Analyzing Playoff Appearances with Full and Ranked Data

Linus Jen

8/25/2020

Packages

```r
library(tidyr)
library(dplyr)
library(ggplot2)
library(lubridate)
library(pROC)
library(corrplot)
library(data.table)
library(leaps)
```

Full Analytics

For this document, we will be using a lot of the insight from the prior document (`Analytics.RMD`) to first determine which subset of variables are important and which subset does the best job at predicting a team's ability to make the playoffs.

After finding the variables (for a logistic function), we will move to compare several models to see which model does the best job at predicting playoff appearances. These models include: multinomial logistic functions, K-Nearest Neighbors, Naive Bayes, and a simple neural network. These models will NOT be built from scratch, but rather, used from their respective packages. In addition to model selection, I will also be checking how many seasons prior should be used to make these predictions. With my knowledge of basketball, I would expect that different eras and shifts in rules or gameplay may influence the statistics slightly, and thus changing the number of years used before as training data may influence our findings.

The key difference here in `Analytics2.RMD`, as compared to `Analytics.RMD`, is the fact that I will be using the FULL dataset, which includes both offensive and defensive stats. I made an oversight when webscraping initially, as I only pulled the offensive stats (hitting), and did all the work with it, before realizing that I was missing half the game.

Before we begin, I recommend skimming over the `Data Scraping.RMD` file to see where the data was pulled, and how it was cleaned. I have already converted all the stats to per game statistics.

Now, let's begin!

Logistic Models and Variable Selection

For this portion, we will be using a logistic function to determine which variables should be kept in our dataset, and which variables can be removed. We will look at multicollinearity (with the VIF function), and use the `step` function to determine the best subset for the number of variables present.

Note that we will not be normalizing the data, as the ranges of each variable seem to be similar overall. In addition to this, we will be using ANOVA to determine how influential variables not, and not solely looking at the slope value to determine which points are more influential.

If time permits (as `Analytics2.RMD` is being done hastily), we can check for interactions in the data (but I need to check if this is viable for logistic regressions).

Logistic Functions for Per Game Statistics First, we'll look to our per-game statistics, and choose the variables we will want to use.

```
# Pull in our full dataset
pergame_stats = read.csv("Datasets/pergame_stats_fullsub.csv")[,-1]

# Check the data to see if there needs to be any cleaning
#glimpse(pergame_stats)
# Looks ok!

# Now, create our null model first
null_mod_pergame = glm(Playoffs ~ 1, data = pergame_stats, family = "binomial")

# Create our full model
full_mod_pergame = glm(Playoffs ~ . - Team - Year - Games_Played, data = pergame_stats,
                       family = "binomial")

summary(full_mod_pergame)
```

```
##
## Call:
## glm(formula = Playoffs ~ . - Team - Year - Games_Played, family = "binomial",
##     data = pergame_stats)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.8785  -0.2179  -0.0337  -0.0006   4.4288
##
## Coefficients: (1 not defined because of singularities)
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -2.735e+02  1.860e+02  -1.470 0.141448
## R.G           1.779e+01  3.492e+00   5.095 3.48e-07 ***
## BA           -1.253e+02  3.948e+02  -0.317 0.750934
## OBP           1.419e+02  3.609e+02   0.393 0.694144
## SLG           3.143e+02  3.641e+02   0.863 0.388144
## OPS           1.855e+02  2.896e+02   0.640 0.521890
## OPS_plus     -7.782e-02  5.407e-02  -1.439 0.150071
## ERA           5.016e-01  2.707e+01   0.019 0.985216
## H9           -5.745e+00  1.844e+00  -3.115 0.001838 **
## HR9          -3.157e+00  2.241e+00  -1.409 0.158868
## BB9          -3.751e+00  2.162e+00  -1.735 0.082689 .
## SO9          -1.286e+00  7.479e-01  -1.720 0.085513 .
## SO.W          4.361e+00  1.934e+00   2.255 0.024110 *
## num_batters   1.585e+01  7.625e+00   2.079 0.037643 *
## PA           -1.312e+00  1.501e+01  -0.087 0.930374
## AB           -1.281e+01  1.556e+01  -0.823 0.410605
## H_own        -1.917e+01  1.352e+01  -1.418 0.156092
## B2           -1.495e+01  8.429e+00  -1.773 0.076208 .
## B3           -2.368e+01  1.690e+01  -1.402 0.160973
## HR_own       -4.288e+01  2.524e+01  -1.699 0.089384 .
## RBI           5.673e+00  3.015e+00   1.881 0.059928 .
## SB           -2.137e-01  7.225e-01  -0.296 0.767432
## CS            2.081e+00  2.644e+00   0.787 0.431161
## BB_own       -2.563e+01  1.566e+01  -1.637 0.101567
## SO_own        1.229e-01  2.631e-01   0.467 0.640438
## TB                   NA         NA      NA       NA
## GDP           2.292e+00  2.337e+00   0.981 0.326706
## HBP_own      -2.389e+01  1.564e+01  -1.528 0.126610
## SH           -2.050e+01  1.493e+01  -1.373 0.169643
## SF           -1.638e+01  1.579e+01  -1.037 0.299511
## IBB_own      -1.691e+00  1.923e+00  -0.879 0.379293
## LOB_own       2.214e+01  3.058e+00   7.241 4.46e-13 ***
## num_pitchers  2.535e+00  1.053e+01   0.241 0.809848
## RA.G         -3.016e+03  7.906e+02  -3.814 0.000136 ***
## CG           -7.134e+00  3.813e+00  -1.871 0.061376 .
## tSho         -6.757e+00  8.143e+00  -0.830 0.406695
## cSho          1.081e+01  1.240e+01   0.872 0.383142
## Saves        -1.381e+00  4.019e+00  -0.344 0.731161
## IP            1.435e+02  1.049e+02   1.368 0.171204
## R_allowed     4.366e+01  3.472e+01   1.257 0.208621
## ER           -1.345e+00  2.730e+01  -0.049 0.960688
## IBB_allowed  -6.071e-01  1.585e+00  -0.383 0.701683
## HBP_allowed  -1.042e+00  2.260e+00  -0.461 0.644636
## BK            2.075e+00  3.748e+00   0.553 0.579935
## WP            1.754e+00  1.880e+00   0.933 0.350674
## BF           -2.670e+01  3.466e+01  -0.770 0.441171
## ERA_plus     -5.843e+00  6.502e+00  -0.899 0.368838
## FIP           4.417e+01  2.172e+02   0.203 0.838879
## WHIP          9.245e+03  2.518e+03   3.672 0.000240 ***
## LOB_allowed   2.681e+01  3.480e+01   0.771 0.440930
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1375.9  on 1263  degrees of freedom
## Residual deviance:  478.2  on 1215  degrees of freedom
## AIC: 576.2
##
## Number of Fisher Scoring iterations: 8
```

```
### Step function! # This will be with AIC
step_model_pergame_AIC = full_mod_pergame %>%
  MASS::stepAIC(trace = FALSE, direction = "both")

# Check out the model
summary(step_model_pergame_AIC)
```

```
##
## Call:
```

```
## glm(formula = Playoffs ~ R.G + OPS + H9 + HR9 + BB9 + SO9 + SO.W +
##     num_batters + AB + H_own + B2 + B3 + HR_own + RBI + BB_own +
##     HBP_own + SH + SF + LOB_own + RA.G + CG + IP + R_allowed +
##     WHIP, family = "binomial", data = pergame_stats)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.9706  -0.2181  -0.0367  -0.0009   4.3685
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -273.3207   137.3708  -1.990 0.046629 *
## R.G            16.6583     3.0654   5.434 5.50e-08 ***
## OPS           362.9388   180.4777   2.011 0.044327 *
## H9             -4.7144     1.5738  -2.996 0.002740 **
## HR9            -2.3520     1.0914  -2.155 0.031154 *
## BB9            -2.8289     1.9235  -1.471 0.141369
## SO9            -1.0607     0.6451  -1.644 0.100092
## SO.W            3.9927     1.8573   2.150 0.031574 *
## num_batters    15.1476     4.6129   3.284 0.001024 **
## AB            -13.3583     4.2421  -3.149 0.001638 **
## H_own         -19.7025    10.0123  -1.968 0.049088 *
## B2            -11.5402     5.3419  -2.160 0.030749 *
## B3            -16.7131    10.8466  -1.541 0.123350
## HR_own        -32.2313    15.8588  -2.032 0.042113 *
## RBI             4.9868     2.7611   1.806 0.070902 .
## BB_own        -26.0481     3.9570  -6.583 4.62e-11 ***
## HBP_own       -24.5900     4.2539  -5.781 7.44e-09 ***
## SH            -20.7327     2.3424  -8.851  < 2e-16 ***
## SF            -17.3663     3.8105  -4.557 5.18e-06 ***
## LOB_own        20.1018     2.3055   8.719  < 2e-16 ***
## RA.G        -2733.8682   707.9697  -3.862 0.000113 ***
## CG             -5.8205     2.1433  -2.716 0.006615 **
## IP             60.7652     6.4181   9.468  < 2e-16 ***
## R_allowed      15.5037     4.5444   3.412 0.000646 ***
## WHIP         8046.5099  2197.8434   3.661 0.000251 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1375.93  on 1263  degrees of freedom
## Residual deviance:  486.65  on 1239  degrees of freedom
## AIC: 536.65
##
## Number of Fisher Scoring iterations: 8
```

```
# Notice that we reduced the number of variables from 49 to now 24, over half of the
# original number!

# Let's check the ANOVA table
anova(step_model_pergame_AIC, test = "Chisq")
```

```
## Analysis of Deviance Table
```

```
##
## Model: binomial, link: logit
##
## Response: Playoffs
##
## Terms added sequentially (first to last)
##
##
##             Df Deviance Resid. Df Resid. Dev  Pr(>Chi)
## NULL                        1263    1375.93
## R.G          1   179.04      1262    1196.89 < 2.2e-16 ***
## OPS          1     0.65      1261    1196.24 0.4218669
## H9           1   325.23      1260     871.01 < 2.2e-16 ***
## HR9          1    45.64      1259     825.37 1.419e-11 ***
## BB9          1    75.25      1258     750.13 < 2.2e-16 ***
## SO9          1    16.22      1257     733.91 5.639e-05 ***
## SO.W         1     0.53      1256     733.38 0.4682063
## num_batters  1     1.14      1255     732.24 0.2847456
## AB           1    20.53      1254     711.71 5.873e-06 ***
## H_own        1     0.23      1253     711.47 0.6296346
## B2           1     1.32      1252     710.15 0.2501918
## B3           1     0.37      1251     709.78 0.5441754
## HR_own       1     0.05      1250     709.74 0.8253133
## RBI          1     3.82      1249     705.91 0.0504967 .
## BB_own       1     6.48      1248     699.43 0.0109025 *
## HBP_own      1     0.69      1247     698.74 0.4059810
## SH           1     2.04      1246     696.70 0.1532056
## SF           1     1.30      1245     695.40 0.2543418
## LOB_own      1     4.00      1244     691.40 0.0454041 *
## RA.G         1    32.15      1243     659.24 1.424e-08 ***
## CG           1     7.09      1242     652.15 0.0077371 **
## IP           1   150.80      1241     501.35 < 2.2e-16 ***
## R_allowed    1     0.31      1240     501.04 0.5807453
## WHIP         1    14.39      1239     486.65 0.0001488 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
# In our ANOVA table, we clearly see that there are a LOT of insignificant variables

# Lastly, let's check for multicollinearity
car::vif(step_model_pergame_AIC)
```

```
##         R.G         OPS          H9         HR9         BB9         SO9
##  149.379153 3259.412196   48.053302    3.551712   37.658658   48.767454
##        SO.W num_batters          AB       H_own          B2          B3
##   59.183849    2.185189  307.167036 1727.182233   86.077802   38.437078
##      HR_own         RBI      BB_own     HBP_own          SH          SF
## 1154.568485  118.379069  202.306982   16.070254    8.133972    3.387194
##     LOB_own        RA.G          CG          IP   R_allowed        WHIP
##   47.125257  450.605657    2.817336   17.134536  291.392876  296.900982
```

```
# Oh no, that's really, really bad. We only want a VIF below 10, and preferably below 5
```

```r
### Step function, but now will BIC
step_model_pergame_BIC = full_mod_pergame %>%
  MASS::stepAIC(trace=FALSE, k = log(nrow(pergame_stats)), direction = "both")

# Check out the second model
summary(step_model_pergame_BIC)
```

```
##
## Call:
## glm(formula = Playoffs ~ R.G + SO.W + num_batters + AB + BB_own +
##     HBP_own + SH + SF + LOB_own + RA.G + IP + R_allowed + WHIP,
##     family = "binomial", data = pergame_stats)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.8442  -0.2415  -0.0480  -0.0019   4.3688
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   -8.4490    16.3952  -0.515 0.606319
## R.G           19.6616     1.6917  11.622  < 2e-16 ***
## SO.W           1.4307     0.4073   3.513 0.000443 ***
## num_batters   14.7542     4.0979   3.600 0.000318 ***
## AB           -19.2386     1.9673  -9.779  < 2e-16 ***
## BB_own       -18.6231     2.0008  -9.308  < 2e-16 ***
## HBP_own      -17.5892     2.2981  -7.654 1.95e-14 ***
## SH           -18.8512     2.1403  -8.808  < 2e-16 ***
## SF           -16.3268     3.0137  -5.418 6.04e-08 ***
## LOB_own       18.8993     2.0418   9.256  < 2e-16 ***
## RA.G        -866.1832   224.7469  -3.854 0.000116 ***
## IP            56.4086     5.8456   9.650  < 2e-16 ***
## R_allowed      2.6952     0.9955   2.707 0.006784 **
## WHIP        2215.8202   691.4789   3.204 0.001353 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1375.93  on 1263  degrees of freedom
## Residual deviance:  519.23  on 1250  degrees of freedom
## AIC: 547.23
##
## Number of Fisher Scoring iterations: 7
```

```r
# Wow, we cut this down to a total of 13 variables!


# Let's check out the ANOVA table
anova(step_model_pergame_BIC, test = "Chisq")
```

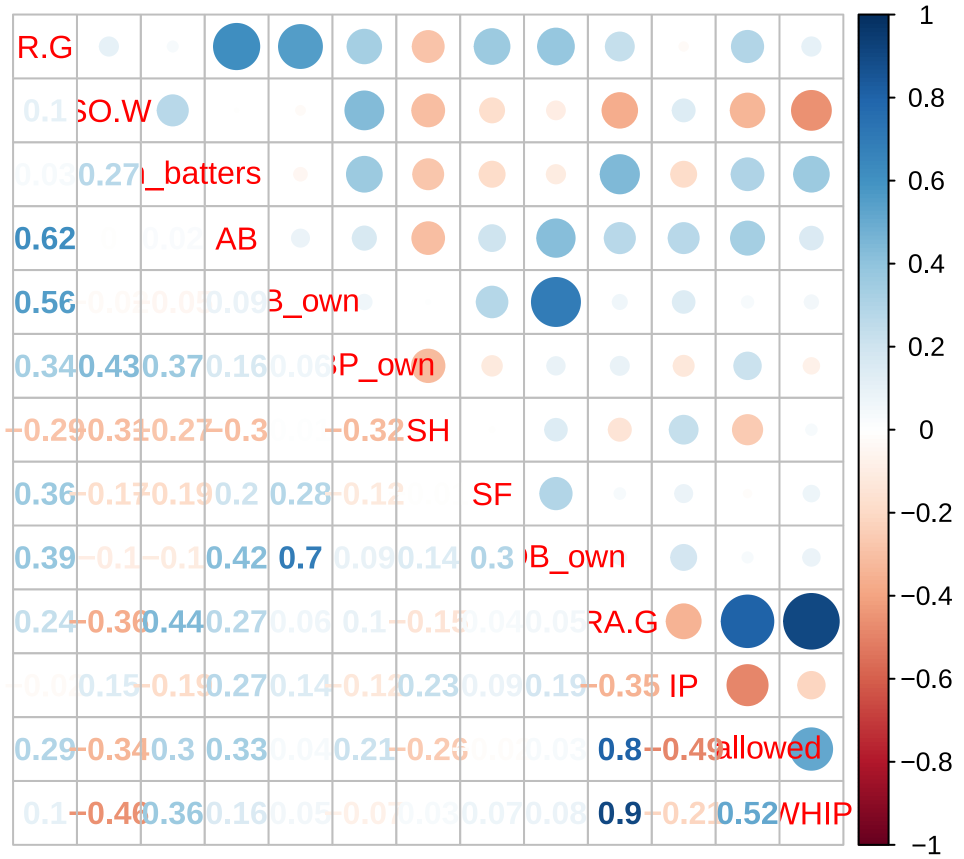
```
## Analysis of Deviance Table
```

```
##
## Model: binomial, link: logit
##
## Response: Playoffs
##
## Terms added sequentially (first to last)
##
##
##              Df Deviance Resid. Df Resid. Dev  Pr(>Chi)
## NULL                         1263    1375.93
## R.G          1  179.039      1262    1196.89 < 2.2e-16 ***
## SO.W         1  160.507      1261    1036.38 < 2.2e-16 ***
## num_batters  1   39.245      1260     997.14 3.739e-10 ***
## AB           1   30.929      1259     966.21 2.676e-08 ***
## BB_own       1    4.105      1258     962.10 0.0427558 *
## HBP_own      1    6.017      1257     956.08 0.0141710 *
## SH           1    5.100      1256     950.98 0.0239222 *
## SF           1    7.141      1255     943.84 0.0075359 **
## LOB_own      1   10.839      1254     933.01 0.0009939 ***
## RA.G         1  196.330      1253     736.67 < 2.2e-16 ***
## IP           1  206.692      1252     529.98 < 2.2e-16 ***
## R_allowed    1    0.112      1251     529.87 0.7379105
## WHIP         1   10.642      1250     519.23 0.0011056 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
# Almost all variables shown below are now significant, with runs allowed being the only
# variable not holding much significance

# Lastly, let's check for multicollinearity
car::vif(step_model_pergame_BIC)
```

```
##         R.G        SO.W num_batters          AB      BB_own     HBP_own
##   50.809819    2.981364    1.900308   72.777198   56.995170    5.058863
##          SH          SF     LOB_own        RA.G          IP   R_allowed
##    7.336404    2.278526   40.602075   51.842722   14.732066   15.208481
##        WHIP
##   32.711122
```

```r
# Much better than the AIC model, but even still, we have some issues with multicollinearity


### Correlation matrix - we want to start removing unnecessary variables
BIC_index = which(names(pergame_stats) %in%
                    c("R.G", "SO.W", "num_batters", "AB", "BB_own", "HBP_own",
                      "SH","SF", "LOB_own", "RA.G", "IP", "R_allowed", "WHIP"))

# Create our data frame
BIC_df = pergame_stats[,BIC_index]

# Plot this out
corrplot::corrplot.mixed(cor(BIC_df))
```



```r
# Because we saw that BIC is a subset of AIC, we can ignore any changes to AIC,
# and will only address the BIC model

# First off, we see that RA.G and WHIP are STRONGLY correlated. Thus, we will remove RA.G
# first
step_mod_subset1 = glm(Playoffs ~ R.G + SO.W + num_batters + AB + BB_own
                       + HBP_own + SH + SF + LOB_own + IP + R_allowed + WHIP,
                       data = pergame_stats, family = "binomial")


# Check VIF
car::vif(step_mod_subset1)
```

```
##         R.G        SO.W num_batters          AB      BB_own     HBP_own
##   49.513194    2.147252    1.950492   70.383100   56.267312    4.887661
##          SH          SF     LOB_own          IP   R_allowed        WHIP
##    6.950146    2.329078   40.314263   14.369295    4.757079    2.004190
```

```r
# Next, AB is a key variable to remove, as it has the largest VIF value

# New subset model, now without AB
step_mod_subset2 = glm(Playoffs ~ R.G + SO.W + num_batters + BB_own
                       + HBP_own + SH + SF + LOB_own + IP + R_allowed + WHIP,
                       data = pergame_stats, family = "binomial")
```

```r
# Check VIF
car::vif(step_mod_subset2)
```

```
##         R.G        SO.W num_batters      BB_own     HBP_own          SH 
##    3.399868    2.096359    1.862819    2.549785    1.649685    1.445029 
##          SF     LOB_own          IP   R_allowed        WHIP 
##    1.324090    2.306945    1.194552    3.077009    1.907732
```

```r
# Finally, a model without any issues of multicollinearity!


### Save this final model
step_mod_final = step_mod_subset2

# Check the coefficients and importance
summary(step_mod_final)
```

```
## 
## Call:
## glm(formula = Playoffs ~ R.G + SO.W + num_batters + BB_own + 
##     HBP_own + SH + SF + LOB_own + IP + R_allowed + WHIP, family = "binomial", 
##     data = pergame_stats)
## 
## Deviance Residuals: 
##     Min       1Q   Median       3Q      Max  
## -2.5620  -0.3875  -0.1204  -0.0111   2.9346  
## 
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)    
## (Intercept)  -11.4027    12.7344  -0.895  0.37056    
## R.G            4.3767     0.3745  11.688  < 2e-16 ***
## SO.W           0.4899     0.3079   1.591  0.11158    
## num_batters   10.5357     3.6465   2.889  0.00386 ** 
## BB_own         0.6399     0.3711   1.724  0.08463 .  
## HBP_own        1.6997     1.1649   1.459  0.14455    
## SH             0.0480     0.8333   0.058  0.95406    
## SF             2.8242     2.0463   1.380  0.16754    
## LOB_own       -0.5363     0.4277  -1.254  0.20989    
## IP             1.0540     1.3842   0.761  0.44639    
## R_allowed     -4.6600     0.3798 -12.270  < 2e-16 ***
## WHIP        -348.6452   160.0711  -2.178  0.02940 *  
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## (Dispersion parameter for binomial family taken to be 1)
## 
##     Null deviance: 1375.93  on 1263  degrees of freedom
## Residual deviance:  668.55  on 1252  degrees of freedom
## AIC: 692.55
## 
## Number of Fisher Scoring iterations: 7
```

```r
# We see that the coefficients for many different variables are not seen as insignificant
# This is quite concerning, as with our BIC model, all variables were seen as significant

# Let's check our ANOVA table
anova(step_mod_final, test = "Chisq")
```

```
## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: Playoffs
##
## Terms added sequentially (first to last)
##
##
##             Df Deviance Resid. Df Resid. Dev  Pr(>Chi)
## NULL                         1263    1375.93
## R.G          1  179.039      1262    1196.89 < 2.2e-16 ***
## SO.W         1  160.507      1261    1036.38 < 2.2e-16 ***
## num_batters  1   39.245      1260     997.14 3.739e-10 ***
## BB_own       1   16.402      1259     980.73 5.124e-05 ***
## HBP_own      1    3.773      1258     976.96  0.052088 .
## SH           1    8.460      1257     968.50  0.003630 **
## SF           1    7.275      1256     961.23  0.006993 **
## LOB_own      1    1.899      1255     959.33  0.168182
## IP           1   36.333      1254     922.99 1.663e-09 ***
## R_allowed    1  249.478      1253     673.52 < 2.2e-16 ***
## WHIP         1    4.966      1252     668.55  0.025856 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
# Here, we see that now only HBP_own and LOB_own are insignificant
```

We see above that our step function, based off AIC and BIC, does a great job at removing a lot of the unnecessary variables.

Based off our AIC model, the best variables to predict playoff appearances are: runs per game by own team, ops, hits per game by opposing team, home runs per game by opposing team, BB per game by opposing team, SO per game by opposing team, SO/W, average number of batters per game, AB, hits per game by own team, number of second and third base hits by own team, home runs by own team, RBI's, own BB's, own HBP, SH, SF, own LOB, RA per game, CG, IP, average runs allowed by own team, and WHIP. This totals to be 24 variables.

Based off our BIC model, the best variables to predict playoff appearances are: runs per game by own team, SO/W, average number of batters per game, AB, own BB, own HBP, SH, SF, own LOB, RA per game, IP, average runs allowed by own team, and WHIP. This totals to be 13 variables.

Some overlap in variables between the two models include: runs per game by own team, SO/W, average number of batters per game, AB, own BB's, own HBP, SH, SF, own LOB, average runs against own team, IP, average runs allowed by own team, and WHIP, meaning that the BIC model is actually a subset of the AIC model.

Below, we will be checking the accuracy, precision, and recall of the full model, the model gained by minimizing AIC, and the model gained by minimizing BIC.

To do this, I will create a function that will take in a function and data, and output the precision, accuracy, recall, and F1 score of the models, just so we can compare quickly and efficiently which models perform the best.

```r
# Create a function that takes in the model, and the data to predict
predFunc = function(model, data, model_type = "logistic") {
  # First, print the model we're running
  print(model_type)

  # First, unlike typical logistic functions, playoff odds aren't simply just above/below 0.5
  # Instead, it's the top x teams (we're ignoring division at the moment)
  # Thus, we need to define the number of teams that make the playoff
  # Then, we need to break it off by year, and find the number of teams that make the playoffs each yea

  # We start with defining what years had what number of playoff teams
  # This was found in baseball-reference.com
  years = 1975:2019
  playoff_teams_num = c(rep(4, 6), 8, rep(4, 12), 0, rep(8, 17), rep(10, 8))

  # Create our data frame
  playoff_num_df = data.frame(years, playoff_teams_num)

  # Create our empty vector of predictions
  comparison = logical()

  # Now, loop over each year
  for(i in 1:length(years)) {
    # Save p as the number of playoff teams
    p = playoff_num_df[i,2]

    # Empty vector to store the comparison of T/F
    comp_year = logical()

    # Create our yearly dataframe
    year_df = data %>% filter(Year == years[i])

    # Now, given our model, predict for that year
    year_pred = predict(model, year_df)

    # From here, select the top n teams with the highest odds to win as our playoff prediction
    top_odds = sort(year_pred, decreasing = TRUE)[1:p]

    # Determine whether or not the team made the playoffs or not
    comp_year = ifelse(year_pred %in% top_odds, TRUE, FALSE)

    # Append this to our comparison vector
    comparison = c(comparison, comp_year)
  }

  # Create our confusion matrix
  table_pred = table(comparison, data$Playoffs)

  # Print our confusion table
  print(table_pred)
  # Next, store these values to be easily accessed
  true_pos = table_pred[2,2]
  true_neg = table_pred[1,1]
  false_pos = table_pred[2,1]
  false_neg = table_pred[1,2]

  # Now, to find our values to compare

  # First is accuracy
  acc = (true_pos + true_neg) / sum(table_pred)

  # Then, precision
  prec = true_pos / (true_pos + false_pos)

  # Recall
  recall = true_pos / (true_pos + false_neg)

  # Lastly, F1 score
  F1 = 2 * (prec * recall) / (prec + recall)

  # Now, to print these to the system to compare
  print(paste("Accuracy: ", acc, sep = ""))
  print(paste("Precision: ", prec, sep = ""))
  print(paste("Recall: ", recall, sep = ""))
  print(paste("F1 score: ", F1, sep = ""))
}
```

Now, let's compare our models!

```
# Time to apply our handy dandy function!

# First, our full model
predFunc(full_mod_pergame, pergame_stats, "Full Model")
```

```
## [1] "Full Model"
##
## comparison FALSE TRUE
##      FALSE   918   49
##      TRUE     50  247
## [1] "Accuracy: 0.921677215189873"
## [1] "Precision: 0.831649831649832"
## [1] "Recall: 0.834459459459459"
## [1] "F1 score: 0.833052276559865"
```

```
# Next, our AIC model
predFunc(step_model_pergame_AIC, pergame_stats, "AIC model")
```

```
## [1] "AIC model"
##
## comparison FALSE TRUE
##      FALSE   919   48
##      TRUE     49  248
## [1] "Accuracy: 0.923259493670886"
## [1] "Precision: 0.835016835016835"
## [1] "Recall: 0.837837837837838"
## [1] "F1 score: 0.836424957841484"
```

```
# Now, our BIC model
predFunc(step_model_pergame_BIC, pergame_stats, "BIC model")
```

```
## [1] "BIC model"
##
## comparison FALSE TRUE
##      FALSE   912   55
##      TRUE     56  241
## [1] "Accuracy: 0.912183544303797"
## [1] "Precision: 0.811447811447811"
## [1] "Recall: 0.814189189189189"
## [1] "F1 score: 0.812816188870152"
```

```
# Lastly, our final step model
predFunc(step_mod_final, pergame_stats, "final step model")
```

```
## [1] "final step model"
##
## comparison FALSE TRUE
##      FALSE   894   73
##      TRUE     74  223
## [1] "Accuracy: 0.88370253164557"
## [1] "Precision: 0.750841750841751"
## [1] "Recall: 0.753378378378378"
## [1] "F1 score: 0.752107925801012"
```

From our output above, we can cleary see that our AIC model actually does a great job, if not better, at predicting playoff appearance than our full model. However, as we said earlier, the AIC model had major issues with multicollinearity.

In our BIC model, we see a small decrease (around 1-2% drop) in our accuracy, precision, recall, and F1 scores. For half the number of predictors, this arguably is an even better model than our AIC step model.

However, when looking at our final step model, we see that there was quite a sharp drop of over 3-5% in accuracy overall, just by dropped those 2 variables. Instead of simply dropping them, I think I'll use PCA to combine those variables with other values. Because this project is mainly focused on easy ways to interpret and apply models for MLB teams (EX: achieving certain stats and seeing if they'll make the playoffs), PCA may not be beneficial due to its complexity and difficulty in interpreting the meaning of the weights, and the combination of different variables. For example,

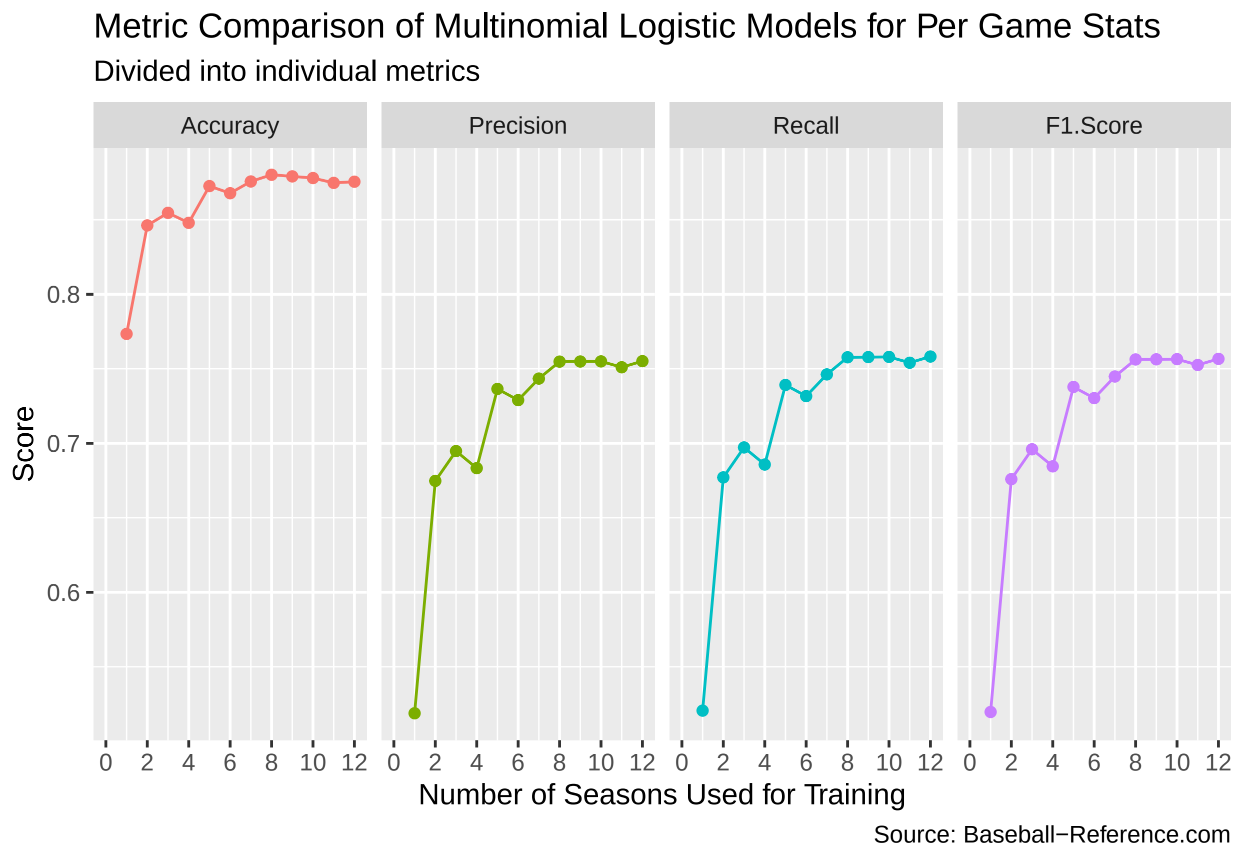
(NOTE: This will be done at a future date, as of 8/26/2020).

Lastly, let's see if there is a subset of data, in essence using n years of data before a certain year to predict playoff appearance of that year, that can improve our model. Here, we will use our final step function, found above. We will loop over using 1 to 12 years of prior data to predict each year.

```
# First, pull in our data
pergame_stats = read.csv("Datasets/pergame_stats_fullsub.csv")[,-1]
```

```r
# Also, pull in the number of teams that make the playoffs
playoff_num_df = read.csv("Datasets/playoff_num_df.csv")[,-1]

# Create an index of the variables we will be using
selected_vars = c("Team", "Playoffs", "Year", "R.G", "SO.W", "num_batters",
                  "BB_own", "HBP_own", "SH", "SF", "LOB_own", "IP",
                  "R_allowed", "WHIP")
col_index = which(names(pergame_stats) %in% selected_vars)

# now, subset our data
pergame_stats_sub = pergame_stats[,col_index]

# Create a vector containing the years to work with
years = unique(pergame_stats_sub$Year)

# Finally, create empty vectors to store our metrics
accuracy_log = numeric()
precision_log = numeric()
recall_log = numeric()
f1_score_log = numeric()



# Now, start our loop
for(num_season in 1:12) {

  # Create a vector to hold predictions
  predictions_log = numeric()

  # Create a logical vector to hold our predictions to compare later on
  comparison = logical()

  # Save the true playoff appearances
  playoffs_true = pergame_stats_sub$Playoffs

  # Create a row index to deselect the years that didn't have predictions
  row_index = numeric()

  # Create a loop over each year to predict for
  for(i in 1:length(years)) {

    # Check to see if we have valid data to create our training model
    if(years[i] - num_season < min(years)) {
      # Add this year to our row index to remove
      remove_year = which(pergame_stats_sub$Year == years[i])
      row_index = c(row_index, remove_year)

    } else {
      # First, create a subset of years to work with
      years_sub = (years[i] - num_season):(years[i] - 1)

      # Subset our data to training and testing
      training = pergame_stats_sub %>% filter(Year %in% years_sub)
      testing = pergame_stats_sub %>% filter(Year == years[i])

      # Remove certain columns
      rem_cols = which(names(training) %in% c("Team", "Year"))

      # Create our model
      log_mod_pg = glm(Playoffs ~ R.G + SO.W + num_batters + BB_own +
                         HBP_own + SH + SF + LOB_own + IP + R_allowed +
                         WHIP, data = training,
                       family = "binomial")

      # Remove more columns from our testing data
      rem_cols_test = which(names(testing) %in% c("Team", "Year", "Playoffs"))

      # Now, create our predictions
      predictions_log = predict(log_mod_pg, testing[,-rem_cols_test])
    }

    # Within each year, we now make our predictions
    # We choose the teams with the top n odds, based off the year

    # First, create a year index to match on the Year row, then pull the value
    year_index = which(playoff_num_df$years == years[i])
    p = playoff_num_df[year_index,2]

    # now, create a vector of the top p odds
    top_odds = sort(predictions_log, decreasing = TRUE)[1:p]

    # If values are in the top odds, saved as TRUE, else FALSE
    comp_year = ifelse(predictions_log %in% top_odds, TRUE, FALSE)

    # Add this to our comparisons vector
    comparison = c(comparison, comp_year)
  }

  # Finally, we create our confusion matrix for each value of num_season
  # And store the values of each metric in its respective vector
  table_log = table(comparison, playoffs_true[-row_index])

  # Creating our metrics
  acc_log = (table_log[2,2] + table_log[1,1]) / sum(table_log)
  prec_log = table_log[2,2] / (table_log[2,2] + table_log[2,1])
  rec_log = table_log[2,2] / (table_log[2,2] + table_log[1,2])
  f1_log = 2 * (prec_log * rec_log) / (prec_log + rec_log)

  # Store values
  accuracy_log = c(accuracy_log, acc_log)
  precision_log = c(precision_log, prec_log)
  recall_log = c(recall_log, rec_log)
  f1_score_log = c(f1_score_log, f1_log)
}

# Store values into a data frame
```

```r
log_acc_df_pg = data.frame(num_season = 1:12,
                           Accuracy = accuracy_log,
                           Precision = precision_log,
                           Recall = recall_log,
                           F1.Score = f1_score_log)

# Convert to tidy format
log_tidy_pg = log_acc_df_pg %>% gather("Metric", "Score", 2:5)

# Convert the metrics into factors
log_tidy_pg$Metric = factor(log_tidy_pg$Metric, levels = c("Accuracy",
                                                           "Precision",
                                                           "Recall",
                                                           "F1.Score"))

# Now, let's graph it
ggplot(log_tidy_pg, aes(x = num_season, y = Score, color = Metric)) +
  geom_point() +
  geom_line() +
  facet_grid(cols = vars(Metric)) +
  theme(legend.position = "none") +
  labs(title = "Metric Comparison of Multinomial Logistic Models for Per Game Stats",
       subtitle = "Divided into individual metrics",
       x = "Number of Seasons Used for Training",
       caption = "Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(0, 12),
                     breaks = seq(0, 12, by = 2))
```



It is good to note that here, we see that logistic models perform better when given more data to work with. The metric scores seem to cap out at around .88 for the accuracy, and .75 for precision, recall, and F1 score. However, there doesn't seem to be any subset of data that performs particularly well in this instance, and all the metrics show that using the full dataset, rather than a portion of it, results in better predictive capabilities.

It might be important to note too that warnings were created when only using 1 year's worth of data to predict the next year's playoff teams. Thus, logistic models in general may perform better with the historical data.

We'll next move onto the ranked statistics.

Logistic Functions for Ranked Statistics In an alternative dataset I've compiled, we're going to see if the ranking of certain variables at a given moment can predict a team's playoff appearance. All stats have already been converted to per-game statistics, so that a team can easily plug in values or their respective ranks, and see their "odds" at making the playoffs.

```r
# Pull in our full dataset
ranked_pergame_stats = read.csv("Datasets/ranked_fullsub.csv")[,-1]

# Ensure the dataset is clean
#glimpse(ranked_pergame_stats)
# Looks good!

# Now, create our null and full models, and run a similar process as the last time

# Null model ranked
```

```r
null_mod_ranked = glm(Playoffs ~ 1, data = ranked_pergame_stats,
                      family = "binomial")

# Full model ranked
full_mod_ranked = glm(Playoffs ~ . - Team - Year - Games_Played,
                      data = ranked_pergame_stats,
                      family = "binomial")

# Let's check out this messy full model
summary(full_mod_ranked)
```

```
##
## Call:
## glm(formula = Playoffs ~ . - Team - Year - Games_Played, family = "binomial",
##     data = ranked_pergame_stats)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.43404  -0.31019  -0.06578  -0.00486   3.02905
##
## Coefficients:
##                       Estimate Std. Error z value Pr(>|z|)
## (Intercept)         -8.5847738  1.3213009  -6.497 8.18e-11 ***
## R.G_ranked           0.1245706  0.0984217   1.266   0.2056
## BA_ranked            0.0342837  0.0849846   0.403   0.6866
## OBP_ranked           0.0348499  0.0680189   0.512   0.6084
## SLG_ranked           0.2164982  0.1111444   1.948   0.0514 .
## OPS_ranked          -0.0238098  0.0988195  -0.241   0.8096
## OPS_plus_ranked     -0.0071059  0.0370624  -0.192   0.8480
## ERA_ranked           0.0333771  0.1520160   0.220   0.8262
## H9_ranked           -0.0302724  0.0504870  -0.600   0.5488
## HR9_ranked          -0.0498835  0.0348916  -1.430   0.1528
## BB9_ranked           0.0090775  0.0383100   0.237   0.8127
## SO9_ranked          -0.0561508  0.0360972  -1.556   0.1198
## SO.W_ranked          0.0530184  0.0489066   1.084   0.2783
## num_batters_ranked   0.0016175  0.0214352   0.075   0.9398
## PA_ranked           -0.0323076  0.0525880  -0.614   0.5390
## AB_ranked           -0.0360249  0.0501292  -0.719   0.4724
## H_own_ranked        -0.0331419  0.0937926  -0.353   0.7238
## B2_ranked           -0.0302431  0.0194401  -1.556   0.1198
## B3_ranked            0.0166638  0.0170748   0.976   0.3291
## HR_own_ranked       -0.0132248  0.0421545  -0.314   0.7537
## RBI_ranked           0.0552398  0.1003759   0.550   0.5821
## SB_ranked            0.0136989  0.0171073   0.801   0.4233
## CS_ranked           -0.0033611  0.0191934  -0.175   0.8610
## BB_own_ranked        0.0303648  0.0383713   0.791   0.4287
## SO_own_ranked       -0.0229250  0.0180653  -1.269   0.2044
## TB_ranked           -0.0975144  0.0990489  -0.985   0.3249
## GDP_ranked          -0.0004237  0.0185248  -0.023   0.9818
## HBP_own_ranked       0.0100780  0.0155513   0.648   0.5170
## SH_ranked           -0.0261249  0.0206194  -1.267   0.2052
## SF_ranked           -0.0046079  0.0169237  -0.272   0.7854
## IBB_own_ranked       0.0273154  0.0181555   1.505   0.1324
## LOB_own_ranked       0.0343568  0.0343686   1.000   0.3175    
## num_pitchers_ranked  0.0287555  0.0222816   1.291   0.1969    
## RA.G_ranked         -0.4682404  0.1863202  -2.513   0.0120 *  
## CG_ranked            0.0229568  0.0203044   1.131   0.2582    
## tSho_ranked          0.0169902  0.0199061   0.854   0.3934    
## cSho_ranked          0.0190778  0.0216845   0.880   0.3790    
## Saves_ranked         0.1137297  0.0180123   6.314 2.72e-10 ***
## IP_ranked            0.0174613  0.0294624   0.593   0.5534    
## R_allowed_ranked     0.2773498  0.1883674   1.472   0.1409    
## ER_ranked           -0.0254808  0.1489208  -0.171   0.8641    
## IBB_allowed_ranked  -0.0145449  0.0158232  -0.919   0.3580    
## HBP_allowed_ranked  -0.0305588  0.0161767  -1.889   0.0589 .  
## BK_ranked            0.0208518  0.0140396   1.485   0.1375    
## WP_ranked            0.0149554  0.0152572   0.980   0.3270    
## BF_ranked            0.0826550  0.0563423   1.467   0.1424    
## ERA_plus_ranked      0.0631213  0.0330035   1.913   0.0558 .  
## FIP_ranked          -0.0053343  0.0528614  -0.101   0.9196    
## WHIP_ranked          0.0114645  0.0662963   0.173   0.8627    
## LOB_allowed_ranked  -0.0160866  0.0376365  -0.427   0.6691    
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## 
## (Dispersion parameter for binomial family taken to be 1)
## 
##     Null deviance: 1375.93  on 1263  degrees of freedom
## Residual deviance:  574.16  on 1214  degrees of freedom
## AIC: 674.16
## 
## Number of Fisher Scoring iterations: 7
```

```r
# Yeah this looks quite bad


### Step functions to the rescue!

# First part will be AIC
step_model_ranked_AIC = full_mod_ranked %>% MASS::stepAIC(trace = F, direction = "both")

# Check the model coefficients and statistics
summary(step_model_ranked_AIC)
```

```
## 
## Call:
## glm(formula = Playoffs ~ R.G_ranked + SLG_ranked + HR9_ranked + 
##     SO9_ranked + SO.W_ranked + AB_ranked + B2_ranked + B3_ranked + 
##     SO_own_ranked + SH_ranked + IBB_own_ranked + LOB_own_ranked + 
##     num_pitchers_ranked + RA.G_ranked + cSho_ranked + Saves_ranked + 
##     HBP_allowed_ranked + BF_ranked + ERA_plus_ranked, family = "binomial", 
##     data = ranked_pergame_stats)
## 
## Deviance Residuals: 
##      Min        1Q    Median        3Q       Max  
## -2.44747  -0.31691  -0.06721  -0.00539   2.78289  
## 
```

```
## Coefficients:
##                      Estimate Std. Error z value Pr(>|z|)    
## (Intercept)          -7.51261    0.95256  -7.887 3.10e-15 ***
## R.G_ranked            0.19667    0.03135   6.273 3.53e-10 ***
## SLG_ranked            0.10095    0.03000   3.365 0.000766 ***
## HR9_ranked           -0.04543    0.01801  -2.523 0.011633 *  
## SO9_ranked           -0.03044    0.01935  -1.573 0.115704    
## SO.W_ranked           0.03388    0.02361   1.435 0.151192    
## AB_ranked            -0.07994    0.02047  -3.904 9.45e-05 ***
## B2_ranked            -0.02557    0.01609  -1.589 0.111992    
## B3_ranked             0.02503    0.01400   1.788 0.073832 .  
## SO_own_ranked        -0.02891    0.01443  -2.003 0.045204 *  
## SH_ranked            -0.02691    0.01666  -1.615 0.106294    
## IBB_own_ranked        0.02481    0.01698   1.461 0.144042    
## LOB_own_ranked        0.04586    0.01536   2.986 0.002828 ** 
## num_pitchers_ranked   0.03327    0.01441   2.309 0.020920 *  
## RA.G_ranked          -0.19434    0.03387  -5.737 9.62e-09 ***
## cSho_ranked           0.03869    0.01427   2.710 0.006723 ** 
## Saves_ranked          0.11016    0.01614   6.823 8.89e-12 ***
## HBP_allowed_ranked   -0.02941    0.01432  -2.053 0.040081 *  
## BF_ranked             0.06210    0.02642   2.351 0.018736 *  
## ERA_plus_ranked       0.07564    0.02378   3.181 0.001469 ** 
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1375.93  on 1263  degrees of freedom
## Residual deviance:  588.82  on 1244  degrees of freedom
## AIC: 628.82
##
## Number of Fisher Scoring iterations: 7
```

```r
# There seem to be quite a few important variables in here, but also many
# insignificant ones too

# Note that there are also ONLY 19 variables, as compared to the 24 from the
# AIC model earlier

# Let's check the ANOVA
anova(step_model_ranked_AIC, test = "Chisq")
```

```
## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: Playoffs
##
## Terms added sequentially (first to last)
##
##
##                      Df Deviance Resid. Df Resid. Dev  Pr(>Chi)    
## NULL                                  1263    1375.93              
## R.G_ranked            1  264.087      1262    1111.84 < 2.2e-16 ***
## SLG_ranked           1    0.561      1261    1111.28 0.4537805
## HR9_ranked           1  191.631      1260     919.65 < 2.2e-16 ***
## SO9_ranked           1   18.939      1259     900.71 1.350e-05 ***
## SO.W_ranked          1   59.860      1258     840.85 1.018e-14 ***
## AB_ranked            1   21.932      1257     818.92 2.824e-06 ***
## B2_ranked            1    3.476      1256     815.44 0.0622789 .
## B3_ranked            1    2.341      1255     813.10 0.1259757
## SO_own_ranked        1    3.734      1254     809.37 0.0533250 .
## SH_ranked            1    0.094      1253     809.27 0.7590960
## IBB_own_ranked       1   15.107      1252     794.16 0.0001016 ***
## LOB_own_ranked       1    4.728      1251     789.44 0.0296802 *
## num_pitchers_ranked  1    1.548      1250     787.89 0.2134986
## RA.G_ranked          1  119.004      1249     668.89 < 2.2e-16 ***
## cSho_ranked          1    0.208      1248     668.68 0.6481292
## Saves_ranked         1   62.307      1247     606.37 2.939e-15 ***
## HBP_allowed_ranked   1    1.898      1246     604.47 0.1682756
## BF_ranked            1    5.252      1245     599.22 0.0219202 *
## ERA_plus_ranked      1   10.399      1244     588.82 0.0012606 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
# Mostly good predictors, but a few bad ones

# Lastly, check VIF
car::vif(step_model_ranked_AIC)
```

```
##          R.G_ranked          SLG_ranked          HR9_ranked          SO9_ranked
##            4.020253            3.995126            1.536796            2.173356
##         SO.W_ranked           AB_ranked           B2_ranked           B3_ranked
##            2.564463            2.531653            1.550036            1.120753
##       SO_own_ranked           SH_ranked      IBB_own_ranked      LOB_own_ranked
##            1.188724            1.777832            1.621195            1.431539
## num_pitchers_ranked         RA.G_ranked         cSho_ranked        Saves_ranked
##            1.272846            4.257581            1.208615            1.177312
##  HBP_allowed_ranked           BF_ranked     ERA_plus_ranked
##            1.184628            3.267015            1.925148
```

```
# Wow. All values are below 5 - this is a pretty good model!



### Step function using BIC - ranked data

# Time to make another model using the BIC penalty instead
step_model_ranked_BIC = full_mod_ranked %>%
  MASS::stepAIC(trace = FALSE, k = log(nrow(ranked_pergame_stats)), direction = "both")

# Check out the coefficients and statistics
summary(step_model_ranked_BIC)
```

```
##
## Call:
## glm(formula = Playoffs ~ R.G_ranked + HR9_ranked + RA.G_ranked +
##     Saves_ranked + BB_own_ranked + CG_ranked + SLG_ranked, family = "binomial",
##     data = ranked_pergame_stats)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.74413  -0.33947  -0.08292  -0.00734   2.89673
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)   -6.11507    0.59756 -10.233  < 2e-16 ***
## R.G_ranked     0.17334    0.03002   5.774 7.72e-09 ***
## HR9_ranked    -0.05985    0.01701  -3.518 0.000434 ***
## RA.G_ranked   -0.20272    0.02098  -9.663  < 2e-16 ***
## Saves_ranked   0.12830    0.01599   8.024 1.02e-15 ***
## BB_own_ranked  0.05037    0.01480   3.402 0.000668 ***
## CG_ranked      0.04351    0.01385   3.141 0.001685 **
## SLG_ranked     0.07092    0.02645   2.681 0.007337 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1375.93  on 1263  degrees of freedom
## Residual deviance:  619.81  on 1256  degrees of freedom
## AIC: 635.81
##
## Number of Fisher Scoring iterations: 7
```

```
# Good! Every variable seems to hold importance!
# Also note how we have much less variables - only 7 now
# A quick glance also shows that different variables were selected, as compared
# to the pergame BIC model from above

# Now, let's check the ANOVA table
anova(step_model_ranked_BIC, test = "Chisq")
```

```
## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: Playoffs
##
## Terms added sequentially (first to last)
##
##
##               Df Deviance Resid. Df Resid. Dev  Pr(>Chi)
## NULL                           1263    1375.93
## R.G_ranked     1  264.087      1262    1111.84 < 2.2e-16 ***
## HR9_ranked     1  188.992      1261     922.85 < 2.2e-16 ***
## RA.G_ranked    1  212.757      1260     710.09 < 2.2e-16 ***
## Saves_ranked   1   62.947      1259     647.14 2.123e-15 ***
## BB_own_ranked  1    9.439      1258     637.70  0.002124 **
## CG_ranked      1   10.547      1257     627.16  0.001164 **
## SLG_ranked     1    7.352      1256     619.81  0.006698 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
# We see that all values carry a significant amount of variation in the data, and
# are all important predictors in this model!

# Lastly, check our VIF
car::vif(step_model_ranked_BIC)
```

```
##    R.G_ranked    HR9_ranked   RA.G_ranked  Saves_ranked BB_own_ranked
##      3.821025      1.421279      1.711345      1.252758      1.215883
##     CG_ranked    SLG_ranked
##      1.215257      3.247876
```

```
# This is great! We see that all variables have a VIF below 4, which is amazing.

# Our BIC model shows no indication that we need to reselect or remove variables,
# which is a huge plus.
```

From the output above, it seems that ranking each team's stats with one another creates valid models - there are little to no signs of collinearity, and all the variables seem to be statistically signifcant, both in their coefficients, and the chi-squared tests.

For our AIC model of the ranked data, the variables included were: R.G, SLG, HR9, SO9, SO/W, AB, B2, B3, own SO, SH, own IBB, own LOB, average number of pitchers per game, RA.G, cSho, average number of saves per game, HBP allowed by the team, BF, and ERA+. Most of these variables had coefficients that were significant, and almost all of them were significant in the chi-squared test. Note that there were only 19 variables in this model, as compared to the 24 from the pergame AIC model.

For our BIC model of the ranked data, the variables included were: R.G, HR9, RA.G, Saves, own BB, CG, and SLG. All of these variables had significant coefficients and were significant in the ANOVA test. In addition, all the variables here, excluding CG and BB, were part of both the AIC and BIC models. Lastly, there were only 7 variables used for this model, which is 12 less than the AIC model, and 6 less (almost half!) of the variables used in the pergame BIC model. This is actually really interesting, as the rankings themselves seem to better indicate if a team will make the playoffs or not.

Let's apply these models to see how well they do against the historical data.

```
# First, our full model
predFunc(full_mod_ranked, ranked_pergame_stats, "Full Ranked Model")
```

```
## [1] "Full Ranked Model"
##
## comparison FALSE TRUE
##      FALSE   910   57
##      TRUE     58  239
## [1] "Accuracy: 0.909018987341772"
## [1] "Precision: 0.804713804713805"
## [1] "Recall: 0.807432432432432"
## [1] "F1 score: 0.806070826306914"
```

```
# Next, our AIC ranked model
predFunc(step_model_ranked_AIC, ranked_pergame_stats, "AIC Ranked Model")
```

```
## [1] "AIC Ranked Model"
##
## comparison FALSE TRUE
##      FALSE   908   59
##      TRUE     60  237
## [1] "Accuracy: 0.905854430379747"
## [1] "Precision: 0.797979797979798"
## [1] "Recall: 0.800675675675676"
## [1] "F1 score: 0.799325463743676"
```

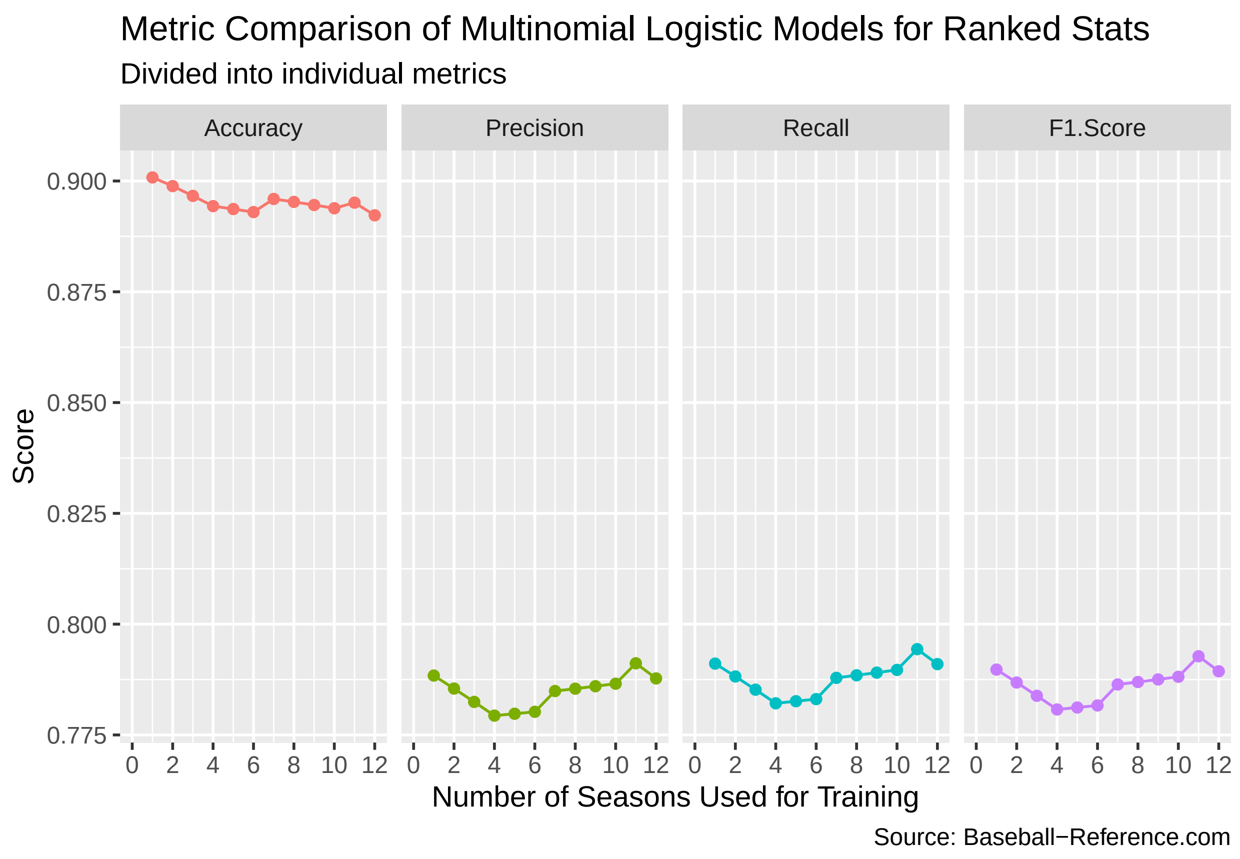
```
# Now, our BIC ranked model
predFunc(step_model_ranked_BIC, ranked_pergame_stats, "BIC Ranked Model")
```

```
## [1] "BIC Ranked Model"
##
## comparison FALSE TRUE
##      FALSE   905   62
##      TRUE     63  234
## [1] "Accuracy: 0.901107594936709"
## [1] "Precision: 0.787878787878788"
## [1] "Recall: 0.790540540540541"
## [1] "F1 score: 0.78920741989882"
```

This is quite impressive, in all honesty. Our full model with ranked data barely predicted playoff appearances better than the AIC and BIC model. Keep in mind that our BIC model ONLY has 7 variables. That's quite impressive, in my opinion. I might have to say that this is a great model, as using one's rankings in certain stats might be easier to better predict if a team makes/doesn't make the playoffs. Teams might go through lulls during the season (especially in baseball, when there are 162 games). However, if they're able to maintain a certain rank, or even plug in the ranks to see how much leeway they have to making the playoffs, I'd argue that this model might take the cake.

```
# First, pull in our data
ranked_stats = read.csv("Datasets/ranked_fullsub.csv")[,-1]

# Also, pull in the number of teams that make the playoffs
playoff_num_df = read.csv("Datasets/playoff_num_df.csv")[,-1]

# Create an index of the variables we will be using
selected_vars = c("Team", "Playoffs", "Year", "R.G_ranked", "HR9_ranked",
                  "RA.G_ranked", "Saves_ranked", "BB_own_ranked", "CG_ranked",
                  "SLG_ranked")
col_index = which(names(ranked_stats) %in% selected_vars)

# now, subset our data
ranked_stats_sub = ranked_stats[,col_index]

# Create a vector containing the years to work with
years = unique(ranked_stats_sub$Year)

# Finally, create empty vectors to store our metrics
accuracy_log = numeric()
precision_log = numeric()
recall_log = numeric()
```

```r
f1_score_log = numeric()



# Now, start our loop
for(num_season in 1:12) {

  # Create a vector to hold predictions
  predictions_log = numeric()

  # Create a logical vector to hold our predictions to compare later on
  comparison = logical()

  # Save the true playoff appearances
  playoffs_true = ranked_stats_sub$Playoffs

  # Create a row index to deselect the years that didn't have predictions
  row_index = numeric()

  # Create a loop over each year to predict for
  for(i in 1:length(years)) {

    # Check to see if we have valid data to create our training model
    if(years[i] - num_season < min(years)) {
      # Add this year to our row index to remove
      remove_year = which(ranked_stats_sub$Year == years[i])
      row_index = c(row_index, remove_year)

    } else {
      # First, create a subset of years to work with
      years_sub = (years[i] - num_season):(years[i] - 1)

      # Subset our data to training and testing
      training = ranked_stats_sub %>% filter(Year %in% years_sub)
      testing = ranked_stats_sub %>% filter(Year == years[i])

      # Remove certain columns
      rem_cols = which(names(training) %in% c("Team", "Year"))

      # Create our model
      log_mod_rk = glm(Playoffs ~ R.G_ranked + HR9_ranked + RA.G_ranked +
                         Saves_ranked + BB_own_ranked + CG_ranked + SLG_ranked,
                       data = ranked_stats_sub,
                       family = "binomial")

      # Now, create our predictions
      predictions_log = predict(log_mod_rk, testing)
    }

    # Within each year, we now make our predictions
    # We choose the teams with the top n odds, based off the year

    # First, create a year index to match on the Year row, then pull the value
    year_index = which(playoff_num_df$years == years[i])
    p = playoff_num_df[year_index,2]

    # now, create a vector of the top p odds
    top_odds = sort(predictions_log, decreasing = TRUE)[1:p]

    # If values are in the top odds, saved as TRUE, else FALSE
    comp_year = ifelse(predictions_log %in% top_odds, TRUE, FALSE)

    # Add this to our comparisons vector
    comparison = c(comparison, comp_year)
  }

  # Finally, we create our confusion matrix for each value of num_season
  # And store the values of each metric in its respective vector
  table_log = table(comparison, playoffs_true[-row_index])

  # Creating our metrics
  acc_log = (table_log[2,2] + table_log[1,1]) / sum(table_log)
  prec_log = table_log[2,2] / (table_log[2,2] + table_log[2,1])
  rec_log = table_log[2,2] / (table_log[2,2] + table_log[1,2])
  f1_log = 2 * (prec_log * rec_log) / (prec_log + rec_log)

  # Store values
  accuracy_log = c(accuracy_log, acc_log)
  precision_log = c(precision_log, prec_log)
  recall_log = c(recall_log, rec_log)
  f1_score_log = c(f1_score_log, f1_log)
}

# Store values into a data frame
log_acc_df_rk = data.frame(num_season = 1:12,
                           Accuracy = accuracy_log,
                           Precision = precision_log,
                           Recall = recall_log,
                           F1.Score = f1_score_log)

# Convert to tidy format
log_tidy_rk = log_acc_df_rk %>% gather("Metric", "Score", 2:5)

# Convert the metrics into factors
log_tidy_rk$Metric = factor(log_tidy_rk$Metric, levels = c("Accuracy",
                                                           "Precision",
                                                           "Recall",
                                                           "F1.Score"))

# Now, let's graph it
ggplot(log_tidy_rk, aes(x = num_season, y = Score, color = Metric)) +
  geom_point() +
  geom_line() +
  facet_grid(cols = vars(Metric)) +
  theme(legend.position = "none") +
  labs(title = "Metric Comparison of Multinomial Logistic Models for Ranked Stats",
       subtitle = "Divided into individual metrics",
       x = "Number of Seasons Used for Training",
       caption = "Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(0, 12),
                     breaks = seq(0, 12, by = 2))
```



From our graph above, it seems as though that when using ranked data, the number of years used for our training does NOT impact how accurate our models are. All the metrics stay about the same, with accuracy being around .9, and precision, recall, and f1 score all around .78. In addition to this, they all seem to follow a similar trend in shape when we change the number of seasons used for training.

Again, note that warnings were produced when creating the models due to the small training data when we used smaller training datasets. Thus, this logistic model performs better when there is a larger portion of data present to use as training data.

Applying Other Models For our next portion, we will be testing other models to see if we can create a better model (using KNN, naive bayes, and a simple neural network). I will be using functions from their respective R packages, and not physically training and making the models.

Subsetting our data Before we begin, however, we must first determine a subset of the data to use. Because of my (lack of) knowledge surrounding variable selection for these models (classes only taught us how to create them from scratch, and their applications), I will be using 2 sets of data. For our pergame models, I will be using the variables used in our step_mod_ranked, which was found by addressing the collinearity issues from our step BIC model. For our ranked models, I will be using the variables from our step_model_ranked_AIC model, as this dataset did not have any issues with collinearity, and larger datasets may provide greater insights to hidden relationships in our data, thus improving our model's ability to predict. The data below will not be evaluated.

```
### First, subsetting our pergame data by pergame BIC variables

# Select the columns we want to keep, and store as an index
cols_keep = c("Team", "Year", "Playoffs", "R.G", "SO.W", "num_batters", "AB",
              "BB_own", "HBP_own", "SH", "SF", "LOB_own", "IP", "R_allowed",
              "WHIP")
# column index
col_index = which(names(pergame_stats) %in% cols_keep)

# Subset the data
final_pergame_data = pergame_stats[,col_index]

### Subsetting our ranked data using the ranked AIC model's variables

# Select the rows we want to keep
cols_keep = c("Team", "Year", "Playoffs", "R.G_ranked", "SLG_ranked",
              "HR9_ranked", "SO9_ranked", "SO.W_ranked", "AB_ranked",
              "B2_ranked", "B3_ranked", "SO_own_ranked", "SH_ranked",
              "IBB_own_ranked", "LOB_own_ranked", "num_pitchers_ranked", "RA.G_ranked",
              "cSho_ranked", "Saves_ranked", "HBP_allowed_ranked", "BF_ranked",
              "ERA_plus_ranked")

# Create our index
col_index = which(names(ranked_pergame_stats) %in% cols_keep)

# Time to subset our data!
final_ranked_data = ranked_pergame_stats[,col_index]


# Save both these files for future use, so that we don't have to always make the data frames
#write.csv(final_pergame_data, "model_pergame_stats.csv")
#write.csv(final_ranked_data, "model_ranked_stats.csv")
```

KNN Model

Let's build our K-Nearest Neighbors model! From online research, the "optimal" K is the square root of the total number of samples. However, because we will later be changing the number of seasons used as training data, the K value will also vary, depending on the number of seasons.

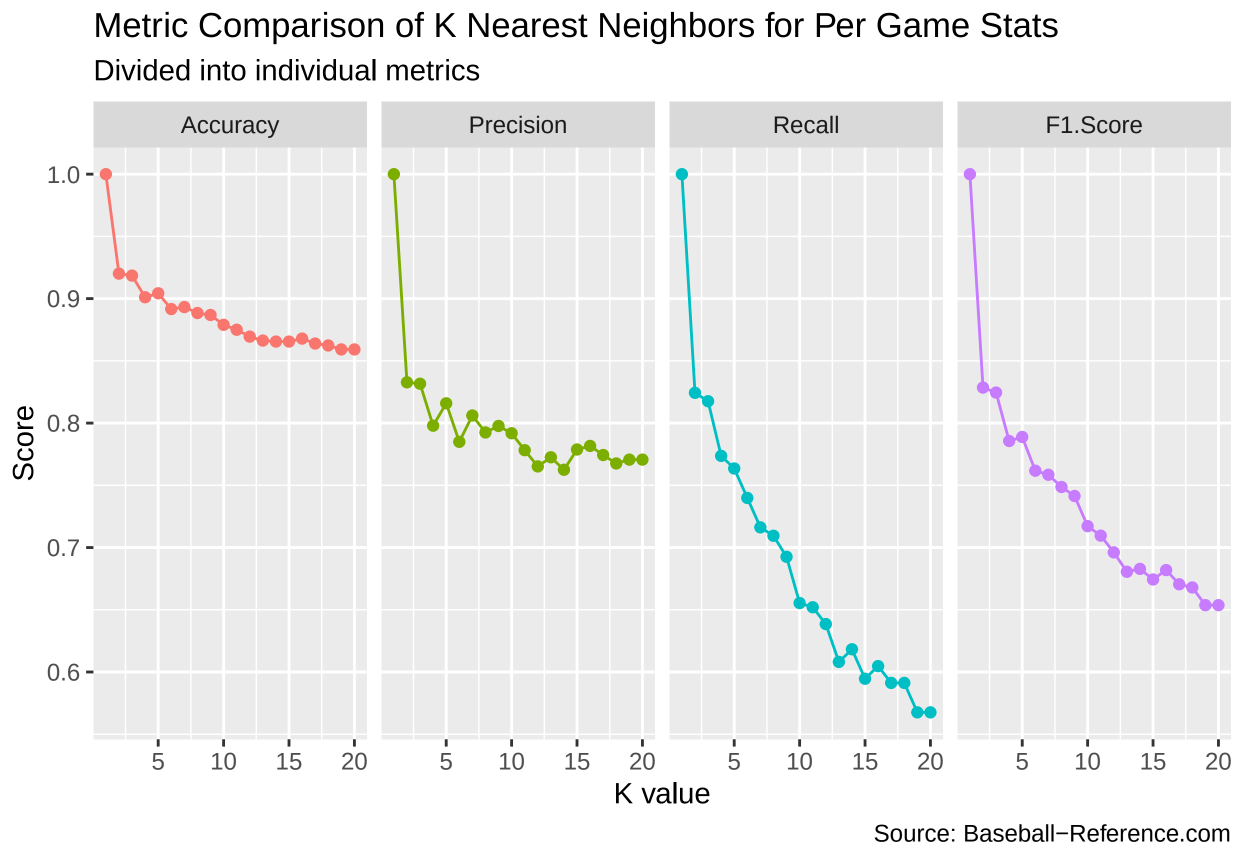
Because of this, we will run a nested loop. The outer loop will define the number of seasons prior of data to be used (we will cap this off at 12 prior seasons), then find the optimal K-value, and graph it.

Per Game Dataset This first code output will be used for our pergame stats model.

First, we will be finding how well the KNN model does with all the data. We will be testing K values from 1 to 20.

```
# Pull in our data
final_pergame_data = read.csv("Datasets/model_pergame_stats.csv")[,-1]
```

```r
# Set the seed for reproducibility
set.seed(5)

# Pull the unique years in our data
years = unique(final_pergame_data$Year)

# Index the columns to not include in our model
index_cols = which(names(final_pergame_data) %in% c("Playoffs", "Team", "Year"))

# Create empty vectors to store our values
acc_knn = numeric()
prec_knn = numeric()
rec_knn = numeric()
f1_knn = numeric()

# Create our model
# First, loop over each k value
for(k in 1:20) {
  # Build our model and predictions
  pred_knn_pg = class::knn(train = final_pergame_data[,-index_cols],
                            test = final_pergame_data[,-index_cols],
                            k = k, prob = F,
                            cl = final_pergame_data$Playoffs)

  # Confusion matrix
  table_knn = table(pred_knn_pg, final_pergame_data$Playoffs)

  # Find the metric values
  accuracy = (table_knn[2,2] + table_knn[1,1]) / sum(table_knn)
  precision = table_knn[2,2] / (table_knn[2,2] + table_knn[2,1])
  recall = table_knn[2,2] / (table_knn[2,2] + table_knn[1, 2])
  f1_score = (2 * (precision * recall) / (precision + recall))

  # Store our metric values
  acc_knn = c(acc_knn, accuracy)
  prec_knn = c(prec_knn, precision)
  rec_knn = c(rec_knn, recall)
  f1_knn = c(f1_knn, f1_score)
}

# Turn these into a data frame
knn_full_pergame = data.frame(K = 1:20,
                              Accuracy = acc_knn,
                              Precision = prec_knn,
                              Recall = rec_knn,
                              `F1 Score` = f1_knn)

# Convert to tidy format
knn_tidy_pergame = knn_full_pergame %>% gather("Metric", "Score", 2:5)

# Convert Metrics into factors
knn_tidy_pergame$Metric = factor(knn_tidy_pergame$Metric, levels = c("Accuracy",
                                                              "Precision",
                                                                "Recall",
                                                                "F1.Score"))

# Now, lets graph these!
ggplot(knn_tidy_pergame, aes(x = K, y = Score, color = Metric)) +
  geom_point() +
  geom_line() +
  facet_grid(cols = vars(Metric)) +
  theme(legend.position = "none") +
  labs(title = "Metric Comparison of K Nearest Neighbors for Per Game Stats",
       subtitle = "Divided into individual metrics",
       x = "K value",
       caption = "Source: Baseball-Reference.com")
```



From our ouput above, we see that for our per game stats, our accuracy value is pretty high - around .88. However, both our recall and F1 scores suffer in KNN models, with our F1 score hovering around .7, and our recall score around .65. Note that we ignore K values of 1, as our training and testing data are exactly the same, so when K = 1, the model will always correctly label a team making the playoffs or not. Lastly, as our K value increases, all metrics decrease - a dangerous insight, as the optimal K value should be close to the square root of the number of observations. Thus, there should be an optimal K value, but here, we don't find any.

Now, let's see if there is a subset of years that can better predict the next season's playoff teams.

CODE EXPLANATION/OVERVIEW:

The following code might be difficult to understand, especially with how it has 2 nested loops. Here's a brief explanation on what was done.

We want to output a data frame. The first column with be our K values, from 1 to 20. From research online, the optimal K value for KNN tends to be the square root of the number of rows in our data. However, as you will see below, the number of rows in our data will fluctuate, given how many years prior we will use to train our data. For each subsequent column, we will use the header to represent the number of years prior used to train our KNN model, going up to 12 years of data used. If baseball is anything like basketball, the sport goes through different trends, so using 1970's data to predict the 2010's may not be beneficial.

The outer loop represents how many years of data we will use to predict the next season. For example, when num_season = 1, that means we will only use 1 years worth of data to predict the next season, such as using 1975 to predict the outcome of 1976. We will use the accuracy vector to store the accuracy for each K value, as the rows of our data frame will represent the accuracy at each K value for the different training/testing groups.

The first nested loop, or second loop, creates models for each K value. Here, we create a logical vector titled predictions, which captures the predictions for each season using the KNN models for each year. This loop also includes a row index that will exclude the years where we couldn't make a predictions due to a lack of prior data. For example, we can't predict for 1975, as we don't have 1974's data. Another example would be when we're using 10 year's worth prior data to try and predict 1984. This can't be possible, as we don't have 1974's data, and thus we must skip this prediction.

The second loop creates a model for each season, checking to see if a model can be made with using the data from x years ago till that season. Then, it will break our data frame into 2 groups, training and testing, and make predictions with our testing data. Then, it will append the True/False values to our predictions vector.

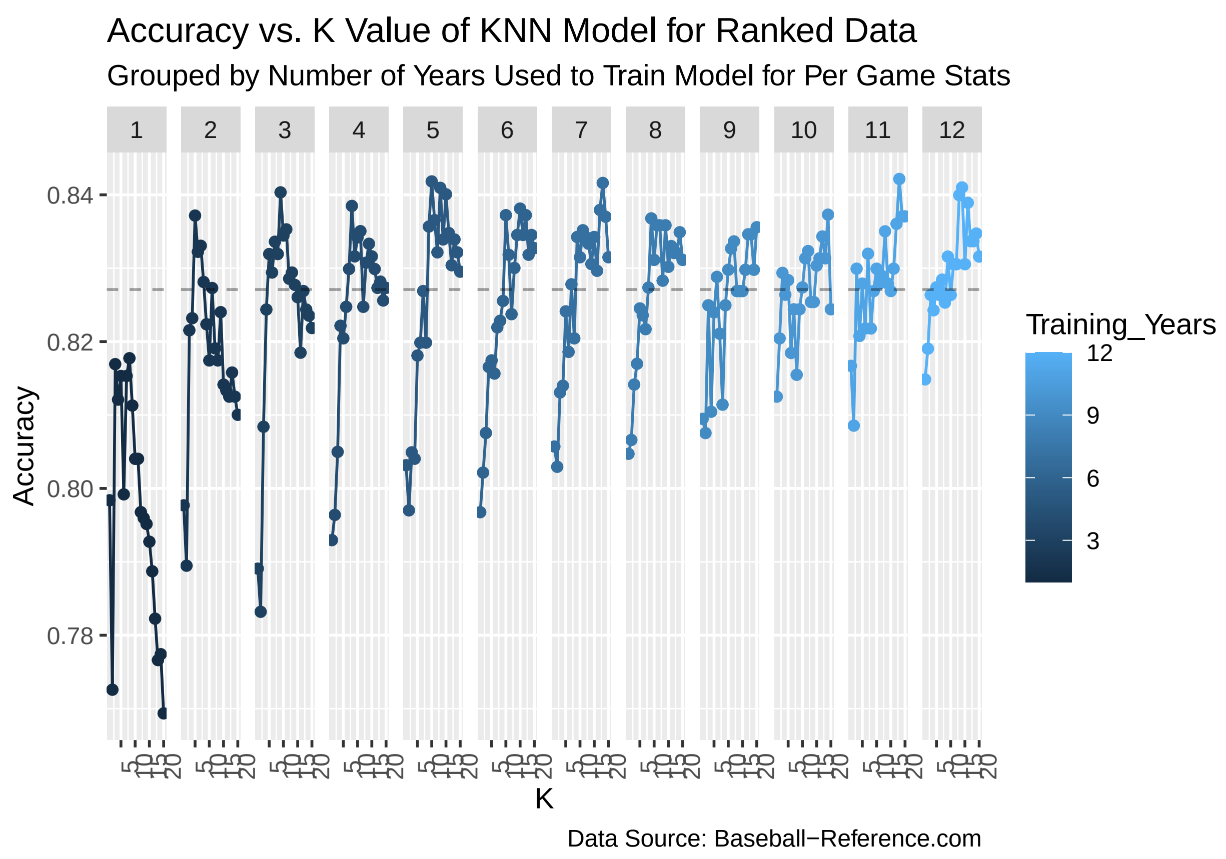
At the end, a confusion matrix is made with the real playoff appearance data and the predicted playoff appearance values, and accuracy is scored and saved in our accuracy vector. Then, once all the K values are run, and accuracy is found for each K value, this accuracy vector will be added to our data frame.

```r
# Create dataframe to store
knn_acc_df = data.frame(K = 1:20)

# First, we want to loop over the number of years prior we will be using for
# our training data

# We will be using either 1-12 years of prior data for predictions
for(num_season in 1:12) {

  # Create an empty accuracy vector to store accuracy values for each year
  accuracy = numeric()

  # Now, we'll loop over each k-value (from 1 to 20)
  for(k in 1:20) {

    # We need to store all the predictions as one vector
    # as we will be individually predicting each season
    predictions = logical()

    # This is the actual playoff appearance values, used for our confusion matrix later
    playoffs_true = final_pergame_data$Playoffs

    # We need a row index to skip over certain values in our playoffs_true vector
    # As there will be seasons we cannot predict for

    # EXAMPLE: if we are predicting with 1 years prior of data
    # we cannot predict for our start year, 1975, as we don't have 1974 data
    row_index = numeric()

    # This inner loop will loop over
    for(i in 1:length(years)) {
      # First, ensure that we have data to work with (EX we don't have 1974, so
      # we can't predict 1975)
      if(years[i] - num_season < min(years)) {
        # Create an index to remove the columns where the years don't apply
        remove_year = which(final_pergame_data$Year == years[i])

        # Add this to the row index for removal later
        row_index = c(row_index, remove_year)

        next
      } else {
        # Save the year to subset by
        year_sub = (years[i] - num_season):(years[i] - 1)

        # Create our training data
        training = final_pergame_data[which(final_pergame_data$Year %in% year_sub),]

        # Create testing data, predict for that specific season
        testing = final_pergame_data[which(final_pergame_data$Year == years[i]),]

        # create an index to not include in the model, such as year, team, or playoffs
        index_cols = which(names(training) %in% c("Playoffs", "Team", "Year"))

        # Create our KNN model using the specific K
        predictions_knn = class::knn(train = training[,-index_cols],
                                     test = testing[,-index_cols], k = k, prob = F,
                                     cl = training$Playoffs)

        # Add our predictions back into our predictions vector
        predictions = c(predictions, predictions_knn)
      }
    }

    # Now that we have stored our predictions matrix, let's create the confusion matrix
    table_knn = table(predictions, playoffs_true[-row_index])

    # Find our accuracy
    accuracy_knn = (table_knn[1,1] + table_knn[2,2]) / sum(table_knn)

    # Bind this with our accuracy vector from before
    accuracy = c(accuracy, accuracy_knn)
  }

  # After each loop for the number of seasons used as the training data,
  # Save the accuracy column in our accuracy data frame
  knn_acc_df = cbind(knn_acc_df, accuracy)
}

# Rename the columns of the dataframe with a for loop
```

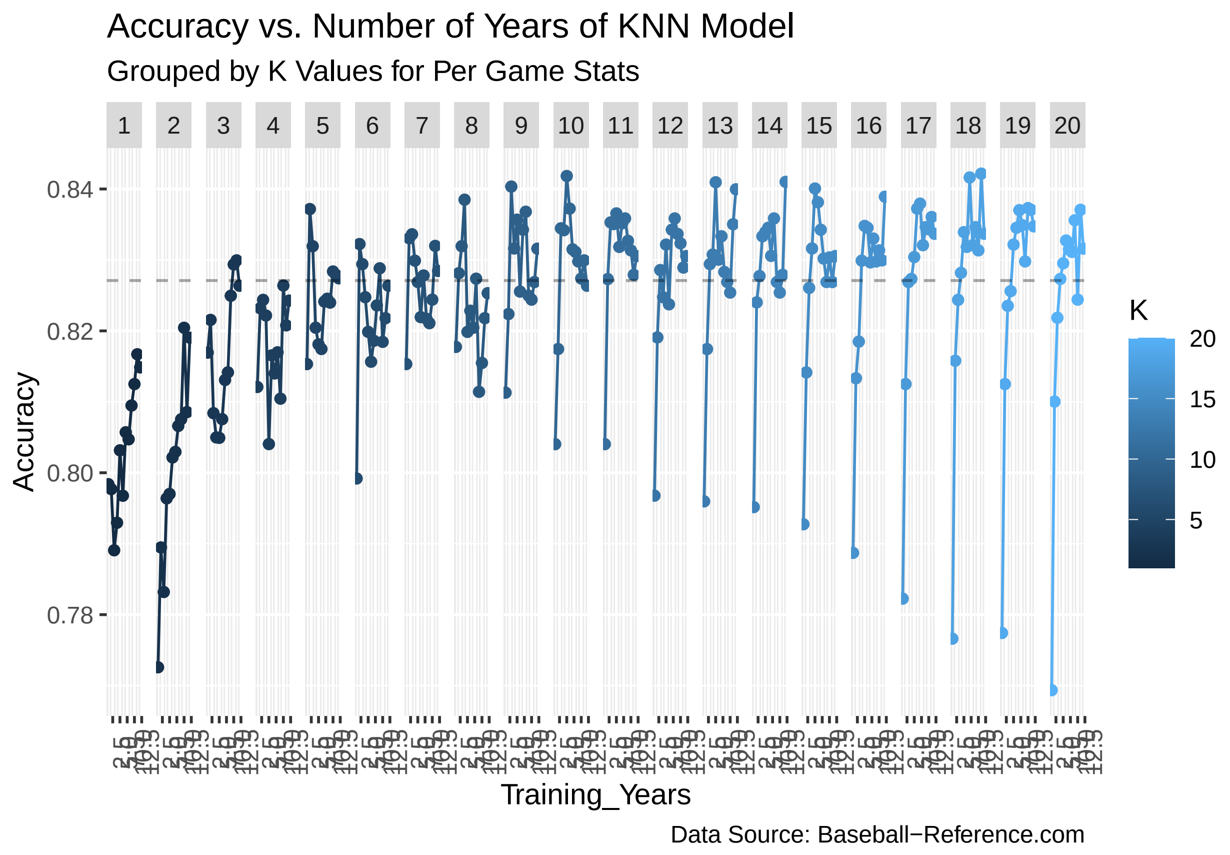
```r
for(i in 2:(ncol(knn_acc_df))) {
  col_header = paste("train_yrs_", i - 1, sep = "")
  names(knn_acc_df)[i] = col_header
}

# Ensure that the data frame came out ok
#knn_acc_df
```

Now, we graph the results to find the best combination of K and number of years used for our training data.

```r
# First, we must turn our data into tidy format
knn_tidy_pergame = knn_acc_df %>% gather("Training_Years", "Accuracy", 2:13)

# Convert Training Years into an ordered factor
knn_tidy_pergame$Training_Years = as.numeric(factor(knn_tidy_pergame$Training_Years,
                                         levels = c("train_yrs_1", "train_yrs_2",
                                                    "train_yrs_3", "train_yrs_4",
                                                    "train_yrs_5", "train_yrs_6",
                                                    "train_yrs_7", "train_yrs_8",
                                                    "train_yrs_9", "train_yrs_10",
                                                    "train_yrs_11",
                                                    "train_yrs_12")))




# Now, create some plots!
ggplot(knn_tidy_pergame, aes(x = K, y = Accuracy, color = Training_Years)) +
  geom_line() +
  geom_point() +
  facet_grid(cols = vars(Training_Years)) +
  geom_hline(yintercept = median(knn_tidy_pergame$Accuracy),
             linetype = "dashed", alpha = .35) +
  labs(title = "Accuracy vs. K Value of KNN Model for Ranked Data",
       subtitle = "Grouped by Number of Years Used to Train Model for Per Game Stats",
       caption = "Data Source: Baseball-Reference.com") +
  theme(axis.text.x = element_text(angle = 90))
```

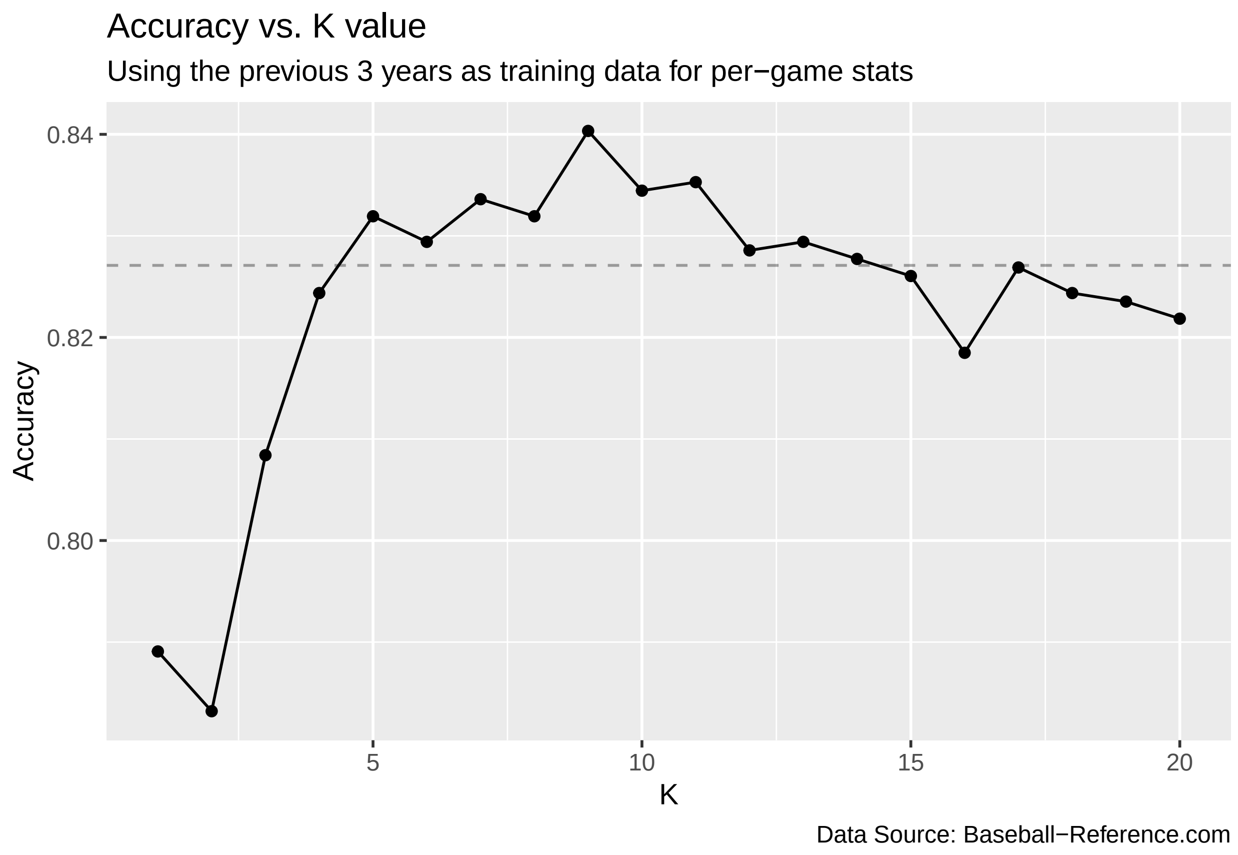


```r
ggplot(knn_tidy_pergame, aes(x = Training_Years, y = Accuracy, color = K)) +
  geom_line() +
  geom_point() +
  facet_grid(cols = vars(K)) +
  geom_hline(yintercept = median(knn_tidy_pergame$Accuracy),
             linetype = "dashed", alpha = .35) +
  labs(title = "Accuracy vs. Number of Years of KNN Model",
       subtitle = "Grouped by K Values for Per Game Stats",
       caption = "Data Source: Baseball-Reference.com") +
  theme(axis.text.x = element_text(angle = 90))
```



Notice that as the number of seasons used increases, we see less variation in the accuracy of the models, regardless of the K value. However, for the earlier models, we can clearly see an optimal value for K, as accuracy tends to drop quite a lot afterwards. The best pairing seems to be using 3 years worth of prior data, with a K value around 15. However, the best overall accuracy based off the total number of prior years is definitely using 11 years of data to predict. In addition to this, it seems that overall, a K value of 11 garners the best overall accuracy, regardless of the number of years used in the training data. Let's inspect these two subsets more.

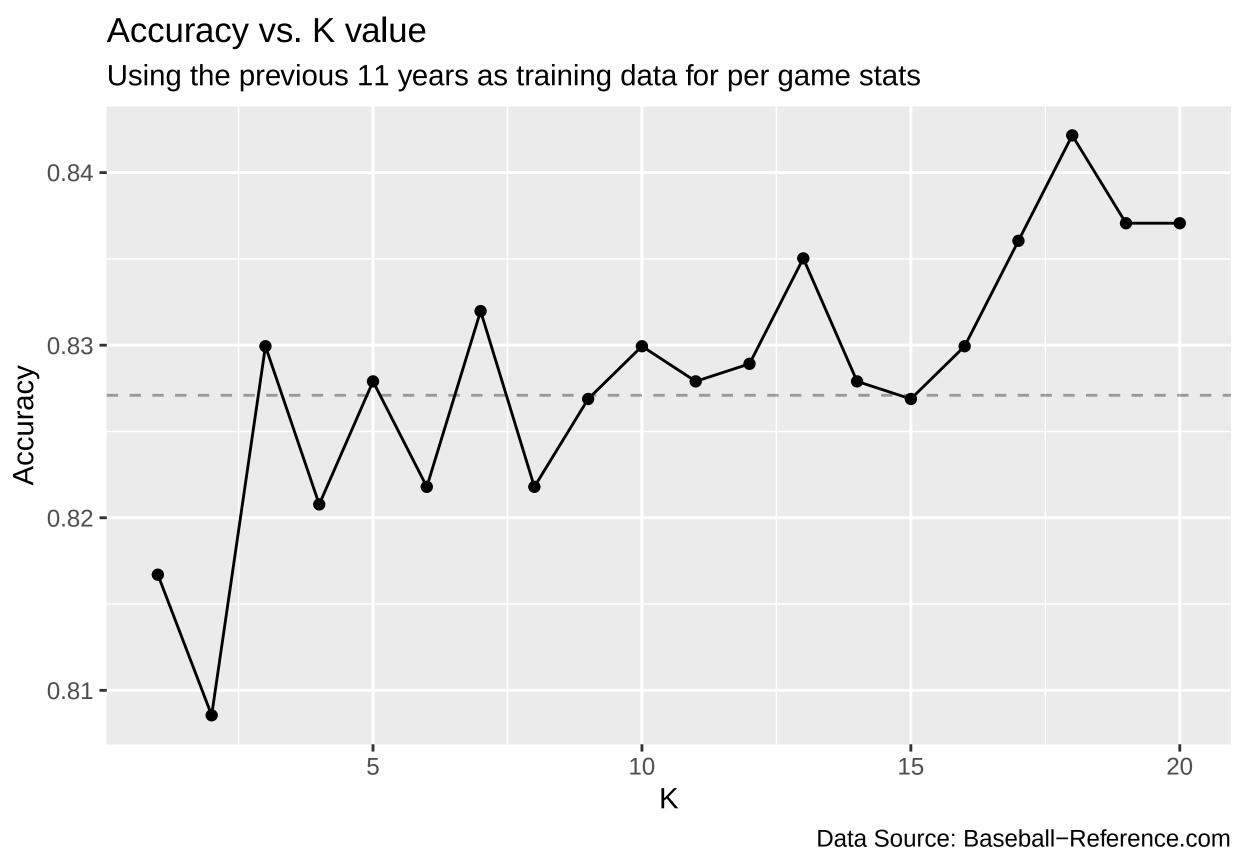
```r
# Filter the data to only include when Training_Years is 3
knn_yrs_3_pergame = knn_tidy_pergame %>% filter(Training_Years == 3)

# Graph this out
ggplot(knn_yrs_3_pergame, aes(x = K, y = Accuracy)) +
  geom_point() + geom_line() +
  geom_hline(yintercept = median(knn_tidy_pergame$Accuracy),
             linetype = "dashed", alpha = .35) +
  labs(title = "Accuracy vs. K value",
       subtitle = "Using the previous 3 years as training data for per-game stats",
       caption = "Data Source: Baseball-Reference.com")
```



From the plot above, we almost get an accuracy of .85 with a K value of 10 using only 3 years of prior data to create our model. This was the best accuracy among all the different combinations, as shown in the prior graph.

Let's look the accuracies when we use 11 years of prior data.

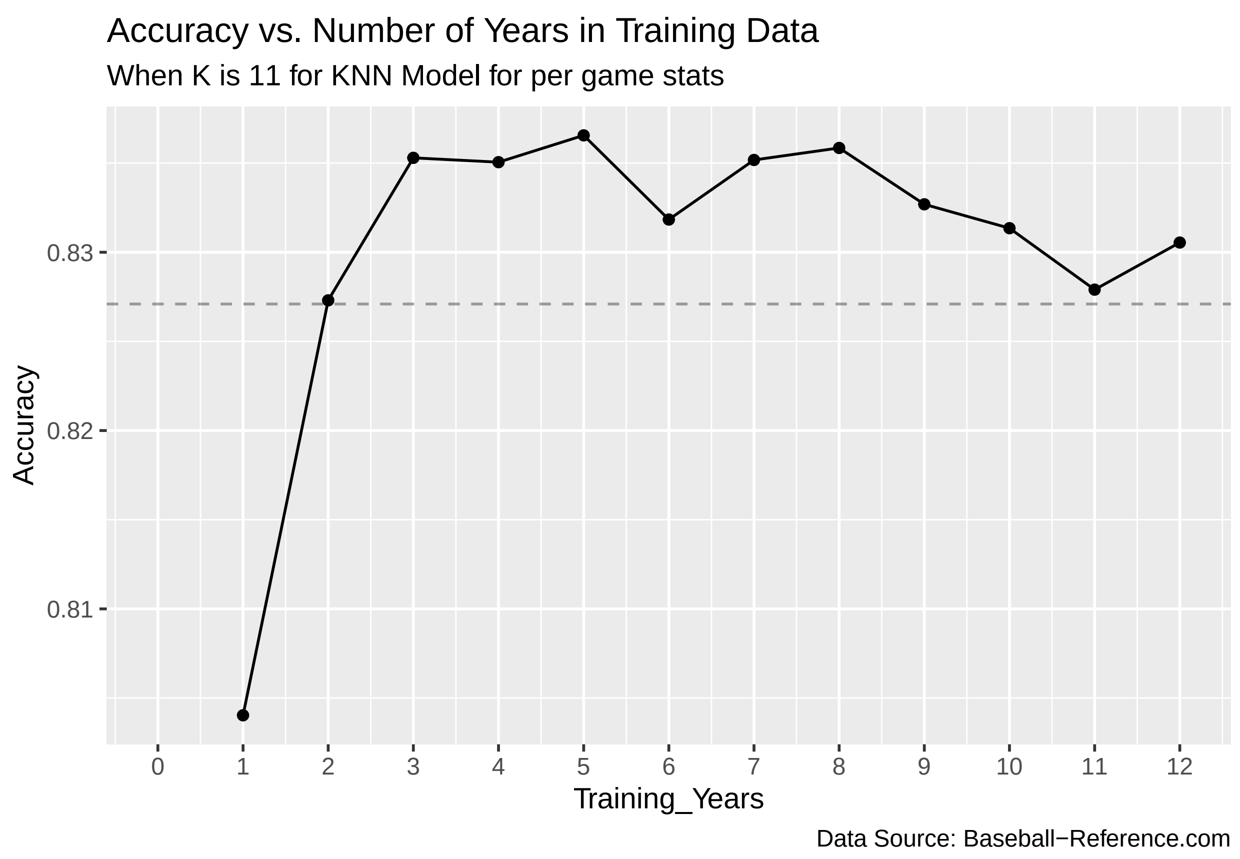
```r
# Filter the data to only include when Training_Years is 11
knn_yrs_11_pergame = knn_tidy_pergame %>% filter(Training_Years == 11)

# Graph this out
ggplot(knn_yrs_11_pergame, aes(x = K, y = Accuracy)) +
  geom_point() + geom_line() +
  geom_hline(yintercept = median(knn_tidy_pergame$Accuracy),
             linetype = "dashed", alpha = .35) +
  labs(title = "Accuracy vs. K value",
       subtitle = "Using the previous 11 years as training data for per game stats",
       caption = "Data Source: Baseball-Reference.com")
```



This is probably the best number of years to use as training data, as we only have 4 K values that are below the average accuracy of all the models.

Lastly, let's look at when K = 11.

```r
# Filter our data so that K = 11
knn_k_11_pergame = knn_tidy_pergame %>% filter(K == 11)

# Create our graph
ggplot(knn_k_11_pergame, aes(x = Training_Years, y = Accuracy)) +
  geom_point() + geom_line() +
  geom_hline(yintercept = median(knn_tidy_pergame$Accuracy),
             linetype = "dashed", alpha = .35) +
  labs(title = "Accuracy vs. Number of Years in Training Data",
       subtitle = "When K is 11 for KNN Model for per game stats",
       caption = "Data Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(0, 12),
                     breaks = 0:12)
```



Here, we see that only twice does this model (when K = 11) have an accuracy below the average of all models.

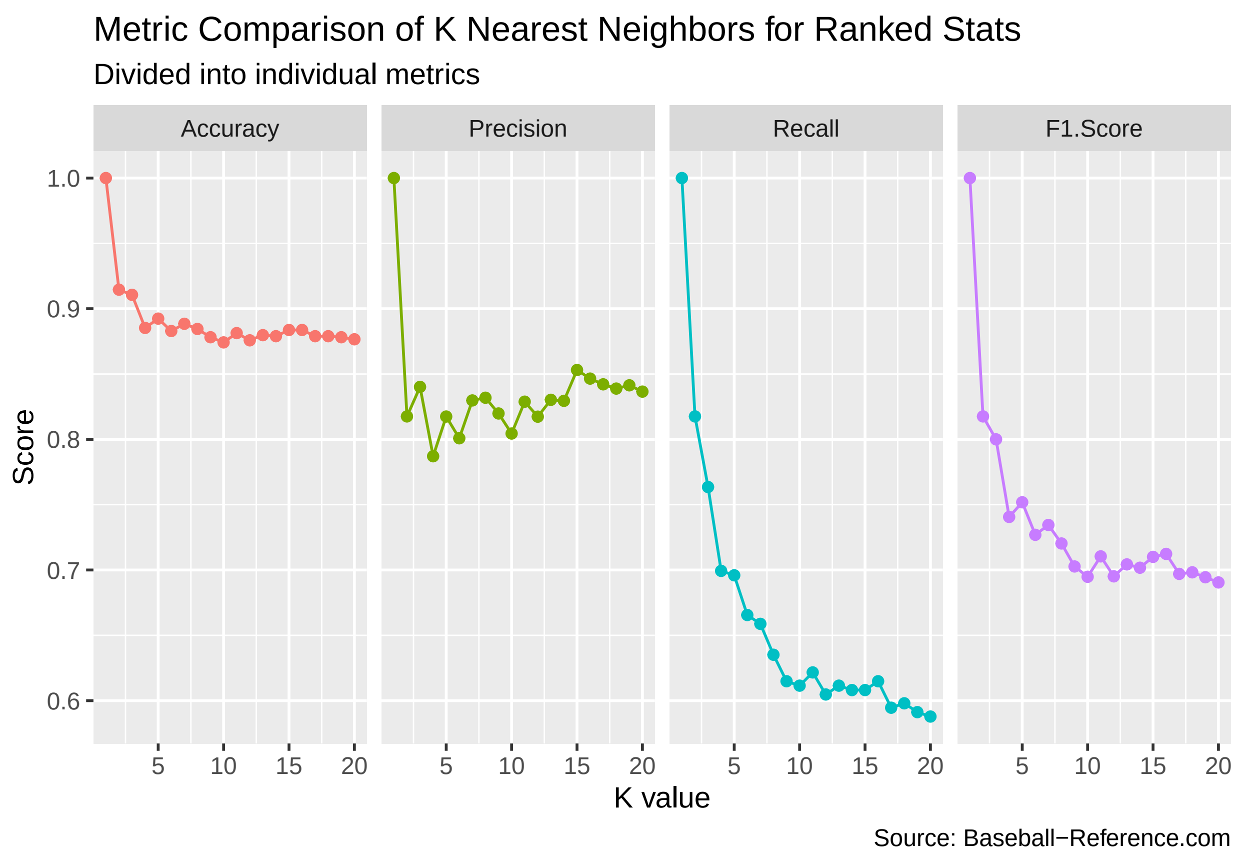
In summary: For KNN, the best number of seasons to use for training data is 11. For our K value, the best overall K value would be 11. The best combination of K values and number of seasons to use for the training data would be K = 3 and training years = 10 years before the season.

KNN on Ranked Data We'll be using an almost identical process as the ranked modeling, shown above. Notes will be in place for key differences.

First, let's use the whole dataset to create our model, using K values from 1 to twenty.

```r
# Set our seed
set.seed(5)

# Pull in our data
final_ranked_data = read.csv("Datasets/model_ranked_stats.csv")[,-1]

# Pull the unique years in our data
years = unique(final_ranked_data$Year)

# Index the columns to not include in our model
index_cols = which(names(final_ranked_data) %in% c("Playoffs", "Team", "Year"))

# Create empty vectors to store our values
acc_knn = numeric()
prec_knn = numeric()
```

```r
rec_knn = numeric()
f1_knn = numeric()

# Create our model
# First, loop over each k value
for(k in 1:20) {
  # Build our model and predictions
  pred_knn_ranked = class::knn(train = final_ranked_data[,-index_cols],
                               test = final_ranked_data[,-index_cols],
                               k = k, prob = F,
                               cl = final_ranked_data$Playoffs)

  # Confusion matrix
  table_knn = table(pred_knn_ranked, final_ranked_data$Playoffs)

  # Find the metric values
  accuracy = (table_knn[2,2] + table_knn[1,1]) / sum(table_knn)
  precision = table_knn[2,2] / (table_knn[2,2] + table_knn[2,1])
  recall = table_knn[2,2] / (table_knn[2,2] + table_knn[1, 2])
  f1_score = (2 * (precision * recall) / (precision + recall))

  # Store our metric values
  acc_knn = c(acc_knn, accuracy)
  prec_knn = c(prec_knn, precision)
  rec_knn = c(rec_knn, recall)
  f1_knn = c(f1_knn, f1_score)
}

# Turn these into a data frame
knn_full_ranked = data.frame(K = 1:20,
                             Accuracy = acc_knn,
                             Precision = prec_knn,
                             Recall = rec_knn,
                             `F1 Score` = f1_knn)

# Convert to tidy format
knn_tidy_ranked = knn_full_ranked %>% gather("Metric", "Score", 2:5)

# Convert Metrics into factors
knn_tidy_ranked$Metric = factor(knn_tidy_ranked$Metric, levels = c("Accuracy",
                                                                   "Precision",
                                                                   "Recall",
                                                                   "F1.Score"))

# Now, lets graph these!
ggplot(knn_tidy_ranked, aes(x = K, y = Score, color = Metric)) +
  geom_point() +
  geom_line() +
  facet_grid(cols = vars(Metric)) +
  theme(legend.position = "none") +
  labs(title = "Metric Comparison of K Nearest Neighbors for Ranked Stats",
       subtitle = "Divided into individual metrics",
       x = "K value",
       caption = "Source: Baseball-Reference.com")
```



There are inherent flaws with the way we are testing our KNN models, as we cannot create a model and then predict for each year. Obviously, we will ignore when K = 1, as the testing data is itself, and thus will predict with 100% accuracy if a team will make the playoffs (as the point will be closest with itself). We see that while accuracy tends to be high (~0.88), recall and F1 score suffer greatly here, garnering values below .75 on average. Precision seems to be middling, around .825, but overall, this doesn't seem to be the greatest model, as expected.

Now, let's see if using a subset of data will improve our model's abilities.

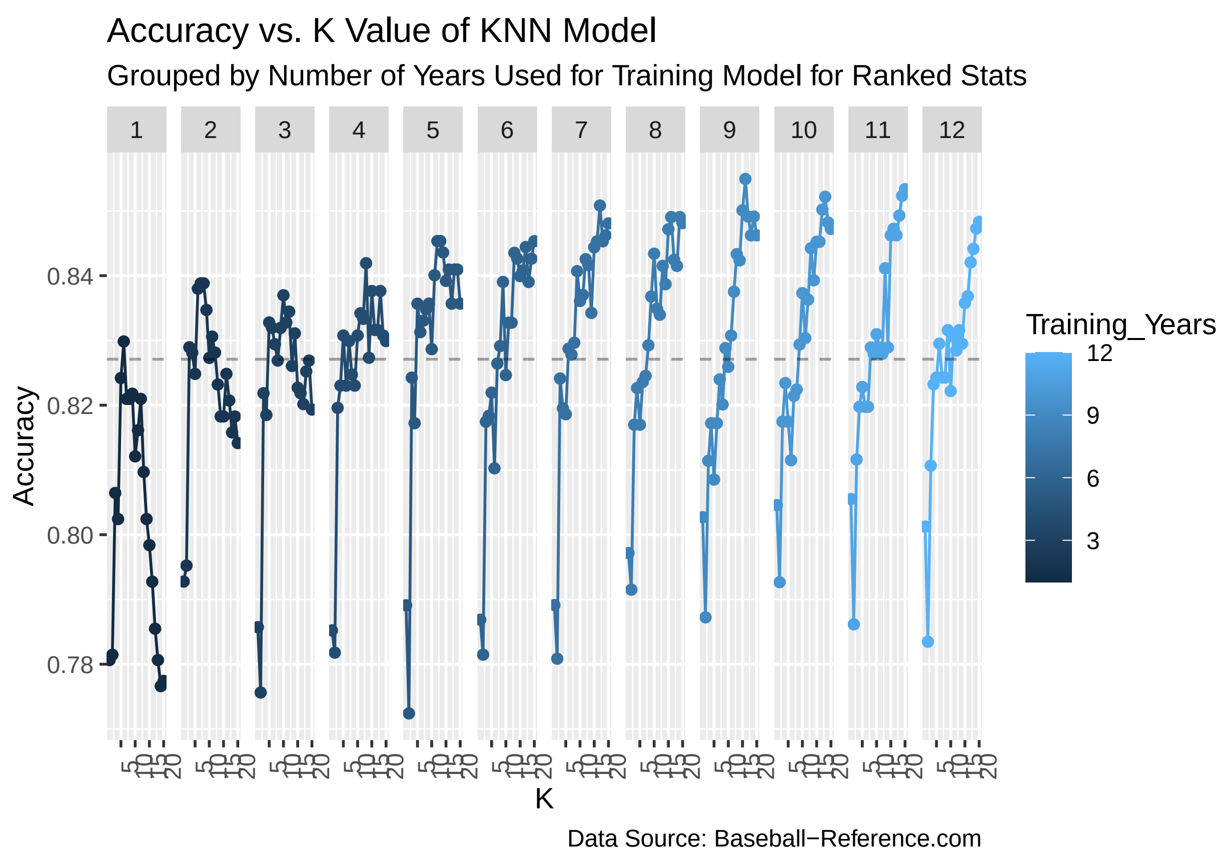
```
# Set the seed for reproducibility
set.seed(5)

# Create dataframe to store
knn_acc_df = data.frame(K = 1:20)

# First, we want to loop over the number of years prior we will be using for
# our training data

# We will be using either 1-12 years of prior data for predictions
for(num_season in 1:12) {

  # Create an empty accuracy vector to store accuracy values for each year
  accuracy = numeric()
```

```r
# Now, we'll loop over each k-value (from 1 to 20)
for(k in 1:20) {

  # We need to store all the predictions as one vector
  # as we will be individually predicting each season
  predictions = logical()

  # This is the actual playoff appearance values, used for our confusion matrix later
  playoffs_true = final_ranked_data$Playoffs

  # We need a row index to skip over certain values in our playoffs_true vector
  # As there will be seasons we cannot predict for

  # EXAMPLE: if we are predicting with 1 years prior of data
  # we cannot predict for our start year, 1975, as we don't have 1974 data
  row_index = numeric()

  # This inner loop will loop over
  for(i in 1:length(years)) {
    # First, ensure that we have data to work with (EX we don't have 1974, so
    # we can't predict 1975)
    if(years[i] - num_season < min(years)) {
      # Create an index to remove the columns where the years don't apply
      remove_year = which(final_ranked_data$Year == years[i])

      # Add this to the row index for removal later
      row_index = c(row_index, remove_year)

      next
    } else {
      # Save the year to subset by
      year_sub = (years[i] - num_season):(years[i] - 1)

      # Create our training data
      training = final_ranked_data[which(final_ranked_data$Year %in% year_sub),]

      # Create testing data, predict for that specific season
      testing = final_ranked_data[which(final_ranked_data$Year == years[i]),]

      # create an index to not include in the model, such as year, team, or playoffs
      index_cols = which(names(training) %in% c("Playoffs", "Team", "Year"))

      # Create our KNN model using the specific K
      predictions_knn = class::knn(train = training[,-index_cols],
                          test = testing[,-index_cols], k = k, prob = F,
                          cl = training$Playoffs)

      # Add our predictions back into our predictions vector
      predictions = c(predictions, predictions_knn)
    }
  }

  # Now that we have stored our predictions matrix, let's create the confusion matrix
```

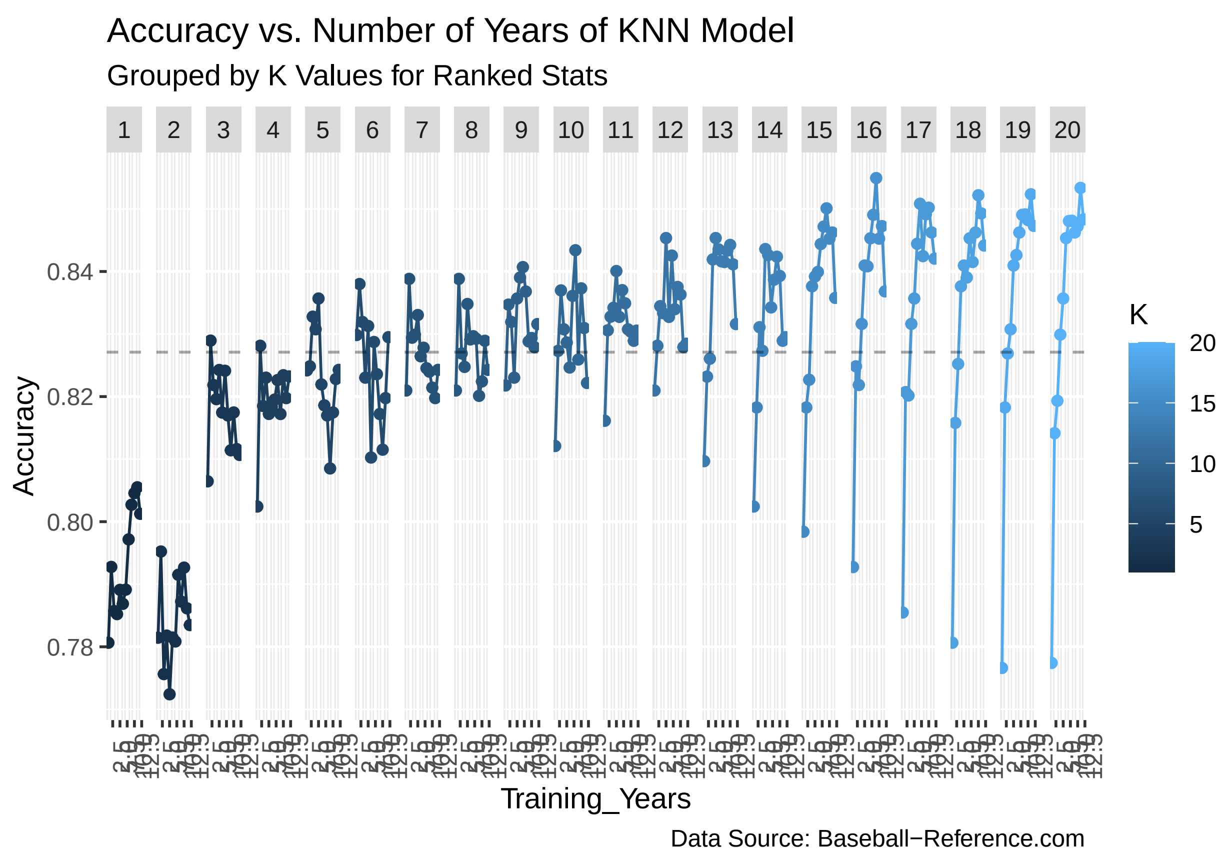
```
    table_knn = table(predictions, playoffs_true[-row_index])

    # Find our accuracy
    accuracy_knn = (table_knn[1,1] + table_knn[2,2]) / sum(table_knn)

    # Bind this with our accuracy vector from before
    accuracy = c(accuracy, accuracy_knn)
  }

  # After each loop for the number of seasons used as the training data,
  # Save the accuracy column in our accuracy data frame
  knn_acc_df = cbind(knn_acc_df, accuracy)
}

# Rename the columns of the dataframe with a for loop
for(i in 2:(ncol(knn_acc_df))) {
  col_header = paste("train_yrs_", i - 1, sep = "")
  names(knn_acc_df)[i] = col_header
}

# Ensure that the data frame came out ok
#knn_acc_df
```

Now, we graph the results to find the best combination of K and number of years used for our training data.

```
# First, we must turn our data into tidy format
knn_tidy_ranked = knn_acc_df %>% gather("Training_Years", "Accuracy", 2:13)

# Convert Training Years into an ordered factor
knn_tidy_ranked$Training_Years = as.numeric(factor(knn_tidy_ranked$Training_Years,
                                          levels = c("train_yrs_1", "train_yrs_2",
                                                     "train_yrs_3", "train_yrs_4",
                                                     "train_yrs_5", "train_yrs_6",
                                                     "train_yrs_7", "train_yrs_8",
                                                     "train_yrs_9", "train_yrs_10",
                                                     "train_yrs_11",
                                                     "train_yrs_12")))




# Now, create some plots!

# First plot compared Accuracy by K value, grouped by years used for training
ggplot(knn_tidy_ranked, aes(x = K, y = Accuracy, color = Training_Years)) +
  geom_line() +
  geom_point() +
  facet_grid(cols = vars(Training_Years)) +
  geom_hline(yintercept = median(knn_tidy_pergame$Accuracy),
             linetype = "dashed", alpha = .35) +
  labs(title = "Accuracy vs. K Value of KNN Model",
       subtitle = "Grouped by Number of Years Used for Training Model for Ranked Stats",
       caption = "Data Source: Baseball-Reference.com") +
  theme(axis.text.x = element_text(angle = 90))
```

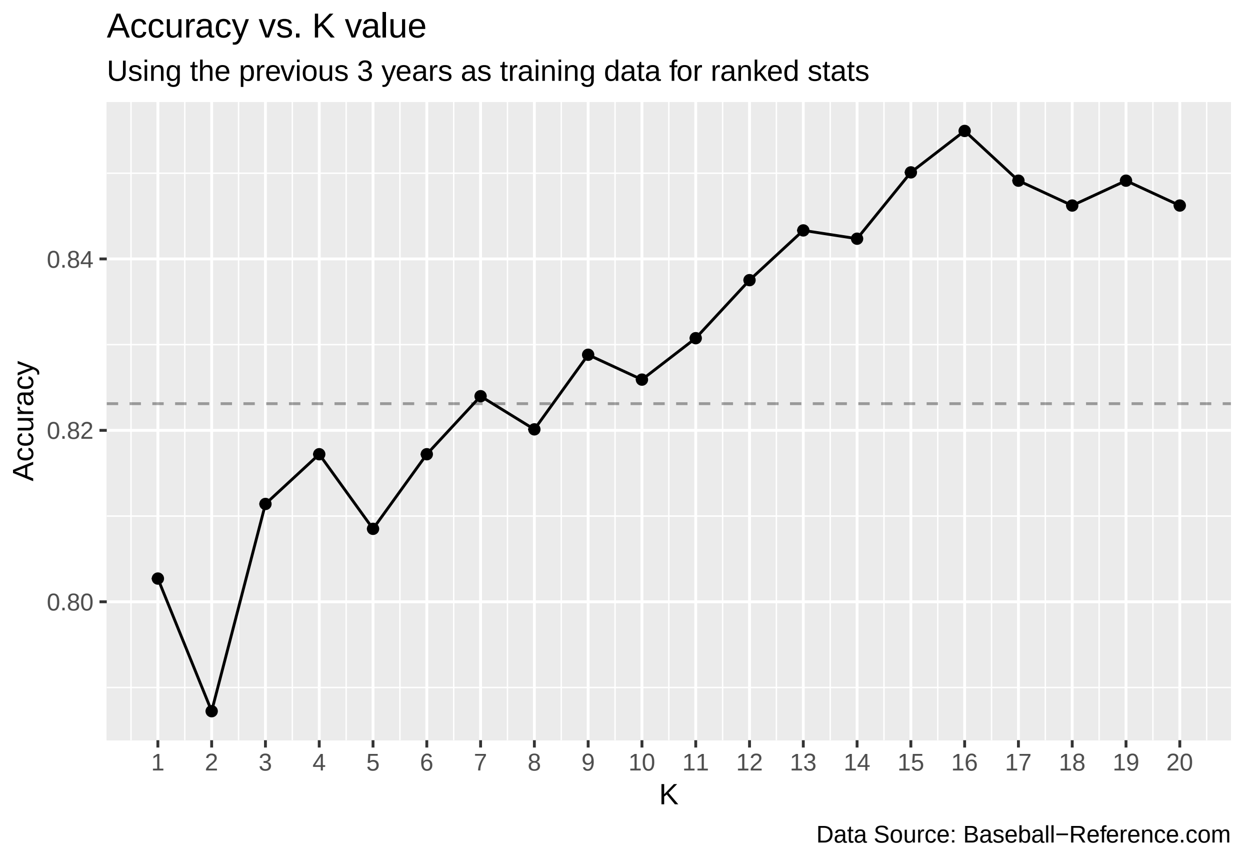


```r
# Second plot compares accuracy with years used for training, grouped by K value
ggplot(knn_tidy_ranked, aes(x = Training_Years, y = Accuracy, color = K)) +
  geom_line() +
  geom_point() +
  facet_grid(cols = vars(K)) +
  geom_hline(yintercept = median(knn_tidy_pergame$Accuracy),
             linetype = "dashed", alpha = .35) +
  labs(title = "Accuracy vs. Number of Years of KNN Model",
       subtitle = "Grouped by K Values for Ranked Stats",
       caption = "Data Source: Baseball-Reference.com") +
  theme(axis.text.x = element_text(angle = 90))
```



Notice that as the number of seasons used increases, more models have accuracies that are above the median. However, the variation seems to be the same all around, but the accuracy overall seems to increase. The best pairing seems to be using 9 years worth of prior data, with a K value around 12. However, the best overall accuracy based off the total number of prior years seems to be using 11 years of data to predict. In addition to this, it seems that overall, a K value of 11 garners the best overall accuracy, regardless of the number of years used in the training data. Let's inspect these two subsets more.

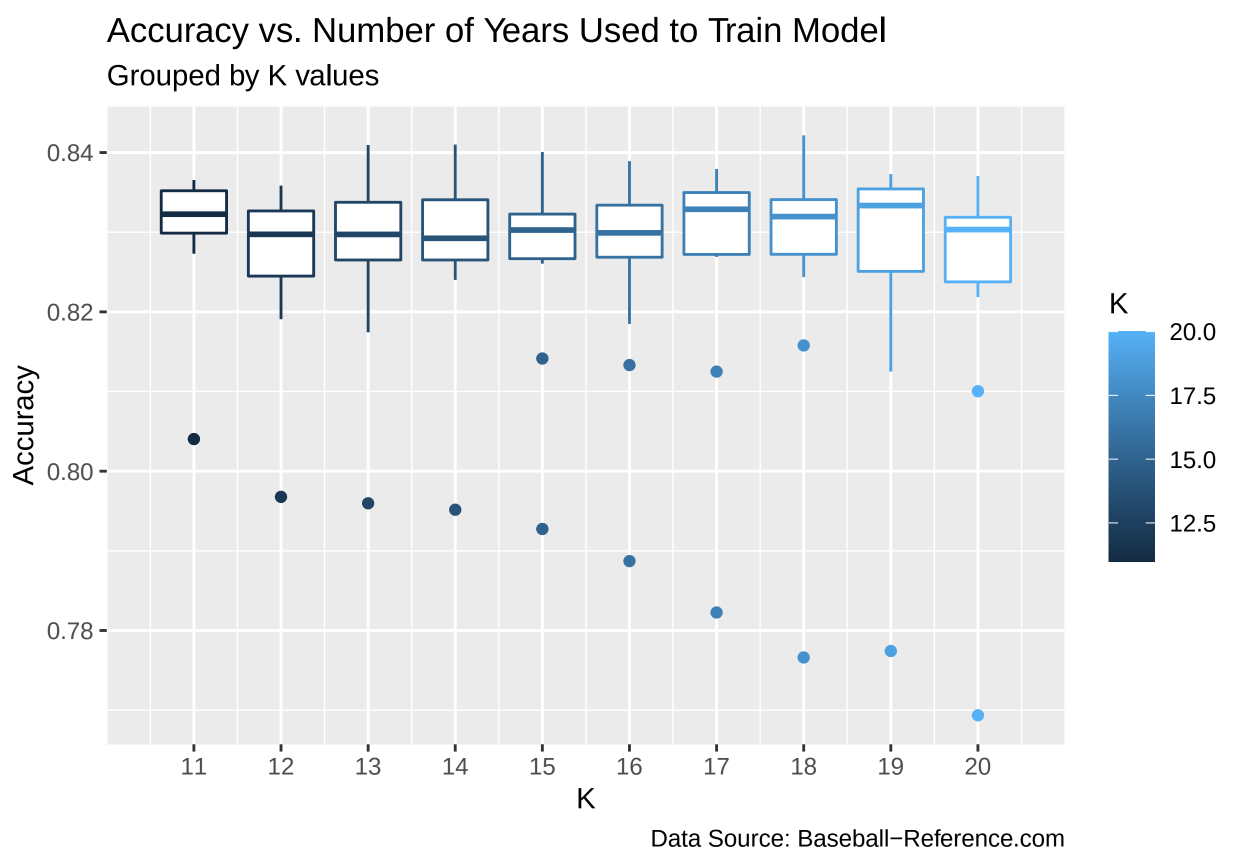
```r
# Filter the data to only include when Training_Years is 3
knn_yrs_9_ranked = knn_tidy_ranked %>% filter(Training_Years == 9)

# Graph this out
ggplot(knn_yrs_9_ranked, aes(x = K, y = Accuracy)) +
  geom_point() + geom_line() +
  geom_hline(yintercept = mean(knn_tidy_pergame$Accuracy),
             linetype = "dashed", alpha = .35) +
  labs(title = "Accuracy vs. K value",
       subtitle = "Using the previous 3 years as training data for ranked stats",
       caption = "Data Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(1, 20),
                     breaks = 1:20)
```



From the plot above, we almost get an accuracy of .852 with a K value of 16 using 11 years of prior data to create our model. This was the best accuracy among all the different combinations, as shown in the prior graph.

Let's look a subset of K's to determine which is the best overall

```r
# Filter the data to only include when Training_Years is 11
knn_yrs_sub_ranked = knn_tidy_pergame %>% filter(K >= 11 & K <= 20)

# Graph this out
ggplot(knn_yrs_sub_ranked, aes(x = K, y = Accuracy, color = K, group = K)) +
  geom_boxplot() +
  labs(title = "Accuracy vs. Number of Years Used to Train Model",
       subtitle = "Grouped by K values",
       caption = "Data Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(10.5, 20.5),
                     breaks = 11:20)
```



We see that when K = 13, we have the highest accuracy of above around .84. However, overall, once the K-value passes 10, overall they seem to have the same median prediction accuracy, with roughly the same amount of variation.

In summary: For our ranked KNN models, all K values above 11 seem to be roughly the same, with a K = 11 having the least variation

In summary: For KNN, the best number of seasons to use for training data is 11. For our K value, the best overall K value would be 11. The best combination of K values and number of seasons to use for the training data would be K = 3 and training years = 10 years before the season.

To sum everything up regarding our KNN models - no, they do not perform as well as our logistic model. The best performing models had maybe an 85% accuracy, but the amount of time taken to tune the parameters and train each model by year detracts from the model.

Simple Neural Networks

Now, we'll be using neural networks to predict playoff appearance for teams. A lot of this will be very similar as Analytics.RMD, as I'll most likely be copying the code from there, and changing as necessary.

Like with our KNN models, we will be selecting a certain number of years prior to the year we want to predict, to use as our training data.

Unlike our neural networks from Analytics.RMD, there are less variables to deal with, so hopefully, there will be less issues encountered.

Per Game Neural Networks Again, we'll be making a loop over each number of years. However, because neural networks use random weights (and depending on the start weight, may never converge), we will run a loop using seeds 1 through 5, then finding the average accuracy among all the weights, to ensure that a valid model is created. Also, because we have 13 variables in this model, we will have 1 internal layer with 14 nodes

```r
# required packages
library(nnet)

# First, pull our data in
final_pergame_data = read.csv("Datasets/model_pergame_stats.csv")[-1]

# Also pull in the data frame detailing how many MLB teams make the playoffs
playoff_num_df = read.csv("Datasets/playoff_num_df.csv")[,-1]

# Choose the years in which we will be predicting
years = unique(final_pergame_data$Year)

# Whole model using historical data

# set seed
set.seed(3)

# Create model
NN_historical_pg = nnet(Playoffs ~ ., size = 14,
                         data = final_pergame_data[,-which(names(training)
                                                        %in% c("Team", "Year"))],
                        maxit = 200,
                        trace = FALSE)

# Use previously created model to check model success rate
predFunc(NN_historical_pg, final_pergame_data, "Neural Network")
```

```
## [1] "Neural Network"
##
## comparison FALSE TRUE
##      FALSE   926   41
##      TRUE     42  255
## [1] "Accuracy: 0.934335443037975"
## [1] "Precision: 0.858585858585859"
## [1] "Recall: 0.861486486486487"
## [1] "F1 score: 0.860033726812816"
```

I believe out of all the models thus far, the neural network does the best job at predicting, as we have a high accuracy of 93% and a precision around 86%. This is really impressive. However, it is good to note that a neural network is much more complex that the other algorithms, and thus will be harder to interpret and more computationally taxing.

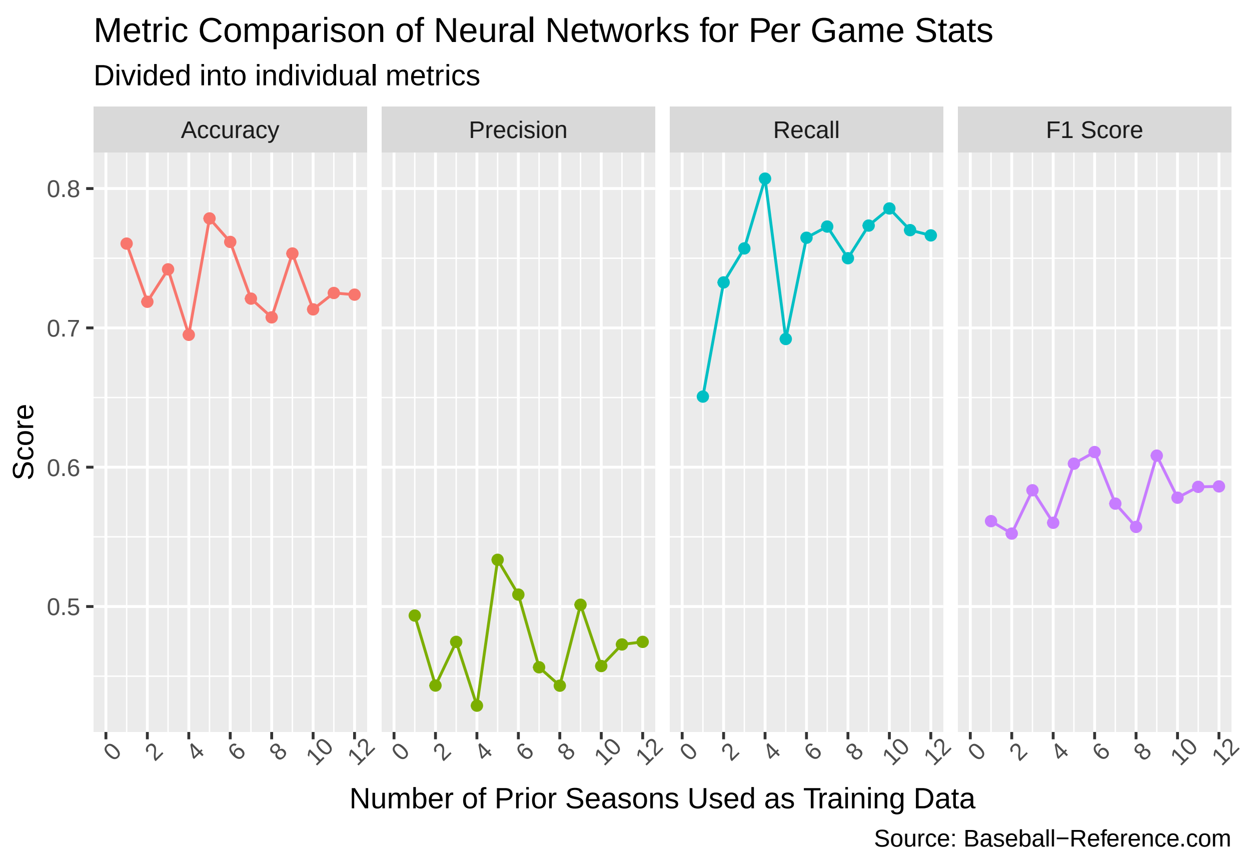
Like before, let's see if there is a subset of years that can improve the accuracy of the model, in essence using x seasons before each season to predict that season to improve our models.

```r
# Accuracy vector to store the accuracies
accuracy_nn = numeric()

# Precision vector to store precision
precision_nn = numeric()
```

```r
# recall vector to store recall
recall_nn = numeric()

# and lastly, F1_score vector
f1_score_nn = numeric()

# Start off with a loop over each season
for(num_season in 1:12) {

  # Create a vector to hold our predictions
  predictions_nn = numeric()

  # Create a logical vector to hold our predictions (to compare)
  comparison = logical()

  # Save the actual playoff appearances
  playoffs_true = final_pergame_data$Playoffs

  # Create a row index to deselect the years that weren't used to make predictions
  row_index = numeric()

  # Create a loop over each year to predict
  for(i in 1:length(years)) {

    # Check to see if we have valid data to create our training model
    if(years[i] - num_season < (min(years))) {
      # Add to our row index that no predictions were made this year
      remove_year = which(final_pergame_data$Year == years[i])
      row_index = c(row_index, remove_year)

    } else {

      # Save a counter for the number of valid models are created
      valid_model = F

      # Create a seed value starting from 1
      seed = 1

      # Loop over each seed now, to ensure a valid model is created
      while(valid_model == F) {
        # Set seed
        set.seed(seed)


        # Create our years to subset by
        years_sub = (years[i] - num_season):(years[i] - 1)

        # Subset our training and testing data
        training = final_pergame_data %>% filter(Year %in% years_sub)
        testing = final_pergame_data %>% filter(Year == years[i])

        # Create our NN model
        NN_mod_pg = nnet(Playoffs ~ ., size = 14,
                          data = training[,-which(names(training) %in%
                                                     c("Team", "Year"))],
                          trace = FALSE)

        # Create our predictions
        pred_NN_pg = predict(NN_mod_pg, testing)

        # When the neural network has issues, it will print that the
        # predictions are all 0, so we will skip when this happens
        if(sum(pred_NN_pg) == 0) {

          # Increase our seed value by 1
          seed = seed + 1

          # Go to the next iteration of new weights
          next
        } else {

          # When we get a valid model, escape our while() loop
          valid_model = TRUE

          # Save the predictions
          predictions_nn = pred_NN_pg
        }
      }
    }

    # We want to select the best n teams that make the playoffs each year
    # Now that we have a valid prediction vector, we need to select the top
    # n teams with the highest vectors

    # First, create a year index to match on the Year row, then pull the value
    year_index = which(playoff_num_df$years == years[i])
    p = playoff_num_df[year_index,2]

    # Now, create a vector of the top 4 odds
    top_odds = sort(predictions_nn, decreasing = TRUE)[1:p]

    # If values are in top odds, will become TRUE, otherwise FALSE
    comp_year = ifelse(predictions_nn %in% top_odds, TRUE, FALSE)

    # Add this to our comparison vector
    comparison = c(comparison, comp_year)
  }
  # Finally, we create our confusion matrix, and find the accuracy,
  # prediction, recall, and F1 score
  table_nn = table(comparison, playoffs_true[-row_index])

  # Pull the necessary values
  acc_nn = (table_nn[2,2] + table_nn[1,1]) / sum(table_nn)
  prec_nn = table_nn[2,2] / (table_nn[2,2] + table_nn[2,1])
  rec_nn = table_nn[2,2] / (table_nn[2,2] + table_nn[1,2])
  f1_nn = 2 * (prec_nn * rec_nn) / (prec_nn + rec_nn)
  # Store the accuracy, precision, recall, and f1 scores
  accuracy_nn = c(accuracy_nn, acc_nn)
  precision_nn = c(precision_nn, prec_nn)
  recall_nn = c(recall_nn, rec_nn)
  f1_score_nn = c(f1_score_nn, f1_nn)
}

# Create an empty data frame to store our variables of number of seasons and accuracy
nn_acc_df = data.frame(num_seasons = 1:12, accuracy_nn, precision_nn, recall_nn, f1_score_nn)

# Change the column headers
names(nn_acc_df)[2:5] = c("Accuracy", "Precision", "Recall", "F1 Score")

# Make this into tidy format, for later
nn_alt_pg = nn_acc_df %>% gather(statistic, score, 2:5)

# Turn into factors
nn_alt_pg$statistic = nn_alt_pg$statistic %>% factor(levels = c("Accuracy", "Precision", "Recall", "F1 S
```

Time to graph the models!

```
ggplot(nn_alt_pg, aes(x = num_seasons, y = score, color = statistic)) +
  geom_point() +
  geom_line() +
  facet_grid(cols = vars(statistic)) +
  theme(axis.text.x = element_text(angle = 45),
        legend.position = "none") +
  labs(title = "Metric Comparison of Neural Networks for Per Game Stats",
       subtitle = "Divided into individual metrics",
       x = "Number of Prior Seasons Used as Training Data",
       y = "Score",
       caption = "Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(0, 12),
                     breaks = seq(0, 12, by = 2))
```



In our graph above, we can cleary see that when attempting to subset our data to use for training data, it greatly hinders on every benchmark. We went from an accuracy of over .93 to an average of .75, and all the other benchmarks, except for recall, dropped significantly.

In short, using the entire dataset to predict which teams will make the playoffs yields the best results.

Neural Network with Ranked Data Similarly, we'll apply the same steps as before.

First off, let's create a neural network using our ranked data.

```
# First, pull our data in
final_ranked_data = read.csv("Datasets/model_ranked_stats.csv")[-1]


# Also pull in the data frame detailing how many MLB teams make the playoffs
playoff_num_df = read.csv("Datasets/playoff_num_df.csv")[,-1]

# Choose the years in which we will be predicting
years = unique(final_ranked_data$Year)

# Whole model using historical data

# set seed
set.seed(1)

# Create model
NN_historical_pg = nnet(Playoffs ~ ., size = 19,
                                data = final_ranked_data[,-which(names(training)
                                                                    %in% c("Team", "Year"))],
                               maxit = 200,
                               trace = FALSE)

# Use previously created model to check model success rate
predFunc(NN_historical_pg, final_ranked_data, "Neural Network")
```

```
## [1] "Neural Network"
##
## comparison FALSE TRUE
##      FALSE   941   26
##      TRUE     27  270
## [1] "Accuracy: 0.958069620253165"
## [1] "Precision: 0.909090909090909"
## [1] "Recall: 0.912162162162162"
## [1] "F1 score: 0.910623946037099"
```

The neural network, using our ranked data, performs the best out of all our models thus far. We get an incredibly high accuracy value of .958, with precision, recall, and f1 scores all around .91. While the neural network is by far the most complex, we also get the best prediction metrics of all the models thus far, using our ranked data.

Let's see if a subset of the data, using a certain number of years prior to predict the next year, will improve our models.

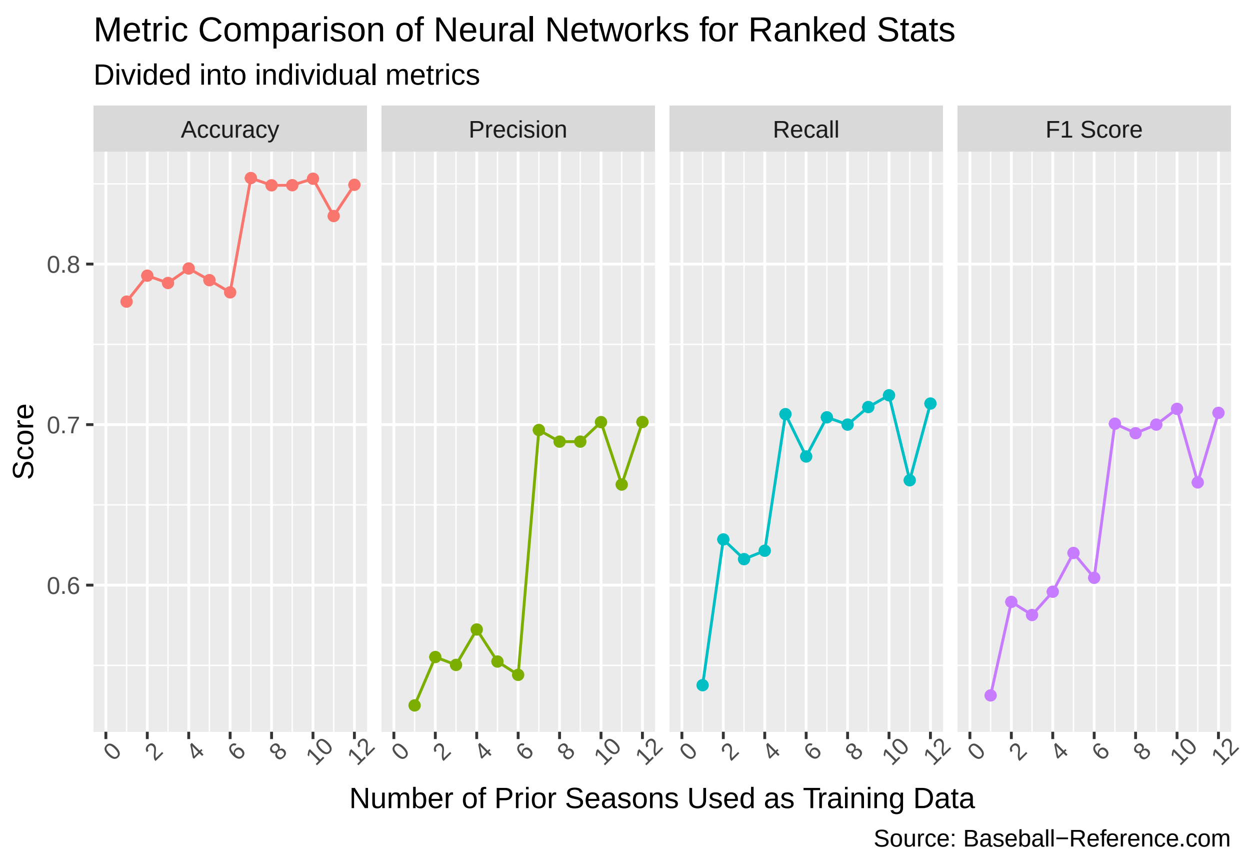
```
# Accuracy vector to store the accuracies
accuracy_nn = numeric()

# Precision vector to store precision
precision_nn = numeric()

# recall vector to store recall
recall_nn = numeric()

# and lastly, F1_score vector
f1_score_nn = numeric()

# Start off with a loop over each season
for(num_season in 1:12) {

  # Create a vector to hold our predictions
  predictions_nn = numeric()

  # Create a logical vector to hold our predictions (to compare)
  comparison = logical()

  # Save the actual playoff appearances
  playoffs_true = final_ranked_data$Playoffs

  # Create a row index to deselect the years that weren't used to make predictions
  row_index = numeric()
```

```r
# Create a loop over each year to predict
for(i in 1:length(years)) {

  # Check to see if we have valid data to create our training model
  if(years[i] - num_season < (min(years))) {
    # Add to our row index that no predictions were made this year
    remove_year = which(final_ranked_data$Year == years[i])
    row_index = c(row_index, remove_year)

  } else {

    # Save a counter for the number of valid models are created
    valid_model = F

    # Create a seed value starting from 1
    seed = 1

    # Loop over each seed now, to ensure a valid model is created
    while(valid_model == F) {
      # Set seed
      set.seed(seed)


      # Create our years to subset by
      years_sub = (years[i] - num_season):(years[i] - 1)

      # Subset our training and testing data
      training = final_ranked_data %>% filter(Year %in% years_sub)
      testing = final_ranked_data %>% filter(Year == years[i])

      # Create our NN model
      NN_mod_rk = nnet(Playoffs ~ ., size = 19,
                       data = training[,-which(names(training) %in%
                                                 c("Team", "Year"))],
                       trace = FALSE)

      # Create our predictions
      pred_NN_rk = predict(NN_mod_rk, testing)

      # When the neural network has issues, it will print that the
      # predictions are all 0, so we will skip when this happens
      if(sum(pred_NN_rk) == 0) {

        # Increase our seed value by 1
        seed = seed + 1

        # Go to the next iteration of new weights
        next
      } else {

        # When we get a valid model, escape our while() loop
        valid_model = TRUE
```

```
          # Save the predictions
          predictions_nn = pred_NN_rk
        }
      }
    }

    # We want to select the best n teams that make the playoffs each year
    # Now that we have a valid prediction vector, we need to select the top
    # n teams with the highest vectors

    # First, create a year index to match on the Year row, then pull the value
    year_index = which(playoff_num_df$years == years[i])
    p = playoff_num_df[year_index,2]

    # Now, create a vector of the top 4 odds
    top_odds = sort(predictions_nn, decreasing = TRUE)[1:p]

    # If values are in top odds, will become TRUE, otherwise FALSE
    comp_year = ifelse(predictions_nn %in% top_odds, TRUE, FALSE)

    # Add this to our comparison vector
    comparison = c(comparison, comp_year)
  }
  # Finally, we create our confusion matrix, and find the accuracy,
  # prediction, recall, and F1 score
  table_nn = table(comparison, playoffs_true[-row_index])

  # Pull the necessary values
  acc_nn = (table_nn[2,2] + table_nn[1,1]) / sum(table_nn)
  prec_nn = table_nn[2,2] / (table_nn[2,2] + table_nn[2,1])
  rec_nn = table_nn[2,2] / (table_nn[2,2] + table_nn[1,2])
  f1_nn = 2 * (prec_nn * rec_nn) / (prec_nn + rec_nn)

  # Store the accuracy, precision, recall, and f1 scores
  accuracy_nn = c(accuracy_nn, acc_nn)
  precision_nn = c(precision_nn, prec_nn)
  recall_nn = c(recall_nn, rec_nn)
  f1_score_nn = c(f1_score_nn, f1_nn)
}

# Create an empty data frame to store our variables of number of seasons and accuracy
nn_acc_df = data.frame(num_seasons = 1:12, accuracy_nn, precision_nn, recall_nn, f1_score_nn)

# Change the column headers
names(nn_acc_df)[2:5] = c("Accuracy", "Precision", "Recall", "F1 Score")

# Make this into tidy format, for later
nn_alt_rk = nn_acc_df %>% gather(statistic, score, 2:5)

# Turn into factors
nn_alt_rk$statistic = nn_alt_rk$statistic %>% factor(levels = c("Accuracy", "Precision", "Recall", "F1 S
```

Now, let's graph this!

```r
ggplot(nn_alt_rk, aes(x = num_seasons, y = score, color = statistic)) +
  geom_point() +
  geom_line() +
  facet_grid(cols = vars(statistic)) +
  theme(axis.text.x = element_text(angle = 45),
        legend.position = "none") +
  labs(title = "Metric Comparison of Neural Networks for Ranked Stats",
       subtitle = "Divided into individual metrics",
       x = "Number of Prior Seasons Used as Training Data",
       y = "Score",
       caption = "Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(0, 12),
                     breaks = seq(0, 12, by = 2))
```



In our graph here, we see similar results as when we investigated our per game statistics. All the metrics take a hit when only using a subset of the data to create our models. And like our per game data, we see that each statistics follows a similar shape. A key difference to note here is that the neural network does improve with more training data. This is evident as our scoring metrics tend to increase with the greater number of seasons used to train the model.

Naive Bayes Modeling

For our last model, we will be using naive bayes to predict playoff appearances for teams.

Per Game Statistics Like before, we will start off with using our historical data, then use the predFunc() function we built before to predict the teams that make the playoffs each year. Then, we'll calculate the different scoring metrics to determine how well naive bayes does at predicting playoff appearances.

```
# Set our seed
set.seed(5)

# First, pull in our data
final_pergame_data = read.csv("Datasets/model_pergame_stats.csv")[,-1]

# Create a vector of our unique years
years = unique(final_pergame_data$Year)

# Index the columns to not include in our model
index_cols = which(names(final_pergame_data) %in% c("Team", "Year"))

# Create empty vectors to store our values
acc_nb = numeric()
prec_nb = numeric()
rec_nb = numeric()
f1_nb = numeric()

# Create our model using historical data
nb_historical_pg = naivebayes::naive_bayes(Playoffs ~ .,
                                           data = final_pergame_data[,-index_cols])

# Because the naive_bayes function returns True and False values,
# We must manually find the accuracy, precision, recall, and f1 scores ourselves

# Create our confusion matrix
nb_predictions_pg = predict(nb_historical_pg, final_pergame_data)
(table_nb = table(nb_predictions_pg, final_pergame_data$Playoffs))
```

```
##
## nb_predictions_pg FALSE TRUE
##             FALSE   857   96
##             TRUE    111  200
```

```
# Find our metrics
acc_nb = (table_nb[2,2] + table_nb[1,1]) / sum(table_nb)
prec_nb = table_nb[2,2] / (table_nb[2,2] + table_nb[2,1])
rec_nb = table_nb[2,2] / (table_nb[2,2] + table_nb[1,2])
f1_nb = 2 * (prec_nb * rec_nb) / (prec_nb + rec_nb)

# Print the values out
print(paste("Accuracy: ", acc_nb, sep = ""))
```

```
## [1] "Accuracy: 0.83623417721519"
```

```
print(paste("Precision: ", prec_nb, sep = ""))
```

```
## [1] "Precision: 0.643086816720257"
```

```r
print(paste("Recall: ", rec_nb, sep = ""))
```

```
## [1] "Recall: 0.675675675675676"
```

```r
print(paste("F1 score: ", f1_nb, sep = ""))
```

```
## [1] "F1 score: 0.658978583196046"
```

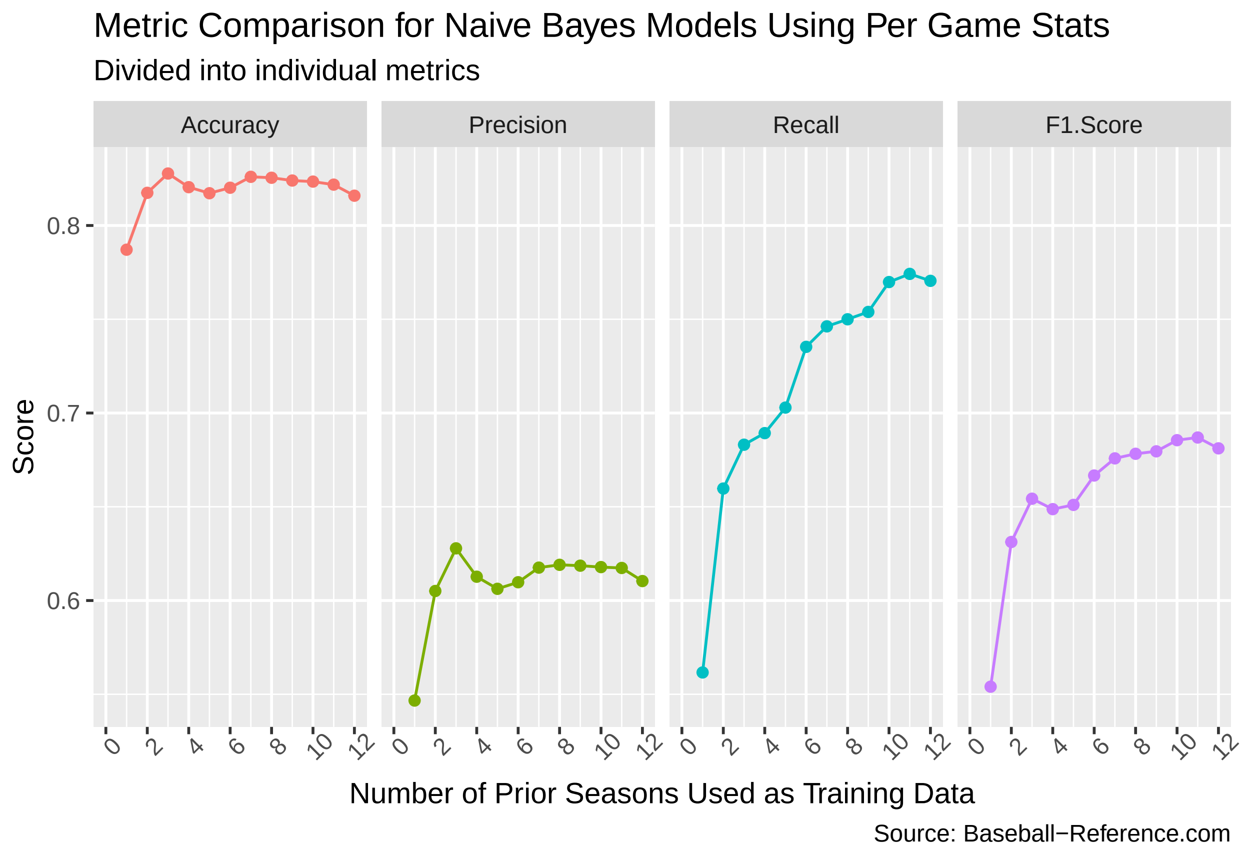
From our output above, we see that naive bayes does not seem to be a good model with our data. Not only are the scoring metrics quite low (accuracy around .84, and precision, recall, and F1 score all around .65), but this model also heavily favors making false positive predictions.

Let's see if we can find a subset of the data that may improve this model.

```r
# Pull in our data
final_pergame_data = read.csv("Datasets/model_pergame_stats.csv")[,-1]

# Work over certain years
years = unique(final_pergame_data$Year)

# Create empty vectors to store our values in
accuracy_nb = numeric()
precision_nb = numeric()
recall_nb = numeric()
f1_score_nb = numeric()

# Now, we'll loop over how many years we'll use in our training data, 1:12
for(num_season in 1:12) {

  # Save the true playoff values
  playoffs_true = final_pergame_data$Playoffs

  # Create a logical vector to hold our comparisons
  comparison = logical()

  # Create a row index for the years we won't be predicting
  row_index = numeric()

  # Loop over each year to predict
  for(i in 1:length(years)) {

    # Check if we have valid training data
    if(years[i] - num_season < min(years)) {

      # Add the years we won't be predicting to our index to remove later
      remove_year = which(final_pergame_data$Year == years[i])
      row_index = c(row_index, remove_year)

    } else {

      # Start with years to subset by
      years_sub = (years[i] - num_season):(years[i] - 1)
      # Create our training and testing data
      training = final_pergame_data %>% filter(Year %in% years_sub)
      testing = final_pergame_data %>% filter(Year == years[i])

      # Create our model
      nb_mod_pg = naivebayes::naive_bayes(
        Playoffs ~ .,
        data = training[,-index_cols])

      # Predict and store the values
      comp_nb = predict(nb_mod_pg, testing)

      # Save these in our comparison
      comparison = c(comparison, comp_nb)
    }
  }

  # Finally, we'll create our own confusion matrix to find all the metrics
  table_nb = table(comparison, playoffs_true[-row_index])

  # Pull the necessary values
  acc_nb = (table_nb[2,2] + table_nb[1,1]) / sum(table_nb)
  prec_nb = table_nb[2,2] / (table_nb[2,2] + table_nb[2,1])
  rec_nb = table_nb[2,2] / (table_nb[2,2] + table_nb[1,2])
  f1_nb = 2 * (prec_nb * rec_nb) / (prec_nb + rec_nb)

  # Store the accuracy, precision, recall, and f1 scores
  accuracy_nb = c(accuracy_nb, acc_nb)
  precision_nb = c(precision_nb, prec_nb)
  recall_nb = c(recall_nb, rec_nb)
  f1_score_nb = c(f1_score_nb, f1_nb)
}

# Store all the metrics and number of seasons as a data frame
nb_acc_df = data.frame(num_seasons = 1:12, accuracy_nb, precision_nb,
                       recall_nb, f1_score_nb)

# Change the column headers
names(nb_acc_df)[2:5] = c("Accuracy", "Precision", "Recall", "F1.Score")

# Convert to tidy format
nb_alt_pg = nb_acc_df %>% gather("Statistic", "Score", 2:5)

# Convert the statistics into factors
nb_alt_pg$Statistic = nb_alt_pg$Statistic %>%
  factor(levels = c("Accuracy", "Precision", "Recall", "F1.Score"))
```

Now, to graph our models to compare scoring metrics!

```r
ggplot(nb_alt_pg, aes(x = num_seasons, y = Score, color = Statistic)) +
  geom_point() +
  geom_line() +
  facet_grid(cols = vars(Statistic)) +
  theme(axis.text.x = element_text(angle = 45),
        legend.position = "none") +
  labs(title = "Metric Comparison for Naive Bayes Models Using Per Game Stats",
       subtitle = "Divided into individual metrics",
       x = "Number of Prior Seasons Used as Training Data",
       y = "Score",
       caption = "Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(0, 12),
                     breaks = seq(0, 12, by = 2))
```



From our output above, we see that naive bayes models tend to do better when more data is available for training. However, when comparing to the model using the historical data, we see that the accuracy when subsetting our data is around the same as the accuracy when using all the data provided. Our recall value, around .75, is roughly 10% higher than it was before (~ .675). Our precision here has decreased by around .05%, and our F1 score has roughly stayed about the same/potentially increased by a few percentage points.

Naive Bayes on Ranked Data Now, we'll be looking at our ranked baseball statistics, first making a model of the cumulative data and seeing how well it does, and then going season by season to see if there is a subset of data that does the best job at predicting playoff appearance.

```r
# Set our seed
set.seed(5)

# First, pull in our data
final_ranked_data = read.csv("Datasets/model_ranked_stats.csv")[,-1]
```

```r
# Create a vector of our unique years
years = unique(final_ranked_data$Year)

# Index the columns to not include in our model
index_cols = which(names(final_ranked_data) %in% c("Team", "Year"))

# Create empty vectors to store our values
acc_nb = numeric()
prec_nb = numeric()
rec_nb = numeric()
f1_nb = numeric()

# Create our model using historical data
nb_historical_rk = naivebayes::naive_bayes(Playoffs ~ .,
                                           data = final_ranked_data[,-index_cols])

# Because the naive_bayes function returns True and False values,
# We must manually find the accuracy, precision, recall, and f1 scores ourselves

# Create our confusion matrix
nb_predictions_rk = predict(nb_historical_rk, final_ranked_data)
(table_nb = table(nb_predictions_rk, final_ranked_data$Playoffs))
```

```
##
## nb_predictions_rk FALSE TRUE
##             FALSE   817   49
##             TRUE    151  247
```

```r
# Find our metrics
acc_nb = (table_nb[2,2] + table_nb[1,1]) / sum(table_nb)
prec_nb = table_nb[2,2] / (table_nb[2,2] + table_nb[2,1])
rec_nb = table_nb[2,2] / (table_nb[2,2] + table_nb[1,2])
f1_nb = 2 * (prec_nb * rec_nb) / (prec_nb + rec_nb)

# Print the values out
print(paste("Accuracy: ", acc_nb, sep = ""))
```

```
## [1] "Accuracy: 0.841772151898734"
```

```r
print(paste("Precision: ", prec_nb, sep = ""))
```

```
## [1] "Precision: 0.620603015075377"
```

```r
print(paste("Recall: ", rec_nb, sep = ""))
```

```
## [1] "Recall: 0.834459459459459"
```

```r
print(paste("F1 score: ", f1_nb, sep = ""))
```

```
## [1] "F1 score: 0.711815561959654"
```

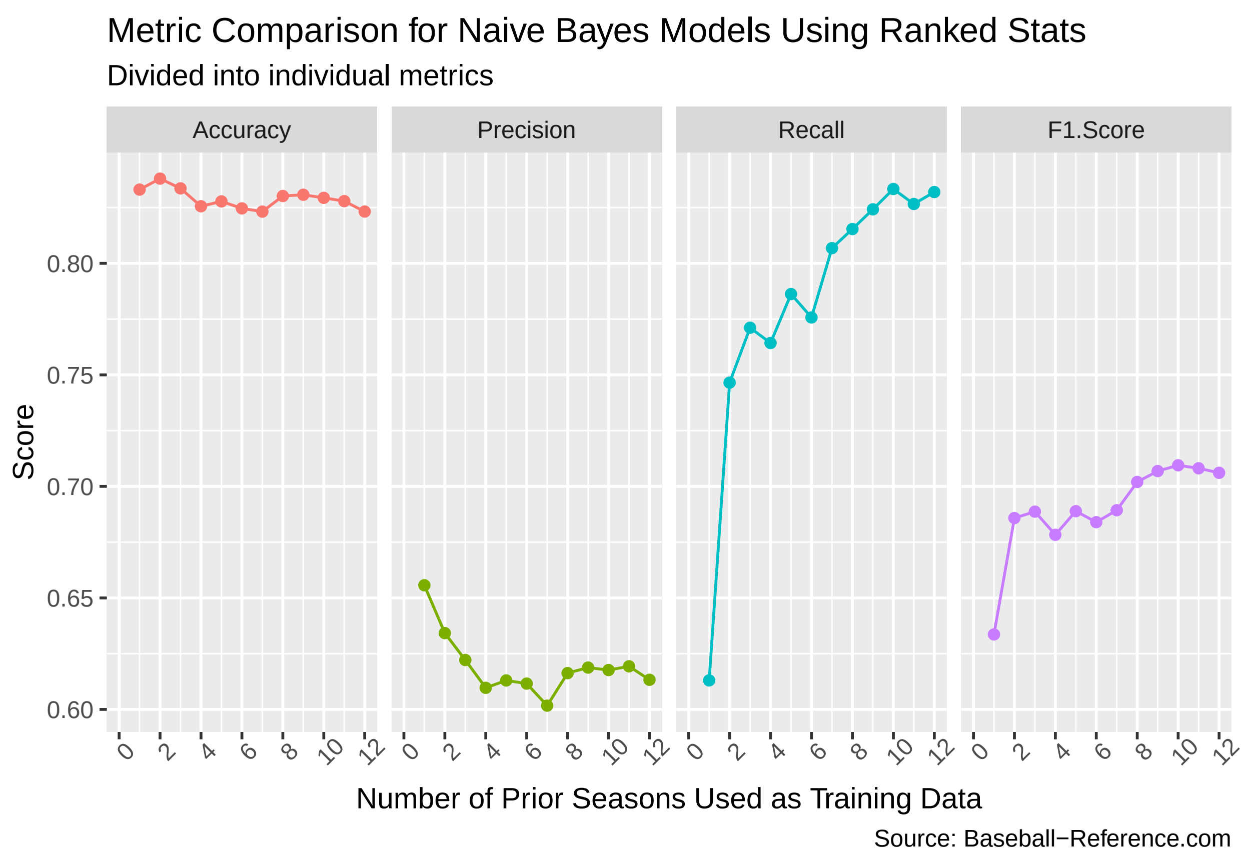
From our output above, we see that naive bayes does not seem to be a good model with our data. Not only are the scoring metrics quite low (accuracy around .84, precision, and F1 score both below .72, and recall around .83), but this model also heavily favors making false positive predictions. These results mirror the same findings when using the full dataset on the per game data.

Let's see if we can find a subset of the data that may improve this model.

```r
# Pull in our data
final_ranked_data = read.csv("Datasets/model_ranked_stats.csv")[,-1]

# Work over certain years
years = unique(final_pergame_data$Year)

# Create empty vectors to store our values in
accuracy_nb = numeric()
precision_nb = numeric()
recall_nb = numeric()
f1_score_nb = numeric()

# Now, we'll loop over how many years we'll use in our training data, 1:12
for(num_season in 1:12) {

  # Save the true playoff values
  playoffs_true = final_ranked_data$Playoffs

  # Create a logical vector to hold our comparisons
  comparison = logical()

  # Create a row index for the years we won't be predicting
  row_index = numeric()

  # Loop over each year to predict
  for(i in 1:length(years)) {

    # Check if we have valid training data
    if(years[i] - num_season < min(years)) {

      # Add the years we won't be predicting to our index to remove later
      remove_year = which(final_ranked_data$Year == years[i])
      row_index = c(row_index, remove_year)

    } else {

      # Start with years to subset by
      years_sub = (years[i] - num_season):(years[i] - 1)

      # Create our training and testing data
      training = final_ranked_data %>% filter(Year %in% years_sub)
      testing = final_ranked_data %>% filter(Year == years[i])

      # Create our model
      nb_mod_rk = naivebayes::naive_bayes(
        Playoffs ~ .,
        data = training[,-index_cols])
      # Predict and store the values
      comp_nb = predict(nb_mod_rk, testing)

      # Save these in our comparison
      comparison = c(comparison, comp_nb)
    }
  }

  # Finally, we'll create our own confusion matrix to find all the metrics
  table_nb = table(comparison, playoffs_true[-row_index])

  # Pull the necessary values
  acc_nb = (table_nb[2,2] + table_nb[1,1]) / sum(table_nb)
  prec_nb = table_nb[2,2] / (table_nb[2,2] + table_nb[2,1])
  rec_nb = table_nb[2,2] / (table_nb[2,2] + table_nb[1,2])
  f1_nb = 2 * (prec_nb * rec_nb) / (prec_nb + rec_nb)

  # Store the accuracy, precision, recall, and f1 scores
  accuracy_nb = c(accuracy_nb, acc_nb)
  precision_nb = c(precision_nb, prec_nb)
  recall_nb = c(recall_nb, rec_nb)
  f1_score_nb = c(f1_score_nb, f1_nb)
}

# Store all the metrics and number of seasons as a data frame
nb_acc_df = data.frame(num_seasons = 1:12, accuracy_nb, precision_nb,
                       recall_nb, f1_score_nb)

# Change the column headers
names(nb_acc_df)[2:5] = c("Accuracy", "Precision", "Recall", "F1.Score")

# Convert to tidy format
nb_alt_rk = nb_acc_df %>% gather("Statistic", "Score", 2:5)

# Convert the statistics into factors
nb_alt_rk$Statistic = nb_alt_pg$Statistic %>%
  factor(levels = c("Accuracy", "Precision", "Recall", "F1.Score"))
```

Now, to graph our models to compare scoring metrics!

```r
ggplot(nb_alt_rk, aes(x = num_seasons, y = Score, color = Statistic)) +
  geom_point() +
  geom_line() +
  facet_grid(cols = vars(Statistic)) +
  theme(axis.text.x = element_text(angle = 45),
        legend.position = "none") +
  labs(title = "Metric Comparison for Naive Bayes Models Using Ranked Stats",
       subtitle = "Divided into individual metrics",
       x = "Number of Prior Seasons Used as Training Data",
       y = "Score",
       caption = "Source: Baseball-Reference.com") +
  scale_x_continuous(limits = c(0, 12),
                     breaks = seq(0, 12, by = 2))
```



From our output above, we get mixed results. Regardless of how many years are used as training data, the accuracy of the different models tend to hover around .825. However, precision, recall, and f1 score have trends dependent on training data. When increasing the number of years used, our precision decreases, eventually hovering around the .625 mark. Recall and F1 score actually increases with number of seasons used in our training. Recall improves to have similar success as accuracy (around .825), while F1 score hovers around .70. This is different that our per game statistics, where we saw each metric's value increase with the larger datasets used.