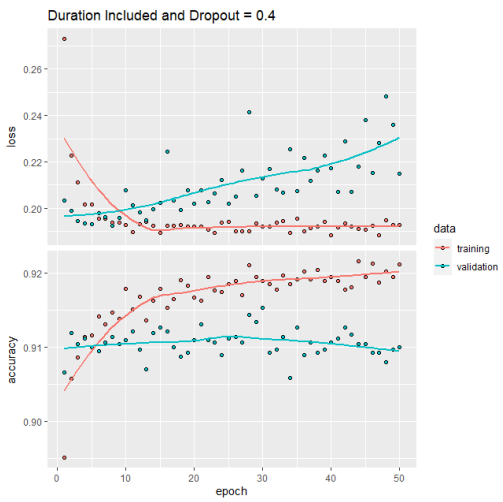
I considered two versions of the model, one with and one without dropout. Dropout is a regularization technique that randomly drops some proportion 'a' of the hidden units in each layer at random adding noise to the hidden layers. Dropout effectively amounts to training many neural networks in parallel and averaging over them which should help prevent overfitting.

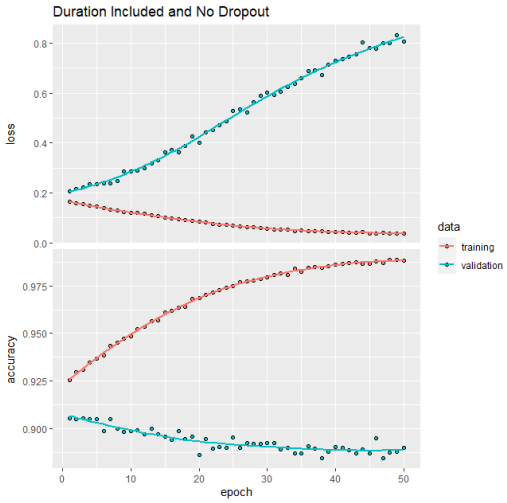
I randomly assigned half the observations in the dataset to the training set and the other half to the test set. In my models I then further divided the training data into a training set and a validation set, with the validation set being one fifth of the training set. This validation set allows for an estimate of the test error to be generated while the model is being fit and can help prevent overfitting, if for example the optimization algorithm is stopped once the validation error starts to increase.

**Results:**

I found that for all models, the validation error was minimized after just one epoch, this implies that just one epoch was enough for the model to begin overfitting the training data. Consequently, the optimal model involves just one epoch. The training and validation loss and accuracy functions as epochs vary illustrate this

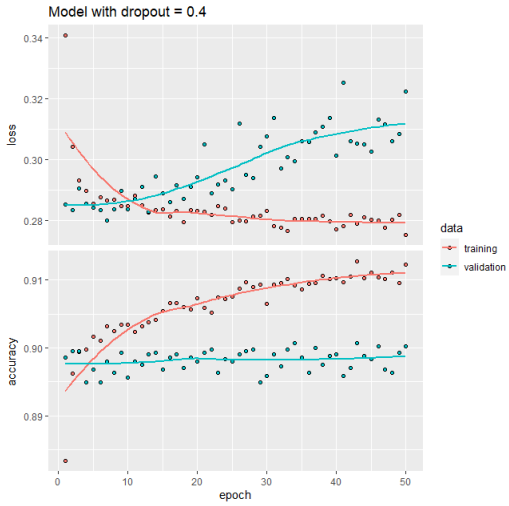
Graphs of Model with duration:

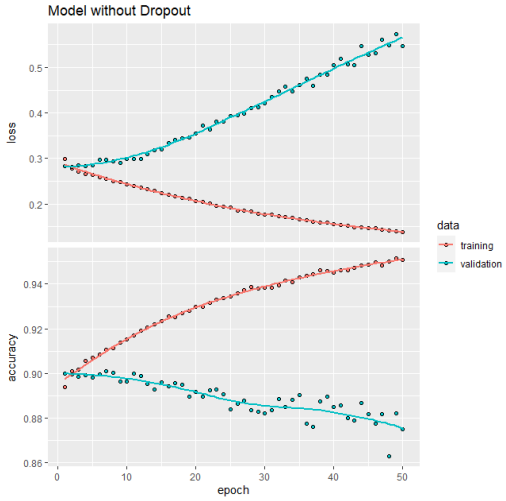


Mahmoud Abu Ghzalah
Machine Learning in Economics Project

Graphs of model without duration:




For models with one epoch that did not include the feature 'duration': the test accuracy was 0.89 for models without dropout and 0.908 for models with dropout set to 0.4. For models with one epoch that did include duration: the test accuracy was 0.907 without dropout and 0.908 with dropout set to 0.4. We can see from these results that dropout yields better out of sample performance regardless of whether duration is included in the dataset. The best model is the one fit on data that does not include duration and incorporates dropout. One way this model could be refined is via tuning the level of dropout and by further tuning the number of hidden layers and hidden units. Limited tuning I conducted suggested that the test error rate has practically converged, even for this relatively simple network structure.

## Approach 2:

### Boosted Tree:

The second approach I used was to model the classification problem using a boosted tree. One advantage of using a boosted tree over a neural network is that the boosted tree is somewhat more interpretable. Because we use recursive binary splitting to determine predictor – cut point pairs, predictors that best segment and classify the response variable will generally be contained in the first few splits. This allows for a ranking of the predictors by their importance/relevance to classification of the response variable. Moreover, because the structure of the eventual model is easier to interpret, the bank and public policy makers can more easily incorporate these results into their strategic decision making when compared to a model like a neural network. However, a boosted tree is still not as interpretable as a simple classification tree because in a boosted tree multiple models are fit and

'averaged', so there are no clear delineations of split-cut point pairs. However, I chose a boosted decision tree over a classical decision tree because boosted decision trees tend to perform better on out of sample prediction than classical decision trees, even at the cost of reduced interpretability. Boosted trees perform better at prediction because the model learns slowly and is asked to learn the residuals, which is where previous trees have failed, and hence we iteratively improve the estimated model in areas it performs poorly. One drawback of using a boosted tree is that the method is computationally demanding and learns slowly compared to other methods.

Boosted trees work by first fitting a tree to the data, and then sequentially fitting new trees on the residuals of the previous tree. I used the package xgboost in R to conduct the analysis. There are several hyperparameters that need to be chosen and tuned. The number of trees is a hyperparameter that will need to be cross-validated, if the number of trees is too high, there is a risk that the model will overfit the data. The number of splits in each tree will also need to be chosen, with more complex trees demanding greater computational power. A boosted tree model with many trees, and with each tree having many splits imposes high computational demands. However, because each tree considers all the trees that were grown before it, trees with many splits are typically not needed. The learning rate of the model controls how quickly the model learns and is generally set to a low level to reduce the probability of overfitting. Finally, the xgboost package provides control over what proportion of the training set you wish to include in each iteration of the optimization, which amounts to implementing stochastic gradient descent if you choose a proportion less than one. This is a parameter that can also be tuned.

The xgboost package in R supports cross validation through specification of the argument 'cv.folds'. I have chosen to use 5-fold cross validation, which involves dividing the training data into five subsets (folds), and then holding one of these sets out as a validation set whilst training the model on the remaining folds. The loss function is calculated for each fold, and then an average of the loss function is taken over the five folds to estimate the test error. In the grid search routine, I specified that the optimization algorithm stop if there were no improvements to the cross validated error after 10 iterations. This is a regularization technique that helps prevent overfitting, as the training error will always monotonically decrease with increased model complexity.

I tuned the hyperparameters by iterating over a grid of combinations of potential hyperparameter values and then choosing the model with hyperparameters that result in the lowest 5-fold cross validation error. If I had access to greater computational power, I would have tuned the hyperparameters over a finer grid but given computational limitations I limited my choice of possible hyperparameter values. For the model with the 'duration' feature excluded, the optimal hyperparameter values were 0.1 for the learning rate, a maximum depth of 4 each model, and a dropout of 0.65 with a corresponding validation error of 0.099. The test error rate for this model was 0.101. The importance plot reveals that the number of people employed nationally, the Euribor three-month rate, and the employment rate are the most important features when predicting the outcome of the response variable.

The model that includes duration has optimal hyperparameters of 0.2 for the learning rate, a maximum number of splits of 4, dropout of 0.65 and 66 as the optimal number of trees with a corresponding validation error of 0.084. The test error rate of this model was also 0.084, and the importance plot reveals that the three most important features are duration, the number of people employed, and the three-month Euribor rate. This makes sense based on economic theory, the

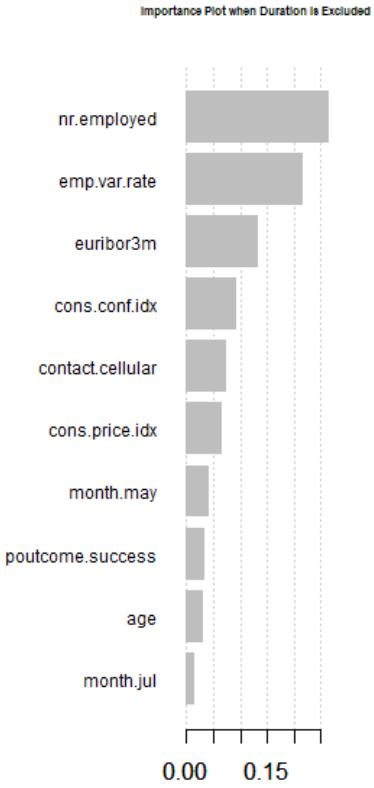
relationship between interest rates and savings is well established. Further, consumer confidence is closely linked to marginal propensity to save and the notion of precautionary savings. Given duration skews results, and because macroeconomic variables are the most important features in this model, I ran a third model in which macroeconomic variables and the 'duration' feature were excluded. From the firm's perspective, they take macroeconomic variables as exogenous, although they may have some optimal response to macroeconomic policy, I hoped to isolate the demographic and socioeconomic features of its customer base that were the most relevant.

In this third model the macroeconomic variables and the variable 'duration' were omitted. The optimal hyperparameter choices were a learning rate of 0.2, a maximum depth of 4, and dropout of 0.8 with a corresponding validation error of 0.101, higher than in the other two models. The test error rate of this model is 0.103 which is again higher than the other models. The importance plot reveals that the most important features out of the remaining variables are: that contact was made via cellular phone, that the outcome of a previous marketing campaign targeting that customer was successful, and customer age.

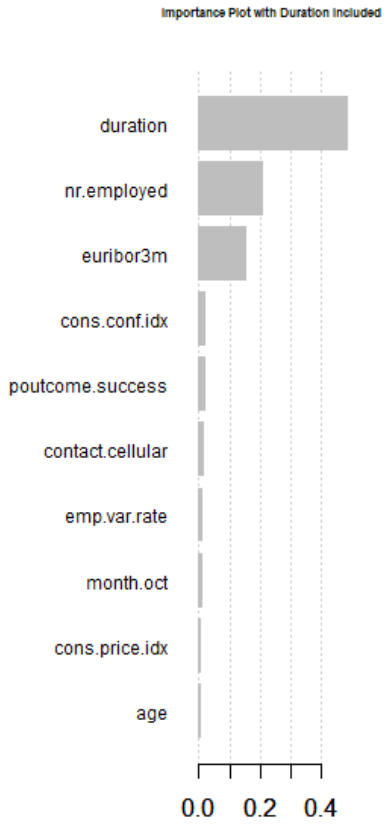
Overall, not many of the individual specific characteristics seem to be particularly important. In particular, temporal variables had little effect on the probability of a positive response, implying that the bank should not be concerned with when a telemarketer reaches out to a customer. Furthermore, type of employment and existing credit status also don't seem to be particularly important, suggesting that perhaps the banks customers are less heterogeneous in their preferences regarding savings than the data would initially suggest.

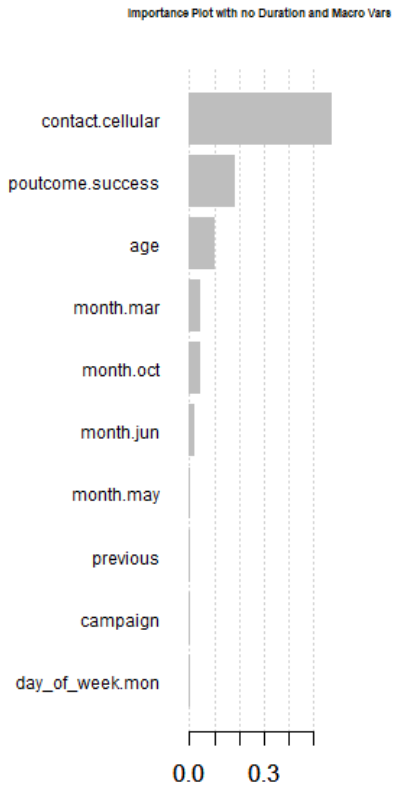
It is also interesting to note that neural networks which did not include duration performed slightly better out of sample than boosted trees that did not include duration, this suggests that if the bank were purely concerned with accurately predicting a customer's response to their customer's telemarketing campaign they should use the neural network over the boosted tree.

Mahmoud Abu Ghzalah
Machine Learning in Economics Project



Mahmoud Abu Ghzalah
Machine Learning in Economics Project






# Appendix

The data was sourced from: *https://archive.ics.uci.edu/ml/datasets/Bank+Marketing*

## Code for Neural Network where the feature 'duration' is omitted

```
# Load required packages

library(tidyr)
library(ISLR)
library(gbm)
library(dplyr)
library(caret)
library(xgboost)
library(keras)
library(tensorflow)


install_tensorflow(version = 'gpu')
```

```
# Change working directory

setwd('C:/Users/abugh/Documents/R/Machine Learning Project/BankData')

# Load in data

bankData <-  read.csv('bank-additional-full.csv', sep = ';')

# Code for model where the feature duration is not included

# Filter out rows with missing observations

bankData <- bankData %>% drop_na()

bankData <- bankData %>% select(-c('duration'))



# Encode categorical variables as n-1 dummy variables

dummy <- dummyVars(" ~ job + marital + education + default + housing +
loan + contact + month + poutcome + y + day_of_week ", data = bankData
)

dummy_pred = data.frame(predict(dummy, newdata = bankData))

bankData = cbind(bankData %>% select(-c('job', "marital" , "education"
, "default" , "housing" , "loan" , "contact" , "month" , "poutcome", '
day_of_week')), dummy_pred)

# Encode our target variable y as a binary variable that takes on a va
lue of 0 when the target is 'no', and 1 when the target is 'yes

bankData$y.yes = (bankData$y.no * -1) + 1

bankData <- bankData %>% select(-c(y, y.no, pdays))

# Set seed for reproducibility

set.seed(1)

# Split the data into test and train sets, with 1/2 of all observation
s in the training set and half the observations in the test test

train_index <- sample(1:nrow(bankData), nrow(bankData)/2)

train <- bankData[train_index,]
```

```
test <- bankData[-train_index,]

# Generate the train and test labels for the response variable

train_labels = train$y.yes
test_labels = test$y.yes

# Remove response variable from train and test arrays

train <- train %>% select(-c(y.yes))
test <- test %>% select(-c(y.yes))


# Recast the training and test labels as categorical variables

train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)

# Reshape the training labels so they can be used in the neural networ
k

# train_labels <- keras:: array_reshape(train_labels, c(20594,1))
# test_labels <- keras:: array_reshape(test_labels, c(20594,1))



# Rescale data using the mean and standard deviation of the training d
ata

mean <- apply(train, 2, mean)
std <- apply(train, 2, sd)

train <-  scale(train, center = mean, scale = std)
test <- scale(test, center = mean, scale = std)

# Reshape the data into the expected tensor shape for the specified ne
twork

train <- keras::array_reshape(train, c(20594, 61))
test <- keras::array_reshape(test, c(20594, 61))


# metrics <- network %>% evaluate(test, test_labels)


build_model <- function(dropout = 0) {
```

```
  model <- keras::keras_model_sequential()

  model %>%
    layer_dense(units = 100, activation = 'relu', input_shape = ncol(t
rain)) %>%
    layer_dropout(dropout) %>%
    layer_dense(units = 100, activation = 'relu') %>%
    layer_dropout(dropout) %>%
    layer_dense(units = 2, activation = 'softmax')

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

epochs <-  50

model <- build_model()

# Print a dot for every completed epoch to track progress

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs){
    if (epoch %% 80 == 0 ) cat("\n")
    cat(".")
  }
)

# Run the model

fitted_model <- model %>% fit(train, train_labels,
                              epochs = epochs,
                              validation_split = 0.2,
                              verbose = 0,
                              callbacks = list(print_dot_callback)

)

# Visualize the training progress

plot(fitted_model)

# Evaluate the model
```

```
model %>% evaluate(test, test_labels, verbose = 0)


# Try and use the model with dropout = 0.4

model_dropout <- build_model(0.4)

fitted_model_dropout <- model_dropout %>% fit(train, train_labels,
                                              epochs = epochs,
                                              validation_split = 0.2,
                                              verbose = 0,
                                              callbacks = list(print_dot_callb
ack)

)

plot(fitted_model_dropout)

model_dropout %>% evaluate(test, test_labels, verbose = 0)

# Run no dropout model with just one epoch

epochs <- 1


fitted_model_final <- model %>% fit(train, train_labels,
                                                    epochs = epochs,
                                                    validation_split =
0.2,
                                                    verbose = 0,
                                                    callbacks = list(p
rint_dot_callback)

)

model %>% evaluate(test, test_labels, verbose = 0)

# Run dropout model with one epoch

fitted_model_dropout_final <- model_dropout %>% fit(train, train_label
s,
                                                    epochs = epochs,
                                                    validation_split =
0.2,
                                                    verbose = 0,
                                                    callbacks = list(p
rint_dot_callback)
```

```
)

model_dropout %>% evaluate(test, test_labels, verbose = 0)
```

## Code for Neural Network Where the feature 'duration' is included:

```
# Load required packages

library(tidyr)
library(ISLR)
library(gbm)
library(dplyr)
library(caret)
library(xgboost)
library(keras)
library(tensorflow)


install_tensorflow(version = 'gpu')

# Change working directory

setwd('C:/Users/abugh/Documents/R/Machine Learning Project/BankData')

# Load in data

bankData <-  read.csv('bank-additional-full.csv', sep = ';')


# Filter out rows with missing observations

bankData <- bankData %>% drop_na()



# Encode categorical variables as n-1 dummy variables

dummy <- dummyVars(" ~ job + marital + education + default + housing +
loan + contact + month + poutcome + y + day_of_week ", data = bankData
)

dummy_pred = data.frame(predict(dummy, newdata = bankData))

bankData = cbind(bankData %>% select(-c('job', "marital" , "education"
, "default" , "housing" , "loan" , "contact" , "month" , "poutcome", '
day_of_week')), dummy_pred)

# Encode our target variable y as a binary variable that takes on a va
```

```
lue of 0 when the target is 'no', and 1 when the target is 'yes

bankData$y.yes = (bankData$y.no * -1) + 1

bankData <- bankData %>% select(-c(y, y.no, pdays))

# Set seed for reproducibility

set.seed(1)

# Split the data into test and train sets, with 1/2 of all observation
s in the training set and half the observations in the test test

train_index <- sample(1:nrow(bankData), nrow(bankData)/2)

train <- bankData[train_index,]
test <- bankData[-train_index,]

# Generate the train and test labels for the response variable

train_labels = train$y.yes
test_labels = test$y.yes

# Remove response variable from train and test arrays

train <- train %>% select(-c(y.yes))
test <- test %>% select(-c(y.yes))


# Recast the training and test labels as categorical variables

train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)



# Rescale data using the mean and standard deviation of the training d
ata

mean <- apply(train, 2, mean)
std <- apply(train, 2, sd)

train <-  scale(train, center = mean, scale = std)
test <- scale(test, center = mean, scale = std)

# Reshape the data into the expected tensor shape for the specified ne
twork
```

```
train <- keras::array_reshape(train, c(20594, 62))
test <- keras::array_reshape(test, c(20594, 62))


# Specify a model as a function allowing the dropout parameter to be c
hanged more easily
# Model consists of two hidden layers with potential dropout between e
ach layer, activation functions are ReLU
# Final layer uses a softmax function to assign a probability

build_model <- function(dropout = 0) {

  model <- keras::keras_model_sequential()

  model %>%
    layer_dense(units = 100, activation = 'relu', input_shape = ncol(t
rain)) %>%
    layer_dropout(dropout) %>%
    layer_dense(units = 100, activation = 'relu') %>%
    layer_dropout(dropout) %>%
    layer_dense(units = 2, activation = 'softmax')

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

epochs <-  50

model <- build_model()

# Print a dot for every completed epoch to track progress

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs){
    if (epoch %% 80 == 0 ) cat("\n")
    cat(".")
  }
)

# Run the model

fitted_model <- model %>% fit(train, train_labels,
```

```
                                epochs = epochs,
                                validation_split = 0.2,
                                verbose = 0,
                                callbacks = list(print_dot_callback)

)

# Visualize the training progress

plot(fitted_model) + ggtitle('Duration Included and No Dropout')

# Evaluate the model

model %>% evaluate(test, test_labels, verbose = 0)


# Try and use the model with dropout = 0.4

model_dropout <- build_model(0.4)

fitted_model_dropout <- model_dropout %>% fit(train, train_labels,
                                              epochs = epochs,
                                              validation_split = 0.2,
                                              verbose = 0,
                                              callbacks = list(print_d
ot_callback)

)

plot(fitted_model_dropout) + ggtitle('Duration Included and Dropout =
0.4')


# Run no dropout model with one epoch

epochs <- 1


fitted_model_final <- model %>% fit(train, train_labels,
                                    epochs = epochs,
                                    validation_split = 0.2,
                                    verbose = 0,
                                    callbacks = list(print_dot_callbac
k)

)
```

```
model %>% evaluate(test, test_labels, verbose = 0)

# Run dropout model with one epoch

fitted_model_dropout_final <- model_dropout %>% fit(train, train_label
s,
                                                    epochs = epochs,
                                                    validation_split =
0.2,
                                                    verbose = 0,
                                                    callbacks = list(p
rint_dot_callback)
)

model_dropout %>% evaluate(test, test_labels, verbose = 0)
```

## Code for Boosted Tree where 'duration' and macro variables are included

```
# Load required packages

library(tidyr)
library(ISLR)
library(gbm)
library(dplyr)
library(caret)
library(xgboost)

# Change working directory

setwd('C:/Users/abugh/Documents/R/Machine Learning Project/BankData')

# Now consider the model where the feature 'duration' is included

bankData <- bankData <-  read.csv('bank-additional-full.csv', sep = ';
')

# Encode categorical variables as n-1 dummy variables

dummy <- dummyVars(" ~ job + marital + education + default + housing +
loan + contact + month + poutcome + y + day_of_week ", data = bankData
)

dummy_pred = data.frame(predict(dummy, newdata = bankData))

bankData = cbind(bankData %>% select(-c('job', "marital" , "education"
, "default" , "housing" , "loan" , "contact" , "month" , "poutcome", '
day_of_week')), dummy_pred)
```

```
# Encode our target variable y as a binary variable that takes on a va
lue of 0 when the target is 'no', and 1 when the target is 'yes

bankData$y.yes = (bankData$y.no * -1) + 1

bankData <- bankData %>% select(-c(y, y.no, pdays))

# Set seed for reproducibility

set.seed(1)

# Split the data into test and train sets, with 1/2 of all observation
s in the training set and half the observations in the test test

train_index <- sample(1:nrow(bankData), nrow(bankData)/2)

train_with_dur <- bankData[train_index,]
test_with_dur <- bankData[-train_index,]

# Generate the train and test labels for the response variable

train_labels = train_with_dur$y.yes
test_labels = test_with_dur$y.yes

# Convert matrix into a sparse matrix that can be used with xgboost

train_dur_final <- Matrix:: sparse.model.matrix(y.yes~.-1,data = train
_with_dur)
test_dur_final <- Matrix:: sparse.model.matrix(y.yes~.-1,data = test_w
ith_dur)

# Use same hyperparameter grid as before

hyperparameter_grid <- expand.grid(
  eta = c(0.001, 0.01, 0.1,0.2),
  max_depth = c(1,2,3,4),
  subsample = c(.65, .8, 1),
  optimal_trees = 0,
  min_error = 0
)

# Create a for loop to iterate over hyperparameter grid and fit differ
ent models, then populate optimal trees and min error columns of
# hyperparameter grid. Early stopping is set to 10, which means if the
re is no improvement to error after 10 iterations the optimization sto
ps
```

```
for(i in 1:nrow(hyperparameter_grid)) {

  # create parameter list
  params <- list(
    eta = hyperparameter_grid$eta[i],
    max_depth = hyperparameter_grid$max_depth[i],
    subsample = hyperparameter_grid$subsample[i]
  )

  set.seed(1)

  # train model
  xgb_tune <- xgb.cv(
    params = params,
    data = train_dur_final,
    label = train_labels,
    nrounds = 5000,
    nfold = 5,
    objective = "binary:hinge",
    verbose = 0,
    early_stopping_rounds = 10
  )

  # add min training error and optimal number of trees to the hyperpar
ameter grid dataframe, test error corresponds to validation error

  hyperparameter_grid$optimal_trees[i] <- which.min(xgb_tune$evaluatio
n_log$test_error_mean)
  hyperparameter_grid$min_error[i] <- min(xgb_tune$evaluation_log$test
_error_mean)
}

# Sort hyperparameter grid by validation error to find the optimal sel
ection of hyperparameters

hyperparameter_grid %>% arrange(min_error)

# Generate params list corresponding to optimal hyperparameters

params <- list(eta = 0.2, max_depth = 4, subsample = 0.65)

# Fit model with optimal parameters

xgb_fit_final_dur <- xgboost(
  params = params,
  data = train_dur_final,
  label = train_labels,
```

```
  nrounds = 66,
  objective = 'binary:hinge',
  verbose = 0)

# Find the test error rate of the chosen model

predictions <- predict(xgb_fit_final_dur, test_dur_final)

test_error <- mean(predictions != test_with_dur$y.yes)

# Plot a graph showing variable importance

importance_matrix <- xgb.importance(model = xgb_fit_final_dur)

xgb.plot.importance(importance_matrix, top_n = 10, measure = "Gain", m
ain = 'Importance Plot with Duration Included', cex.main = 0.5)
```

## Code for model where 'duration' is excluded

```
# Load required packages

library(tidyr)
library(ISLR)
library(gbm)
library(dplyr)
library(caret)
library(xgboost)

# Change working directory

setwd('C:/Users/abugh/Documents/R/Machine Learning Project/BankData')

# Load in data

# For the model without the feature duration

bankData <-  read.csv('bank-additional-full.csv', sep = ';')


bankData <- bankData %>% drop_na()

bankData <- bankData %>% select(-c('duration'))



# Encode categorical variables as n-1 dummy variables

dummy <- dummyVars(" ~ job + marital + education + default + housing +
```

```
loan + contact + month + poutcome + y + day_of_week ", data = bankData
)

dummy_pred = data.frame(predict(dummy, newdata = bankData))

bankData = cbind(bankData %>% select(-c('job', "marital" , "education"
, "default" , "housing" , "loan" , "contact" , "month" , "poutcome", '
day_of_week')), dummy_pred)

# Encode our target variable y as a binary variable that takes on a va
lue of 0 when the target is 'no', and 1 when the target is 'yes

bankData$y.yes = (bankData$y.no * -1) + 1

bankData <- bankData %>% select(-c(y, y.no, pdays))

# Set seed for reproducibility

set.seed(1)

# Split the data into test and train sets, with 1/2 of all observation
s in the training set and half the observations in the test test

train_index <- sample(1:nrow(bankData), nrow(bankData)/2)

train <- bankData[train_index,]
test <- bankData[-train_index,]

# Generate the train and test labels for the response variable

train_labels = train$y.yes
test_labels = test$y.yes


# Convert matrix into a sparse matrix that can be used with xgboost

train_final <- Matrix:: sparse.model.matrix(y.yes~.-1,data = train)
test_final <- Matrix:: sparse.model.matrix(y.yes~.-1,data = test)

# Create grid of hyperparameters that need to be tuned

hyperparameter_grid <- expand.grid(
  eta = c(0.001, 0.01, 0.1,0.2),
  max_depth = c(1,2,3,4),
  subsample = c(.65, .8, 1),
  optimal_trees = 0,
  min_error = 0
```

```
)

# Create a for loop to iterate over hyperparameter grid and fit differ
ent models, then populate optimal trees and min error columns of
# hyperparameter grid. Early stopping is set to 10, which means if the
re is no improvement to error after 10 iterations the optimization sto
ps

for(i in 1:nrow(hyperparameter_grid)) {

  # create parameter list
  params <- list(
    eta = hyperparameter_grid$eta[i],
    max_depth = hyperparameter_grid$max_depth[i],
    subsample = hyperparameter_grid$subsample[i]
  )

  set.seed(1)

  # train model
  xgb_tune <- xgb.cv(
    params = params,
    data = train_final,
    label = train_labels,
    nrounds = 5000,
    nfold = 5,
    objective = "binary:hinge",
    verbose = 0,
    early_stopping_rounds = 10
  )

  # add min training error and optimal number of trees to the hyperpar
ameter grid dataframe, test error corresponds to validation error

  hyperparameter_grid$optimal_trees[i] <- which.min(xgb_tune$evaluatio
n_log$test_error_mean)
  hyperparameter_grid$min_error[i] <- min(xgb_tune$evaluation_log$test
_error_mean)
}

# Sort hyperparameter grid by test error to find the optimal selection
of hyperparameters

hyperparameter_grid %>% arrange(min_error)

# Fit the model corresponding to tuned hyperparameters
```

```
# Generate params list corresponding to optimal hyperparameters

params <- list(eta = 0.1, max_depth = 4, subsample = 0.65)

xgb_fit_final <- xgboost(
                    params = params,
                    data = train_final,
                    label = train_labels,
                    nrounds = 67,
                    objective = 'binary:hinge',
                    verbose = 0)



# Find the test error rate of the chosen model

predictions <- predict(xgb_fit_final, test_final)

test_error <- mean(predictions != test$y.yes)

# Plot a graph showing variable importance

importance_matrix <- xgb.importance(model = xgb_fit_final)

xgb.plot.importance(importance_matrix, top_n = 10, measure = "Gain", m
ain = 'Importance Plot when Duration is Excluded', cex.main = 0.5)
```

## Code for model where 'duration' and macro variables are excluded

```
# Load required packages

library(tidyr)
library(ISLR)
library(gbm)
library(dplyr)
library(caret)
library(xgboost)

# Change working directory

setwd('C:/Users/abugh/Documents/R/Machine Learning Project/BankData')

# Repeat but without duration and macro variables

bankData <- bankData <-  read.csv('bank-additional-full.csv', sep = ';
')

bankData <- bankData %>% drop_na()
```

```
bankData <- bankData %>% select(-c('duration', 'emp.var.rate', 'cons.p
rice.idx', 'cons.conf.idx', 'euribor3m', 'nr.employed'))

# Encode categorical variables as n-1 dummy variables

dummy <- dummyVars(" ~ job + marital + education + default + housing +
loan + contact + month + poutcome + y + day_of_week ", data = bankData
)

dummy_pred = data.frame(predict(dummy, newdata = bankData))

bankData = cbind(bankData %>% select(-c('job', "marital" , "education"
, "default" , "housing" , "loan" , "contact" , "month" , "poutcome", '
day_of_week')), dummy_pred)

# Encode our target variable y as a binary variable that takes on a va
lue of 0 when the target is 'no', and 1 when the target is 'yes

bankData$y.yes = (bankData$y.no * -1) + 1

bankData <- bankData %>% select(-c(y, y.no, pdays))

# Set seed for reproducibility

set.seed(1)

# Split the data into test and train sets, with 1/2 of all observation
s in the training set and half the observations in the test test

train_index <- sample(1:nrow(bankData), nrow(bankData)/2)

train <- bankData[train_index,]
test <- bankData[-train_index,]

# Generate the train and test labels for the response variable

train_labels = train$y.yes
test_labels = test$y.yes


# Convert matrix into a sparse matrix that can be used with xgboost

train_final <- Matrix:: sparse.model.matrix(y.yes~.-1,data = train)
test_final <- Matrix:: sparse.model.matrix(y.yes~.-1,data = test)

# Create grid of hyperparameters that need to be tuned
```

```
hyperparameter_grid <- expand.grid(
  eta = c(0.001, 0.01, 0.1,0.2),
  max_depth = c(1,2,3,4),
  subsample = c(.65, .8, 1),
  optimal_trees = 0,
  min_error = 0
)

# Create a for loop to iterate over hyperparameter grid and fit differ
ent models, then populate optimal trees and min error columns of
# hyperparameter grid. Early stopping is set to 10, which means if the
re is no improvement to error after 10 iterations the optimization sto
ps

for(i in 1:nrow(hyperparameter_grid)) {

  # create parameter list
  params <- list(
    eta = hyperparameter_grid$eta[i],
    max_depth = hyperparameter_grid$max_depth[i],
    subsample = hyperparameter_grid$subsample[i]
  )

  set.seed(1)

  # train model
  xgb_tune <- xgb.cv(
    params = params,
    data = train_final,
    label = train_labels,
    nrounds = 5000,
    nfold = 5,
    objective = "binary:hinge",
    verbose = 0,
    early_stopping_rounds = 10
  )

  # add min training error and optimal number of trees to the hyperpar
ameter grid dataframe, test error corresponds to validation error

  hyperparameter_grid$optimal_trees[i] <- which.min(xgb_tune$evaluatio
n_log$test_error_mean)
  hyperparameter_grid$min_error[i] <- min(xgb_tune$evaluation_log$test
_error_mean)
}
```

```
# Sort hyperparameter grid by validation error to find the optimal sel
ection of hyperparameters

hyperparameter_grid %>% arrange(min_error)

# Fit the model corresponding to tuned hyperparameters

# Generate params list corresponding to optimal hyperparameters

params <- list(eta = 0.2, max_depth = 4, subsample = 0.8)

xgb_fit_final <- xgboost(
  params = params,
  data = train_final,
  label = train_labels,
  nrounds = 30,
  objective = 'binary:hinge',
  verbose = 0)



# Find the test error rate of the chosen model

predictions <- predict(xgb_fit_final, test_final)

test_error <- mean(predictions != test$y.yes)

# Plot a graph showing variable importance

importance_matrix <- xgb.importance(model = xgb_fit_final)

xgb.plot.importance(importance_matrix, top_n = 10, measure = "Gain", m
ain = 'Importance Plot with no Duration and Macro Vars', cex.main = 0.
5)
```