Technical Appendix

Jonathan Che and David Green

16 December 2016

Getting the Data

We found data at data.gov on the "Most Popular Baby Names by Sex and Mother's Ethnic Group, New York City". It was contained in .csv form, with variables for child name, year of birth, child gender, mother's race/ethnicity, number of children (of that name/ethnicity/gender/year), and popularity rank of the name (for that ethnicity/gender/year). (For more information about the variables, see the preliminary analysis turned in).

After downloading the file as `babynames.csv`, we ran the following code to fix the inconsistent labelling of some race/ethnicity:

```
df <- read_csv("babynames.csv")

df2 <- df %>%
  mutate(ETHCTY=ifelse(ETHCTY=="ASIAN AND PACI", "ASIAN AND PACIFIC ISLANDER", ETHCTY)) %>%
  mutate(ETHCTY=ifelse(ETHCTY=="BLACK NON HISP", "BLACK NON HISPANIC", ETHCTY)) %>%
  mutate(ETHCTY=ifelse(ETHCTY=="WHITE NON HISP", "WHITE NON HISPANIC", ETHCTY)) %>%
  distinct()

write_csv(df2, path="~/Baby-Names/babynames2.csv")
```

We then fed the code into a python script. Since this is a .Rmd, I won't include that part here (the script will be attached separately as `parsenames.py`). The script extracted a variety of characteristics from the data, including first/last letter, name length, number of syllables in the name, vowel/consonant counts/proportions, indicators for double letters/vowels/consonants, and the number of each letter (again, these are described in more detail in the preliminary analysis turned in, though vowel/consonant counts/proportions were added since then). One interesting thing to note is the syllable counter. We pulled in syllable counting script for English words from here. While it did not get exact syllable counts for all of the names (it is right about 50-75% of the time), we thought that it served as a decent proxy for syllable count. As it turned out, syllable count wasn't too important, but it if were, we would have done more examination into determining a better syllable counting algorithm.

In this file, the python script output is saved as `babynames4.csv`. The file can be downloaded from Jonathan's github page if you'd like to run the code in this file.

```
df <- read_csv("babynames4.csv")
```

After loading in the overall data, we did some more cleaning. We cleaned data in two different ways: unweighted and weighted. For our unweighted cleaning, we just read in the data as it was formatted originally, with one observation per child name. For our weighted cleaning, we repeated names according to their popularity, so it would be one observation per child. We used our unweighted data to explore names, and our weighted data to actually classify names.

We felt that this split was appropriate for a number of reasons. When we are merely interested in exploring the features of names, we want all names to get equal representation in our analyses. When we are training a classifier, we want it to be practically useful, so actual representation of names is important.

Here, we'll run the unweighted data cleaning code first (the weighted cleaning is just displayed, not run) before we go into our preliminary analyses.

```r
# Unweighted data cleaning

# Making factors (as appropriate)
df2 <- data.frame(lapply(df[,1:3], function(x) as.factor(x)),
                  df[,4:6],
                  lapply(df[,7:8], function(x) as.factor(x)),
                  df[,9:10],
                  lapply(df[,11:13], function(x) as.logical(x)),
                  df[14:44])
df3 <- df2 %>%   # Deselecting unused variables
  select(-count, -rank, -name)
df2011 <- df3 %>%
  filter(year == "2011")
df2011a <- df2011 %>%   # Only numerical variables, no letter counts
  select(gender, name_length, vowel_consonant_prop, double_letter, double_vowel,
         double_consonant, num_syllables, num_consonants, num_vowels,
         consonant_prop, vowel_prop)
df2011b <- df2011[,12:37]   # Only letter counts
```

```r
# Weighted data cleaning

# Making factors (as appropriate)
df2 <- data.frame(lapply(df[,1:3], function(x) as.factor(x)),
                  df[,4:6],
                  lapply(df[,7:8], function(x) as.factor(x)),
                  df[,9:10],
                  lapply(df[,11:13], function(x) as.logical(x)),
                  df[14:44])
df2.rep <- df2[rep(seq(dim(df2)[1]), df2$count), ]   # WE REPEAT THE DATA BY COUNT
df2.rep.samp <- df2.rep %>%
  sample_n(10000, replace=TRUE)   # Sample too large after repetition, so we sample back down
df3 <- df2.rep.samp %>%
  select(-count, -rank, -name)
df2011 <- df3 %>%
  filter(year == "2011")
df2011a <- df2011 %>%   # Only numerical variables, no letter counts
  select(gender, name_length, vowel_consonant_prop, double_letter, double_vowel,
         double_consonant, num_syllables, num_consonants, num_vowels,
         consonant_prop, vowel_prop)
df2011b <- df2011[,12:37]   # Only letter counts
```

Preliminary analyses

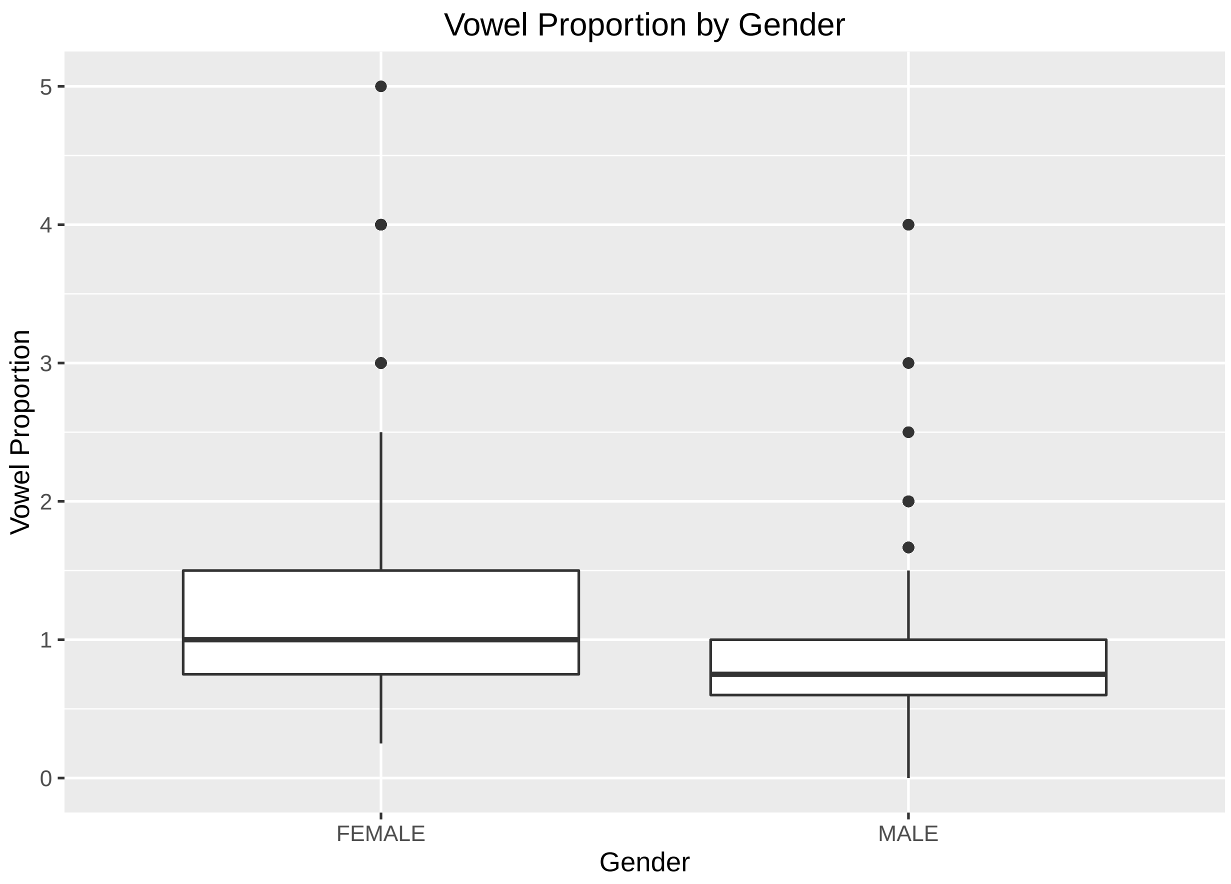
Right off the bat, some analyses were not available to us. Most of our variables were either factors or highly discrete. As such, techniques that are more focused on numerical variables (such as PCA) would not be highly applicable.

We were less interested in finding underlying factors explaining our data than we were in simply finding the differences between gender/race in our data, so we turned to visualizations and classification over factor analysis and clustering (though we did try some clustering).

Visualizations

We pretty much did an exhaustive search for interesting relationships in our data by making many 1-2 variable boxplots/scatterplots/barplots/tables, depending on the types of variables being compared. I've included two interesting findings below.

```r
ggplot(df3, aes(x=gender, y=vowel_consonant_prop)) +
  geom_boxplot() +
  labs(title="Vowel Proportion by Gender",
       y="Vowel Proportion",
       x="Gender")
```



This parallel boxplot shows that female names tend to have a greater proportion of vowels, compared to consonants, than male names. It suggests that there could be separability between genders based on specific letters, or based on vowel/consonant use.

```r
tally(df2011$race ~ df2011$z_count, format="proportion")
```

```
##                               df2011$z_count
## df2011$race                            0         1
##   ASIAN AND PACIFIC ISLANDER 0.1565309 0.1527778
##   BLACK NON HISPANIC         0.1951348 0.2083333
##   HISPANIC                   0.3178213 0.2361111
##   WHITE NON HISPANIC         0.3305130 0.4027778
```

...

Clustering

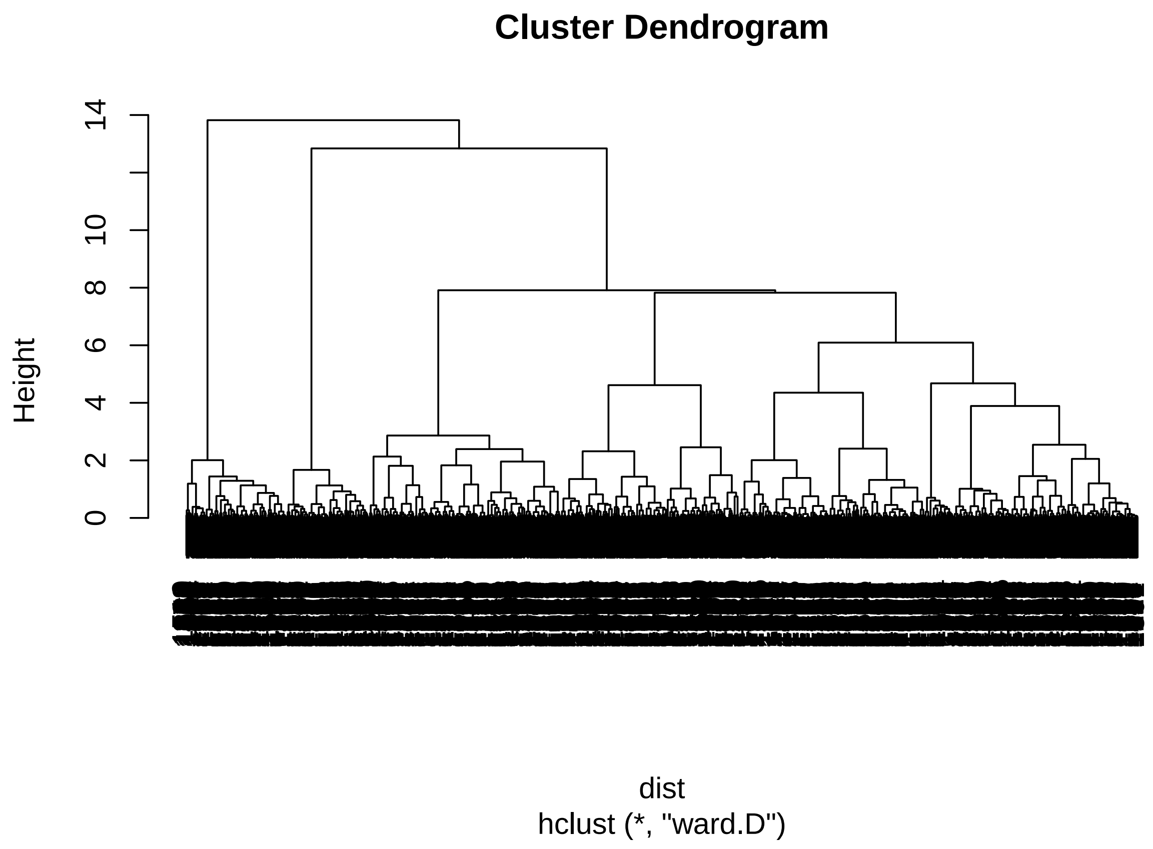
We tried to perform some cluster analyses on our data. Cluster analysis is very sensitive to highly correlated variables, and many of our variables are either computed directly from each other or otherwise highly correlated for obvious reasons. We tried to cluster on two different sets of variables; one with only (vowel_prop, double_letter, name_length, num_syllables), and one with only the letter counts. Since clustering was not extremely interesting overall, I'll just include one cluster example we did with the letter counts.

We chose to cluster on a single year in order to try to keep results comparable with classification on a single year. Also, we wanted to see if clusters would separate based on gender or race before adding in the year as another variable to examine.

```
# Takes a long time to run, messy output, no significant correlations
# cor(df2011b, method="kendall")
```

After checking for outliers in our data (and not finding any), we proceeded with heirarchical clustering based on Gower's distances, since the variables were effectively ordinal categoricals, being wary of the fact that some of the variables were only binary (0 or 1). We found Ward linkages to give the clearest cluster distinctions, so we used Ward's.

```
dist = daisy(df2011b, metric="gower", stand=T)
hcward = hclust(dist, method="ward.D")
plot(hcward)
```



We see three clear clusters emerge, but since one of the clusters would be so much larger than the others, we use 6 clusters, since the 6-cluster solution is also pretty distinct.

```r
hcward.sol=as.factor(cutree(hcward, k = 6))
summary(hcward.sol)
```

```
##   1   2   3   4   5   6 
## 394 382 439 208 170 370
```

With 6 clusters, the cluster sizes are quite consistent. We now check to see if any of the clusters are significantly linked to gender or race. We start with gender.

```r
bar <- data.frame(df2011, hcward.sol)
round(tally(gender~hcward.sol, data=bar, format="proportion"), 2)
```

```
##         hcward.sol
## gender      1    2    3    4    5    6
##   FEMALE 0.49 0.51 0.62 0.36 0.56 0.48
##   MALE   0.51 0.49 0.38 0.64 0.44 0.52
```

Clearly, the clusters are not based on gender. Cluster 3 appears slightly more male and cluster 4 appears slightly more female, though it's not really significant enough to be interesting. Next we check race.

```r
round(tally(race~hcward.sol, data=bar, format="proportion"), 2)
```

```
##                             hcward.sol
## race                            1    2    3    4    5    6
##   ASIAN AND PACIFIC ISLANDER 0.14 0.19 0.16 0.18 0.13 0.14
##   BLACK NON HISPANIC         0.19 0.17 0.16 0.22 0.27 0.22
##   HISPANIC                   0.33 0.30 0.29 0.43 0.32 0.27
##   WHITE NON HISPANIC         0.35 0.34 0.39 0.17 0.28 0.36
```

The clusters are also not significantly based on race. Cluster 4 is slightly more Hispanic and less White Non-Hispanic, but again, not extremely so. Further examinations of parallel boxplots, tallies, and other visuals did not give us any intuitive groupings of the data.

In general, we didn't consider our clustering results because of the nature of our data. In our dataset, we computed various simple characteristics of names. In general, though, names are highly cultural constructs. As such, in our cluster solutions, we would only be interested in seeing whether clustering based on simple characteristics would result in groupings based on the cultural ideas of gender or race. We weren't very interesting in natural groupings of names based on their letters, for example, if it didn't reveal something more about the social implications.

Classification

Since an unsupervised approach was unsuccessful, we moved onto the supervised approach of classification. Initially, we classified based on the unweighted dataset. We realized, though, that it would make more sense to have our classifiers weigh popular names more heavily if they were to be of any practical use.

Also, we decided to classify based on gender rather than race. This decision came from the fact that we were able to find more distinguishing features between genders than between races in our exploratory analyses, as well as from the fact that initial classifiers performed much better when classifying based on gender than based on race.

```r
# Weighted data cleaning

# Making factors (as appropriate)
df2 <- data.frame(lapply(df[,1:3], function(x) as.factor(x)),
                  df[,4:6],
                  lapply(df[,7:8], function(x) as.factor(x)),
                  df[,9:10],
                  lapply(df[,11:13], function(x) as.logical(x)),
                  df[14:44])
df2.rep <- df2[rep(seq(dim(df2)[1]), df2$count), ]   # WE REPEAT THE DATA BY COUNT
df2.rep.samp <- df2.rep %>%
  sample_n(10000, replace=TRUE)   # Sample too large after repetition, so we sample back down
df3 <- df2.rep.samp %>%
  select(-count, -rank, -name)
df2011 <- df3 %>%
  filter(year == "2011")
df2011a <- df2011 %>%   # Only numerical variables, no letter counts
  select(gender, name_length, vowel_consonant_prop, double_letter, double_vowel,
         double_consonant, num_syllables, num_consonants, num_vowels,
         consonant_prop, vowel_prop)
df2011b <- df2011[,12:37]   # Only letter counts
```

We tried out all of the classification methods we implemented in class (excluding LDA/QDA, since we had many non-numerical variables with very non-normal distributions). Again, we note that there aren't any significant outliers. We do have highly correlated variables, but trees are highly robust to correlated variables. Nearest neighbors should not suffer too much correlations either, since we know that the correlations between variables will be consistent for our training and testing data (since it's 'baked in' to the variables, so to speak; vowel proportion will always be correlated with consonant proportion, for example). With these considerations in mind, we proceed with classification.

Simple Tree

We start by training a tree on a smaller subset of the variables, not including first/last letter and letter counts.

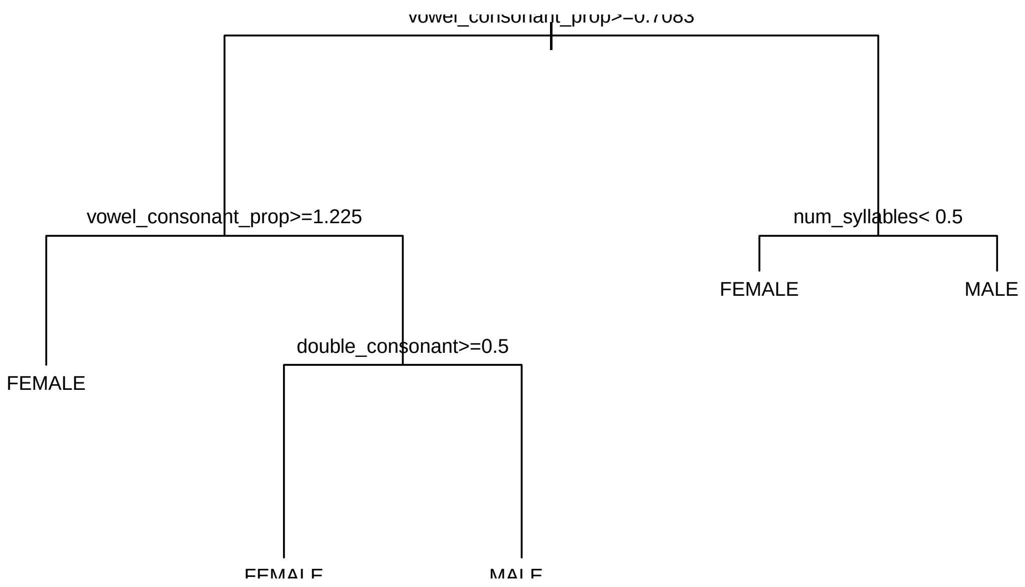
```r
bf.control=rpart.control(minsplit=100,minbucket=20,xval=5)
bf.treeorig=rpart(gender ~.,data=df2011a,method="class",control=bf.control)
printcp(bf.treeorig)
```

```
##
## Classification tree:
## rpart(formula = gender ~ ., data = df2011a, method = "class",
##     control = bf.control)
##
## Variables actually used in tree construction:
## [1] double_consonant     num_syllables        vowel_consonant_prop
##
## Root node error: 1072/2504 = 0.42812
##
## n= 2504
##
##         CP nsplit rel error  xerror     xstd
## 1 0.123134      0   1.00000 1.00000 0.023097
## 2 0.098881      1   0.87687 0.88526 0.022646
## 3 0.021455      3   0.67910 0.68657 0.021265
## 4 0.010000      4   0.65765 0.67817 0.021188
```

Our tree solution is not particularly effective, even after some tuning of the minsplit/minbucket parameters. We have an estimated TER of approximately 30%, which is not great given that random guessing would get us to 50%.

Trees are useful because they are highly interpretable, so we plot out this tree.

```
plot(bf.treeorig)
text(bf.treeorig,cex=.7)
```



There is nothing too insightful in the tree, though there are some interesting confirmations of things we knew. We recognize that vowel/consonant proportion is important (female names have more vowels), as we saw before. Also, it seems that females tend to have longer names, which we also saw before.

We then train the tree on all of the variables.

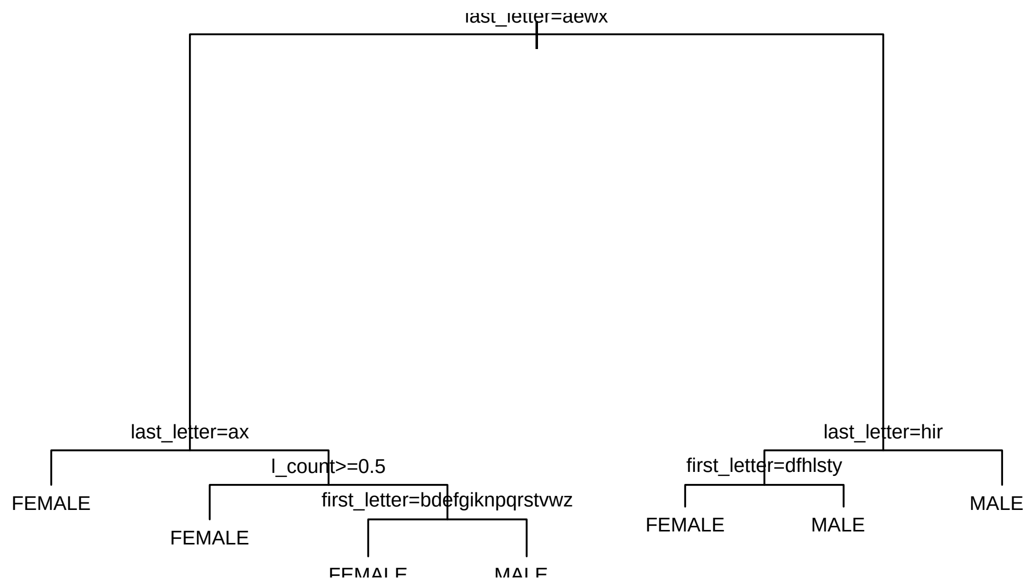
```
bf.control=rpart.control(minsplit=100,minbucket=20,xval=5)
bf.treeorig=rpart(gender ~.,data=df2011,method="class",control=bf.control)
printcp(bf.treeorig)
```

```
##
```

```
## Classification tree:
## rpart(formula = gender ~ ., data = df2011, method = "class",
##     control = bf.control)
##
## Variables actually used in tree construction:
## [1] first_letter l_count      last_letter
##
## Root node error: 1072/2504 = 0.42812
##
## n= 2504
##
##         CP nsplit rel error  xerror     xstd
## 1 0.587687      0   1.00000 1.00000 0.023097
## 2 0.020833      1   0.41231 0.41698 0.017876
## 3 0.015392      4   0.34981 0.37687 0.017171
## 4 0.010000      6   0.31903 0.34701 0.016602
```

Using all of the variables more than halves the estimated TER, which is a good sign. We examine the tree diagram.

```
plot(bf.treeorig)
text(bf.treeorig,cex=.7)
```



Interestingly, it seems that the first and last letter of names are very important. The results of this tree, though, are not at all interpretable. As such, we think that 'black-box' methods such as the random forest or nearest neighbors might be more useful, since they're typically stronger and won't be any less interpretable.

Random Forest

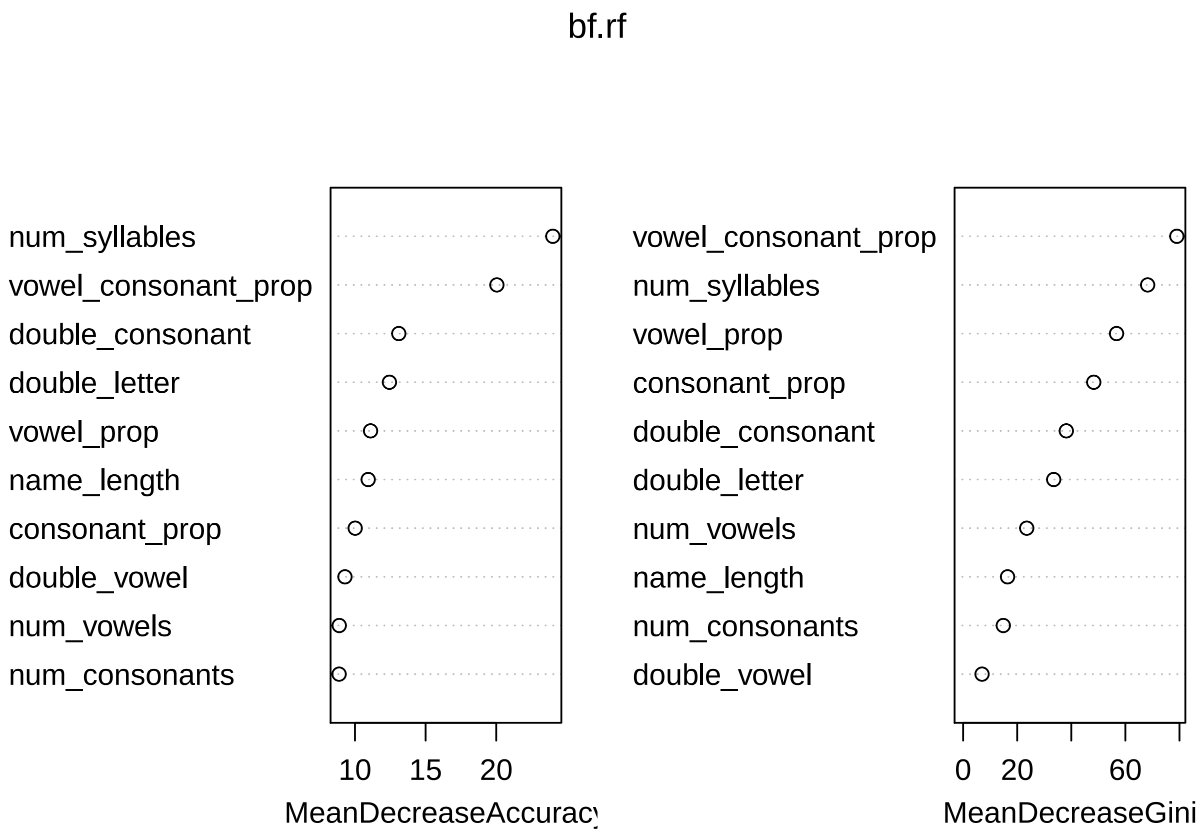
We began by fitting and tuning simple models.

```
set.seed(40)
bf.rf=randomForest(gender ~., data=df2011a,mtry=3,ntree=100,importance=T,proximity=T)
bf.rf
```

```
##
## Call:
##  randomForest(formula = gender ~ ., data = df2011a, mtry = 3,      ntree = 100, importance = T, prox
##                Type of random forest: classification
##                      Number of trees: 100
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 26.68%
## Confusion matrix:
##        FEMALE MALE class.error
## FEMALE    601  471   0.4393657
## MALE      197 1235   0.1375698
```

With only a few variables, we achieve an OOB error estimate of about 25%.

```
varImpPlot(bf.rf)
```



Vowel/Consonant proportion and number of syllables are the most important variables here, as it was for the simple tree.

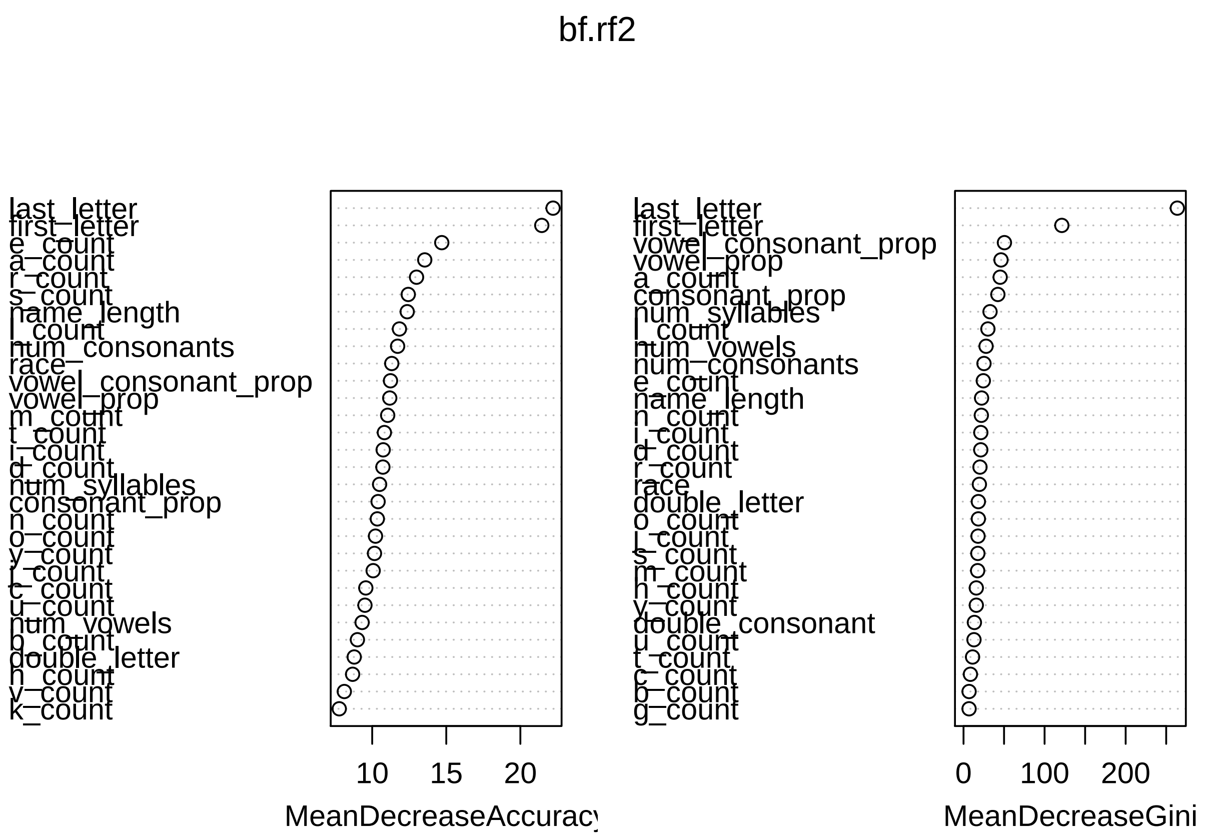
Next, we use all of the variables.

```r
set.seed(34)
bf.rf2=randomForest(gender ~., data=df2011,mtry=3,ntree=100,importance=T,proximity=T)
bf.rf2
```

```
##
## Call:
##  randomForest(formula = gender ~ ., data = df2011, mtry = 3, ntree = 100,      importance = T, proxi
##                Type of random forest: classification
##                      Number of trees: 100
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 3.59%
## Confusion matrix:
##        FEMALE MALE class.error
## FEMALE   1011   61  0.05690299
## MALE       29 1403  0.02025140
```

With all of the variables, the OOB error estimate to around 5%.

```r
varImpPlot(bf.rf2)
```



Again, like in the simple tree, the first and last letter matter now.

NN Method

We run NN on the numerical variables.

```
bf.knn=knn(df2011a[,-1],df2011a[,-1],df2011[,"gender"],k=3,prob=T)
table(df2011a$gender,bf.knn)
```

```
##         bf.knn
##          FEMALE MALE
##   FEMALE    655  417
##   MALE      216 1216
```

> The nearest neighbor method gives similar results to the random forest (with the numerical variables only). Since it is not as flexible as the random forest for binary variables and categorical variables, we decide to just stick with the random forests moving forward.

XGBoost

XGBoost uses sparse matrix representations of training/testing data. Luckily, it has some easy functions to produce the required data format, if we just give it the data split up into a vector of labels and a matrix of observations. We never went through the trouble of training and testing the XGBoost on the same year, since it would involve some extra train/test set sampling code and we were reasonably confident that it would perform well on the time study section of our project (described in the last section). In any case, this is what the code to produce the training set would look like. `train2011.final` is a fixed-up version of our training set (this fixing will be described later).

```
xgb.train.label <- ifelse(train2011.final[,1]=="FEMALE",0,1)
xgb.train.data <- data.matrix(train2011.final[,-1])
dtrain <- xgb.DMatrix(data = xgb.train.data, label = xgb.train.label)
```

XGBoost has a huge number of tunable hyperparameters. For this project, we didn't have the time to perform any very exhaustive gridsearches to optimize hyperparameters, so we chose relatively standard parameters (with some minimal tuning). In other words, the XGBoost results could most likely be improved with further model tuning.

```
bst <- xgb.train(data = dtrain, max.depth = 6, eta = 0.1, nthread = 2, nround = 100, nfold=5,
                 objective = "binary:logistic",
                 watchlist=list(train=dtrain, test=dtest),
                 verbose=TRUE)
```

Due to some difficulties with factor levels, I don't run the code here. I'll run it all at once when we do the time study in the next section. In any case, XGBoost gave promising initial results with our data.

As far as feature importance goes, we would run the code below. I'll show the output at the end of this file.

```
importance_matrix <- xgb.importance(names(train2011.final)[-1], model = bst)
xgb.plot.importance(importance_matrix[1:10,]) +
  ggtitle("Variable Importance Plot") +
  theme(title=element_text(size=18),
        legend.title=element_text(size=16),
        legend.text=element_text(size=12),
        axis.text=element_text(size=12),
        axis.title=element_text(size=16,face="bold"))
```

Time study

Producing test data

First, we need to produce all of the testing datasets that we want to use. The testdataXXXX.csv files are all on Jonathan's github page if you'd like to run the code here. The testdataXXXX.csv files are produced by the `parsenames_testdata.py` file on Jonathan's github, which is essentially the same code as the original `parsenames.py`.

We fed in slightly modified versions of the text files found on the Social Security Administration's website for each year of baby names. For each text file, we took only the names with >1000 occurrences in the data. We then divided the count for each name by 1000, resulting in a scaled-down version of the most popular names in the US. In this way, we would have a usable sample size of names (not too many, not too few) after we weighed the names by their counts. We used 500 as our cutoff for 1900 rather than 1000 due to small sample sizes before the 1910s.

The parsing script gives the test data all of the relevant variables we considered for our training data. We load in the test data below.

```r
test2012 <- read_csv("testdata2012.csv")
test2005 <- read_csv("testdata2005.csv")
test2000 <- read_csv("testdata2000.csv")
test1995 <- read_csv("testdata1995.csv")
test1980 <- read_csv("testdata1980.csv")
test1960 <- read_csv("testdata1960.csv")
test1940 <- read_csv("testdata1940.csv")
test1920 <- read_csv("testdata1920.csv")
test1900 <- read_csv("testdata1900.csv")
```

We build a function to do some final cleaning of each year's data.

The factor issue that we ran into had to do with the fact that some samples of data did not have all 26 letters as factor levels for first_letter or last_letter. Using the `levels()` command after loading in the data didn't work, because if, say, only 20 letters were used, the `levels()` command would only fill in the first 20 of the 26 factor levels you gave it, rather than matching the appropriate factor levels and leaving unfilled factor levels as 0.

After a lot of searching online, we decided that it would just be easiest to do a very quick and dirty method to fix the problem. We added 26 dummy names to the data, one for each letter of the alphabet (for first_letter and last_letter). Then, we'd factor the data and remove the dummy names. This way, we'd have matching factor levels for all of our training and testing data without too much hassle.

```r
clean_test_data <- function(df){
  # Sloppy code, but it gets the job done
  # Add 26 dummy names to data to ensure factor correctness
  for (letter in levels(train2011.final$first_letter)){
    df <- rbind(df, c("Foo","F",1,1,letter,letter,0,0,FALSE,FALSE,0,0,0,0,0,0,0,0,0,
                      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0))
    }
  df <- data.frame(df[,1],
                   lapply(df[,2], function(x) as.factor(x)),
                   df[,3:4],
                   lapply(df[,5:6], function(x) as.factor(x)),
                   df[,7:8],
                   lapply(df[,9:11], function(x) as.logical(x)),
                    df[12:42])
  df <- df %>%
    filter(name != "Foo")
  df.rep <- df[rep(seq(dim(df)[1]), df$count_norm), ]   # WE REPEAT THE DATA BY COUNT
  df.final <- df.rep %>%
    select(-count_norm, -name, -count) %>%
    mutate(gender = ifelse(gender=="M", "MALE", "FEMALE")) %>%
    mutate(gender = as.factor(gender))
    return(df.final)
}
```

Obviously, we still need to clean all of the test data.

```
test2012.final <- clean_test_data(test2012)
test2005.final <- clean_test_data(test2005)
test2000.final <- clean_test_data(test2000)
test1995.final <- clean_test_data(test1995)
test1980.final <- clean_test_data(test1980)
test1960.final <- clean_test_data(test1960)
test1940.final <- clean_test_data(test1940)
test1920.final <- clean_test_data(test1920)
test1900.final <- clean_test_data(test1900)
```

We also clean the 2011 training data (to fix the factor problem), which code I will evaluate but not display here. It outputs training data as a `train2011.final` data frame.

Random Forest

We train a random forest model on the 2011 training data.

```
set.seed(50)
rf_2011=randomForest(gender ~., data=train2011.final,mtry=10,ntree=100,importance=T,proximity=T)
```

We build a function that allows us to analyze a given year. It returns a vector of the confusion matrix [in the order (1,1), (2,1), (1,2), (2,2)], overall error rate, error rate for males (misclassified as females), and error rate for females.

```
analyze_year <- function(year){
  df <- get(paste("test", year, ".final", sep=""))
  table_name <- paste("table", year, sep="")
  error <- paste("error", year, sep="")
  errorm <- paste("error", year, "_m", sep="")
  errorf <- paste("error", year, "_f", sep="")

  assign(table_name, table(df$gender,predict(rf_2011, df)))
  table <- get(table_name)
  assign(error, (table[1,2]+table[2,1])/sum(table))
  assign(errorm, (table[2,1])/sum(table))
  assign(errorf, (table[1,2])/sum(table))

  return(c(table, get(error), get(errorm), get(errorf)))
}
```

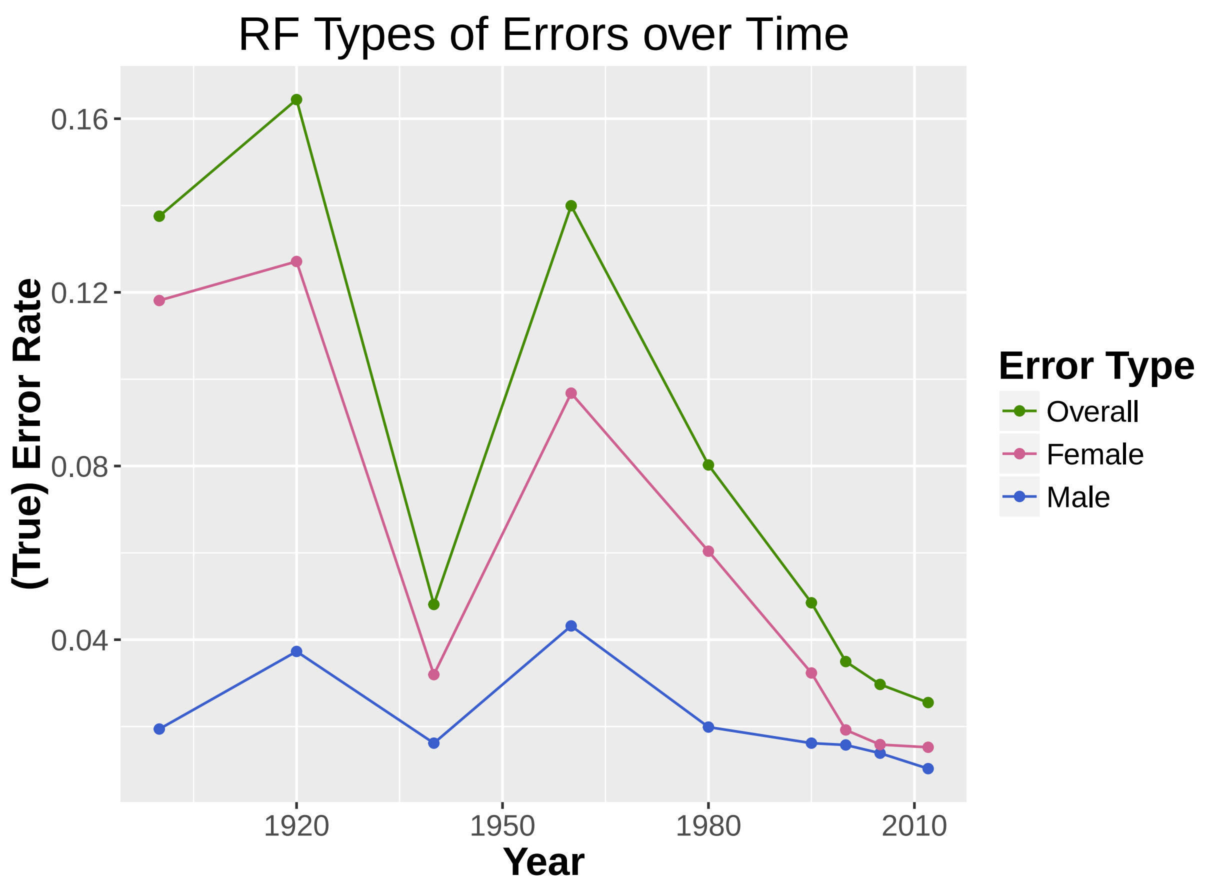
We run the function on each year.

```
info2012 <- analyze_year(2012)
info2005 <- analyze_year(2005)
info2000 <- analyze_year(2000)
info1995 <- analyze_year(1995)
info1980 <- analyze_year(1980)
info1960 <- analyze_year(1960)
info1940 <- analyze_year(1995)
info1920 <- analyze_year(1920)
info1900 <- analyze_year(1900)
# rbind(info1900[c(1,3)], info1900[c(2,4)])   If confusion matrix is desired
```

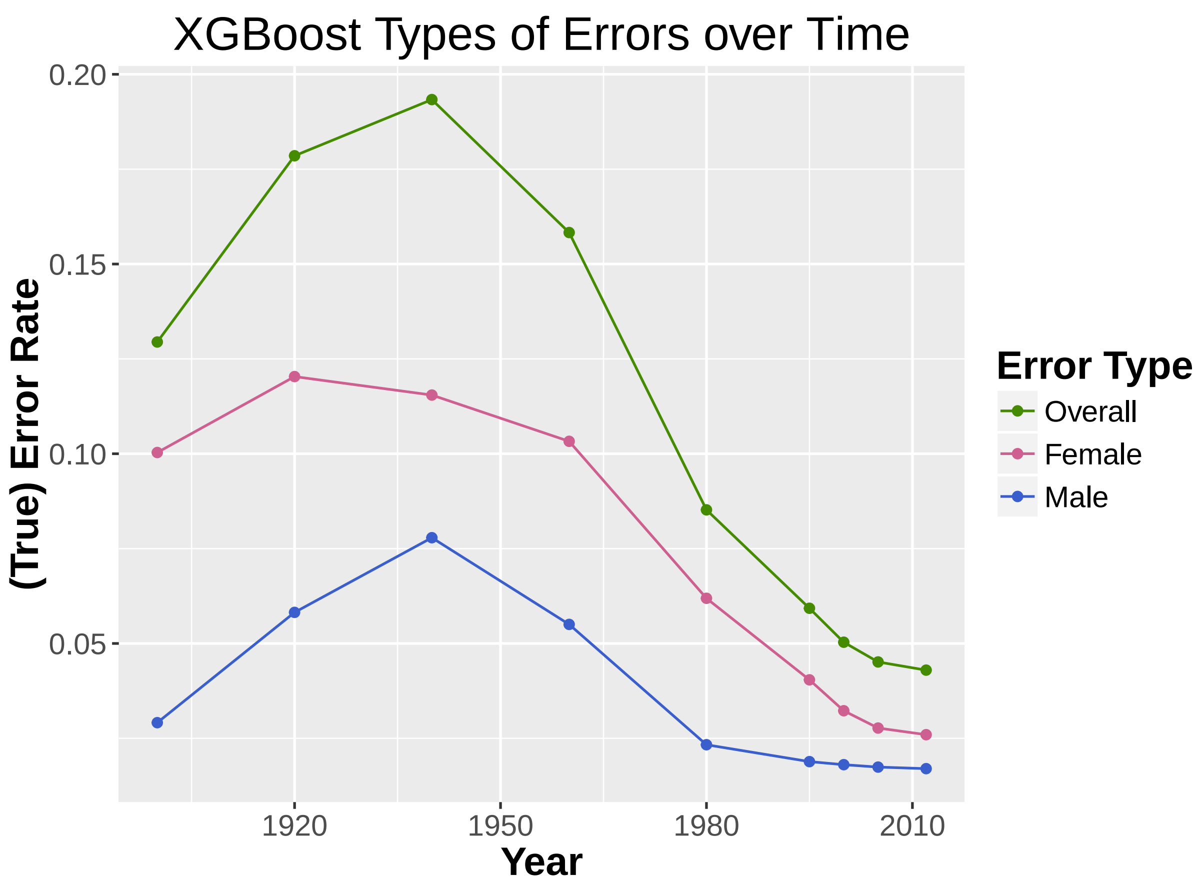
Finally, we build a visualizable data frame and output our error over time graph.

```
years <- c(1900, 1920, 1940, 1960, 1980, 1995, 2000, 2005, 2012)
errors <- c(info1900[5], info1920[5], info1940[5], info1960[5], info1980[5], info1995[5], info2000[5], 
errorsm <- c(info1900[6], info1920[6], info1940[6], info1960[6], info1980[6], info1995[6], info2000[6],
errorsf <- c(info1900[7], info1920[7], info1940[7], info1960[7], info1980[7], info1995[7], info2000[7],
df <- data.frame(years, errors, errorsm, errorsf)
df <- df %>%
  gather(error_type, rate, errors:errorsf)

ggplot(df, aes(x=years, y=rate, group=error_type, color=error_type)) +
  geom_point() + geom_line() +
  labs(title="RF Types of Errors over Time",
       y="(True) Error Rate",
       x="Year",
       color="Error Type") +
  theme(title=element_text(size=16),
        legend.title=element_text(size=16,face="bold"),
        legend.text=element_text(size=12),
        axis.text=element_text(size=12),
        axis.title=element_text(size=16,face="bold")) +
  scale_color_manual(labels=c("Overall","Female","Male"),
                     values=c("chartreuse4","hotpink3","royalblue3"))
```

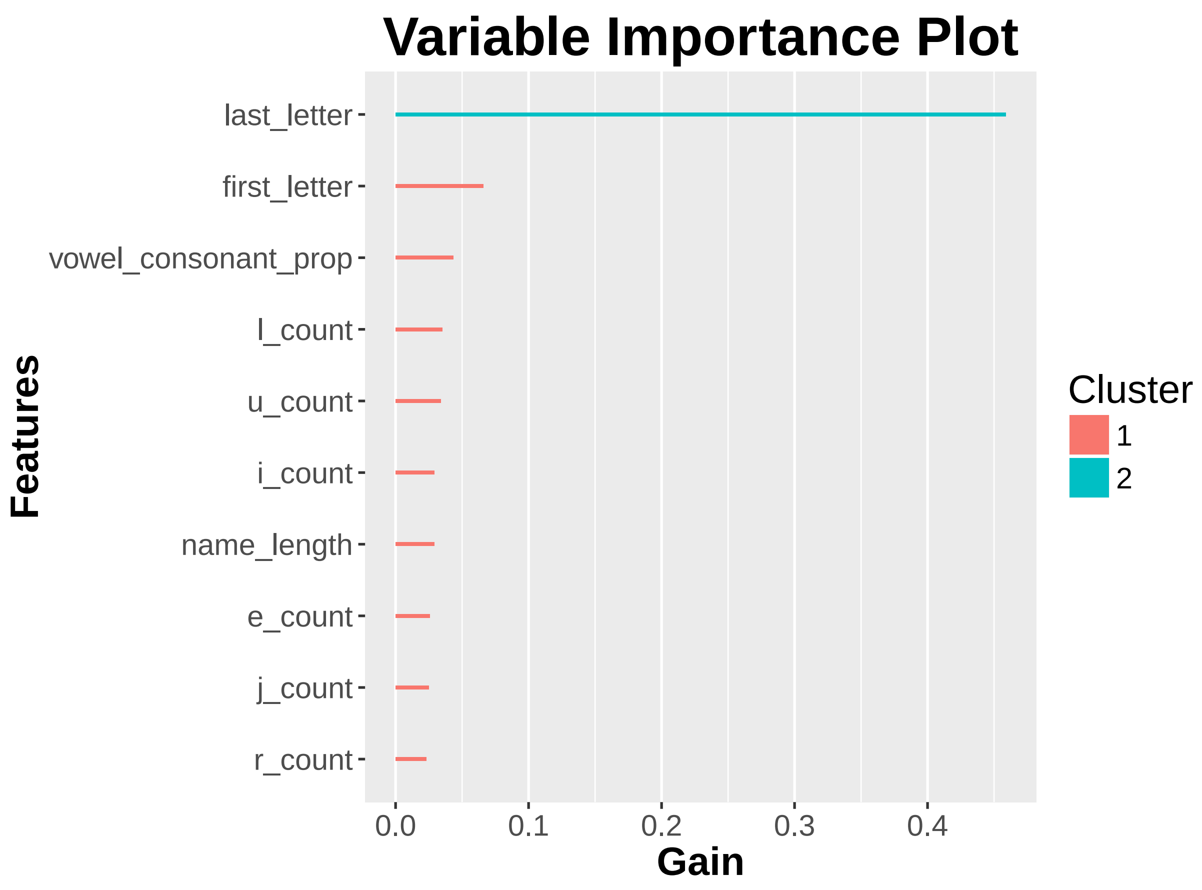


We do the same thing for the XGBoost. Since the code isn't particularly different or interesting, I'll run it but not display it here (if you want to see it, just check the .Rmd file).



As far as variable importance goes, the XGBoost variable importance plot looks as follows.

```
importance_matrix <- xgb.importance(names(train2011.final)[-1], model = bst)
xgb.plot.importance(importance_matrix[1:10,]) +
  ggtitle("Variable Importance Plot") +
  theme(title=element_text(size=18),
        legend.title=element_text(size=16),
        legend.text=element_text(size=12),
        axis.text=element_text(size=12),
        axis.title=element_text(size=16,face="bold"))
```



The XGBoost variable importance clusters variables by importance. The clusters aren't particularly informative in our case, since last letter is so enormously important that it dwarfs all of the other importances.