Wine Quality Prediction

Introduction

In this project, I'm diving into a data set with information on various wines and their characteristics. With some exploratory data analysis and statistical inference models, I will find important variables that contribute to wine quality and then predict the quality with a health dose of machine learning models.

Load Packages and Data

```
library(tidyverse)
```

```
Warning: package 'tidyr' was built under R version
4.2.3

Warning: package 'dplyr' was built under R version
4.2.3

── Attaching core tidyverse packages
──────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.4     ✔ readr     2.1.4
✔ forcats   1.0.0     ✔ stringr   1.5.0
✔ ggplot2   3.5.2     ✔ tibble    3.2.1
✔ lubridate 1.9.2     ✔ tidyr     1.3.1
✔ purrr     1.0.4
── Conflicts
──────────────────────────────────────────────
```

```
tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-
lib.org/>) to force all conflicts to become errors
```

```
library(corrplot)
```

```
corrplot 0.92 loaded
```

```
library(scales)
```

```

Attaching package: 'scales'

The following object is masked from
'package:purrr':

    discard

The following object is masked from
'package:readr':

    col_factor
```

```
library(tidymodels)
```

```
── Attaching packages
──────────────────────────────────────── tidymodels
1.3.0 ──
✔ broom        1.0.8     ✔ rsample      1.3.0
✔ dials        1.4.0     ✔ tune         1.3.0
✔ infer        1.0.8     ✔ workflows    1.2.0
✔ modeldata    1.4.0     ✔ workflowsets 1.1.0
✔ parsnip      1.3.1     ✔ yardstick    1.3.2
```

```
✔ recipes      1.3.0

Warning: package 'workflowsets' was built under R
version 4.2.3

── Conflicts
──────────────────────────────────────────────────
tidymodels_conflicts() ──
✖ scales::discard() masks purrr::discard()
✖ dplyr::filter()   masks stats::filter()
✖ recipes::fixed()  masks stringr::fixed()
✖ dplyr::lag()      masks stats::lag()
✖ yardstick::spec() masks readr::spec()
✖ recipes::step()   masks stats::step()
```

```
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
```

```
library(themis)
library(vip)
```

```

Attaching package: 'vip'

The following object is masked from
```

```
'package:utils':

    vi
```

```
data <- read_csv("WineQT.csv")
```

```
Rows: 1143 Columns: 13
── Column specification
────────────────────────────────────────────────────────
─────
Delimiter: ","
dbl (13): fixed acidity, volatile acidity, citric
acid, residual sugar, chlo...

ℹ Use `spec()` to retrieve the full column
specification for this data.
ℹ Specify the column types or set `show_col_types =
FALSE` to quiet this message.
```

EDA and Cleaning

The data comes from Kaggle in the form of a csv. It's important to get to know the data and see what its structure is. The function below gives me all the important information I need right now. I'm dealing with 1,143 records (wines), and 14 variables. Already, I can see I'll need to alter a few things when we get into cleaning and manipulation.

```
str(data)
```

```
spc_tbl_ [1,143 × 13] (S3:
spec_tbl_df/tbl_df/tbl/data.frame)
 $ fixed acidity       : num [1:1143] 7.4 7.8 7.8
```

```
11.2 7.4 7.4 7.9 7.3 7.8 6.7 ...
 $ volatile acidity    : num [1:1143] 0.7 0.88 0.76
0.28 0.7 0.66 0.6 0.65 0.58 0.58 ...
 $ citric acid         : num [1:1143] 0 0 0.04 0.56
0 0 0.06 0 0.02 0.08 ...
 $ residual sugar      : num [1:1143] 1.9 2.6 2.3
1.9 1.9 1.8 1.6 1.2 2 1.8 ...
 $ chlorides           : num [1:1143] 0.076 0.098
0.092 0.075 0.076 0.075 0.069 0.065 0.073 0.097 ...
 $ free sulfur dioxide : num [1:1143] 11 25 15 17
11 13 15 15 9 15 ...
 $ total sulfur dioxide: num [1:1143] 34 67 54 60
34 40 59 21 18 65 ...
 $ density             : num [1:1143] 0.998 0.997
0.997 0.998 0.998 ...
 $ pH                  : num [1:1143] 3.51 3.2 3.26
3.16 3.51 3.51 3.3 3.39 3.36 3.28 ...
 $ sulphates           : num [1:1143] 0.56 0.68
0.65 0.58 0.56 0.56 0.46 0.47 0.57 0.54 ...
 $ alcohol             : num [1:1143] 9.4 9.8 9.8
9.8 9.4 9.4 9.4 10 9.5 9.2 ...
 $ quality             : num [1:1143] 5 5 5 6 5 5 5
7 7 5 ...
 $ Id                  : num [1:1143] 0 1 2 3 4 5 6
7 8 10 ...
 - attr(*, "spec")=
  .. cols(
  ..   `fixed acidity` = col_double(),
  ..   `volatile acidity` = col_double(),
  ..   `citric acid` = col_double(),
  ..   `residual sugar` = col_double(),
  ..   chlorides = col_double(),
  ..   `free sulfur dioxide` = col_double(),
  ..   `total sulfur dioxide` = col_double(),
  ..   density = col_double(),
  ..   pH = col_double(),
  ..   sulphates = col_double(),
```

```
  ..   alcohol = col_double(),
  ..   quality = col_double(),
  ..   Id = col_double()
  .. )
 - attr(*, "problems")=<externalptr>
```

I also like to use the summary function a bird's eye view of all the variables. It looks most of the wines are a 5-6 rating with a max rating of 8 and a minimum of 3. Wow, with a rating system out of 10, there isn't even a 9! The rest of the summary statistics gives great into the range of all the variables, since they are all numeric.

```
summary(data)
```

```
 fixed acidity    volatile acidity  citric acid
residual sugar
 Min.   : 4.600   Min.   :0.1200   Min.   :0.0000
Min.   : 0.900
 1st Qu.: 7.100   1st Qu.:0.3925   1st Qu.:0.0900
1st Qu.: 1.900
 Median : 7.900   Median :0.5200   Median :0.2500
Median : 2.200
 Mean   : 8.311   Mean   :0.5313   Mean   :0.2684
Mean   : 2.532
 3rd Qu.: 9.100   3rd Qu.:0.6400   3rd Qu.:0.4200
3rd Qu.: 2.600
 Max.   :15.900   Max.   :1.5800   Max.   :1.0000
Max.   :15.500
   chlorides       free sulfur dioxide total sulfur
dioxide    density
 Min.   :0.01200   Min.   : 1.00       Min.   :
6.00       Min.   :0.9901
 1st Qu.:0.07000   1st Qu.: 7.00       1st Qu.:
21.00       1st Qu.:0.9956
```

```
 Median :0.07900   Median :13.00       Median :
37.00       Median :0.9967
 Mean   :0.08693   Mean   :15.62       Mean   :
45.91       Mean   :0.9967
 3rd Qu.:0.09000   3rd Qu.:21.00       3rd Qu.:
61.00       3rd Qu.:0.9978
 Max.   :0.61100   Max.   :68.00       Max.
:289.00       Max.   :1.0037
       pH          sulphates         alcohol
quality
 Min.   :2.740   Min.   :0.3300   Min.   : 8.40
Min.   :3.000
 1st Qu.:3.205   1st Qu.:0.5500   1st Qu.: 9.50
1st Qu.:5.000
 Median :3.310   Median :0.6200   Median :10.20
Median :6.000
 Mean   :3.311   Mean   :0.6577   Mean   :10.44
Mean   :5.657
 3rd Qu.:3.400   3rd Qu.:0.7300   3rd Qu.:11.10
3rd Qu.:6.000
 Max.   :4.010   Max.   :2.0000   Max.   :14.90
Max.   :8.000
       Id
 Min.   :   0
 1st Qu.: 411
 Median : 794
 Mean   : 805
 3rd Qu.:1210
 Max.   :1597
```

Before going any further, I want to do a few things. First, let's rename all the variables with a space, as best practice is to add a "_" instead.

```
data <- data %>%
  rename(fixed_acidity = `fixed acidity`,
```

```
          volatile_acidity = `volatile acidity`,
          citric_acid = `citric acid`,
          residual_sugar = `residual sugar`,
          free_sulfur_dioxide = `free sulfur dioxide`
          total_sulfur_dioxide = `total sulfur dioxi
```

I also want to add a new column called “quality_group” to act as a predictor variables. I think it would be easier to predict 3 columns instead the 10 in some cases of machine learning. The split will be, any wine with a rating 4 or below is considered low quality, any wine rated 5 to 6 is considered medium quality, and anything 7 or higher is high quality wine. There are arguments to be made for different group binning, but this is just how I chose to do it.

```
data <-data %>%
  mutate(quality_group = case_when(
    quality >= 0 & quality <= 4 ~ "Low",
    quality > 4 & quality <= 6 ~ "Medium",
    quality >= 7 ~ "High"
  ))

#Making the new variable, plus others into a factor
data$quality_group <- as.factor(data$quality_group)
data$quality <- as.factor(data$quality)
data$Id <- as.character(data$Id)

#Reorder variable for future visuals
data$quality_group <- factor(data$quality_group, lev
```

Let’s get a look at the count of our groups now. This variable is quite imbalanced and we will keep this in mind for later.

```
data %>%
  group_by(quality_group) %>%
  summarize(count = n())
```

```
# A tibble: 3 × 2
  quality_group count
  <fct>         <int>
1 Low              39
2 Medium          945
3 High            159
```

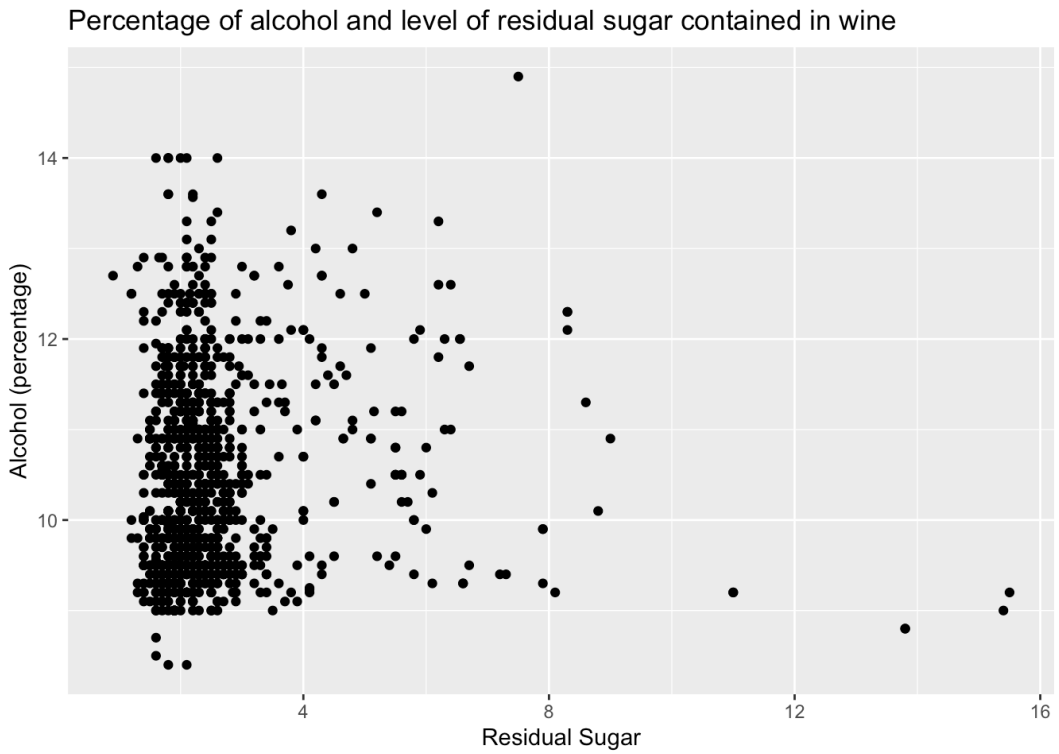
Grapes grown in cooler climates tend to have higher acidity, while warmer climates often result in lower acidity levels. Riper grapes also tend to have lower acidity

Visuals

With this data, box plots will do a lot to show what variables are important to making a quality wine.

I suspected that a high residual sugar may factor into the alcohol percentage, but this graph shows there is essentially no relationship. Clearly, a lot of the wines have a lower residual sugar and their alcohol percentage ranges from 6% to even 14%.

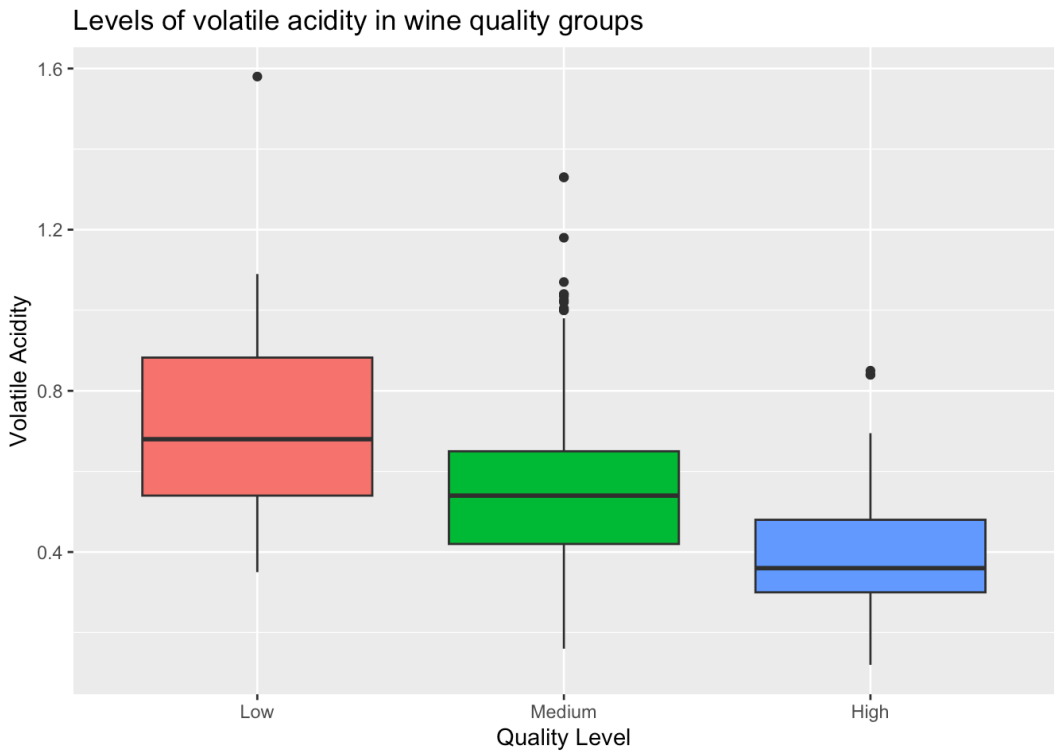
```
data %>%
  ggplot(aes(x = residual_sugar, y = alcohol))+
  geom_point()+
  labs(title = "Percentage of alcohol and level of r
       x = "Residual Sugar",
       y = "Alcohol (percentage)")
```



Let's look at volatile acidity (VA), a key attribute in wine. In small amounts, VA can add complexity to flavor and contribute to a wine's aroma profile. However, high VA levels are undesirable and can lead to off-flavors and bad aromas that can be detrimental to a wine's quality.

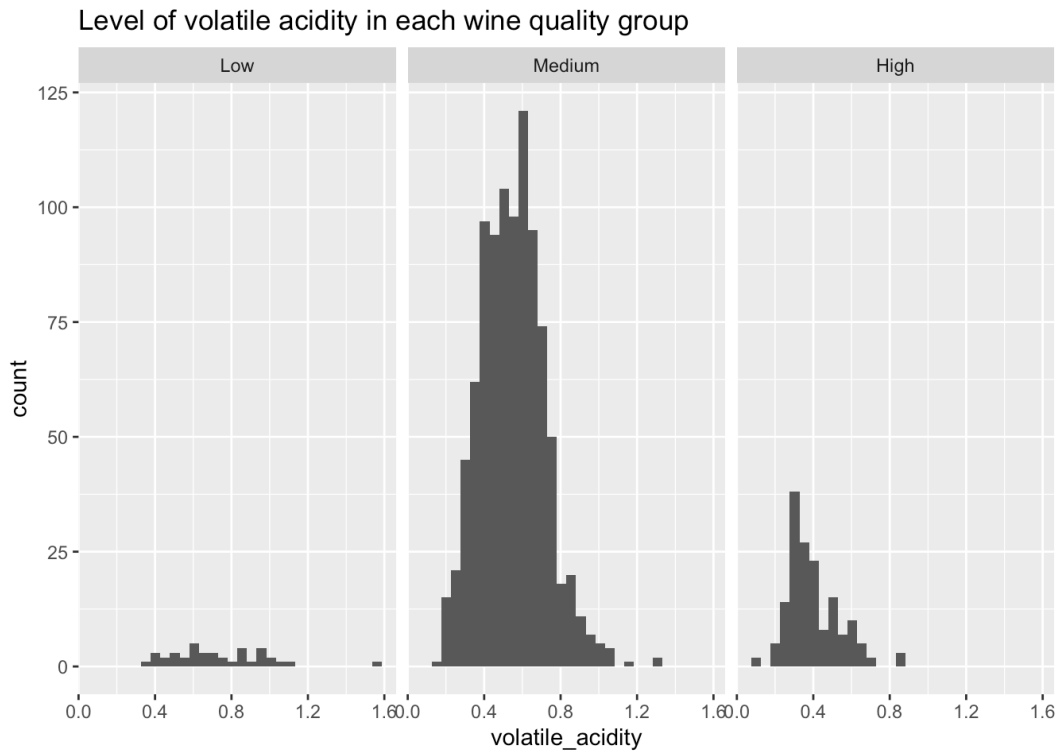
The box plot clearly shows the lower quality wines have higher levels of volatile acidity.

```
data %>%
  ggplot(aes(x = quality_group, y = volatile_acidity
  geom_boxplot(show.legend = FALSE)+
  labs(title = "Levels of volatile acidity in wine c
       x = "Quality Level",
       y = "Volatile Acidity")
```




```
data %>%
  ggplot(aes(x = volatile_acidity))+
  geom_histogram()+
  facet_wrap(~quality_group)+
  labs(title = "Level of volatile acidity in each wi
```

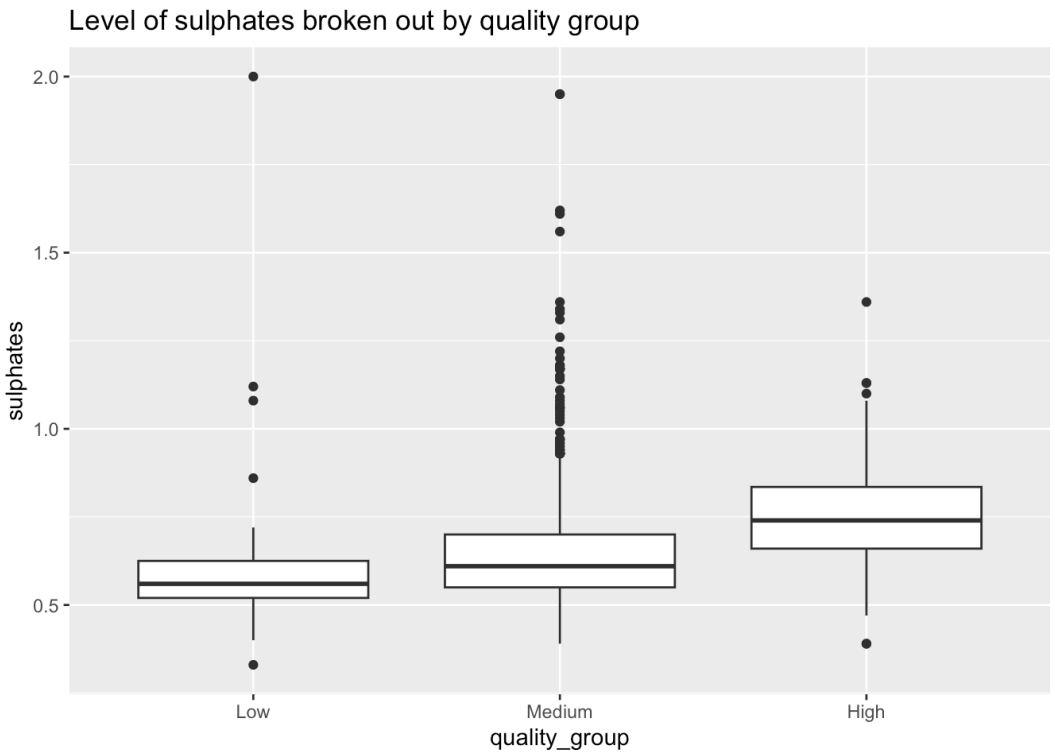
```
`stat_bin()` using `bins = 30`. Pick better value
with `binwidth`.
```



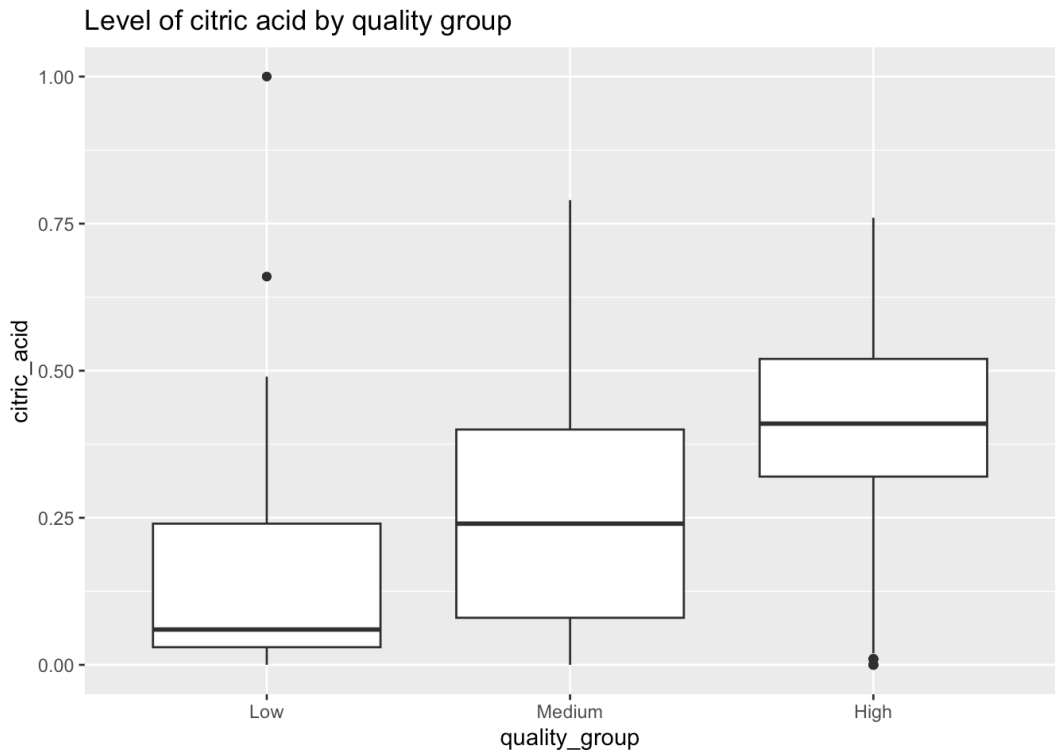

```
data %>%
  group_by(quality_group) %>%
  summarise(var(volatile_acidity))
```

```
# A tibble: 3 × 2
  quality_group `var(volatile_acidity)`
  <fct>                           <dbl>
1 Low                            0.0624
2 Medium                         0.0285
3 High                           0.0182
```

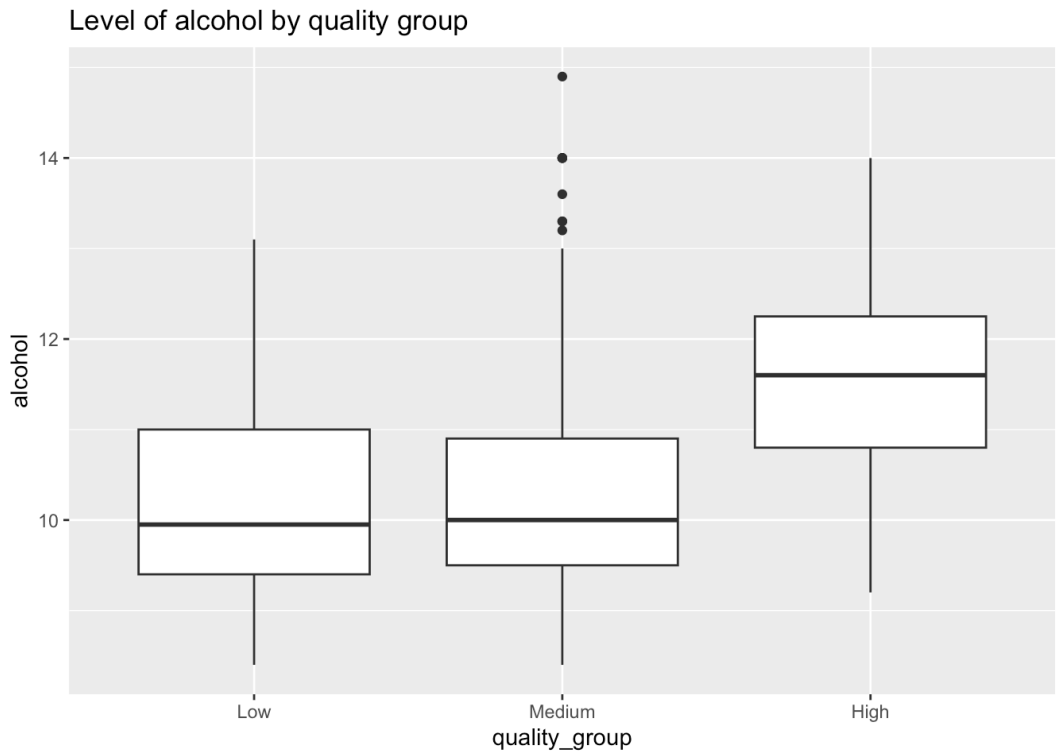
```
data %>%
  ggplot(aes(x = quality_group, y = sulphates))+
  geom_boxplot()+
  labs(title = "Level of sulphates broken out by qua
```




```
data %>%
  ggplot(aes(x = quality_group, y = citric_acid))+
  geom_boxplot()+
  labs(title = "Level of citric acid by quality grou
```



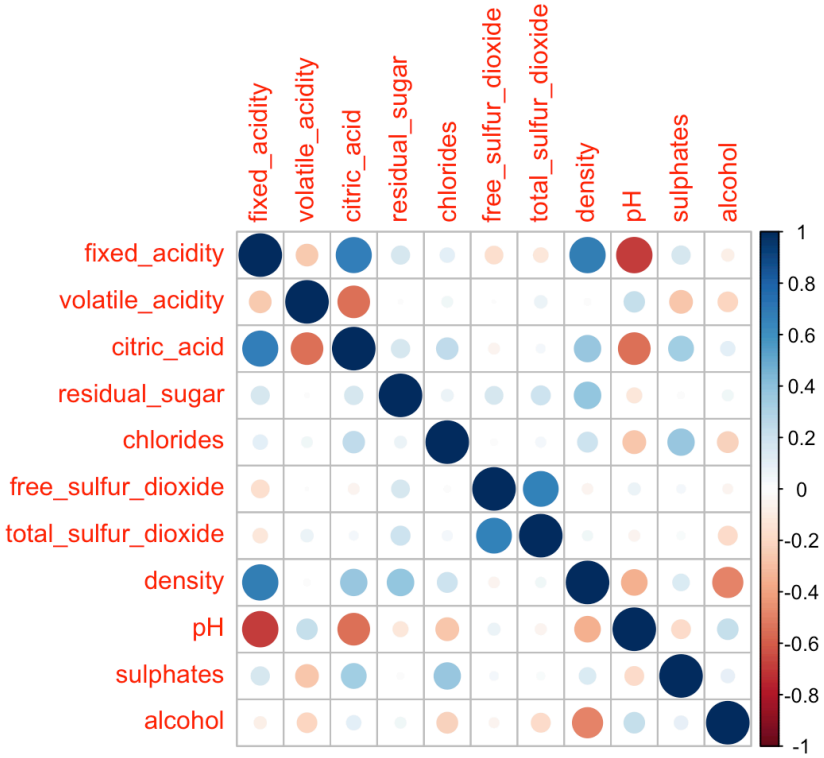

```
data %>%
  ggplot(aes(x = quality_group, y = alcohol))+
  geom_boxplot()+
  labs(title = "Level of alcohol by quality group")
```



Visually, there are clear differences in levels of citric acid, alcohol, sulphates, and volatile acidity when you separate them by quality groups. High quality wines have higher levels of all these variables besides volatile acidity, where levels are quite low.

Here is a classic correlation matrix with all the numeric variables.

```
data %>%
  select(1:11) %>%
  cor() %>%
  corrplot()
```

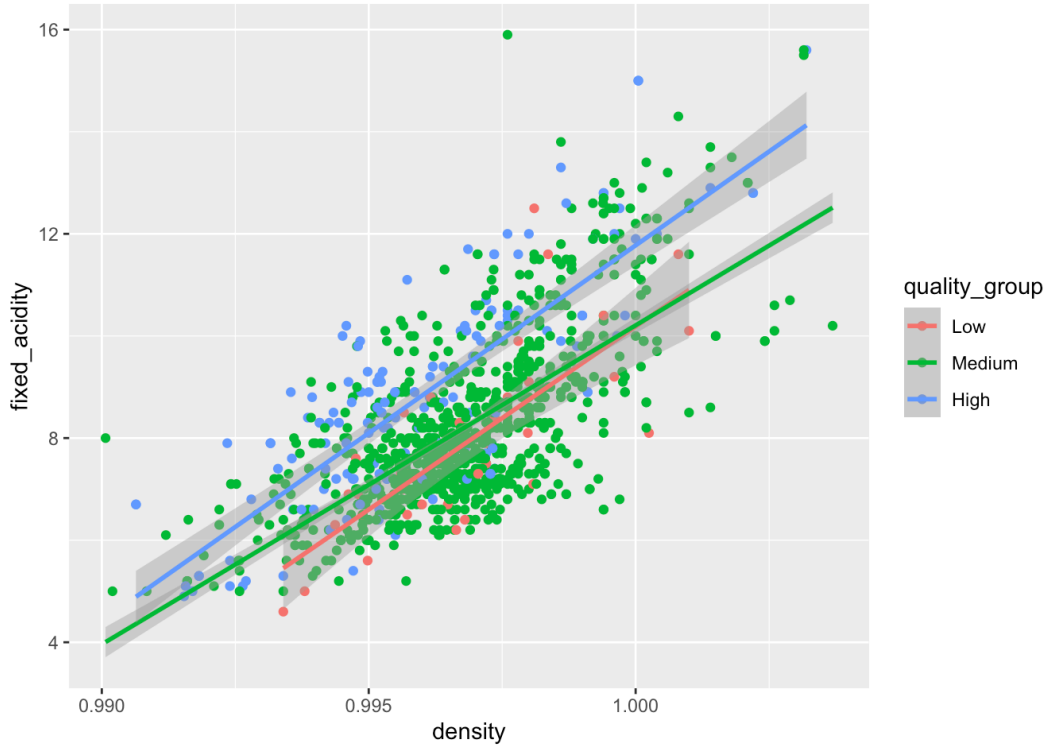


Key Takeaways: - Positive correlation between the two sulfur dioxide variables - pH and fixed acidity are negatively correlated - Positive correlation fixed acidity and citric acid, as well as fixed acidity and density

Let's look at these as individually graphed.

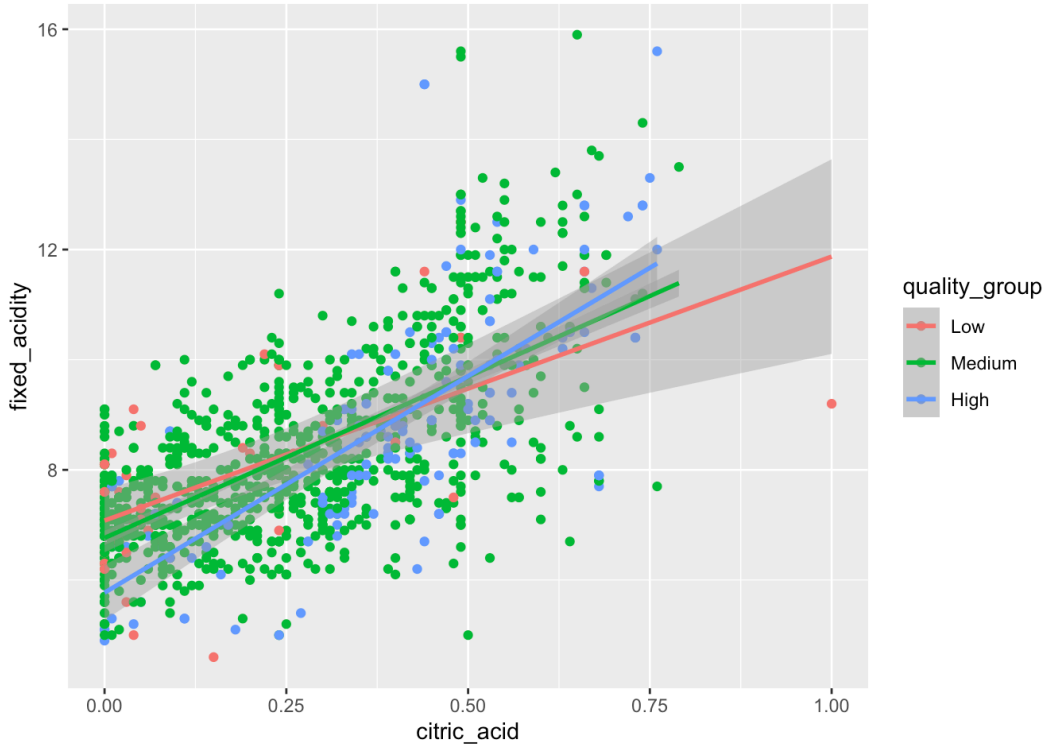
```
data %>%
  ggplot(aes(x = density, y = fixed_acidity, color =
  geom_point()+
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```



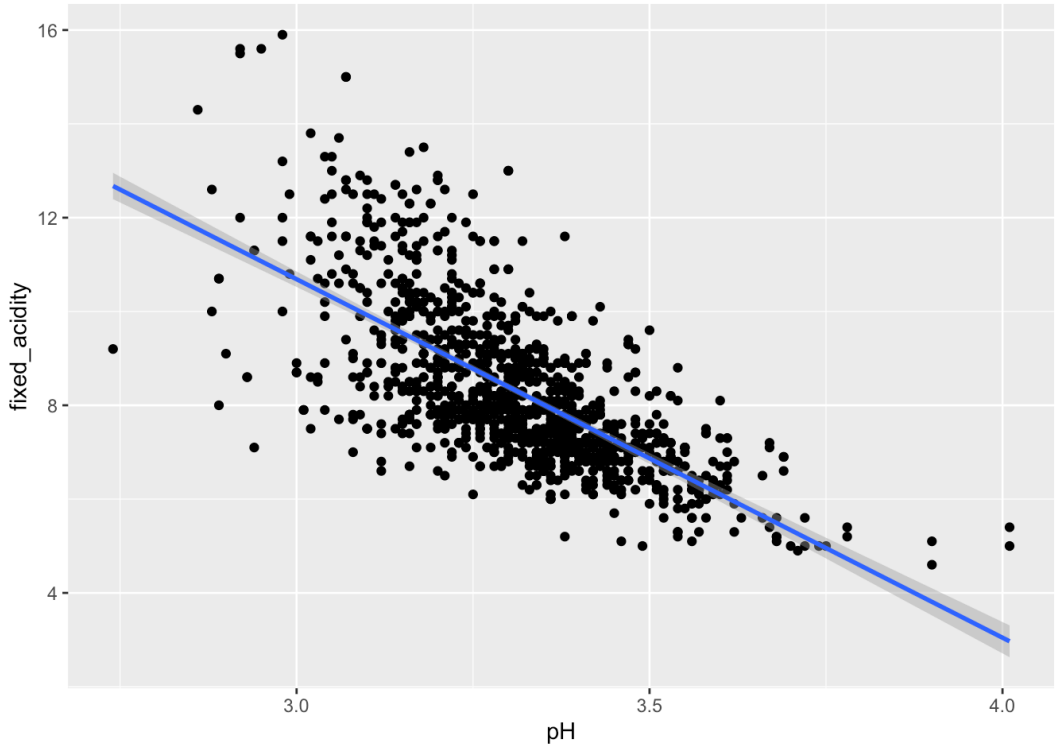

```
data %>%
  ggplot(aes(x = citric_acid, y = fixed_acidity,colo
  geom_point()+
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```



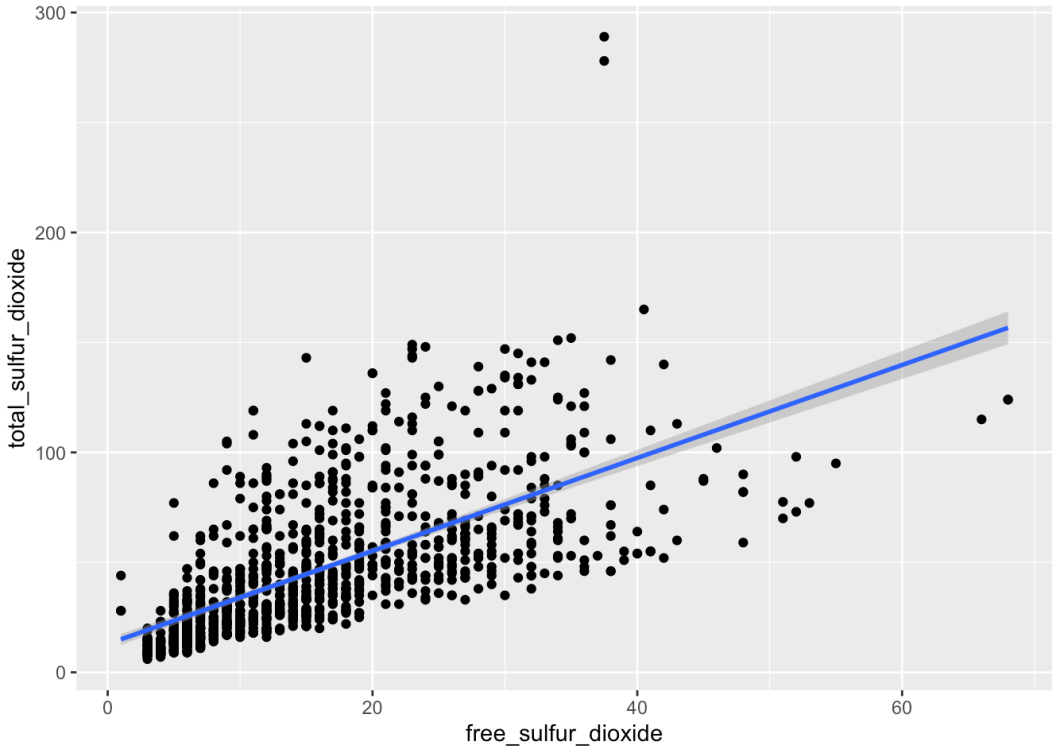

```
data %>%
  ggplot(aes(x = pH, y = fixed_acidity))+
  geom_point()+
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```



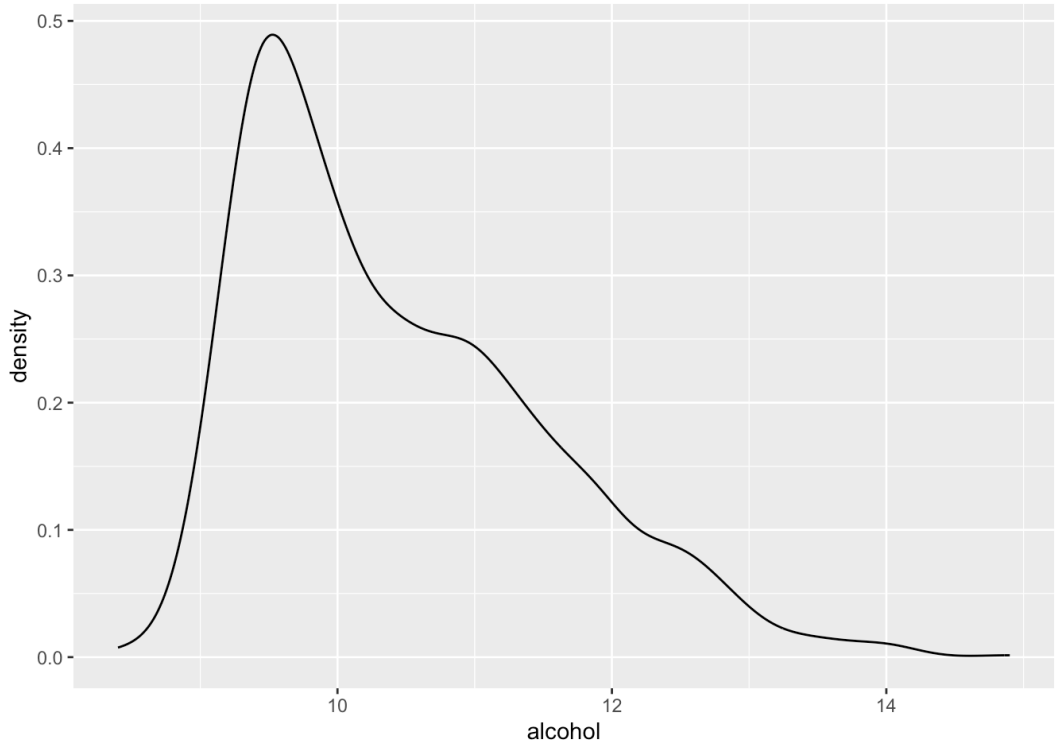

```
data %>%
  ggplot(aes(x = free_sulfur_dioxide, y = total_sul
  geom_point()+
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```



I just want to get a sense of what most of these wine alcohol levels are. It looks like most of the wines in this data set are slightly under 10% alcohol percentage.

```
data %>%
  ggplot(aes(x = alcohol))+
  geom_density()
```



I'm surprised, I did not think the level of alcohol would necessarily matter much when it came to the quality of wine. Boy, was I wrong. Higher quality wine has a higher percentage of alcohol, at least in this data set. It's interesting, even though most of the wines are under 10% abv, as per the density plot, we can observe the median abv of high quality wine is closer to 12%.

Statistical Analysis

ANOVA tests are great for this data set because I'm looking at the difference in means for multiple categorical variables. I'm not going to dive too deep here because the primary focus is model building and the data set is quite small so this is all I need to see that at least these variables are quite significant.

```
mod <- aov(volatile_acidity ~ quality_group, data =
```

```r
mod2 <- aov(sulphates ~ quality_group, data = data)

mod3 <- aov(alcohol ~ quality_group, data = data)

summary(mod)
```

```
               Df Sum Sq Mean Sq F value Pr(>F)
quality_group    2   4.69  2.3452   83.14 <2e-16
***
Residuals     1140  32.16  0.0282
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1
```

```r
summary(mod2)
```

```
               Df Sum Sq Mean Sq F value   Pr(>F)
quality_group    2   1.45  0.7248   26.06 8.58e-12
***
Residuals     1140  31.71  0.0278
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1
```

```r
summary(mod3)
```

```
               Df Sum Sq Mean Sq F value Pr(>F)
quality_group    2  218.3  109.15   111.2 <2e-16
***
Residuals     1140 1119.2    0.98
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05
'.' 0.1 ' ' 1
```

```
TukeyHSD(mod)
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = volatile_acidity ~
quality_group, data = data)

$quality_group
                   diff        lwr        upr p adj
Medium-Low  -0.1843740 -0.2487796 -0.1199685     0
High-Low    -0.3350701 -0.4055031 -0.2646372     0
High-Medium -0.1506961 -0.1844827 -0.1169095     0
```

Models Using TidyModels

Here's the main focus of the project, implementing machine learning models to predict quality of wine group given the set of variables.

Data Split

Now I need to split the data into my testing set and training set. Below I get rid of the Id and quality columns since quality group is built from quality and Id is not necessary. Also, you always want to set a seed to make sure the results are reproducible.

```
df <- data %>%
  select(-Id,-quality)

set.seed(222)
```

```
split <- initial_split(df,strata = quality_group) #s
train <- training(split)
test <- testing(split)

#Get a look at the break down of the split
split
```

```
<Training/Testing/Total>
<856/287/1143>
```

The usemodels package is nice to pair with tidymodels because it shows me a bare skeleton of the code I may want to use for the modeling process. It outputs boiler plate code, providing the structure for a workflow. I've never used it before, so I am just trying it out for random forest using the ranger engine.

```
library(usemodels)

use_ranger(quality_group ~., data = train)
```

```
ranger_recipe <-
  recipe(formula = quality_group ~ ., data = train)

ranger_spec <-
  rand_forest(mtry = tune(), min_n = tune(), trees
= 1000) %>%
  set_mode("classification") %>%
  set_engine("ranger")

ranger_workflow <-
  workflow() %>%
  add_recipe(ranger_recipe) %>%
  add_model(ranger_spec)
```

```
set.seed(16354)
ranger_tune <-
  tune_grid(ranger_workflow, resamples = stop("add
your rsample object"), grid = stop("add number of
candidate points"))
```

Recipe Creation

Here I'm creating the recipe we will use for any models going forward. This is the preprocessing step to get our data model ready. See notes in code for some more detail (also for myself in the future). The recipe, workflow, etc. setup is standard for tidymodels.

The preprocessing step is extremely important in model building and tidymodels makes it simple and effictive.

```
tidy_rec <- recipe(quality_group ~.,data = train) %>
  step_center(all_predictors(),-all_nominal()) %>%
  step_scale(all_predictors(),-all_nominal()) %>%
  step_impute_knn(volatile_acidity) %>% #fills any M
  step_range(all_predictors(),-all_nominal(),min = 0
  step_corr(all_numeric_predictors(),threshold = .8)
  step_dummy(all_nominal_predictors(), one_hot = T)
  #step_dummy(all_nominal(),-all_outcomes(), one_hot
#all_nominal_predictors instead for step_dummy

#See what the recipe did on the training data
prep <- prep(tidy_rec)

juiced <- juice(prep)
```

Create Model Specifications

Since this is a classification problem, I'll want to try two different model types and see which performs the best. I'll be using random forest and xgboost gradient boosting. The great thing about tidymodels is that it makes tuning hyperparameters easy to do with a few lines of code.

A useful tip for myself and R Studio users reading this, if you go to the "Addins" drop down and select "Generate parsnip model specifications", it will give me an interface to help figure out what type of engine to use. It's extremely useful.

Here is where I'll set the number of trees, say that I'm doing a classification model, etc.

XGBoost performs best when all the hyperparameters are tuned and trained at different values, then choose which one did best.

```
#Random Forest

rand_spec <- rand_forest(
  mtry = tune(),
  trees = 2000,
  min_n = tune()
) %>%
  set_mode("classification") %>% #Set to either clas
  set_engine("randomForest")#Set the engine means ju


#XGBoost

boost_spec <- boost_tree(
  trees = 500,
  tree_depth = tune(), min_n = tune(), loss_reductio
  learn_rate = tune(),
```

```
  sample_size = tune(), mtry = tune()
) %>%
  set_engine("xgboost") %>%
  set_mode("classification")
```

For xgboost. Here are all the possible hyper parameters to train the data on.What values are we going to try? Well let's do some tuning. I could use grid_regular here or max entropy, but I'll choose grid latin hypercube.

To tune I'll enter in all the parameters I said I wanted to tune in the above function.

```
xgb_grid <- grid_latin_hypercube(
  tree_depth(),
  min_n(),
  loss_reduction(),
  sample_size = sample_prop(),#needs to be proportic
  finalize(mtry(),train),#Since I do not know the mt
  learn_rate(),
  size = 25 #Can always change this to train more th
)
```

```
Warning: `grid_latin_hypercube()` was deprecated in
dials 1.3.0.
ℹ Please use `grid_space_filling()` instead.
```

```
xgb_grid
```

```
# A tibble: 25 × 6
   tree_depth min_n loss_reduction sample_size
mtry learn_rate
        <int> <int>          <dbl>       <dbl>
```

```
<int>      <dbl>
 1         11    31   0.00000642         0.630
3   5.40e- 9
 2          7    38   0.000000392        0.653
5   3.14e- 8
 3          4     7   0.0000199          0.731
12   2.88e- 3
 4          7    12   0.0000000341       0.336
4   1.50e- 7
 5         14    20   5.24               0.975
3   2.05e- 9
 6         11     2   0.768              0.841
4   4.28e- 6
 7         13    34  24.6                0.541
11   1.82e-10
 8          6    17   0.0000724          0.404
9   4.98e- 4
 9          6    14   0.00235            0.692
6   3.24e-10
10         12    40   0.000197           0.907
7   7.02e- 8
# ℹ 15 more rows
```

Now out of all these 25 model combinations, we will train the model and see what performs best on the data.

Creating Cross Validation Resamples

Here I'm using k folds cross validation for the resampling method. Further down I implement the bootstrap method.

```
set.seed(555)

#Strata would be what we're predicting and v is k
cv <- vfold_cv(train, v = 10, strata = quality_grou
```

Running Workflows to Check Models

Workflows are a tidymodels feature that just makes it easier to move the models around to try different tuning parameters, etc.

```
#Random forest workflow
rand_wf <- workflow() %>%
  add_recipe(tidy_rec) %>%
  add_model(rand_spec)

#Xgboost workflow
boost_wf <- workflow() %>%
  add_formula(quality_group ~.) %>%
  add_model(boost_spec)
```

Tuning and Training Both Models

Now we need some data to tune on. Here I'm using cross validation.

Let's tune the models and see what parameters work the best!

```
set.seed(45632)

#random forest
rand_tune <- tune_grid(
  rand_wf,
  resamples = cv,
  grid = 10
)
```

```
i Creating pre-processing data to finalize unknown
parameter: mtry
```

```
#xgboost tuning process
registerDoParallel()

set.seed(222)

xgb_res <- tune_grid(
  boost_wf,
  resamples = cv,
  grid = xgb_grid,
  control = control_grid(save_pred = TRUE)
)
```

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

```
xgb_res
```

```
# Tuning results
# 10-fold cross-validation using stratification
# A tibble: 10 × 5
   splits           id     .metrics
.notes           .predictions
   <list>           <chr>  <list>
<list>           <list>
 1 <split [769/87]> Fold01 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
 2 <split [769/87]> Fold02 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
 3 <split [769/87]> Fold03 <tibble [75 × 10]>
```

```
<tibble [0 × 3]> <tibble>
 4 <split [771/85]> Fold04 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
 5 <split [771/85]> Fold05 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
 6 <split [771/85]> Fold06 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
 7 <split [771/85]> Fold07 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
 8 <split [771/85]> Fold08 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
 9 <split [771/85]> Fold09 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
10 <split [771/85]> Fold10 <tibble [75 × 10]>
<tibble [0 × 3]> <tibble>
```

Models are done, tuned, and ready. Let's see what parameters were chosen.

Exploring Model Results

XGBoost

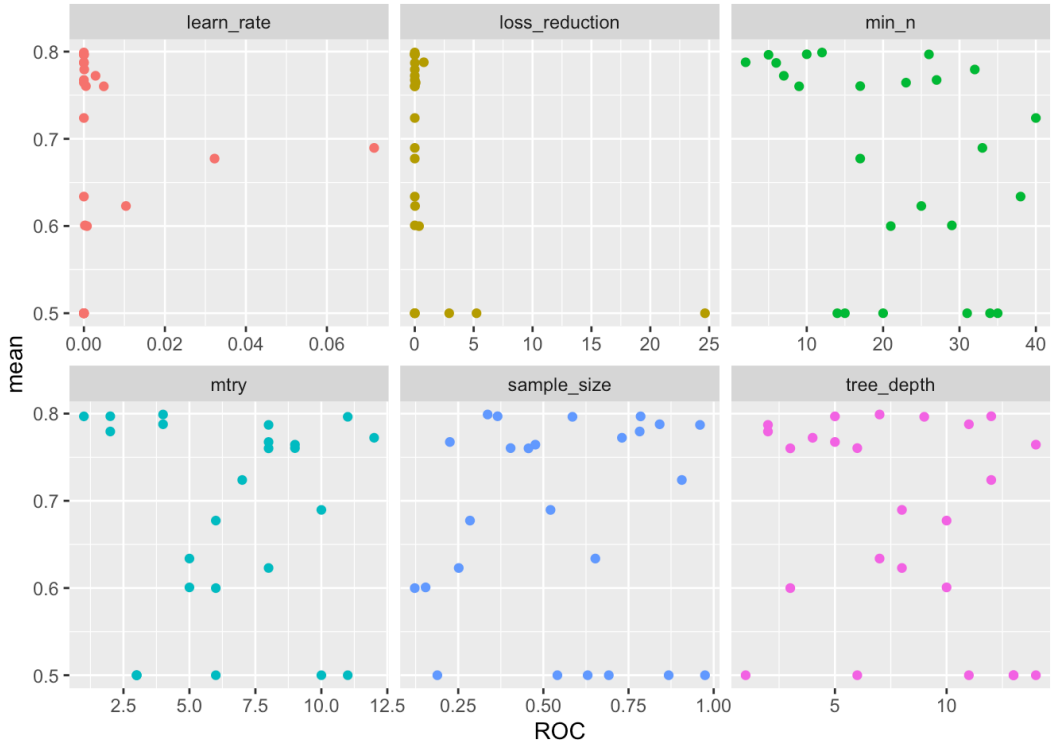
```
xgb_res %>%
  collect_metrics()
```

```
# A tibble: 75 × 12
    mtry min_n tree_depth    learn_rate
loss_reduction sample_size .metric
   <int> <int>      <int>         <dbl>
<dbl>       <dbl> <chr>
 1     3    31         11 0.00000000540
0.00000642         0.630 accuracy
 2     3    31         11 0.00000000540
0.00000642         0.630 brier_class
```

```
 3     3    31          11 0.00000000540
0.00000642         0.630 roc_auc
 4     5    38           7 0.0000000314
0.000000392        0.653 accuracy
 5     5    38           7 0.0000000314
0.000000392        0.653 brier_class
 6     5    38           7 0.0000000314
0.000000392        0.653 roc_auc
 7    12     7           4 0.00288         0.0000199
0.731 accuracy
 8    12     7           4 0.00288         0.0000199
0.731 brier_class
 9    12     7           4 0.00288         0.0000199
0.731 roc_auc
10     4    12           7 0.000000150
0.0000000341       0.336 accuracy
# ℹ 65 more rows
# ℹ 5 more variables: .estimator <chr>, mean <dbl>,
n <int>, std_err <dbl>,
#   .config <chr>
```

Let's see a plot of ROC for all the models and how they performed with the various different metrics.

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




```
#Lets see the five best models
xgb_res %>%
show_best(metric = "roc_auc")
```

```
# A tibble: 5 × 12
   mtry min_n tree_depth  learn_rate loss_reduction
sample_size .metric
  <int> <int>      <int>       <dbl>          <dbl>
<dbl> <chr>
1     4    12          7 0.000000150       3.41e- 8
0.336 roc_auc
2     2    10         12 0.00000633        4.27e-10
0.366 roc_auc
3     1    26          5 0.000000348       3.44e- 6
0.785 roc_auc
4    11     5          9 0.0000300         1.52e- 8
0.585 roc_auc
5     4     2         11 0.00000428        7.68e- 1
0.841 roc_auc
```

```
# ℹ 5 more variables: .estimator <chr>, mean <dbl>,
n <int>, std_err <dbl>,
#   .config <chr>
```

```
#Now lets select the best model
best_auc_boost <- select_best(xgb_res, metric = "roc

best_auc_boost
```

```
# A tibble: 1 × 7
   mtry min_n tree_depth  learn_rate loss_reduction
sample_size .config
  <int> <int>      <int>       <dbl>          <dbl>
<dbl> <chr>
1     4    12          7 0.000000150   0.0000000341
0.336 Preprocessor1_M…
```

```
#Now finalize the model and get it ready to use on t
final_xgb <- finalize_workflow(boost_wf, best_auc_bo

final_xgb
```

```
══ Workflow
═══════════════════════════════════════════════════
═══════════════════
Preprocessor: Formula
Model: boost_tree()

── Preprocessor
─────────────────────────────────────────────────────
─────────────
quality_group ~ .

── Model
```

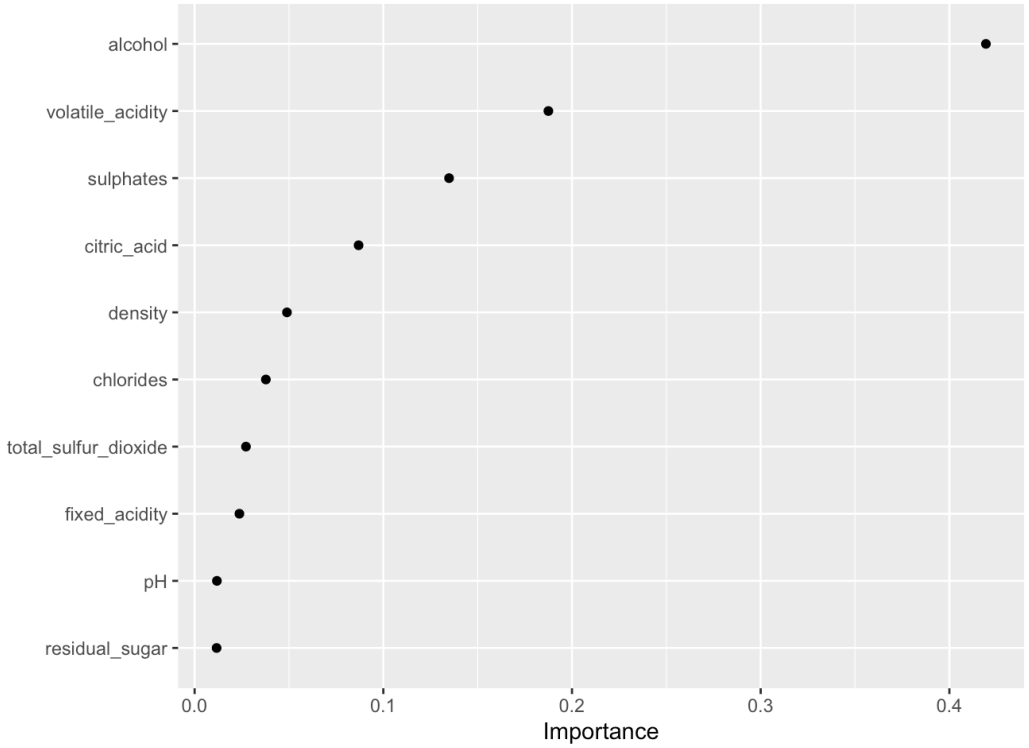
```
Boosted Tree Model Specification (classification)

Main Arguments:
  mtry = 4
  trees = 500
  min_n = 12
  tree_depth = 7
  learn_rate = 1.5031863803415e-07
  loss_reduction = 3.40616832584084e-08
  sample_size = 0.336314659048803

Computational engine: xgboost
```

Great! The xgboost model is done and in the Main Arguments section, it shows the best tuned parameters.

I want to see what the gradient boosting model says the most important variables are when predicting the quality group for the wine.

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

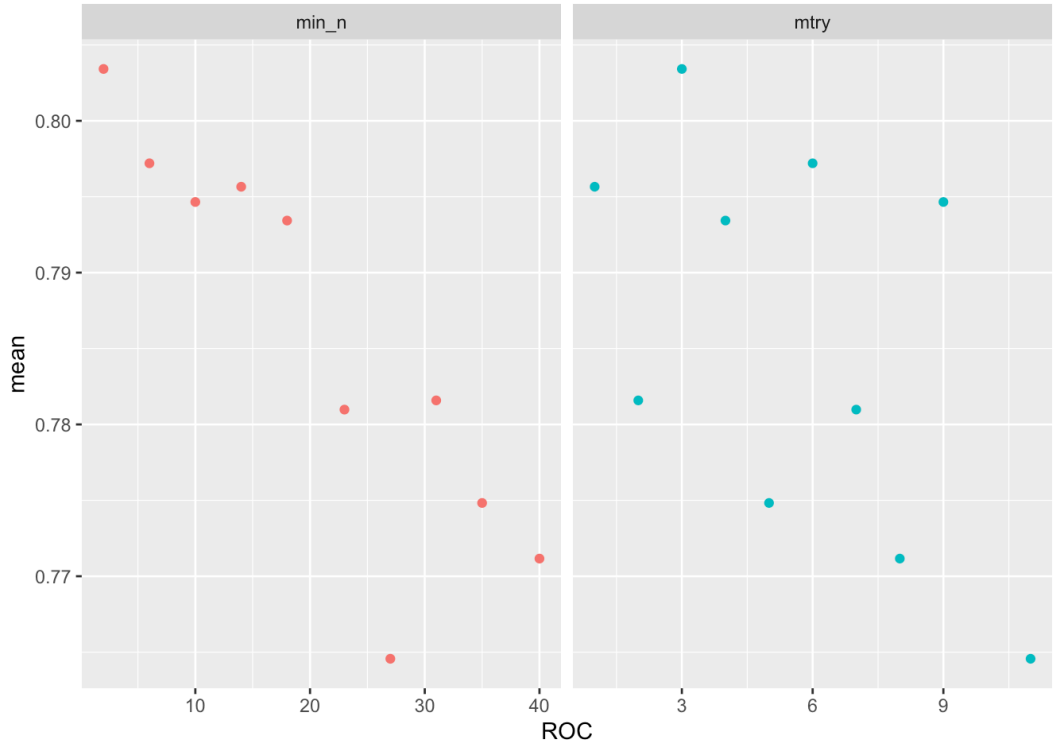
```
Warning: `pull_workflow_fit()` was deprecated in
workflows 0.2.3.
ℹ Please use `extract_fit_parsnip()` instead.
```



Random Forest

I find this quite notable. According to the gradient boosting method, alcohol is the most important predictor. Following far behind is volatile acidity, which I suspected would be more important.

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



Let me try doing an updated tuned model for random forest based on this AUC

```
rf_grid <- grid_regular(
  mtry(range = c(2,12)),
  min_n(range = c(2,6)),
  levels = 10
)


registerDoParallel()
set.seed(45632)

#random forest
rand_tune_update <- tune_grid(
  rand_wf,
  resamples = cv,
  grid = rf_grid
)
```

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

Evaluate on test set of data

Ok, with the xgboost model trained and tuned to the best performance, it's time to evaluate it on that test data we separated before. Again, I love tidymodels because the process is standard and easy when using different machine learning models.

```
final_boost_result <- last_fit(final_xgb, split)

final_boost_result %>%
  collect_metrics()
```

```
# A tibble: 3 × 4
  .metric     .estimator .estimate .config
  <chr>       <chr>          <dbl> <chr>
1 accuracy    multiclass     0.829
Preprocessor1_Model1
2 roc_auc     hand_till      0.812
Preprocessor1_Model1
3 brier_class multiclass     0.333
Preprocessor1_Model1
```

Not horrible here. The ROC is at 83% and the accuracy is 85%. I feel like that could be better though. It does not look like there is any over fitting so that's good.

```
#Making a confusion matrix
final_boost_result %>%
  collect_predictions() %>%
  conf_mat(quality_group, .pred_class)
```

```
          Truth
Prediction Low Medium High
    Low      0      0    0
    Medium   8    238   41
    High     0      0    0
```

Random Forest

Let's look at the results of that first model

```
rand_tune %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc")
```

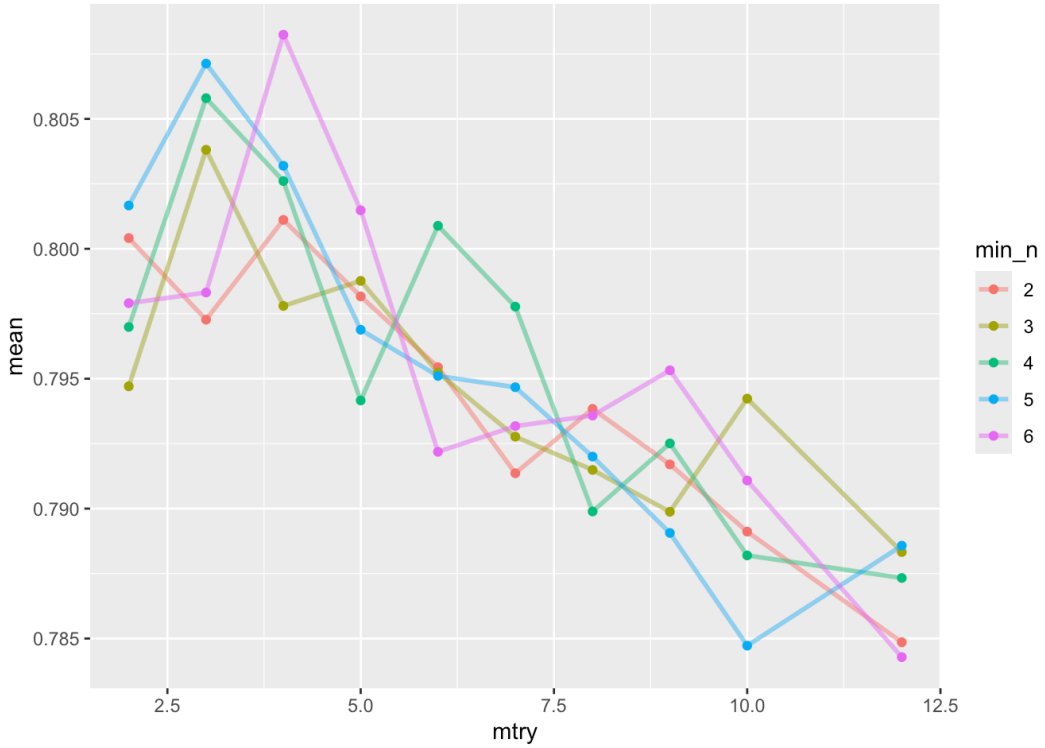
```
# A tibble: 10 × 8
    mtry min_n .metric .estimator  mean     n
std_err .config
   <int> <int> <chr>   <chr>      <dbl> <int>
<dbl> <chr>
 1     1    14 roc_auc hand_till  0.796    10
0.0262 Preprocessor1_Model01
 2     2    31 roc_auc hand_till  0.782    10
0.0283 Preprocessor1_Model02
 3     3     2 roc_auc hand_till  0.803    10
0.0307 Preprocessor1_Model03
 4     4    18 roc_auc hand_till  0.793    10
0.0287 Preprocessor1_Model04
 5     5    35 roc_auc hand_till  0.775    10
0.0301 Preprocessor1_Model05
 6     6     6 roc_auc hand_till  0.797    10
0.0299 Preprocessor1_Model06
```

```
 7     7    23 roc_auc hand_till  0.781    10
0.0313 Preprocessor1_Model07
 8     8    40 roc_auc hand_till  0.771    10
0.0282 Preprocessor1_Model08
 9     9    10 roc_auc hand_till  0.795    10
0.0318 Preprocessor1_Model09
10    11    27 roc_auc hand_till  0.765    10
0.0361 Preprocessor1_Model10
```

Now let me check the results of the second tuning model I did.

```
rand_tune_update %>%
  collect_metrics() %>%
  filter(.metric == "roc_auc") %>%
  mutate(min_n = factor(min_n)) %>%
  ggplot(aes(mtry,mean,color = min_n))+
  geom_line(alpha = 0.5, size = 1)+
  geom_point()
```

```
Warning: Using `size` aesthetic for lines was
deprecated in ggplot2 3.4.0.
ℹ Please use `linewidth` instead.
```



Select the best updated random forest model

```
rand_tune_update %>%
  select_best(metric = "roc_auc")
```

```
# A tibble: 1 × 3
   mtry min_n .config
  <int> <int> <chr>
1     4     6 Preprocessor1_Model43
```

#Last fit and finalize rf

```
Best_roc_rand <- select_best(rand_tune, metric = "r

final_rf <- finalize_model(
  rand_spec,
  Best_roc_rand
)

final_wf_rand <- workflow() %>%
```

```
  add_recipe(tidy_rec) %>%
  add_model(final_rf)
```

Now I'll use last fit to fit the final best model of random forest to the training data and evaluate it on the test data for final predictions.

```
final_random_forest_model <- final_wf_rand %>%
  last_fit(split)#just needs that splitting object
```

Now it's time to collect the final metrics to see how the model performed on the test data.

```
final_random_forest_model %>%
  collect_metrics()
```

```
# A tibble: 3 × 4
  .metric     .estimator .estimate .config
  <chr>       <chr>          <dbl> <chr>
1 accuracy    multiclass    0.885
Preprocessor1_Model1
2 roc_auc     hand_till     0.864
Preprocessor1_Model1
3 brier_class multiclass    0.0970
Preprocessor1_Model1
```

Looks like we have a final roc of 85% and final accuracy of 86%. Not too bad, that's a respectable prediction model.

Bootstrapping with Random Forest

In this extra section, I utilize the bootstrap method for sampling since the data set is relatively small and I think this data set may benefit from that.

I need to create the actual bootstrap split.

```
boot <- bootstraps(train, strata = quality_group)
```

Let me create a new recipe for this model just to make any changes easier to keep track of. Usually, I can just reuse the same recipe (the beauty of tidymodels), but I find this a bit neater. I'll be doing essentially the same recipe steps for preprocessing.

```
rand2_rec <- recipe(quality_group ~.,data = train) %
  step_normalize(all_predictors()) %>%
  step_range(all_predictors(),-all_nominal(),min = 0
  step_dummy(all_nominal(),-all_outcomes(), one_hot
  step_smote(quality_group) #Balances out variable s
```

Prepping and juicing the recipe below, just to make sure everything looks good. Yes, prepping and juicing is standard tidymodels jargon.

```
#Acts as a check to look at what we did to our recip
prep <- prep(rand2_rec)

#juice(prep)
```

Again, here is a new model specification object with 1000 trees and a new workflow combining everything.

```
rand2_spec <- rand_forest(trees = 1000) %>%
  set_mode("classification") %>%
  set_engine("ranger")#Use ranger here because it is

rand2_wf <- workflow() %>%
  add_recipe(rand2_rec) %>%
  add_model(rand2_spec)
```

Alrighty, with everything prepped and ready. It's time to create the model with the bootstrap resampling from earlier.

```
#create model with bootsraps
random_forest2_result_model <- fit_resamples(
  rand2_wf,
  resamples = boot,
  control = control_resamples(save_pred = T,verbose
```

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

Bootstrapped Model Evaluation

```
collect_metrics(random_forest2_result_model)
```

```
# A tibble: 3 × 6
  .metric     .estimator  mean     n std_err
```

```
.config
  <chr>       <chr>      <dbl> <int>   <dbl> <chr>
1 accuracy    multiclass 0.819    25 0.00399
Preprocessor1_Model1
2 brier_class multiclass 0.135    25 0.00163
Preprocessor1_Model1
3 roc_auc     hand_till  0.786    25 0.00637
Preprocessor1_Model1
```

```
#Could also use this code to see individual metrics
#show_best(random_forest2_result_model, metric = "rc
#show_best(random_forest2_result_model, metric = "ac
```

```
#Predictions
random_forest2_result_model %>%
  collect_predictions() %>% #Just this on its own wi
  conf_mat(quality_group,.pred_class)
```

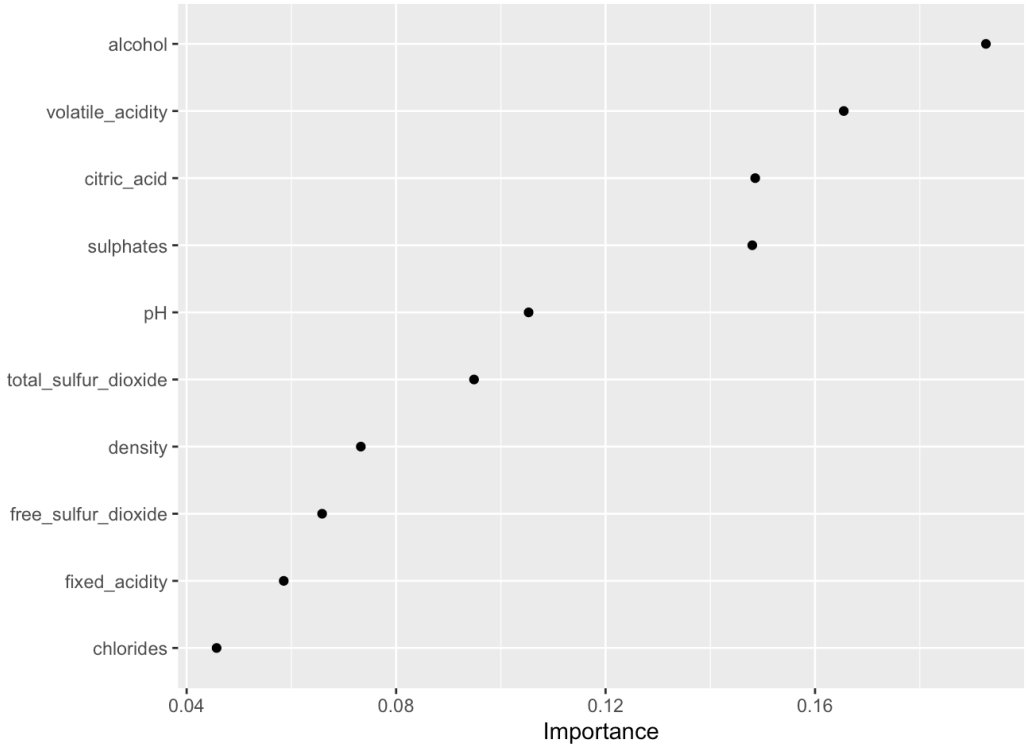
```
          Truth
Prediction  Low Medium High
    Low       8    179    2
    Medium  265   5812  431
    High      5    556  678
```

Variable Importance for Bootstrapped Model

With random forest classification using bootstrapping, volatile acidity and alcohol heavily influence the quality of wine group.

```
rand2_spec %>%
  set_engine("ranger", importance = "permutation") %
```

```
  fit(
    quality_group ~., data = juice(prep)
  ) %>%
  vip(geom = "point")
```



Final Fitting for Bootstrapped Model

Hmmm it looks like our models only performed ok. I'm not happy with ~82% usually, but for the sake of this worksheet and wanting to work on different data, I am going to call it here.

```
final_rf_boot_wf <- workflow() %>%
  add_recipe(rand2_rec) %>%
  add_model(rand2_spec)
```

```
#Again, all this needs to fit the data on the train
final_rf_res <- final_rf_boot_wf %>%
  last_fit(split)

final_rf_res %>%
  collect_metrics()
```

```
# A tibble: 3 × 4
  .metric     .estimator .estimate .config
  <chr>       <chr>          <dbl> <chr>
1 accuracy    multiclass     0.861
Preprocessor1_Model1
2 roc_auc     hand_till      0.878
Preprocessor1_Model1
3 brier_class multiclass     0.125
Preprocessor1_Model1
```

Conclusion

In this project I looked at wines with varying quality and the factors that determine said quality. The qualities were grouped together for better modeling. I performed random forest and gradient boosting models. Then I tried the bootstrapping method with random forest to see if that would change performance. Models performed okay, with more time, I would probably get a more accurate prediction.

I discovered the two most important variables usually are the alcohol percentage and level of volatile acidity when it comes to wine quality. Chances are if it's high in alcohol and low in volatile acidity, you're in for some good drinking.