An Analysis of Probabilistic Factors in Association of Tennis Professionals Winners

Final Project

Name: Rares Finatan
Student Number: 685688202
Course: IST 687
Term: Fall/Winter 2022

Section 1 - Abstract

Of all the major competitive sporting events occurring on an annual basis, tennis is arguable one of the best suited for advanced analytical analysis due to its large amount of data generated on a per-match basis[1]. To better understand the factors involved in what determines a winner in a given tennis match, chronological data across a vast recorded match history is required.

This study aims to identify the highest-contributing attributes of an Association of Tennis Professionals (ATP) winner's performance for the years 2000 to 2022. The study is aimed at tennis fans and sports betting enthusiasts looking to gain an understanding of a player's performance from a list of hundreds of ATP-registered players. In addition to standard match data available via public repositories, the study will attempt to engineer features for modelling pertaining to player matchups, environmental scenarios, and tournament-specific performance.

The data will be modelled and evaluated using several random forests algorithms. Baseline accuracy will be established by a default random forest, accompanied by several tuned random forests, and a naive Bayes classifier. Lastly, the models will be evaluated against one another for model-specific accuracy.

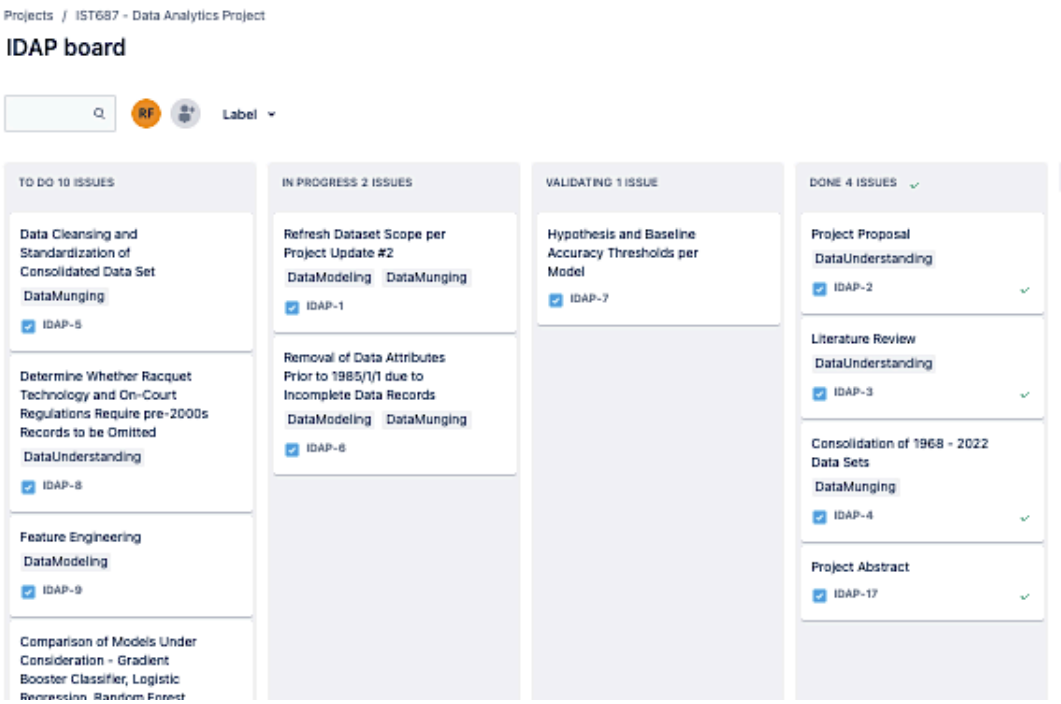
[1] Tandon, K. (2020, January 17). *The role of analytics in tennis is on a long, slow rise*. Tennis.com. Retrieved December 18, 2022, from https://www.tennis.com/news/articles/the-role-of-analytics-in-tennis-is-on-a-long-slow-rise

Section 2 - Data Description and Methodology

The data under study consists of 22 comma-separated value files tracking ATP-level tennis matches on an annual basis sourced by Jeff Sackmann[2]. The files contain information about ATP players, including their IDs, names, hand preferences, birth dates, countries of origin, and heights. Ranking information is also included, with data available from 2000 to the present. Results and statistics for ATP matches are included in separate files for each season, covering tour-level main draw matches, tour-level qualifying and challenger main-draw matches, and futures matches. These files contain biographical information and ranking data for each player, as well as age and ranking points as of the start of the event. MatchStats, which provide detailed statistics about the matches, are available for tour-level matches from 2000 to 2022, for challengers from 2008 to the 2022, and for tour-level qualifying matches from 2011 to 2022. Some tour-level matches may be missing statistics due to unavailability from ATP. Davis Cup matches are included in the tour-level files, but do not have MatchStats available until recent seasons.

A Kanban board in Jira was used to visualize the progress of completed work by organizing it into columns and cards. The board and its contents were used to guide the process of completing elementary data analysis, generation of summary statistics, data munging and cleaning, model preparation, model building, and model evaluation.

[2] Sackmann, J. (n.d.). *ATP Tennis Rankings, results, and stats*. GitHub. Retrieved December 18, 2022, from https://github.com/JeffSackmann/tennis_atp



A partial screenshot of the Kanban board used as part of the project, showcasing backlogged items To-Do, In-Progress, Validating, and Complete stages.

Section 3 - Data Analysis

3.1 - Data Import

Primary data analysis was conducted to better understand the data, its structure, its distribution, and analyze any anomalies.

Analysis begins with compiling all of the downloaded data sets. Create a data frame of all the file names, then filter for files ending in `'atp_matches_20'`, where the ending string denotes datasets for the year 2000 and onwards.

```
#Import dependencies
library(tidyverse)
library(caret)
library(kernlab)
library(lubridate)
library(DataExplorer)
library(skimr)
library(lares)
```

```r
#Extract relevant file names from local forked repo
df <- data.frame(list.files(path = "~/Documents/Masters/Syracuse/IST 687/FINAL_PROJECT/tennis_atp-master", pattern = "*.csv"))

df <- df %>% rename(file_name = list.files.path......Documents.Masters.Syracuse.IST.687.FINAL_PROJECT.tennis_atp.master...)

df <- df %>% filter(grepl('atp_matches_20', file_name))

file_list <- paste('./tennis_atp-master/', df$file_name, sep = '')
file_list
```

```
 [1] "./tennis_atp-master/atp_matches_2000.csv"
 [2] "./tennis_atp-master/atp_matches_2001.csv"
 [3] "./tennis_atp-master/atp_matches_2002.csv"
 [4] "./tennis_atp-master/atp_matches_2003.csv"
 [5] "./tennis_atp-master/atp_matches_2004.csv"
 [6] "./tennis_atp-master/atp_matches_2005.csv"
 [7] "./tennis_atp-master/atp_matches_2006.csv"
 [8] "./tennis_atp-master/atp_matches_2007.csv"
 [9] "./tennis_atp-master/atp_matches_2008.csv"
[10] "./tennis_atp-master/atp_matches_2009.csv"
[11] "./tennis_atp-master/atp_matches_2010.csv"
[12] "./tennis_atp-master/atp_matches_2011.csv"
[13] "./tennis_atp-master/atp_matches_2012.csv"
[14] "./tennis_atp-master/atp_matches_2013.csv"
[15] "./tennis_atp-master/atp_matches_2014.csv"
[16] "./tennis_atp-master/atp_matches_2015.csv"
[17] "./tennis_atp-master/atp_matches_2016.csv"
[18] "./tennis_atp-master/atp_matches_2017.csv"
[19] "./tennis_atp-master/atp_matches_2018.csv"
[20] "./tennis_atp-master/atp_matches_2019.csv"
[21] "./tennis_atp-master/atp_matches_2020.csv"
[22] "./tennis_atp-master/atp_matches_2021.csv"
[23] "./tennis_atp-master/atp_matches_2022.csv"
```

Data import of all datasets under study.

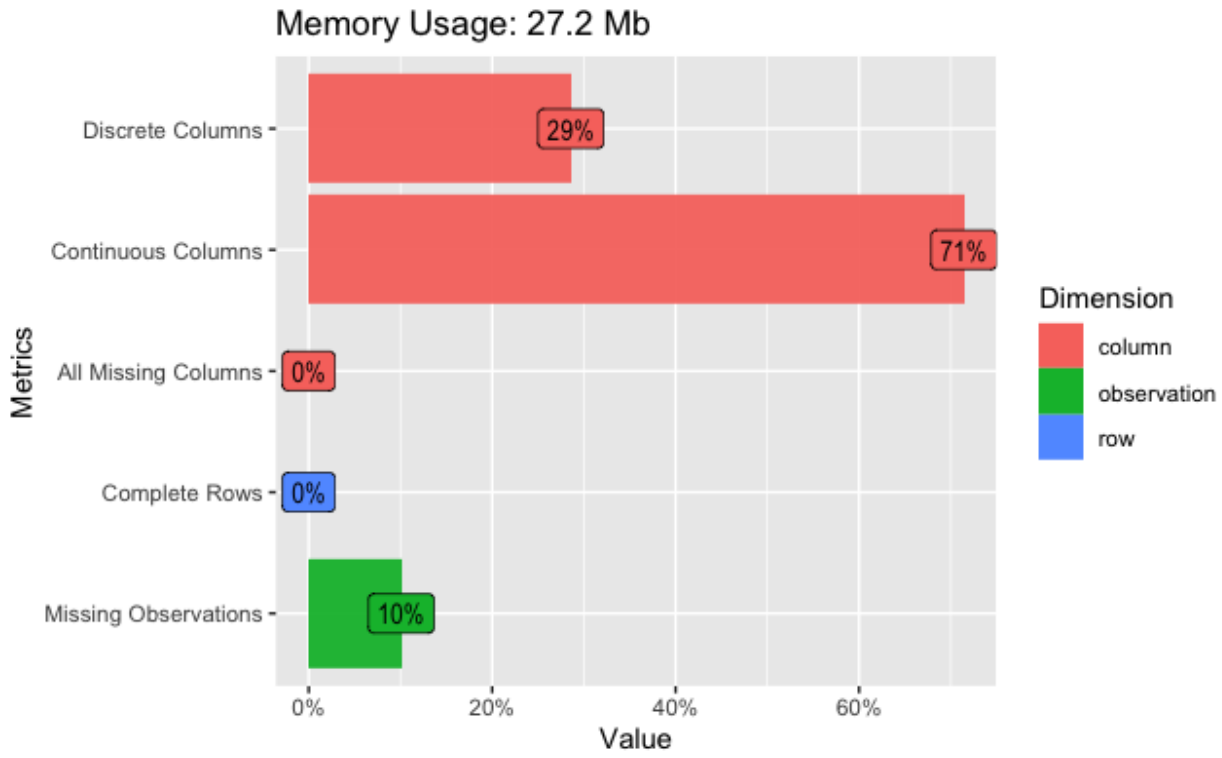
Lastly, merge all the file's contents into a singular data frame, named `merged_df`, which will be iteratively used from herein to clean the raw files' data contents.

```r
#Merge the contents of the files into singular data frame
merged_df <- file_list %>% map_df(~read_csv(., show_col_types = FALSE, col_types = list(winner_seed = 'd', loser_seed = 'd')))
```

3.2 - Exploratory Data Analysis

Data analysis continues with a fundamental visualization of the data, including its allocation size to system memory, discrete columns, continuous columns, missing data columns, complete rows, and missing observations. This information is visualized via one line using the `DataExplorer` package via the `DataExplore::plot_intro()` command.

```
# Basic analysis of data set
plot_intro(merged_df)
```



3.3 - Distribution of Data

Next, one can analyze the distribution of the data, firstly by year. The original data set contains match data pertaining to each match date, but it is consolidated into one concatenated attribute. Split the date attribute into year, month, and day columns.

```
#Apply consistent date formatting
merged_df$tourney_date <- ymd(merged_df$tourney_date)
```

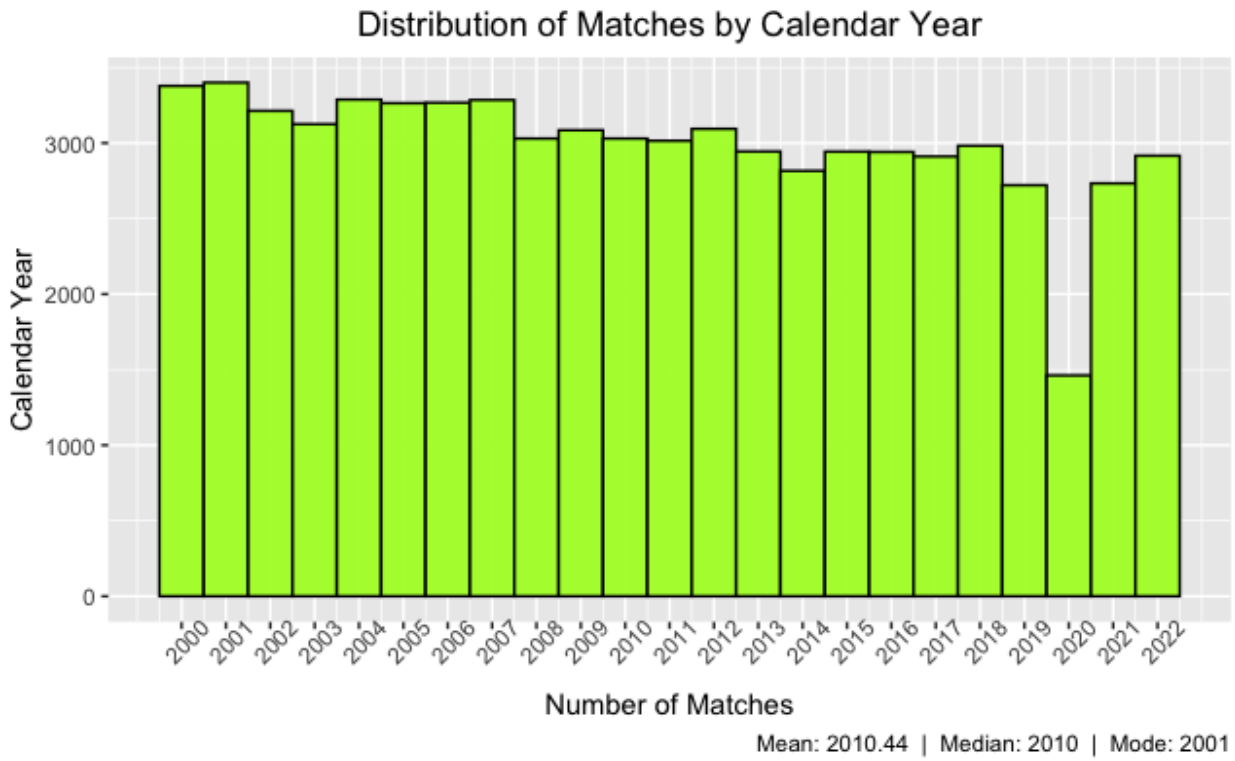
```
#Create year, month, and date columns
merged_df$tourney_year <- as.numeric(strftime(merged_df$tourney_date, 
'%Y'))
merged_df$tourney_month <- as.numeric(strftime(merged_df$tourney_date, 
'%m'))
merged_df$tourney_day <- as.numeric(strftime(merged_df$tourney_date, 
'%d'))
```

Afterwards, instantiate variables containing the mean, median, and mode of the match years, and store their respective values as a character string in `central_tendency_metrics`.

```
#Create variable inclusive of summary statistics
mean <- round(mean(merged_df$tourney_year),2)
median <- median(merged_df$tourney_year)
mode <- table(merged_df$tourney_year)
mode <- as.numeric(names(mode[which(mode == max(mode))]))
central_tendency_metrics <- paste(sprintf('Mean: %s', mean), 
sprintf('Median: %s', median), sprintf('Mode: %s', mode), sep = '   | 
')
```

The variable storing the central tendency metrics is then subsequently used in a `ggplot` caption via a command to show the distribution of matches by calendar year. A histogram is plotted with a green-yellow color, representative of a tennis ball. Appropriate titles are associated with the plot, and labels are given to the x and y axes respectively.

```
#Plot distribution of match data, by year
ggplot(merged_df, aes(x = tourney_year)) +
  geom_histogram(color = 'black', fill = 'greenyellow', bins = 23) +
  scale_x_continuous(breaks=seq(2000,2022,1)) +
  theme(axis.text.x = element_text(angle = 45), plot.title = 
element_text(hjust = 0.5)) +
  ggtitle('Distribution of Matches by Calendar Year') +
  xlab('Number of Matches') +
  ylab('Calendar Year') +
  labs(caption = central_tendency_metrics)
```



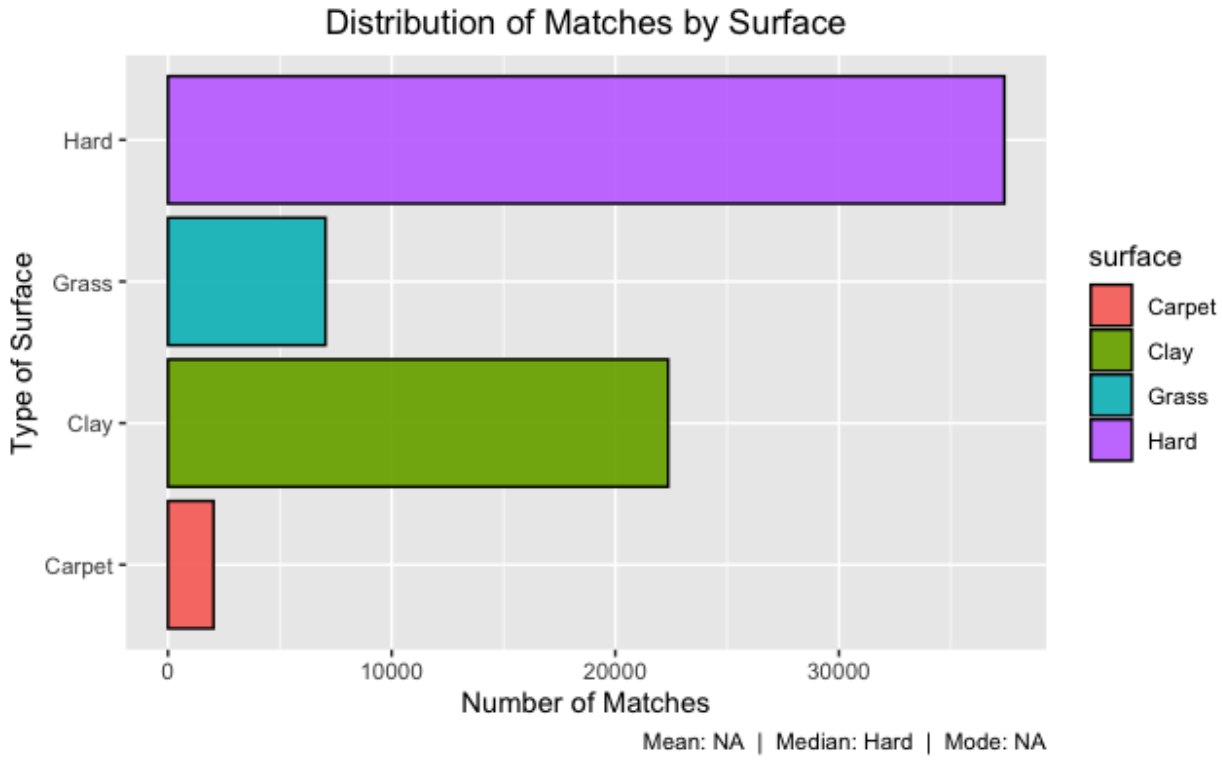
One can also analyze the distribution of match data by surface. Tennis is played on many surface types, but the most common ones are hard-court, clay-court, and grass-court surfaces.

Repeat the same procedure of measures of central tendency outlined for the year attribute, but this time for the surface attribute. This time, the values return NAs, because the categorical values cannot be quantified using the built in functions for summary statistics.

```
#Create variable inclusive of summary statistics
mean <- round(mean(merged_df$surface),2)
median <- median(merged_df$surface)
mode <- table(merged_df$surface)
mode <- as.numeric(names(mode[which(mode == max(mode))]))
central_tendency_metrics <- paste(sprintf('Mean: %s', mean),
sprintf('Median: %s', median), sprintf('Mode: %s', mode), sep = '  |
')
```

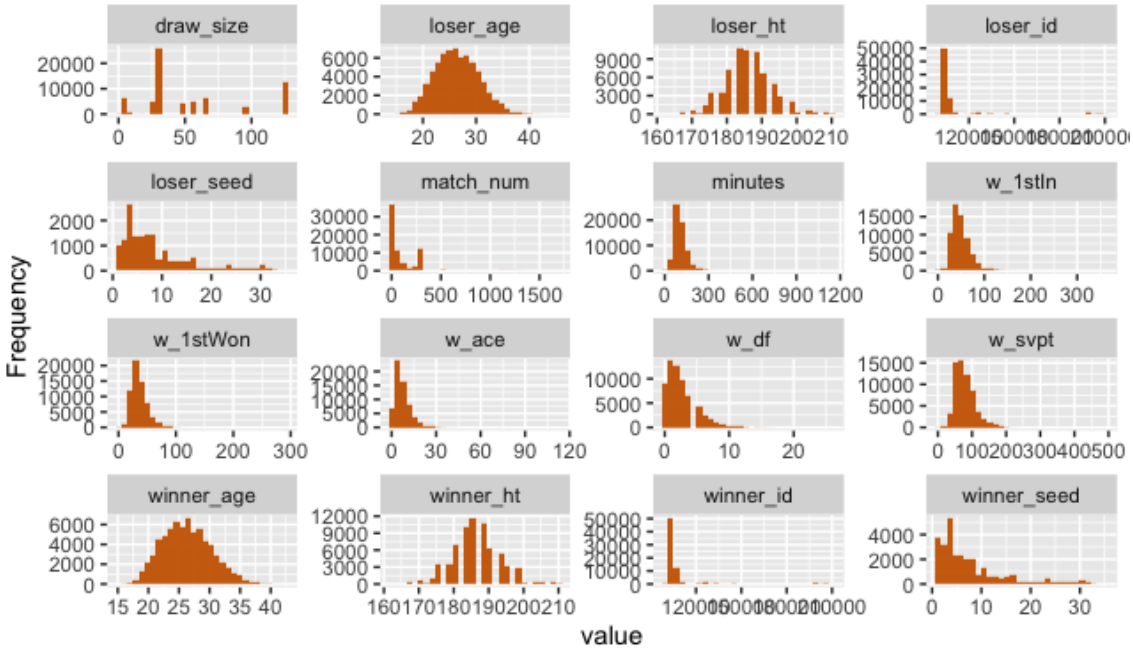
Additionally, repeat the same procedure of plotting the match data distribution, but this time for the surface attribute.

```
#Plot distribution of match data, by surface
ggplot(merged_df, aes(y = surface, fill = surface)) +
  geom_bar(color = 'black') +
  theme(axis.text.x = element_text(angle = 0), plot.title =
element_text(hjust = 0.5)) +
  ggtitle('Distribution of Matches by Surface') +
  xlab('Number of Matches') +
  ylab('Type of Surface') +
  labs(caption = central_tendency_metrics)
```

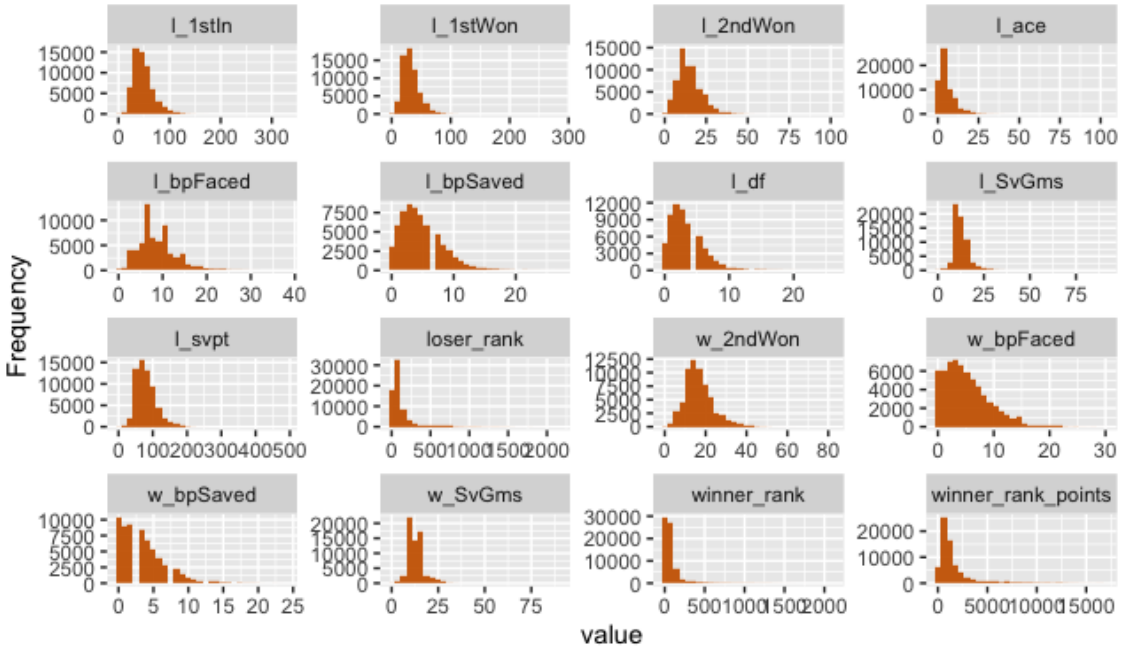


Lastly, one can create histograms for all the attributes found in the dataset automatically by using the `DataExplorer` package and the `DataExplorer::plot_histogram()` command. Notably, most attributes follow a normal distribution. There are several indications that sparse data is an issue for some attributes' observations given the gaps in the data visualized. Attributes like $draw_size, $loser_id, $winner_id, and $tourney_month are all examples of sparse data not showing any particular distribution. Many attributes show very little variance, which will be explored later as part of primary dimensionality reduction.

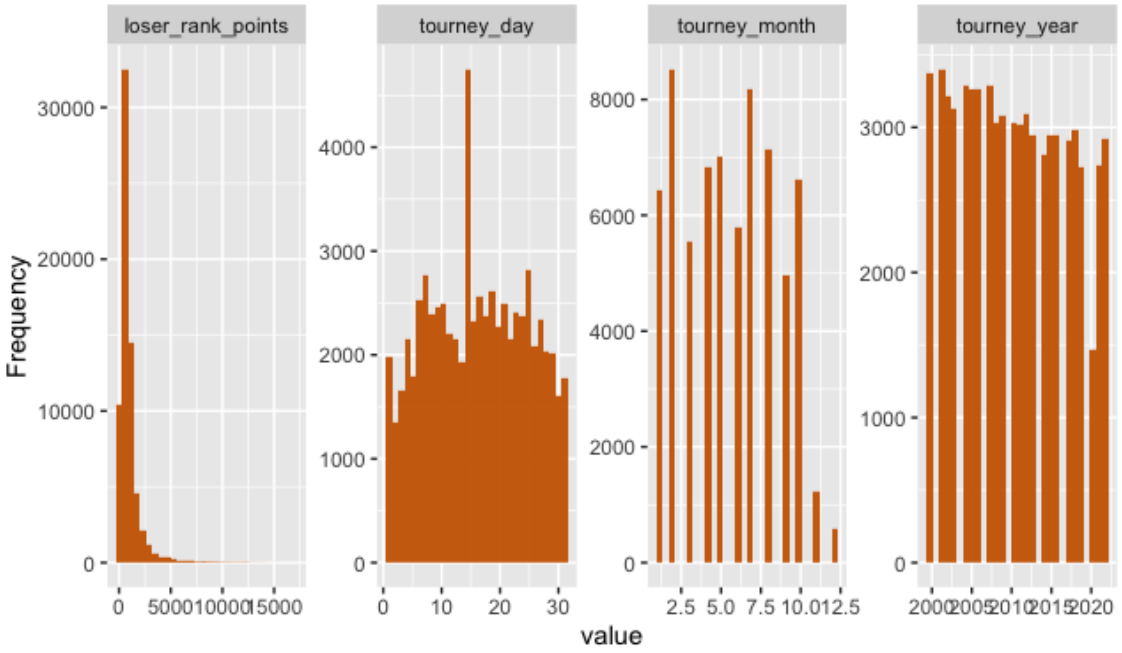
```
#Create a histogram of all attributes' distributions using the
DataExplorer package
plot_histogram(merged_df)
```



Page 1



Page 2



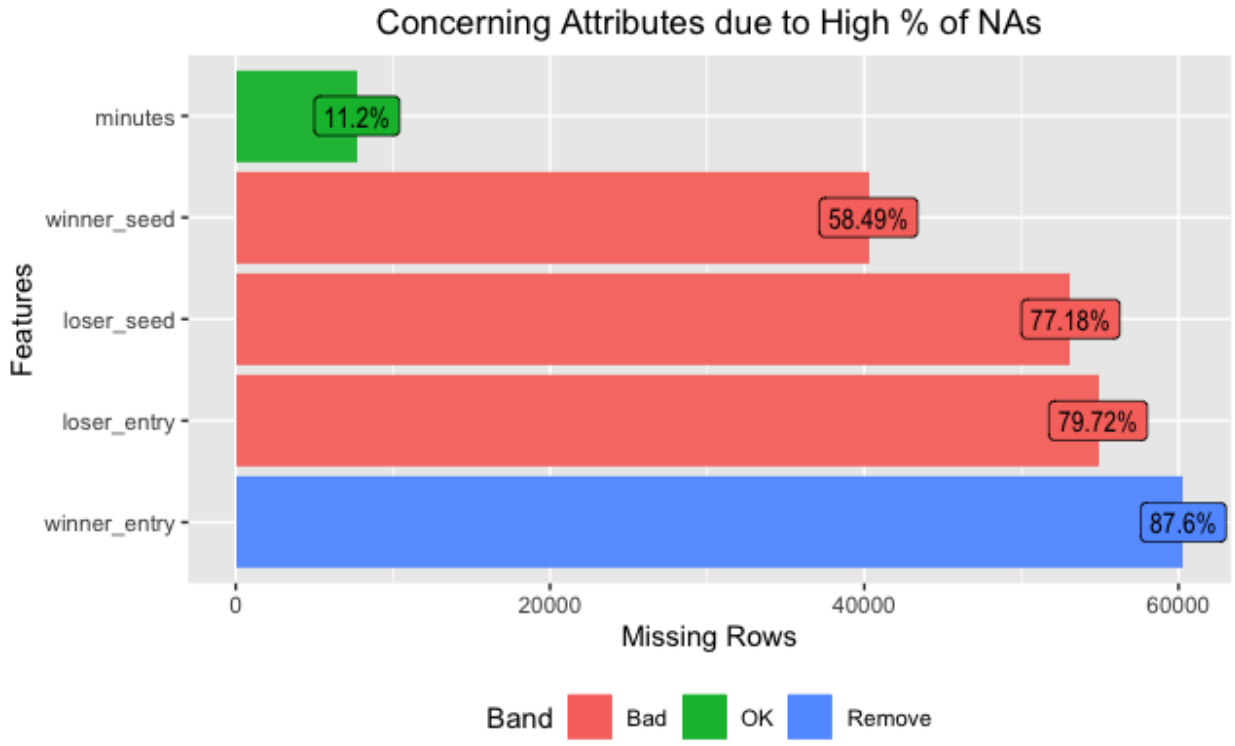
Section 4 - Data Cleaning and Dimensionality Reduction

4.1 - Remove Attributes with High NA Percentages

Variables included in the data set that have high NA percentages in their observations are likely to be noisy due to their incomplete nature. These are typically removed to improve computational times of the models being constructed, dependent on a NA completeness threshold. Given a 90% threshold, identify attributes that do not have at least 90% completeness of observation data. Afterwards, visualize missing row amounts, and remove the noisy attributes from the dataset.

```
# Identify high NA %age attributes
skim_df <- data.frame(skim(merged_df))
high_NA_perc <- skim_df %>% filter(complete_rate < 0.9)
high_NA_perc_list <- high_NA_perc$skim_variable
high_NA_df <- merged_df %>% select(high_NA_perc_list)
```

```
# Visualize high NA %age attributes
plot_missing(high_NA_df) +
  ggtitle('Concerning Attributes due to High % of NAs') +
  theme(plot.title = element_text(hjust = 0.5))
```




```
#Drop high NA %age attributes
merged_df <- subset(merged_df, select =
-c(winner_seed,loser_seed,loser_entry,winner_entry))
```

4.2 - Analyzing Highly Correlated Variables

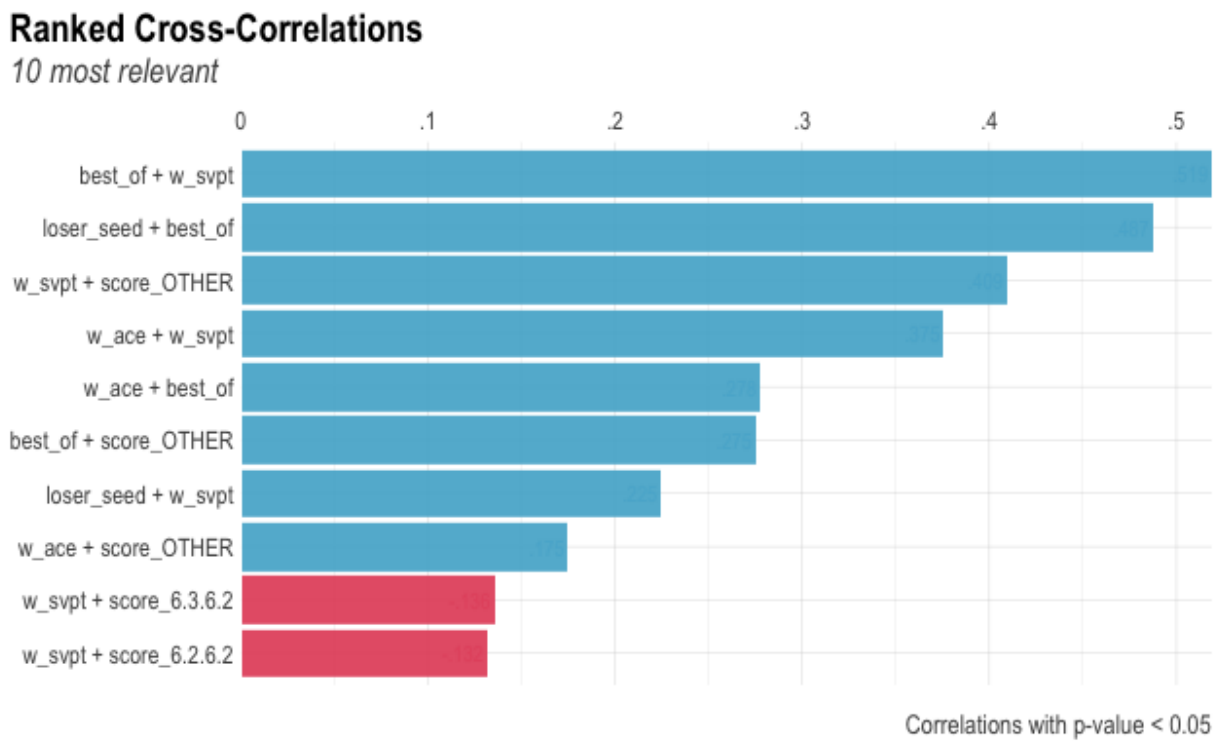
When cleaning a data set due to be used in training a machine learning model, highly correlated variables can be problematic. When two variables are highly correlated, they can make the model difficult to interpret, lead to unstable and inconsistent model coefficients, decrease the accuracy of the model, and increase the risk of overfitting. One can begin to reduce the impact of highly correlated variables by identifying them, and if the relationship exceeds a correlation threshold, the attributes ought to be removed.

Begin by storing the correlations of all attributes in a data frame where the cutoff of the relationship strength via Pearson's correlation coefficient is 0.9. Apply this to only numeric variable types that have full observation sets.

```
# Analyze correlations for all variables
correlation_df <- (cor(merged_df[, unlist(lapply(merged_df, 
is.numeric))], use = 'complete.obs'))

# Filter for only highly correlated variables
highly_correlated <- findCorrelation(correlation_df, cutoff = 0.9, 
verbose = TRUE)
```

From a visual perspective, plot the top 10 cross-correlations, visualizing both negative and positive correlations that are statistically significant with a maximum p-value of 0.05.

```
# Correlation plot for all top 10 highly correlated variables
highly_correlated_df <- subset(merged_df, select = highly_correlated)
corr_cross(highly_correlated_df, max_pvalue = 0.05, top = 10)
```



From the graphical output, one can see that several match statistics highly correlate with one another. These will be noted upon, but not removed until more advanced dimensionality reduction techniques are employed, like variable importance via Gini coefficients.

4.3 - Addressing NAs and Zeros

The dataset contains many players who have no ATP rank due to their missing playing history. This data once again can prove to be a noisy variable given its inconsistent observations. Unfortunately, imputation is not possible here because rank is a discrete value that is assigned at a particular point in time to any given player. To avoid biasing the model with noisy observations, remove all instances of players where ranking is unavailable after having played 100 matches.

```
#Remove winners with less than 100 matches and no rank
winners <- table(merged_df$winner_name)
winners_less_100 <- merged_df[merged_df$winner_name %in%
names(winners[winners < 100]), ]
winners_less_100_no_rank <- winners_less_100 %>%
subset(is.na(winners_less_100$winner_rank))
merged_df <- anti_join(merged_df, winners_less_100_no_rank, by =
"winner_id")

#Remove losers with less than 100 matches and no rank
losers <- table(merged_df$loser_name)
losers_less_100 <- merged_df[merged_df$loser_name %in%
names(losers[losers < 100]), ]
losers_less_100_no_rank <- losers_less_100 %>%
subset(is.na(losers_less_100$loser_rank))
merged_df <- anti_join(merged_df, losers_less_100_no_rank, by =
"loser_id")
```

Additionally, there are some observations where no minute data is recorded for the match, meaning the match was not played, or stopped during proceedings due to weather, injury, scheduling conflicts, travel restrictions, and other administrative issues.

These observations are of no value given their lack of completeness and are to be removed.

```
merged_df <- merged_df[complete.cases(merged_df[, "minutes"]),]
```

4.4 - COVID-19 ATP Interruptions

Due to the COVID-19 pandemic, a lot of player match-ups did not take place during the 2020 season. For all players eligible or ineligible to compete, their competitive ranking was frozen for the duration of the season. This freezing of the rank data adds noise to the model's understanding of fluctuating rankings, by which all 2020 match observations in the dataset will be removed.

```
str(merged_df$tourney_date)
matches_in_2020 <- filter(merged_df, merged_df$tourney_date >= 
'2020-01-01' & merged_df$tourney_date <= '2020-12-31')
merged_df <- anti_join(merged_df, matches_in_2020, by = 
“tourney_date”)
```

4.5 - Obscuring Target Variables

In order to not reveal the data of match winners and their respective match attributes, '`winner`' and '`loser`' attribute values are reassigned into new variables independent of win and loss statuses.

`player_1` attributes are assigned to the player's name occurring first when alphabetically compared, and `player_2` attributes are for the player's name occurring second when alphabetically compared. These are assigned alphabetically as to mask observation-specific indexing tied to winner and losers. Winners are assigned to a new attribute, `result`.

```
#Create target variable
merged_df$result <- merged_df$winner_name

#Create function for sorting players in player_1
first_player_sort = function(x,y) {paste(sort(c(x, y))[1])}
first_player_sort = Vectorize(first_player_sort)

#Assign player_1 values to the player occurring first in the
alphabetical comparison
merged_df <- merged_df %>% mutate(player_1 =
first_player_sort(winner_name, loser_name))

#Create function for sorting players in player_1
second_player_sort = function(x,y) {paste(sort(c(x, y))[2])}
second_player_sort = Vectorize(second_player_sort)

#Assign player_2 values to the player occurring second in the
alphabetical comparison
merged_df <- merged_df %>% mutate(player_2 =
second_player_sort(winner_name, loser_name))

#Rename winner columns to player_1 columns, rename loser columns to
player_2 columns
colnames(merged_df) <- gsub("winner", "player_1", colnames(merged_df))
colnames(merged_df) <- gsub("loser", "player_2", colnames(merged_df))
```

player_1_id	player_1_seed	player_1_entry	player_1_name
103163	1	*NA*	Tommy Haas
102607	*NA*	Q	Juan Balcells
103252	*NA*	*NA*	Alberto Martin
103507	7	*NA*	Juan Carlos Ferrero

An example of masked winner and loser attributes.

4.6 - Administrative Attribute Removal

The dataset also contains some observations which are populated with alphanumeric symbols in the score attribute. Alphanumeric characters in the score attribute are indicative of a player retirement, default, or walkover. These rows are to be removed from the `merged_df` data frame due to their high row-wise missingness.

```
alpha_rows <- grep("[A-Za-z]", merged_df$score)
merged_df <- subset(merged_df, !(row.names(merged_df) %in% 
alpha_rows))
```

The merged data frame also contains many hyper-granular statistics about the match pertaining to each player. For macro-level analysis of who is likely to win, these statistics won't serve as useful primary dimensions and are to be removed to avoid hyperdimensionality of the model. These features are not expected to have a significant contribution to the target variable, and are expected to contribute to overfitting, added computational cost, and reduced interpretability.

```
#Remove granular attributes of winners
granular_w_stat_columns <- grep("w_", names(merged_df))
merged_df <- subset(merged_df, select = -granular_w_stat_columns)

#Remove granular attributes of losers
granular_l_stat_columns <- grep("l_", names(merged_df))
merged_df <- subset(merged_df, select = -granular_l_stat_columns)
```

There are also several administrative attributes in the dataset that have no added predictive value and are only existing for recording purposes to identify match data in ATP databases. These are to be removed, and include player's nationality, and individual player ID.

```
merged_df <- select(merged_df, -c("player_1_ioc", "player_2_ioc", 
"player_1_id", "player_2_id"))
```

4.7 - Non-Imputable Attributes

A small number of rows of players' heights and hand preference is missing. They are for players ranked very low and have low number of matches played. Player's height could be imputed using the mean height value of the group, but height is important in tennis, and to not bias the data, the rows are removed. Similarly, hand preference can be imputed using the mode, but it would not be an accurate method of

imputation. Given that these are the only NA values remaining, one can use a sweeping `na.omit()` function to eliminate the observations.

```
merged_df <- na.omit(merged_df)
```

Lastly, we can preview the final cleansed data frame, consisting of 54,889 observations and 28 attributes.

```
> glimpse(merged_df)
Rows: 54,889
Columns: 28
$ tourney_id           <chr> "2000-301", "2000-301", "2000-301", "2000-301", "2000-301", "…
$ tourney_name         <chr> "Auckland", "Auckland", "Auckland", "Auckland", "Auckland", "…
$ surface              <chr> "Hard", "Hard", "Hard", "Hard", "Hard", "Hard", "Hard", "Hard…
$ tourney_level        <chr> "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "…
$ tourney_date         <date> 2000-01-10, 2000-01-10, 2000-01-10, 2000-01-10, 2000-01-10, …
$ match_num            <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18…
$ player_1_name        <chr> "Tommy Haas", "Juan Balcells", "Alberto Martin", "Juan Carlos…
$ player_1_hand        <chr> "R", "R", "R", "R", "R", "R", "R", "R", "R", "R", "R", "R", "…
$ player_1_ht          <dbl> 188, 190, 175, 183, 180, 175, 185, 180, 193, 193, 175, 190, 2…
$ player_1_age         <dbl> 21.7, 24.5, 21.3, 19.9, 27.3, 27.8, 33.0, 24.7, 23.3, 25.0, 2…
$ player_2_name        <chr> "Jeff Tarango", "Franco Squillari", "Alberto Berasategui", "R…
$ player_2_hand        <chr> "L", "L", "R", "R", "R", "R", "R", "R", "L", "R", "R", "R", "…
$ player_2_ht          <dbl> 180, 183, 173, 185, 185, 175, 185, 188, 193, 188, 180, 183, 1…
$ player_2_age         <dbl> 31.1, 24.3, 26.5, 18.4, 23.7, 30.2, 22.2, 23.7, 28.3, 22.1, 2…
$ score                <chr> "7-5 4-6 7-5", "7-5 7-5", "6-3 6-1", "6-4 6-4", "0-6 7-6(7) 6…
$ best_of              <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3…
$ round                <chr> "R32", "R32", "R32", "R32", "R32", "R32", "R32", "R32", "R32"…
$ minutes              <dbl> 108, 85, 56, 68, 115, 140, 84, 77, 90, 57, 144, 96, 59, 70, 9…
$ player_1_rank        <dbl> 11, 211, 48, 45, 167, 50, 60, 43, 37, 115, 72, 38, 46, 66, 57…
$ player_1_rank_points <dbl> 1612, 157, 726, 768, 219, 722, 626, 785, 843, 344, 553, 831, …
$ player_2_rank        <dbl> 63, 49, 59, 61, 34, 70, 246, 334, 68, 95, 52, 76, 53, 56, 80,…
$ player_2_rank_points <dbl> 595, 723, 649, 616, 873, 563, 135, 88, 571, 416, 715, 514, 70…
$ tourney_year         <dbl> 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2000, 2…
$ tourney_month        <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1…
$ tourney_day          <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 1…
$ result               <chr> "Tommy Haas", "Juan Balcells", "Alberto Martin", "Juan Carlos…
$ player_1             <chr> "Jeff Tarango", "Franco Squillari", "Alberto Berasategui", "J…
$ player_2             <chr> "Tommy Haas", "Juan Balcells", "Alberto Martin", "Roger Feder…
```

A summary of the cleaned data frame of merged .CSVs.

Section 5 - Feature Engineering

5.1 - Head-to-Head

To better understand the historical results of a matchup, a head-to-head record is required for each pairing of players in the dataset.

To create a head-to-head feature, first tabulate the total encounters between a pair of players across all tournament instances.

```
#Register total encounters
total_encounters <- data.frame(table(merged_df$player_1, 
merged_df$player_2))
merged_df <- merge(merged_df, total_encounters, by.x = c("player_1", 
"player_2"), by.y = c("Var1", "Var2"), all.x = TRUE)
merged_df <- rename(merged_df, total_encounters = Freq)
```

Next, match observations of `player_1` to observations where `player_1` is the winner, whilst simultaneously matching for `player_2` results ending in the same `player_1` victory. Logically, one is looking for instances of observations where `player_1` played `player_2` and won on `player_1`'s record, and `player_2`'s record. This makes sense because in a match there is only one winner, but the matchup can be made both ways, such as `player_2` playing `player_1`. The frequency of these simultaneous occurrences is then tabulated in a frequency table to show the times these player pairings were recorded.

```
#Register player_1 victories
player_1_h2h_w <- data.frame(
                    table(
                      merged_df$player_1[merged_df$result == 
merged_df$player_1], merged_df$player_2[merged_df$result == 
merged_df$player_1]
                        )
                    )
```

result	total_encounters	player_1_h2h	player_2_h2h
Adam Chadaj	1	1	0
Jose Acasuso	1	0	1
Andrew Whittington	1	0	1
Diego Schwartzman	1	0	1
Adam Pavlasek	1	1	0
Adam Pavlasek	1	1	0

A screen grab of the merged data frame and the new head-to-head features.

Adjustments are subsequently made to renaming the head-to-head data frame's columns, and the `player_1` head-to-head data is merged to the greater `merged_df` data frame by a `player_1` and `player_2` pairing join.

```
player_1_h2h_w <- rename(player_1_h2h_w, player_1_h2h = Freq)
merged_df <- merge(merged_df, player_1_h2h_w, by.x = c("player_1",
"player_2"), by.y = c("Var1", "Var2"), all.x = TRUE)
```

This accounts for observations on `player_1`'s side of the head-to-head, and the populated values for `player_2`'s side are currently NA. Those are populated with zeros as `player_2`'s head-to-head record is calculated shortly.

```
#Replace NAs in the player_1_h2h column with the value 0

library(imputeTS)
merged_df$player_1_h2h <- na_replace(merged_df$player_1_h2h, 0)
```

The exact same procedure is then replicated for `player_2`.

```
#Replace NAs in the player_1_h2h column with the value 0
library(imputeTS)
merged_df$player_1_h2h <- na_replace(merged_df$player_1_h2h, 0)
```

```
#Register player_2 victories
player_2_h2h_w <- data.frame(
                    table(
                      merged_df$player_1[merged_df$result ==
merged_df$player_2], merged_df$player_2[merged_df$result ==
merged_df$player_2]
                      )
                    )
player_2_h2h_w <- rename(player_2_h2h_w, player_2_h2h = Freq)
merged_df <- merge(merged_df, player_2_h2h_w, by.x = c("player_1",
"player_2"), by.y = c("Var1", "Var2"), all.x = TRUE)

#Replace NAs in the player_2_h2h column with the value 0
merged_df$player_2_h2h <- na_replace(merged_df$player_2_h2h, 0)
```

5.2 - % Win on Surface

Given that variability of match results may depend on the surface the players are playing on, it is important to understand the cumulative win percentage of a player on a particular surface type. A similar approach to the head-to-head procedure outlined in 5.1 is prescribed for the new feature, `$player_x_surface_perc`.

For each player, the amount of matches played on a surface are tabulated.

```
#Calculate number of matches per player per surface
player_1_surface <- data.frame(
                    table(
                      merged_df$player_1, merged_df$surface
                      )
                    )
```

Afterwards, the same match up of player pairings and match winners is made. The only difference in the greater `merged_df` data frame is the engineered column is now dynamically calculated by dividing a player's total wins on a surface to the total wins they've registered historically. Finally, some minor alterations are subsequently made to the `merged_df` data frame for legibility.

```
#Calculate number of wins per player per surface
player_1_surface_wins <- data.frame(
                    table(
                      merged_df$player_1[merged_df$result ==
merged_df$player_1], merged_df$surface[merged_df$result ==
merged_df$player_1]
                      )
                    )

#Calculate win % per player per surface
player_1_surface_w_perc <- merge(player_1_surface,
player_1_surface_wins, by.x = c("Var1", "Var2"), by.y = c("Var1",
"Var2"), all.x = TRUE)
player_1_surface_w_perc$Freq.y <-
na_replace(player_1_surface_w_perc$Freq.y, 0)
player_1_surface_w_perc$w_perc <-
round((player_1_surface_w_perc$Freq.y/player_1_surface_w_perc$Freq.x),
2)
player_1_surface_w_perc$w_perc <-
na_replace(player_1_surface_w_perc$w_perc, 0)

#Drop columns used for calculation
player_1_surface_w_perc <- player_1_surface_w_perc %>% select(-Freq.x,
-Freq.y)

#Rename player 1 win % data frame columns
player_1_surface_w_perc <- player_1_surface_w_perc %>%
rename(player_1_surface_perc = w_perc)
```

The same process is then repeated for `player_2`.:

```
#Calculate number of matches per player per surface
player_2_surface <- data.frame(
                    table(
                      merged_df$player_2, merged_df$surface
                      )
                    )

#Calculate number of wins per player per surface
player_2_surface_wins <- data.frame(
                    table(
                      merged_df$player_2[merged_df$result ==
merged_df$player_2], merged_df$surface[merged_df$result ==
merged_df$player_2]
                      )
                    )
```

```
#Calculate win % per player per surface
player_2_surface_w_perc <- merge(player_2_surface,
player_2_surface_wins, by.x = c("Var1", "Var2"), by.y = c("Var1",
"Var2"), all.x = TRUE)
player_2_surface_w_perc$Freq.y <-
na_replace(player_2_surface_w_perc$Freq.y, 0)
player_2_surface_w_perc$w_perc <-
round((player_2_surface_w_perc$Freq.y/player_2_surface_w_perc$Freq.x),
2)
player_2_surface_w_perc$w_perc <-
na_replace(player_2_surface_w_perc$w_perc, 0)

#Drop columns used for calculation
player_2_surface_w_perc <- player_2_surface_w_perc %>% select(-Freq.x,
-Freq.y)

#Rename player 2 win % data frame columns
player_2_surface_w_perc <- player_2_surface_w_perc %>%
rename(player_2_surface_perc = w_perc)

#Consolidate player_1 and player_2 data into merged_df
merged_df <- merge(merged_df, player_1_surface_w_perc, by.x =
c("player_1", "surface"), by.y = c("Var1", "Var2"), all.x = TRUE)
merged_df <- merge(merged_df, player_2_surface_w_perc, by.x =
c("player_2", "surface"), by.y = c("Var1", "Var2"), all.x = TRUE)
```

5.3 - % Win at General Tournament Stage

Using the exact same method seen in 5.2, one is able to calculate the percentage of a player's wins at a general tournament stage. This means that if they were to reach a particular round of any given tournament, how likely would they be to win that match?

```
#For player_1:

#Calculate number of matches per player per tournament stage
player_1_round <- data.frame(
                    table(
                      merged_df$player_1, merged_df$round
                      )
                    )
```

```r
#Calculate number of wins per player per tournament stage
player_1_round_wins <- data.frame(
                    table(
                      merged_df$player_1[merged_df$result ==
merged_df$player_1], merged_df$round[merged_df$result ==
merged_df$player_1]
                      )
                    )
#Calculate win % per player per tournament stage
player_1_round_w_perc <- merge(player_1_round, player_1_round_wins,
by.x = c("Var1", "Var2"), by.y = c("Var1", "Var2"), all.x = TRUE)
player_1_round_w_perc$Freq.y <-
na_replace(player_1_round_w_perc$Freq.y, 0)
player_1_round_w_perc$w_perc <- round((player_1_round_w_perc$Freq.y/
player_1_round_w_perc$Freq.x), 2)
player_1_round_w_perc$w_perc <-
na_replace(player_1_round_w_perc$w_perc, 0)

#Drop columns used for calculation
player_1_round_w_perc <- player_1_round_w_perc %>% select(-Freq.x,
-Freq.y)

#Rename player 1 win % data frame columns
player_1_round_w_perc <- player_1_round_w_perc %>%
rename(player_1_round_perc = w_perc)

#For player_2:

#Calculate number of matches per player per tournament stage
player_2_round <- data.frame(
                    table(
                      merged_df$player_2, merged_df$round
                      )
                    )

#Calculate number of wins per player per tournament stage
player_2_round_wins <- data.frame(
                    table(
                      merged_df$player_2[merged_df$result ==
merged_df$player_2], merged_df$round[merged_df$result ==
merged_df$player_2]
                      )
                    )
```

```
#Calculate win % per player per tournament stage
player_2_round_w_perc <- merge(player_2_round, player_2_round_wins,
by.x = c("Var1", "Var2"), by.y = c("Var1", "Var2"), all.x = TRUE)
player_2_round_w_perc$Freq.y <-
na_replace(player_2_round_w_perc$Freq.y, 0)
player_2_round_w_perc$w_perc <- round((player_2_round_w_perc$Freq.y/
player_2_round_w_perc$Freq.x), 2)
player_2_round_w_perc$w_perc <-
na_replace(player_2_round_w_perc$w_perc, 0)

#Drop columns used for calculation
player_2_round_w_perc <- player_2_round_w_perc %>% select(-Freq.x,
-Freq.y)

#Rename player 2 win % data frame columns
player_2_round_w_perc <- player_2_round_w_perc %>%
rename(player_2_round_perc = w_perc)

#Consolidate player_1 and player_2 data into merged_df
merged_df <- merge(merged_df, player_1_round_w_perc, by.x =
c("player_1", "round"), by.y = c("Var1", "Var2"), all.x = TRUE)
merged_df <- merge(merged_df, player_2_round_w_perc, by.x =
c("player_2", "round"), by.y = c("Var1", "Var2"), all.x = TRUE)
```

5.4 - % Win at Tournament Level

Using the exact same method seen in 5.3, one is able to calculate the percentage of a player's wins at a tournament. This means that if they were to reach a particular tournament level (Master's, Grand Slam, Challenger, etc.), how likely would they be to win that match?

```
#For player_1:

#Calculate number of matches per player per tournament level
player_1_tourney_level <- data.frame(
                    table(
                      merged_df$player_1, merged_df$tourney_level
                      )
                    )
```

```
#Calculate number of wins per player per tournament level
player_1_tourney_level_wins <- data.frame(
                    table(
                      merged_df$player_1[merged_df$result ==
merged_df$player_1], merged_df$tourney_level[merged_df$result ==
merged_df$player_1]
                      )
                    )

#Calculate win % per player per tournament level
player_1_tourney_level_w_perc <- merge(player_1_tourney_level,
player_1_tourney_level_wins, by.x = c("Var1", "Var2"), by.y =
c("Var1", "Var2"), all.x = TRUE)
player_1_tourney_level_w_perc$Freq.y <-
na_replace(player_1_tourney_level_w_perc$Freq.y, 0)
player_1_tourney_level_w_perc$w_perc <-
round((player_1_tourney_level_w_perc$Freq.y/
player_1_tourney_level_w_perc$Freq.x), 2)
player_1_tourney_level_w_perc$w_perc <-
na_replace(player_1_tourney_level_w_perc$w_perc, 0)

#Drop columns used for calculation
player_1_tourney_level_w_perc <- player_1_tourney_level_w_perc %>%
select(-Freq.x, -Freq.y)

#Rename player 1 win % data frame columns
player_1_tourney_level_w_perc <- player_1_tourney_level_w_perc %>%
rename(player_1_tourney_level_perc = w_perc)

#For player_2:

#Calculate number of matches per player per tournament level
player_2_tourney_level <- data.frame(
                    table(
                      merged_df$player_2, merged_df$tourney_level
                      )
                    )

#Calculate number of wins per player per tournament level
player_2_tourney_level_wins <- data.frame(
                    table(
                      merged_df$player_2[merged_df$result ==
merged_df$player_2], merged_df$tourney_level[merged_df$result ==
merged_df$player_2]
                      )
                    )
```

```
#Calculate win % per player per tournament level
player_2_tourney_level_w_perc <- merge(player_2_tourney_level,
player_2_tourney_level_wins, by.x = c("Var1", "Var2"), by.y =
c("Var1", "Var2"), all.x = TRUE)
player_2_tourney_level_w_perc$Freq.y <-
na_replace(player_2_tourney_level_w_perc$Freq.y, 0)
player_2_tourney_level_w_perc$w_perc <-
round((player_2_tourney_level_w_perc$Freq.y/
player_2_tourney_level_w_perc$Freq.x), 2)
player_2_tourney_level_w_perc$w_perc <-
na_replace(player_2_tourney_level_w_perc$w_perc, 0)

#Drop columns used for calculation
player_2_tourney_level_w_perc <- player_2_tourney_level_w_perc %>%
select(-Freq.x, -Freq.y)

#Rename player 2 win % data frame columns
player_2_tourney_level_w_perc <- player_2_tourney_level_w_perc %>%
rename(player_2_tourney_level_perc = w_perc)

#Consolidate player_1 and player_2 data into merged_df
merged_df <- merge(merged_df, player_1_tourney_level_w_perc, by.x =
c("player_1", "tourney_level"), by.y = c("Var1", "Var2"), all.x =
TRUE)
merged_df <- merge(merged_df, player_2_tourney_level_w_perc, by.x =
c("player_2", "tourney_level"), by.y = c("Var1", "Var2"), all.x =
TRUE)
```

5.5 - % Win at Specific Tournament Stage

Given the aforementioned attributes, the next engineered attribute will be a combination of tournament specific round advancement win percentage. That is to say, when a player reaches x round in y tournament, what is his chance to win the matchup? This value is a simple multiplication and inserted as a dynamically populating column in `merged_df`.

```
#Calculate win %age of player 1 at x tournament in the nth round
merged_df$player_1_tourney_round_perc <-
round((merged_df$player_1_tourney_level_perc *
merged_df$player_1_round_perc), 2)

#Calculate win %age of player 2 at x tournament in the nth round
merged_df$player_2_tourney_round_perc <-
round((merged_df$player_2_tourney_level_perc *
merged_df$player_2_round_perc), 2)
```

5.6 - Removal of Redundant Features Used in Feature Engineering Calculations

With all desired features engineered, the attributes that were supplying the data to the features can be removed as they are now redundant. Remove them from the `merged_df` data frame.

```
redundant_features <- c('tourney_month', 'tourney_year',
'tourney_day', 'tourney_id', 'match_num')
merged_df <- merged_df %>% select(-redundant_features)
```

```
features_to_keep <- c('player_1_age',
                      'player_2_age',
                      'player_1_rank',
                      'player_2_rank',
                      'player_1_h2h',
                      'player_2_h2h',
                      'player_1_round_perc',
                      'player_1_surface_perc',
                      'player_1_tourney_level_perc',
                      'player_1_tourney_round_perc',
                      'player_2_round_perc',
                      'player_2_surface_perc',
                      'player_2_tourney_level_perc',
                      'player_2_tourney_round_perc',
                      'surface',
                      'tourney_level',
                      'result')
merged_df <- merged_df %>% select(features_to_keep)
```

5.7 - Removing $result levels with Low Counts

In an effort to reduce an already high number of classification levels as part of the $results attribute, improve computational time and model accuracy by removing levels that have less than 5 counts.

```
#To simplify future classification, remove noisy results from $results
where few counts are detected

less_than_5_wins <- data.frame(table(merged_df$result))
less_than_5_wins <- less_than_5_wins %>% filter(Freq < 5)

merged_df <- merged_df[!merged_df$result %in% less_than_5_wins$Var1,]
```

5.8 - Final Organization of Merged Data Frame

Lastly, all the columns in the dataset are ordered alphabetically for easy visual parsing.

```
#Order the columns alphabetically
merged_df <- merged_df[, order(names(merged_df))]
```

Section 6 - Data Splitting

The `merged_df` data frame served as the basis of data splitting for training and testing data sets. Given the amount of computationally-intensive runtimes for the planned models, k-fold cross validation of 10 folds with 3 repeats was initially considered. However, R consistently crashed during attempted chunks of k-fold cross validation, and as such, the technique is missing from the study.

Training and testing data sets were split via a 70/30 proportionality, with stratified sampling to ensure an equal distribution of target variable classes in both sets.

```
library(caret)
#Set the seed for reproducibility
set.seed(123)

#Set target variable as factor
merged_df$result <- as.factor(merged_df$result)

#Split the data into a training set (70%) and a testing set (30%)
train_idx <- createDataPartition(merged_df$result, p = 0.7, list =
FALSE)
train <- merged_df[train_idx, ]
test <- merged_df[-train_idx, ]

#Split the data into x_train, x_test, y_train, y_test sets
x_train <- train %>% select(-result)
y_train <- train %>% select(result)
y_train <- as.factor(y_train$result)

x_test <- test %>% select(-result)
y_test <- test %>% select(result)
y_test <- as.factor(y_test$result)
```

Section 7 - Model Building and Evaluation

The models chosen for the target variable `$result` classification task included random forests and naive Bayes. From the preliminary project updates, gradient boosted random forests and logistic regressions were also to be compared, however, cumbersome use of `xgb.matrix()` data types and highly computationally expensive runtimes removed these two algorithms from consideration.

7.1 - Random Forest Classifier, Default Settings

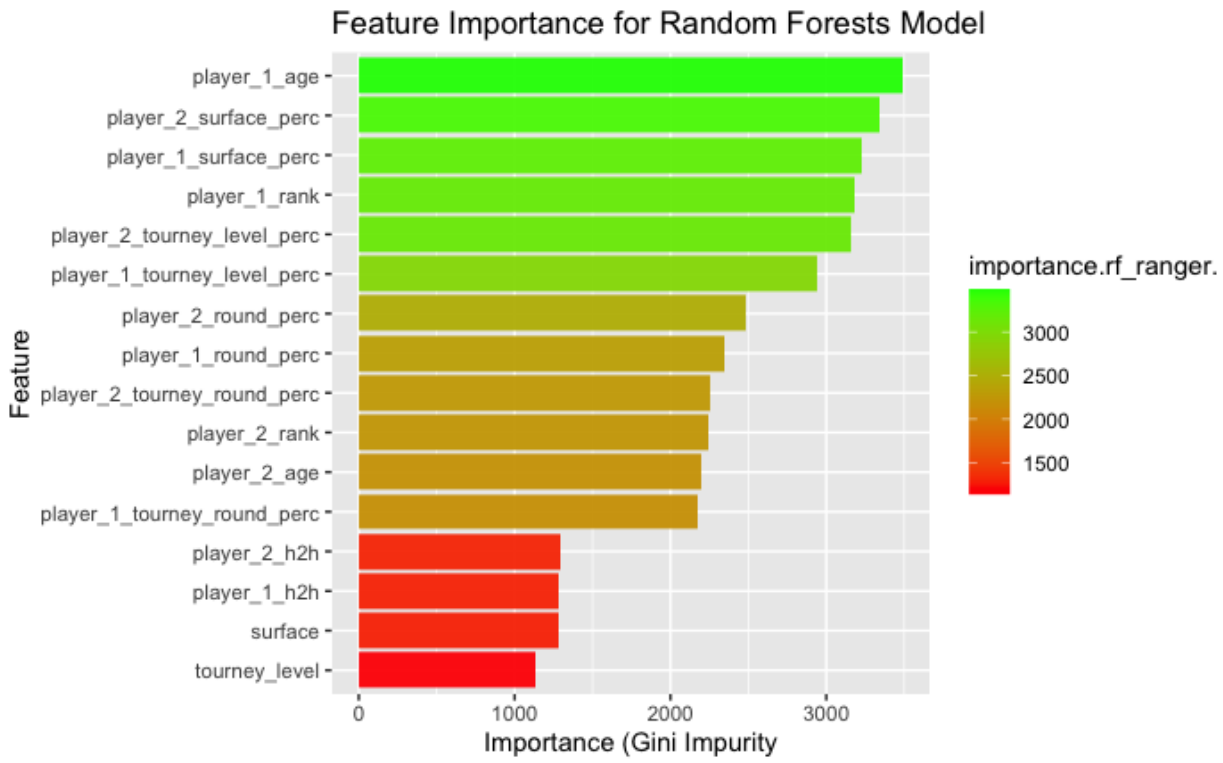
A random forest classifier was created using the `ranger` package due to it being computationally more efficient than the `randomforests` package. Variable importance is measured during model creation using Gini impurity as the chosen metric of evaluation.

```
library(caret)
library(ranger)

#Set seed for replicability
set.seed(123)

#Create model, measuring variable importance while building forest
rf_ranger <- ranger(result ~., data = train, importance = “impurity”)

#Print the model results
print(rf_ranger)
```

```
Call:
 ranger(result ~ ., data = train, importance = "impurity")

Type:                             Classification
Number of trees:                  500
Sample size:                      38264
Number of independent variables:  16
Mtry:                             4
Target node size:                 1
Variable importance mode:         impurity
Splitrule:                        gini
OOB prediction error:             30.21 %
```

The model's most important variables are displayed using a customary `ggplot` implementation.

```
#Print the model's most important variables
rf_ranger_vi <- data.frame(importance(rf_ranger))
rf_ranger_vi <- rownames_to_column(rf_ranger_vi)

#Plot the model's most important variables
rf_ranger_vi %>%
  ggplot(aes(reorder(rowname, importance.rf_ranger.),
importance.rf_ranger.)) +
  geom_col(aes(fill = importance.rf_ranger.)) +
  scale_fill_gradient(low = "red", high = "green") +
  coord_flip() +
  labs(x = "Feature", y = "Importance (Gini Impurity") +
  ggtitle("Feature Importance for Random Forests Model")
```



Using the training data and the `mlMetrics` package, call `mlMetrics::Accuracy` to determine the model accuracy of the default random forests model.

```
#Determine default model accuracy on train data
library(MLmetrics)

y_pred <- rf_ranger$predictions
round((Accuracy(y_pred, y_train)), 4)
```

7.2 - Random Forest Classifier, Grid-Search Settings

To optimize the hyperparameters of the random forests model, conduct a grid-search of different hyperparameter values.

The hyperparameters to be iterated upon are:

- **mtry:** the number of variables randomly sampled as candidates at each split
- **min.node.size:** the minimum number of observations at a terminal node
- **num.trees:** number of trees in the forest

Conduct a grid-search with the values below, apply them to a random forest model using the training data, and append the root mean squared error (RMSE) values to a data frame. Note this is very computationally expensive and may take a long time.

```
n_features = rf_ranger$num.independent.variables
hyper_grid <- expand.grid(
  mtry = floor(n_features * c(.15, .25, .35)),
  min.node.size = c(1, 3, 5),
  num.trees = n_features * c(5, 10, 15)
)

for(i in seq_len(nrow(hyper_grid))) {
rf_ranger_opt <- ranger(
    formula         = result ~ .,
    data            = train,
    num.trees       = n_features * 10,
    mtry            = hyper_grid$mtry[i],
    min.node.size   = hyper_grid$min.node.size[i],
    verbose         = FALSE,
    seed            = 123,
    respect.unordered.factors = 'order',
  )

#store results
hyper_grid$rmse[i] <- sqrt(rf_ranger_opt$prediction.error)
}
```

Once complete, review the results, and compare the percentage change of the hyperparameter tweaked models to the default model created in section 7.1.

```
#Compare grid-search optimized settings to model defaults

rf_ranger_rmse <- sqrt(rf_ranger_opt$prediction.error)
hyper_grid$default_rmse <- rf_ranger_rmse
hyper_grid <- hyper_grid %>% arrange(rmse) %>% mutate(percentage_gain
= (rmse - rf_ranger_rmse) / rf_ranger$prediction.error * 100)

print(hyper_grid)
```

hyper_grid

Filter

	mtry	min.node.size	num.tress	rmse	percentage_gain	default_rmse
1	5	1	40	0.5582334	-2.118771	0.5646339
2	5	1	80	0.5582334	-2.118771	0.5646339
3	5	1	120	0.5582334	-2.118771	0.5646339
4	5	1	160	0.5582334	-2.118771	0.5646339
5	5	3	40	0.5595660	-1.677616	0.5646339
6	5	3	80	0.5595660	-1.677616	0.5646339

A lower RMSE score is better, meaning the model is able to fit the data better. With mtry = 5, min.node.size = 1, num.trees = 160, the RMSE from the default model decreases marginally by 2.11%.

```
#Generate the model with the grid-search optimized parameters
rf_ranger_opt_final <- ranger(result ~.,
                              data = train,
                              importance = "impurity",
                              mtry = 5,
                              min.node.size = 1,
                              num.trees = 160)

#Print the optimized model's results
print(rf_ranger_opt_final)
print(paste("RMSE:", sqrt(rf_ranger_opt_final$prediction.error)))

#Determine optimized model accuracy on train data
y_pred_opt <- rf_ranger_opt_final$predictions
round((Accuracy(y_pred_opt, y_train)), 4)
```

```
Call:
 ranger(result ~ ., data = train, importance = "impurity", mtry = 5,      min.node.size = 1, num.trees = 160)

Type:                             Classification
Number of trees:                  160
Sample size:                      38264
Number of independent variables:  16
Mtry:                             5
Target node size:                 1
Variable importance mode:         impurity
Splitrule:                        gini
OOB prediction error:             31.15 %
[1] "RMSE: 0.558139737474579"
[1] 0.6885
```

7.3 - Random Forest Classifier, Manually Optimized Settings

From the previously created grid-search optimized model, the largest decrease in RMSE was observed when mtry (the number of variables randomly sampled at each split) increased. Create a random forest model manually using the hyperparameter values determined by the grid-search, but increase the mtry value and report on the model's performance.

```
#Generate the model with the manually optimized parameters
rf_ranger_opt_manual <- ranger(result ~.,
                               data = train,
                               importance = "impurity",
                               mtry = 10,
                               min.node.size = 1,
                               num.trees = 160)
```

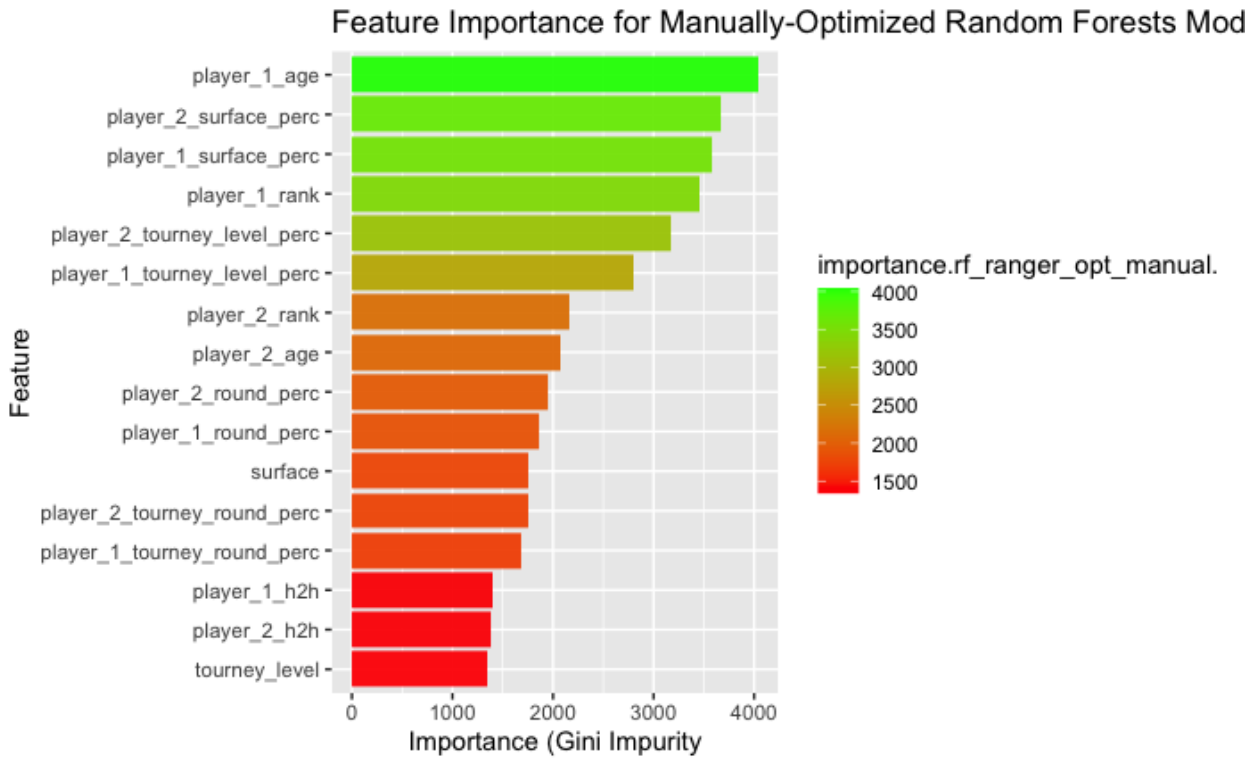
```
#Print the optimized model's results
print(rf_ranger_opt_manual)
print(paste("RMSE:", sqrt(rf_ranger_opt_manual$prediction.error)))
#Determine optimized model accuracy on train data
y_pred_manual <- rf_ranger_opt_manual$predictions
round((Accuracy(y_pred_manual, y_train)), 4)
```

```
Call:
 ranger(result ~ ., data = train, importance = "impurity", mtry = 10,      min.node.size = 1, num.trees = 160)

Type:                             Classification
Number of trees:                  160
Sample size:                      38264
Number of independent variables:  16
Mtry:                             10
Target node size:                 1
Variable importance mode:         impurity
Splitrule:                        gini
OOB prediction error:             30.33 %
[1] "RMSE: 0.550739339232619"
[1] 0.6967
```

7.4 - Random Forest Classifier, Truncated Features Settings

From the previously created manually-tuned model in section 7.3, there may be the possibility that the model may perform better if its low-importance variables are removed. To test this, analyze the importance of variables part of the manually-tuned model, and drop the least important variable from the training and test data sets.

```
#Print the model's most important variables
rf_ranger_vi_opt_manual <-
data.frame(importance(rf_ranger_opt_manual))
rf_ranger_vi_opt_manual <- rownames_to_column(rf_ranger_vi_opt_manual)

#Plot the model's most important variables
rf_ranger_vi_opt_manual %>%
  ggplot(aes(reorder(rowname, importance.rf_ranger_opt_manual.),
importance.rf_ranger_opt_manual.)) +
  geom_col(aes(fill = importance.rf_ranger_opt_manual.)) +
  scale_fill_gradient(low = "red", high = "green") +
  coord_flip() +
  labs(x = "Feature", y = "Importance (Gini Impurity") +
  ggtitle("Feature Importance for Manually-Optimized Random Forests
Model")
```



```
#Truncate the least important feature from the training and test data
sets
trunc_train <- subset(train, select = -tourney_level)
trunc_test <- subset(test, select = -tourney_level)

#Generate the model with the manually optimized parameters for the
truncated data sets
rf_ranger_opt_manual_red <- ranger(result ~.,
                                   data = trunc_train,
                                   importance = "impurity",
                                   mtry = 10,
                                   min.node.size = 1,
                                   num.trees = 160)

#Determine optimized model accuracy on truncated train data
y_pred_manual_red <- rf_ranger_opt_manual_red$predictions
round((Accuracy(y_pred_manual_red, y_train)), 4)
[1] 0.6805
```

The truncated features model is marginally worse that the manually-tuned model, with an accuracy score of 68.05%.

7.5 - Naive Bayes Classifier

Despite the data under study likely to be too complex for a naive Bayes model, it is worth testing due to naive Bayes being very generalizable and highly efficient in terms of computation cost.

```
library(naivebayes)

#Create naive Bayes model
nb <- naive_bayes(result ~., data = train, usekernel = T, laplace = 1)

#Create naive Bayes model predictions on training data
y_pred_nb <- predict(nb, newdata = train)

#Assess accuracy of naive Bayes model on training data
Accuracy(y_pred_nb, y_train)
[1] 0.4311102
```

As expected the model performance is quite poor, likely due to the near 600 classes found in the target variable factor.

Section 8 - Conclusions

Having completed building and evaluating the models selected for the classification task, one can compare model performance in a data frame storing the individual model's performance. The metrics of out-of-bag (OOB) error will be reported on for random forest models, and RMSE. Accuracy for training and testing sets for each model will be automatically populated using the `mlMetrics::Accuracy()` function as seen in section 7.

```
#Create data frame to store model comparisons
columns <- c("Model", "OOB Error", "RMSE", "Accuracy_Train",
"Accuracy_Test")
model_comparison <- data.frame(matrix(nrow = 0, ncol =
length(columns)))
colnames(model_comparison) = columns
```

Evaluate the default random forest model using the following command:

```
#On train data
y_pred_train <- rf_ranger$predictions
train_acc <- round((Accuracy(y_pred_train, y_train)), 4)

#On test data
pred_ranger <- predict(rf_ranger, test)
y_pred_test <- pred_ranger$predictions
test_acc <- round((Accuracy(y_pred_test, y_test)), 4)

#Model data
oob_error <- round((rf_ranger$prediction.error), 4)
rmse <- round((sqrt(oob_error)), 4)

model_comparison[1,] <- c("Random Forest - Default",
                          oob_error,
                          rmse,
                          train_acc,
                          test_acc)
```

Evaluate the grid-search optimized random forest model using the following command:

```
#On train data
y_pred_train <- rf_ranger_opt_final$predictions
train_acc <- round((Accuracy(y_pred_train, y_train)), 4)

#On test data
pred_ranger <- predict(rf_ranger_opt_final, test)
y_pred_test <- pred_ranger$predictions
test_acc <- round((Accuracy(y_pred_test, y_test)), 4)

#Model data
oob_error <- round((rf_ranger_opt_final$prediction.error), 4)
rmse <- round((sqrt(oob_error)), 4)

model_comparison[2,] <- c("Random Forest - Grid Search Optimized",
                          oob_error,
                          rmse,
                          train_acc,
                          test_acc)
```

Evaluate the manually optimized random forest model using the following command:

```
#On train data
y_pred_train <- rf_ranger_opt_manual$predictions
train_acc <- round((Accuracy(y_pred_train, y_train)), 4)

#On test data
pred_ranger <- predict(rf_ranger_opt_manual, test)
y_pred_test <- pred_ranger$predictions
test_acc <- round((Accuracy(y_pred_test, y_test)), 4)

#Model data
oob_error <- round((rf_ranger_opt_manual$prediction.error), 4)
rmse <- round((sqrt(oob_error)), 4)

model_comparison[3,] <- c("Random Forest - Manually Optimized",
                          oob_error,
                          rmse,
                          train_acc,
                          test_acc)
```

Evaluate the truncated features random forest model using the following command:

```
#On train data
y_pred_train <- rf_ranger_opt_manual_red$predictions
train_acc <- round((Accuracy(y_pred_train, y_train)), 4)

#On test data
pred_ranger <- predict(rf_ranger_opt_manual_red, test)
y_pred_test <- pred_ranger$predictions
test_acc <- round((Accuracy(y_pred_test, y_test)), 4)

#Model data
oob_error <- round((rf_ranger_opt_manual_red$prediction.error), 4)
rmse <- round((sqrt(oob_error)), 4)

model_comparison[4,] <- c("Random Forest - Truncated Features",
                          oob_error,
                          rmse,
                          train_acc,
                          test_acc)
```

Evaluate the naive Bayes model using the following command:

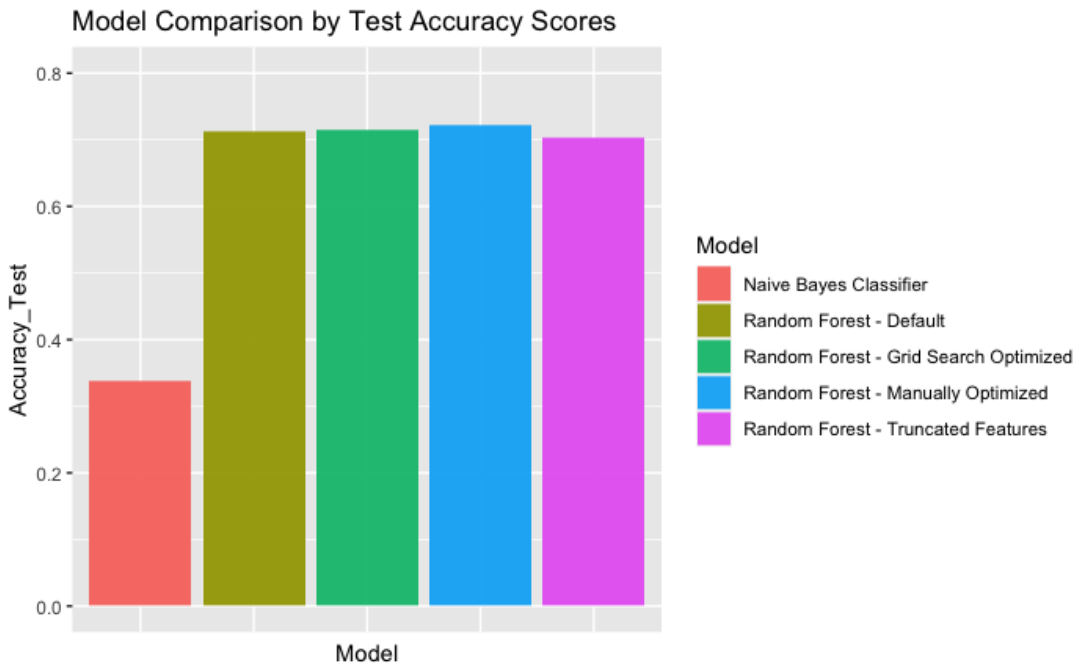
```
#On train data
y_pred_train <- predict(nb, newdata = train)
train_acc <- round((Accuracy(y_pred_train, y_train)), 4)

#On test data
y_pred_test <- predict(nb, newdata = test)
test_acc <- round((Accuracy(y_pred_test, y_test)), 4)

#Model data
oob_error <- "NA"
rmse <- "NA"

model_comparison[5,] <- c("Naive Bayes Classifier",
                          oob_error,
                          rmse,
                          train_acc,
                          test_acc)
```

To conclude the comparison, visualize the model performance in both a table form and visual bar plot form.

```
model_comparison$Accuracy_Test <-
as.numeric(model_comparison$Accuracy_Test)
model_comparison
```

Model	OOB Error	RMSE	Accuracy_Train	Accuracy_Test
Random Forest - Manually Optimized	0.3033	0.5507	0.6967	0.7211
Random Forest - Grid Search Optimized	0.3115	0.5581	0.6885	0.7148
Random Forest - Default	0.3021	0.5496	0.6979	0.7136
Random Forest - Truncated Features	0.3195	0.5652	0.6805	0.7043
Naive Bayes Classifier	NA	NA	0.4311	0.3378

```
ggplot(data = model_comparison, aes(x = Model, y = Accuracy_Test)) +
  geom_col(aes(fill = Model)) +
  scale_y_continuous(limits = c(0, .8)) +
  ggtitle("Model Comparison by Test Accuracy Scores") +
  theme(axis.text.x=element_blank(),
        axis.ticks.x=element_blank())
```



From the created models, the best fitting one in terms of lowest out-of-bag error is the default random forests model. In terms of its goodness of fit, the lowest RMSE model is also the default random forests model. The manually optimized model is the most accurate on the testing data set. Of the 4 random forests, the worst optimized one is the one with truncated features, and is not recommended for deployment into a production environment. The naive Bayes classifier exists as a formality for model diversification as it is not sufficiently complex to handle the excessive amount of factor levels found in the target variable, $result.

Given that the manually optimized model returns the highest overall accuracy, one can proceed with assuming that the computed variable importance for the manually-fitted model is the most appropriate for answering the initial study question. That is to say, the top 5 most important factors in determining a winner in ATP matches for the date range under study are:

1. Player age
2. Player win percentage per surface
3. Player rank
4. Player win percentage per tournament level
5. Player win percentage at general tournament stages / rounds

Section 9 - GitHub Repository

Accompanying .RMD files, knitted HMTML outputs, and additional documentation for this study can be found at https://github.com/rfinatan/Factors-Association-Tennis-Professionals-Winners.