Initial Results - Predicting the Popularity of Open Datasets

Project Overview

Open.Canada.ca, the Government of Canada's Open Data Portal contains over 85,000 open datasets or open information resources. These datasets are published by many different government organizations and covers subject matter on a variety of topic areas. On Open.Canada.ca certain datasets receive several thousand downloads per month, while others receive little to no usage on a monthly basis.

Having the ability to predict the popularity of a dataset at the time of publication would enable open data publishers to surface the most relevant and in demand content to users on the open data portal, as well as determine which newly released datasets to promote via other channels such as social media.

In the last 12 months an average of 365 new open datasets were released each month on Open.Canada.ca . The velocity of data release means that it would require significant effort from a person to monitor the release of all these new datasets and use their intuition or some other heuristic to determine what newly released datasets to promote or recommend to users. As such, this problem is well suited to be augmented with a predictive model that can identify newly published datasets at the time of publication which are likely to be popular.

Datasets

This project relies on two source datasets. The first dataset is the metadata catalogue from Open.Canada.ca. The Government of Canada publishes an open dataset of the metadata for all the data and information resources available on Open.Canada.ca. This dataset is updated every night with the latest data.

The second dataset to be used is a listing of the number of downloads from Open.Canada.ca for the last 12 months, by dataset. This dataset is published as an .xls workbook and is updated on the 1st business day of each month with data from the previous 12 months.

The latest data from Open.Canada.ca is available at https://open.canada.ca/data/dataset/2916fad5-ebcc-4c86-b0f3-4f619b29f412/resource/4ebc050f-6c3c-4dfd-817e-875b2caf3ec6/download/downloads-012020-012021.xls for downloads, and at https://open.canada.ca/static/od-do-canada.jsonl.gz for the metadata. For the purposes of the research paper, files as they existed on 2 Feb 2021 will be used.

Github Link

The code and data developed for this project is available at https://github.com/PatLittle/Ryerson-Big-Data-Analytics-Final-Project

Results

Data Load and Preperation

Here we are taking the downloads dataset, and removing two unneeded worksheets, then maping the data from the remaining 86 tabs in the workbook into a coherent dataframe.

```r
library(readxl)
library(openxlsx)
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.0     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
wb<-loadWorkbook("downloads-012020-012021.xlsx")
removeWorksheet(wb, 1)#remove the unwanted tabs
removeWorksheet(wb, 1)#do it again for the 2nd unwanted
saveWorkbook(wb,"downloads.xlsx", overwrite = T)

path<-"downloads.xlsx"
dls<-lapply(excel_sheets(path), read_excel, path = path)

dl_df<-map_dfr(dls,`[`, c("ID / Identificateur","Title English / Titre en anglais","Number of downloads
dl_df<-na.omit(dl_df)
dl_df$`Title English / Titre en anglais`<-NULL
names(dl_df)<-c("ID","downloads")
```

Next we Gunzip the metadata catalogue dataset and then read in the JSON object.

```r
R.utils::gunzip("od-do-canada.jsonl.gz", remove=F)
query1<-readLines("od-do-canada.jsonl")
lines <- lapply(query1,unlist)
```

We then parse the relevant data from the JSON object into a dataframe.

```r
library(jsonlite)
```

```
##
## Attaching package: 'jsonlite'

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

```r
q1<-fromJSON(lines[[1]])
ID<-q1$id
org<-q1$organization$name
desc<-as.numeric(length(unlist(strsplit(q1$notes," "))))
collection<-q1$collection
```

```
freq<-q1$frequency
jurisdiction<-q1$jurisdiction
key1<-q1$keywords$en[1]
key2<-q1$keywords$en[2]
key3<-q1$keywords$en[3]
num_keys<-as.numeric(length(q1$keywords$en))
num_res<-as.numeric(q1$num_resources)
subj1<-q1$subject[1]
subj2<-q1$subject[2]
subj3<-q1$subject[3]
subj4<-q1$subject[4]
date_created<-q1$metadata_created
date_last_mod<-q1$metadata_modified
q1data<-data.frame(ID,org,desc,collection,freq,jurisdiction,key1,key2,key3,num_keys,num_res,subj1,subj2
names(q1data)<-c("ID","org","desc","collection","freq","jurisdiction","key1","key2","key3","num_keys","

for(i in 2:length(lines)){ #loop over this for each line of json - except the 1st line
q1<-fromJSON(lines[[i]])
ID<-q1$id
org<-q1$organization$name
desc<-as.numeric(length(unlist(strsplit(q1$notes," "))))
collection<-q1$collection
freq<-q1$frequency
jurisdiction<-q1$jurisdiction
key1<-q1$keywords$en[1]
key2<-q1$keywords$en[2]
key3<-q1$keywords$en[3]
num_keys<-as.numeric(length(q1$keywords$en))
num_res<-as.numeric(q1$num_resources)
subj1<-q1$subject[1]
subj2<-q1$subject[2]
subj3<-q1$subject[3]
subj4<-q1$subject[4]
date_created<-q1$metadata_created
date_last_mod<-q1$metadata_modified
q1data<- q1data %>% add_row(ID,org,desc,collection,freq,jurisdiction,key1,key2,key3,num_keys,num_res,su
}
```

We can then join the downloads data on to the metadata by using the 'ID' of each dataset. We will also covert date values to the correct format for R.

```
library(gtools)
combined<-merge(x = q1data, y = dl_df, by = "ID", all.x = TRUE)
combined<-na.replace(combined,0)


combined$date_created<-as.Date(combined$date_created)
combined$date_last_mod<-as.Date(combined$date_last_mod)
```

Exploratory Data Analysis

Some of the datasets contained in our data did not exist for the complete 12 months during the analytics collection. We can look at how the age of datasets affects the number of downloads, for datasets that only existed for part of the period.

Since our downloads data ranges from 2020-02-01 - 2021-01-31, we can add a column in the dataset that gives the data created and date last modified in relation to 2021-01-31.

```
combined_factor<-combined
```

```
library(ggpubr)
library(plyr)
```

```
## ------------------------------------------------------------------------------
```

```
## You have loaded plyr after dplyr - this is likely to cause problems.
## If you need functions from both plyr and dplyr, please load plyr first, then dplyr:
## library(plyr); library(dplyr)
```

```
## ------------------------------------------------------------------------------
```

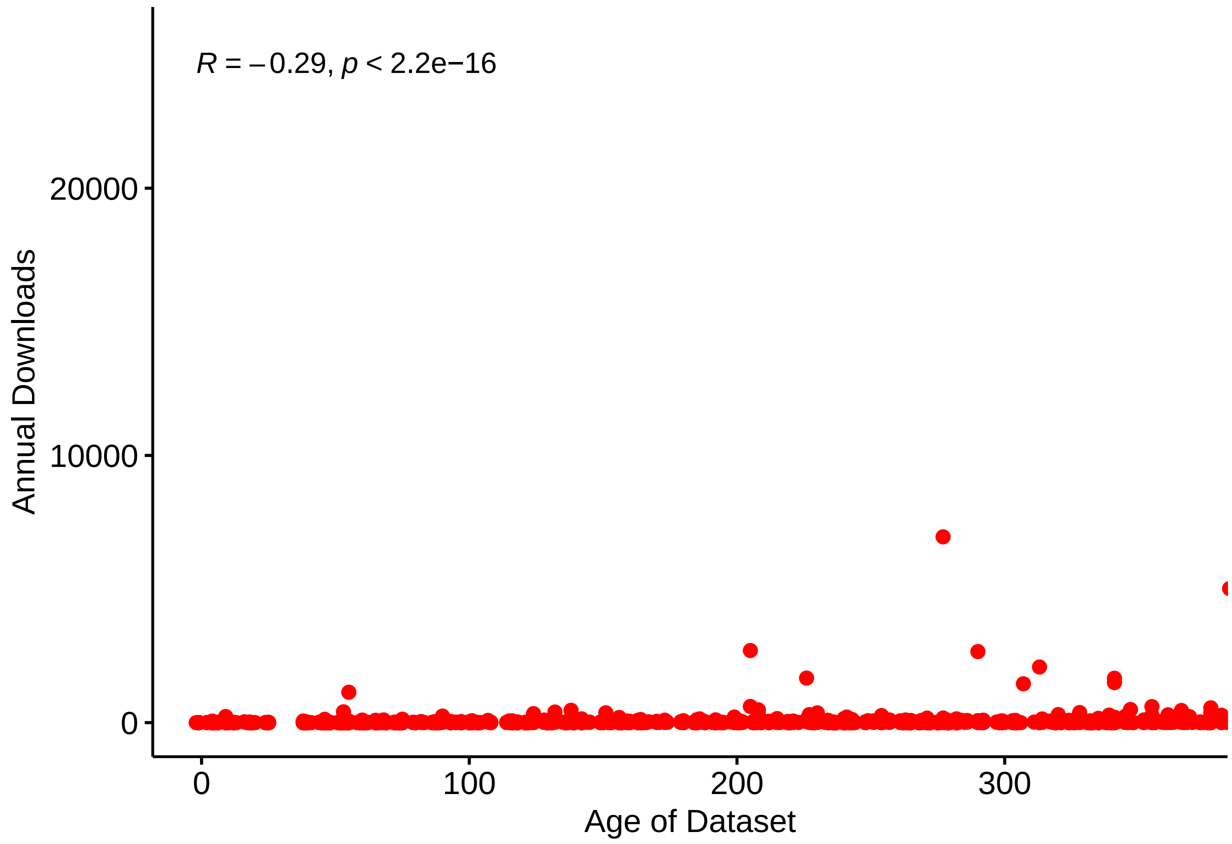
```
##
## Attaching package: 'plyr'
```

```
## The following object is masked from 'package:ggpubr':
##
##     mutate
```

```
## The following objects are masked from 'package:dplyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize
```

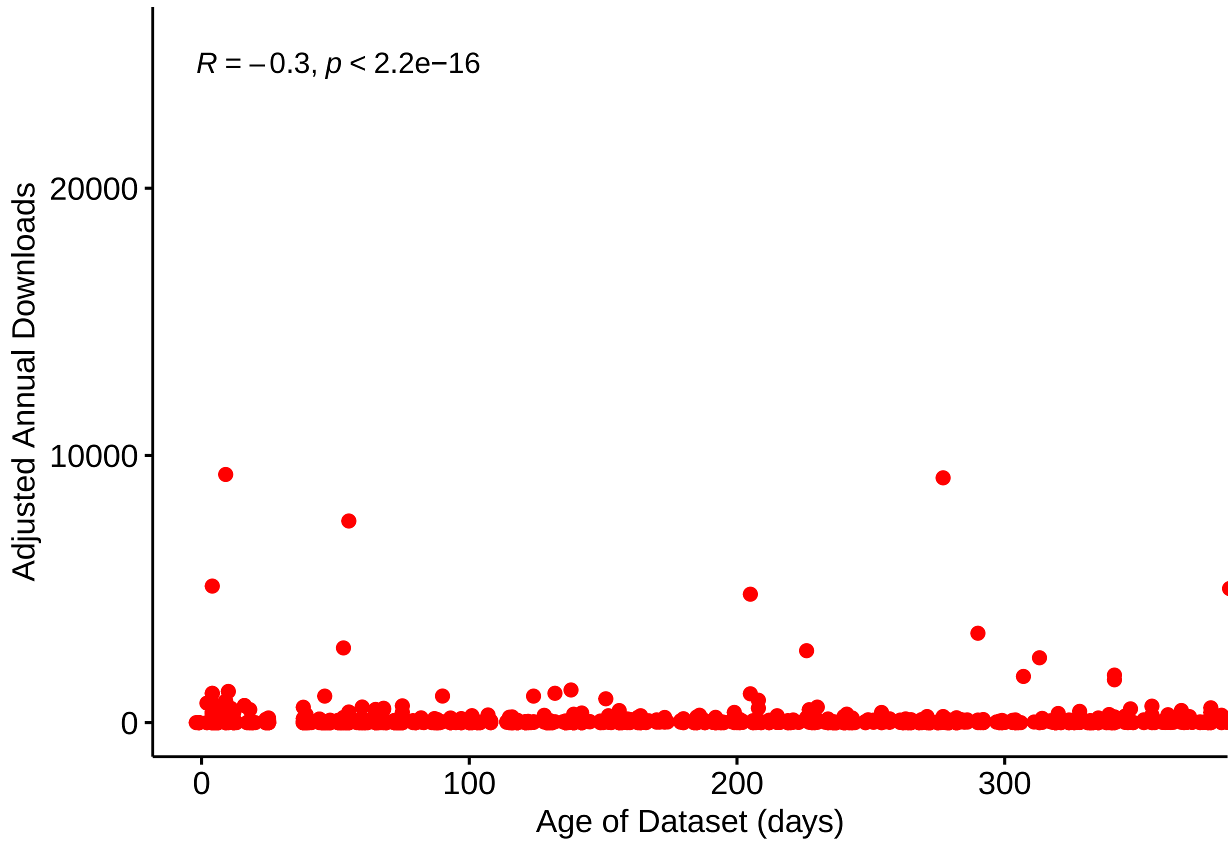
```
## The following object is masked from 'package:purrr':
##
##     compact
```

```
library(dplyr)

date_of_downloads<-as.Date.character("2021-01-31","%Y-%m-%d")
combined_factor$created_days<-as.numeric(difftime(date_of_downloads,combined_factor$date_created, units=
combined_factor$modified_days<-as.numeric(difftime(date_of_downloads,combined_factor$date_last_mod, uni

p_non_adj<-ggscatter(combined_factor, x = "created_days", y = "downloads",
          color = "red", cor.coef = TRUE,
          cor.method = "spearman",
          xlab = "Age of Dataset", ylab = "Annual Downloads")
ggpar(p_non_adj, xlim =c (0,365))
```



Based on our proportion of the year a dataset existed for, we can scale our number of downloads to an annualized amount based their performance during their lifespan.

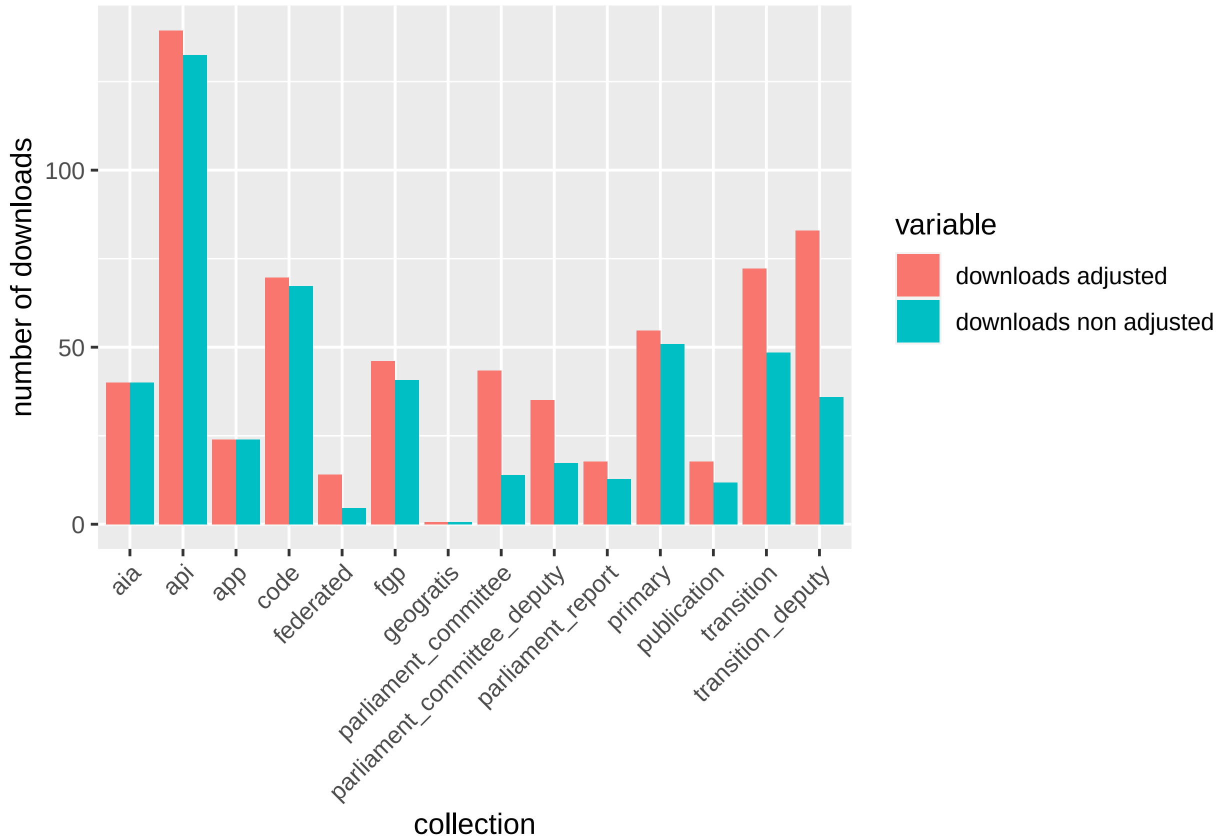
```
j<-1
for(i in 1:length(combined_factor$downloads)){
  if (combined_factor$created_days[i]< 365){
  combined_factor$adj_downloads[j]<-round(365*(combined_factor$downloads[j]/combined_factor$created_day
  } else combined_factor$adj_downloads[j]<-combined_factor$downloads[j]
  j<-j+1
}

p_adj<-ggscatter(combined_factor, x = "created_days", y = "adj_downloads",
          color = "red", cor.coef = TRUE,
          cor.method = "spearman",
          xlab = "Age of Dataset (days)", ylab = "Adjusted Annual Downloads")
ggpar(p_adj, xlim =c (0,365))
```



We can then explore how individual factors might affect their popularity by looking at the mean downloads for each factor within a variable.

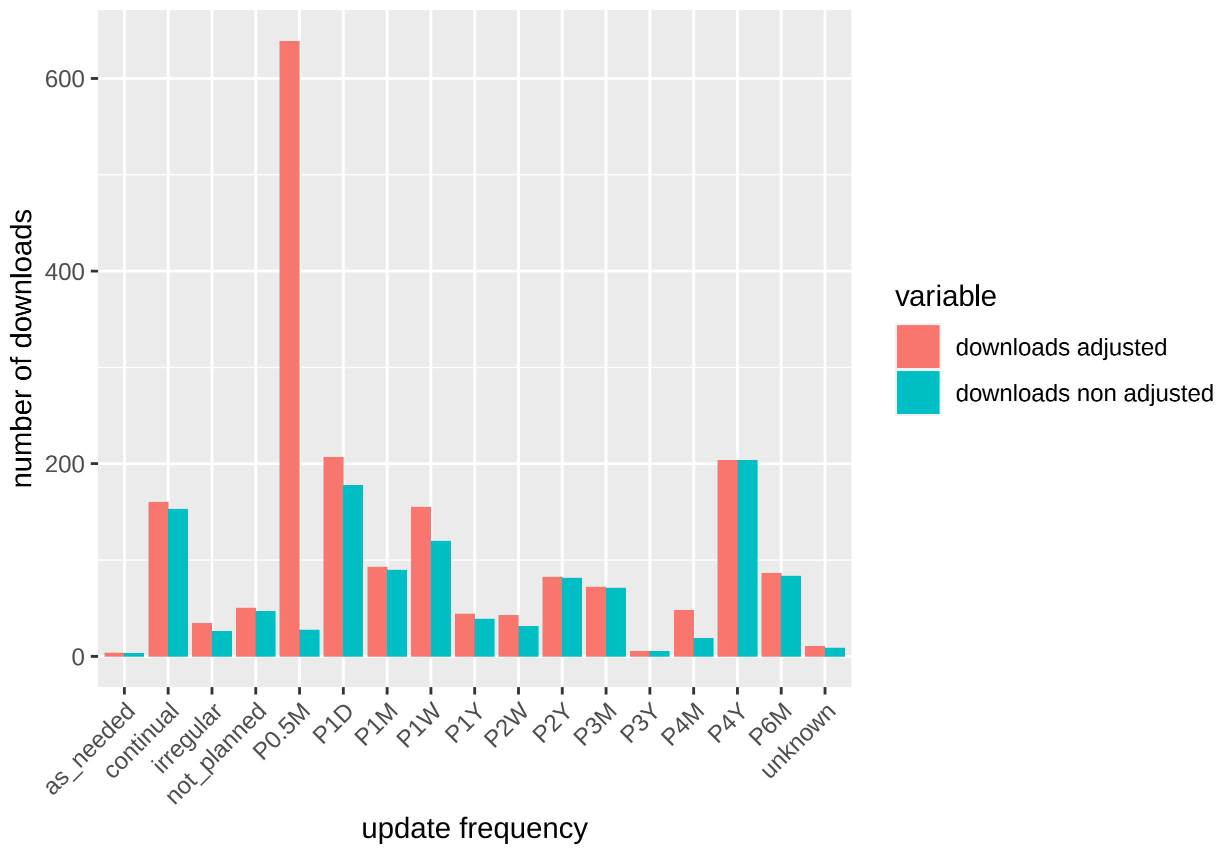
First we can look at the mean downloads by collection type. We can also see how adjusting the downloads for datasets that existed for less than 12 months affects the mean downloads.

```
collection_mean_non_adj<-ddply(combined_factor, .(combined_factor$collection), summarize, mean_download
collection_mean_adj<-ddply(combined_factor, .(combined_factor$collection), summarize, mean_downloads=mea
collection_mean<-cbind(collection_mean_adj,collection_mean_non_adj$mean_downloads)
names(collection_mean)[names(collection_mean)=="combined_factor$collection"]<-"collection"
names(collection_mean)[names(collection_mean)=="collection_mean_non_adj$mean_downloads"]<-"downloads no
names(collection_mean)[names(collection_mean)=="mean_downloads"]<-"downloads adjusted"
collection_mean.long <- gather(collection_mean, variable,value, -collection)
names(collection_mean.long)[names(collection_mean.long)=="value"]<-"number of downloads"

collection_dl<-ggplot(data=collection_mean.long, aes(x=collection, y=`number of downloads`, fill=variab
  geom_bar(stat="identity", position=position_dodge())
collection_dl+theme(axis.text.x = element_text(angle=45, hjust=1))
```



Next we can look at the frequency of update in the same method as above.

```
freq_mean_non_adj<-ddply(combined_factor, .(combined_factor$freq), summarize, mean_downloads=mean(downlo
freq_mean_adj<-ddply(combined_factor, .(combined_factor$freq), summarize, mean_downloads=mean(adj_downlo
freq_mean<-cbind(freq_mean_adj,freq_mean_non_adj$mean_downloads)
names(freq_mean)[names(freq_mean)=="combined_factor$freq"]<-"update frequency"
names(freq_mean)[names(freq_mean)=="freq_mean_non_adj$mean_downloads"]<-"downloads non adjusted"
names(freq_mean)[names(freq_mean)=="mean_downloads"]<-"downloads adjusted"
freq_mean.long <- gather(freq_mean, variable,value, -`update frequency`)
names(freq_mean.long)[names(freq_mean.long)=="value"]<-"number of downloads"

collection_dl<-ggplot(data=freq_mean.long, aes(x=`update frequency`, y=`number of downloads`, fill=varia
  geom_bar(stat="identity", position=position_dodge())
collection_dl+theme(axis.text.x = element_text(angle=45, hjust=1))
```



Next we can look at a percentile chart and determine how many adjusted downloads a dataset needs to get to put it in X percentile.

```
summary(combined_factor$adj_downloads)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##     0.00     0.00     0.00    10.03     0.00 25509.00
```

```
quant_list<-as.list(quantile(combined_factor$adj_downloads, probs = seq(0, 1, by= 0.01)))
as.tibble(quant_list[82:91])
```

```
## Warning: `as.tibble()` was deprecated in tibble 2.0.0.
## Please use `as_tibble()` instead.
## The signature and semantics have changed, see `?as_tibble`.

## # A tibble: 1 x 10
##   `81%` `82%` `83%` `84%` `85%` `86%` `87%` `88%` `89%` `90%`
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     0     0     0     0     0     0     0     4     4     4
```

```
as.tibble(quant_list[92:101])
```

```
## # A tibble: 1 x 10
##   `91%` `92%` `93%` `94%` `95%` `96%` `97%` `98%` `99%` `100%`
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>
## 1     5     8    11    12    18    24    36    56   120  25509
```

For our classification problem we are going consider that datasets in the 95th percentile of adjusted downloads are popular enough to satisfy our business requirement of being a dataset worth promoting as newly released datasets that data consumers might be interested in, and worth promoting.

Therefore we will add a column to the dataframe that will contain our binary encoded popularity. We will encode datasets below the 95th percentile for adjusted downloads as a 0 and datasets in the 95th percentile or above as a 1.

```
j<-1
for(i in 1:length(combined_factor$adj_downloads)){
  if (combined_factor$adj_downloads[j]< quant_list[96]){
    combined_factor$bin_downloads[j]<-0
  } else {
    combined_factor$bin_downloads[j]<-1
  }
  j<-j+1
}

combined_factor$bin_downloads<-as.factor(combined_factor$bin_downloads)
```

Model Building and Training

In this section we will begin by getting rid of anything we added to our dataframe for EDA, which will not be used to train our model. We will then make sure it is in a data.table format for tidymodels.

```
library(data.table)
```

```
##
## Attaching package: 'data.table'

## The following objects are masked from 'package:dplyr':
##
##     between, first, last

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

```
library(tidymodels)
```

```
## -- Attaching packages -------------------------------------- tidymodels 0.1.2 --

## v broom     0.7.5      v recipes   0.1.15
## v dials     0.0.9      v rsample   0.0.9
## v infer     0.5.4      v tune      0.1.3
## v modeldata 0.1.0      v workflows 0.2.2
## v parsnip   0.1.5      v yardstick 0.0.7

## -- Conflicts ----------------------------------------- tidymodels_conflicts() --
## x plyr::arrange()         masks dplyr::arrange()
## x data.table::between()   masks dplyr::between()
```

```
## x plyr::compact()         masks purrr::compact()
## x plyr::count()           masks dplyr::count()
## x scales::discard()       masks purrr::discard()
## x plyr::failwith()        masks dplyr::failwith()
## x dplyr::filter()         masks stats::filter()
## x data.table::first()     masks dplyr::first()
## x recipes::fixed()        masks stringr::fixed()
## x jsonlite::flatten()     masks purrr::flatten()
## x plyr::id()              masks dplyr::id()
## x dplyr::lag()            masks stats::lag()
## x data.table::last()      masks dplyr::last()
## x plyr::mutate()          masks ggpubr::mutate(), dplyr::mutate()
## x rsample::permutations() masks gtools::permutations()
## x plyr::rename()          masks dplyr::rename()
## x yardstick::spec()       masks readr::spec()
## x recipes::step()         masks stats::step()
## x plyr::summarise()       masks dplyr::summarise()
## x plyr::summarize()       masks dplyr::summarize()
## x data.table::transpose() masks purrr::transpose()
```

```
df<-combined_factor

df$ID<-NULL
df$date_created<-NULL
df$date_last_mod<-NULL
df$downloads<-NULL
df$created_days<-NULL
df$modified_days<-NULL
df$adj_downloads<-NULL

df<-as.data.table(df,keep.rownames = F)
```

We are then doing to convert all of our factor variables, except our target variable into numeric variables.

```
df$org<-as.numeric(as.factor(df$org))
df$collection<-as.numeric(as.factor(df$collection))
df$freq<-as.numeric(as.factor(df$freq))
df$jurisdiction<-as.numeric(as.factor(df$jurisdiction))
df$key1<-as.numeric(as.factor(df$key1))
df$key2<-as.numeric(as.factor(df$key2))
df$key3<-as.numeric(as.factor(df$key3))
df$subj1<-as.numeric(as.factor(df$subj1))
df$subj2<-as.numeric(as.factor(df$subj2))
df$subj3<-as.numeric(as.factor(df$subj3))
df$subj4<-as.numeric(as.factor(df$subj4))
```

Next we will split our data into the training and testing split. We will set a seed to get reproducible splits. We will also use our binary encoded popularity as our stratification variable. Stratification will ensure we get samples that have a good mix of popular and non-popular datasets in each sample. If our data was more normally distributed we might not need to worry about this, but since we are using the 95th percentile as the threshold for popularity, within in a already skewed dataset, stratification is important.

```r
set.seed(888)

pop_split<- initial_split(df,strata = bin_downloads)
pop_train<-training(pop_split)
pop_test<-testing(pop_split)
```

Next we will setup our model specification. We are using the XGBoost model, in the classification mode. We will run 100 trees, which is the default hyperparameter for this model. During the model tuning phase we will tune our hyperparameters: tree depth, min n, loss reduction, sample size, mtry, and learn rate.

```r
xgb_spec <- boost_tree(
  trees = 100,
  tree_depth = tune(), min_n = tune(),
  loss_reduction = tune(),                     ## first three: model complexity
  sample_size = tune(), mtry = tune(),         ## randomness
  learn_rate = tune()                         ## step size
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

xgb_spec
```

```
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = 100
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Computational engine: xgboost
```

In order to tune our hyperparameters we need to give the model a set of values to try from. We will use a latin hypercube as our search strategy. As a form of local search optimization, latin hypercube should be more performant than other options such as grid seach or random search.

```r
xgb_grid <- grid_latin_hypercube(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  finalize(mtry(), pop_train),
  learn_rate(),
  size = 20
)

xgb_grid
```

```
## # A tibble: 20 x 6
```

```
##    tree_depth min_n loss_reduction sample_size  mtry learn_rate
##         <int> <int>          <dbl>       <dbl> <int>      <dbl>
##  1         10    15       1.75e- 4       0.283    11   7.88e- 8
##  2          4    13       7.04e+ 0       0.189     2   6.98e- 9
##  3          4    10       1.18e- 8       0.815     7   1.71e- 4
##  4          9    23       1.62e- 5       0.504     2   6.36e- 2
##  5         12     4       8.82e- 2       0.974    14   4.79e- 5
##  6          3    34       4.92e- 8       0.627    13   1.27e- 3
##  7          1    32       1.46e- 3       0.654     7   7.78e- 3
##  8         14    20       1.24e+ 1       0.525     6   3.74e-10
##  9         11     4       2.57e- 7       0.824     2   2.38e- 7
## 10          2    29       1.23e- 5       0.888     9   3.29e- 3
## 11         15    38       4.07e- 2       0.202    10   7.33e- 7
## 12          7    25       4.04e- 1       0.708     5   2.23e-10
## 13         13    35       6.49e-10       0.257    10   1.13e- 9
## 14          7    10       4.56e- 3       0.749    12   5.16e- 4
## 15          5    27       1.87e- 6       0.397     3   1.51e- 2
## 16          8    38       2.52e- 9       0.583    14   2.75e- 9
## 17         13    18       1.84e+ 0       0.128    15   2.57e- 8
## 18          8    15       1.53e-10       0.445     4   1.64e- 5
## 19          6     7       2.48e- 4       0.326     6   1.39e- 6
## 20         10    23       3.67e- 7       0.950     9   3.66e- 6
```

Next we will setup our workflow. We provide our model specification and our formula of predicting bin_downloads by all other variables.

```r
xgb_wf <- workflow() %>%
  add_formula(bin_downloads ~ .) %>%
  add_model(xgb_spec)

xgb_wf
```

```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## bin_downloads ~ .
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = 100
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Computational engine: xgboost
```

Next we will setup cross validation samples in order to tune the model. We will use 5 fold cross validation.

```
pop_folds<-vfold_cv(pop_train,v=5,strata=bin_downloads)
pop_folds
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits               id
##   <list>               <chr>
## 1 <split [52639/13160]> Fold1
## 2 <split [52639/13160]> Fold2
## 3 <split [52639/13160]> Fold3
## 4 <split [52639/13160]> Fold4
## 5 <split [52640/13159]> Fold5
```

Next we will actually train the model based on our data and hyperparameters. Here we need to setup parallel processing to allow the model to train in a reasonable amount of time. In my model training environment, using 4 cores gave the best results. Dedicated more cores to the model seemed to cause sporadic failures.

```
library(doParallel)
```

```
## Loading required package: foreach
```

```
##
## Attaching package: 'foreach'
```

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

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

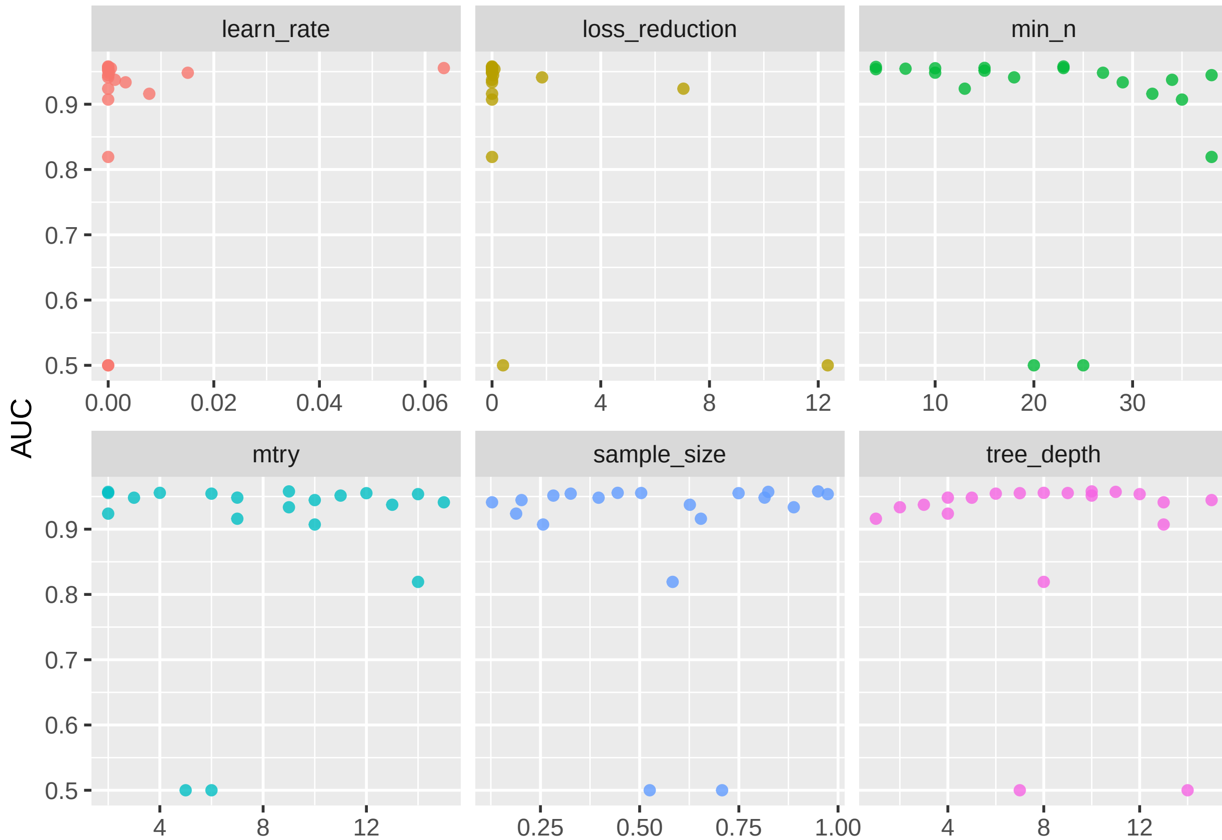
```
cores<-detectCores()
cl<- makeCluster(cores[1]-4)
registerDoParallel(cl)

set.seed(888)
xgb_res <- tune_grid(
  xgb_wf,
  resamples = pop_folds,
  grid = xgb_grid,
  control = control_grid(save_pred = TRUE)
)
```

Next we can illustrate how the different hyperparmeters influenced the performance of the model. We evaluate the model based on the *Area Under the Receiver Operator Curve*. In the evaluation 1 would indicate a perfect prediction, and 0.5 would indicate a prediction as good a 50-50 chance.

```
collect_metrics(xgb_res)
```

```
## # A tibble: 40 x 12
##     mtry min_n tree_depth    learn_rate loss_reduction sample_size .metric
##    <int> <int>      <int>         <dbl>          <dbl>       <dbl> <chr>
##  1    11    15         10 0.0000000788    0.000175           0.283 accuracy
##  2    11    15         10 0.0000000788    0.000175           0.283 roc_auc
##  3     2    13          4 0.00000000698   7.04               0.189 accuracy
##  4     2    13          4 0.00000000698   7.04               0.189 roc_auc
##  5     7    10          4 0.000171        0.0000000118       0.815 accuracy
##  6     7    10          4 0.000171        0.0000000118       0.815 roc_auc
##  7     2    23          9 0.0636          0.0000162          0.504 accuracy
##  8     2    23          9 0.0636          0.0000162          0.504 roc_auc
##  9    14     4         12 0.0000479       0.0882             0.974 accuracy
## 10    14     4         12 0.0000479       0.0882             0.974 roc_auc
## # ... with 30 more rows, and 5 more variables: .estimator <chr>, mean <dbl>,
## #   n <int>, std_err <dbl>, .config <chr>
```

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



Above we can see we are achieving mutliple combinations of hyperparameters returning good results, with some outliers that are low performing.

Here we can examine our best performing combinations of hyperparameters.

```
best_auc <- select_best(xgb_res, "roc_auc")
best_auc
```

```
## # 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     9    23         10 0.00000366    0.000000367       0.950 Preprocessor1_Mo~
```

Using our best performing set of hyperparameters we can finalize our model.

```
final_xgb <- finalize_workflow(
  xgb_wf,
  best_auc
)

final_xgb
```

```
## == Workflow ===================================================================
## Preprocessor: Formula
## Model: boost_tree()
```

```
##
## -- Preprocessor ----------------------------------------------------------------
## bin_downloads ~ .
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = 9
##   trees = 100
##   min_n = 23
##   tree_depth = 10
##   learn_rate = 3.65543876318453e-06
##   loss_reduction = 3.67168216416153e-07
##   sample_size = 0.949866834131535
##
## Computational engine: xgboost
```

With our finalized model we can look at the variable importance of each of our variables in the model.

```
library(vip)
```

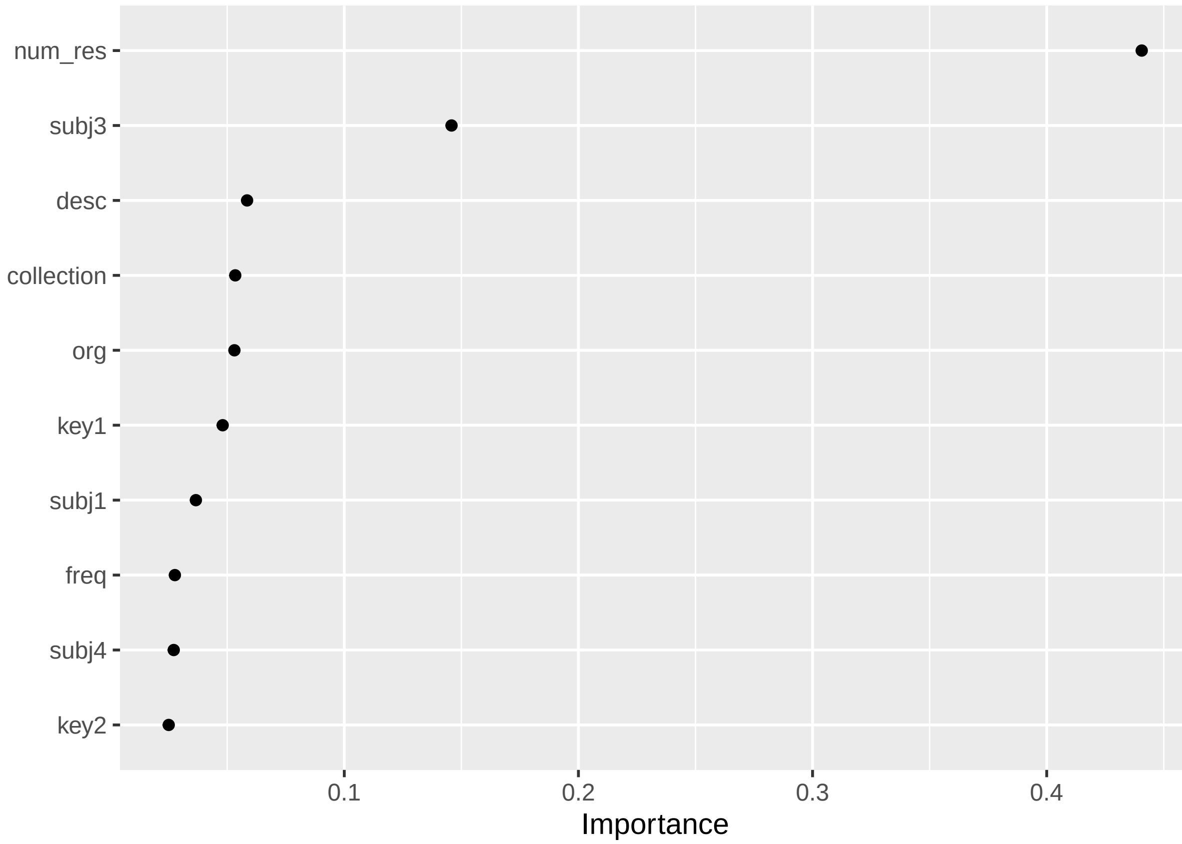
```
##
## Attaching package: 'vip'

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

```
install.packages("vip")
```

```
## Warning: package 'vip' is in use and will not be installed
```

```
final_xgb %>%
  fit(data = pop_train) %>%
  pull_workflow_fit() %>%
  vip(geom = "point")
```

```
## [15:58:55] WARNING: amalgamation/../src/learner.cc:1061: Starting in XGBoost 1.3.0, the default evalu
```



Looking at the importance of each indicator, we can observe that the number of resources (files a user can download) contained within a dataset is our best predictor of popularity, which seems to make intuitive sense. Additionally jurisdiction and frequency seem to have low predictive power within the model. This is somewhat counter intuitive, I would have expected that datasets that a more frequently updated would recieve more downloads, as there would be a reason to periodically re-download the same dataset for the latest data.

lastly we can take our finalized model and fit it to our test data.

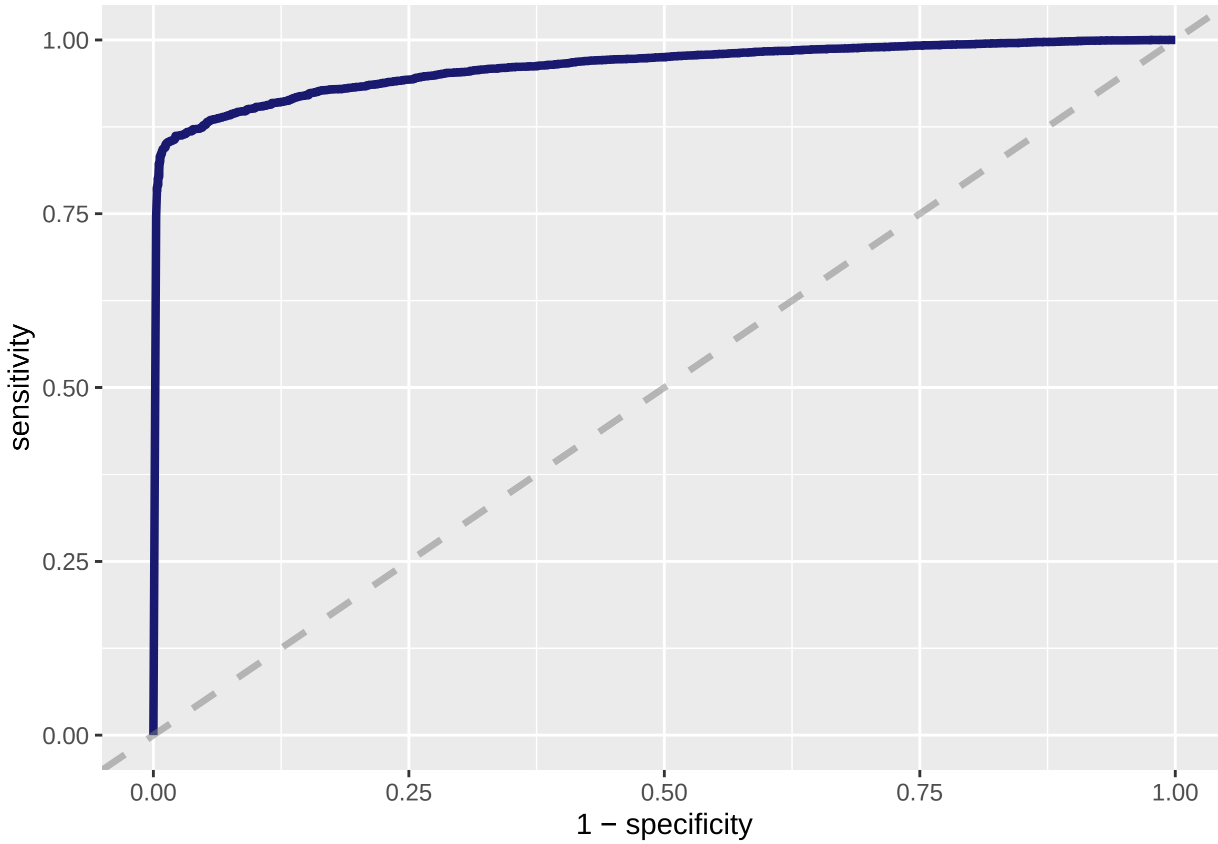
```
final_res <- last_fit(final_xgb, pop_split)
collect_metrics(final_res)
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.953 Preprocessor1_Model1
## 2 roc_auc  binary         0.961 Preprocessor1_Model1
```

In our initial result we are getting a 95.22% accuracy and a 96.11% area under the curve.

```
final_res %>%
  collect_predictions() %>%
  roc_curve(bin_downloads, .pred_0) %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity)) +
  geom_line(size = 1.5, color = "midnightblue") +
  geom_abline(
```

```
    lty = 2, alpha = 0.5,
    color = "gray50",
    size = 1.2
  )
```



Next we can use a confusion matrix to validate our results of our assessment made with the AUC.

```
final_res %>%
  collect_predictions() %>%
  conf_mat(truth = bin_downloads, estimate = .pred_class)
```

```
##            Truth
## Prediction     0     1
##          0 20797  1005
##          1    29   101
```

Here we can see that the model is getting a good score at correctly predicting a true negative, meaning it can normally identify datasets that are not popular, but cannot identify datasets that will be popular.

Corrective Action

Since we have such a large number of datasets with zero downloads, I hypothesize that if we remove a certain collection type of records from the model we may be able to achieve better results. Here we are going to omit records in the geogratis collection, and then precede to re-run the analysis.

```r
df2<-combined_factor

df2$ID<-NULL
df2$date_created<-NULL
df2$date_last_mod<-NULL
df2$downloads<-NULL
df2$created_days<-NULL
df2$modified_days<-NULL
df2$adj_downloads<-NULL

df2<-subset(df2, collection!="geogratis")


df2$org<-as.numeric(as.factor(df2$org))
df2$collection<-as.numeric(as.factor(df2$collection))
df2$freq<-as.numeric(as.factor(df2$freq))
df2$jurisdiction<-as.numeric(as.factor(df2$jurisdiction))
df2$key1<-as.numeric(as.factor(df2$key1))
df2$key2<-as.numeric(as.factor(df2$key2))
df2$key3<-as.numeric(as.factor(df2$key3))
df2$subj1<-as.numeric(as.factor(df2$subj1))
df2$subj2<-as.numeric(as.factor(df2$subj2))
df2$subj3<-as.numeric(as.factor(df2$subj3))
df2$subj4<-as.numeric(as.factor(df2$subj4))
```

Here we will setup a second model to hold our new data subset. Since we have less data in the dataset with the subset, we can try more trees and a larger hyperparameter search space and still have reasonable training times.

```r
set.seed(888)

pop_split2<- initial_split(df2,strata = bin_downloads)
pop_train2<-training(pop_split2)
pop_test2<-testing(pop_split2)


xgb_spec2 <- boost_tree(
  trees = 500,
  tree_depth = tune(), min_n = tune(),
  loss_reduction = tune(),                     ## first three: model complexity
  sample_size = tune(), mtry = tune(),         ## randomness
  learn_rate = tune()                         ## step size
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

xgb_spec2
```

```
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
##   trees = 500
```

```
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Computational engine: xgboost
```

```
xgb_grid2 <- grid_latin_hypercube(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  finalize(mtry(), pop_train2),
  learn_rate(),
  size = 30
)

xgb_grid2
```

```
## # A tibble: 30 x 6
##    tree_depth min_n loss_reduction sample_size  mtry   learn_rate
##         <int> <int>          <dbl>       <dbl> <int>        <dbl>
##  1         11    22 0.0000327            0.571    12 0.0000000114
##  2         15    17 0.00433              0.775     4 0.0000000633
##  3          7     8 0.0000000184         0.852     5 0.0000547
##  4         13     2 0.000000449          0.480     7 0.00000116
##  5         13    21 0.372                0.513     7 0.0127
##  6         13    37 0.00000264           0.267    13 0.0000000335
##  7          6    12 5.02                 0.582    15 0.0000245
##  8          2    17 0.000000000560       0.890    11 0.000423
##  9          3    26 0.0972               0.749    10 0.000000183
## 10          9    13 0.000401             0.205     8 0.0470
## # ... with 20 more rows
```

```
xgb_wf2 <- workflow() %>%
  add_formula(bin_downloads ~ .) %>%
  add_model(xgb_spec2)

xgb_wf2
```

```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## bin_downloads ~ .
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
##
## Main Arguments:
##   mtry = tune()
```

```
##   trees = 500
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Computational engine: xgboost
```

re-setting our fivefold cross validation.

```r
pop_folds2<-vfold_cv(pop_train2,v=5,strata=bin_downloads)
pop_folds2
```

```
## #  5-fold cross-validation using stratification
## # A tibble: 5 x 2
##   splits               id
##   <list>               <chr>
## 1 <split [10989/2748]> Fold1
## 2 <split [10989/2748]> Fold2
## 3 <split [10989/2748]> Fold3
## 4 <split [10990/2747]> Fold4
## 5 <split [10991/2746]> Fold5
```

training the new model

```r
library(doParallel)
cores<-detectCores()
cl<- makeCluster(cores[1]-4)
registerDoParallel(cl)

set.seed(888)
xgb_res2 <- tune_grid(
  xgb_wf2,
  resamples = pop_folds2,
  grid = xgb_grid2,
  control = control_grid(save_pred = TRUE)
)
```

selecting our best result and finalizing our workflow

```r
best_auc2 <- select_best(xgb_res2, "roc_auc")
best_auc2
```

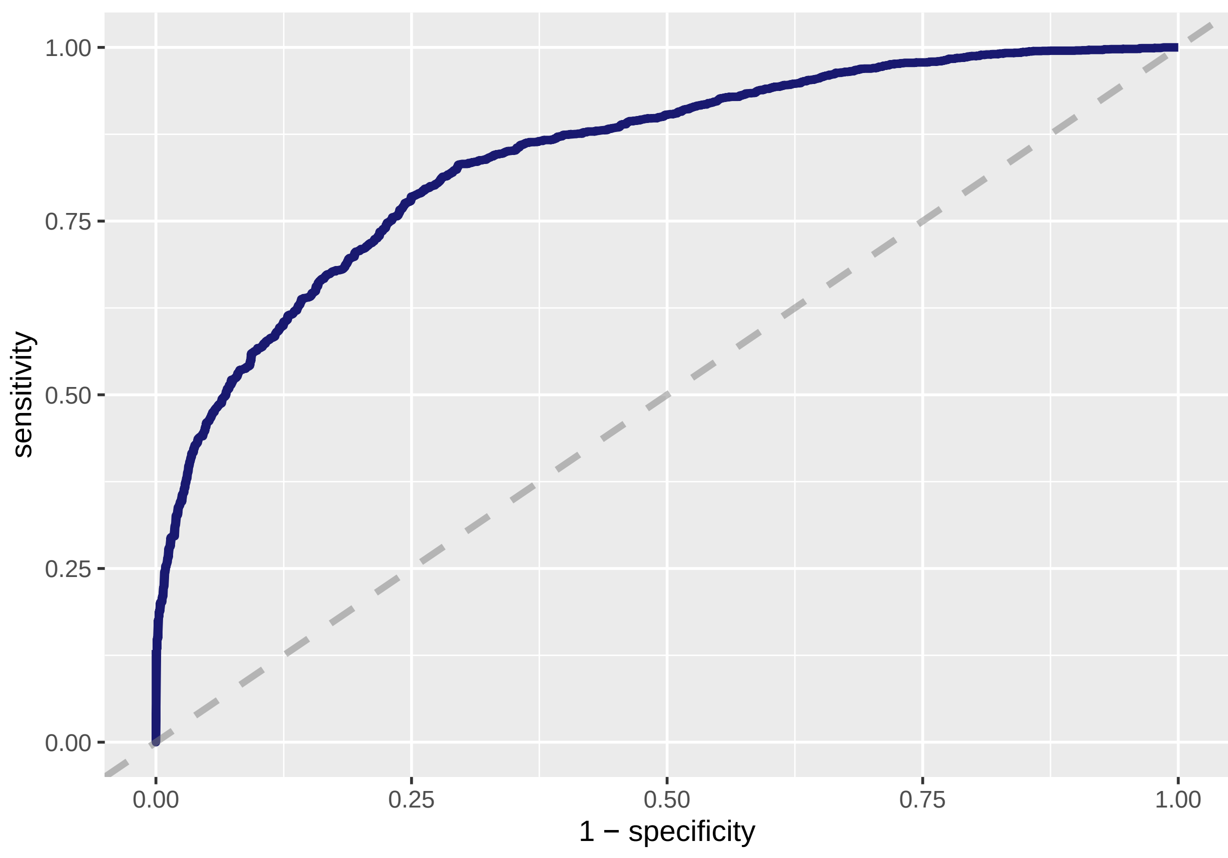
```
## # A tibble: 1 x 7
##    mtry min_n tree_depth learn_rate loss_reduction sample_size .config
##   <int> <int>      <int>      <dbl>          <dbl>       <dbl> <chr>
## 1     7     2         13 0.00000116    0.000000449       0.480 Preprocessor1_Mo~
```

```r
final_xgb2 <- finalize_workflow(
  xgb_wf2,
  best_auc2
)
```

```r
final_res2 <- last_fit(final_xgb2, pop_split2)
collect_metrics(final_res2)
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.820 Preprocessor1_Model1
## 2 roc_auc  binary         0.843 Preprocessor1_Model1
```

```r
final_res2 %>%
  collect_predictions() %>%
  roc_curve(bin_downloads, .pred_0) %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity)) +
  geom_line(size = 1.5, color = "midnightblue") +
  geom_abline(
    lty = 2, alpha = 0.5,
    color = "gray50",
    size = 1.2
  )
```




```r
final_res2 %>%
  collect_predictions() %>%
  conf_mat(truth = bin_downloads, estimate = .pred_class)
```

```
##           Truth
```

```
## Prediction    0    1
##          0 3549  774
##          1   52  204
```

In this model the ability to predict true positives is greatly improved from our first iteration. At this stage the type 1 errors still out number the correct predictions, therefore the model is not suitable for implementation at this stage.

Since we are using the 95% percentile as the definition of "popular" there may be too few examples to saturate the model with enough relevant training data. If we expand our definition of "popular" to the 90th percentile we would double the amount of records considered popular, which may provide the model with enough examples to make a better binary classification.

```
df3<-combined_factor
df3<-subset(df3, collection!="geogratis")

summary(df3$adj_downloads)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0     0.0     0.0    45.4    12.0 25509.0
```

```
quant_list<-as.list(quantile(df3$adj_downloads, probs = seq(0, 1, by= 0.01)))
as.tibble(quant_list[72:81])
```

```
## # A tibble: 1 x 10
##   `71%` `72%` `73%` `74%` `75%` `76%` `77%` `78%` `79%` `80%`
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    12    12    12    12    12    16    16    16    19    20
```

```
as.tibble(quant_list[82:91])
```

```
## # A tibble: 1 x 10
##   `81%` `82%` `83%` `84%` `85%` `86%` `87%` `88%` `89%` `90%`
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    20    24    24    28    28    32    36    39    43    48
```

```
as.tibble(quant_list[92:101])
```

```
## # A tibble: 1 x 10
##   `91%` `92%` `93%` `94%` `95%` `96%` `97%` `98%` `99%` `100%`
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <dbl>
## 1    52    60  69.0  83.1   104   135   185  314.  678.  25509
```

Here we can remap the binary classification of popular to the 75th percentile

```
j<-1
for(i in 1:length(df3$adj_downloads)){
  if (df3$adj_downloads[j]< quant_list[76]){
    df3$bin_downloads[j]<-0
  } else {
    df3$bin_downloads[j]<-1
```

```
  }
  j<-j+1
}

df3$bin_downloads<-as.factor(df3$bin_downloads)
```

Here we will complete our required data preparation

```
df3$ID<-NULL
df3$date_created<-NULL
df3$date_last_mod<-NULL
df3$downloads<-NULL
df3$created_days<-NULL
df3$modified_days<-NULL
df3$adj_downloads<-NULL


df3$org<-as.numeric(as.factor(df3$org))
df3$collection<-as.numeric(as.factor(df3$collection))
df3$freq<-as.numeric(as.factor(df3$freq))
df3$jurisdiction<-as.numeric(as.factor(df3$jurisdiction))
df3$key1<-as.numeric(as.factor(df3$key1))
df3$key2<-as.numeric(as.factor(df3$key2))
df3$key3<-as.numeric(as.factor(df3$key3))
df3$subj1<-as.numeric(as.factor(df3$subj1))
df3$subj2<-as.numeric(as.factor(df3$subj2))
df3$subj3<-as.numeric(as.factor(df3$subj3))
df3$subj4<-as.numeric(as.factor(df3$subj4))
```

Here we will setup the same model again based on the 75th percentile data

```
set.seed(888)

pop_split3<- initial_split(df3,strata = bin_downloads)
pop_train3<-training(pop_split3)
pop_test3<-testing(pop_split3)


xgb_spec3 <- boost_tree(
  trees = 1000,
  tree_depth = tune(), min_n = tune(),
  loss_reduction = tune(),                     ## first three: model complexity
  sample_size = tune(), mtry = tune(),         ## randomness
  learn_rate = tune()                         ## step size
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

xgb_spec3
```

```
## Boosted Tree Model Specification (classification)
##
```

```
## Main Arguments:
##   mtry = tune()
##   trees = 1000
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Computational engine: xgboost
```

```
xgb_grid3 <- grid_latin_hypercube(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),
  finalize(mtry(), pop_train3),
  learn_rate(),
  size = 45
)

xgb_grid3
```

```
## # A tibble: 45 x 6
##    tree_depth min_n loss_reduction sample_size  mtry    learn_rate
##         <int> <int>          <dbl>       <dbl> <int>         <dbl>
##  1         11     9 0.000000000858       0.819     5 0.000333
##  2         12    11 0.000000000424       0.158    15 0.00000000213
##  3          8    19 0.00000000341        0.128    14 0.00464
##  4          9     7 0.212                0.944     9 0.0000142
##  5          9    21 0.0000000148         0.750     5 0.000000442
##  6          5    39 0.00000000365        0.277     9 0.00000640
##  7          9     2 0.0717               0.397     2 0.0000000187
##  8          2     8 0.0345               0.306     4 0.000726
##  9          3    39 0.00426              0.879     4 0.00000127
## 10          6    24 0.000148             0.708     6 0.0000238
## # ... with 35 more rows
```

```
xgb_wf3 <- workflow() %>%
  add_formula(bin_downloads ~ .) %>%
  add_model(xgb_spec3)

xgb_wf3
```

```
## == Workflow ====================================================================
## Preprocessor: Formula
## Model: boost_tree()
##
## -- Preprocessor ----------------------------------------------------------------
## bin_downloads ~ .
##
## -- Model -----------------------------------------------------------------------
## Boosted Tree Model Specification (classification)
```

```
##
## Main Arguments:
##   mtry = tune()
##   trees = 1000
##   min_n = tune()
##   tree_depth = tune()
##   learn_rate = tune()
##   loss_reduction = tune()
##   sample_size = tune()
##
## Computational engine: xgboost
```

Next we will setup the cross validation and train the new model. We will additionally increase our validation to 10 fold.

```
pop_folds3<-vfold_cv(pop_train3,v=10,strata=bin_downloads)
pop_folds3
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 10 x 2
##    splits               id
##    <list>               <chr>
##  1 <split [12362/1375]> Fold01
##  2 <split [12362/1375]> Fold02
##  3 <split [12362/1375]> Fold03
##  4 <split [12363/1374]> Fold04
##  5 <split [12364/1373]> Fold05
##  6 <split [12364/1373]> Fold06
##  7 <split [12364/1373]> Fold07
##  8 <split [12364/1373]> Fold08
##  9 <split [12364/1373]> Fold09
## 10 <split [12364/1373]> Fold10
```

```
library(doParallel)
cores<-detectCores()
cl<- makeCluster(cores[1]-4)
registerDoParallel(cl)

set.seed(888)
xgb_res3 <- tune_grid(
  xgb_wf3,
  resamples = pop_folds3,
  grid = xgb_grid3,
  control = control_grid(save_pred = TRUE)
)
```

```
best_auc3 <- select_best(xgb_res3, "roc_auc")
best_auc3
```

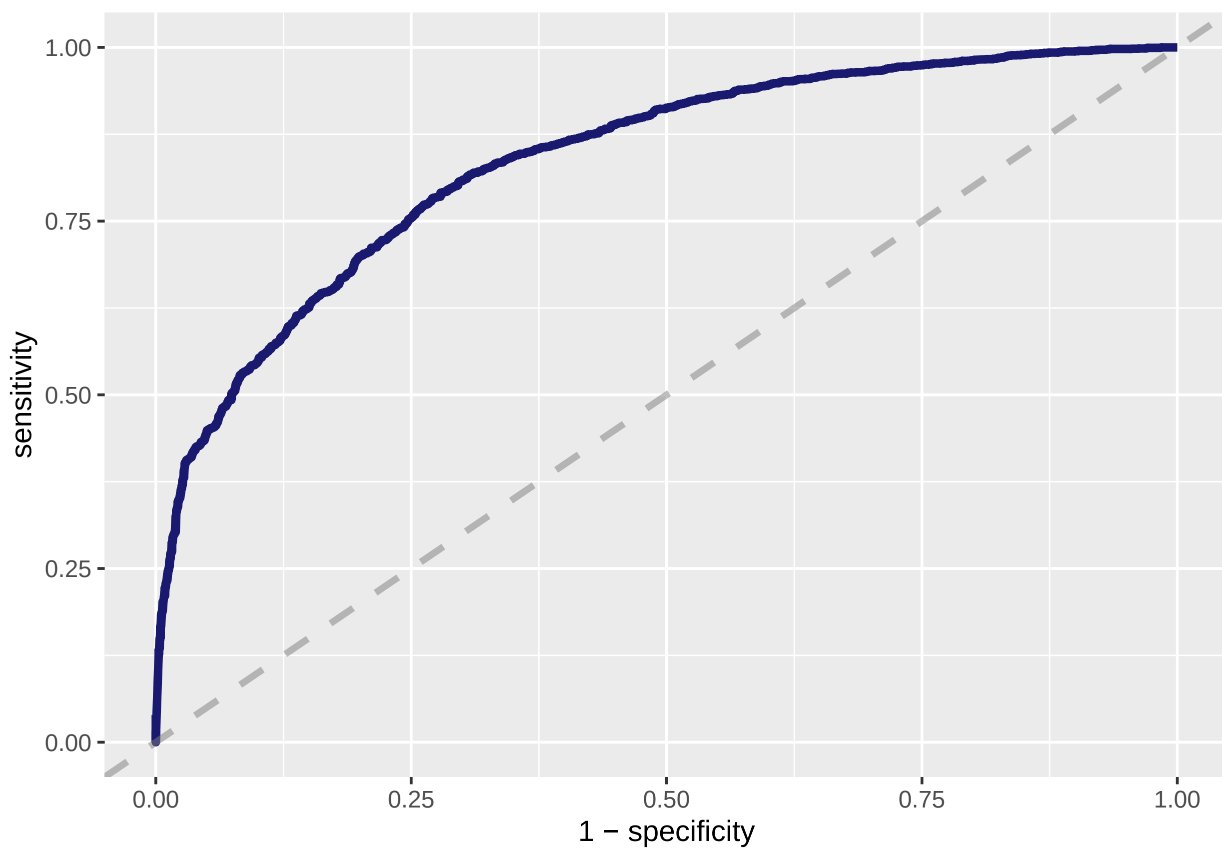
```
## # A tibble: 1 x 7
##    mtry min_n tree_depth learn_rate loss_reduction sample_size .config
##   <int> <int>      <int>      <dbl>          <dbl>       <dbl> <chr>
## 1     7    27         13     0.0321  0.00000000135       0.828 Preprocessor1_Mo~
```

```r
final_xgb3 <- finalize_workflow(
  xgb_wf3,
  best_auc3
)
```

```r
final_res3 <- last_fit(final_xgb3, pop_split3)
collect_metrics(final_res3)
```

```
## # A tibble: 2 x 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy binary         0.790 Preprocessor1_Model1
## 2 roc_auc  binary         0.837 Preprocessor1_Model1
```

```r
final_res3 %>%
  collect_predictions() %>%
  roc_curve(bin_downloads, .pred_0) %>%
  ggplot(aes(x = 1 - specificity, y = sensitivity)) +
  geom_line(size = 1.5, color = "midnightblue") +
  geom_abline(
    lty = 2, alpha = 0.5,
    color = "gray50",
    size = 1.2
  )
```



```r
final_res3 %>%
  collect_predictions() %>%
  conf_mat(truth = bin_downloads, estimate = .pred_class)
```

```
##           Truth
## Prediction    0    1
##          0 2893  615
##          1  348  723
```

Based on the results here we have significantly reduced the proportion of type 1 error, however we are still seeing slightly more type 1 errors over correct predictions.

With further refinement we should be able to select a "popularity percentile" that allows us to build a usable model. In addition if we were to deploy a larger model with more trees and a larger hyperparameter search space, we may be able to drive further performance gains without decreasing the selectivity of what we consider to be a popular download.