PD_Detection

Alex Canfield and Jeremy Giese

3/12/2020

Parkinson's Disease Detection

The goal of this project is to create a prediction model for detecting Parkinson's Disease (PD) by using voice recording metrics from a patient. The major aspects of our project include a correlation analysis, hypothesis testing, building a prediction model, and an analysis of that model. This methods paper will primarily focus on the first three of those tasks. There is a brief analysis at the end, but the analysis will be the focus of our next paper. The main goal of this paper is to detail the model creation process from reading in our dataset to obtaining results from our model.

First, we need to read in our dataset. This dataset contains metrics on 240 voice recordings of patients making a sustained /a/ vowel sound. There are three recordings per patient, 40 patients with PD and 40 without PD. We do not have access to the recordings themselves, as those are subject to HIPAA restrictions, but Naranjo developed a system to take these audio files and calculate metrics such as pitch, fluctuation, and jitter. Naranjo showed that these metrics can be used to detect PD in a patient. This is based on the fact that PD affects the vocal cords before becoming traditionally symptomatic with hand tremors and other symptoms. This project seeks to validate these results and, if possible, improve on them.
Here, we read in our dataset and import the libraries that this project will require. Most of these will be used towards the end when our model is created.

```r
library(corrplot)
library(randomForest)
library(caret)
library(e1071)
require(caTools)

pd.data = read.csv("parkinsons.csv")
dim(pd.data)
```

```
## [1] 240  48
```

```r
sapply(pd.data, class)
```

```
##          ID  Recording     Status     Gender Jitter_rel Jitter_abs Jitter_RAP
##    "factor"  "integer"  "integer"  "integer"  "numeric"  "numeric"  "numeric"
## Jitter_PPQ   Shim_loc    Shim_dB  Shim_APQ3  Shim_APQ5  Shi_APQ11      HNR05
##   "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##       HNR15      HNR25      HNR35      HNR38       RPDE        DFA        PPE
##   "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##         GNE     MFCC0      MFCC1      MFCC2      MFCC3      MFCC4      MFCC5
##   "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##       MFCC6      MFCC7      MFCC8      MFCC9     MFCC10     MFCC11     MFCC12
##   "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##      Delta0     Delta1     Delta2     Delta3     Delta4     Delta5     Delta6
##   "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##     Delta7     Delta8     Delta9    Delta10    Delta11    Delta12
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
```

Data Preprocessing

The first challenge we must correct is the problem of independence in our dataset. Our dataset has 240 recordings for 80 patients, so each patient has three recordings. For each patient then, those three recordings are not independent. Most prediction algorithms require independent data entries, so this is a problem we must correct before trying to predict PD in patients. Our solution to this problem is to combine the three data entries per patient into one data entry per patient. This will give us 80 independent data entries: 40 patients with PD, and 40 without PD.
Our dataset has 48 attributes. 44 of those attributes are numeric recording metrics. The next question is how to combine the values in each numeric column of the three dependent rows. We chose to take the mean of the three values. There are other possibilities for combining the values of each numeric column, but taking the mean is a simple and effective way of representing these three values.

```r
#Create an empty data frame with the same columns as our dependent data frame
pd.data.ind = pd.data[FALSE,]

#Get the patient ID's for all our patients.
patientIDs = levels(pd.data$ID)

# Now we get the rows that are dependent, and merge them into one row.
for(i in 1:length(patientIDs)){
  dependent.rows = pd.data[grep(patientIDs[i], pd.data$ID),]
  newRow = list()

  #We have our three dependent rows, we iterate over the columns and combine them into a single row.
  for(i in 1:ncol(dependent.rows)){
    #For the first few rows, we just need the first value. These are static
    if(i <= 4){
      newRow[i] = dependent.rows[1,i]
    }
    else{
      column = dependent.rows[,i]
      newRow[i] = mean(column)
    }
  }

  #Here we add our new, combined row, onto our independent dataset.
  names(newRow) = names(dependent.rows)
  pd.data.ind = rbind(pd.data.ind, newRow)
}

#Here we are changing our ID variable back into a factor and printing information about our dataset
pd.data.ind$ID = factor(pd.data.ind$ID, levels= pd.data.ind$ID, labels=c(patientIDs))
dim(pd.data.ind)
```

```
## [1] 80 48
```

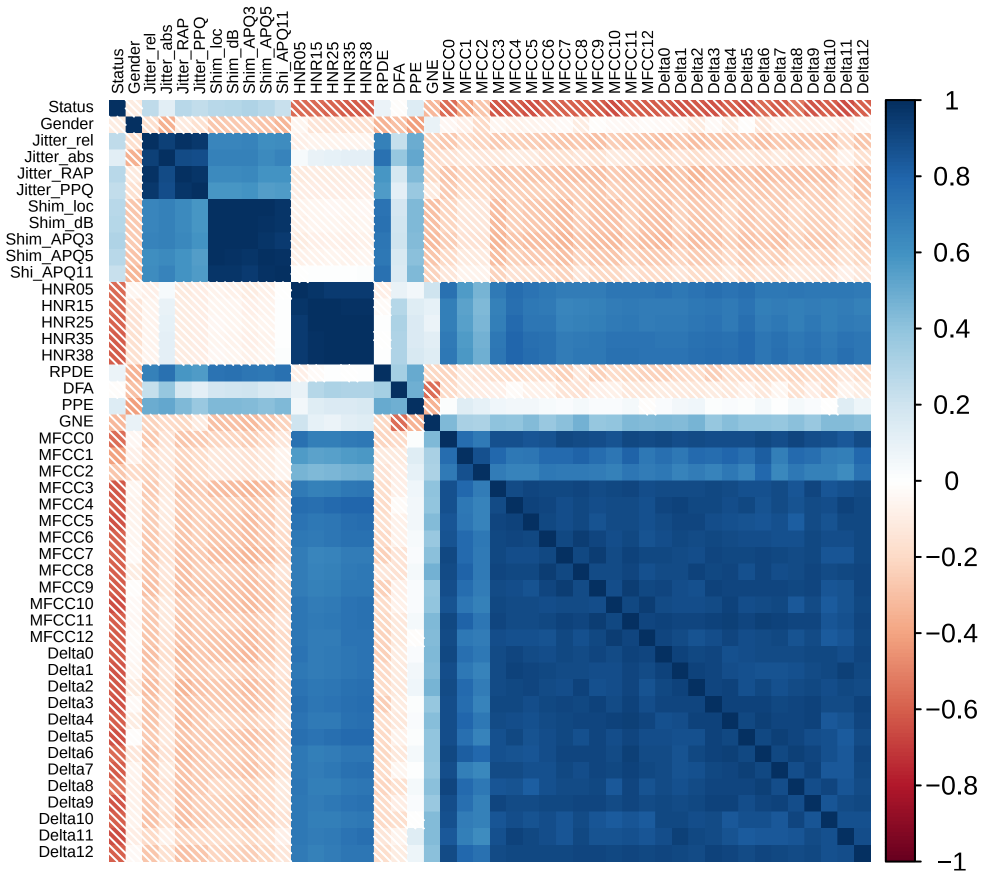
```r
sapply(pd.data.ind, class)
```

```
##          ID  Recording     Status     Gender Jitter_rel Jitter_abs Jitter_RAP
##    "factor"  "integer"  "integer"  "integer"  "numeric"  "numeric"  "numeric"
## Jitter_PPQ    Shim_loc    Shim_dB  Shim_APQ3  Shim_APQ5  Shi_APQ11      HNR05
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##      HNR15      HNR25      HNR35      HNR38       RPDE        DFA        PPE
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##        GNE     MFCC0      MFCC1      MFCC2      MFCC3      MFCC4      MFCC5
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##      MFCC6      MFCC7      MFCC8      MFCC9     MFCC10     MFCC11     MFCC12
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##     Delta0     Delta1     Delta2     Delta3     Delta4     Delta5     Delta6
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
##     Delta7     Delta8     Delta9    Delta10    Delta11    Delta12
##  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"  "numeric"
```

Correlation Analysis

You can see that we now have a data frame of 80 independent data recordings with all the same attributes as our dependent recordings. Now we can start developing a model of PD prediction. To start, we look at the correlation value of each attribute with PD presence (the 'Status' attribute). We will use Corrplot to get an initial look at these relationships. While these correlation values will not be used directly in our model, it will be helpful to see which values appear to have a linear relationship with a patient having PD or not. The most effective way to visualize this kind of relationship is with a heatmap.

```
corVals = cor(pd.data.ind[,-1:-2])
corrplot(corVals, method="shade", tl.cex=0.5,tl.col = "black")
```



This graph here gives us an idea of which recording metrics are linearly related to PD in a patient. We are focused on the top row of this chart which tells us the attributes that have a linear relationship with PD (Status). This chart shows us that MCF and Delta attributes have a negative linear relationship with PD, but it is hard to see exactly how strong that relationship is. The next step is to see how strongly related our attributes are to PD in a patient. We don't necessarily need to know the correlation value for every attribute, but knowing the extremes will give us an idea of the extent of the linear relationship.

```
# Find the most correlated attributes to Status (PD).
corrVals = c()
for(i in 4:ncol(pd.data.ind)){
  corrVals[i-3] = cor(pd.data.ind[,i], pd.data.ind[,3])
}

summary(corrVals)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.64598 -0.60621 -0.58436 -0.35712 -0.01343  0.30108
```

From these results, we see that our strongest linear relationships are negative. This also confirms what our corrplot chart indicated. Most of the relationships with PD are negatively linear. We can tell this because the median of our correlation values is very close to the minimum correlation value. We now have some grounds to say that PD and many of our dataset attributes are somewhat linearly related. The next step for our project is to test the hypothesis that our dataset attributes indicate PD in a patient. The null hypothesis here is that these attributes give no indication of PD in a patient.

Hypothesis Testing

Our explanatory variable in this hypothesis test is PD in a patient, and our response variables would be the voice recording metrics. Given that our explanatory variable is two-tiered and our response variables are quantitative, we can use a t-test to test this hypothesis for each of our dataset's attributes. Using the p-value from these tests, we can get an idea of how many of our attributes indicate PD in a patient.
While this data will not be used directly in our prediction model, it is an important step in confirming that the data we will use in our model truly does indicate PD in a patient. Without this test, we risk building a model with data that does not actually indicate PD. The hypothesis testing here is a necessary step in ensuring that our model is based on legitimate relationships in the data.

```
#this takes the index of the column you want and returns a p-value with the status attribute.
extractPVal = function(column){
  formula = paste(names(pd.data.ind)[column], " ~ Status")
  return(t.test(as.formula(formula), data = pd.data.ind)$p.value)
}

#We can then calculate the p-values between Status and the other attributes with Status as our explanat
pValues = rep(NA, ncol(pd.data.ind)-3)
pValues = sapply(4:ncol(pd.data.ind), extractPVal)
pValues = p.adjust(pValues, method="BH")

#This will print out the number of adjusted p-values that reject the null hypothesis.
print(paste("Number of significant adjusted p-values (<= 0.05):", sum(pValues <= 0.05)))
```

```
## [1] "Number of significant adjusted p-values (<= 0.05): 40"
```

```
print(paste("Number of non-significant adjusted p-values (> 0.05):", sum(pValues > 0.05)))
```

```
## [1] "Number of non-significant adjusted p-values (> 0.05): 5"
```

```
print(paste("Number of significant adjusted p-values (<= 0.01):", sum(pValues <= 0.01)))
```

```
## [1] "Number of significant adjusted p-values (<= 0.01): 32"
```

```
print(paste("Number of non-significant adjusted p-values (> 0.01):", sum(pValues > 0.01)))
```

```
## [1] "Number of non-significant adjusted p-values (> 0.01): 13"
```

To test our hypotheses, we performed multiple hypothesis testing with each attribute of our dataset against the presence of PD in a patient. We then adjusted the p-values from this process using the Benjamini-Hochberg procedure and set our significance value at 0.01. Our results indicate that we can reject the null hypothesis for 32/45 of our tests. With a significance of 0.05, we can reject the null hypothesis for 40/45 of our tests. This gives us strong reason to believe that most of our attributes indicate PD in a patient.
Given that we are using adjusted p-values, we also have an approximated false-positive rate of 1% or 5% for our hypothesis tests, both of which are very low. Again, while these data will not be used in our prediction model, it establishes that the data we want to use really does indicate PD in a patient with a reasonable level of confidence.
Based on the results from our hypothesis tests, it is now reasonable for us to try and predict PD using the data from this dataset. For our project, we have decided to do so using the RandomForest model. RandomForest is a popular, strong prediction model that is fairly simple to implement in R. We will use this model and our independent dataset to predict PD in patients.
We considered running a PCA on our dataset to reduce the number of attributes, but we only have 80 data entries. Even with 48 attributes in our dataset, the compute time should be relatively quick since we have so few data entries. For this reason, we are going to build our prediction model on our independent dataset, unmodified.

Prediction Model

Setting up our Random Forest model requires some processing of our dataset. The first thing is to change some of the datatypes of our attributes. These were fine as other datatypes in our previous analyses, but the random forest package has certain requirements for datatypes to function. We start by making those necessary changes, turning some attributes into factors. We have already imported the necessary packages at the beginning of our code. The RandomForest package is used to build our model. The caTools package is used to split our data into testing and training data. The caret package is used for our confusion matrix in evaluating the model. We also reset the random seed before building our model. Setting our random seed will allow us to replicate the same results every time we run this model. With these changes made, we are now ready to build our model.
The first step is to separate our dataset into test data and training data. We will explore later what the "optimal" split is, but for now, we chose an arbitrary split of 60% training and 40% testing data. Our random forest model will be built with the training data, and its predictions will be made using the testing data.

```r
# we need to make these variables factors for our prediction model
pd.data.ind$Status = as.factor(pd.data.ind$Status)
pd.data.ind$Gender = as.factor(pd.data.ind$Gender)

# In the pd.data.ind the Status variable must be a Label to use Classification. Otherwise the forest us

set.seed(42)#it seems that the random forest uses the random function so setting the seed keeps the res

#We need to split our independent records into training and testing data
sample = sample.split(pd.data.ind$Status, SplitRatio = 0.6)

#Creating the training/testing sets along our split.
train = subset(pd.data.ind, sample==TRUE)
test = subset(pd.data.ind, sample==FALSE)

#Shows the dimensions of our training set vs. our testing set.
dim(train)
```

```
## [1] 48 48
```

```r
dim(test)
```

```
## [1] 32 48
```

You can see that our training data uses 48 rows, and the testing data uses 32 rows from our split. After splitting our data, we are ready to build our random forest model. Doing so is fairly simple with the RandomForest package in R. You may notice that our formula is slightly different from the one we used in our t-tests. This is to accommodate the RandomForest package and to indicate that, for our model, we want to predict Status (PD) and not any of our other attributes. We also indicate here that we want our model to be built off the training data. From this training data, we tell our model to ignore the first two columns: ID and Recording. These attributes have nothing to do with predicting PD, so we do not want them to skew our results. With this section of code, we have built our random forest model.

```
#Now we create our RandomForest model using the data we just split off
# Our formula is different from our t-tests because here we are placing Status as the item to be predic
rf = randomForest(
  Status ~ .,
  #We ignore the ID and Recording attributes in our model.
  data = train[,-1:-2]
)

#prints out the model information
rf
```

```
##
## Call:
##  randomForest(formula = Status ~ ., data = train[, -1:-2])
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 6
##
##         OOB estimate of  error rate: 20.83%
## Confusion matrix:
##    0  1 class.error
## 0 19  5   0.2083333
## 1  5 19   0.2083333
```

From our random forest model, we can see that the estimated error rate is about 20%, and we also see a confusion matrix from the building of our model. Keep in mind that this model was built with a 60/40 split in our data, so a 20% estimated error rate is not so bad. After an initial evaluation, we will look to see how other splits in our data affect the model.
For now, we can use this model to predict the presence of PD in a patient, and we can look at the confusion matrix from this prediction to evaluate the effectiveness of our model. Predicting from our model is easy in R, we just call the predict method from the RandomForest package. We indicate in this prediction call the model we want to use, and the data to be used for this prediction. To see the results from our prediction, we use the caret confusionMatrix method. This method gives us a confusion matrix for our prediction as well as some statistics on our confusion matrix. A discussion of our prediction results is below.

Results and Evaluation

```
#Use our random forest model to predict Status.
#This prediction ignores Status and Recording in its prediction
pred = predict(rf, newdata = test[-1:-3])

#Print out a confusion matrix and evaluation information for our model.
confusionMatrix(pred, test[,3])
```

```
## Confusion Matrix and Statistics
##
```

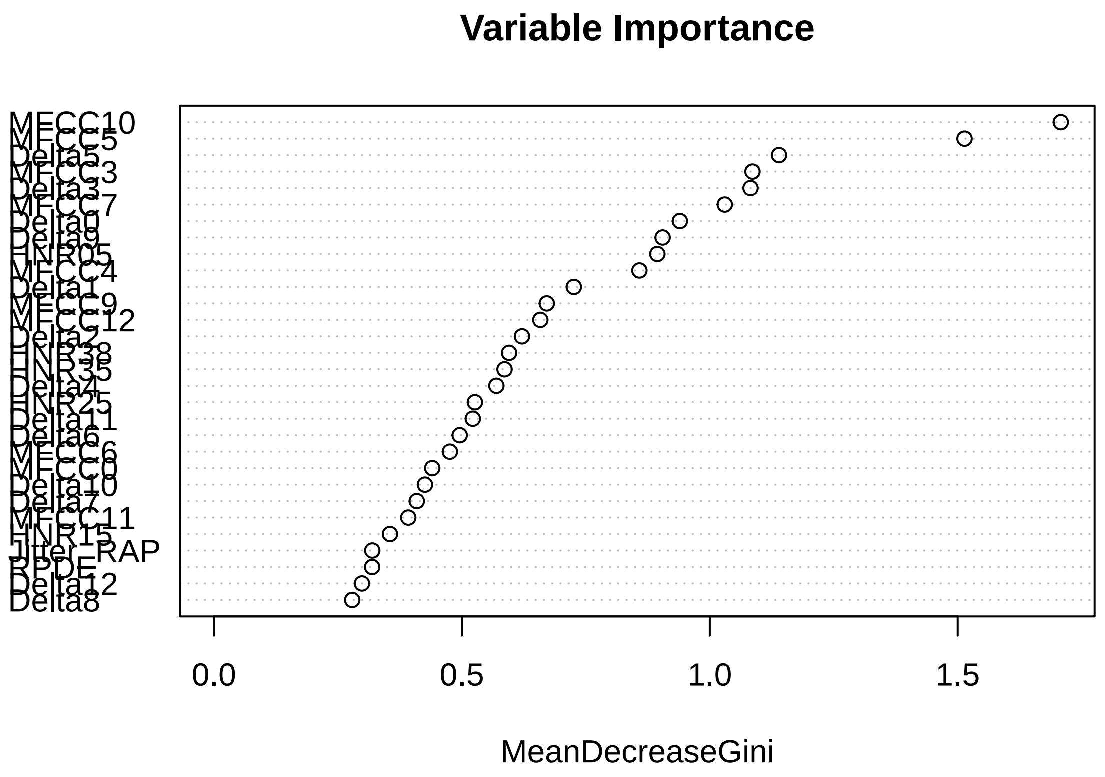
```
##           Reference
## Prediction  0  1
##          0 12  1
##          1  4 15
##
##                Accuracy : 0.8438
##                  95% CI : (0.6721, 0.9472)
##     No Information Rate : 0.5
##     P-Value [Acc > NIR] : 5.654e-05
##
##                   Kappa : 0.6875
##
##  Mcnemar's Test P-Value : 0.3711
##
##             Sensitivity : 0.7500
##             Specificity : 0.9375
##          Pos Pred Value : 0.9231
##          Neg Pred Value : 0.7895
##              Prevalence : 0.5000
##          Detection Rate : 0.3750
##    Detection Prevalence : 0.4062
##       Balanced Accuracy : 0.8438
##
##        'Positive' Class : 0
##
```

From the confusion matrix of our prediction, our model has 84% accuracy and a 0.69 kappa value. The P-value from this matrix is also significant. We can see from our matrix that we only had four false positives and one false negative out of 32 predictions. These are promising results, and it shows that we can predict PD in patients from these voice metrics with a fair measure of accuracy.

There are still some questions around our model that we would like to address. For one, what variables played the biggest role in these predictions? How would different test/train splits affect our results? After all, we want the size of our training data to be as small as possible without impacting accuracy. Right now, we have a 60/40 split between training and testing data. How small can we make the training data before we compromise our results? These are the questions that we will address in the final part of our project.

First, the question of examining which variables affect the PD prediction the most. We can do this fairly easily using the varImp caret function. This allows us to see the importance factor of every attribute in our dataset concerning our Random Forest model.

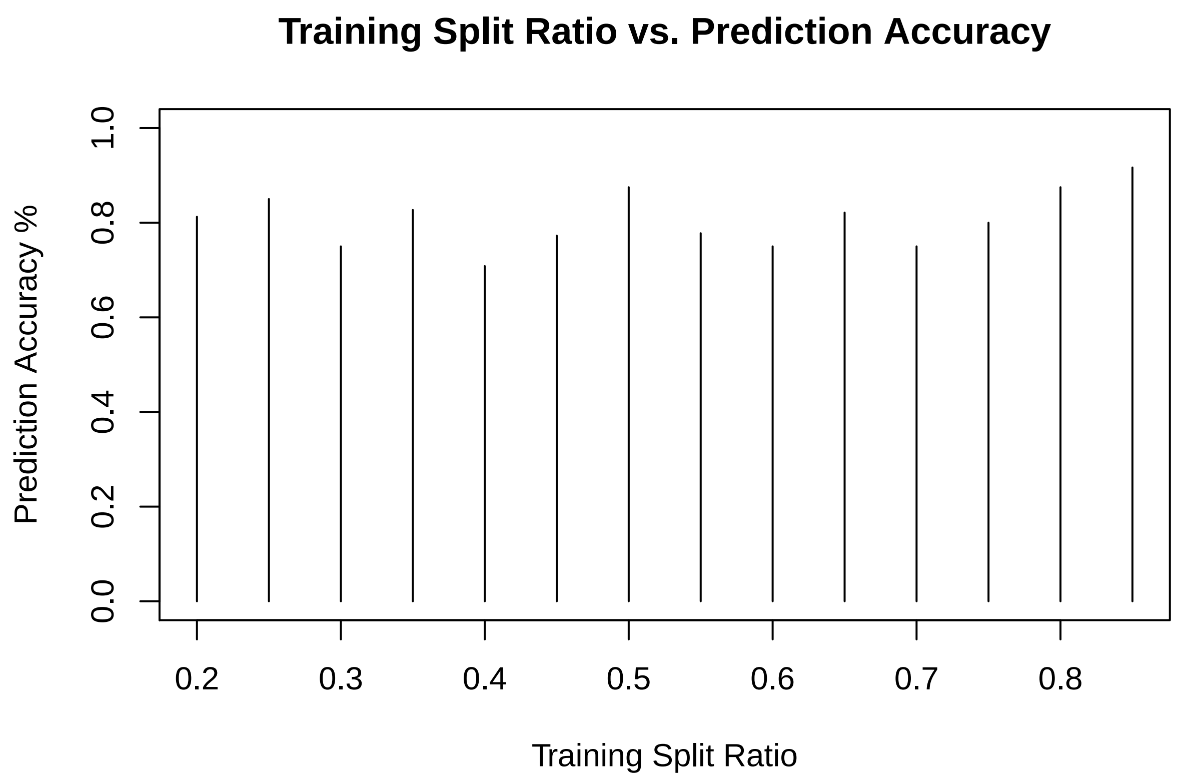
```
#Prints out a plot of variable importance for our random forest model.
varImpPlot(rf, main="Variable Importance")
```



This chart gives us an idea of which variables in our model have the biggest weight in determining PD in a patient. We can see that MFCC10 and MFCC5 carry much of the weight in our model. This chart provides some insight into what our Random Forest model is considering for its predictions. The next interesting question to consider is the optimal test/train split for our data. We arbitrarily chose a 60/40 split in our data, but how small can we make our training dataset before it starts to impact our results? We can make a chart to visualize the impact of different test/train sizes on our model.

```
# Create a graph showing different test/train sizes and accuracy.
#Reset our seed so we get the same result every time.
set.seed(42)
trainSizes = seq(from=0.2, to=0.85, by=0.05)
accuracyVals = rep(NA, length(trainSizes))

#Function to get the Accuracy value from a given training set size.
getAccuracyValue = function(trainSize){
  sample = sample.split(pd.data.ind$Status, SplitRatio = trainSize)

  #Creating the training/testing sets along our split.
  train = subset(pd.data.ind, sample==TRUE)
  test = subset(pd.data.ind, sample==FALSE)

  #creat random forest model
  rf = randomForest(
    Status ~ .,
    data = train[,-1:-2]
  )

  #make prediction and get accuracy from that prediction. Return accuracy value
  pred = predict(rf, newdata = test[-1:-3])
  accuracy = confusionMatrix(pred, test[,3])$overall[['Accuracy']]
```

```r
  return(accuracy)
}

# Get an accuracy value for every training size we want to consider.
accuracyVals = sapply(trainSizes, getAccuracyValue)

#Plot out our datapoints. Accuracy vs ratio
plot(x = trainSizes, y = accuracyVals,
     xlab = "Training Split Ratio", ylab = "Prediction Accuracy %",
     main = "Training Split Ratio vs. Prediction Accuracy",
     ylim = c(0,1),
     type = 'h')
```



```r
# See how much our accuracy varies based on training set sizes.
summary(accuracyVals)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.7083  0.7557  0.8063  0.8062  0.8442  0.9167
```

With this plot, we get an idea of how our training/testing split affects the accuracy of our prediction. This data would indicate that the ratio of test/train actually doesn't impact our predictions as much as one might think. In a range from 0.2-0.85 training ratio split, the accuracy of our predictions varies between 70-90% with a mean of 80%. Our chart indicates that even with very small training sets down to 20% of our data, the accuracy of our prediction remains about 80%. This would indicate that our data gives a good indication of PD regardless of sample size. Given how variable human voices are, detecting minute fluctuations in those voice patterns is a challenging task. An accuracy rate between 70-90% is good given that we only have 80 data entries to work with. Having more recording data to work with would likely make our predictions more accurate. Regardless, we have shown through these steps that we can successfully create a prediction model for PD in patients from metrics taken off voice recordings. While the accuracy of our predictions is not perfect, this method provides a quick, non-invasive PD test for asymptomatic patients.

Cross Validation

While the work above gives some indication as to the prediction accuracy of our model, it does so on a limited basis. It may give us a measure of our model's overall accuracy, but it does not give us an indication as to whether the predictions errors were distributed evenly between classes or not. It also does not tell us whether these errors were errors of inclusion or exclusion. One way for us to get a deeper understanding of our model is through a cross-validation analysis. This should indicate what kinds of errors our model is prone to. The test/train split analysis gives a surface level view of accuracy, but a cross-validation analysis will give a much better indication of the quality of this model.

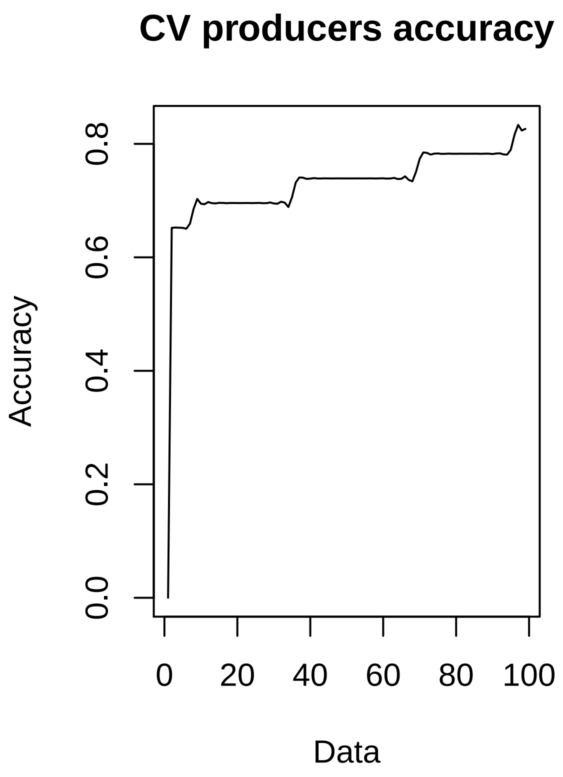
```
library(rfUtilities)
rf.cv=rf.crossValidation(rf,train[,-1:-2])
```

```
## running: classification cross-validation with 99 iterations
```

```
rf.cv
```

```
## Classification accuracy for cross-validation
##
##                        0   1
## users.accuracy       100 100
## producers.accuracy    NA  NA
##
## Cross-validation Kappa = 0.7391
## Cross-validation OOB Error = 0.1304348
## Cross-validation error variance = 0.001817975
##
##
## Classification accuracy for model
##
##                       0  1
## users.accuracy       87 87
## producers.accuracy 87 87
##
## Model Kappa = 0.7391
## Model OOB Error = 0.1304348
## Model error variance = 0.0004445325
```

```
par(mfrow=c(1,2))
     plot(
       rf.cv,
       type = "cv",
       main = "CV producers accuracy",
       ylab="Accuracy",
       xlab="Data"
       )
```

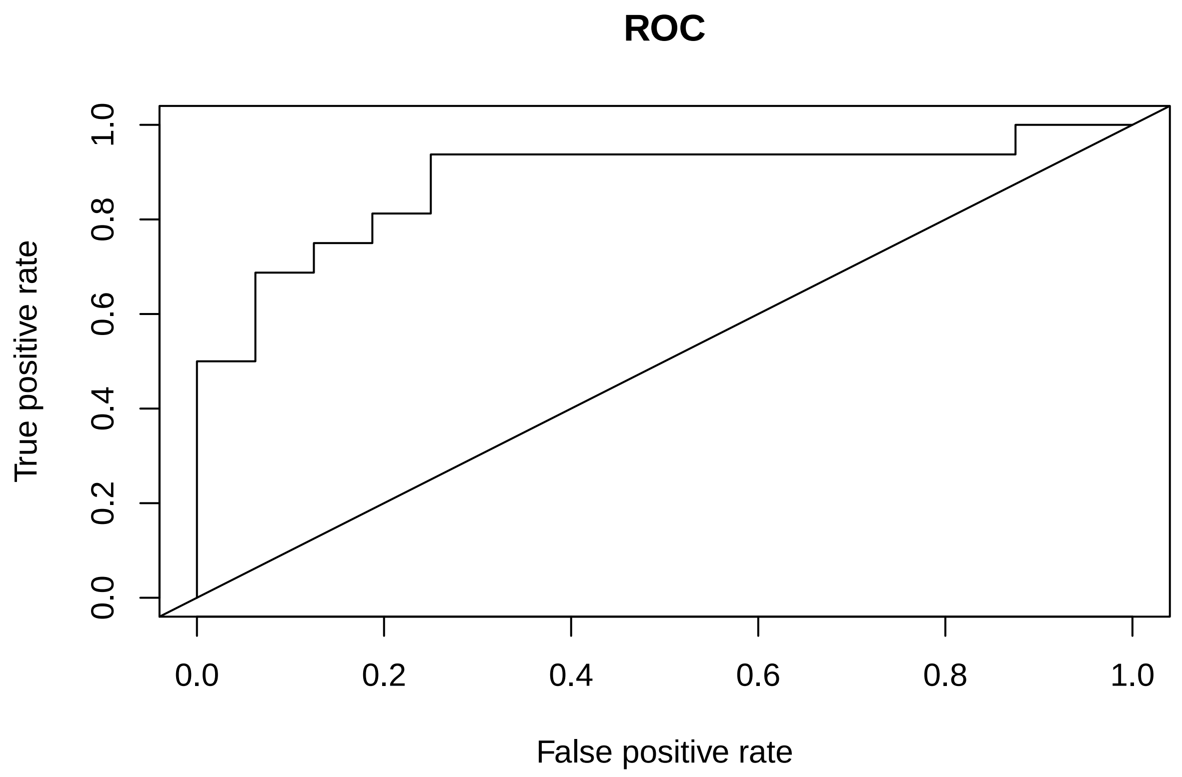


In the graph above, we see the producer accuracy for our cross-validated model. Producer accuracy is being determined by the errors of omission in our model. In a sense, we are validating the accuracy of our model by exposing it more data than just our initial testing dataset. Here, we can see that our CV accuracy is generally in the range that we saw before, centering around 80%. This would show that outside of our main testing set, our model is capable of producing similar levels of accuracy with different sets of data. This is important in evaluating the quality of our model in that we have shown our model can make good predictions with different, even unseen data.

We are trying to make predictions, so the question is how well is our model making those predictions? We have already seen some measures of this. The first being the confusion matrix. We saw from that matrix that our model's accuracy was about 85%. We then evaluated how our training/testing data split affected our model accuracy and found that the accuracy of our model centered around 80% regardless of how much data we used for the training dataset. This indicates that our model is strong given that we can still achieve a high level of accuracy with a small amount of training data. However, we want to evaluate our model using something other than accuracy. Accuracy can be misleading, so it might not tell the whole story in terms of model quality. We need another metric with which to evaluate our model.

One metric that we can use to evaluate our model is an ROC curve. This curve indicates our model's true positive rate against it's false positive rate. The perfect model follows the left and top side of the graph, so we want our model to be as close to that as possible. We use this ROC curve to see how well our model makes predictions. This plot will give us a model evaluation that does not rely on our accuracy measure. We can graph the ROC curve for our model using the ROCR package.

```r
library(ROCR)
set.seed(42)
roc.predict = predict(rf, type="prob", newdata = test[-1:-3])
forestpred = prediction(roc.predict[,2], test$Status)
forestperf = performance(forestpred, "tpr", "fpr")
plot(forestperf, main="ROC")
abline(a=0, b= 1)
```



The ROC curve for our graph looks pretty good. It isn't perfect, but it is closer to the upper left corner than it is to the diagonal line. This would indicate that our model reliably makes correct predictions about PD. We would like to have some number that points to what our graph shows. For an ROC curve, the number that indicates how well your model makes predictions is the Area Under the Curve (AUC). The greater the AUC value, the better your model is. We can calculate our AUC value using the same ROCR package.

```
auc.perf = performance(forestpred, measure = "auc")
auc.perf@y.values
```

```
## [[1]]
## [1] 0.8828125
```

The perfect AUC value is 1, so we want to be as close to that value as possible. We can see from our calculation above that our AUC value is about 0.88 which is good. This is just a numeric representation of what our ROC curve is telling us. Our model isn't perfect, but it reasonably predicts PD in a patient with minimal errors. With this, we have evaluated our model on several fronts using both accuracy and an ROC curve. These are commonly used metrics to evaluate a model, and both indicate that our model provides reasonably correct predictions. We could spend years fine-tuning our model, but time limits the scale of this project. For now, we are accepting 85% accuracy and an 0.88 ROC AUC value as these indicate that our model is strong as it stands.

Conclusions and Future Work

With this work, we have shown that we can correctly predict Parkinsons Disease in a patient using data calculated from patient voice recordings. We first showed that the voice data was indeed related to Parkinson's Disease using correlation values and multiple hypothesis testing. After showing that the data was related, we then build a random forest model that predicted PD in a patient. We showed that the Accuracy of our model varied only slightly with different training/testing data splits. We then used metrics like Accuracy, Kappa Value, and an ROC curve to show that our model was reliably making the correct prediction of PD in a patient. With this, we have constructed a reasonable body of knowledge surrounding PD predictions that confirms what Naranjo proposed in her research paper. In terms of future work, we would like to be able to make these voice recording calculations using our own method. The actual voice recordings were not provided with this dataset, so we have no way of knowing how these recording metrics were calculated. It would be interesting to see if we can develop our own method of analyzing voice recording files. It would also be worth trying to improve the quality of our prediction model. Accuracy of 85% is good, but there are certainly methods that we can explore to improve on those results. For one, we could explore reducing the dimensionality of our dataset in an attempt to improve our results. Finally, the quality of this project would be greatly improved by having a larger dataset to study with. Having only 80 data entries really limits the results of any model we build. For this project to grow, it would be necessary to have additional patients give voice data that we could include in our model.