Stellar Classification

Adelaide Zhang (amz316)
CS-GY 6923 Machine Learning
Prof. Raman Kannan

March 03, 2022

Abstract

The Sloan Digital Sky Survey (SDSS) [1] is a longstanding project that uses a dedicated optical telescope to take high-resolution pictures of the night sky, with the goal of generating a map of one quarter of the entire sky. These images are processed and encoded into digital data measuring the position, shape, brightness, and color of the hundreds of millions of astronomical objects detected. Astronomers can use this data to catalogue the observed objects into fundamental categories; namely distinguishing whether they are stars, galaxies, or quasars.

Contents

1 Introduction

For this project, I will be using the Stellar Classification Dataset sourced from Kaggle [2]. The dataset compiles a variety of measured attributes collected by the Sloan Digital Sky Survey into a tabular format, with each instance corresponding to an astronomical object. The goal is to use the observed qualities of the objects to classify each into one of three possible types: star, quasar, or galaxy.

1.1 Feature Overview

The dataset consists of 100,000 observations with following columns, consisting of 17 independent variables and 1 categorical dependent variable ("class"). Table 1 lists the feature descriptions provided on Kaggle.

1.2 Loading into R

The Kaggle dataset is provided as a .csv file. To load it into R, we can use the read command.

```
> data_all <- read.csv("star_classification.csv", header=T)
```

I have imported the data into an object called "data_all", which will be used to maintain a reference of the original values.

To quickly confirm that the data was correctly processed, we can look at the first several rows using the head() function.

```
> head(data_all)
```

```
> head(data_all)
        obj_ID    alpha      delta        u        g        r        i        z run_ID rerun_ID cam_col field_ID  spec_obj_ID  class  redshift plate   MJD fiber_ID
1 1.237661e+18 135.6891 32.4946318 23.87882 22.27530 20.39501 19.16573 18.79371   3606      301       2       79 6.543777e+18 GALAXY 0.6347936  5812 56354      171
2 1.237665e+18 144.8261 31.2741849 24.77759 22.83188 22.58444 21.16812 21.61427   4518      301       5      119 1.176014e+19 GALAXY 0.7791360 10445 58158      427
3 1.237661e+18 142.1888 35.5824442 25.26307 22.66389 20.60976 19.34857 18.94827   3606      301       2      120 5.152200e+18 GALAXY 0.6441945  4576 55592      299
4 1.237663e+18 338.7410 -0.4028276 22.13682 23.77656 21.61162 20.50454 19.25010   4192      301       3      214 1.030107e+19 GALAXY 0.9323456  9149 58039      775
5 1.237680e+18 345.2826 21.1838656 19.43718 17.58028 16.49747 15.97711 15.54461   8102      301       3      137 6.891865e+18 GALAXY 0.1161227  6121 56187      842
6 1.237680e+18 340.9951 20.5894763 23.48827 23.33776 21.32195 20.25615 19.54544   8102      301       3      110 5.658977e+18    QSO 1.4246590  5026 55855      741
```

Table 1: Kaggle Feature Descriptions

No.	**Feature**	**Description**
1	obj_ID	Object Identifier, the unique value that identifies the object in the image catalog used by the CAS
2	alpha	Right Ascension angle (at J2000 epoch)
3	delta	Declination angle (at J2000 epoch)
4	u	Ultraviolet filter in the photometric system
5	g	Green filter in the photometric system
6	r	Red filter in the photometric system
7	i	Near Infrared filter in the photometric system
8	z	Infrared filter in the photometric system
9	run_ID	Run Number used to identify the specific scan
10	rerun_ID	Rerun Number to specify how the image was processed
11	cam_col	Camera column to identify the scanline within the run
12	field_ID	Field number to identify each field
13	spec_obj_ID	Unique ID used for optical spectroscopic objects (this means that 2 different observations with the same spec_obj_ID must share the output class)
14	class	object class (galaxy, star or quasar object)
15	redshift	redshift value based on the increase in wavelength
16	plate	plate ID, identifies each plate in SDSS
17	MJD	Modified Julian Date, used to indicate when a given piece of SDSS data was taken
18	fiber_ID	fiber ID that identifies the fiber that pointed the light at the focal plane in each observation

2 Exploratory Data Analysis

For exploratory data analysis the goal is to get an understanding of the makeup of the data, including what types of independent and dependent variables there are, whether the output classes are balanced or not, and whether there are missing values or outliers, and so on. This information will be used to inform if it is necessary to take preprocessing steps, such as feature reduction or normalization, as well as the types of classifiers that might be the best to try later on.

2.1 Understanding the Data & Data Preprocessing

We can use R's structure function to get an overview of what kinds of features there are.

```
> str(data_all)
'data.frame':        100000 obs. of  18 variables:
 $ obj_ID     : num  1.24e+18 1.24e+18 1.24e+18 1.24e+18 1.24e+18 ...
 $ alpha      : num  136 145 142 339 345 ...
 $ delta      : num  32.495 31.274 35.582 -0.403 21.184 ...
 $ u          : num  23.9 24.8 25.3 22.1 19.4 ...
 $ g          : num  22.3 22.8 22.7 23.8 17.6 ...
 $ r          : num  20.4 22.6 20.6 21.6 16.5 ...
 $ i          : num  19.2 21.2 19.3 20.5 16 ...
 $ z          : num  18.8 21.6 18.9 19.3 15.5 ...
 $ run_ID     : int  3606 4518 3606 4192 8102 8102 7773 7773 3716 5934 ...
 $ rerun_ID   : int  301 301 301 301 301 301 301 301 301 301 ...
 $ cam_col    : int  2 5 2 3 3 3 2 2 5 4 ...
 $ field_ID   : int  79 119 120 214 137 110 462 346 108 122 ...
 $ spec_obj_ID: num  6.54e+18 1.18e+19 5.15e+18 1.03e+19 6.89e+18 ...
 $ class      : chr  "GALAXY" "GALAXY" "GALAXY" "GALAXY" ...
 $ redshift   : num  0.635 0.779 0.644 0.932 0.116 ...
 $ plate      : int  5812 10445 4576 9149 6121 5026 11069 6183 6625 2444 ...
 $ MJD        : int  56354 58158 55592 58039 56187 55855 58456 56210 56386 54082 ...
 $ fiber_ID   : int  171 427 299 775 842 741 113 15 719 232 ...
```

First, we will take a closer look at the categorical dependent variable "class". As it is currently encoded as a character type, it will need to be converted into a factor for later use.

```
> data <- data_all
> data$class <- as.factor(data$class)
```

To determine whether it is a binary or multiclass problem, we can ask R to calculate the number of names in the output category.

```
> print(ifelse(length(names(table(data$class)))==2,"Binary  Classification",
  "Multiclass Classification"))
[1] "Multiclass Classification"
```

So, this is a multiclass classification problem. We can also create a table to see that there are three classes as expected as well as show how many observations belong to each class.

```
> table(data_all$class)

GALAXY    QSO   STAR
 59445  18961  21594
```

As seen above, the classes are imbalanced; there are nearly three times as many galaxy observations as there are quasars (QSO) or stars or a roughly 3:1:1 ratio. We will keep this in mind when evaluating the results of our models.

All of the independent features have number or integer types with numerical values, but they are not all numerical. For example, the `cam_col` feature identifies the column within the run where the observation was seen [3]. It consists only of integer values between 1 and 6 and as such is actually an ordinal categorical variable. However, before moving forward with an evaluation of the remaining fields, there are a few that can be preemptively eliminated.

2.1.1 Identifying Irrelevant Features

Firstly, there are quite a few features in this dataset that are used to identify the unique observations. The SDSS Glossary describes the field `obj_ID` as follows:

> *A number identifying an object in the image catalog used by the CAS. It is a bit-encoded integer of run, rerun, camcol, field, object."* [3]

Since each of these columns is part of SDSS' system for distinguishing which observation is which, we can assume that there is no correlation to the final output label and they can be removed from consideration.

Furthermore, we could also have noticed that the `rerun_ID` column is a constant feature; that is across all observations it has only a single value. This would also have eliminated the feature from the dataset as it cannot provide any information. This can be confirmed by checking whether any feature has 0 variance. The `data_all_indep` object is simply `data_all` with the `class` column removed, which is necessary here as var() cannot be called on a factor.

```
> x <- sapply(data_all_indep, function(v) var(v)==0)
> x[TRUE]
     obj_ID       alpha       delta           u           g           r           i
      FALSE       FALSE       FALSE       FALSE       FALSE       FALSE       FALSE
          z      run_ID    rerun_ID     cam_col    field_ID spec_obj_ID    redshift
      FALSE       FALSE        TRUE       FALSE       FALSE       FALSE       FALSE
      plate         MJD    fiber_ID
      FALSE       FALSE       FALSE
> x[x==TRUE]
    rerun_ID
      TRUE
```

Similarly, the `spec_obj_ID` feature is described as follows:

> *A unique bit-encoded 64-bit ID used for optical spectroscopic objects... It is generated from plateid, mjd, and fiberid.* [3]

So, as these fields are also used for identifying objects rather than their characteristics, we will remove them from the dataset. Incidentally, the `MJD` feature indicates Modified Julian Date, referring to the date when

the image containing each observation was taken represented in a particular date formatting convention sometimes used by astronomers. We would otherwise have made the same assumption that this also will not provide any information about the object class.

2.1.2 Missing Data & Duplicates

We want to check if there are any particular observations that are irregular and may be better off removed from the dataset. This should be done before discarding the irrelevant columns, as it's possible the observations' values for these features could highlight some inconsistencies that would otherwise be lost.

We can first quickly check that there are no null values present.

```
> any(is.na(data_all))
[1] FALSE
```

We can also check for duplicated data, which should be removed if present as it could adversely affect our models later on.

```
> any(duplicated(data_all))
[1] FALSE
```

Then, to get a visual indication of if there are any outliers in the data, we will generate box plots for each column. Since the ranges of each feature vary quite a bit and we have not standardized the data, we can make a separate plot for each.

```
> for (i in 1:length(data_all_indep)) {
        boxplot(data_all_indep[,i], main=names(data_all_indep[i]), type="l"  }
```

In the u, g, and z plots, there was a clear irregularity.

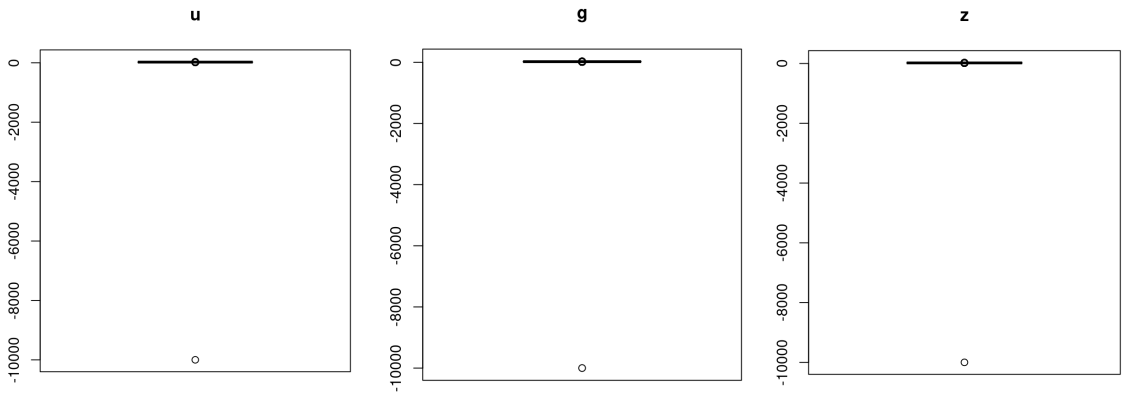

Figure 1: u, g, r original boxplots

Displaying the summaries of these three features, we can see that the outlying value is equal to -9999, so we can then look for any observations where this number is present.

```
> summary(subset(data_all, select=c(u,g,r)))
       u                 g                 r
 Min.   :-9999.00   Min.   :-9999.00   Min.   : 9.822
 1st Qu.:   20.35   1st Qu.:   18.96   1st Qu.:18.136
 Median :   22.18   Median :   21.10   Median :20.125
```

```
     Mean   :   21.98   Mean   :   20.53   Mean   :19.646
     3rd Qu.:   23.69   3rd Qu.:   22.12   3rd Qu.:21.045
     Max.   :   32.78   Max.   :   31.60   Max.   :29.572
```

```
> data_all[data_all$u == -9999 | data_all$g == -9999 | data_all$r == -9999,]
```

```
> data_all[data_all$u == -9999 | data_all$g == -9999 | data_all$r == -9999,]
           obj_ID    alpha      delta     u     g       r        i     z run_ID rerun_ID cam_col field_ID  spec_obj_ID
79544 1.237649e+18 224.0065 -0.6243039 -9999 -9999 18.1656 18.01675 -9999    752      301       2      537 3.731277e+18
      class     redshift plate   MJD fiber_ID
79544  STAR 8.934163e-05  3314 54970      162
```

There is only one observation returned, the one with id 79544. Since its values for u, g, and z are so far off from the rest of the observations, we will consider them erroneous and remove the observation from the dataset.

```
> data <- data_all[-c(79544), ]
> dim(data)
[1] 99999    18
```

The plots for these three features now show a much more typical distribution.

```
> boxplot(data[,c("u", "g", "r")])
```

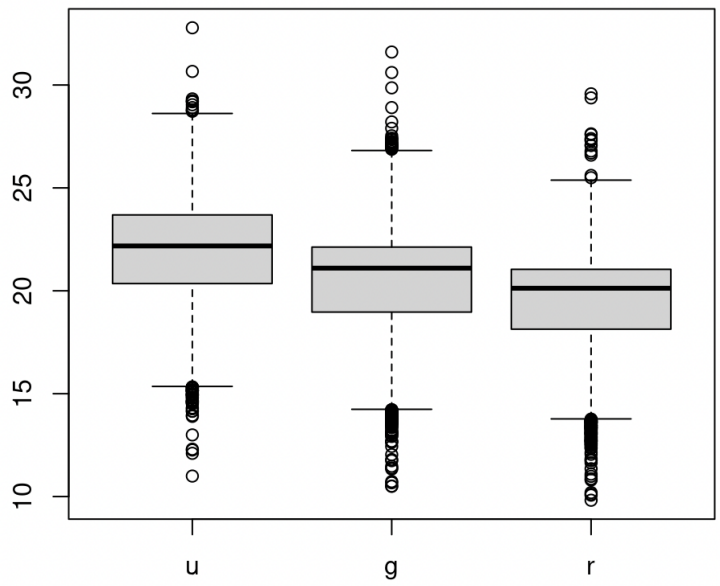

Figure 2: u, g, r boxplots after outlier removal

Now that we expect the data to be free of irregularities, we can go ahead and remove the features as described in part 2.1.1.

```
> drops <- c('obj_ID', 'run_ID', 'rerun_ID', 'cam_col', 'field_ID', 'spec_obj_ID',
'plate', 'MJD', 'fiber_ID' )
> data <- data_all[ , !(names(data_all) %in% drops)]
> dim(data)
```

```
[1] 99999     9
```

2.1.3 Feature Distribution

We are left with eight independent variables that will be used for prediction purposes.

```
> summary(data)
```

alpha	delta	u	g	r
Min. : 0.0055	Min. :-18.785	Min. :11.00	Min. :10.50	Min. : 9.822
1st Qu.:127.5177	1st Qu.: 5.147	1st Qu.:20.35	1st Qu.:18.97	1st Qu.:18.136
Median :180.9005	Median : 23.646	Median :22.18	Median :21.10	Median :20.125
Mean :177.6287	Mean : 24.136	Mean :22.08	Mean :20.63	Mean :19.646
3rd Qu.:233.8950	3rd Qu.: 39.902	3rd Qu.:23.69	3rd Qu.:22.12	3rd Qu.:21.045
Max. :359.9998	Max. : 83.001	Max. :32.78	Max. :31.60	Max. :29.572

i	z	class	redshift
Min. : 9.47	Min. : 9.612	Length:99999	Min. :-0.009971
1st Qu.:17.73	1st Qu.:17.461	Class :character	1st Qu.: 0.054522
Median :19.41	Median :19.005	Mode :character	Median : 0.424176
Mean :19.08	Mean :18.769		Mean : 0.576667
3rd Qu.:20.40	3rd Qu.:19.921		3rd Qu.: 0.704172
Max. :32.14	Max. :29.384		Max. : 7.011245

The summary of the remaining data shows the descriptive statistics for each feature. Each is numerical, so we will use boxplots and density plots to get a better idea of how they are distributed. First, in Figure 3, we have for each feature a comparison between the boxplots per class. In Figure 4, for each feature we have plotted a line per class corresponding to how frequently values for the features occur.

In the boxplots, we can see that the `alpha` and `delta` features follow a uniform distribution, while the remaining features are either normal skewed normal.[4]

In both sets of plots, it appears that quite a few features have similar distributions across each class, particularly so in `alpha` and `delta`. As such, this may be an indication that these features will not be able to provide much distinguishing information in regards class identification; however we will require a more rigorous examination of these features' information gain capacity later on.

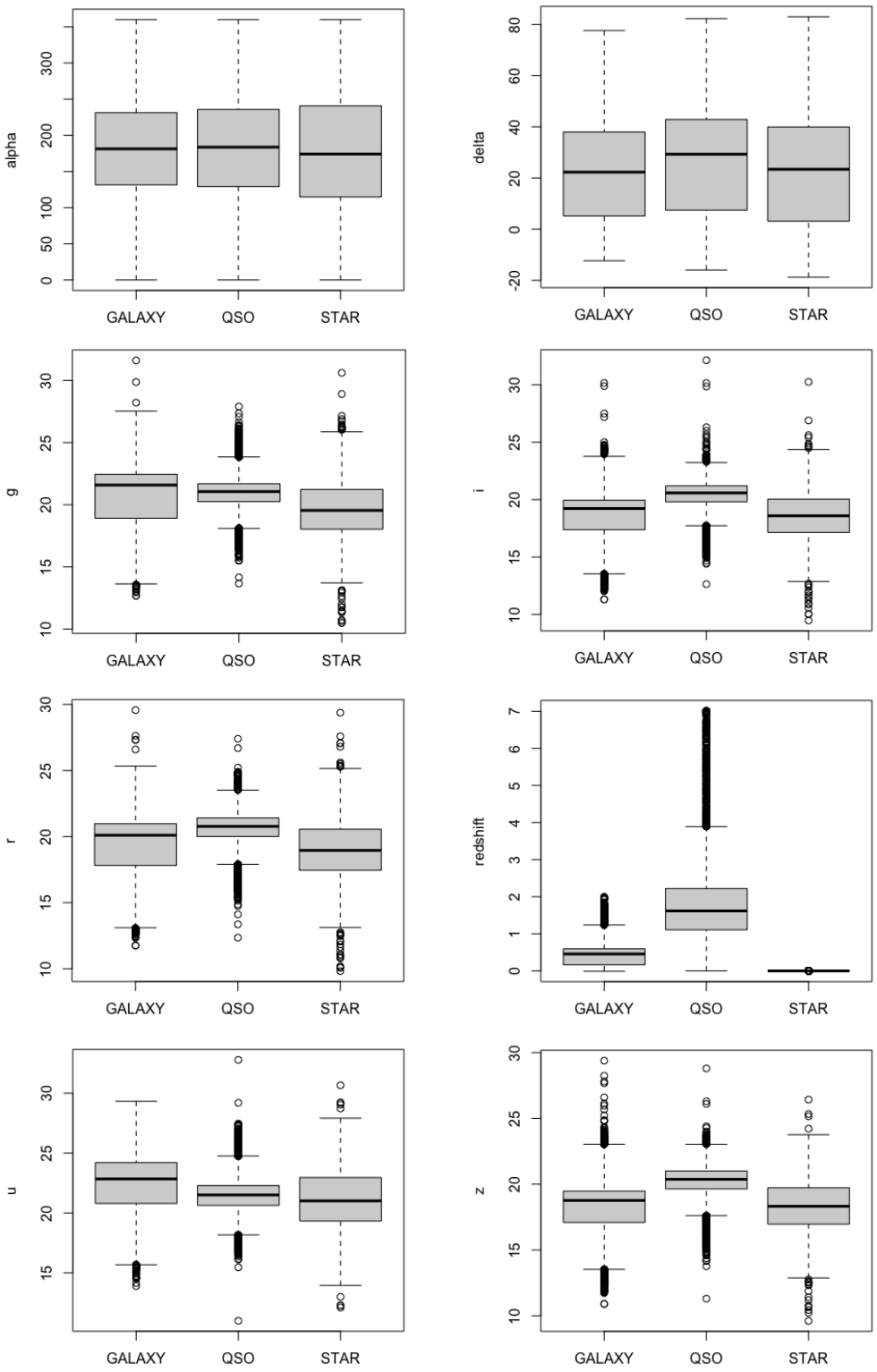

Figure 3: Boxplots for all Independent Features

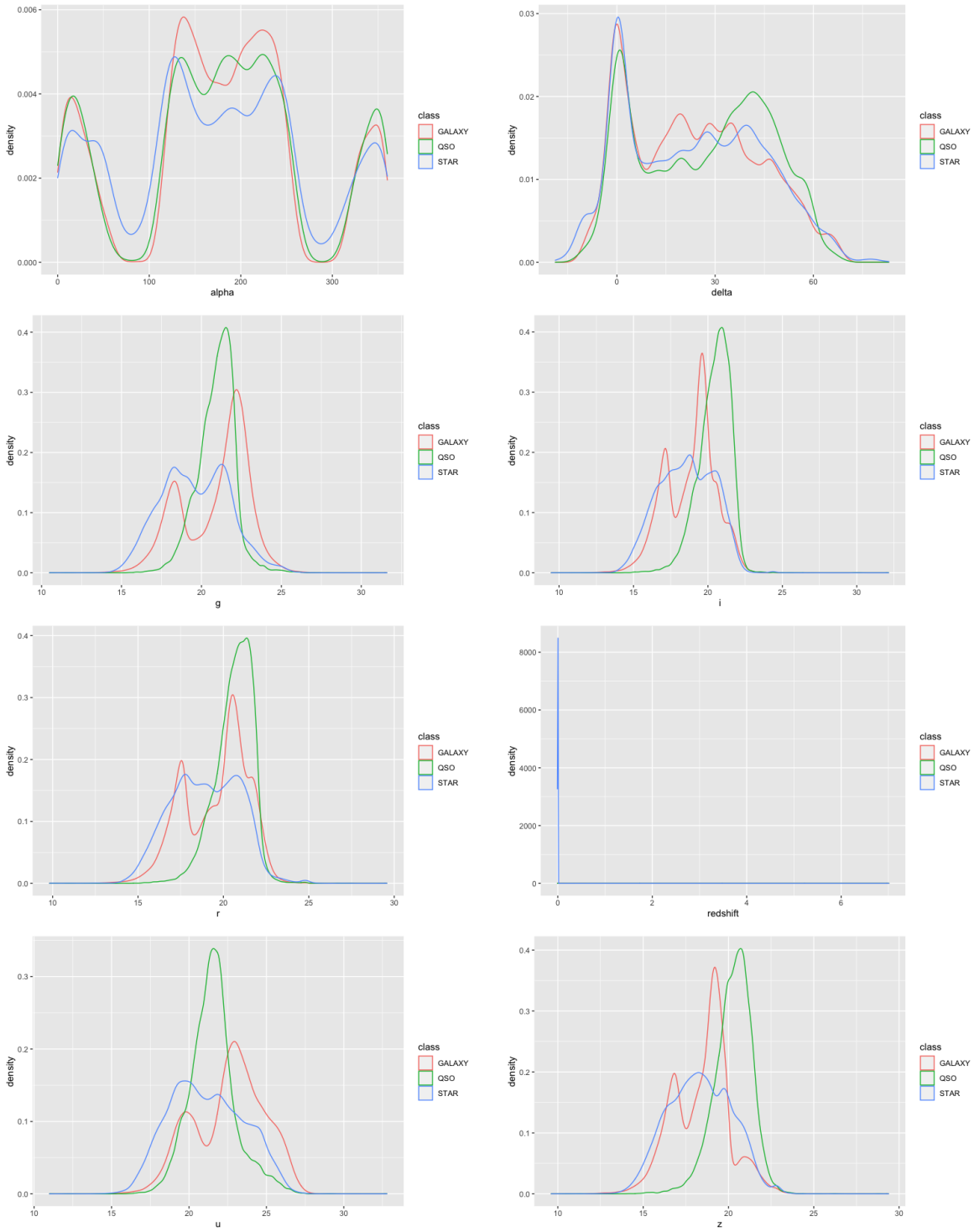

Figure 4: Density Plots for all Independent Features

Of note is the redshift feature, whose values are unevenly distributed. Specifically, looking at the boxplot for STAR and redshift, we can see that the range of values is significantly smaller than that of the other classes and makes it is difficult to see all three together in a single graph. We could instead plot the log(redshift) values as it might provide a clearer picture of the relationship between classes, but this would omit some any observations that have negative values of redshift. We can see that this would eliminate 14,136 rows or approximately 14 percent of the dataset, likely leading to a skewed plot.

```
# find the number of observations with redshift less than or equal to 0
> nrow(data[data$redshift<=0,])
[1] 14136

# find percentage of observations with redshift less than or equal to 0
> (nrow(data[data$redshift<=0,])/nrow(data)) * 100
[1] 14.13614

# distribution of redshift values less than or equal to 0
> table(data[data$redshift<=0,]$class)

GALAXY    QSO   STAR
454       0  13682
```

As expected, plotting the density graph of the original data leads observations being removed from the result. Also, a large majority of the omitted rows are coming from the STAR class, which is a significant imbalance.

```
> ggplot(data[ , c("redshift", "class")] , aes(x = log(redshift), colour = class))
    + geom_density()
Warning messages:
1: In log(redshift) : NaNs produced
2: In log(redshift) : NaNs produced
3: Removed 14136 rows containing non-finite values (stat_density).
```

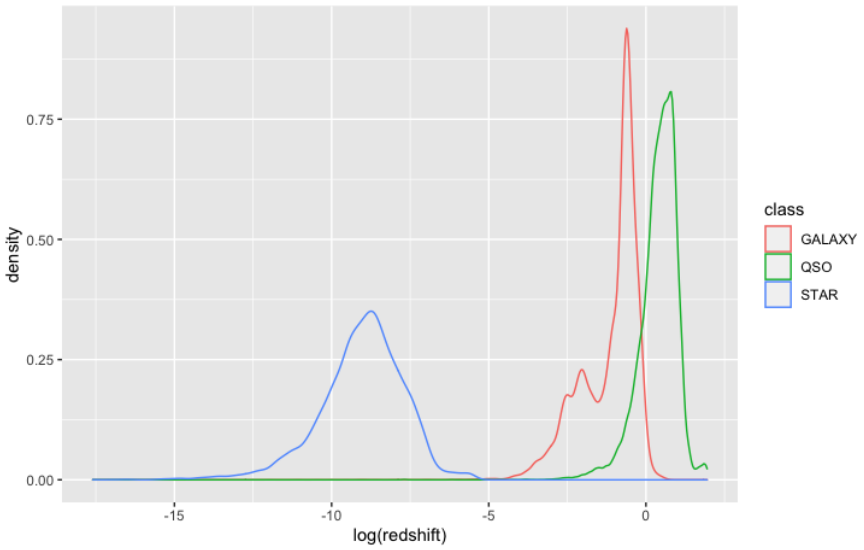

Figure 5: Log(redshift) density plot; original data

To avoid this, we can look at log plots of min-max normalized data instead. The min-max function is described further in Section 2.3.

```
# define Min-Max normalization function
> min_max_norm <- function(x) {
          (x - min(x)) / (max(x) - min(x)) }

# scale redshift data to between 0 and 1
> redshift_norm <- as.data.frame(lapply(subset(data, select = c(redshift)),
    min_max_norm))

# generate density plot of log scaled data
> redshift_norm = red_norm
> ggplot(redshift_norm , aes(x = log(redshift_norm$redshift), colour = class))
    + geom_density()
Warning messages:
1: Use of `redshift_norm$redshift` is discouraged. Use `redshift` instead.
2: Removed 1 rows containing non-finite values (stat_density).

# generate boxplot of log scaled data
> boxplot(log(redshift)~class,data=redshift_norm, xlab="Output Class",
ylab="log(redshift_norm)")
Warning message:
In bplt(at[i], wid = width[i], stats = z$stats[, i], out = z$out[z$group ==  :
  Outlier (-Inf) in boxplot 1 is not drawn
```

The normalized plots show a much different picture. Both generate a warning that one row with value -Inf is omitted (the result of attempting to take log(0)), which corresponds to the minimum value in the set. Now, we can more clearly see the distribution across classes, and it seems that `redshift` may even be a good predictor of class.

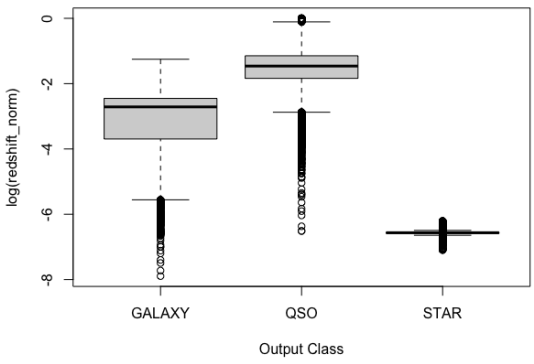

Figure 6: Log(redshift) boxplot; normalized data

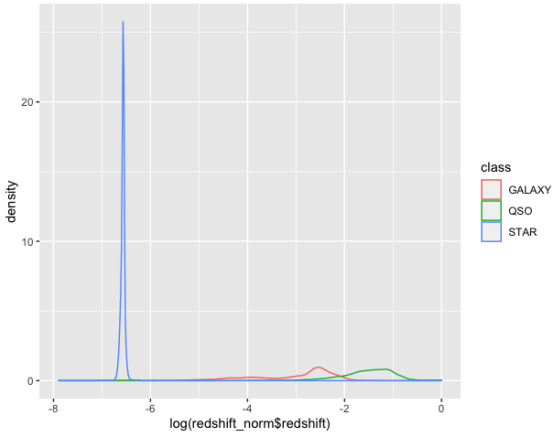

Figure 7: Log(redshift) density plot; normalized data

2.2 Feature Correlation

The plots from the previous section also seemed to show that several features had similar profiles, seeming to indicate that there may be some multicollinearity between them. To investigate further, we can start with a pairwise scatterplot.

```
> pairs(data_indep,
+ col = c("green", "cornflowerblue", "purple")[data$class],
+ pch = c(".", ".", ".")[data$class],
+ labels = names(data_indep))
```

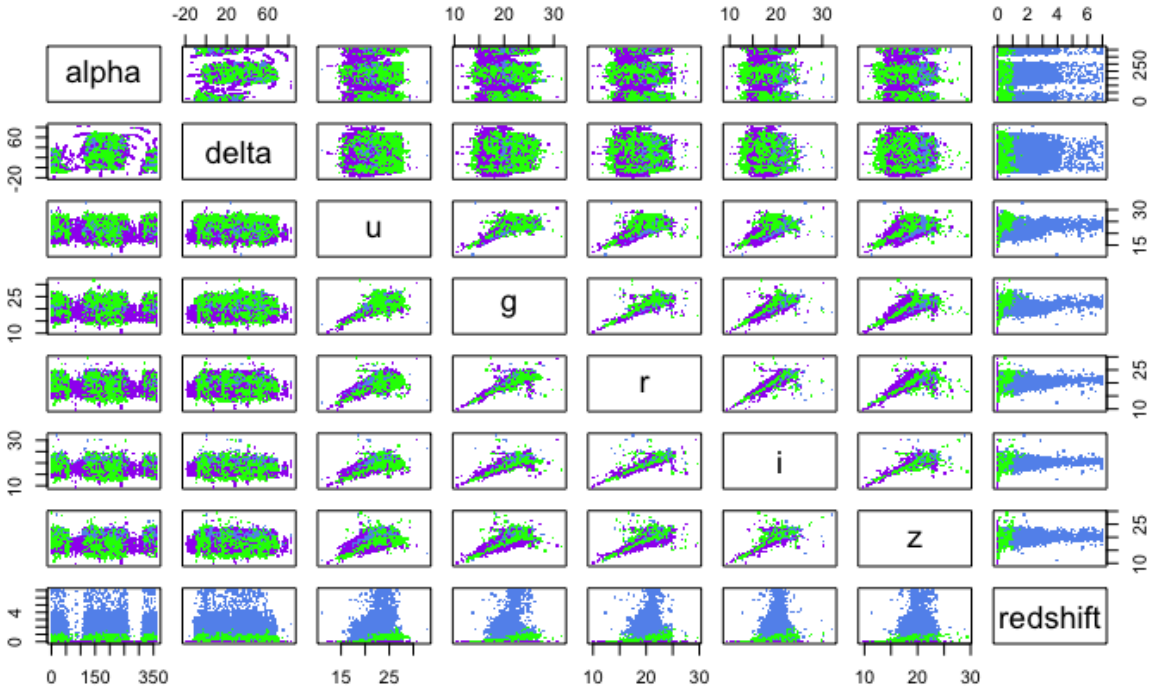

Figure 8: R pairs()

As suspected, each of the plots between `u`, `g`, `r`, `i`, and `z` show a fairly distince positive correlation, which can be more clearly enumerated using the `corrplot` package (Figure 9.

```
> library(corrplot)
> corrplot.mixed(cor(data_indep), order = 'AOE')
```

Again the correlation is quite clear, and most likely we will want to remove one or more from consideration. However, the question remains as to which of the features should be kept and which discarded, which we will attempt to answer further on with an investigation of feature entropy and which feature(s) can provide the most identifying information.

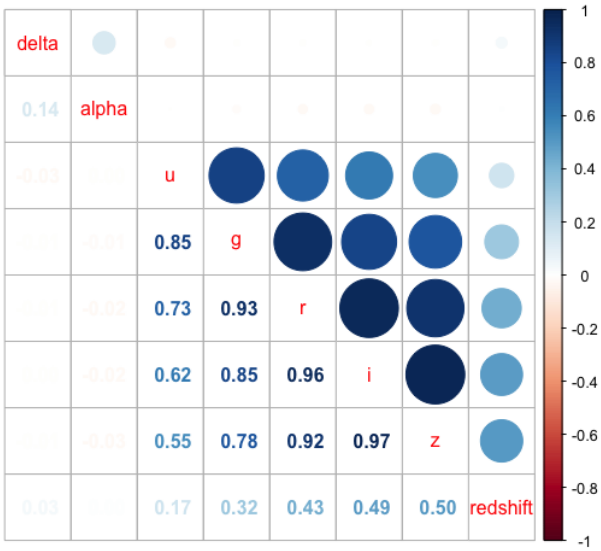

Figure 9: Feature Correlation Plot

2.3 Normalization

As we can see from the density plots above, the features' values all exist in varying ranges. So, we will need to perform data normalization so that our models are not affected by the differing scales. In particular, distance-based methods such as KNN and SVM would be highly susceptible to this discrepancy (although tree-based methods would not be).

The majority of features do not follow a "normal" Gaussian distribution, which we noted in Section 2.1.3. As such, we will choose to perform normalization, or min-max scaling, as opposed to standardizing using unit mean and standard deviation. [5] Normalization scales all features so that their ranges are between 0 and 1 while maintaining the relationships to other values.

```
# define Min-Max normalization function
> min_max_norm <- function(x) {
      (x - min(x)) / (max(x) - min(x))
  }

# apply scaling and save in new df named X
> X <- as.data.frame(lapply(data_indep, min_max_norm))
> summary(data_norm)
 alpha            delta              u                g                r
 Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000
 1st Qu.:0.3542   1st Qu.:0.2351   1st Qu.:0.4295   1st Qu.:0.4012   1st Qu.:0.4210
 Median :0.5025   Median :0.4169   Median :0.5133   Median :0.5024   Median :0.5217
 Mean   :0.4934   Mean   :0.4217   Mean   :0.5088   Mean   :0.4802   Mean   :0.4974
 3rd Qu.:0.6497   3rd Qu.:0.5766   3rd Qu.:0.5826   3rd Qu.:0.5509   3rd Qu.:0.5682
 Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.0000
    i                z              redshift
 Min.   :0.0000   Min.   :0.0000   Min.   :0.000000
 1st Qu.:0.3644   1st Qu.:0.3970   1st Qu.:0.009185
 Median :0.4382   Median :0.4750   Median :0.061833
 Mean   :0.4241   Mean   :0.4631   Mean   :0.083552
 3rd Qu.:0.4820   3rd Qu.:0.5214   3rd Qu.:0.101712
 Max.   :1.0000   Max.   :1.0000   Max.   :1.000000

# Replace class feature
> X$class <- data$class
```

3 Individual Classifiers

For individual classifiers, we will focus on Decision Tree, KNN, and Naive Bayes. SVM will also be used as a baseline metric. The major factors in this choice are as follows:

- Multicollinearity of features. We need some techniques to help evaluate which of the similar features detected during EDA we should keep and which to remove.
- Class imbalance. In our classifiers, we need to keep in mind that the `GALAXY` class has approximately twice as many observations as the other two and account for this when possible.
- Large size of the dataset. This represents a major advantage, and we can expect that the results of even individual classifiers will be fairly good.

Decision trees are a good general purpose classifier, as they are flexible for use with both categorical and numerical features. In addition, their results are easy to interpret and visualize, which will help when drawing conclusions from the data. We will also use the splitting process of Decision Trees to further evaluate the usefulness of features, aiding in reducing multicollinearity.

The K-Nearest Neighbors classifier is distance-based, so we can apply this to our features (all of which are numerical) without requiring any further modification aside from the normalization which we have already done. Low values of k are prone to high variance, while high values of k are prone to high bias particularly given the imbalance of the dataset. So, we will briefly attempt a few different values in order to determine what might be the best balance between the two.

Naive Bayes is also a good general purpose classifier, with one major disadvantage being the underlying assumption that it assumes each input variable is independent of the others. However, despite the high bias it tends to perform fairly well, and particularly as there is an abundance of data we expect that it will be successful in this case.

3.1 Train/Test Split

Before generating any models, we will separate the data into a training set and a testing set. Since the number of observations present is quite large, we will use a 70/30 split.

```
> train_size = floor(nrow(data)*0.7)

# make sure that sampling is reproducible
> set.seed(42)
> train_idx=sample(1:nrow(data), size=train_size)  # without replacement is default

> head(train_idx)
[1] 61413 54425 99556 74362 46208 47128
```

Now, using the normalized data from before, we can split into training and test data sets.

```
> df_train = X[train_idx,]
> dim(df_train)
[1] 69999     9
> df_test=X[-train_idx,]
> dim(df_test)
[1] 30000     9
```

We also want to verify that our two sets have similar distributions across classes. By looking at the proportions of each subset compared to that of the full set, we can see that they are very close, which is as expected as the original data set is large enough.

```
> prop.table(table(X$class))

   GALAXY       QSO      STAR
0.5944559 0.1896119 0.2159322
> prop.table(table(df_train$class))

   GALAXY       QSO      STAR
0.5949228 0.1894027 0.2156745
> prop.table(table(df_test$class))

   GALAXY       QSO      STAR
0.5933667 0.1901000 0.2165333
```

3.2 Decision Tree

In the EDA section, we determined that several features contain some multicollinearity. In general, we want to avoid using variables which are too similar to each other as this can affect the outcome of the model, so we would like to determine which of the features should be used and which discarded. Since decision trees are built based on the amount of information a given feature can provide about the output class, we will first create a tree using built-in functionality, then take a closer look at the results. We will use the training and test sets created above which have been normalized; this is not a requirement for Decision Trees but it does not affect the model, either.

3.2.1 rpart()

To create a decision tree model, we will use the `rpart()` package. Since all of the features in question are continuous numerical values, using this will handle for us the process of determining how to split each feature.

```
> library(rpart)
> library(rpart.plot)
> dt.mod <- rpart(class~., data = df_train, method = 'class')
> rpart.plot(dt.mod)
```

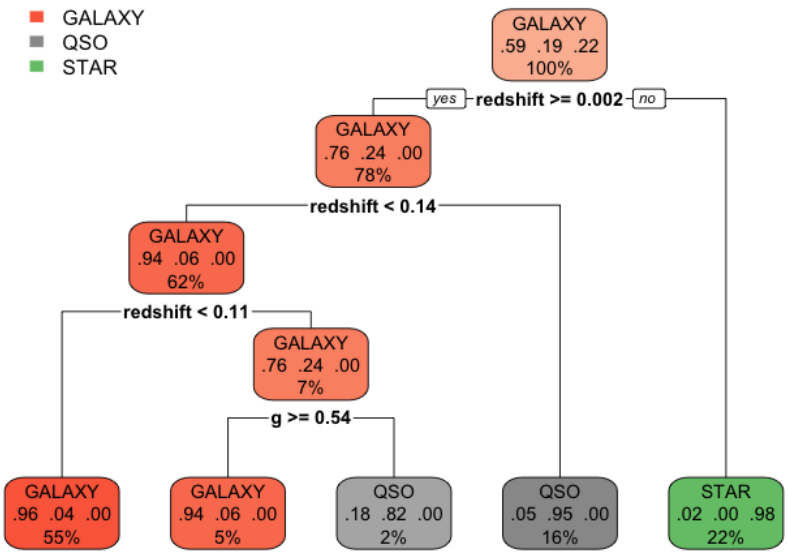

Figure 10: rpart decision tree

As shown in the plot (Figure 10), only the `redshift` and `g` features have been used in the construction of the tree.

We will check the accuracy of the model to get an idea of its baseline performance, creating a function called `accuracy` which takes a confusion matrix as input.

```
# function to compute accuracy
> accuracy <- function(x){sum(diag(x)/(sum(rowSums(x)))) * 100}
```

```
# prediction with test dataset
> dt.pred <- predict(dt.mod, df_test, type = 'class')

# resulting confusion matrix
> table(dt.pred, df_test$class)

dt.pred  GALAXY   QSO  STAR
  GALAXY  17251   782     0
  QSO       394  4921     0
  STAR      156     0  6496

# accuracy
> accuracy(table(dt.pred, df_test$class))
[1] 95.56
```

The tree seems to do fairly well, with an accuracy of over 95%, when given inputs which it has never seen. However, accuracy alone does not provide the whole picture of how well the model actually performs and this is particularly important to note when the output classes are imbalanced as they are here [6].

We will also look at the F1 score, as this is a more suitable measure for imbalanced data. For multiclass classification, this calculated on a per-class basis, with "positive" for a class indicating that a given observation was labeled with that class. The F1 score is defined as "the harmonic mean of precision and recall":

$$F1(class = a) = 2 \cdot \frac{precision * recall}{precision + recall}$$

Precision is a measure of correct labels (true positives) as a fraction of the total number of positive labels made by the model, including those which were incorrect (false positives).

$$Precision(class = a) = \frac{TP(class = a)}{TP(class = a) + FP(class = a)}$$

Recall is a measure of correct labels as a fraction of the total number of actually positive observations, including those which were missed (false negatives).

$$Recall(class = a) = \frac{TP(class = a)}{TP(class = a) + FN(class = a)}$$

The `confusionMatrix` function of `caret` calculates the F1 measure when "everything" is passed to the mode argument.

```
> confusionMatrix(table(dt.pred, df_test$class), mode="everything")
Confusion Matrix and Statistics

dt.pred  GALAXY   QSO  STAR
  GALAXY  17251   782     0
  QSO       394  4921     0
  STAR      156     0  6496
```

```
Overall Statistics

               Accuracy : 0.9556
                 95% CI : (0.9532, 0.9579)
    No Information Rate : 0.5934
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9209

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: GALAXY Class: QSO Class: STAR
Sensitivity                 0.9691     0.8629      1.0000
Specificity                 0.9359     0.9838      0.9934
Pos Pred Value              0.9566     0.9259      0.9765
Neg Pred Value              0.9540     0.9683      1.0000
Precision                   0.9566     0.9259      0.9765
Recall                      0.9691     0.8629      1.0000
F1                          0.9628     0.8933      0.9881
Prevalence                  0.5934     0.1901      0.2165
Detection Rate              0.5750     0.1640      0.2165
Detection Prevalence        0.6011     0.1772      0.2217
Balanced Accuracy           0.9525     0.9233      0.9967
```

A measure of 1 is considered a "perfect" score, so we can see that the F1 score is high for `GALAXY` and `STAR`, and bit lower for `QSO`. This seems to stem from a slightly lower recall indicating that a number of `QSO` observations were missed. That being said, the measures are still in an acceptable range so it stands to reason that the two features `redshift` and `g` together are quite descriptive of class output. In the other models, we will compare the performance of using just these features versus the entire set.

3.2.2 Entropy

To find the amount of entropy we can create a function based on the formula for entropy [7]. Applying this to the entire dataset, we get an entropy of 1.37. This is greater than one as expected as the dependent variable is multiclass.

```
> entropy<-function(v) { p=v/sum(v); -1*sum(p*log(p,2))}
> total_entropy<-entropy(table(data$class))
> total_entropy
[1] 1.378412
```

Then, we want to find the the maximum amount of information gain (entropy lost) given a particular feature. Since the independent variables are numeric, we can look at the output from `rpart()`. The first node contains all observations. Looking at the primary splits, dividing the dataset based on the redshift feature (where any observation with a value less than 0.002014167 goes to the right) will improve the information gain the most. The surrogate splits are used in case of missing values in the feature to split on, which we can disregard in

this case since we have no missing values.

```
> summary(dt)
Call:
rpart(formula = class ~ ., data = df_train, method = "class")
  n= 69999

          CP nsplit rel error    xerror        xstd
1 0.51983777      0 1.0000000 1.0000000 0.004580527
2 0.35143714      1 0.4801622 0.4801622 0.003693271
3 0.01292541      2 0.1287251 0.1294657 0.002080008
4 0.01000000      4 0.1028743 0.1093282 0.001919619

Variable importance
redshift        z        u        i        g        r    delta
      80        8        3        3        3        2        1

Node number 1: 69999 observations,    complexity param=0.5198378
  predicted class=GALAXY  expected loss=0.4050772  P(node) =1
    class counts: 41644 13258 15097
   probabilities: 0.595 0.189 0.216
  left son=2 (54542 obs) right son=3 (15457 obs)
  Primary splits:
      redshift < 0.002014167  to the right, improve=18686.090, (0 missing)
      z        < 0.5234651    to the left,  improve= 4209.846, (0 missing)
      i        < 0.4664107    to the left,  improve= 2564.543, (0 missing)
      u        < 0.5204515    to the right, improve= 2512.278, (0 missing)
      g        < 0.538862     to the right, improve= 2140.938, (0 missing)
  Surrogate splits:
      u     < 0.33641      to the right, agree=0.787, adj=0.035, (0 split)
      g     < 0.2895119    to the right, agree=0.783, adj=0.017, (0 split)
      delta < 0.0923453    to the right, agree=0.782, adj=0.013, (0 split)
      r     < 0.7362       to the left,  agree=0.780, adj=0.002, (0 split)
      i     < 0.1115334    to the right, agree=0.779, adj=0.000, (0 split)
```

The right side of this split is already a leaf node, with a prediction of STAR (in green). We can check that this actually contains all star observations.

```
> table(df_train[df_train$redshift< 0.00201416,]$class)
GALAXY    QSO   STAR
   357      3  15097
> table(df_train$class)

GALAXY    QSO   STAR
 41644  13258  15097
```

On the left side of the split, we repeat the process to find the most informative way to divide the reduced

dataset into two. In this case, we again determine that splitting using the redshift feature is ideal, this time with values less than 0.1437241 going to the left. This process is continued until we have either minimized the entropy per node, that is each node is as pure as possible, or we have reached the minimum split value (default minsplit=20) which requires that at least that many observations must be present in the node for the split to occur.

```
Node number 2: 54542 observations,    complexity param=0.3514371
  predicted class=GALAXY  expected loss=0.2430237  P(node) =0.7791826
    class counts: 41287 13255     0
   probabilities: 0.757 0.243 0.000
  left son=4 (43375 obs) right son=5 (11167 obs)
  Primary splits:
      redshift < 0.1437241    to the left,  improve=13885.540, (0 missing)
      z        < 0.5234651    to the left,  improve= 5534.100, (0 missing)
      i        < 0.4664107    to the left,  improve= 3480.601, (0 missing)
      u        < 0.529037     to the right, improve= 2011.504, (0 missing)
      r        < 0.4388239    to the left,  improve= 1433.779, (0 missing)
  Surrogate splits:
      z     < 0.5268958    to the left,  agree=0.841, adj=0.222, (0 split)
      i     < 0.5025805    to the left,  agree=0.810, adj=0.073, (0 split)
      delta < 0.0791406    to the right, agree=0.795, adj=0.001, (0 split)
      alpha < 9.425899e-05 to the right, agree=0.795, adj=0.000, (0 split)

Node number 3: 15457 observations
  predicted class=STAR    expected loss=0.02329042  P(node) =0.2208174
    class counts:   357     3 15097
   probabilities: 0.023 0.000 0.977

... (nodes omitted)

Node number 18: 3725 observations
  predicted class=GALAXY  expected loss=0.06147651  P(node) =0.05321505
    class counts:  3496   229     0
   probabilities: 0.939 0.061 0.000

Node number 19: 1159 observations
  predicted class=QSO     expected loss=0.1837791  P(node) =0.01655738
    class counts:   213   946     0
   probabilities: 0.184 0.816 0.000
```

3.2.3 Variable Importance

To note, while the variable importance determined by `rpart()` did show that `redshift` had by far the most distinguishing power, the second highest importance was the feature `z` rather than `g`. We can use the `varImp()` to see this as well.

```
> varImp(dt.mod)
            Overall
alpha       78.6475
g         3232.3192
i         6397.3483
r         2151.0334
redshift 32922.6872
u         5577.9002
z        10039.7768
delta        0.0000
```

3.3 K-Nearest Neighbors

Another classifier we will try is KNN, a non-parametric distance-based algorithm.

First we will run it from the `class` package using all of the relevant features that we have so far to get a baseline understanding of how well it performs. The `knn()` function takes as input training and test datasets (without labels) as well as the ground truth classes of the training set. The default value for the number of neighbors is k=1.

```
> library(class)
# separate labels from independent variables
train_labels=df_train[,9]
test_labels=df_test[,9]

# generate model
knn.mod <- knn(df_train[,-c(9)], df_test[, -c(9)], cl=train_labels)

# check accuracy of results
> accuracy(table(knn.mod, test_labels))
[1] 92
```

As an experiment, we will test a few different values of k to get a rough idea of what will result in the best bias-variance trade-off. We focus on the range between k=1 and k=$\sqrt(n)$, as research suggests that this is often the optimal value [8].

```
# list of k-values
> kval = c(1, 3, 5, 9, 55, 101, 150, 250, 315)
> knn.acc = list()

# run model for each value of k
> for (x in kval) {
+     print(paste("Accuracy for k =", x))
+     mod <- knn(df_train[,-c(9)],df_test[, -c(9)],cl=train_labels,k=x)
+     acc <- accuracy(table(mod, test_labels))
+     print(acc)
+     knn.acc = append(knn.acc, acc)
+ }
[1] "Accuracy for k = 1"
[1] 92
[1] "Accuracy for k = 3"
[1] 92.87
[1] "Accuracy for k = 5"
[1] 92.7
[1] "Accuracy for k = 9"
[1] 92.48333
[1] "Accuracy for k = 55"
[1] 89.89333
[1] "Accuracy for k = 101"
```

```
[1] 88.43333
[1] "Accuracy for k = 150"
[1] 87.13333
[1] "Accuracy for k = 250"
[1] 85.19333
[1] "Accuracy for k = 315"
[1] 84.21
```

The results can be plotted for easier evaluation.

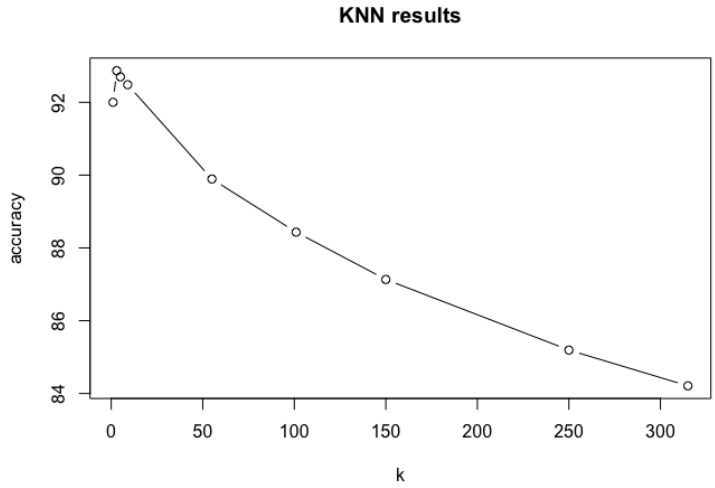

Figure 11: KNN with different values of k

As we can see, the accuracy peaks at k=3. Naturally this is not a rigorous test of the ideal k-value, but it clearly shows that for this dataset, smaller values of k result in better performance. This may be a result of the imbalanced output classes, as when there are more observations of a given class KNN will tend to be biased towards the dominant class.

This model was built using all of the features in the set, but we can repeat this test using only the features that we found had the most variable importance from the examination of the Decision Tree output, `redshift` and `g`.

```
> knn.2.acc = list()

# generate models using only redshift and g
> for (x in kval) {
+     print(paste("Accuracy for k =", x))
+     mod <- knn(df_train[,c("redshift", "g")],df_test[, c("redshift", "g")],cl=train_labels,k=x)
+     acc <- accuracy(table(mod, test_labels))
+     print(acc)
+     knn.2.acc = append(knn.2.acc, acc)
+ }
[1] "Accuracy for k = 1"
[1] 94.30333
```

```
[1] "Accuracy for k = 3"
[1] 95.97667
[1] "Accuracy for k = 5"
[1] 96.29333
[1] "Accuracy for k = 9"
[1] 96.37667
[1] "Accuracy for k = 55"
[1] 96.00333
[1] "Accuracy for k = 101"
[1] 95.66667
[1] "Accuracy for k = 150"
[1] 95.41
[1] "Accuracy for k = 250"
[1] 94.95333
[1] "Accuracy for k = 315"
[1] 94.65667
```

Plotting the results together, we see that the KNN model with limited feature input consistently performs better than when using the additional multicollinear independent variables. This again confirms the theory that `redshift` and `g` alone are quite good indicators of class, and moreover in the case of KNN, including the remaining multicollinear features results in too much noise for a successful classifier.

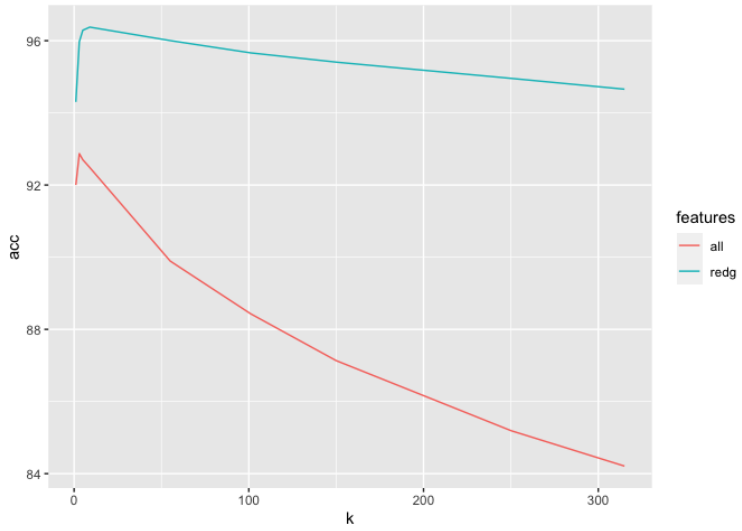

Figure 12: KNN with different feature sets

3.4 Naive Bayes

For Naive Bayes, we will again start with the base classifier using all features.

```
> library(naiveBayes)

# train model
> nb.mod = naiveBayes(class ~ ., data = df_train)

# predict with test data
> nb.pred <- predict(nb.mod, newdata = df_test)

> confusionMatrix(table(nb.pred, df_test$class), mode="everything")
Confusion Matrix and Statistics


nb.pred  GALAXY   QSO  STAR
  GALAXY  16204   704    35
  QSO      1453  4926    29
  STAR      144    73  6432

Overall Statistics

               Accuracy : 0.9187
                 95% CI : (0.9156, 0.9218)
    No Information Rate : 0.5934
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.859

 Mcnemar's Test P-Value : < 2.2e-16

Statistics by Class:

                     Class: GALAXY Class: QSO Class: STAR
Sensitivity                 0.9103     0.8638      0.9901
Specificity                 0.9394     0.9390      0.9908
Pos Pred Value              0.9564     0.7687      0.9674
Neg Pred Value              0.8777     0.9671      0.9973
Precision                   0.9564     0.7687      0.9674
Recall                      0.9103     0.8638      0.9901
F1                          0.9328     0.8135      0.9786
Prevalence                  0.5934     0.1901      0.2165
Detection Rate              0.5401     0.1642      0.2144
Detection Prevalence        0.5648     0.2136      0.2216
Balanced Accuracy           0.9249     0.9014      0.9905
```

We can again compare these results with a model generated using only redshift and g.

```
# model using redshift, g only
> nb.mod.2 = naiveBayes(class ~ ., data = df_train[,c("redshift", "g")])
> nb.pred.2 <- predict(nb.mod.2, newdata = df_test)
> confusionMatrix(table(nb.pred.2, df_test$class), mode="everything")
Confusion Matrix and Statistics

nb.pred.2 GALAXY   QSO  STAR
   GALAXY  17452  1196    53
   QSO       205  4456     0
   STAR      144    51  6443

Overall Statistics

               Accuracy : 0.945
                 95% CI : (0.9424, 0.9476)
    No Information Rate : 0.5934
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9005

 Mcnemar's Test P-Value : < 2.2e-16

Statistics by Class:

                     Class: GALAXY Class: QSO Class: STAR
Sensitivity                 0.9804     0.7813      0.9918
Specificity                 0.8976     0.9916      0.9917
Pos Pred Value              0.9332     0.9560      0.9706
Neg Pred Value              0.9691     0.9508      0.9977
Precision                   0.9332     0.9560      0.9706
Recall                      0.9804     0.7813      0.9918
F1                          0.9562     0.8599      0.9811
Prevalence                  0.5934     0.1901      0.2165
Detection Rate              0.5817     0.1485      0.2148
Detection Prevalence        0.6234     0.1554      0.2213
Balanced Accuracy           0.9390     0.8865      0.9918
```

Again the model with only redshift and g performs slightly better than the one using all features. Although both accuracy and F1 measures are relatively high, this can be attributed to the abundance of data, as typically Naive Bayes classification is susceptible to class imbalance.

3.5 Support Vector Machine

Lastly, we will use use a support vector machine model to get a different view of how well the data can be generalized. SVM models are trained by finding an $n-1$ dimensional hyperplane between all n independent variables, then new data can be classified by finding to which side each observation falls. The major downside of SVM is that the results are difficult to interpret, as anything more than a 3-dimensional hyperplane cannot be visualized. That being said, SVM tends to generate quite successful learners by exploiting all facets of the data, so we will use the model to determine a baseline for future improvement or modifications. Below we can see that in fact, the SVM predictions had higher accuracy and f1 scores per class than any other individual classifier.

```
> library(e1071)
> svm.mod = svm(class~., data=df_train, gamma=0.1, cost=10)
> svm.pred = predict(svm.mod, df_test, gamma=0.1, cost=10)
> confusionMatrix(svm.pred, df_test$class, mode="everything")
Confusion Matrix and Statistics

          Reference
Prediction GALAXY   QSO  STAR
    GALAXY  17326   489     4
    QSO       210  5208     1
    STAR      265     6  6491

Overall Statistics

               Accuracy : 0.9675
                 95% CI : (0.9654, 0.9695)
    No Information Rate : 0.5934
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9424

 Mcnemar's Test P-Value : < 2.2e-16

Statistics by Class:

                     Class: GALAXY Class: QSO Class: STAR
Sensitivity                 0.9733     0.9132      0.9992
Specificity                 0.9596     0.9913      0.9885
Pos Pred Value              0.9723     0.9611      0.9599
Neg Pred Value              0.9610     0.9799      0.9998
Precision                   0.9723     0.9611      0.9599
Recall                      0.9733     0.9132      0.9992
F1                          0.9728     0.9365      0.9792
Prevalence                  0.5934     0.1901      0.2165
Detection Rate              0.5775     0.1736      0.2164
```

```
Detection Prevalence     0.5940     0.1806     0.2254
Balanced Accuracy        0.9665     0.9523     0.9939
```

3.6 Summary

Below is a table showcasing the results (accuracy and F1 measures) of each individual classifier. For KNN and Naive Bayes, we have included the different values when the models were evaluated with all features versus only the two with the highest variable importance. SVM and KNN with `redshift` and `g` were the most successful, with an accuracy of over 96%.

	Decision Tree	KNN		Naive Bayes		SVM
		k=3; all features	k=9; Redshift, g	All features	redshift, g	
Accuracy	95.56	92.87	96.38	91.87	94.5	96.75
F1: Galaxy	0.9628	0.9419	0.9697	0.9328	0.9562	0.9728
F1: Quasar	0.8933	0.9286	0.9210	0.8135	0.8599	0.9365
F1: Star	0.9881	0.8926	0.9839	0.9786	0.9811	0.9792

Figure 13: Accuracy and F1 Score for Individual Classifiers

4 Ensemble Techniques

Ensemble machine learning methods are techniques that combine multiple individual classifiers in a variety of different ways. For this experiment, we will use the same kind of individual classifiers, decision trees, as our base or weak learners and apply three different ensemble methods to compare their results: Random Forest, Boosting, and Bagging.

In training ensemble models, the goal to reduce one of the two sources of test error, variance and bias. Random Forest and Boosting are both capable of reducing bias in a model. Bagging on the other hand is a variance reduction technique, and since the large size of the dataset naturally aids in reducing variance already, we predict that there will not be much improvement when using this technique.

To implement Boosting and Bagging, we will load the adabag package, which "implements Freund and Schapire's Adaboost.M1 algorithm and Breiman's Bagging algorithm using classification trees as individual classifiers." [9]

```
> install.packages("adabag")
> library(adabag)
```

4.1 Random Forest

Random Forest is an ensemble classifier which combines the outputs of several decision trees, in particular, each time choosing a random subset of independent variables to use to grow the tree. Random Forest is often used in order to improve the performance of single Decision Tree learners, which have a tendency to overfit to training data; although this is not particularly of concern in this dataset given when the large available training data, it is still worth trying. Although we have seen improvements in single trees when we use only the most "important" features, it is possible that there is some information contained in the unused features which can no longer be taken advantage of. Since the Random Forest classifier combines the learning ability of decision trees using different subsets, we hope that it will be able to improve model performance by making use of all available features.

A Note About Memory Usage

When running this particular classifier, the size advantage of the dataset was also a minor disadvantage, as trying to train the Random Forest using the full training dataset (69,999 observations) actually crashed R with the message:

> Error: vector memory exhausted (limit reached?) random forest

In the previous models, the size of the dataset resulted in slower processing time but space usage was not an issue; however Random Forest uses many unpruned (and therefore deep) decision trees thus needing the additional memory. As such, we have used a subset of the training data to generate the model, and we will see that it still performs better than a single tree.

```
> library(randomForest)
# get 20000 samples of df_train
> set.seed(42)
> rf.sample = sample(1:nrow(df_train), size=20000)

> rf.mod = randomForest(class ~ .,
```

```
                      data = df_train[rf.sample,],
                      importance = TRUE,
                      proximity = TRUE)
> print(rf.mod)
Call:
 randomForest(formula = class ~ ., data = df_train[rf.sample, ], importance = TRUE,
 proximity = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 2

        OOB estimate of  error rate: 2.46%
Confusion matrix:
       GALAXY  QSO STAR  class.error
GALAXY  11641  148   76 0.0188790560
QSO       267 3493    0 0.0710106383
STAR        2    0 4373 0.0004571429

> confusionMatrix(rf.pred, df_test$class, mode="everything")
Confusion Matrix and Statistics

          Reference
Prediction GALAXY   QSO  STAR
    GALAXY  17387   440     0
    QSO       267  5263     0
    STAR      147     0  6496

Overall Statistics

               Accuracy : 0.9715
                 95% CI : (0.9696, 0.9734)
    No Information Rate : 0.5934
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9496

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: GALAXY Class: QSO Class: STAR
Sensitivity                 0.9767     0.9228      1.0000
Specificity                 0.9639     0.9890      0.9937
Pos Pred Value              0.9753     0.9517      0.9779
Neg Pred Value              0.9660     0.9820      1.0000
```

Precision	0.9753	0.9517	0.9779
Recall	0.9767	0.9228	1.0000
F1	0.9760	0.9371	0.9888
Prevalence	0.5934	0.1901	0.2165
Detection Rate	0.5796	0.1754	0.2165
Detection Prevalence	0.5942	0.1843	0.2214
Balanced Accuracy	0.9703	0.9559	0.9969

The out-of-bag error rate is quite small at only 2.46%. The accuracy was 97.1%, compared to the single tree accuracy of 95%, and the F1 scores for each class are improved as well.

We can plot the performance of the model as a function between error and the number of trees used. The default parameter for ntree is 500 [10]. We see that performance plateaus, as due to the relatively small number of features the model quickly runs out of different combinations. The model output indicated that each split used 2 variables, so there are only $\binom{8}{2} = 28$ unique subsets of size 2.

```
> plot(rf.mod)
```

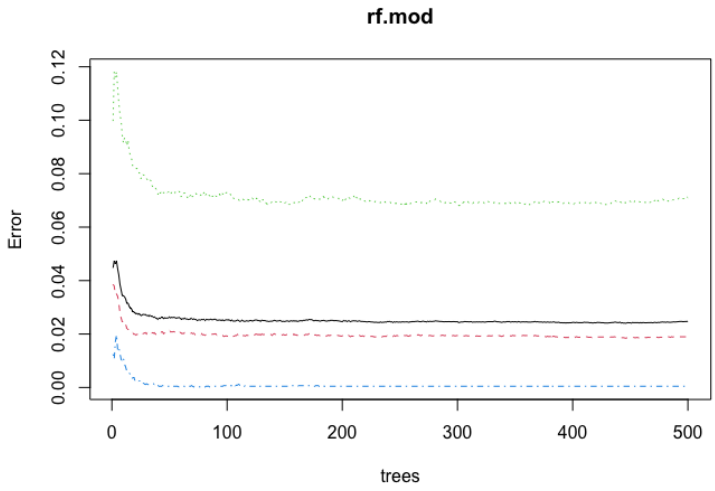

Figure 14: Random Forest performance

4.1.1 Variable Importance

The Random Forest classifier also generates two measures of importance, accuracy-based and Gini-based importance. First is the mean decrease in accuracy which indicates the loss of accuracy when a given feature is omitted, and is also broken down by class. For example we can see again that in particular for the `STAR` class, the `redshift` feature is a strong indicator, with a large decrease in accuracy if `redshift` is not used. The other measure is the mean decrease in Gini importance, which is calculated each time that a node is split.[11]

```
# variable importance
> rf.mod$importance
          GALAXY         QSO         STAR MeanDecreaseAccuracy MeanDecreaseGini
alpha  0.004859475 0.001708658 0.0011288291          0.003450193         179.6720
```

```
delta     0.007090203 0.002556884 0.0005765994          0.004811962         179.7898
u         0.073862408 0.058914370 0.0332691975          0.062169095         739.5700
g         0.171992698 0.111296722 0.0485120410          0.133597495         892.7181
r         0.132275425 0.070431235 0.0460542690          0.101798750         547.1256
i         0.135428211 0.155116776 0.0308755634          0.116287680         766.7718
z         0.077218427 0.160861017 0.0369968268          0.084141696        1103.3230
redshift  0.373069327 0.441511256 0.7310603331          0.464162384        6888.3421
```

```
> varImpPlot(rf.mod,main='Variable Importance (Random Forest)')
```

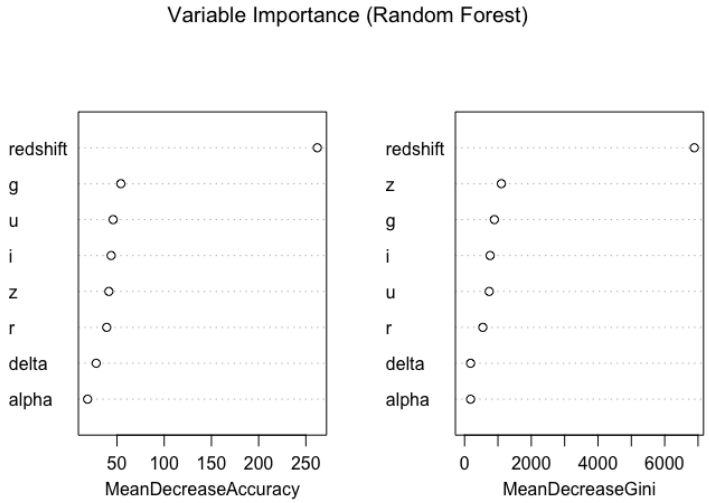

Figure 15: Random Forest variable importance

We can see that as before, `redshift` has by far the greatest importance in both measures. For the most part the rest of the features follow the same pattern, although interestingly the second most important feature as based on accuracy decrease is `g`, where as when based on Gini importance the second ranked feature is `z`. Despite this, the two secondary features still fall fairly far behind the measure for `redshift`.

4.2 Boosting

Next up is boosting, a technique that iteratively trains weak models, by modifying successive models to improve on previous ones. In particular, we will use AdaBoost which attempts to "adapt" by putting more weight on observations that were difficult to classify. Although AdaBoost is one of the older boosting algorithms, it is still quite effective.

```
# generate model
> ada.mod = boosting(class~., data=df_train)

# predict
> ada.pred = predict(ada.mod, df_test)

> confusionMatrix(ada.pred$confusion, mode="everything")
Confusion Matrix and Statistics

                Observed Class
Predicted Class GALAXY   QSO  STAR
          GALAXY  17471   454     0
          QSO       300  5249     0
          STAR       30     0  6496

Overall Statistics

               Accuracy : 0.9739
                 95% CI : (0.972, 0.9756)
    No Information Rate : 0.5934
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9536

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: GALAXY Class: QSO Class: STAR
Sensitivity                 0.9815     0.9204      1.0000
Specificity                 0.9628     0.9877      0.9987
Pos Pred Value              0.9747     0.9459      0.9954
Neg Pred Value              0.9727     0.9814      1.0000
Precision                   0.9747     0.9459      0.9954
Recall                      0.9815     0.9204      1.0000
F1                          0.9781     0.9330      0.9977
Prevalence                  0.5934     0.1901      0.2165
Detection Rate              0.5824     0.1750      0.2165
Detection Prevalence        0.5975     0.1850      0.2175
```

```
Balanced Accuracy          0.9721      0.9540      0.9994
```

So, AdaBoost performs about as well as Random Forest did, with an improved accuracy of 97.4% and similarly high F1 scores.

The adabag package also provides functions to calculate the error evolution of as the number of ensemble members increases (by default, boosting adds 100 members). We can plot this error for the train and test datasets to compare them.

```
> evol.train <-errorevol(ada.mod, newdata=df_train)
> evol.test <-errorevol(ada.mod, newdata=df_test)
> plot.errorevol(evol.test,evol.train)
```

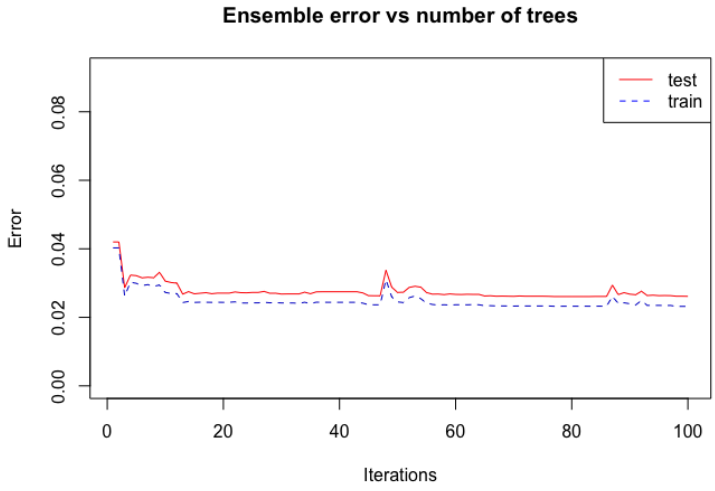

Figure 16: Adaboost evolution error

Though the test error is still quite low, it is slightly higher than training error across the board, which is to be expected. After $\tilde{1}0$ iterations, the error falls to its lowest point, after which there is no further improvement given more ensemble members. The spikes in error rate could simply be due to some quirks in the data.

4.2.1 With Cross Validation

The adabag package also provides cross validation functionality through boosting.cv. However, since cross validation is primarily useful when the dataset is small, as it allows each observation to to become part of the training set, we see as expected that the improvement is minimal at best given that we already have the benefit of a large number of training observations.

```
> ada.mod.cv = boosting.cv(class~., data=df_train, control=rpart.control(cp=0.01),
        mfinal=50, v=10, par=TRUE)
> confusionMatrix(ada.mod.cv$confusion, mode="everything")
Confusion Matrix and Statistics

                Observed Class
Predicted Class GALAXY   QSO  STAR
         GALAXY  40911  1103     5
```

```
          QSO        609 12152     0
          STAR       124     3 15092

Overall Statistics

               Accuracy : 0.9737
                 95% CI : (0.9724, 0.9748)
    No Information Rate : 0.5949
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9531

 Mcnemar's Test P-Value : < 2.2e-16

Statistics by Class:

                     Class: GALAXY Class: QSO Class: STAR
Sensitivity                 0.9824     0.9166      0.9997
Specificity                 0.9609     0.9893      0.9977
Pos Pred Value              0.9736     0.9523      0.9917
Neg Pred Value              0.9738     0.9807      0.9999
Precision                   0.9736     0.9523      0.9917
Recall                      0.9824     0.9166      0.9997
F1                          0.9780     0.9341      0.9956
Prevalence                  0.5949     0.1894      0.2157
Detection Rate              0.5845     0.1736      0.2156
Detection Prevalence        0.6003     0.1823      0.2174
Balanced Accuracy           0.9717     0.9529      0.9987
```

4.3 Bagging

Lastly, we will implement bagging, or "Bootstrap Aggregation", again using the adabag package. Bagging is primarily a variance reduction technique, which uses bootstrapped sampling, or sampling with replacement, in order to generate more data on which to train.

Since the dataset is already quite large, we may not expect a significant improvement as far as accuracy goes, however we want to test this theory and confirm whether variance in the previous models is in fact already low.

Since each bagging tree is modeled independently, this method is parallelizable. We can set the par argument to TRUE to tell the function to run with multiple cores using doParallel in the background.

```
# create model
> bag.mod <- bagging(class~., data=df_train, mfinal = 100, control=rpart.control(),
    par=TRUE)

# predict with test dataset
> bag.pred <- predict.bagging(bag.mod, newdata=df_test)

> confusionMatrix(bag.pred$confusion, mode="everything")
Confusion Matrix and Statistics

                Observed Class
Predicted Class GALAXY   QSO  STAR
         GALAXY  17265   794     0
         QSO       380  4909     0
         STAR      156     0  6496

Overall Statistics

               Accuracy : 0.9557
                 95% CI : (0.9533, 0.958)
    No Information Rate : 0.5934
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.921

 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: GALAXY Class: QSO Class: STAR
Sensitivity                 0.9699     0.8608      1.0000
Specificity                 0.9349     0.9844      0.9934
Pos Pred Value              0.9560     0.9282      0.9765
Neg Pred Value              0.9551     0.9679      1.0000
```

Precision	0.9560	0.9282	0.9765
Recall	0.9699	0.8608	1.0000
F1	0.9629	0.8932	0.9881
Prevalence	0.5934	0.1901	0.2165
Detection Rate	0.5755	0.1636	0.2165
Detection Prevalence	0.6020	0.1763	0.2217
Balanced Accuracy	0.9524	0.9226	0.9967

4.4 Summary

Below are comparisons for accuracy and f1 scores across all individual and ensemble classifiers tested. As predicted, the ensemble techniques with the ability to reduce bias were successful in increasing model performance compared to the individual classifiers, whereas bagging and cross validation, which are variance-reduction mechanisms, were not. This follows as the large size of the dataset already lends itself to variance reduction. The AdaBoost results were the most capable of generalizing, closely followed by Random Forest.

Along the same lines, within each of these techniques we saw that the improvement in performance was halted at a certain point, most likely we have reached the limit of irreducible error (inherent bias which cannot be removed by modeling) in the dataset.

	Decision Tree	KNN		Naive Bayes		SVM
		k=3; all features	k=9; Redshift, g	All features	redshift, g	
Accuracy	95.56	92.87	96.38	91.87	94.5	96.75
F1: Galaxy	0.9628	0.9419	0.9697	0.9328	0.9562	0.9728
F1: Quasar	0.8933	0.9286	0.9210	0.8135	0.8599	0.9365
F1: Star	0.9881	0.8926	0.9839	0.9786	0.9811	0.9792

Figure 17: Accuracy and F1 Score for Individual Classifiers

	Random Forest	AdaBoost		Bagging
		Without CV	With CV	
Accuracy	97.15	97.39	97.37	95.57
F1: Galaxy	0.9760	0.9781	0.9780	0.9629
F1: Quasar	0.9371	0.9330	0.9341	0.8932
F1: Star	0.9888	0.9977	0.9956	0.9881

Figure 18: Accuracy and F1 Scores for Ensemble Methods

5 References

[1] The Sloan Digital Sky Survey. SkyServer: About SDSS. http://skyserver.sdss.org/dr1/en/sdss/. Accessed: 2022-02-15.

[2] fedesoriano (Kaggle user). Stellar Classification Dataset - SDSS17. https://www.kaggle.com/fedesoriano/stellar-classification-dataset-sdss17. Accessed: 2022-02-15.

[3] The Sloan Digital Sky Survey. Glossary of SDSS-III Terminology. https://www.sdss3.org/dr8/glossary.php. Accessed: 2022-03-09.

[4] PhD Paul Julian II. Too much outside the box - Outliers and Boxplots. https://swampthingecology.org/blog/too-much-outside-the-box-outliers-and-boxplots/. Accessed: 2022-04/01.

[5] Geeks for Geeks. Normalization vs Standardization. https://www.geeksforgeeks.org/normalization-vs-standardization/. Accessed: 2022-03-09.

[6] Joos Korstanje. The F1 Score. https://towardsdatascience.com/the-f1-score-bec2bbc38aa6 note = Accessed: 2022-04-20.

[7] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning with Applications in R.* Springer, 2nd edition edition, 2021.

[8] Amey Band. How to find the optimal value of K in KNN? https://towardsdatascience.com/how-to-find-the-optimal-value-of-k-in-knn-35d936e554eb. Accessed: 2022-04-15.

[9] Matias Alfaro, Esteban; Gamez and Noelia; with contributions from Li Guo Garcia. *Applies Multiclass AdaBoost.M1, SAMME and Bagging*, January 2018. Accessed: 2022-04-15.

[10] RDocumentation. randomForest: Classification and Regression with Random Forest. https://www.rdocumentation.org/packages/randomForest/versions/4.7-1/topics/randomForest. Accessed: 2022-04-20.

[11] Jake Hoare. How is Variable Importance Calculated for a Random Forest? https://www.displayr.com/how-is-variable-importance-calculated-for-a-random-forest/ note = Accessed: 2022-05-10.