ST310 Group Project: Modelling the success of Kickstarter projects

51126_50537_38134

Contents:

1. Baseline Model
2. Interpretable Models
   2.1 Interpretable Logistic Regression Model
   2.2 Interpretable Classification Tree Model
3. Gradient Descent Model
4. Prediction Focused Models
   4.1 Boosted Trees
   4.2 Random Forest
5. High Dimensional Model

Introduction

Crowdfunding platforms like Kickstarter have transformed the entrepreneurial landscape by enabling individuals to raise capital directly from the public to fund a variety of creative endeavours. Our goal with this project was to investigate using Machine Learning techniques what factors make it more likely for a project to succeed. This could be useful for creators, who gain insight into how to increase their chances of a successful campaign, as well as for backers, who could find out the probability a specific campaign will be successful.

We conducted our analysis on a sample of over 200,000 campaigns launched between 2009 and 2024, which we obtained from a web scraping website [1]. As there have only been slightly more than 650,000 Kickstarter campaigns ever [2], we find our dataset to be a representative sample. It includes information on a variety of project characteristics:

- The project's funding goal (in US dollars)
- A project's category (ranging from photography to food)
- The currency a project was launched in
- Whether the project is a 'staff-pick' (a special designation given to projects the Kickstarter staff finds particularly noteworthy or deserving of extra visibility)
- Whether a video is included in the Kickstarter page
- The campaign's desired duration prior to launch
- The date a campaign was launched (broken down into the year, month and day of the week)
- The number of characters in a project's blurb
- The final outcome of a project (whether it met its goal)- this is what we predict

Our interpretable model found there are ways to reliably improve the probability a project meets its goal; including a video, setting a reasonable goal, including a higher number of blurb characters, as well as launching shorter campaigns are all recommended. Our prediction focused model was built on only 20% of the total dataset due to overly long runtimes, however, it achieved an accuracy of 78%, compared to a baseline model accuracy of 70%. We also built a high dimensional model on an even smaller subset of the dataset, as well as a gradient descent model, that reinforced the findings of our interpretable model.

[1] Vitulskis, T., & Jonaitis, P. (2025, April 11). Kickstarter Datasets. Retrieved from Web Robots: https://webrobots.io/kickstarter-datasets/ [2] Woodward, M. (2025, April 11). Search Logistics. Retrieved from KICKSTARTER STATS & FACTS 2025: EVERYTHING YOU NEED TO KNOW : https://www.searchlogistics.com/learn/statistics/kickstarter-stats-facts/

Loading libraries and some preprocessing

```
library(tidymodels)
```

```
## Warning: package 'tidymodels' was built under R version 4.3.3

## -- Attaching packages -------------------------------------- tidymodels 1.3.0 --

## v broom        1.0.8     v recipes      1.2.1
## v dials        1.4.0     v rsample      1.3.0
## v dplyr        1.1.4     v tibble       3.2.1
## v ggplot2      3.5.2     v tidyr        1.3.1
## v infer        1.0.8     v tune         1.3.0
## v modeldata    1.4.0     v workflows    1.2.0
## v parsnip      1.3.1     v workflowsets 1.1.0
## v purrr        1.0.4     v yardstick    1.3.2

## Warning: package 'broom' was built under R version 4.3.3

## Warning: package 'dials' was built under R version 4.3.3

## Warning: package 'modeldata' was built under R version 4.3.3

## Warning: package 'parsnip' was built under R version 4.3.3

## Warning: package 'purrr' was built under R version 4.3.3

## Warning: package 'recipes' was built under R version 4.3.3

## Warning: package 'rsample' was built under R version 4.3.3

## Warning: package 'tune' was built under R version 4.3.3

## Warning: package 'workflows' was built under R version 4.3.3

## Warning: package 'workflowsets' was built under R version 4.3.3

## Warning: package 'yardstick' was built under R version 4.3.3

## -- Conflicts ----------------------------------------- tidymodels_conflicts() --
## x purrr::discard() masks scales::discard()
## x dplyr::filter()  masks stats::filter()
## x dplyr::lag()     masks stats::lag()
## x recipes::step()  masks stats::step()
```

```
library(dplyr)
```

```
kick_data <- read.csv('cleaned310.csv')

#Centred year since later found it made more sense as a centred predictor
kick_data$year_centred <- kick_data$year_launched - 2009
kick_data <- kick_data %>% select(-year_launched)
```

```r
# creating a log_goal_usd since it worked better for later models
kick_data$log_goal_usd <- log2(kick_data$goal_usd)

#Made the following columns the correct type for later models since they are categorical
kick_data <- kick_data %>%
  mutate(
    category = as.factor(category),
    currency = as.factor(currency),
    day_launched = as.factor(day_launched),
    month_launched = as.factor(month_launched),
  )

kick_data <- na.omit(kick_data) #removed all remaining missing rows
```

1. Baseline Model

For the baseline model predicting the state of Kickstarter projects, we created a logistic regression model on all predictors.

```r
baseline_model <- glm(state~. -(log_goal_usd), data=kick_data)
#the baseline model does not use the logarithmic transform of goal
summary(baseline_model)
```

```
##
## Call:
## glm(formula = state ~ . - (log_goal_usd), data = kick_data)
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
## (Intercept)              5.982e-01  1.008e-02  59.323  < 2e-16 ***
## categoryComics           6.604e-02  6.206e-03  10.642  < 2e-16 ***
## categoryCrafts          -3.103e-01  6.082e-03 -51.024  < 2e-16 ***
## categoryDance           -1.061e-01  1.022e-02 -10.379  < 2e-16 ***
## categoryDesign          -7.675e-02  6.721e-03 -11.419  < 2e-16 ***
## categoryFashion         -7.053e-02  5.168e-03 -13.646  < 2e-16 ***
## categoryFilm & Video    -4.403e-02  4.158e-03 -10.587  < 2e-16 ***
## categoryFood            -2.820e-01  4.938e-03 -57.102  < 2e-16 ***
## categoryGames           -9.475e-02  5.577e-03 -16.989  < 2e-16 ***
## categoryJournalism      -2.988e-01  8.314e-03 -35.939  < 2e-16 ***
## categoryMusic            4.854e-02  4.190e-03  11.585  < 2e-16 ***
## categoryPhotography     -1.955e-01  6.150e-03 -31.795  < 2e-16 ***
## categoryPublishing      -2.856e-02  4.486e-03  -6.366 1.95e-10 ***
## categoryTechnology      -1.703e-01  4.475e-03 -38.050  < 2e-16 ***
## categoryTheater         -6.916e-02  6.546e-03 -10.565  < 2e-16 ***
## currencyCAD              2.954e-02  8.061e-03   3.664 0.000248 ***
## currencyCHF             -8.460e-02  1.822e-02  -4.642 3.45e-06 ***
## currencyDKK              3.105e-02  1.744e-02   1.780 0.075011 .
## currencyEUR             -5.795e-02  7.404e-03  -7.826 5.06e-15 ***
## currencyGBP              6.627e-02  7.238e-03   9.155  < 2e-16 ***
## currencyHKD              2.606e-01  1.143e-02  22.800  < 2e-16 ***
## currencyJPY              9.926e-02  1.554e-02   6.387 1.70e-10 ***
## currencyMXN             -1.258e-01  1.014e-02 -12.410  < 2e-16 ***
## currencyNOK             -4.501e-02  2.404e-02  -1.872 0.061196 .
## currencyNZD              1.858e-03  1.736e-02   0.107 0.914744
## currencyPLN             -1.707e-01  4.054e-02  -4.210 2.55e-05 ***
## currencySEK             -1.766e-03  1.410e-02  -0.125 0.900316
## currencySGD              9.119e-02  1.612e-02   5.658 1.54e-08 ***
## currencyUSD              3.670e-02  6.715e-03   5.465 4.63e-08 ***
## staff_pickTRUE           3.206e-01  2.980e-03 107.575  < 2e-16 ***
## videoTRUE                1.519e-01  2.291e-03  66.323  < 2e-16 ***
## goal_usd                -1.027e-08  8.835e-10 -11.621  < 2e-16 ***
## desired_duration        -2.643e-04  3.390e-06 -77.971  < 2e-16 ***
## day_launchedMonday      -4.908e-03  3.624e-03  -1.354 0.175710
## day_launchedSaturday    -2.112e-02  4.422e-03  -4.775 1.79e-06 ***
## day_launchedSunday      -2.003e-02  4.727e-03  -4.238 2.26e-05 ***
## day_launchedThursday     5.529e-03  3.707e-03   1.491 0.135893
## day_launchedTuesday      2.505e-02  3.444e-03   7.275 3.49e-13 ***
## day_launchedWednesday    5.586e-03  3.616e-03   1.545 0.122447
## month_launchedAugust    -1.434e-02  4.900e-03  -2.926 0.003437 **
## month_launchedDecember  -7.948e-03  5.409e-03  -1.469 0.141713
## month_launchedFebruary   3.351e-03  4.941e-03   0.678 0.497679
## month_launchedJanuary    8.761e-03  4.842e-03   1.809 0.070400 .
## month_launchedJuly      -2.887e-02  4.802e-03  -6.013 1.83e-09 ***
## month_launchedJune      -6.283e-03  4.928e-03  -1.275 0.202294
## month_launchedMarch      1.432e-02  4.819e-03   2.972 0.002956 **
## month_launchedMay        3.329e-03  4.856e-03   0.686 0.493002
## month_launchedNovember  -8.175e-03  5.105e-03  -1.601 0.109315
## month_launchedOctober   -7.291e-03  4.996e-03  -1.459 0.144468
## month_launchedSeptember -5.275e-03  4.990e-03  -1.057 0.290499
## blurb_characters        -1.705e-04  3.390e-05  -5.029 4.93e-07 ***
## year_centred             1.146e-02  3.167e-04  36.194  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.1949967)
##
##     Null deviance: 46287  on 188929  degrees of freedom
## Residual deviance: 36831  on 188878  degrees of freedom
## AIC: 227356
##
## Number of Fisher Scoring iterations: 2
```

```r
#Confusion Matrix for baseline model- in order to find predictive accuracy
set.seed(53)
train <- sample(1:nrow(kick_data), nrow(kick_data)/2)
test <- (-train)

training_data <- kick_data[train, ]
test_data <- kick_data[test, ]

baseline_dummy <- glm(state~. -(log_goal_usd), data=training_data)
predictions <- predict(baseline_dummy, newdata=test_data)

confusion_matrix <- table(test_data$state, predictions>0.5)
#so if the model has probability greater than a half that a project is successful it
#comes up as successful
print(confusion_matrix)
```

```
##
##         FALSE  TRUE
##   FALSE 23579 16996
##   TRUE  11491 42399
```

```
print(sum(diag(confusion_matrix))/sum(confusion_matrix))
#to find accuracy I summed diagonals over total
```

```
## [1] 0.6984386
```

So the baseline model has around a 70% accuracy.

2. Interpretable models

2.1 Interpretable Logistic Regression Model

Our method for fitting this model was to first try multiple different subsets of predictors and find the one which led to the lowest BIC- since BIC penalises excessive use of parameters, encouraging interpretable models. We initially wanted to use the leaps package mentioned in ISLR for best subset selection, but we didn't find this to be easy to do for logistic regression, so fitted the models manually.

```
BIC(baseline_model)
```

```
## [1] 227894.4
```

```
interpret_model_goal <- glm(state~. -(goal_usd), data=kick_data)
BIC(interpret_model_goal)
```

```
## [1] 211199.6
```

```
interpret_model1 <- glm(state~. -(day_launched) -(log_goal_usd), data=kick_data)
BIC(interpret_model1)
```

```
## [1] 228013.5
```

```
interpret_model2 <- glm(state~. -(month_launched) -(log_goal_usd), data=kick_data)
BIC(interpret_model2)
```

```
## [1] 227883.4
```

```
interpret_model3 <- glm(state ~. -(blurb_characters) -(log_goal_usd), data=kick_data)
BIC(interpret_model3)
```

```
## [1] 227907.5
```

```
interpret_model4 <- glm(state ~. -(desired_duration) -(log_goal_usd), data=kick_data)
BIC(interpret_model4)
```

```
## [1] 233867.5
```

```
interpret_model5 <- glm(state~. -(currency) -(log_goal_usd), data=kick_data)
BIC(interpret_model5)
```

```
## [1] 229746.5
```

```
interpret_model6 <- glm(state~. -(category) -(log_goal_usd), data=kick_data)
BIC(interpret_model6)
```

```
## [1] 238292.2
```

```
interpret_model7 <- glm(state~. -(year_centred) -(log_goal_usd), data=kick_data)
BIC(interpret_model7)
```

```
## [1] 229188
```

```
#We didn't bother checking removing predictors like staff_pick as they appeared
#extremely significant in the baseline model already
#By utilising the BIC we found including a logarithmic transform for goal, and
#removing month of launch as a predictor led to a decrease in BIC- leading us to
#do that for our model
```

```
interpret_model <- glm(state~. -(month_launched) -(goal_usd), data=kick_data)
summary(interpret_model)
```

```
##
## Call:
## glm(formula = state ~ . - (month_launched) - (goal_usd), data = kick_data)
##
## Coefficients:
##                         Estimate Std. Error  t value Pr(>|t|)
## (Intercept)            1.126e+00  9.939e-03  113.259  < 2e-16 ***
## categoryComics         6.906e-02  5.938e-03   11.630  < 2e-16 ***
## categoryCrafts        -2.964e-01  5.821e-03  -50.930  < 2e-16 ***
## categoryDance         -6.933e-02  9.789e-03   -7.083 1.42e-12 ***
## categoryDesign        -4.438e-02  6.436e-03   -6.895 5.42e-12 ***
## categoryFashion       -1.636e-02  4.963e-03   -3.297 0.000977 ***
## categoryFilm & Video   3.016e-02  4.019e-03    7.505 6.17e-14 ***
## categoryFood          -1.545e-01  4.824e-03  -32.032  < 2e-16 ***
## categoryGames         -1.979e-02  5.368e-03   -3.687 0.000227 ***
## categoryJournalism    -2.405e-01  7.969e-03  -30.183  < 2e-16 ***
## categoryMusic          7.934e-02  4.017e-03   19.752  < 2e-16 ***
## categoryPhotography   -1.503e-01  5.896e-03  -25.486  < 2e-16 ***
## categoryPublishing    -1.220e-02  4.294e-03   -2.842 0.004486 **
## categoryTechnology    -4.748e-02  4.382e-03  -10.833  < 2e-16 ***
## categoryTheater       -2.263e-02  6.274e-03   -3.607 0.000310 ***
## currencyCAD            2.598e-02  7.715e-03    3.367 0.000759 ***
## currencyCHF           -2.585e-02  1.745e-02   -1.482 0.138434
## currencyDKK            2.710e-02  1.669e-02    1.624 0.104400
## currencyEUR           -3.706e-02  7.088e-03   -5.230 1.70e-07 ***
## currencyGBP            6.015e-02  6.927e-03    8.684  < 2e-16 ***
## currencyHKD            2.144e-01  1.095e-02   19.587  < 2e-16 ***
## currencyJPY            6.643e-02  1.488e-02    4.466 7.98e-06 ***
## currencyMXN           -1.702e-01  9.705e-03  -17.534  < 2e-16 ***
## currencyNOK           -2.973e-02  2.301e-02   -1.292 0.196207
## currencyNZD           -1.914e-03  1.661e-02   -0.115 0.908273
## currencyPLN           -1.746e-01  3.880e-02   -4.499 6.83e-06 ***
## currencySEK            7.849e-03  1.349e-02    0.582 0.560836
## currencySGD            7.921e-02  1.542e-02    5.135 2.82e-07 ***
## currencyUSD            5.757e-02  6.428e-03    8.957  < 2e-16 ***
## staff_pickTRUE         3.748e-01  2.881e-03  130.121  < 2e-16 ***
## videoTRUE              2.019e-01  2.223e-03   90.828  < 2e-16 ***
## desired_duration      -1.660e-04  3.327e-06  -49.876  < 2e-16 ***
## day_launchedMonday    -2.148e-03  3.469e-03   -0.619 0.535756
## day_launchedSaturday  -2.591e-02  4.233e-03   -6.121 9.29e-10 ***
## day_launchedSunday    -2.517e-02  4.524e-03   -5.565 2.63e-08 ***
## day_launchedThursday   7.833e-03  3.548e-03    2.208 0.027276 *
## day_launchedTuesday    2.898e-02  3.296e-03    8.791  < 2e-16 ***
## day_launchedWednesday  8.755e-03  3.461e-03    2.530 0.011418 *
## blurb_characters       3.021e-05  3.247e-05    0.930 0.352192
## year_centred           1.110e-02  3.020e-04   36.739  < 2e-16 ***
## log_goal_usd          -6.087e-02  4.590e-04 -132.595  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.1786256)
##
##     Null deviance: 46287  on 188929  degrees of freedom
## Residual deviance: 33740  on 188889  degrees of freedom
## AIC: 210778
##
## Number of Fisher Scoring iterations: 2
```

```r
#Predictive power of more interpretable model:
interpret_dummy <- glm(state~. -(month_launched) -(goal_usd), data=training_data)
predictions2 <- predict(interpret_dummy, newdata=test_data)

confusion_matrix2 <- table(test_data$state, predictions2>0.5)
print(confusion_matrix2)
```

```
##
##         FALSE  TRUE
##   FALSE 25911 14664
##   TRUE   9908 43982
```

```r
print(sum(diag(confusion_matrix2))/sum(confusion_matrix2))
```

```
## [1] 0.7398825
```

This model had higher predictive power than the baseline simply by including a logarithmic transform and removing one predictor. It can also be relatively easily interpreted from the summary output above.

For one, we can gain insight into the direction different predictors affect project success. For example, the positive coefficient on year suggests projects are getting more successful year over year, whilst the high positive coefficients of staff pick and video suggests a project that is a staff pick (a special designation given to projects the Kickstarter team wants to boost on their page) has a higher probability of success, and including a video also boosts the probability of success. We can also see projects launched with pounds (GBP) are more likely to succeed than Euros, with the currency that predicts the highest chance of success being the Japanese yen.

By plugging the values for a few examples into the coefficients provided by the summary output, we can also gain more specific insight into the way certain predictors affect success. For example, a project launched in the comics category that also has a few other characteristics (has a video, launched in 2025 on a Friday, has a goal of $1000), is predicted a 72.6% chance of success by the model (we used the inverse logistic function on the specific sum of values). Additionally, a project with the exact same characteristics, except launched in the crafts category, is only predicted a 61% chance of success. This demonstrates the importance the category of project has on likelihood of success.

Our results also align with past research. Oduro, Yu, & Huang, (2022)* employed logistic regression and classification tree methodologies to predict the probability a project meets its goal. They similarly emphasised the predictive value of factors such as funding goal and the inclusion of promotional videos on the project page, which they hypothesised increased engagement and trust among potential backers.

Compared to the baseline model, this model is very similar (as the only difference is one less predictor and a logarithmic transform), and so the results are largely similar to what could be taken from the baseline.

*Source: Oduro, M. S., Yu, H., & Huang, H. (2022). Predicting the Entrepreneurial Success of Crowdfunding Campaigns Using Model-Based Machine Learning Methods. International Journal of Crowd Science, 7-16.

2.2 Interpretable Classification Tree Model

We will utilise classification trees to build a non-baseline model that is interpretable. In particular, we make use of the tree structure and the variable importance feature of classification trees to conduct some inference.

```r
set.seed(123)
library(rpart)
```

```
## Warning: package 'rpart' was built under R version 4.3.3

##
## Attaching package: 'rpart'

## The following object is masked from 'package:dials':
##
##     prune
```

```r
library(rpart.plot)
```

```
## Warning: package 'rpart.plot' was built under R version 4.3.3
```

```r
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.3.3

## Loading required package: lattice

## Warning: package 'lattice' was built under R version 4.3.3

##
## Attaching package: 'caret'

## The following objects are masked from 'package:yardstick':
##
##     precision, recall, sensitivity, specificity

## The following object is masked from 'package:purrr':
##
##     lift
```

```r
library(tidymodels)

kick_data_split <- initial_split(kick_data, prop= 0.8, strata = state)
kick_data_train <- training(kick_data_split)
kick_data_test <- testing(kick_data_split)


# Initially, we will use a low initial cp value to allow the classification tree
# to grow a deep tree
classification_tree <- rpart(state ~ ., data = kick_data_train, method = "class",
                             control = rpart.control(minsplit = 20, cp = 0.0001))

# print cp table to inspect
cp_table_full <- printcp(classification_tree)
```

```
##
```

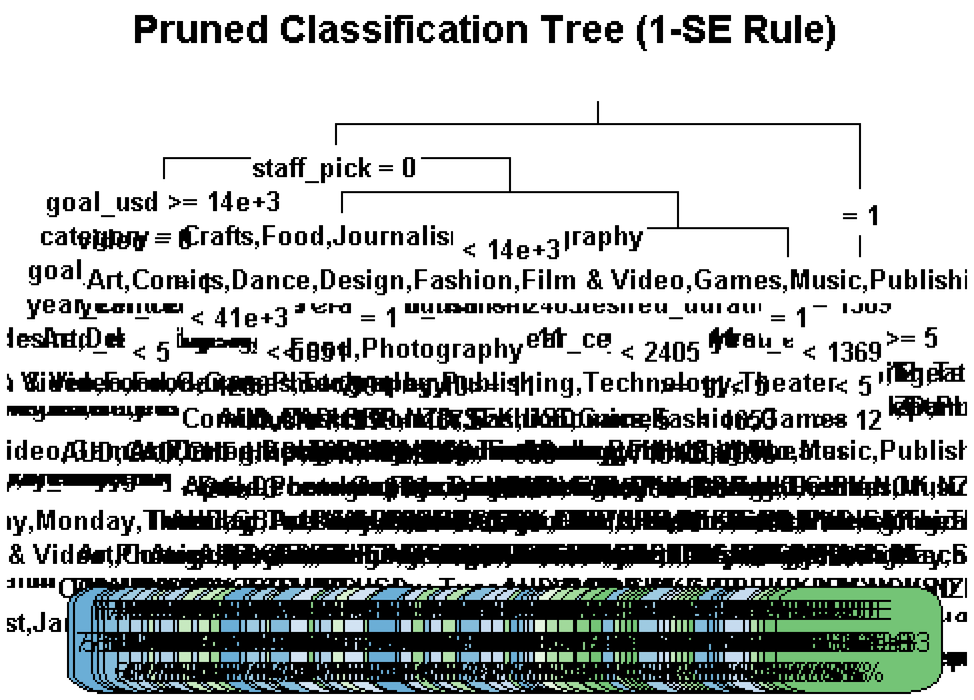
```
## Classification tree:
## rpart(formula = state ~ ., data = kick_data_train, method = "class",
##     control = rpart.control(minsplit = 20, cp = 1e-04))
##
## Variables actually used in tree construction:
##  [1] blurb_characters category         currency         day_launched
##  [5] desired_duration goal_usd         month_launched   staff_pick
##  [9] video            year_centred
##
## Root node error: 64881/151143 = 0.42927
##
## n= 151143
##
##            CP nsplit rel error  xerror      xstd
## 1  0.10893790      0   1.00000 1.00000 0.0029659
## 2  0.08893204      2   0.78212 0.78219 0.0028298
## 3  0.02613246      3   0.69319 0.69321 0.0027395
## 4  0.00510935      5   0.64093 0.64760 0.0026845
## 5  0.00432587      7   0.63071 0.62880 0.0026600
## 6  0.00278202     10   0.61773 0.62111 0.0026496
## 7  0.00194715     13   0.60887 0.61138 0.0026363
## 8  0.00126385     18   0.59825 0.60132 0.0026222
## 9  0.00120734     23   0.58945 0.59498 0.0026131
## 10 0.00110972     26   0.58583 0.59076 0.0026070
## 11 0.00107119     28   0.58361 0.58862 0.0026038
## 12 0.00106349     30   0.58146 0.58769 0.0026025
## 13 0.00104807     32   0.57934 0.58695 0.0026014
## 14 0.00092785     33   0.57829 0.58498 0.0025985
## 15 0.00086312     40   0.57106 0.58046 0.0025918
## 16 0.00084771     41   0.57020 0.57909 0.0025897
## 17 0.00081688     42   0.56935 0.57733 0.0025871
## 18 0.00075523     43   0.56853 0.57710 0.0025867
## 19 0.00066275     46   0.56627 0.57494 0.0025835
## 20 0.00064734     47   0.56560 0.57424 0.0025824
## 21 0.00063193     48   0.56496 0.57390 0.0025819
## 22 0.00060110     52   0.56243 0.57353 0.0025814
## 23 0.00058569     53   0.56183 0.57333 0.0025810
## 24 0.00050862     54   0.56124 0.57137 0.0025781
## 25 0.00046238     55   0.56073 0.56920 0.0025748
## 26 0.00045468     56   0.56027 0.56838 0.0025735
## 27 0.00040073     58   0.55936 0.56792 0.0025728
## 28 0.00038532     63   0.55736 0.56773 0.0025725
## 29 0.00032881     65   0.55659 0.56753 0.0025722
## 30 0.00032367     68   0.55560 0.56688 0.0025712
## 31 0.00031596     70   0.55495 0.56678 0.0025710
## 32 0.00030826     72   0.55432 0.56696 0.0025713
## 33 0.00028771     73   0.55401 0.56712 0.0025716
## 34 0.00027743     76   0.55315 0.56614 0.0025701
## 35 0.00027229     78   0.55260 0.56625 0.0025702
## 36 0.00026202     82   0.55139 0.56611 0.0025700
## 37 0.00025431     85   0.55061 0.56605 0.0025699
## 38 0.00024661     87   0.55010 0.56553 0.0025691
## 39 0.00022349     90   0.54936 0.56547 0.0025690
## 40 0.00021578     96   0.54802 0.56454 0.0025676
## 41 0.00021064    102   0.54660 0.56383 0.0025665
## 42 0.00020807    105   0.54597 0.56389 0.0025666
## 43 0.00020037    109   0.54514 0.56355 0.0025661
## 44 0.00019266    111   0.54474 0.56326 0.0025656
## 45 0.00018495    114   0.54415 0.56325 0.0025656
## 46 0.00017982    118   0.54341 0.56305 0.0025653
## 47 0.00016954    121   0.54287 0.56251 0.0025644
## 48 0.00016183    126   0.54193 0.56174 0.0025632
## 49 0.00015927    132   0.54096 0.56214 0.0025639
## 50 0.00015413    138   0.53956 0.56203 0.0025637
## 51 0.00014385    168   0.53438 0.56183 0.0025634
## 52 0.00013872    180   0.53230 0.56137 0.0025627
## 53 0.00013101    191   0.53077 0.56137 0.0025627
## 54 0.00012844    197   0.52989 0.56132 0.0025626
## 55 0.00012639    200   0.52951 0.56124 0.0025625
## 56 0.00012330    218   0.52696 0.56120 0.0025624
## 57 0.00011817    237   0.52451 0.56106 0.0025622
## 58 0.00011560    240   0.52416 0.56077 0.0025617
## 59 0.00010789    246   0.52347 0.56063 0.0025615
## 60 0.00010404    251   0.52293 0.56066 0.0025616
## 61 0.00010275    256   0.52240 0.56064 0.0025615
## 62 0.00010018    259   0.52209 0.56095 0.0025620
## 63 0.00010000    269   0.52109 0.56095 0.0025620
```

```r
# one method we will try out is to select a CP using the 1-SE rule
min_xerror_index <- which.min(cp_table_full[, "xerror"])
min_xerror <- min(cp_table_full[, "xerror"])
min_xerror_se <- cp_table_full[min_xerror_index, "xstd"]
threshold <- min_xerror + min_xerror_se
eligible_rows <- which(cp_table_full[, "xerror"] <= threshold)
best_cp_index <- min(eligible_rows)  # Get the largest CP value which is the simplest model
one_se_cp <- cp_table_full[best_cp_index, "CP"]
pruned_model_one_se <- prune(classification_tree, cp = one_se_cp)
one_se_cp
```

```
## [1] 0.0001798164
```

```r
#  0.0002846898, hence we will prune it with this cp value
one_se_pruned_model <- prune(classification_tree, cp = one_se_cp)
# visualise
rpart.plot(one_se_pruned_model, type = 3, extra = 101, fallen.leaves = TRUE,
           main = "Pruned Classification Tree (1-SE Rule)", cex = 0.8)
```

```
## Warning: labs do not fit even at cex 0.15, there may be some overplotting
```

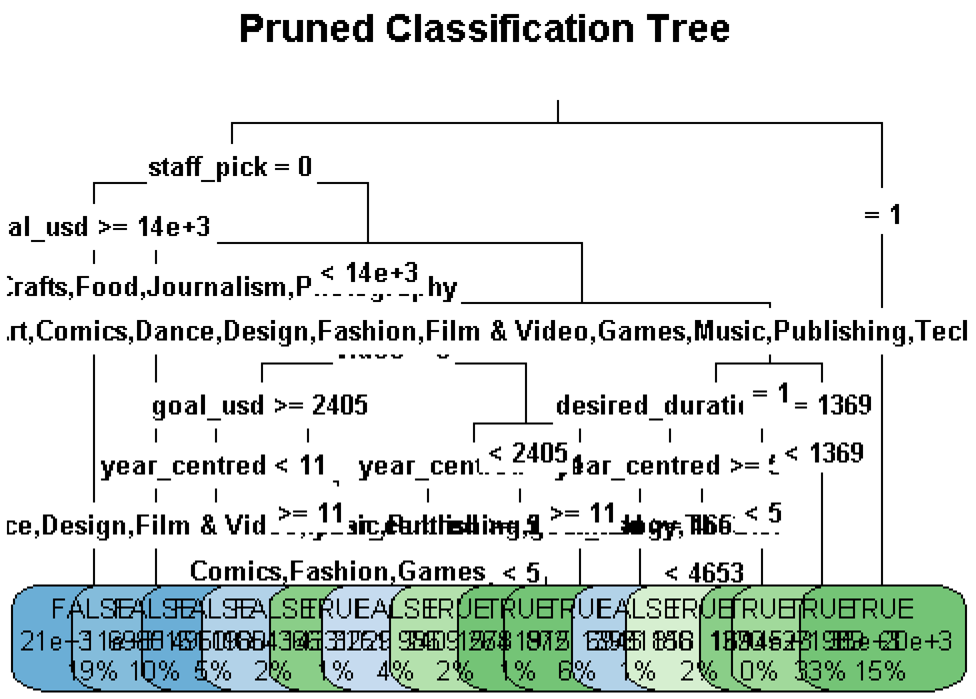


```
# As we can see from here this is no where near an interpretable tree, suggesting
# our CP is way too small.
# This can be due to the fact the we have a very large dataset with near 200,000
# entries, the SE we get is extremely small that the 1 SE rule just results in
# selecting the CP value with the minimum error.
# Therefore as this method does not work for us, we will inspect the CP table and
# manually select a CP value that will give us an interpretable tree
```

```
classification_tree <- rpart(state ~ ., data = kick_data_train, method = "class",
                             control = rpart.control(minsplit = 20, cp = 0.0001))
# print CP table to inspect
cp_table_full <- printcp(classification_tree)
```

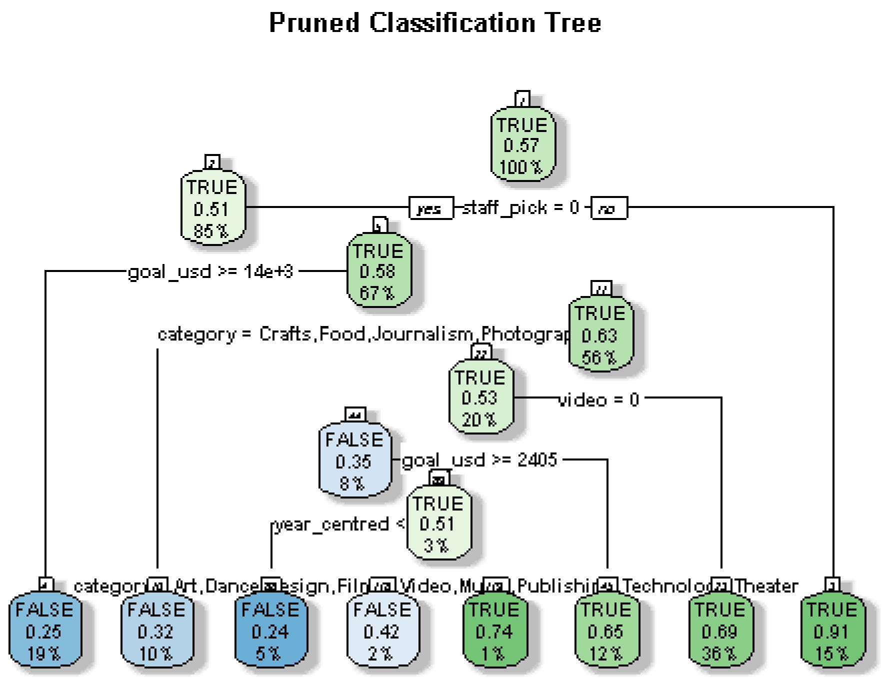
```
##
## Classification tree:
## rpart(formula = state ~ ., data = kick_data_train, method = "class",
##     control = rpart.control(minsplit = 20, cp = 1e-04))
##
## Variables actually used in tree construction:
##  [1] blurb_characters category         currency         day_launched
##  [5] desired_duration goal_usd         month_launched   staff_pick
##  [9] video            year_centred
##
## Root node error: 64881/151143 = 0.42927
##
## n= 151143
##
##             CP nsplit rel error  xerror      xstd
## 1   0.10893790      0   1.00000 1.00000 0.0029659
## 2   0.08893204      2   0.78212 0.78373 0.0028312
## 3  0.02613246      3   0.69319 0.69395 0.0027404
## 4  0.00510935      5   0.64093 0.64786 0.0026848
## 5  0.00432587      7   0.63071 0.63116 0.0026631
## 6  0.00278202     10   0.61773 0.62155 0.0026503
## 7  0.00194715     13   0.60887 0.61083 0.0026355
## 8  0.00126385     18   0.59825 0.60182 0.0026229
## 9  0.00120734     23   0.58945 0.59614 0.0026147
## 10 0.00110972     26   0.58583 0.59316 0.0026104
## 11 0.00107119     28   0.58361 0.58994 0.0026058
## 12 0.00106349     30   0.58146 0.58937 0.0026049
## 13 0.00104807     32   0.57934 0.58879 0.0026041
## 14 0.00092785     33   0.57829 0.58632 0.0026005
## 15 0.00086312     40   0.57106 0.58153 0.0025934
## 16 0.00084771     41   0.57020 0.58083 0.0025923
## 17 0.00081688     42   0.56935 0.58039 0.0025917
## 18 0.00075523     43   0.56853 0.57769 0.0025876
## 19 0.00066275     46   0.56627 0.57686 0.0025864
## 20 0.00064734     47   0.56560 0.57585 0.0025849
## 21 0.00063193     48   0.56496 0.57585 0.0025849
## 22 0.00060110     52   0.56243 0.57536 0.0025841
## 23 0.00058569     53   0.56183 0.57464 0.0025830
## 24 0.00050862     54   0.56124 0.57234 0.0025795
## 25 0.00046238     55   0.56073 0.57074 0.0025771
## 26 0.00045468     56   0.56027 0.57043 0.0025766
## 27 0.00040073     58   0.55936 0.56943 0.0025751
## 28 0.00038532     63   0.55736 0.56835 0.0025735
## 29 0.00032881     65   0.55659 0.56744 0.0025721
## 30 0.00032367     68   0.55560 0.56738 0.0025720
## 31 0.00031596     70   0.55495 0.56678 0.0025710
## 32 0.00030826     72   0.55432 0.56661 0.0025708
## 33 0.00028771     73   0.55401 0.56627 0.0025703
## 34 0.00027743     76   0.55315 0.56564 0.0025693
## 35 0.00027229     78   0.55260 0.56565 0.0025693
## 36 0.00026202     82   0.55139 0.56453 0.0025676
## 37 0.00025431     85   0.55061 0.56433 0.0025673
## 38 0.00024661     87   0.55010 0.56329 0.0025657
## 39 0.00022349     90   0.54936 0.56274 0.0025648
## 40 0.00021578     96   0.54802 0.56229 0.0025641
## 41 0.00021064    102   0.54660 0.56201 0.0025637
## 42 0.00020807    105   0.54597 0.56177 0.0025633
## 43 0.00020037    109   0.54514 0.56147 0.0025628
## 44 0.00019266    111   0.54474 0.56137 0.0025627
## 45 0.00018495    114   0.54415 0.56114 0.0025623
## 46 0.00017982    118   0.54341 0.56107 0.0025622
## 47 0.00016954    121   0.54287 0.56084 0.0025618
## 48 0.00016183    126   0.54193 0.56089 0.0025619
## 49 0.00015927    132   0.54096 0.56052 0.0025613
## 50 0.00015413    138   0.53956 0.56038 0.0025611
## 51 0.00014385    168   0.53438 0.56073 0.0025617
## 52 0.00013872    180   0.53230 0.56078 0.0025618
## 53 0.00013101    191   0.53077 0.56066 0.0025616
## 54 0.00012844    197   0.52989 0.56106 0.0025622
## 55 0.00012639    200   0.52951 0.56098 0.0025621
## 56 0.00012330    218   0.52696 0.56100 0.0025621
## 57 0.00011817    237    0.52451 0.56112 0.0025623
## 58 0.00011560    240    0.52416 0.56097 0.0025620
## 59 0.00010789    246    0.52347 0.56120 0.0025624
## 60 0.00010404    251    0.52293 0.56117 0.0025624
## 61 0.00010275    256    0.52240 0.56117 0.0025624
## 62 0.00010018    259    0.52209 0.56189 0.0025635
## 63 0.00010000    269    0.52109 0.56189 0.0025635
```

```
# lets try a CP of  0.00247757 which has a xerror 10% higher then the minimum
pruned_model <- prune(classification_tree, cp = 0.00247757)
# visualise
rpart.plot(pruned_model, type = 3, extra = 101, fallen.leaves = TRUE,
           main = "Pruned Classification Tree", cex = 0.8)
```



```
# still too deep

# lets try a CP of  0.005
pruned_model <- prune(classification_tree, cp = 0.005)
# visualise
rpart.plot(pruned_model, box.palette = "auto", shadow.col = "gray", nn = TRUE,
           main = "Pruned Classification Tree", fallen.leaves = TRUE, tweak = 2)
```



```r
# this looks reasonable
prediction_tree <- predict(pruned_model, kick_data_test, type = "class")
kick_data_test$state <- as.factor(kick_data_test$state)
confusion_matrix_tree <- confusionMatrix(prediction_tree, kick_data_test$state)
print(confusion_matrix_tree)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE  9628  3706
##      TRUE   6593 17860
##
##                Accuracy : 0.7274
##                  95% CI : (0.7229, 0.7319)
##     No Information Rate : 0.5707
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.4312
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.5936
##             Specificity : 0.8282
##          Pos Pred Value : 0.7221
##          Neg Pred Value : 0.7304
##              Prevalence : 0.4293
##          Detection Rate : 0.2548
##    Detection Prevalence : 0.3529
##        Balanced Accuracy : 0.7109
##
##         'Positive' Class : FALSE
##
```

```
# show variable importance
importance_tree <- pruned_model$variable.importance
print(importance_tree)
```

```
##         goal_usd     log_goal_usd       staff_pick         category
##       6266.78658       6266.78658       6129.71898       2984.22089
##            video     year_centred blurb_characters desired_duration
##        934.61744        409.09006         72.43474         27.55862
##         currency
##         22.04174
```

We can see that with this classification tree, we have obtained an accuracy of 73%, which is slightly higher than our baseline model.

From the tree plot, we observe that being a staff pick automatically predicts the project to have a 0.91 chance of success, indicating how strong a predictor it is. This is also verified in the variable importance table, with staff pick being the 2nd most important predictor after goal. If the project is not a staff pick, then the goal amount becomes extremely important in predicting success, with goals larger than 14,000 being predicted to fail with 0.25 probability of success, this shows an intuitive and obvious result of having a larger goal amount, you are less likely to succeed.

As we saw in the logistic model, we observe that category is also an important predictor; depending on which category the project is in, the chances of reaching the goal amount changes significantly. Here, we can see how categories that are quite niche in the crowd funding industry like food, crafts(which we also saw in the previous logistic model), journalism and photography are predicted to fail, suggesting that projects in these categories get less attention from the public and hence gain fewer backers, making it harder for them to succeed.

Lastly, we observe that video is also an important predictor, with projects that include a video being more likely to succeed. Including a video allows backers to gain more information about the project and potentially increases its perceived trustworthiness. Overall, this classification tree provides interpretable results and meaningful insights into the data, offering consistent findings with those from the logistic regression model.

3. Gradient Descent Model

To create a model using gradient descent, we first split the data into training (for training the model), and testing (for the evaluation and comparison of the model) sets. Then, we prepared the data for a gradient descent algorithm using the `recipes` package of `tidymodels`[1] by one hot dummy coding categorical (>2 categories) variables and dummy coding binary variables, and then standardising all variables to have a mean of 0 and a standard deviation of 1. This 'recipe' was then applied to the training and testing data sets.

Our gradient descent function computes the loss for a given set of input coefficients, finds the gradient of that loss function (we use cross entropy loss, along with a penalty term for regularisation), and then updates the coefficient weights based on the learning rate and computed gradient again and again for a number of epochs. Our evaluation function (inspired by our 311 course) visualises how the accuracy of our model changes over the epochs. Using these functions we then carried out a grid search for a variety of possible best values for our hyper parameters: lambda, type of penalty (if any), learning rate, and how many epochs to train the model for to achieve the best test accuracy. We found that regularisation only improved our accuracy by 0.1% but did allow the model to learn the data and reach this test accuracy much faster. Also, regularisation should have helped avoid overfitting to the training data, allowing for better generalisation.

We then trained the model using these tuned hyperparameters, and our results and interpretations can be found at the end of this section.

references: [1] `https://recipes.tidymodels.org/reference/index.html`

Train / Test Split for data

```r
set.seed(53)
data_split <- initial_split(kick_data, prop = 0.8)
train_data <- training(data_split)
test_data  <- testing(data_split)
```

Preparing data for Gradient Descent Algorithm

```r
## getting logical predictors so that
logi_vars <- train_data %>% select(where(is.logical)) %>% names()

## creating recipe
rec <- recipe(state ~ ., data = train_data) %>%
  step_rm(goal_usd) %>% # improved test performance (found heuristically)
  step_rm(month_launched) %>% # improved test performance (found heuristically)
  step_dummy(all_nominal_predictors(), one_hot = TRUE) %>%
  # making dummy variables of factors, assuming no multicollinearity
  step_mutate(across(all_of(logi_vars), as.factor)) %>%
  # converting logicals to factors for step_dummy
  step_dummy(all_of(logi_vars), -all_outcomes()) %>%
  # making dummy variables of factored logicals
  step_normalize(all_numeric_predictors()) # standardisation

# saving data specific recipe
gd_rec <- prep(rec, training = train_data)

## making prepared ("baked") test and train sets
gd_train <- bake(gd_rec, new_data = NULL)
gd_test  <- bake(gd_rec, new_data = test_data)
```

Gradient descent function

```r
### creating sigmoid function for logistic regression
sigmoid <- function(z) {
  1 / (1 + exp(-z))
}

### creating gradient descent function
grad_desc <- function(data, original_data, recipe, lr = 5e-2, epochs = 600,
                      updates = TRUE, lambda = 0.01, penalty = "l2") {
  # getting target
  target <- outcome_names(recipe)

  ## preparing predictor (X) data
  # removing the target column if necessary
  if (target %in% names(data)) {
    data <- data %>% select(-!!sym(target))
  }
```

```r
# converting data frame to matrix
X <- as.matrix(data)
X <- cbind(intercept = 1, X) # adding an intercept column

# getting new pred names (incl. factor levels)
all_preds <- colnames(X)

## preparing target (y) data
# getting the target from the original data
y <- original_data[[target]]
y <- as.numeric(y) # ensuring binary values

## preparing rest of vars for regression
# initialising coefficients
coeffs <- matrix(rep(0, ncol(X)), nrow = ncol(X))

# getting amount of training data
n <- nrow(X)

# creating tracker for keeping a log of performance
loss_tracker <- numeric(epochs)

## performing gradient descent
for (i in 1:epochs) {
  # calculating loss and gradient
  z <- X %*% coeffs
  y_hat <- sigmoid(z)

  # calculating cross-entropy loss
  eps <- 1e-8 # avoiding log(0)
  loss <- -mean(y * log(y_hat + eps) + (1 - y) * log(1 - y_hat + eps))

  # adding regularisation to the loss
  if (penalty == "l1") {
    loss <- loss + lambda * sum(abs(coeffs[-1])) # L1 regularisation
  } else if (penalty == "l2") {
    loss <- loss + lambda * sum(coeffs[-1]^2)    # L2 regularisation
  }

  # derivative of cross-entropy loss
  err <- y_hat - y
  grad <- (1 / n) * t(X) %*% err # getting new gradient

  # adding regularisation to the gradient
  if (penalty == "l1") {
    # derivative for L1 regularisation: lambda * sign(coeffs)
    grad_penalty <- matrix(0, nrow = nrow(coeffs), ncol = 1)
    grad_penalty[-1] <- lambda * sign(coeffs[-1]) # not including the coefficient
    grad <- grad + grad_penalty
  } else if (penalty == "l2") {
    # derivative for L2 regularisation: 2 * lambda * coeffs
    grad_penalty <- matrix(0, nrow = nrow(coeffs), ncol = 1)
    grad_penalty[-1] <- 2 * lambda * coeffs[-1] # not including the coefficient
```

```r
      grad <- grad + grad_penalty
    } # NB: else if (penalty == "none") {'skip this section'}

    # updating coefficients and loss tracker
    coeffs <- coeffs - lr * grad
    loss_tracker[i] <- loss

    # displaying current loss
    if (i %% (epochs / 10) == 0 || i == epochs) {
      if (updates) cat("\n Epoch:", i, ", Loss:", loss, "\n")

      # checking for divergence/ sigmoid saturation
      if (is.na(loss) || is.infinite(loss)) {
        warning("divergence occurring/ sigmoid saturating")
        break
      }
    }
  }

  ## outputting trained model
  coef_names <- colnames(X)
  rownames(coeffs) <- coef_names # naming coeffs

  # preparing list of all necessary info for model_eval and interpretation
  model_output <- list(
    coefficients = coeffs,
    loss_tracker = loss_tracker,
    target = target,
    final_coefficient_names = coef_names
  )
  return(model_output)
}
```

Evaluation function

```r
### creating function to evaluate the model on test data
eval_model <- function(model, ml_test_data, original_test_data, cutoff = 0.5) {

  ## preparing test predictor (X_test) data
  # remove the target column from the preprocessed test data frame
  if (model$target %in% names(ml_test_data)) {
      ml_test_data <- ml_test_data %>% select(-!!sym(model$target))
  }

  # convert data frame to matrix
  X_test <- as.matrix(ml_test_data)
  X_test <- cbind(intercept = 1, X_test) # adding an intercept column

  ## preparing binary test target (y_test)
  y_test <- original_test_data[[model$target]]
  y_test <- as.numeric(y_test) # ensuring binary values

  ## making predictions and saving outputs
  coeffs <- model$coefficients
  metrics <- list()

  # calculating predictions
  z <- X_test %*% coeffs
  probs <- sigmoid(z)
  pred_class <- ifelse(probs > cutoff, 1, 0)

  # converting back to factors for confusion matrix
  pred_class_factor <- factor(pred_class, levels = c(0, 1))
  y_test_factor <- factor(y_test, levels = c(0, 1))

  # creating confusion matrix and accuracy
  conf_matrix <- table(Actual = y_test_factor, Predicted = pred_class_factor)
  accuracy <- sum(diag(conf_matrix)) / sum(conf_matrix)

  # saving metrics
  metrics$probs <- probs
  metrics$pred_class <- ifelse(probs > cutoff, 1, 0)
  metrics$confusion_matrix <- conf_matrix
  metrics$accuracy <- accuracy

  return(metrics)
}
```

Parameter tuning

```
# tuning options
tuning_epochs <- 500
lr_opts <- c(5e-3, 1e-2, 5e-2, 1e-1)
penalty_opts <- c("l1", "l2", "none")
lambda_opts <- c(0.005, 0.01, 0.05, 0.1)

# setting low value for best parameters
best_params <- list(best_acc=0, best_lr=0, best_epochs=1000,
                    best_penalty=NULL, best_lambda=0)

# running through tuning parameters to get best options based on test accuracy
for (curr_lr in lr_opts) {
  for (curr_penalty in penalty_opts) {
    if (curr_penalty == "none") {
      curr_lambda <- "none"

      # training
      curr_model <- grad_desc(gd_train, train_data, gd_rec, lr=curr_lr,
                              updates=FALSE, epochs=tuning_epochs)

      # evaluation
      curr_eval <- eval_model(curr_model, gd_test, test_data)
      if (curr_eval$accuracy > best_params[["best_acc"]]) {
        best_params <- list(best_acc=curr_eval$accuracy, best_lr=curr_lr,
                            best_epochs=1000, best_penalty=curr_penalty,
                            best_lambda=curr_lambda)
```

```r
      }
    } else {
      # looping through lambda options for l1 or l2 penalties
      for (curr_lambda in lambda_opts) {
        # training
        curr_model <- grad_desc(gd_train, train_data, gd_rec, lr=curr_lr,
                                updates=FALSE, penalty=curr_penalty,
                                lambda=curr_lambda, epochs=tuning_epochs)

        # evaluation
        curr_eval <- eval_model(curr_model, gd_test, test_data)
        if (curr_eval$accuracy > best_params[["best_acc"]]) {
          best_params <- list(best_acc=curr_eval$accuracy, best_lr=curr_lr,
                              best_epochs=1000, best_penalty=curr_penalty,
                              best_lambda=curr_lambda)
        }
      }
    }
  }
}

# checking what number of epochs is best for training
epoch_opts <- seq(300, 1000, by = 100)
for (curr_epochs in epoch_opts) {
  curr_model <- grad_desc(gd_train, train_data, gd_rec, lr=best_params$best_lr,
                          updates=FALSE, penalty=best_params$best_penalty,
                          lambda=best_params$best_lambda, epochs=curr_epochs)

  curr_eval <- eval_model(curr_model, gd_test, test_data)
  if (curr_eval$accuracy > best_params[["best_acc"]]) {
    best_params <- list(best_acc=curr_eval$accuracy, best_lr=best_params$best_lr,
                        best_epochs=curr_epochs,
                        best_penalty=best_params$best_penalty,
                        best_lambda=best_params$best_lambda)
  }
}

# outputting best parameters for training
cat("Best Parameters \n",
    "Learning Rate: ", best_params[["best_lr"]], "\n",
    "Epochs: ", best_params[["best_epochs"]], "\n",
    "Penalty type: ", best_params[["best_penalty"]], "\n",
    "Lambda: ", best_params[["best_lambda"]], "\n",
    "Test Accuracy: ", best_params[["best_acc"]], "\n")
```

```
## Best Parameters
##  Learning Rate:  0.05
##  Epochs:  600
##  Penalty type:  l2
##  Lambda:  0.01
##  Test Accuracy:  0.7393744
```

Running gradient decent algorithm and evaluating performance

```r
# training the model
model_gd <- grad_desc(gd_train, train_data, gd_rec)
```

```
##
##  Epoch: 60 , Loss: 0.5785789
##
##  Epoch: 120 , Loss: 0.5536346
##
##  Epoch: 180 , Loss: 0.5456513
##
##  Epoch: 240 , Loss: 0.5424951
##
##  Epoch: 300 , Loss: 0.5410966
##
##  Epoch: 360 , Loss: 0.5404345
##
##  Epoch: 420 , Loss: 0.5401082
##
##  Epoch: 480 , Loss: 0.5399431
##
##  Epoch: 540 , Loss: 0.5398582
##
##  Epoch: 600 , Loss: 0.539814
```

```r
cat("\n Trained Model Coefficients: \n")
```

```
##
##  Trained Model Coefficients:
```

```r
print(model_gd$coefficients) # printing coefficients for interpretation
```

```
##                                  [,1]
## intercept                0.3692743787
## desired_duration        -0.2591988714
## blurb_characters        -0.0003004802
## year_centred             0.1942564309
## log_goal_usd            -0.6427866039
## category_Art             0.0411857316
## category_Comics          0.1379821520
## category_Crafts         -0.2332770367
## category_Dance          -0.0196791202
## category_Design         -0.0079688150
## category_Fashion         0.0081479342
## category_Film...Video    0.0768363726
## category_Food           -0.1702037374
## category_Games          -0.0003859389
## category_Journalism     -0.1320715746
## category_Music           0.1672406522
## category_Photography    -0.1053812705
## category_Publishing      0.0487038146
## category_Technology     -0.0406704754
## category_Theater         0.0025785017
## currency_AUD            -0.0221180562
## currency_CAD            -0.0103269523
## currency_CHF           -0.0177472166
## currency_DKK           -0.0004986925
## currency_EUR           -0.0982995106
## currency_GBP            0.0445714976
## currency_HKD            0.1025669045
## currency_JPY            0.0157384188
## currency_MXN           -0.1158821043
## currency_NOK           -0.0149273757
## currency_NZD           -0.0131485753
## currency_PLN           -0.0247170312
## currency_SEK           -0.0076781026
## currency_SGD            0.0123668570
## currency_USD            0.0531418521
## day_launched_Friday    -0.0131278631
## day_launched_Monday    -0.0126664802
## day_launched_Saturday  -0.0374918297
## day_launched_Sunday    -0.0345234871
## day_launched_Thursday   0.0050418956
## day_launched_Tuesday    0.0558426260
## day_launched_Wednesday  0.0093771238
## staff_pick_TRUE.        0.6951279400
## video_TRUE.             0.4376526667
```

```
# evaluating model
model_gd_eval <- eval_model(model_gd, gd_test, test_data)
cat("\n Model Evaluation Results: \n")
```

```
##
##  Model Evaluation Results:
```

```
print(model_gd_eval[c("confusion_matrix", "accuracy")])
```

```
## $confusion_matrix
##       Predicted
## Actual     0     1
##      0 10294  5843
##      1  4005 17644
##
## $accuracy
## [1] 0.7393744
```

This model ended up having similar accuracy to our interpretable model (and higher than baseline), which we found to be very good, given we tuned our gradient descent algorithm from scratch. Similar to the interpretable and baseline models; being a staff pick, including a video, including more blurb characters in a description and having shorter campaigns predicted higher chances of success.

4. Prediction Focused Models

4.1 Boosted Trees

As the dataset we have is huge for a complex prediction model (around 200,000 rows), we randomly subset 20% of the data to make code run faster, (that is around 40,000 data points which is still a sufficient amount of data to work with).

```
set.seed(123)
kick_data<- sample_n(kick_data, 0.2 * nrow(kick_data))
```

```r
View(kick_data) # 38000 entries
# check imbalance
kick_data %>% count(state)
# it is not severely imbalanced, so we will use accuracy as our metric for the model
```

```
##   state     n
## 1 FALSE 16217
## 2  TRUE 21569
```

```r
kick_data$state <- as.factor(kick_data$state)
```

In this section, we will build our prediction heavy model where our main goal is to produce a model that is focused on predictive accuracy. Given our large dataset, we will try tree-based models (which are quite good for classification).

```r
# let us try xG boost
library(caret)
library(xgboost)
```

```
## Warning: package 'xgboost' was built under R version 4.3.3

##
## Attaching package: 'xgboost'

## The following object is masked from 'package:dplyr':
##
##     slice
```

```r
library(tidymodels)

# split dataset 80% training and 20% testing
set.seed(123)
kick_data_split <- initial_split(kick_data, prop = 0.8, strata = state)
kick_data_train <- training(kick_data_split)
kick_data_test <- testing(kick_data_split)
# check the split
kick_data_split %>%
  summary()
```

```
##        Length Class      Mode
## data      12  data.frame list
## in_id  30228  -none-     numeric
## out_id     1  -none-     logical
## id         1  tbl_df     list
```

```r
# Our first model is xgboost, here what we will do is to tune all the parameters
# that has parameter set to tune() using a latin hypercube grid search, note as
# the outcome variable we are trying to predict "state" is a binary variable, our
# task is a classification task

# Even with 20% of the entire dataset, that is still about 40,000 entries which is
# very large. Therefore, running an entire grid search will be quite computational
# heavy. On the other hand, a random search also has its downside of missing important
# combinations of parameters. Therefore, we will use a latin hypercube grid search
# which is a compromise between the two, allowing us to search for the optimal
# parameters while also being computationally efficient.
```

```r
model_1 <- boost_tree(
  trees = 1000,
  tree_depth = tune(), min_n= tune(),
  loss_reduction = tune(),
  sample_size = tune(), mtry = tune(),
  learn_rate = tune()
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

#define the latin hypercube grid, with size 30
model_1_grid <- grid_latin_hypercube(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  finalize(mtry(), kick_data_train),
  learn_rate(),
  size = 30
)
```

```
## Warning: `grid_latin_hypercube()` was deprecated in dials 1.3.0.
## i Please use `grid_space_filling()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```r
# set up the workflow
model_1_wf <- workflow() %>%
  add_model(model_1) %>%
  add_formula(state ~ .)
```

```r
# create a 10 fold cross validation object while stratifying state to ensure that
# each fold has a similar proportion of the outcome classes in state
kick_data_folds <- vfold_cv(kick_data_train, strata = state)
# for parallel processing to achieve faster training
library(doParallel)
```

```
## Warning: package 'doParallel' was built under R version 4.3.3

## Loading required package: foreach

## Warning: package 'foreach' was built under R version 4.3.3

##
## Attaching package: 'foreach'

## The following objects are masked from 'package:purrr':
##
##     accumulate, when

## Loading required package: iterators

## Warning: package 'iterators' was built under R version 4.3.3

## Loading required package: parallel
```

```r
doParallel::registerDoParallel()
```

```
model_1_tune <- tune_grid(
  model_1_wf,
  resamples = kick_data_folds,
  grid = model_1_grid,
  control = control_grid(
    save_pred = TRUE,
    verbose = TRUE,
    allow_par = TRUE
  )
)
```

```
## Warning: ! tune detected a parallel backend registered with foreach but no backend
##   registered with future.
## i Support for parallel processing with foreach was soft-deprecated in tune
##   1.2.1.
## i See ?parallelism (`?tune::parallelism()`) to learn more.
```

```
model_1_tune
```

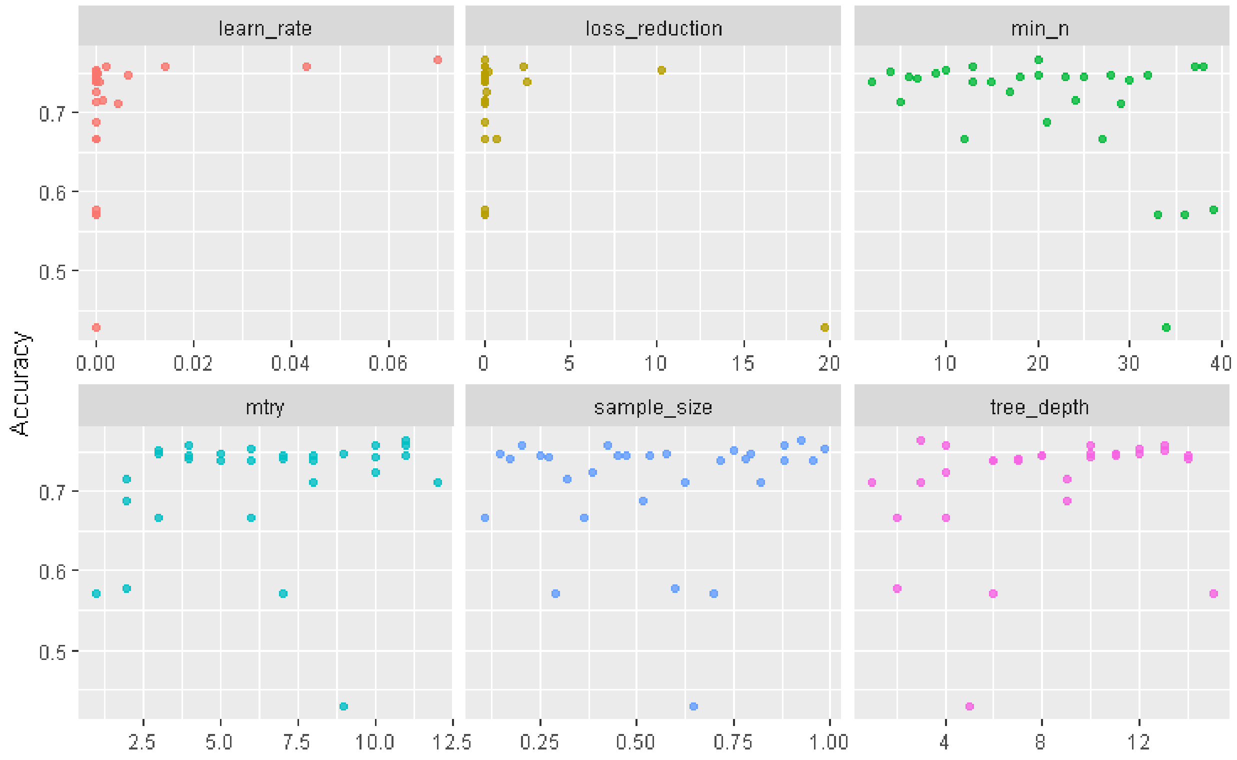
```
## # Tuning results
## # 10-fold cross-validation using stratification
## # A tibble: 10 x 5
##    splits               id     .metrics           .notes           .predictions
##    <list>               <chr>  <list>             <list>           <list>
##  1 <split [27204/3024]> Fold01 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
##  2 <split [27204/3024]> Fold02 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
##  3 <split [27204/3024]> Fold03 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
##  4 <split [27205/3023]> Fold04 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
##  5 <split [27205/3023]> Fold05 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
##  6 <split [27206/3022]> Fold06 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
##  7 <split [27206/3022]> Fold07 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
##  8 <split [27206/3022]> Fold08 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
##  9 <split [27206/3022]> Fold09 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
## 10 <split [27206/3022]> Fold10 <tibble [90 x 10]> <tibble [0 x 3]> <tibble>
```

```
collect_metrics(model_1_tune)
```

```
## # A tibble: 90 x 12
##     mtry min_n tree_depth    learn_rate loss_reduction sample_size .metric
##    <int> <int>      <int>         <dbl>          <dbl>       <dbl> <chr>
##  1     2    21          9 0.00000217       0.000000162       0.515 accuracy
##  2     2    21          9 0.00000217       0.000000162       0.515 brier_class
##  3     2    21          9 0.00000217       0.000000162       0.515 roc_auc
##  4     6    27          2 0.00000000494    0.722             0.363 accuracy
##  5     6    27          2 0.00000000494    0.722             0.363 brier_class
##  6     6    27          2 0.00000000494    0.722             0.363 roc_auc
##  7     2    39          2 0.0000403        0.00000447        0.598 accuracy
##  8     2    39          2 0.0000403        0.00000447        0.598 brier_class
##  9     2    39          2 0.0000403        0.00000447        0.598 roc_auc
## 10     5     2          6 0.000505         2.50              0.956 accuracy
## # i 80 more rows
## # i 5 more variables: .estimator <chr>, mean <dbl>, n <int>, std_err <dbl>,
## #   .config <chr>
```

```
# visualise results
model_1_tune %>%
```

```r
  collect_metrics() %>%
  filter(.metric == "accuracy") %>%
  select(mean, mtry:sample_size) %>%
  pivot_longer(mtry:sample_size,
               values_to = "value",
               names_to = "parameter") %>%
  ggplot(aes(x = value, y = mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE) + facet_wrap(~ parameter, scales = "free_x") +
             labs(x = NULL, y = "Accuracy")
```



```r
# Looks like higher values of tree depth were better, from this plot, it looks like
# there are several combinations of parameters that can perform well, so we will select
# the best performing model based on the accuracy metric as the dataset is fairly
# balanced. (We have also tried using ROC_AUC but it ended up with the same results)


show_best(model_1_tune, metric = "accuracy")
```

```
## # A tibble: 5 x 12
##    mtry min_n tree_depth learn_rate loss_reduction sample_size .metric
##   <int> <int>      <int>      <dbl>          <dbl>       <dbl> <chr>
## 1    11    20          3   6.99e- 2       3.08e-10       0.924 accuracy
## 2    10    13         13   1.99e- 3       2.23e+ 0       0.423 accuracy
## 3     4    38          4   1.41e- 2       6.67e- 7       0.880 accuracy
## 4    11    37         10   4.32e- 2       1.08e- 4       0.204 accuracy
## 5     6    10         12   8.95e-10       1.02e+ 1       0.986 accuracy
## # i 5 more variables: .estimator <chr>, mean <dbl>, n <int>, std_err <dbl>,
## #   .config <chr>
```

```r
# Select best model
best_acc <- select_best(model_1_tune, metric = "accuracy")
best_acc
```

```
## # A tibble: 1 x 7
##    mtry min_n tree_depth learn_rate loss_reduction sample_size .config
##   <int> <int>      <int>      <dbl>          <dbl>       <dbl> <chr>
## 1    11    20          3     0.0699       3.08e-10       0.924 Preprocessor1_Mo~
```

```r
# finalise tunable workflow with the above parameters
model_1_final <- finalize_workflow(model_1_wf, best_acc)

model_1_final
```

```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## state ~ .
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = 11
##   trees = 1000
##   min_n = 20
##   tree_depth = 3
##   learn_rate = 0.0698908064444402
##   loss_reduction = 3.08362445001625e-10
##   sample_size = 0.924187292442657
##
## Computational engine: xgboost
```

```r
# we can look into what are the most important parameters for variable importance
library(vip)
```

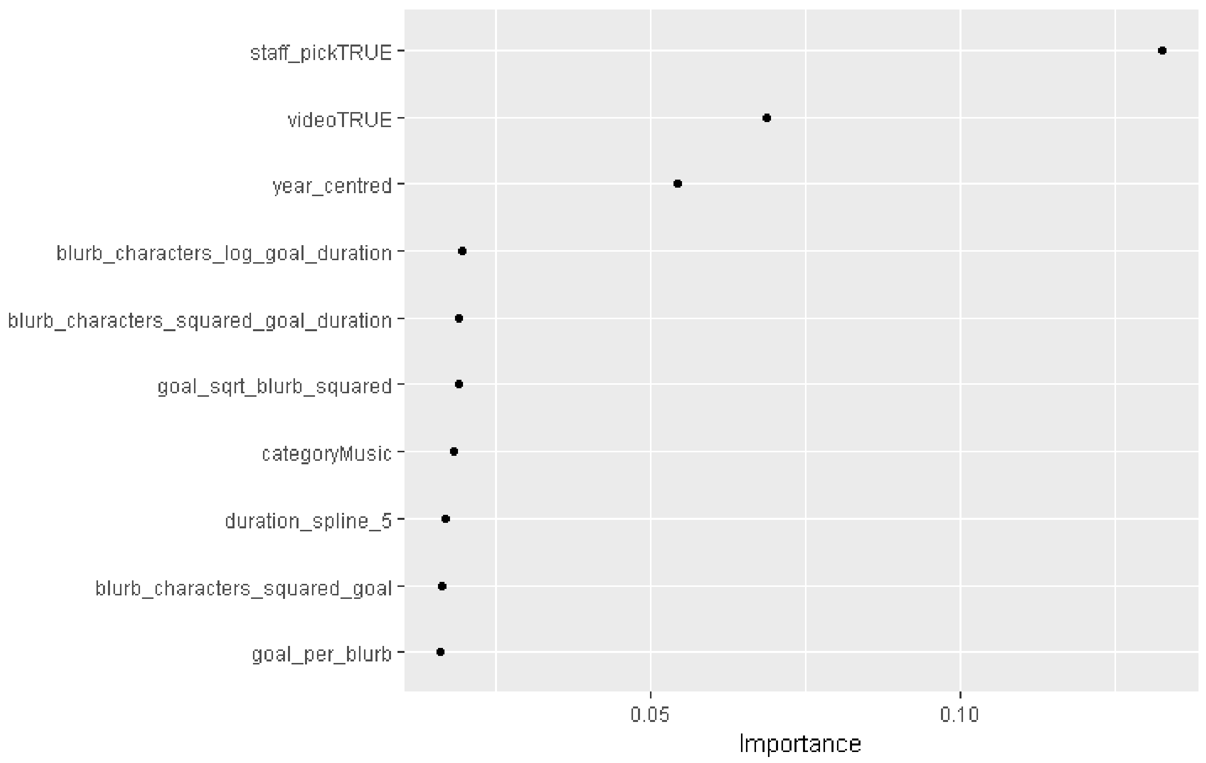
```
## Warning: package 'vip' was built under R version 4.3.3
```

```
##
## Attaching package: 'vip'
```

```
## The following object is masked from 'package:utils':
##
##     vi
```

```r
model_1_final %>%
  fit(data = kick_data_train) %>%
  pull_workflow_fit() %>%
  vip(geom = "point")
```

```
## Warning: `pull_workflow_fit()` was deprecated in workflows 0.2.3.
## i Please use `extract_fit_parsnip()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```



```r
# we see that goal is the most important variable, followed by staff pick and
# year. These results are quite intuitive as variables like goal and staff pick
# should affect the chances of a Kickstarter project being successful or not.
# note projects with staff pick on Kickstarter website get more spotlights in
# the Kickstarter website.

# evaluate model on test set
final_res <- last_fit(model_1_final, kick_data_split)
collect_metrics(final_res)
```

```
## # A tibble: 3 x 4
##   .metric     .estimator .estimate .config
##   <chr>       <chr>          <dbl> <chr>
## 1 accuracy    binary         0.769 Preprocessor1_Model1
## 2 roc_auc     binary         0.847 Preprocessor1_Model1
## 3 brier_class binary         0.156 Preprocessor1_Model1
```

4.2 Random Forest Model

```r
set.seed(123)
# Now lets try random forest to see if it gives us higher accuracy, again similar
# to the above we will utilise the latin hypercube grid, but obviously will less
# parameters to tune.
model_2 <- rand_forest(
  trees = 1000,
  min_n = tune(),
  mtry = tune()
) %>%
  set_engine("ranger") %>%
  set_mode("classification")

model_2_grid <- grid_latin_hypercube(
  min_n(),
  finalize(mtry(), kick_data_train),
  size = 30
)

# set workflow
model_2_wtf <- workflow() %>%
  add_model(model_2) %>%
  add_formula(state ~ .)
```

```r
library(doParallel)
library(ranger)
```

```
## Warning: package 'ranger' was built under R version 4.3.3
```

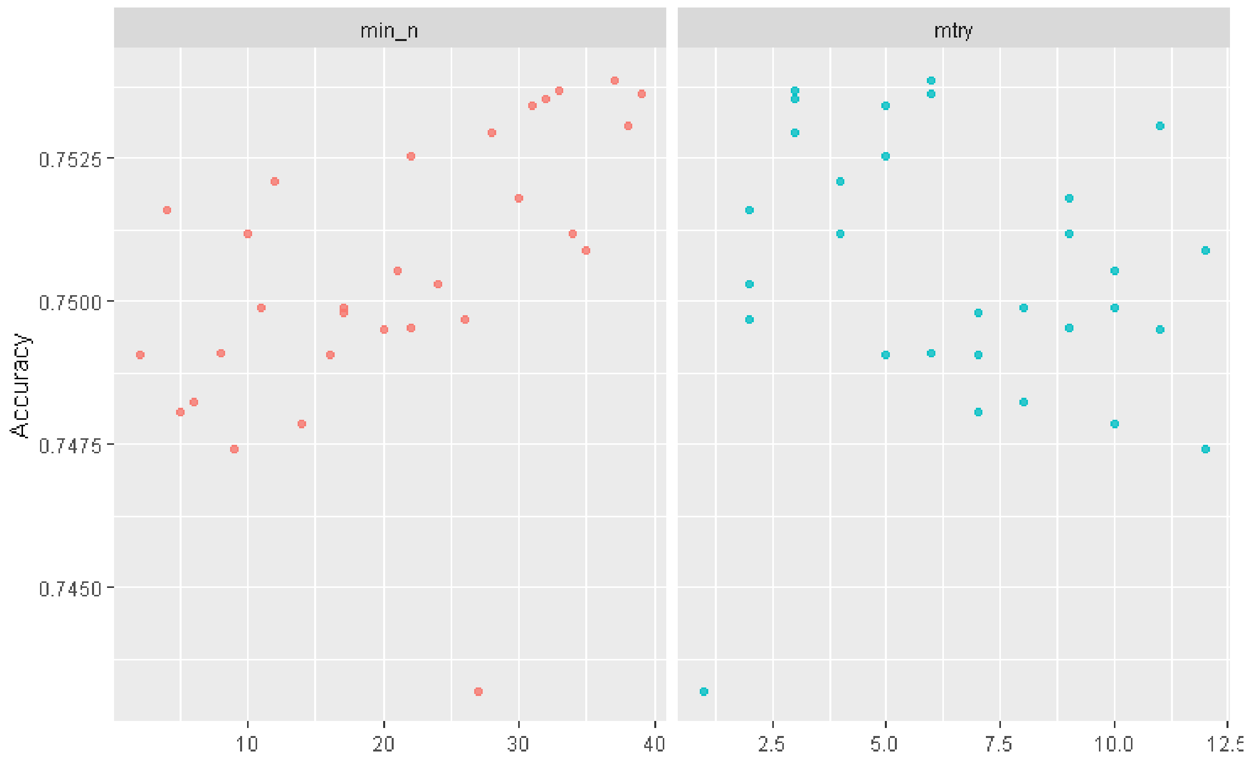
```r
doParallel::registerDoParallel()
model_2_tune <- tune_grid(
  model_2_wtf,
  resamples = kick_data_folds,
  grid = model_2_grid,
  control = control_grid(
    save_pred = TRUE,
    verbose = TRUE,
    allow_par = TRUE
  )
)
```

```
## Warning: ! tune detected a parallel backend registered with foreach but no backend
##   registered with future.
## i Support for parallel processing with foreach was soft-deprecated in tune
##   1.2.1.
## i See ?parallelism (`?tune::parallelism()`) to learn more.
```

```r
model_2_tune
```

```
## # Tuning results
## # 10-fold cross-validation using stratification
## # A tibble: 10 x 5
##    splits               id     .metrics          .notes           .predictions
##    <list>               <chr>  <list>            <list>           <list>
##  1 <split [27204/3024]> Fold01 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
##  2 <split [27204/3024]> Fold02 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
##  3 <split [27204/3024]> Fold03 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
##  4 <split [27205/3023]> Fold04 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
##  5 <split [27205/3023]> Fold05 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
##  6 <split [27206/3022]> Fold06 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
##  7 <split [27206/3022]> Fold07 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
##  8 <split [27206/3022]> Fold08 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
##  9 <split [27206/3022]> Fold09 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
## 10 <split [27206/3022]> Fold10 <tibble [90 x 6]> <tibble [0 x 3]> <tibble>
```

```r
#Visualise results
model_2_tune %>%
  collect_metrics() %>%
  filter(.metric == "accuracy") %>%
  select(mean, mtry:min_n) %>%
  pivot_longer(mtry:min_n,
               values_to = "value",
               names_to = "parameter") %>%
  ggplot(aes(x = value, y = mean, color = parameter)) +
  geom_point(alpha = 0.8, show.legend = FALSE) + facet_wrap(~ parameter, scales = "free_x") +
             labs(x = NULL, y = "Accuracy")
```



```r
# we do not see any particular pattern here

collect_metrics(model_2_tune)
```

```
## # A tibble: 90 x 8
##     mtry min_n .metric     .estimator  mean     n std_err .config
##    <int> <int> <chr>       <chr>      <dbl> <int>   <dbl> <chr>
##  1    12    35 accuracy    binary     0.751    10 0.00176 Preprocessor1_Model01
##  2    12    35 brier_class binary     0.165    10 0.00108 Preprocessor1_Model01
##  3    12    35 roc_auc     binary     0.829    10 0.00226 Preprocessor1_Model01
##  4     5    22 accuracy    binary     0.753    10 0.00169 Preprocessor1_Model02
##  5     5    22 brier_class binary     0.165    10 0.00101 Preprocessor1_Model02
##  6     5    22 roc_auc     binary     0.830    10 0.00209 Preprocessor1_Model02
##  7    10    17 accuracy    binary     0.750    10 0.00167 Preprocessor1_Model03
##  8    10    17 brier_class binary     0.166    10 0.00106 Preprocessor1_Model03
##  9    10    17 roc_auc     binary     0.827    10 0.00218 Preprocessor1_Model03
## 10     6    39 accuracy    binary     0.754    10 0.00187 Preprocessor1_Model04
## # i 80 more rows
```

```r
show_best(model_2_tune, metric = "accuracy")
```

```
## # A tibble: 5 x 8
##    mtry min_n .metric  .estimator  mean     n std_err .config
##   <int> <int> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1     6    37 accuracy binary     0.754    10 0.00190 Preprocessor1_Model11
## 2     3    33 accuracy binary     0.754    10 0.00190 Preprocessor1_Model08
## 3     6    39 accuracy binary     0.754    10 0.00187 Preprocessor1_Model04
## 4     3    32 accuracy binary     0.754    10 0.00218 Preprocessor1_Model26
## 5     5    31 accuracy binary     0.753    10 0.00215 Preprocessor1_Model16
```

```r
best_acc_2 <- select_best(model_2_tune, metric = "accuracy")
best_acc_2
```

```
## # A tibble: 1 x 3
##    mtry min_n .config
##   <int> <int> <chr>
## 1     6    37 Preprocessor1_Model11
```

```r
model_2_final <- finalize_workflow(model_2_wtf, best_acc_2)

model_2_final
```

```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
## state ~ .
##
## -- Model -----------------------------------------------------------------------
## Random Forest Model Specification (classification)
##
## Main Arguments:
##   mtry = 6
##   trees = 1000
##   min_n = 37
##
## Computational engine: ranger
```

```r
model_2_final_res <- last_fit(model_2_final, kick_data_split)
collect_metrics(model_2_final_res)
```

```
## # A tibble: 3 x 4
##   .metric     .estimator .estimate .config
##   <chr>       <chr>          <dbl> <chr>
## 1 accuracy    binary         0.756 Preprocessor1_Model1
## 2 roc_auc     binary         0.831 Preprocessor1_Model1
## 3 brier_class binary         0.165 Preprocessor1_Model1
```

As can be seen, our prediction-heavy model achieved an accuracy of 78% with xgBoost and 75% with the Random Forest model. This is an acceptable result, given that we have managed to achieve a higher accuracy than any of the other models we have built here. We see that the random forest model we created is not as accurate as the xgboost model, with an accuracy of 75% on the test set. This is likely due to the fact that random forest is a bagging method and, therefore, does not perform as well as boosting methods like xgboost, which does require more careful hyperparameter tuning, albeit if done well, can outperform bagging methods like random forest.

Therefore, it is good to see that our prediction heavy model(boosted trees) achieved a higher accuracy than the baseline models by a relatively large margin. However, it is important to note that some of the academic papers we have read use other methods as well. For example, "KickPredict: Predicting Kickstarter Success Kevin Chen, Brock Jones, Isaac Kim, Brooklyn Schlamp Dept. of Computing and Mathematical Sciences California Institute of Technology 1200 E California Blvd Pasadena" used Support Vector Machine models using the ScikitLearn package on Python, also, "Launch Hard or Go Home! Predicting the Success of Kickstarter Campaigns Vincent Etter Matthias Grossglauser Patrick Thiran School of Computer and Communication Sciences École Polytechnique Fédérale de Lausanne (EPFL)" covered more advanced modelling methods like the Markov Model. Although we did not cover these methods in this project, as it was slightly more advanced and were methods that we did not cover in our course, it is worth noting that these methods could potentially lead to even higher accuracy than the models we have built here.

Lastly, we must note the computational constraint we had as well, as we have only used 20% of our total data; if we had more computing power we may have been able to achieve even higher accuracy.

Note the method used was based on a method used here "https://juliasilge.com/blog/xgboost-tune-volleyball/"

5. High Dimensional Model

As our dataset has nearly 200,000 entries with only 10 predictors, our dataset is not exactly 'high dimensional.' As such, to create a high dimensional model, we use only 1% of our total dataset (2000 entries) and create additional predictors through non-linear transformations, interaction terms and local smoothing. In total, our 'high dimensional' dataset ended up having around 2000 entries with 95 predictors that were created from the original 10. We then fit models that are suited to dealing with high dimensional data.

```r
kick_data <- read.csv('cleaned310.csv')

#Centred year since later found it made more sense as a centred predictor
kick_data$year_centred <- kick_data$year_launched - 2009
kick_data <- kick_data %>% select(-year_launched)

#Made the following columns the correct type for later models
kick_data <- kick_data %>%
  mutate(
    category = as.factor(category),
    currency = as.factor(currency),
    day_launched = as.factor(day_launched),
    month_launched = as.factor(month_launched),
    state = as.factor(state)
  )

# As the dataset we have is huge for tree models with nearly 200,000 rows, I am
# going to randomly subset 1% of the data to make it easier to work with, (that
# is around 2,000 data points
set.seed(123)
kick_data<- sample_n(kick_data, 0.01 * nrow(kick_data))
#View(kick_data)
# check imbalance
kick_data %>% count(state)
# it is not severely imbalanced so we will just use a threshold of 0.5 to measure accuracy
```

```
##   state    n
## 1 FALSE  763
## 2  TRUE 1169
```

```r
# Now we will add more predictor variables by introducing non-linear transformations,
# interaction terms, and local smoothing

# First we will introduce some non-linear transformations
kick_data <- kick_data %>%
  mutate(
    goal_squared = (goal_usd)^2,
    goal_log= log(goal_usd),
    goal_sqrt = sqrt(goal_usd),
    duration_log = log(desired_duration),
    duration_sqrt = sqrt(desired_duration),
    duration_squared = (desired_duration)^2,
    blurb_characters_log = log(blurb_characters),
    blurb_characters_sqrt = sqrt(blurb_characters),
    blurb_characters_squared = (blurb_characters)^2,
    blurb_per_day = blurb_characters / desired_duration,
    goal_per_day = goal_usd / desired_duration,
    goal_per_blurb = goal_usd / blurb_characters,
    duration_per_goal = desired_duration / goal_usd,
    goal_cubed = (goal_usd)^3,
    duration_cubed = desired_duration^3,
    blurb_characters_cubed = blurb_characters^3,
    goal_per_day_squared = (goal_usd / desired_duration)^2,
    goal_per_day_cubed = (goal_usd / desired_duration)^3,
    goal_per_day_log = log(goal_usd / desired_duration),
    goal_per_day_sqrt = sqrt(goal_usd / desired_duration),
    goal_quartic = (goal_usd)^4,
    duration_quartic = (desired_duration)^4,
    blurb_characters_quartic = (blurb_characters)^4,

  )

# Now we will introduce some interaction terms


kick_data <- kick_data %>%
  mutate(
    goal_duration = goal_usd * desired_duration,
    goal_blurb = goal_usd * blurb_characters,
    blurb_duration = blurb_characters * desired_duration,
    goal_blurb_duration = goal_usd * blurb_characters * desired_duration,
    goal_log_duration = goal_log * desired_duration,
    goal_log_blurb = goal_log * blurb_characters,
    goal_log_blurb_duration = goal_log * blurb_characters * desired_duration,
    goal_sqrt_duration = goal_sqrt * desired_duration,
    goal_sqrt_blurb = goal_sqrt * blurb_characters,
    goal_sqrt_blurb_duration = goal_sqrt * blurb_characters * desired_duration,
    goal_squared_duration = goal_squared * desired_duration,
    goal_squared_blurb = goal_squared * blurb_characters,
    goal_squared_blurb_duration = goal_squared * blurb_characters * desired_duration,
    duration_log_blurb = duration_log * blurb_characters,
    duration_log_goal = duration_log * goal_usd,
    duration_log_goal_blurb = duration_log * goal_usd * blurb_characters,
    duration_sqrt_blurb = duration_sqrt * blurb_characters,
    duration_sqrt_goal = duration_sqrt * goal_usd,
    duration_sqrt_goal_blurb = duration_sqrt * goal_usd * blurb_characters,
    duration_squared_blurb = duration_squared * blurb_characters,
    duration_squared_goal = duration_squared * goal_usd,
    duration_squared_goal_blurb = duration_squared * goal_usd * blurb_characters,
    blurb_characters_log_duration = blurb_characters_log * desired_duration,
    blurb_characters_log_goal = blurb_characters_log * goal_usd,
    blurb_characters_log_goal_duration = blurb_characters_log * goal_usd * desired_duration,
    blurb_characters_sqrt_duration = blurb_characters_sqrt * desired_duration,
    blurb_characters_sqrt_goal = blurb_characters_sqrt * goal_usd,
    blurb_characters_sqrt_goal_duration = blurb_characters_sqrt * goal_usd * desired_duration,
    blurb_characters_squared_duration = blurb_characters_squared * desired_duration,
    blurb_characters_squared_goal = blurb_characters_squared * goal_usd,
    blurb_characters_squared_goal_duration = blurb_characters_squared * goal_usd * desired_duration,
    blurb_per_day_duration = blurb_per_day * desired_duration,
    blurb_per_day_goal = blurb_per_day * goal_usd,
    blurb_per_day_goal_duration = blurb_per_day * goal_usd * desired_duration,
    goal_per_day_duration = goal_per_day * desired_duration,
    goal_per_day_blurb = goal_per_day * blurb_characters,
    goal_per_day_blurb_duration = goal_per_day * blurb_characters * desired_duration,
    goal_log_duration_squared = goal_log * duration_squared,
    goal_log_blurb_squared = goal_log * blurb_characters_squared,
    goal_log_blurb_duration_squared = goal_log * blurb_characters_squared * desired_duration,
    goal_sqrt_blurb_squared = goal_sqrt * blurb_characters_squared,
    goal_sqrt_blurb_duration_squared = goal_sqrt * blurb_characters_squared * desired_duration,
    goal_squared_blurb_squared = goal_squared * blurb_characters_squared,
    goal_squared_blurb_duration_squared = goal_squared * blurb_characters_squared * desired_duration,
    goal_sqrt_blurb_log = goal_sqrt * blurb_characters_log,
    goal_squared_blurb_log = goal_squared * blurb_characters_log,
    goal_log_blurb_log = goal_log * blurb_characters_log

  )



# Now we will introduce some local smoothing
library(splines)

# Add natural spline basis for 'goal'
goal_spline <- as.data.frame(ns(kick_data$goal_usd, df = 5))
colnames(goal_spline) <- paste0("goal_spline_", 1:5)

# Add natural spline basis for 'desired_duration'
duration_spline <- as.data.frame(ns(kick_data$desired_duration, df = 5))
colnames(duration_spline) <- paste0("duration_spline_", 1:5)

# Add natural spline basis for 'blurb_characters'
blurb_spline <- as.data.frame(ns(kick_data$blurb_characters, df = 5))
colnames(blurb_spline) <- paste0("blurb_spline_", 1:5)

# Bind to original data
kick_data <- cbind(kick_data, goal_spline, duration_spline, blurb_spline)
```

```r
# see how many columns we have
ncol(kick_data)
```

```
## [1] 96
```

```r
# We see we have 96 columns, with one of them being the outcome variable, so now
# we have increased it from 10 to 95 predictors, which is (relatively) high dimensional
# for 2000 data points.
```

```r
library(tidymodels)
library(glmnet)
```

```
## Warning: package 'glmnet' was built under R version 4.3.3
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loaded glmnet 4.1-8
```

```r
set.seed(123)
kick_data_split <- initial_split(kick_data, prop = 0.8, strata = state)
kick_data_train <- training(kick_data_split) %>% filter(complete.cases(.))
kick_data_test <- testing(kick_data_split) %>% filter(complete.cases(.))
# check the split
kick_data_split %>%
  summary()
```

```
##        Length Class      Mode
## data     96   data.frame list
## in_id  1545   -none-     numeric
## out_id    1   -none-     logical
## id        1   tbl_df     list
```

```r
# Convert to model matrix and make sure intercept is not duplicated
x_train <- model.matrix(state ~ . - 1, data = kick_data_train)
y_train <- kick_data_train$state
x_test <- model.matrix(state ~ . - 1, data = kick_data_test)
y_test <- kick_data_test$state

# Fit ridge
ridge_model <- cv.glmnet(x_train, y_train, type.measure = "deviance", alpha = 0, family = "binomial")
ridge_predicted <- predict(ridge_model, s=ridge_model$lambda.1se, newx = x_test)

ridge_predicted <- ifelse(ridge_predicted > 0.5, "TRUE", "FALSE")
# Evaluate model performance
mean(ridge_predicted == y_test)
```

```
## [1] 0.7055703
```

```r
# Fit lasso
lasso_model <- cv.glmnet(x_train, y_train, type.measure = "deviance", alpha = 1, family = "binomial")
lasso_predicted <- predict(lasso_model, s=lasso_model$lambda.1se, newx = x_test)
lasso_predicted <- ifelse(lasso_predicted > 0.5, "TRUE", "FALSE")
# Evaluate model performance
mean(lasso_predicted == y_test)
```

```
## [1] 0.6949602
```

```r
# Fit elastic net
elastic_net_model <- cv.glmnet(x_train, y_train, type.measure = "deviance", alpha = 0.5,
                               family = "binomial")
```

```
## Warning: from glmnet C++ code (error code -97); Convergence for 97th lambda
## value not reached after maxit=100000 iterations; solutions for larger lambdas
## returned
```

```r
elastic_net_predicted <- predict(elastic_net_model, s=elastic_net_model$lambda.1se, newx = x_test)
elastic_net_predicted <- ifelse(elastic_net_predicted > 0.5, "TRUE", "FALSE")
# Evaluate model performance
mean(elastic_net_predicted == y_test)
```

```
## [1] 0.7002653
```

```r
# Let us try a different values of alpha
list_of_fits <- list()
for (i in 0:10) {
  fit_name <- paste0("alpha", i/10)
  list_of_fits[[fit_name]] <- cv.glmnet(x_train, y_train, type.measure = "deviance",
                                        alpha = i/10, family = "binomial")
}
```

```
## Warning: from glmnet C++ code (error code -99); Convergence for 99th lambda
## value not reached after maxit=100000 iterations; solutions for larger lambdas
## returned
```

```r
results <- data.frame()
for (i in 0:10) {
  fit_name <- paste0("alpha", i/10)
  predicted <-
    predict(list_of_fits[[fit_name]], s=list_of_fits[[fit_name]]$lambda.1se, newx = x_test)
  accuracy <- mean(ifelse(predicted > 0.5, "TRUE", "FALSE") == y_test)
  temp <- data.frame(
    alpha = i / 10,
    accuracy = accuracy,
    fit_name = fit_name
  )
  results <- rbind(results, temp)
  }

results # we see alpha of 0.1 has the highest accuracy of 71%
```

```
##    alpha  accuracy fit_name
## 1    0.0 0.7002653   alpha0
## 2    0.1 0.7188329 alpha0.1
## 3    0.2 0.7055703 alpha0.2
## 4    0.3 0.7082228 alpha0.3
## 5    0.4 0.7029178 alpha0.4
## 6    0.5 0.7002653 alpha0.5
## 7    0.6 0.7002653 alpha0.6
## 8    0.7 0.6923077 alpha0.7
## 9    0.8 0.6896552 alpha0.8
## 10   0.9 0.6896552 alpha0.9
## 11   1.0 0.7055703   alpha1
```

Therefore, with 1% of the original dataset and 95 predictors, our modelling was somewhat high-dimensional. With an elastic net model (alpha of 0.1), we managed to achieve an accuracy of 73.4%, which is slightly higher than the baseline model. Given that we used 1% of the dataset, this is an impressive result, indicating some of the variables we added were useful in the model, although with 95 predictors, it is likely that they were many variables that added small/no value to model accuracy which is represented by the relatively high accuracy we see in the Lasso model. (i.e. some predictors were likely to have diminished to 0)

The U shape we see in accuracy across various values of alphas suggest the bias variance trade off we see in the different type of penalised regression models. Where accuracy is quite high with a Ridge regression, and then declines to the minimum point at 0.8 but then accuracy increases from there again to bounce back to around 70% with Lasso regression.