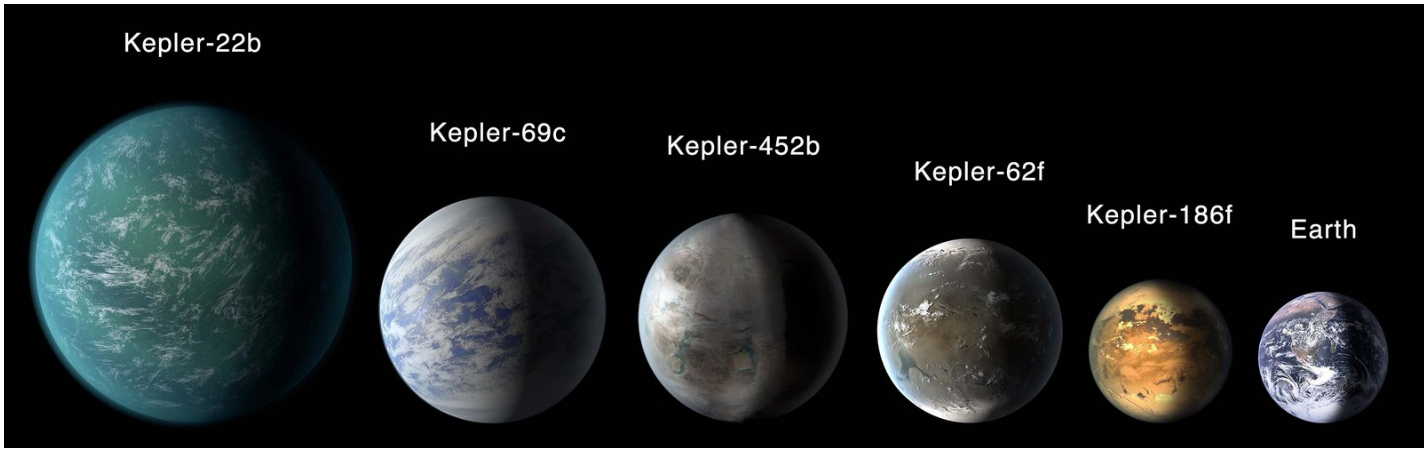
# Predicting Dispositions of Kepler Objects of Interest

Code

TJ Sipin, Preeti Kulkarni

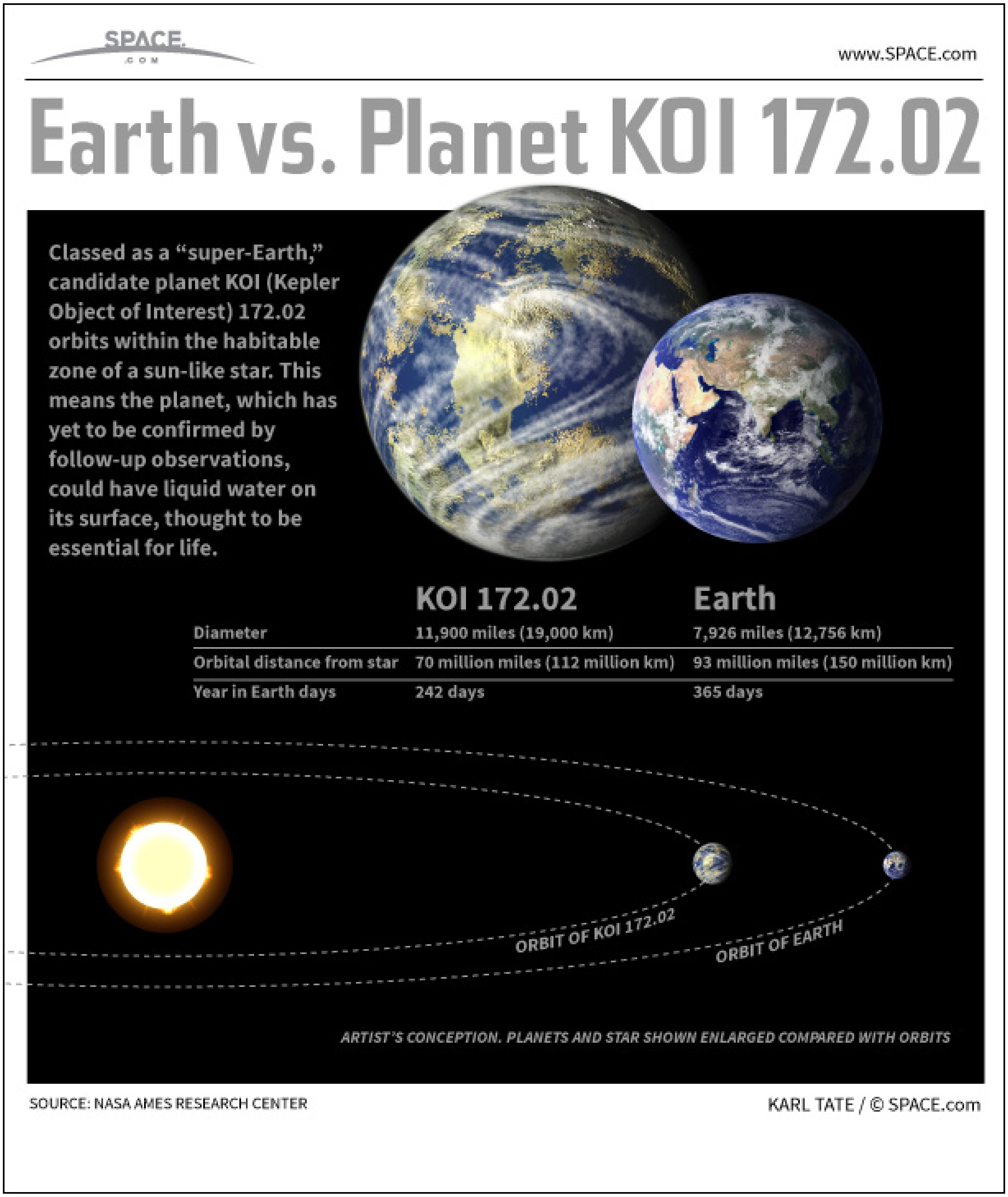
3/2/2022




# Introduction

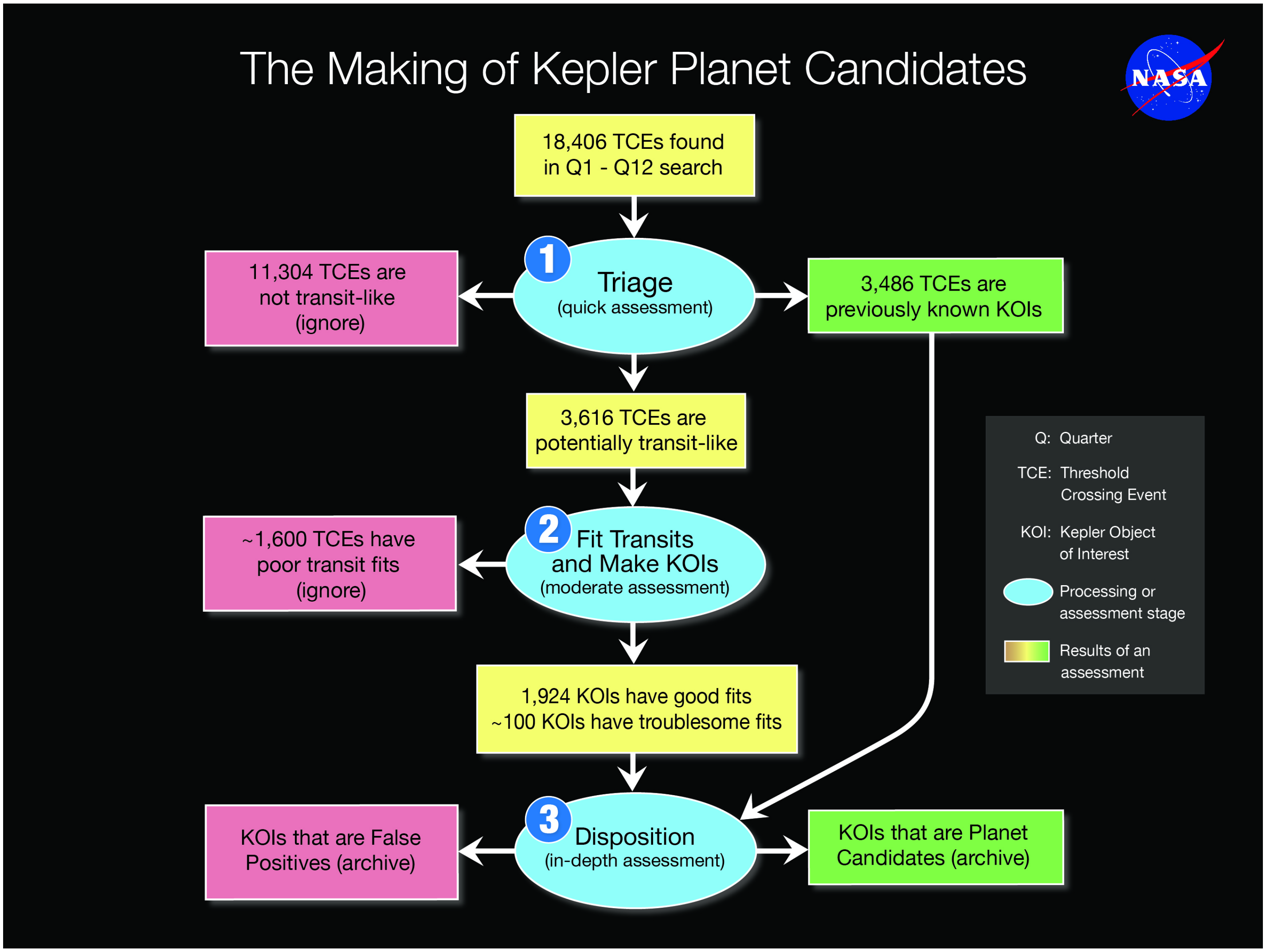
## What is KOI?

NASA studies foreign bodies and planets in space by using highly developed technology. This project uses data from the Kepler Space Observatory, which is a NASA built satellite that was launched in 2009. NASA's ultimate goal is finding other habitable planets besides Earth. NASA calls these possible habitable planets exoplanets. As of now, there are over 3000 confirmed exoplanets total, while the camera continues to do further exploration.




KOI stands for Kepler Objects of Interest. The data set we are exploring contains 10,000 exoplanet candidates that the Kepler Space Observatory has observations on (KOI). The observations depend on several variables and have many properties. We are interested in predicting the KOI disposition, which says whether a planet is confirmed, a false positive, or a candidate of being a KOI.

# Why Might This Model Be Useful?




This model is useful because it allows us to gain a better understanding and confidence on whether a planet is habitable or not, and to prioritize resources to planets predicted to be categorized as CONFIRMED or CANDIDATEs, which would save researchers time and money. It also allows us to figure out what factors are the most important when declaring a planet an exoplanet.

There is so much in space that we have not explored, and new discoveries can make a difference in history. The steps leading to those discoveries can be accomplished by using this model.

# Data Wrangling and Cleaning

The first thing we do is obtain the data set from https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=koi (https://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=koi) and select only a handful of predictors.

The most interesting predictors include:

- `koi_disposition` (our target variable): The category of this KOI from the Exoplanet Archive. Current values are CANDIDATE, FALSE POSITIVE, or CONFIRMED. All KOIs marked as CONFIRMED are also listed in the Exoplanet Archive Confirmed Planet table. Designations of CANDIDATE and FALSE POSITIVE are taken from the Disposition Using Kepler Data.
- `koi_period` : The interval between consecutive planetary transits.
- `koi_prad` : The radius of the planet. Planetary radius is the product of the planet star radius ratio and the stellar radius.

- `koi_teq` : Approximation for the temperature of the planet. The calculation of equilibrium temperature assumes a) thermodynamic equilibrium between the incident stellar flux and the radiated heat from the planet, b) a Bond albedo (the fraction of total power incident upon the planet scattered back into space) of 0.3, c) the planet and star are blackbodies, and d) the heat is evenly distributed between the day and night sides of the planet.
- `koi_insol` : Insolation flux is another way to give the equilibrium temperature. It depends on the stellar parameters (specifically the stellar radius and temperature), and on the semi-major axis of the planet. It's given in units relative to those measured for the Earth from the Sun.
- `koi_dor` : The distance between the planet and the star at mid-transit divided by the stellar radius. For the case of zero orbital eccentricity, the distance at mid-transit is the semi-major axis of the planetary orbit.

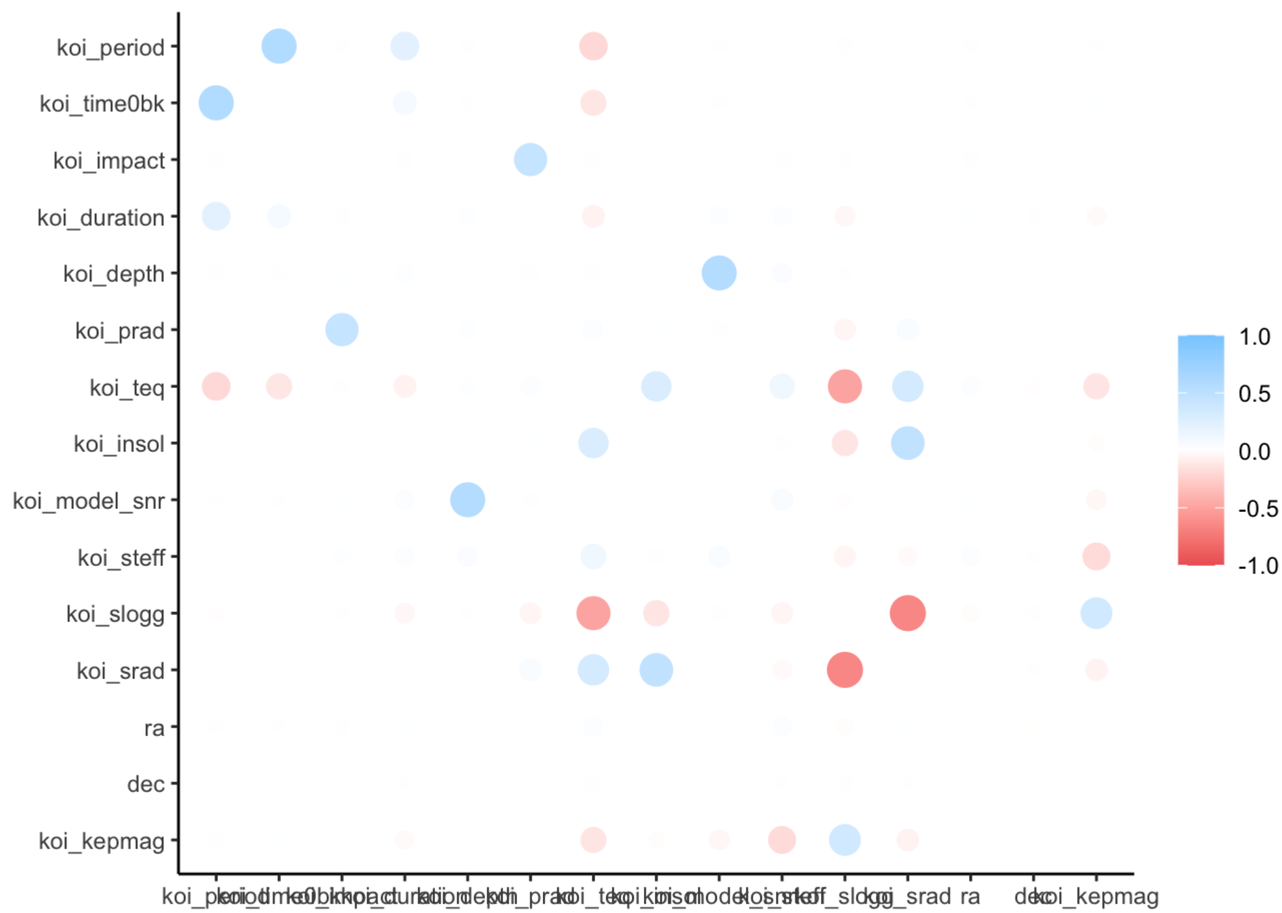
The definitions are taken from the official dictionary of this data set: https://exoplanetarchive.ipac.caltech.edu/docs/API_kepcandidate_columns.html (https://exoplanetarchive.ipac.caltech.edu/docs/API_kepcandidate_columns.html). All other definitions may be explored here.

Code

# Exploratory Data Analysis

We first begin by looking at the correlation between predictors not including those that are of class factor and integer. We can see the rplot between the numeric factors. Overall, it seems as if predictors do not have a high correlation with each other, as there are many red dots.

Code



Next, we look at some descriptive statistics of the data set. We can use the `summary()` function to find the mean, median, min, max, and quartiles of the numeric predictors. We can use the `colnames()` function to familiarize ourselves with the column names.

# Data Exploration

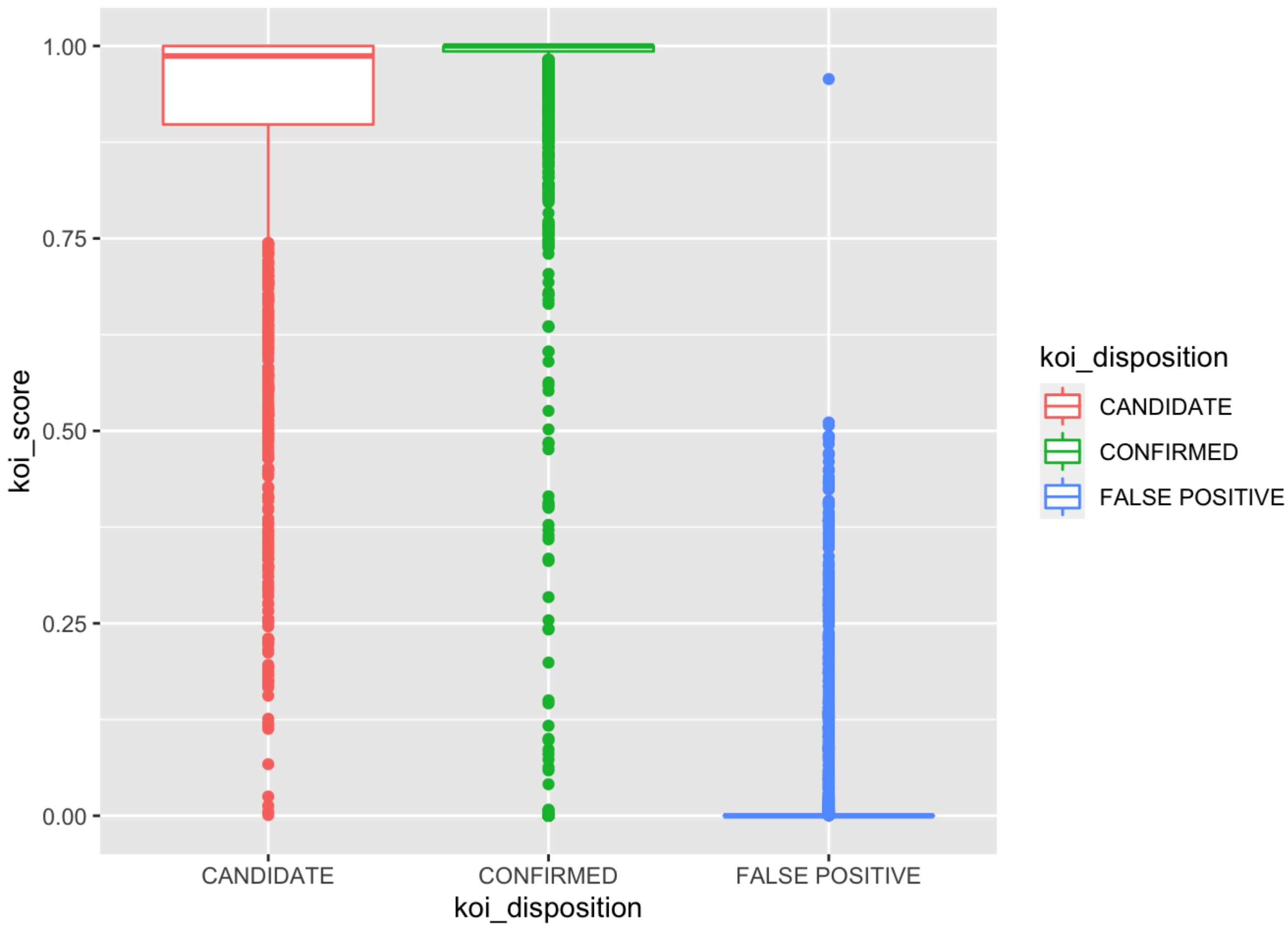
Code

```
##       koi_disposition koi_fpflag_nt koi_fpflag_ss koi_fpflag_co koi_fpflag_ec
##  CANDIDATE     :1772   0:6907        0:5744        0:6124        0:6760       
##  CONFIRMED     :2268   1: 896        1:2059        1:1679        1:1043       
##  FALSE POSITIVE:3763                                                          
##                                                                               
##                                                                               
##                                                                               
##    koi_period         koi_time0bk       koi_impact       koi_duration     
##  Min.   :   0.2598   Min.   : 120.6   Min.   : 0.0000   Min.   :  0.3028  
##  1st Qu.:   2.4555   1st Qu.: 132.6   1st Qu.: 0.2130   1st Qu.:  2.4170  
##  Median :   7.6984   Median : 136.0   Median : 0.5790   Median :  3.7360  
##  Mean   :  37.3143   Mean   : 157.9   Mean   : 0.6097   Mean   :  5.3495  
##  3rd Qu.:  24.0892   3rd Qu.: 159.8   3rd Qu.: 0.9075   3rd Qu.:  5.9730  
##  Max.   :1071.2326   Max.   :1472.5   Max.   :25.2240   Max.   :138.5400  
##    koi_depth           koi_prad           koi_teq        koi_insol       
##  Min.   :     4.5   Min.   :    0.14   Min.   :   92   Min.   :       0  
##  1st Qu.:   162.3   1st Qu.:    1.41   1st Qu.:  610   1st Qu.:      33  
##  Median :   447.8   Median :    2.46   Median :  934   Median :     180  
##  Mean   : 25728.0   Mean   :   26.97   Mean   : 1134   Mean   :    7680  
##  3rd Qu.:  1862.3   3rd Qu.:   19.02   3rd Qu.: 1426   3rd Qu.:     976  
##  Max.   :921670.0   Max.   :26042.90   Max.   :14667   Max.   :10947555  
##  koi_model_snr       koi_steff       koi_slogg       koi_srad      
##  Min.   :   2.40   Min.   : 2661   Min.   :0.047   Min.   :  0.109  
##  1st Qu.:  14.40   1st Qu.: 5306   1st Qu.:4.228   1st Qu.:  0.826  
##  Median :  27.40   Median : 5753   Median :4.442   Median :  0.993  
##  Mean   : 298.20   Mean   : 5689   Mean   :4.320   Mean   :  1.678  
##  3rd Qu.:  99.35   3rd Qu.: 6100   3rd Qu.:4.545   3rd Qu.:  1.319  
##  Max.   :9054.70   Max.   :15896   Max.   :5.364   Max.   :180.013  
##        ra              dec          koi_kepmag    
##  Min.   :279.9   Min.   :36.58   Min.   : 6.966  
##  1st Qu.:288.7   1st Qu.:40.81   1st Qu.:13.519  
##  Median :292.3   Median :43.72   Median :14.582  
##  Mean   :292.1   Mean   :43.84   Mean   :14.310  
##  3rd Qu.:295.9   3rd Qu.:46.73   3rd Qu.:15.334  
##  Max.   :301.7   Max.   :52.34   Max.   :19.065
```

# Graphs of univariate, multivariate relationships between outcome and predictor(s) or between predictors

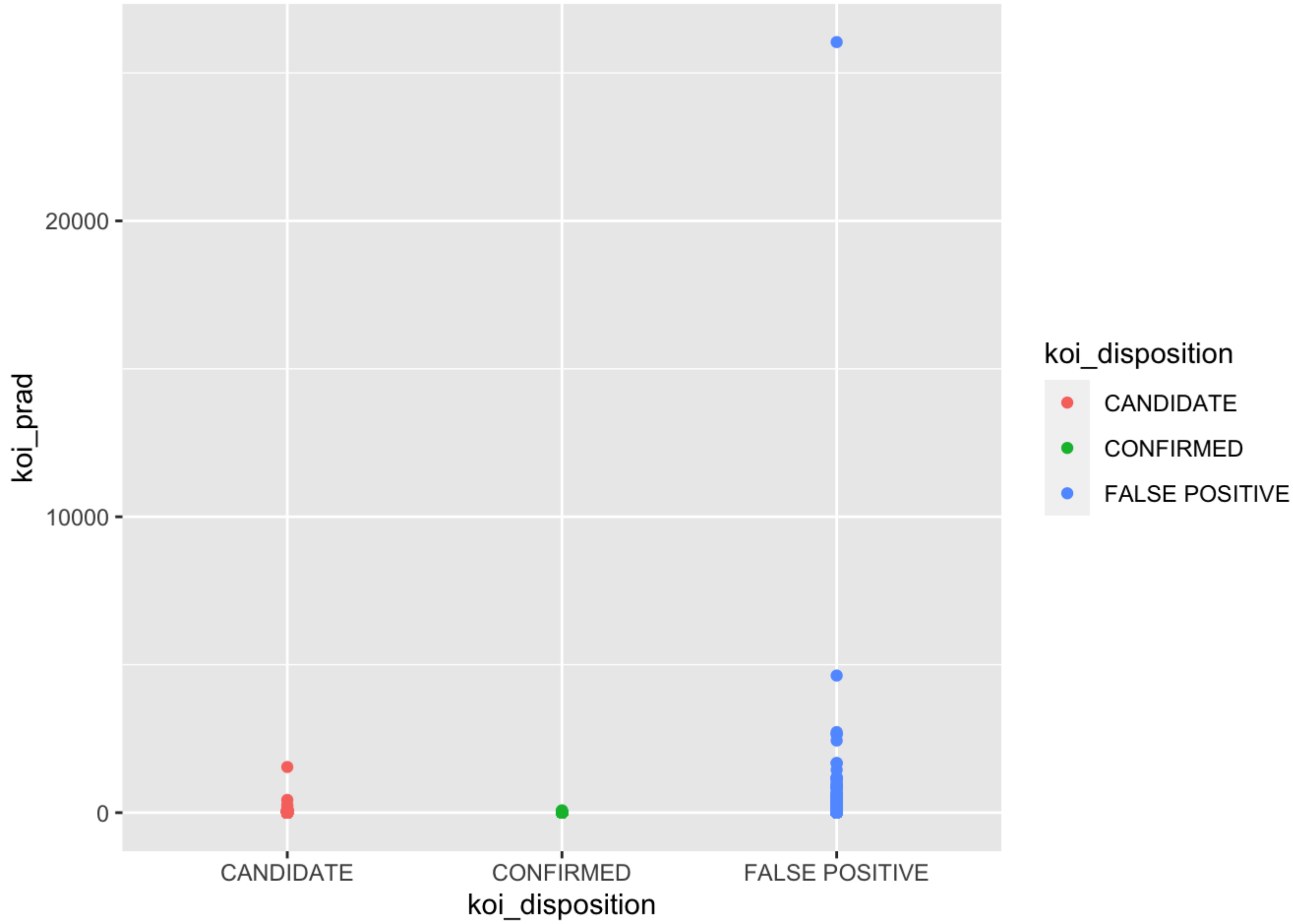
First, we plot `koi_score` versus `koi_disposition`. `koi_disposition` has the current classes: CANDIDATE, FALSE POSITIVE, or CONFIRMED. `koi_score` is a value between 0 and 1 that indicates the confidence in the KOI disposition. For CANDIDATEs, a higher value indicates more confidence in its disposition, while for FALSE POSITIVEs, a higher value indicates less confidence in that disposition. We can see that CONFIRMED cases tend to have a higher `koi_score`, where most of them are above 0.5. For CANDIDATE, the koi_score ranges from 25-75%. For the FALSE POSITIVE the `koi_score` ranges from 0-50%. Logically, this makes sense, as confirmed planets would have a higher confidence level.

Code




Next, we plot `koi_disposition` against `koi_prad`, where `koi_prad` is the radius of the planet. We are trying to see if there is a relationship with the radius and outcome of `koi_disposition`. It seems as if the confirmed planets have a smaller radius than the CANDIDATEs or FALSE POSITIVEs. We can also see one extrema in the FALSE POSITIVEs with a radius of over 20,000.

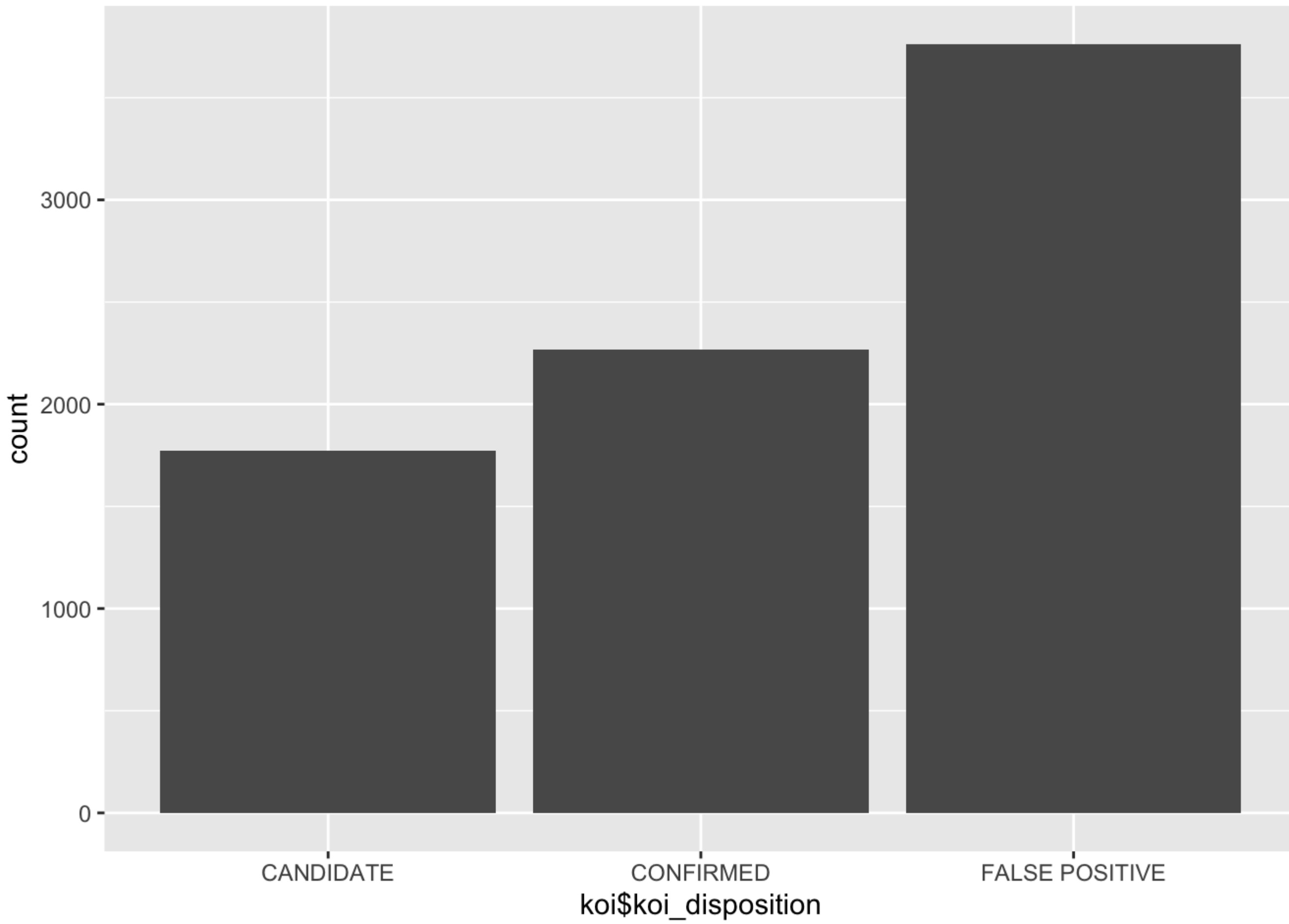
Code




Now, focusing on `koi_disposition`, we can create a histogram to see the count of observations in each class. We can see that FALSE POSITIVEs have the highest count, being over 3,000. Confirmed follows next with around 2,300 and CANDIDATE has the smallest number of counts at around 1,800.

# Histograms

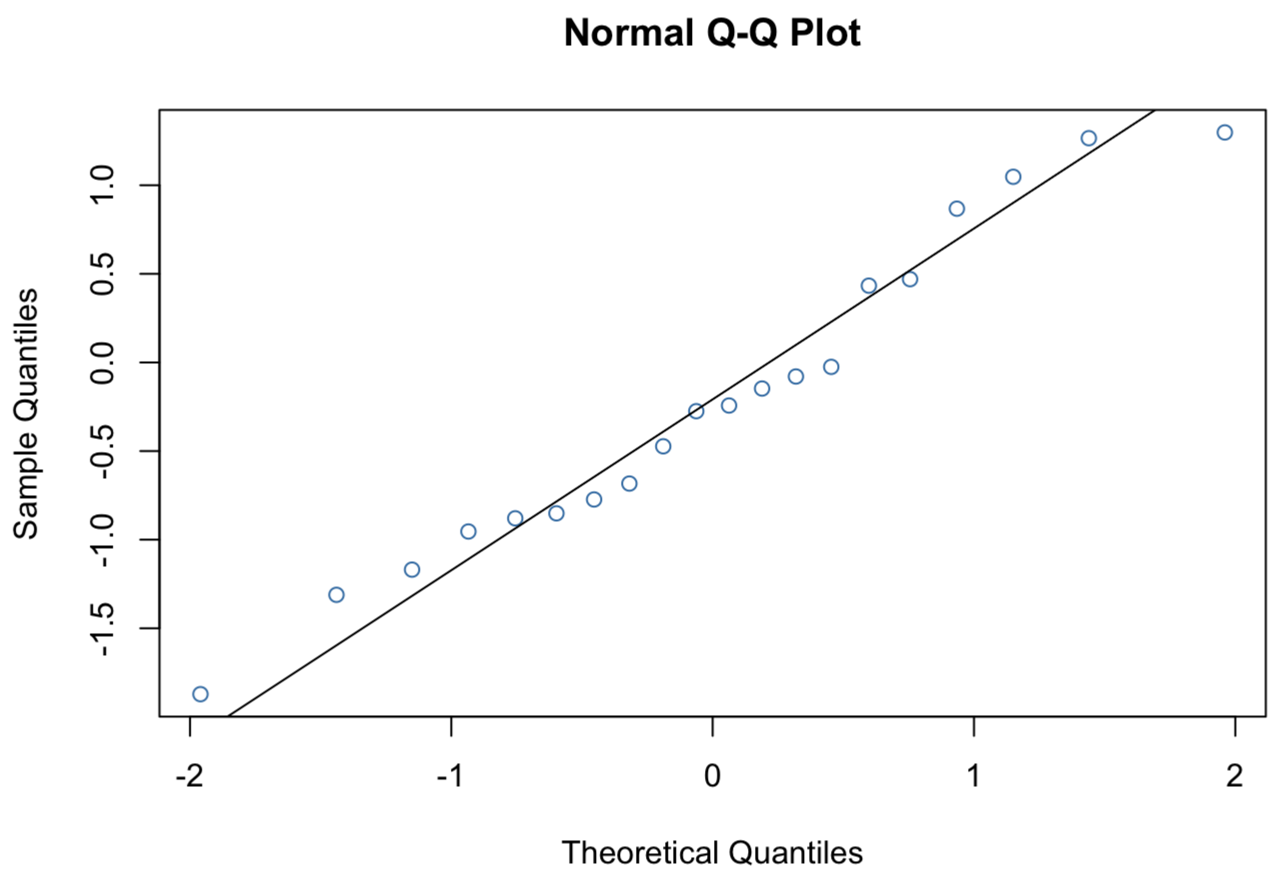
Code




Finally, we can create a qqplot. The plot show us that the the entire data set is normally distributed, because it follows a linear trend. The tails are not directly on the qqline, but mostly stay consistent with the rest of the points.

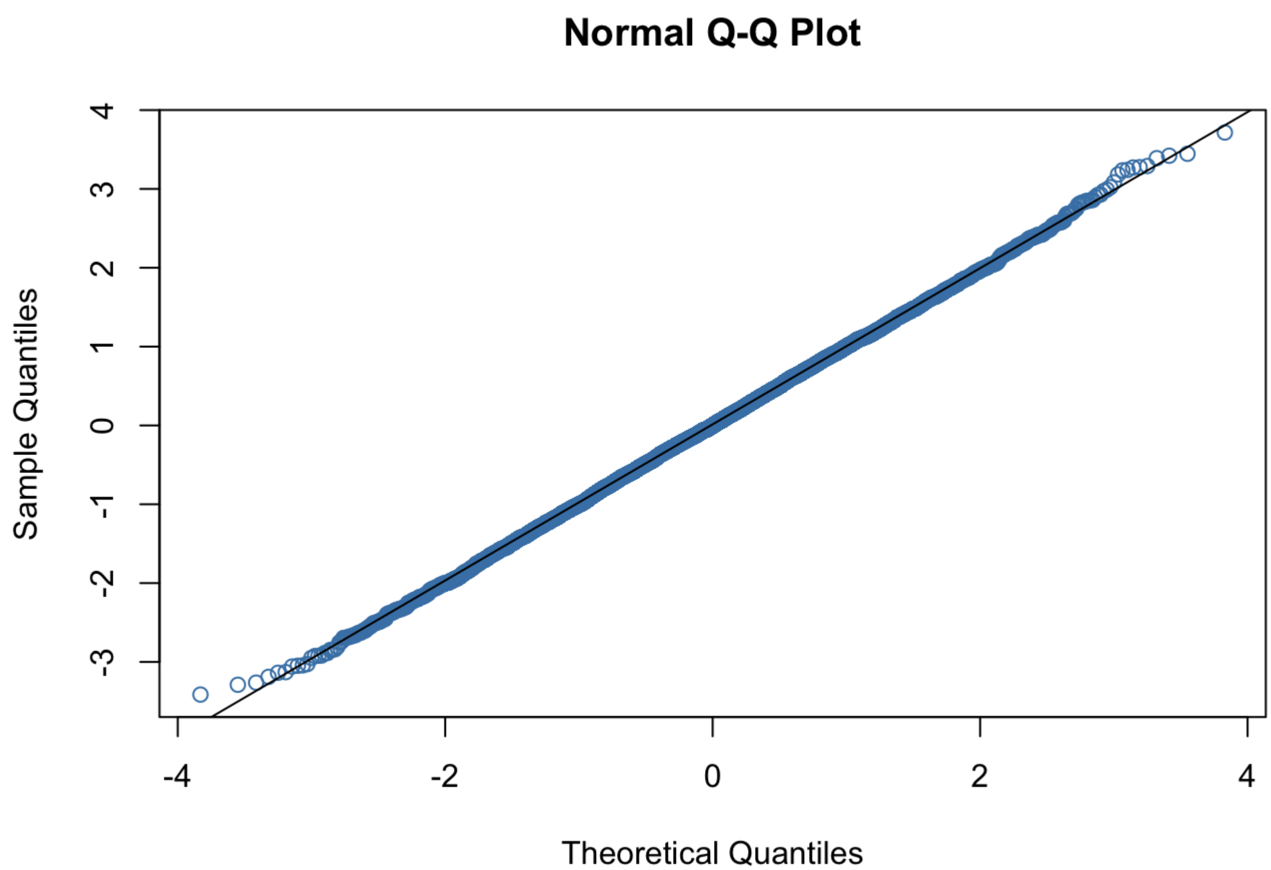
# QQ Plot

Code




Our next qqplot is to check the distribution of `koi_disposition`. Again, we can see that `koi_dispostion` is normally distributed. The points follow a much straighter line.

Code




# Pre-Model Set-up

## Splitting KOI Data into Training and Testing Sets

So that the resamples have equivalent proportions as the original data set, we use the `tidymodels` version of training/test set splitting, with the `strata` argument set so that each resample is created within `koi_disposition` .

Code

## Recipe

Using the framework from https://rviews.rstudio.com/2019/06/19/a-gentle-intro-to-tidymodels/ (https://rviews.rstudio.com/2019/06/19/a-gentle-intro-to-tidymodels/), we create our recipe, centering and scaling numeric data to have a mean of zero and standard deviation of one. This recipe will be conveniently used for three out of our four models.

Hide

```
# Recipe
recipe <- recipe(
  koi_disposition ~ koi_fpflag_nt + koi_fpflag_ss + koi_fpflag_co +
    koi_fpflag_ec + koi_period + koi_time0bk + koi_impact + koi_duration + koi_depth + koi_prad
 +
    koi_teq + koi_insol + koi_model_snr + koi_steff + koi_slogg + koi_srad +
    ra + dec + koi_kepmag, data = koi_train) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_normalize(all_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_impute_median(all_predictors())
```

# Setting Up k-Fold Cross Validation

Here, we use `vfold_cv` from the `rsample` library to set up our k-fold cross-validation. This is a simple, one-step method of k-fold cross-validation, unlike the method used in Lab 04. Like our recipe, this will be used in three of our four models.

Code

# PCA

One of the dangers of performing PCA is that we lose interpretability in the predictors. Thus, we shouldn't use PCA in our algorithms. However, we perform it anyway for data exploration. Here is a quick look of our `prcomp` object using `summary()` .

Code

```
## Importance of components:
##                           PC1    PC2    PC3    PC4     PC5     PC6     PC7
## Standard deviation     1.6717 1.3991 1.3329 1.2205 1.12136 1.01122 0.99154
## Proportion of Variance 0.1863 0.1305 0.1184 0.0993 0.08383 0.06817 0.06554
## Cumulative Proportion  0.1863 0.3168 0.4353 0.5345 0.61838 0.68655 0.75210
##                            PC8     PC9    PC10    PC11    PC12   PC13    PC14
## Standard deviation     0.89301 0.84217 0.74149 0.66395 0.61476 0.6000 0.54529
## Proportion of Variance 0.05316 0.04728 0.03665 0.02939 0.02519 0.0240 0.01982
## Cumulative Proportion  0.80526 0.85254 0.88920 0.91859 0.94378 0.9678 0.98760
##                          PC15
## Standard deviation     0.4312
## Proportion of Variance 0.0124
## Cumulative Proportion  1.0000
```

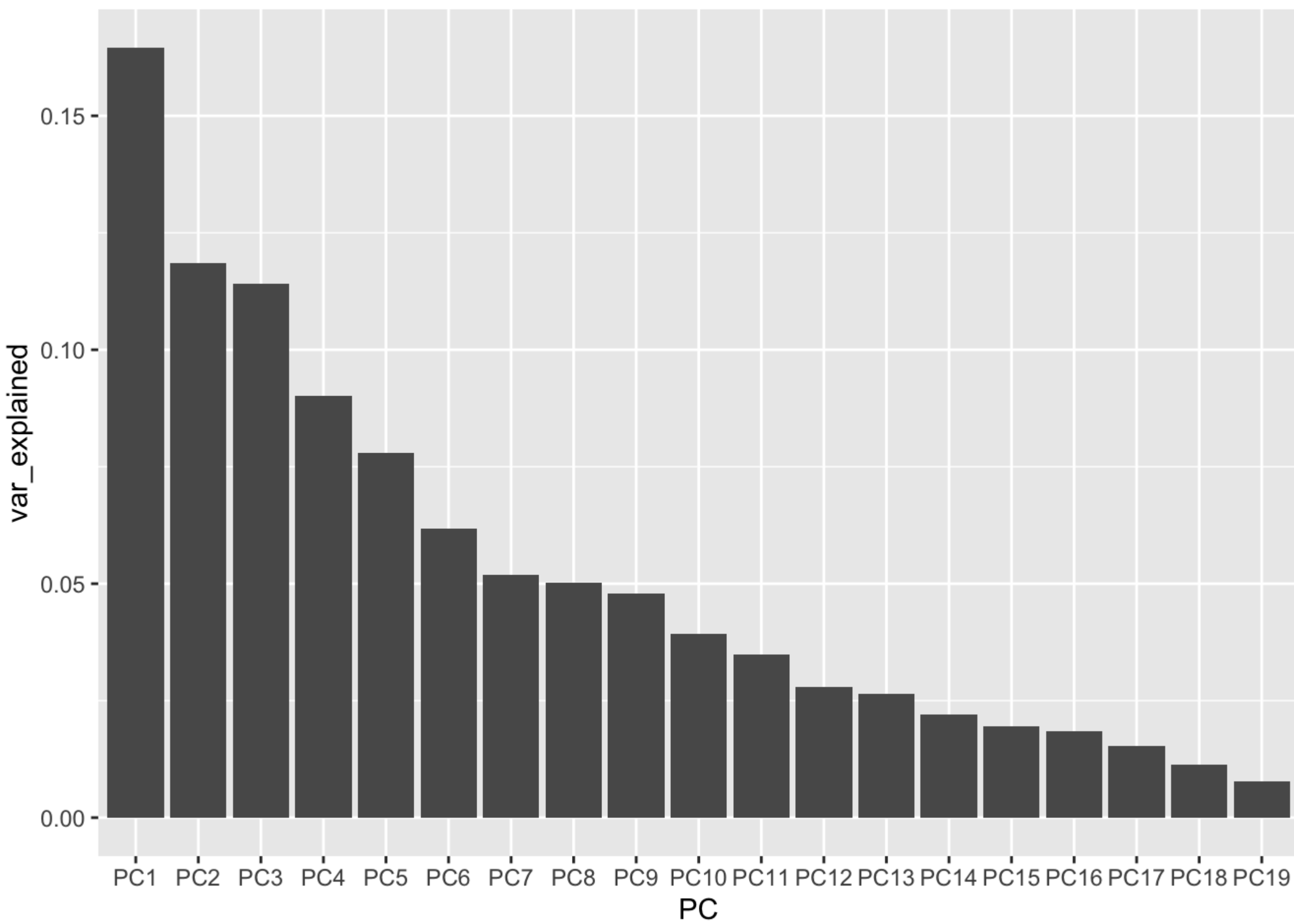
We set up our workflow, taking inspiration from https://cmdlinetips.com/2020/06/pca-with-tidymodels-in-r/ (https://cmdlinetips.com/2020/06/pca-with-tidymodels-in-r/).

Hide

```
pca_trans <- recipe %>%
  step_center(all_numeric()) %>%
  step_scale(all_numeric()) %>%
  step_pca(all_numeric())

pca_estimates <- prep(pca_trans)

sdev <- pca_estimates$steps[[7]]$res$sdev

percent_variation <- sdev^2 / sum(sdev^2)

var_df <- data.frame(PC=paste0("PC", 1:length(sdev)),
                     var_explained=percent_variation,
                     stringsAsFactors = FALSE)
```

And now, we make a scree plot.

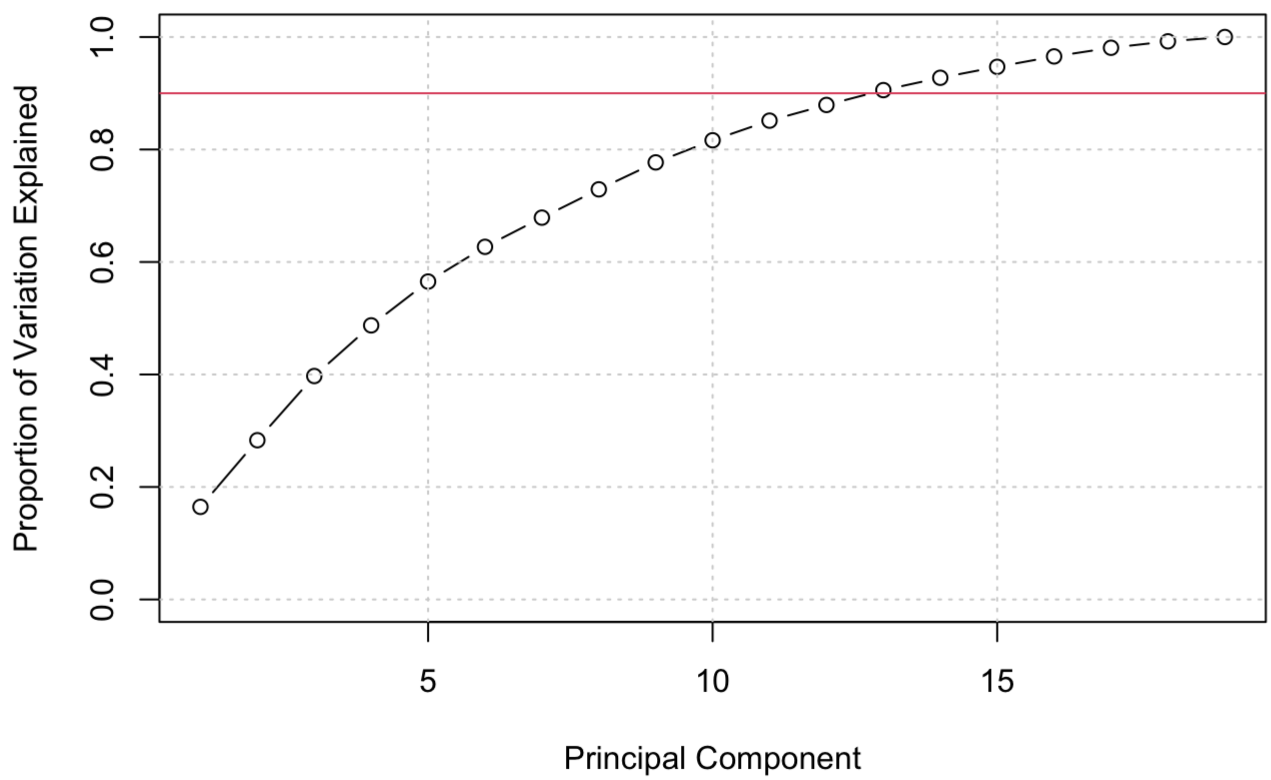
Code




As we can see, there’s no PC that particularly stands out as none exceed 20% variance explained.

And here’s a plot of the cumulative sum of explained variance by each PC.

Code




If we were to do PCA, we would use 13 PCs in our model building.

# Models

The four models we'll use are K-Nearest Neighbors, Decision Tree, Boosted Trees, and Random Forest. The choice to use a decision tree is that it is not uncommon for it to be less accurate, so we wanted to see how it compares to the other models.

# K-Nearest Neighbors

First, we split the training and testing data sets into X (predictor) and Y (outcome) subsets.

Hide

```
# Lab04

XTrain = koi_train %>% dplyr::select(-koi_disposition)
YTrain = koi_train$koi_disposition
XTest = koi_test %>% dplyr::select(-koi_disposition)
YTest = koi_test$koi_disposition
```

Then, using the code from Lab 04, we create a function `do.chunk`, which carries out k-fold cross-validation and returns a data frame of all possible values of folds, with their training and validation errors.

Code

Like in the lab, we try a 3-fold cross-validation, trying from one to 50 neighbors. Then we save data frame `error.folds` so we don't have to run it again.

Hide

```
# Set 3-fold CV

nfold = 3

set.seed(123)
folds = cut(1:nrow(koi_train), breaks=nfold, labels=F) %>% sample()
folds



# Set error.folds as a vector to save validation errors in future
error.folds = NULL

# Give possible number of nearest neighbors to be considered
allK = 1:50

set.seed(234)

# Loop through different number of neighbors
for(k in allK) {
  # Loop through different chunk id
  for (j in seq(3)) {
    tmp = do.chunk(chunkid = j, folddef = folds, Xdat = XTrain, Ydat = YTrain, k = k)

    tmp$neighbors = k # Records last number of neighbor

    error.folds = rbind(error.folds, tmp) # combines results
  }
}

save(error.folds, file = "error.rda")
```

From `error.folds`, we choose the best number of neighbors based on validation error. If there was a tie in error rate, then we would choose the larger number of neighbors for a simpler model.

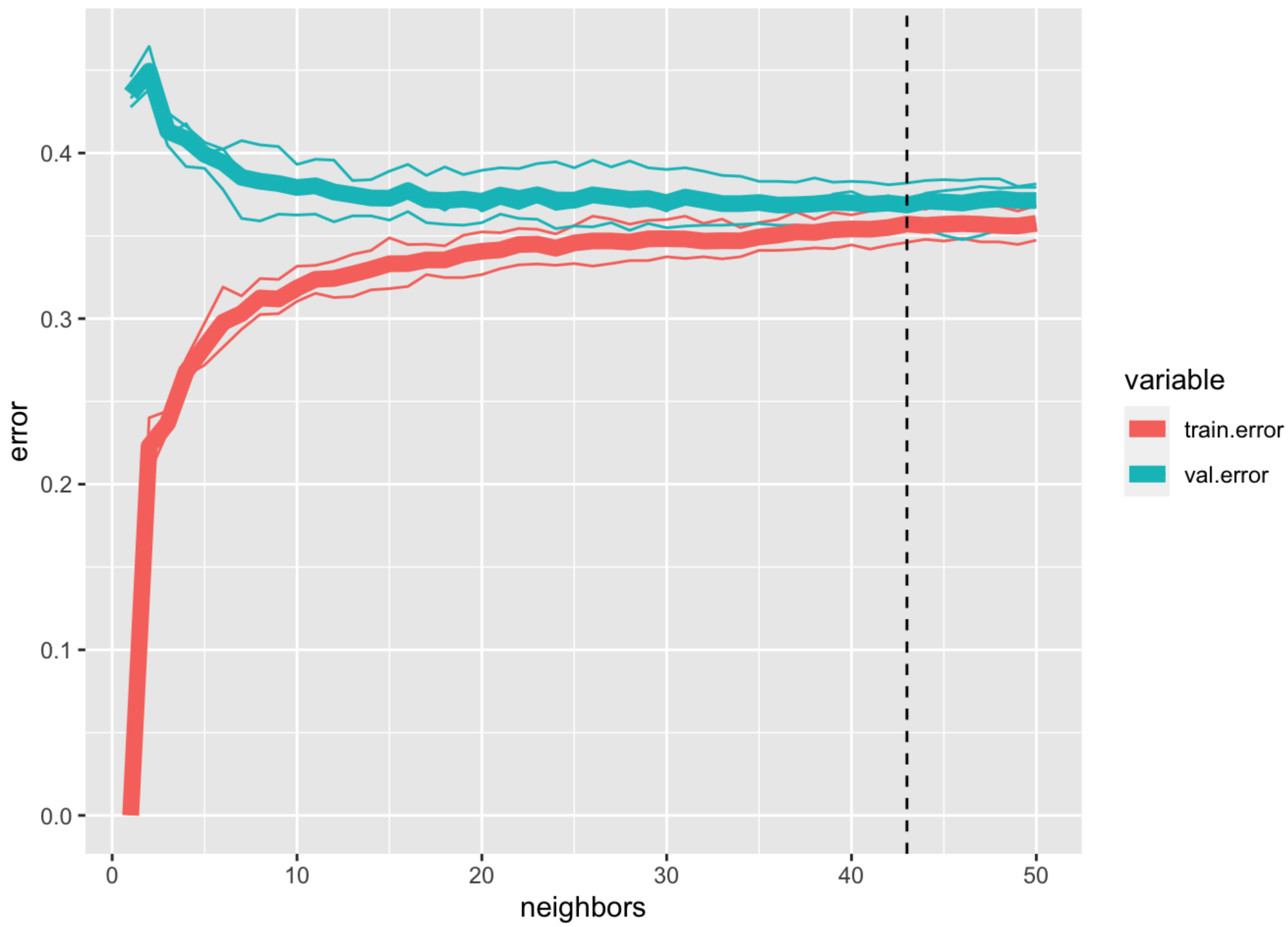
Code

```
##    fold train.error val.error neighbors
## 1     1   0.0000000 0.4459252         1
## 2     2   0.0000000 0.4276923         1
## 3     3   0.0000000 0.4331112         1
## 4     1   0.2063573 0.4643772         2
## 5     2   0.2401333 0.4384615         2
## 6     3   0.2222507 0.4454126         2
## 7     1   0.2317355 0.4243977         3
## 8     2   0.2442337 0.4046154         3
## 9     3   0.2345552 0.4105587         3
## 10    1   0.2655729 0.4161968         4
```

Code

```
## [1] "Test error rate:  0.349051768323936"
```

Code




It appears that 38 is the best number of neighbors, with a test error rate of ~ 0.36 (test accuracy rate of ~ 0.64). This isn't too terrible, but as we see later, we can definitely do better. The low accuracy rate is probably due to high dimensionality (19 predictors).

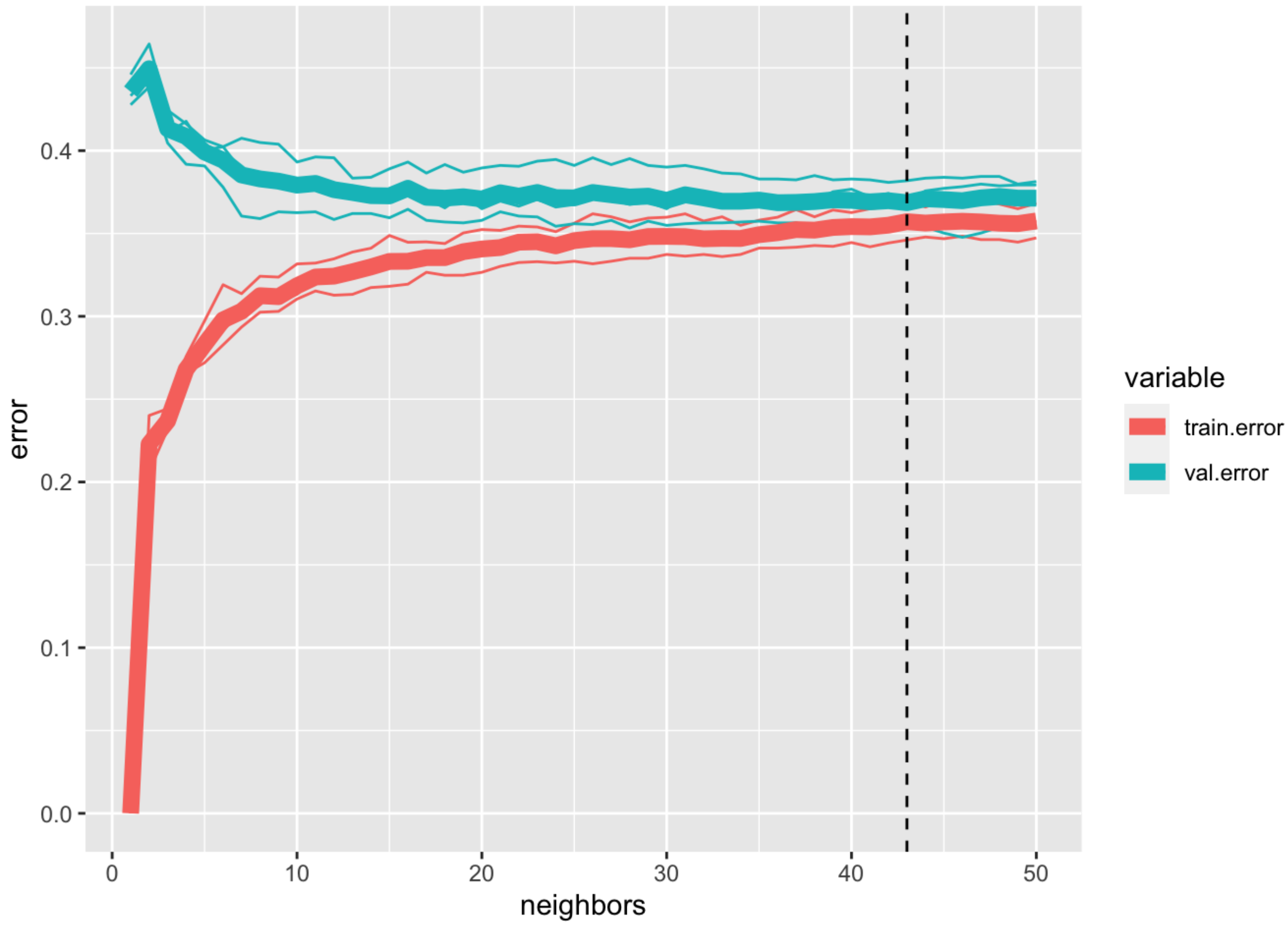
Out of curiosity, let's try KNN on only a few predictors ( `koi_period` , `koi_teq` , `koi_insol` , `koi_srad` , `koi_impact` , and `koi_steff` ). These predictors were chosen because we think that the measurements can be obtained easier than the others.

Code

Code

```
## [1] "Test error rate:  0.349051768323936"
```

Code




It appears that the test error rate increased by 0.01, despite losing 13 predictors, which is evidence against my claim that KNN would likely work better for KOI with little data on it.

# Decision Tree

Hide

```
#creating the model

tree_model <- decision_tree() %>%
  set_engine("rpart") %>%
  set_mode("classification")

set.seed(1)
koi$koi_disposition <- as.factor(koi$koi_disposition)
cell_folds <- vfold_cv(koi_train, v = 5)
```

Our first steps with the tree is to create the framework and workflow of the classification model. The `rpart` engine is used, and `koi_disposition` is transformed from character to factor.

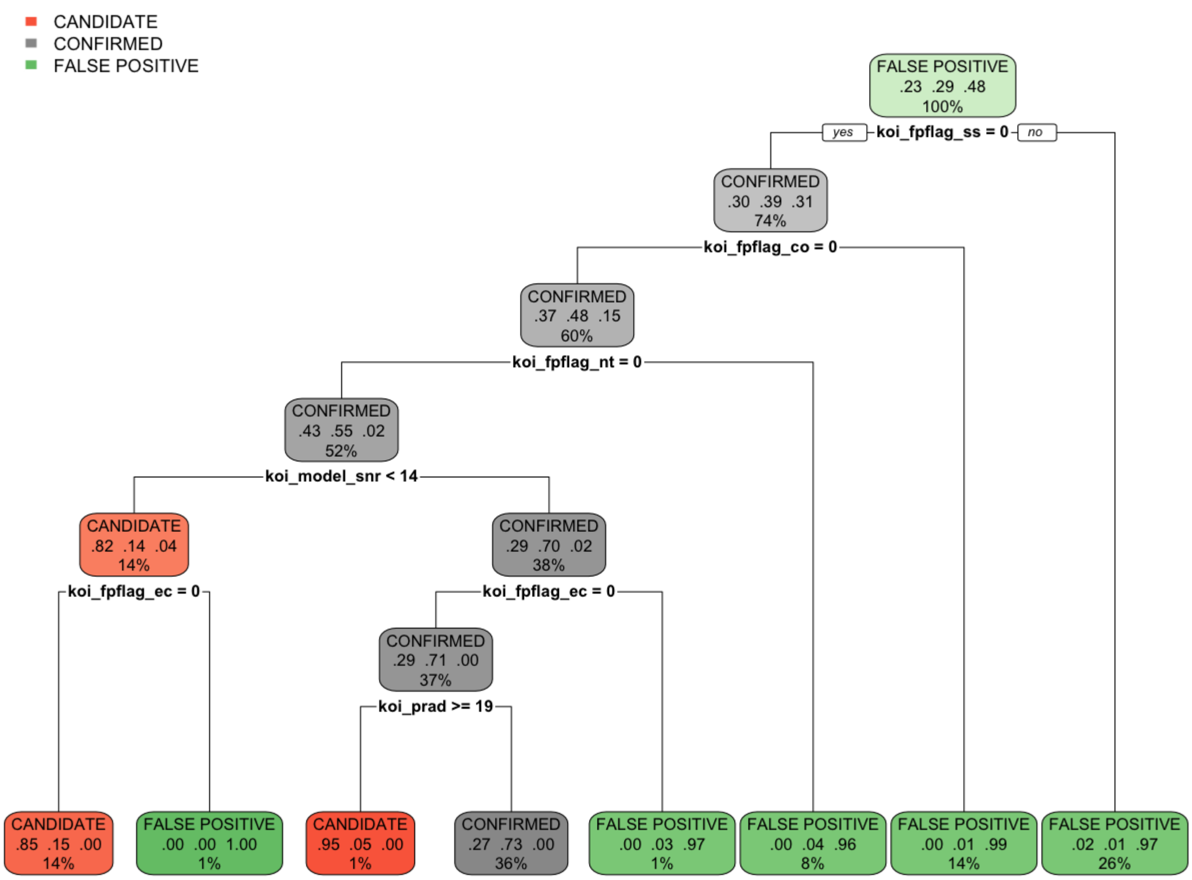
Hide

```
tree_wf <- workflow() %>%
  add_recipe(recipe)%>%
  add_model(tree_model)

# Tree Grid
tree_grid <- grid_regular(
  cost_complexity(),
  tree_depth(),
  levels = 5
)

tree_res <- tree_wf %>%
  tune_grid(
    resamples = cell_folds,
    grid = tree_grid
  )

param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)

tune_res <- tune_grid(
  tree_wf,
  resamples = cell_folds,
  grid = param_grid,
  metrics = metric_set(accuracy)
)
```

Our next step is to set up different parts of the model, beginning with the workflow and tree grid. The tree grid shows us a great visual table of the `cost_complexity` and the `tree_depth` . Then, for cross validation, we create 5 folds, and a tree grid with cost complexity and depth, that would be tuned through `param_grid` . `param_grid` is a grid created with different values that the model could go through, with 10 levels. Then, for the classification problem, we are looking at accuracy, so our metric is always set to accuracy.

Hide

```
tree_spec <- decision_tree() %>%
  set_engine("rpart")

#classification tree

class_tree_spec <- tree_spec %>%
  set_mode("classification")

#making column into factor

koi_train$koi_disposition <-as.factor(koi_train$koi_disposition)

class_tree_fit <- class_tree_spec %>%
  fit(koi_disposition ~ ., data = koi_train)

class_tree_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```




Now, we use the training data to fit `koi_disposition` against the other predictors, and can see the tree we made.

Code

```
##                 Truth
## Prediction       CANDIDATE CONFIRMED FALSE POSITIVE
##   CANDIDATE            733       122              0
##   CONFIRMED            571      1531              0
##   FALSE POSITIVE        25        48           2822
```

Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.869
```

Next, we try our model on the training data and obtain its confusion matrix, which gives us an accuracy of ~0.87, which is much better than expected and much better than the KNN model. Even without tuning, this is a great model. Let’s try tuning.

Hide

```
#fitting the model on training data

class_tree_wf <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_recipe(recipe)

param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)
```

Now that we fit our tree model for cross-validation, we use a popular workflow used by many people in the machine learning community.

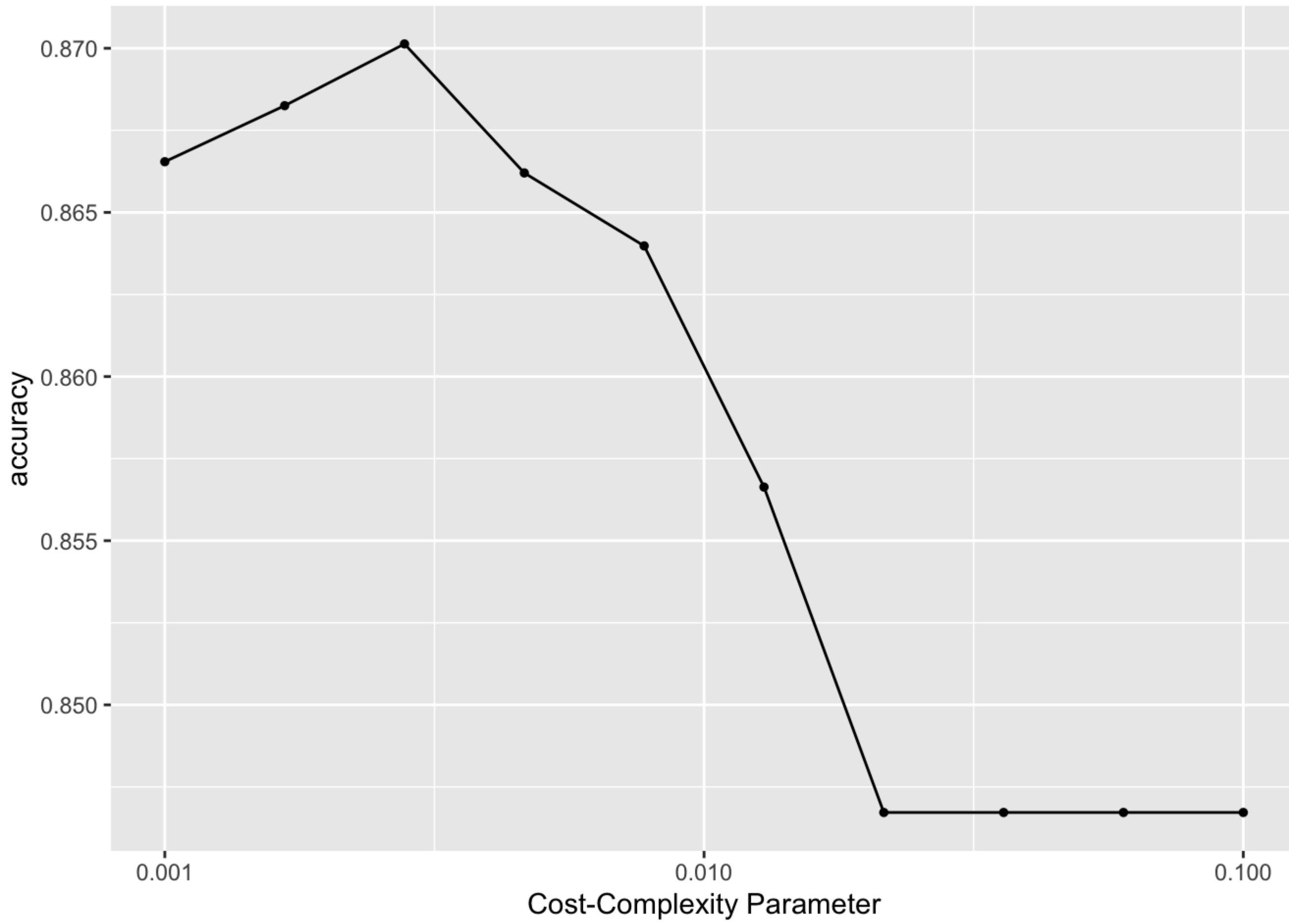
Hide

```
tune_res <- tune_grid(
  class_tree_wf,
  resamples = cell_folds,
  grid = param_grid,
  metrics = metric_set(accuracy))

save(tune_res, file = "tree_tune.rda")
```

The autoplot generated shows the accuracy and ROC AUC score. The graph itself is a Cost-Complexity Parameter graph, and we can see the highest accuracy is around 88% with the highest ROC AUC score being at 0.001.

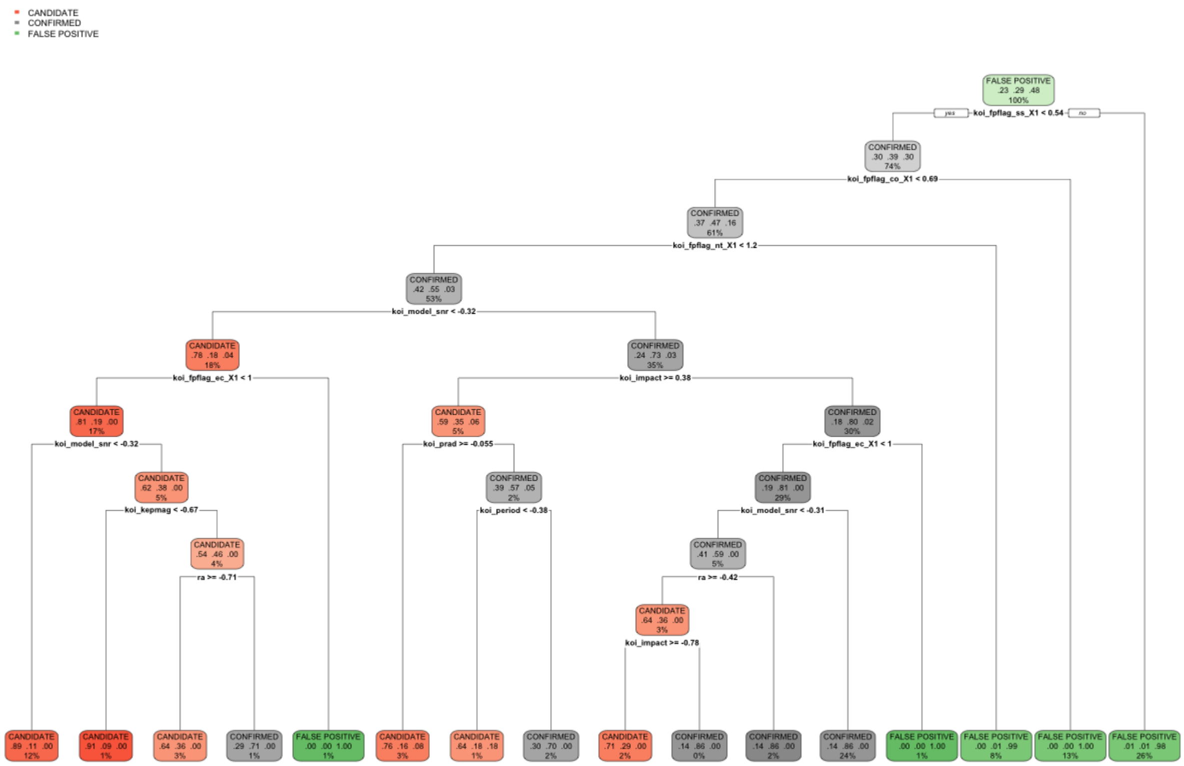
Hide

```
load("tree_tune.rda")

autoplot(tune_res)
```




Hide

```
best_complexity <- select_best(tune_res)

class_tree_final <- finalize_workflow(class_tree_wf, best_complexity)

koi_test$koi_disposition <- as.factor(koi_test$koi_disposition)

class_tree_final_fit <- fit(class_tree_final, data = koi_test)


class_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```




Hide

```
final_tree_fit = class_tree_final %>% last_fit(koi_split)

final_tree_fit %>%
  collect_metrics()
```

```
## # A tibble: 2 × 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy multiclass     0.890 Preprocessor1_Model1
## 2 roc_auc  hand_till      0.937 Preprocessor1_Model1
```

Now, we use the best complexity, which we found to be 0.001 to tune the parameters, and create a final fit of the tree, and collect the estimated metrics for Accuracy and ROC AUC.

# Boosted Tree

As we did previously, we first begin by creating the framework of the model, using classification and xgboost as the engine. Our tuning parameters are min_n, mtry, and learn_rate. Again, we add a workflow model, and parameters to tune.

As we did previously, we first begin by creating the framework of the model, using classification and `xgboost` as the engine. Our tuning parameters are `min_n` , `mtry` , and `learn_rate` .

Code

Now, we define the tuning grid, with 3 levels for each class this time. We tune the model using the `koi_folds` we made earlier, and call the metrics from workflow. We can see `accuracy` , `kap` , `roc_auc` , `recall` , and more. The runtime for `boost_tune` and `boost_res` is quite long, so we save it to a file to be loaded in later.

Code

Finally, after collecting the metrics, and seeing the accuracy, we can use autoplot to plot out the results.

Code

```
## [19:28:17] WARNING: amalgamation/../src/learner.cc:1115: Starting in XGBoost 1.3.0, the defau
lt evaluation metric used with the objective 'multi:softprob' was changed from 'merror' to 'mlog
loss'. Explicitly set eval_metric if you'd like to restore the old behavior.
```

Code

```
## # A tibble: 2 × 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy multiclass     0.890 Preprocessor1_Model1
## 2 roc_auc  hand_till      0.954 Preprocessor1_Model1
```

Our boosted tree seems to do just a bit better than the decision tree at an accuracy of 0.886.

# Random Forest

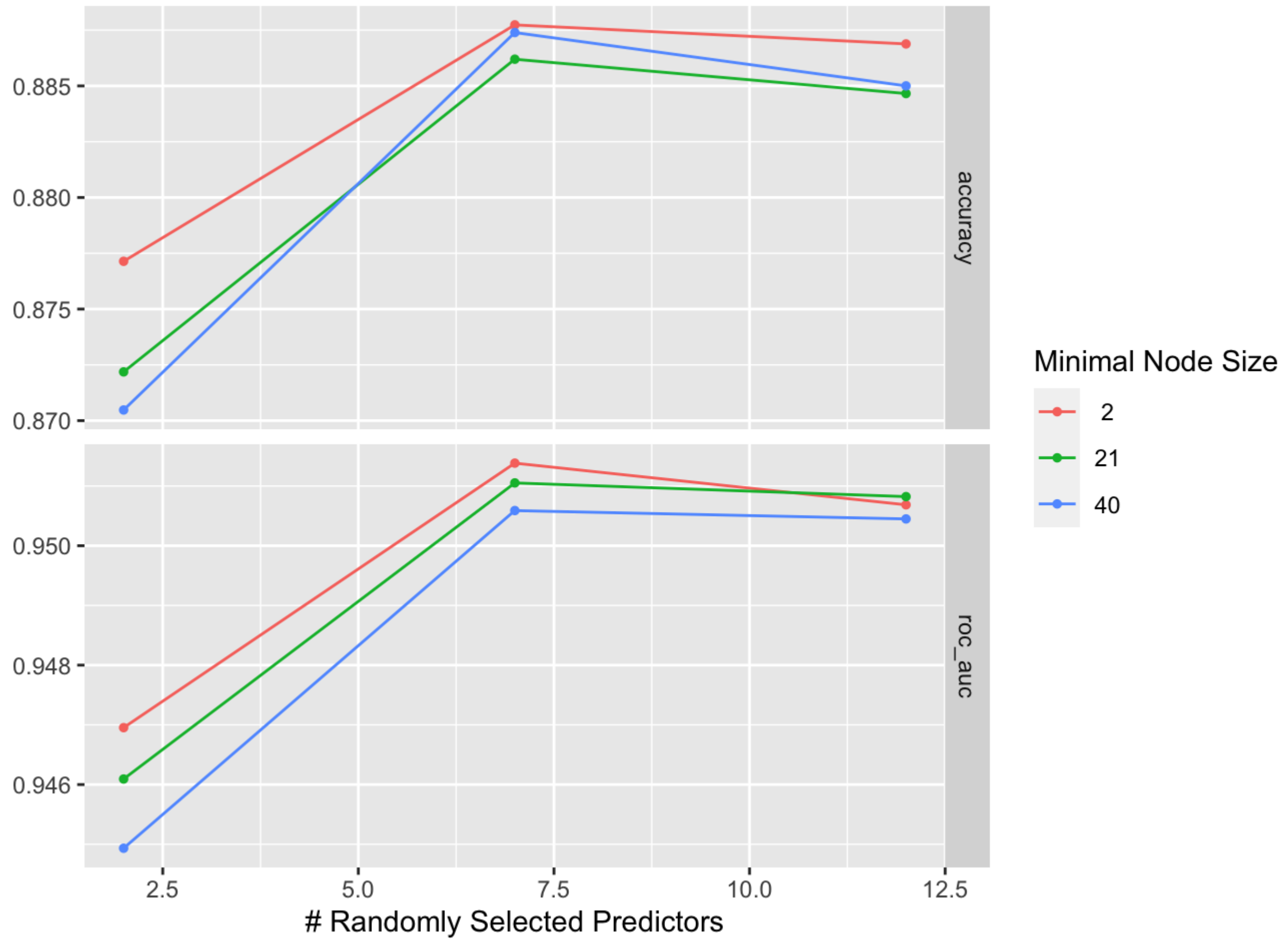
As usual, we set up our model workflow. We set `mtry` to a range of two to 12 in our cross-validation.

Code

Due to long runtimes, we save `rf_tune` , `rf_workflow` , and `rf_res` to a file to be read later.

Code

Code




Code

```
## # A tibble: 5 × 6
##    mtry min_n .metric   mean     n std_err
##   <int> <int> <chr>    <dbl> <int>   <dbl>
## 1     7     2 accuracy 0.888    10 0.00444
## 2     7    40 accuracy 0.887    10 0.00457
## 3    12     2 accuracy 0.887    10 0.00461
## 4     7    21 accuracy 0.886    10 0.00527
## 5    12    40 accuracy 0.885    10 0.00501
```

We see an accuracy of ~0.888 for a random forest with parameters `mtry = 7` and `min_n = 2`.

Code

```
## # A tibble: 2 × 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy multiclass     0.890 Preprocessor1_Model1
## 2 roc_auc  hand_till      0.955 Preprocessor1_Model1
```

Our random forest performs better by only 0.001 compared to the boosted tree model. Even though the improvement is negligible, we'll still use the random forest as our best model and explore it.

# Best Model

Upon comparison among all four models, the random forest performed the best on the test data, but by only a margin of 0.001 compared to the boosted tree model.

We use https://rviews.rstudio.com/2019/06/19/a-gentle-intro-to-tidymodels/ (https://rviews.rstudio.com/2019/06/19/a-gentle-intro-to-tidymodels/) as a guide to explore other metrics.

First, we make a final model by taking the best parameters obtained ( `min_n = 2` and `mtry = 7` ) in the previous section and run the fit.

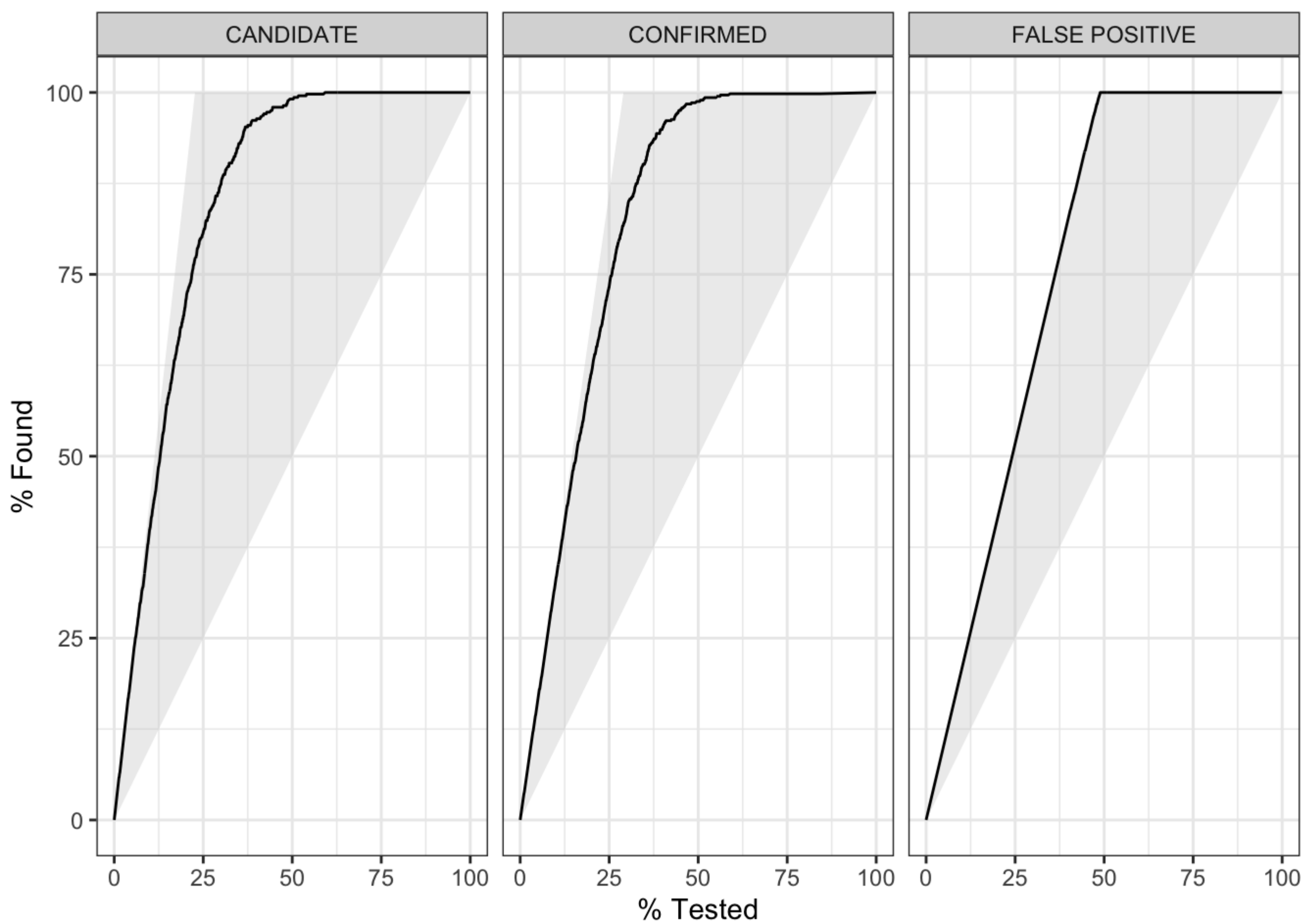
Code

```
## # A tibble: 2 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.894
## 2 kap      multiclass     0.832
```

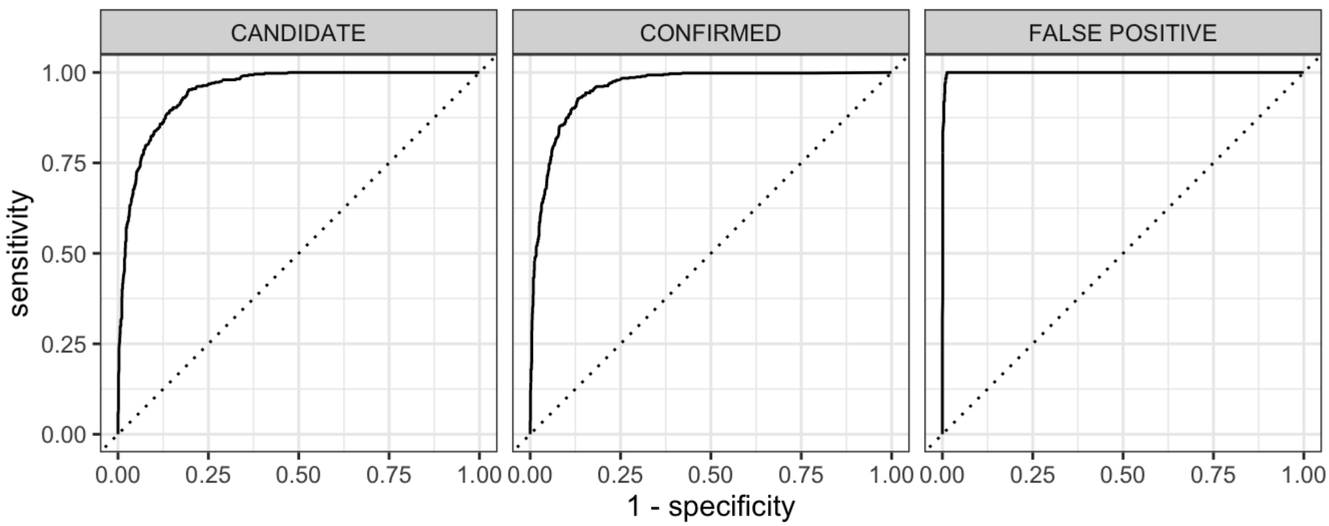
With an accuracy of 0.891, the model performs quite impressively.

Now we can explore the performance of our model by each predicted value in our sample space: CANDIDATE, CONFIRMED, and FALSE POSITIVE.

Code




Code




Code

We ran both a gain curve plot and an ROC AUC plot. The ROC AUC plot is highly promising, with the FALSE POSITIVE plot very close to the optimal plot. The plots for the other outcomes are also similarly good. Additionally, the gain curve plot tells us that for CANDIDATE and CONFIRMED, among the 25% of observations with highest probability of being CANDIDATE and CONFIRMED respectively, we will get ~75% of allCANDIDATE and CONFIRMED observations.

We now run different metrics and see the estimates for each.

Code

```
## # A tibble: 4 × 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 accuracy    multiclass     0.894
## 2 kap         multiclass     0.832
## 3 mn_log_loss multiclass     0.278
## 4 roc_auc     hand_till      0.956
```

Overall, we have a very high accuracy similar to our boosted tree and pruned models on our testing data.

Like with what we did with KNN, let’s try running the tree on the same few predictors that we believe are easier to obtain, like `koi_period` , `koi_teq` , `koi_insol` , `koi_srad` , `koi_impact` , and `koi_steff` .

We create a mini recipe first.

Hide

```
recipe.few <- recipe(
  koi_disposition ~ koi_period + koi_teq + koi_insol + koi_srad + koi_impact + koi_steff, data =
koi_train) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_normalize(all_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_impute_median(all_predictors())
```

Then we perform the same process as above to find the best parameters.

Hide

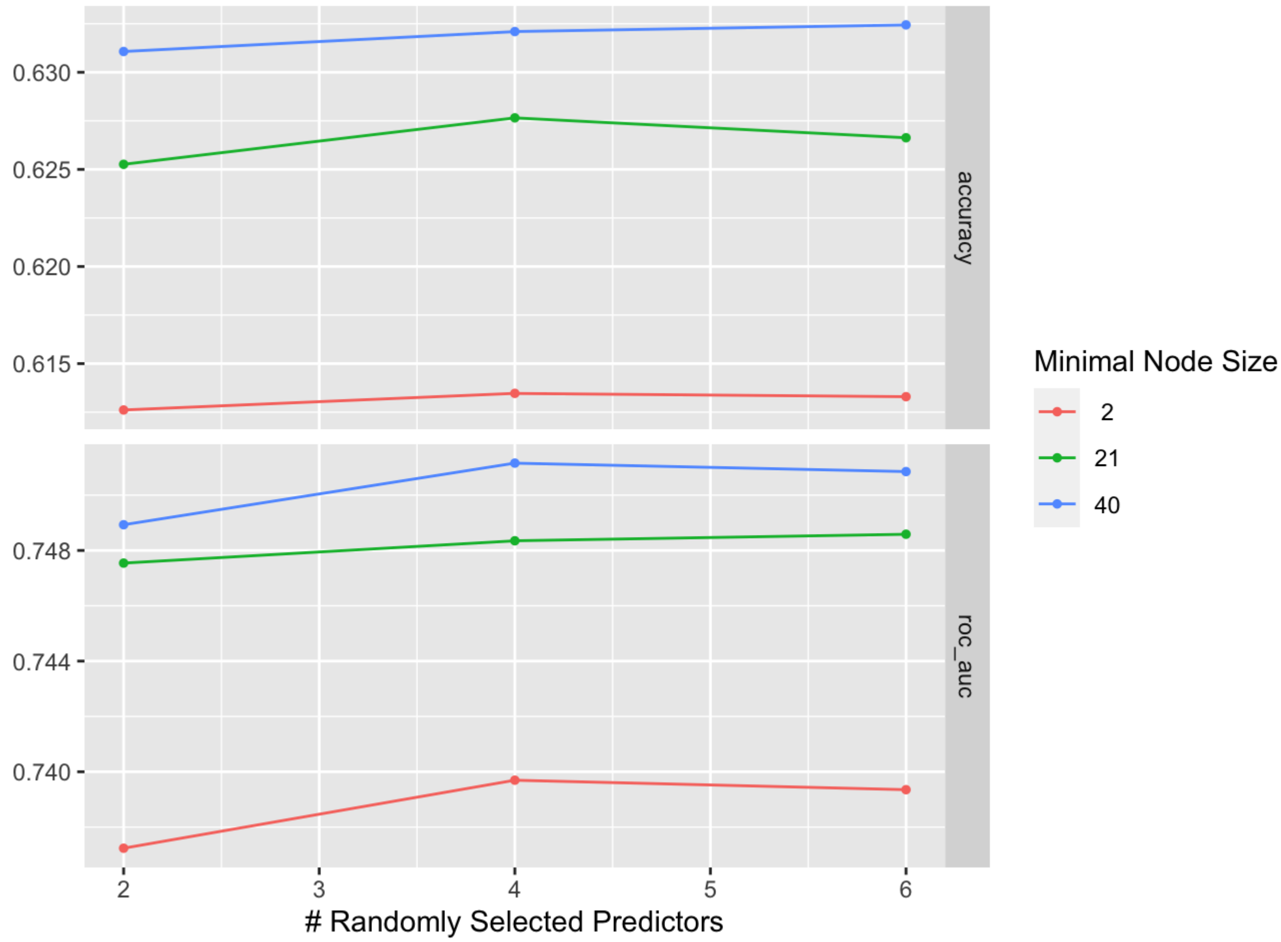
```
rf_model <-
  rand_forest(min_n = tune(),
              mtry = tune(),
              mode = "classification") %>%
  set_engine("ranger")


rf_workflow.few <- workflow() %>%
  add_model(rf_model)  %>%
  add_recipe(recipe.few)

rf_params.few <- parameters(rf_model) %>%
  update(mtry = mtry(range = c(2, 6)))


rf_grid.few <- grid_regular(rf_params.few,
                        levels = 3)
```

Hide

```
rf_tune.few <- rf_workflow.few %>%
  tune_grid(resamples = koi_folds,
            grid = rf_grid.few)

save(rf_tune.few, rf_workflow.few, file = "rf_tune_few.rda")
```

Hide

```
load("rf_tune_few.rda")

autoplot(rf_tune.few)
```




Hide

```
show_best(rf_tune.few, metric = "accuracy") %>% dplyr::select(-.estimator, -.config)
```

```
## # A tibble: 5 × 6
##    mtry min_n .metric   mean     n std_err
##   <int> <int> <chr>    <dbl> <int>   <dbl>
## 1     6    40 accuracy 0.632    10 0.00853
## 2     4    40 accuracy 0.632    10 0.00924
## 3     2    40 accuracy 0.631    10 0.00836
## 4     4    21 accuracy 0.628    10 0.00847
## 5     6    21 accuracy 0.627    10 0.00839
```

Hide

```
best_complexity.few <- select_best(rf_tune.few, metric = "accuracy")

class_rf_final.few <- finalize_workflow(rf_workflow.few, best_complexity.few)

class_rf_final_fit.few <- fit(class_rf_final.few, data = koi_test)



final_rf_fit.few = class_rf_final.few %>% last_fit(koi_split)

final_rf_fit.few %>%
  collect_metrics()
```

```
## # A tibble: 2 × 4
##   .metric  .estimator .estimate .config
##   <chr>    <chr>          <dbl> <chr>
## 1 accuracy multiclass     0.635 Preprocessor1_Model1
## 2 roc_auc  hand_till      0.756 Preprocessor1_Model1
```

As expected, the model performs much worse than with more predictors. It’s my guess that this is because each predictor is relatively similarly equivalent in significance, as we see in the PCA section. However, maybe through cross-validation, we can obtain the right parameters for the random forest to use with these predictors.

With an accuracy of 61.7% on testing data, the random forest isn’t as good as our reduced KNN model earlier. Thus, for observations with less data, we can should KNN to classify between CANDIDATE, FALSE POSITIVE, and CONFIRMED.

# Conclusion

We used four models: KNN, Decision Tree, Boosted Trees, and Random Forest. For the full set of predictors, KNN did not work as well as the other three, but nevertheless, still produced results with an accuracy greater than 33% (assuming . Our cleaned data set included 22 variables which did not have a strong correlation to each other. This made it difficult to remove these variables, because we did not originally know what affected another, or how removing one would affect the rest. We were able to use the properties of Random Forests to determine what variables to use from this large list. The model was able to perform so well because of its use of random subsets (bootstrapping), random set of features deciding the best split, and the average of all predictions from the decision trees. We thought this model would work the best because of our variable size and the use of many trees instead of just one.

We did not expect our KNN model to work as well as the rest because of the sensitivity to outliers, the assumption that similar points share similar labels, and the fact that we have a large data set. Furthermore, we know KNN is not good for large data sets since the training data is processed for every predictor, making it an inefficient model. However, for observations with less data, KNN did work better than the random forest model.

For the full set of predictors, since the Decision Tree, Boosted Trees, and Random Forest models pushed towards an accuracy of 90% on testing data, we can be quite confident that implementation of our model can result in a more efficient workflow for KOI researchers as they can now prioritize objects predicted to be CANDIDATEs or CONFIRMED. It must be reiterated that this only works for observations with the properties we used in our model already. Further development of models that use properties that are easier to obtain is a necessary next step to further workflow efficiency for those studying KOIs.