Capstone Project I: Data Wrangling

A. BACKGROUND

For my capstone project 1, **pitch prediction**, I will apply machine learning to predict the next pitch thrown by a Major League Baseball (MLB) starting pitcher. In the 2006 postseason (playoffs), MLB first began using a camera-based system to track the trajectory, speed, spin, break, and location of a pitched ball, called **PITCH*f/x***. Such information about each pitch also allows for the "pitch type" (e.g., Fastball, Cutter, Change-up) to be determined. Early models for classifying pitch types, however, were not very accurate for pitches similar in speed and break and the data from early years includes many misclassified pitch types. Since 2015, a more advanced system (**Statcast**) is being used which integrates doppler radar with high definition video to track all aspects of a game, including pitches, hits, and players. For the 2017 season, **Trackman**, a component of **Statcast**, officially replaced the previous **PITCH*f/x*** system.

B. DATA SOURCE

Considering the goal of the project, predict the next pitch thrown by an **MLB** starting pitcher, I needed to first assure I could collect pitch-by-pitch data for each game where the pitcher of interest pitched at least one (1) inning. Initially, the plan was to use the publicly available data collected by the **PITCH*f/x*** system along with additional resources such as **MLB.com**, **BrooksBaseball.net** (applied machine learning to correct misclassified pitch types by **PITCH*f/x***), and **Fangraphs.com** to interpret the data. To access the **PITCH*f/x*** data, I used **Rstudio** with an **R** package called **pitchRx** and imported the downloaded XML files into an **SQLite** database, later converted to **MySQL** to play with during the SQL training modules. While **PITCH*f/x*** includes data from 2006 to current date, I quickly realized, using SQL queries, groupby's and aggregations, that the data was missing a lot of important information. Many player names, for example, that were associated to unique ID's generated by the system, were missing and could not easily be verified or accessed automatically through other resources. To identify the missing player(s), it would require accessing the teams roster on **MLB.com** and comparing it back to the database to populate the missing information.

Fortunately, as I looked into the best approach to update missing information in the **PITCH*f/x*** data, I came across many other repositories to collect MLB data (both current and historical) by means of **web scraping**, **direct file downloads**, and **API**'s. One such API for MLB data is provided by **Sportradar.com** which also includes the same pitch types, speed, and location provided by either **PITCH*f/x*** or **Statcast**, depending on the year (season). In addition, they also include game-by-game statistics, player profiles and injury reports, and team and venue data that is all well organized and accessible through standard **HTTP Requests** using an **API key**. Upon further review of the documentation, it was apparent that the best approach for wrangling the data I needed was to use **Python** with the API from **Sportradar**. The remainder of this document describes the data wrangling and cleaning steps applied to the MLB data collected using **Python 3.6** and **Sportradar MLB v6.5 API** in a **Linux** (**Ubuntu 18.04**) environment. The code is maintained in a collection of **Jupyter Notebooks** available through my **GitHub** account, https://github.com/markrojas.

C. DATA WRANGLING

MLB utilized the **PITCH*f/x*** system from 2006 – 2014 and then switched to **Statcast** in 2015. Because of the change in systems used to track pitches, I decided I would not consider any data prior to 2015 and for this project, I decided to collect data from the 2016 and 2017 seasons. It is important to note that the **API key** required by **Sportradar** to access the data is **fee-based**, thus, I am currently using a **90-day free trial** with up to 1,000 calls per month. Due to this restriction, I limited the number of pitchers of interest to twenty (20) who were considered "top" pitchers during the 2016 or 2017 or both seasons. The list of pitchers of interest are:

```
pitchers = ['aaron_nola', 'carlos_carrasco', 'carlos_martinez', 'chris_archer',
              'chris_sale', 'clayton_kershaw', 'corey_kluber', 'dallas_keuchel',
              'david_price', 'gerrit_cole', 'jacob_degrom', 'jake_arrieta', 'jose_quintana',
       'marcus_stroman', 'justin_verlander', 'max_scherzer', 'michael_fulmer',
       'stephen_strasburg', 'yu_darvish', 'zack_greinke']
```

Approximately half of the pitchers in the list play for the **National League** while the other half play for the **American League** with some switching teams during the off-season or getting traded mid-season. While there are only a couple of pitchers that switched teams and even less that moved to different leagues (e.g., American → National), it might be interesting to see if this impacted the pitchers pitch selection.

Jupyter Notebooks

initial_api_requests.ipynb

The first step was to use standard **HTTP requests** with **API key** to download the **Sportradar Glossary** in JSON format which provides descriptions for pitch types (e.g., FA = Fastball, CT = Cutter, SL = Slider) and pitch outcomes (e.g., aBK = Balk, kF = Foul Ball, oG0 = Ground Out).

Next, I downloaded the **League Schedules** for 2016 and 2017 regular seasons and **Team Profiles** in JSON format which includes the **Game ID's** and **Team Names and ID's**, respectively. The **Game ID's** are unique identifiers for every game played and required to download the **play-by-play** data files. In a separate **combine_team_and_game_ids.ipynb** notebook, I associated **Team Names** and **ID's** with their respective **Game ID's** and stored as a single CSV file for each year.

I purposely saved each JSON file using `json.dump(...)` to avoid making duplicate HTTP requests since my calls were limited per month.

I also used `json.loads(...)` to open the saved files and then convert to DataFrame so I could select specific rows/columns I believe to be of use for further analysis. The resulting DataFrames were then saved as CSV files using `to_csv(...)`.

update_datetimes.ipynb

Step not included in notebook, I used **baseball-reference.com** to manually download the **pitching game logs** for each of the 20 pitchers for 2016 and 2017 as a CSV file since this site offered a quick drop down menu to select player and season and file type to download. The first challenge (described in DATA WRANGLING CHALLENGES below) was to sync the **Sportradar** game dates with the **baseball-reference.com** dates as they were not formatted the same. This step was important so that I could download the exact game from **Sportradar** in which a pitcher of interest pitched.

Using the game logs from **baseball-reference** and **Sportradar** game ID's with updated dates, I was able to filter the complete list of game ID's for all teams to only include those where a pitcher of interest pitched.

play_by_play_requests.ipynb

Again, using HTTP Requests with API key and list of game ID's, I downloaded every play-by-play JSON file for each pitcher for each season and saved them to device.

D. DATA WRANGLING CHALLENGES

Game Dates: The game dates listed in Sportradar are in UTC standard format but offset by +7 hours more than the actual game date and time. For many evening games, this caused the game date to increase by one day, thus excluding the game ID from list to be downloaded.

- To correct this, I created a function in Python that receives a list of UTC dates as strings and returns a list of new dates, offset by 7 hours using `timedelta` and formatted to exclude Hours, Minutes, and Seconds.
- This allowed the new dates to sync with actual game dates and include all necessary game ID's needed to get play-by-play files.

HTTP Request Limit: Overall, the total number of HTTP Requests I had to make to grab all the play-by-play data for only 20 pitchers and leagues schedules for two regular seasons were more than 2,250 calls.

- I communicated with Sportradar support to gain additional calls, however, if I had included additional pitchers or more seasons, I would have been required to pay a fee.
- I believe that the 2016 data is sufficient for training and the 2017 data for testing different models, thus, will not require additional data for this project at this time.

E. DATA CLEANING

Cleaning the play-by-play data was the most challenging and time consuming process, yet it was also the most informative and educational experience gained from this project thus far. Many of the challenges encountered were, I believe, a result of attempting to store too much information in a single file as compared to utilizing a relational database to spread the data across multiple tables. The play-by-play JSON files consists of detailed real-time information on every pitch and game event. When converting the JSON file to a DataFrame to review shape, info, and description, it is plain to see that many of the columns are deeply nested with dictionaries and lists of dictionaries.

Jupyter Notebooks

clean_play_by_play_data.ipynb

The goal for cleaning the play-by-play data was to eliminate redundancy and unnecessary information such as jersey numbers, lineups, warm-ups, and any information that is not related to the pitcher of interest. In this notebook, I created multiple functions to automate the cleaning process.

The first function, **`make_dfs(…)`**, iterates through the list of pitchers of interest for each year (2016 & 2017) and then calls a second function, **`clean_dfs(...)`** to open and normalize JSON files, convert to DataFrame, de-nest nested values, drop and rename columns, handle null values and merge cleaned data into a single CSV for each pitcher.

In the **`clean_dfs(...)`** function, a single JSON file is opened while a list comprehension extracts the list of entries from the file and normalizes it using **`json_normalize(…)`**

- As a result, a DataFrame is created with columns: ‘number’ which is later renamed to ‘inning’ and ‘halfs’ indicating whether it is the (T)op or (B)ottom of the inning.

After the initial JSON normalization, for columns that contained **nested dictionaries**, I simply convert the DataFrame to a dictionary using **`to_dict(…)`** with an **`orientation=’record’`** and call again **`json_normalize(…)`** with list of columns to keep and column(s) to de-nest.

There are some columns that contain a **list of multiple nested dictionaries** which cannot be de-nested using the **`json_normalize(…)`**.

- In such cases, my approach is to use **`apply.(pd.Series)`** to the column to de-nest the list of dictionaries and concatenate it with the same DataFrame (after dropping the nested column) to create a new DataFrame. Here is an example:

```
# de-nest 'col' using apply(pd.Series) and replace original 'col' column
df_new = pd.concat([df.drop('col', axis=1), df['col'].apply(pd.Series)], axis=1, sort=False)
```

Because I am only interested in pitch events related to the pitcher of interest, I use the **pitcher ID** as a conditional to remove events for all other pitchers. This removes a majority of rows from the DataFrame.

In the end, every cleaned play-by-play data set is merged into a single file for each player for each season to be analyzed during the Exploratory Data Analysis stage.

End results includes forty (40) cleaned CSV files.

- 1 - CSV for each pitcher (x20 pitchers) for every game pitched each season (x2 seasons)

F. DATA CLEANING CHALLENGES

- Inconsistencies in columns: Not all JSON files contained the same columns. Some included ‘Hitter’ and ‘Pitcher’ columns while others did not. Also, some columns only existed if an event occurred, such as ‘Runners’ and ‘Errors’. These columns may not exist at times if there were no runners on base or no errors committed during the game.
    - To address occasional missing columns planned to be dropped, I simply use an `if` statement to check if the column is present in the DataFrame.
    - Because I will need to check for runners on base during EDA, I add ‘Runners’ columns with null values if not present in the DataFrame.

- Double-headers: It is possible for a team to play two (2) games on the same day, referred to as a double-header. Normally only occurs once a season but can happen more if games are postponed due to weather or tragedy. In such cases, a pitcher will pitch only one (1) of the games to avoid injury.
    - To address this, I download and check both games for pitcher of interest. If pitcher of interest did not pitch a game, his ID will not be found and I simply ignore the game and move on.

- Null Values for Pitch Speeds and Pitch Types: I found that there are some events where the Pitch Speed and Pitch Type are null. This occurs for various reasons, such as, camera malfunction, intentional walk, corrupt data.
    - To address Pitch Speeds, I replace the null values with **`mean()`**. I compute mean on a per game basis as the pitchers average pitch speed can vary from game to game.
    - To address Pitch Types, I first check if the pitch outcome was an intentional walk and correctly label the pitch type as ‘IB’. For other cases, I label as ‘UN’ for unknown.
    - For cases where majority or all of the Pitch Speeds or Types are null, I ignore the game to avoid any bias.

- Data available in 2017 data but not in 2016: In 2017, when Statcast implemented a new system called Trackman, they also added two additional data points (X and Y coordinates).
    - The x-coordinate indicates location across x-axis in strike zone.
    - The y-coordinate indicates location across y-axis in strike zone.
    - I kept this information in the 2017 data sets and leave them as null values in 2016 since I will most likely not use it in analysis.

- Special characters in Pitcher names: For some pitchers, there are special characters such as accents and tildes that cause an issue if I attempt to search by first or last name.
    - I implemented a function to replace the special characters but later decided that going based on a non-unique first and last name to identify pitcher was prone to future errors.
    - Instead, I left names as they are and utilized the unique pitcher ID as primary identifier.

- Unwanted columns: As deeply nested columns were un-nested, many unwanted columns and columns with all null values would pop up.
    - To address this, I created a function to remove columns where all rows had null values and to drop columns that were passed in a list.

As a final step to clean the data, before performing EDA, after opening CSV files in DataFrame, I convert data types to reduce memory usage. This may not have a big impact on current data set but can have major impact on much larger data sets collected over an extended period.

- Reduce memory usage: As a final step to ‘clean’ the data was to convert data types to more appropriately correspond to the type of data the DataFrame column is storing.
    - Rather than using an ‘int64’ or ‘float64’ to store 2-3 digit values, I convert the dtype to ‘int8’.
    - For object data types with low cardinality, I convert them to ‘catergory’.
    - While this data set is relatively small, the amount of memory usage decreased approximately **42mb** from the original **167mb** after converting the data types.