Nicolas Moy

DSCI 551

Semester Project Report

Intro: For my semester project, I chose to do the SQL option by myself. I ended up completing the project requirements of implementing CSV parsing and an in-memory data frame with SQL like operations, all without pandas, csv, and other prohibited libraries. I also finished the application using the Streamlit library that demonstrates these functions. I chose the theme of my project to be a NBA data analysis app since my love for basketball and sports had been incredibly strong since before I can even remember. I specifically focused on my favorite team the Golden State Warriors and the 2022/2023 season which was the season after they won the Championship.

Dataset: My datasets for the project were 2 CSVs. I got these CSV datasets from both the NBA API and Kaggle's website. The first CSV is "WarriorsStats.csv" which consists of the 16 warriors players from that season. This was my dataset with all the statistical numbers that is mainly used for the analytical functions. The columns consisted of player id, full name, position, games played, per game stats (minutes, points, rebounds, assists, steals, blocks, turnovers, fouls) field goal pct, three point pct, free throw pct, age, and salary. The second CSV is "Player.csv" which was a list of all the active and inactive players in the NBA. It consisted of an ID column (every NBA player has a unique ID given to them when joining the league), full name, first name, last name, and is active columns. These 2 datasets connect with the IDs since those values are unique and the basically the primary key of each dataset. These 2 datasets have a good variety of datatypes and were fun to work with.

Data loading and Parsing: My project code starts with my function for data loading and parsing. My loading function uses .readlines() to read all the data and everything coming in from .readlines() comes in as a string, but we need the other python datatypes to manipulate the data. So I have a helper function for the loading called convert_value. This function takes in the value and first removes any whitespace with .strip(). Then it removes any quotes if there are any present from the csv file wrapping text fields. Then the datatype value detection begins. The function uses .isdigit() to check if the value is a number. If there any "." in the value, then it is a decimal value so it will convert to a float type. And if the value is true or false the function will put the value as a Boolean type. Finally if none of those checks pass, then the value will be defaulted to a string type. Now that the datatype conversion is handled, my main loading function is called load_csv with the parameters being csv_file and the separator variable being a comma since I am working with comma separated value files. Like I said earlier I open the files with .readlines(). I start with initializing an empty dictionary and storing the first line data (which is the column names) into the dictionary as the keys. Then the rest of the actual data rows, I use

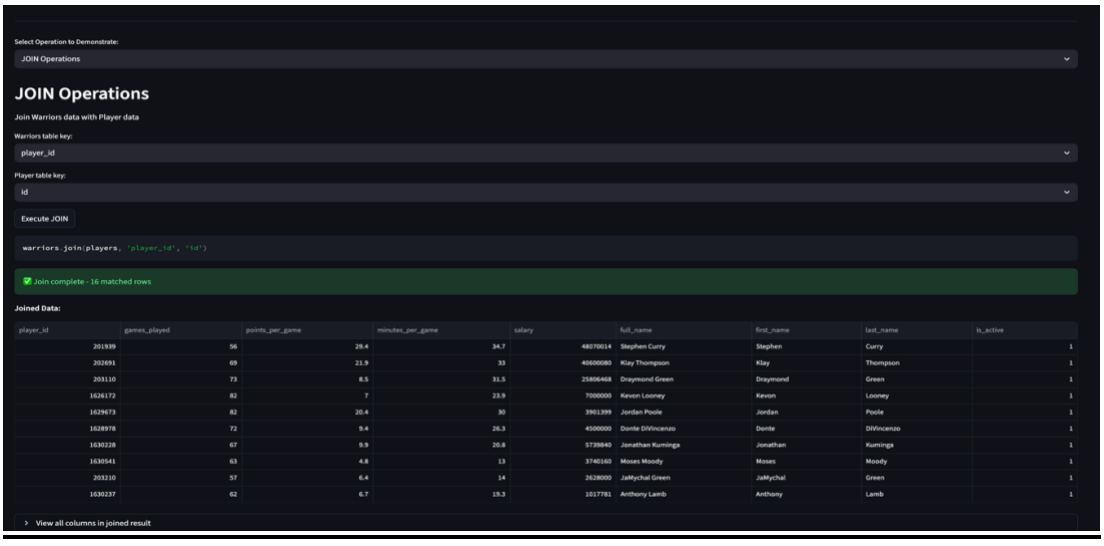
.strip() and .split() to get each individual data value that is run through my convert_value function to store it as its proper data type.

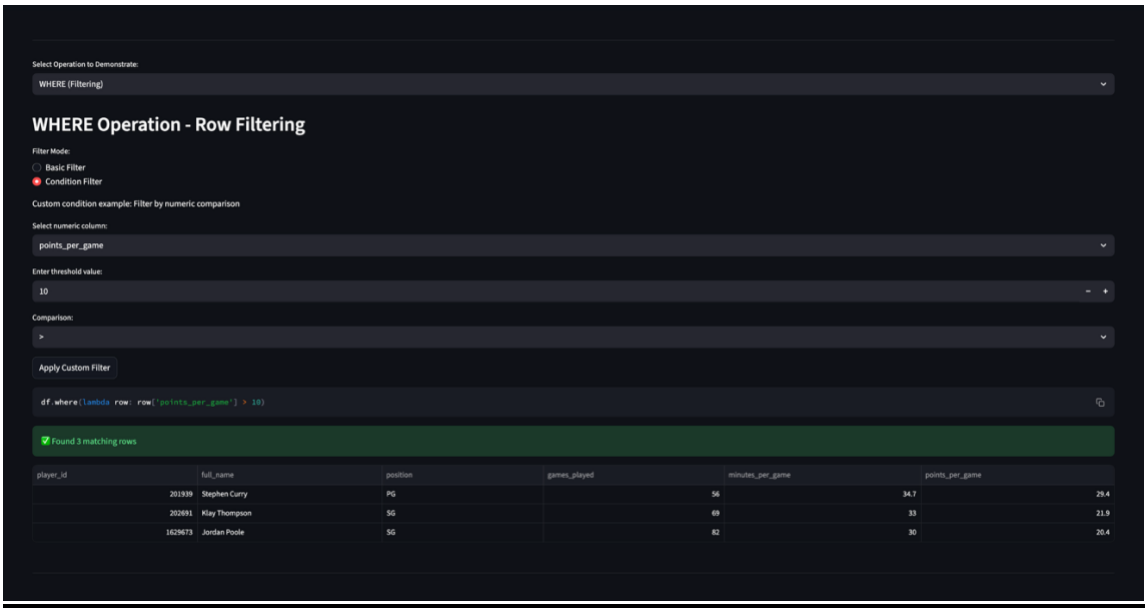
Convert_value and load_csv functions:

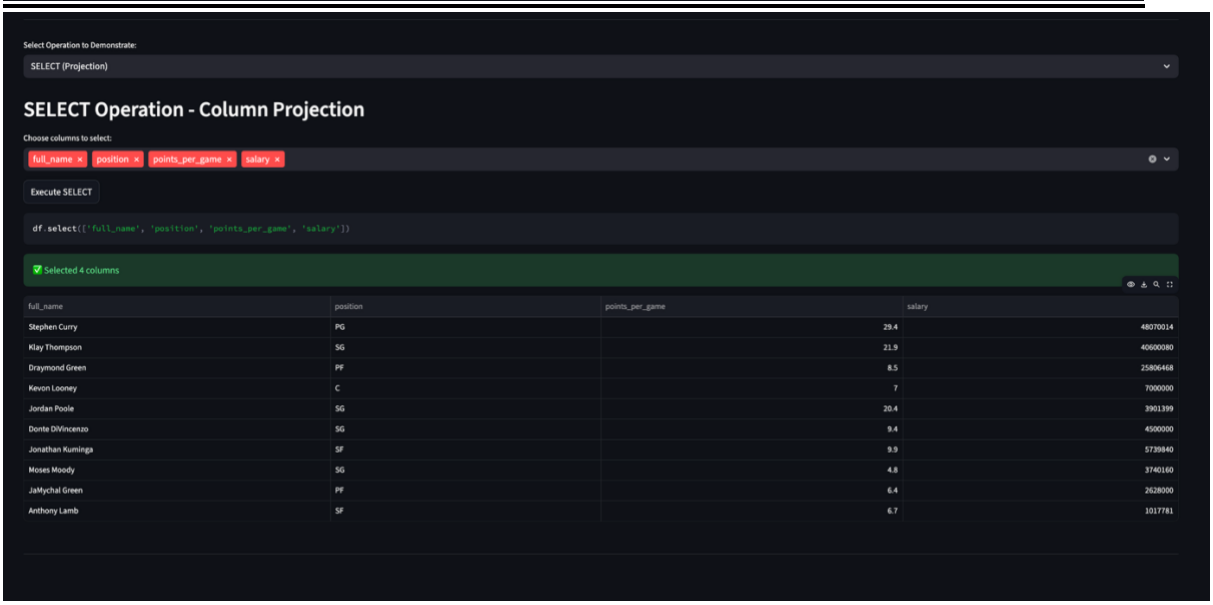
```
Project.py > ...
    # converting values to correct data type
def convert_value(value):
    #Getting ride of whitespace
    value = value.strip()

    # Remove quotes if present
    if (value.startswith('"') and value.endswith('"')) or (value.startswith("'") and value.endswith("'")):
        value = value[1:-1]  # Strip first and last character

    # Int vals
    if value.isdigit() or (value.startswith('-') and value[1:].isdigit()):
        return int(value)

    # Float vals
    try:
        if '.' in value:
            return float(value)
    except ValueError:
        pass

    # Boolean vals
    if value.lower() in ['true', 'false']:
        return value.lower() == 'true'

    # else strings
    return value


#////////////////////  Data loading and parsing  ///////////////////////////////////
def load_csv(csv_file, separator = ','):
    data = {}
    column_names = []

    with open(csv_file, 'r') as file:
        lines = file.readlines()
        column_names = lines[0].strip().split(separator)

        #cleaning column names
        column_names = [col.strip().strip('"').strip("'") for col in column_names]

        # dictionary for storing data
        data = {x: [] for x in column_names}


        for line in lines[1:]:
            stripped_values = line.strip()

            if not stripped_values:
                continue
            values = stripped_values.split(separator)

            for i, column_name in enumerate(column_names):

                #Converting values to their data type
                if i < len(values):
                    converted_value = convert_value(values[i])
                    data[column_name].append(converted_value)

                else:
                    data[column_name].append(None)



    return data, column_names
```

Application: For my application I chose to use the Streamlit library to create the app. It is an interactive dashboard that takes in the 2 CSV files and then has a dropdown menu where the user can apply the SQL functions. I chose Streamlit since it can make the entire application in python with no JS or CSS needed. It uses all my functions imported from the Project.py file. It starts with using load_csv in the sidebar part of the code where player.csv and warriorsstats.csv are submitted and loaded into 2 separate dataframe objects. It uses Streamlit session state which tracks the user's current session, so the program loads the data once and doesn't have to reload for every operation. When the data is loaded and you're in the main homepage, you have a data preview that shows the columns and the first 10 rows of data using the __str__ method. The first operation is the SELECT function where the application calls the df.select method. You can choose which columns to project and then those columns are displayed along with the line of code that runs the select operation. The next operation is the WHERE function where the app calls the df.where method. This operation has 2 choices where the user can pick simple filtering or a conditional filter. The basic filter is dictionary-based filtering looking for an exact match, while the conditional filter uses a lambda function to do numeric comparison with <, >, <=, >=, and ==. In the AGGREGATION operations, the user has the option to choose an aggregation function (MAX, MIN, SUM, AVG, COUNT) and which column to group by, then the app will run whichever corresponding method that was chosen. Finally the last operation is the JOIN operation. The user specifies which columns to join the data on, and in my project I use the player IDs. After the app calls df.join, the result is the combined data of the 2 csv files. Overall, I think my app has a clean layout that is easy to follow and informative.

Application Pics:




Select Operation to Demonstrate:
WHERE (Filtering)
WHERE Operation - Row Filtering
Filter Mode:
Basic Filter
Condition Filter
Custom condition example: Filter by numeric comparison
Select numeric column:
points_per_game
Enter threshold value:
10
Comparison:
>
Apply Custom Filter
df.where(lambda row: row['points_per_game'] > 10)
Found 3 matching rows
player_id full_name position games_played minutes_per_game points_per_game
201939 Stephen Curry PG 56 34.7 29.4
202691 Klay Thompson SG 69 33 21.9
1629673 Jordan Poole SG 82 30 20.4


Select Operation to Demonstrate:
SELECT (Projection)
SELECT Operation - Column Projection
Choose columns to select:
full_name
position
points_per_game
salary
Execute SELECT
df.select(['full_name', 'position', 'points_per_game', 'salary'])
Selected 4 columns
full_name position points_per_game salary
Stephen Curry PG 29.4 48070014
Klay Thompson SG 21.9 40600080
Draymond Green PF 8.5 25806468
Kevon Looney C 7 7000000
Jordan Poole SG 20.4 3901399
Donte DiVincenzo SG 9.4 4500000
Jonathan Kuminga SF 9.9 5739840
Moses Moody SG 4.8 3740160
JaMychal Green PF 6.4 2628000
Anthony Lamb SF 6.7 1017781


Select Operation to Demonstrate:
Aggregation Functions
Aggregation Functions
Select Aggregation:
MAX
Group By Column:
position
Column to Aggregate:
points_per_game
Execute Aggregation
df.max('position', 'points_per_game')
Aggregation complete - 5 groups
position points_per_game_max
PG 29.4
SG 21.9
PF 8.5
C 7
SF 9.9

Dataframe Class and SQL Operations: The heart of my project, the DATAFRAME class. The dataframe class is initialized with a data dictionary, the column names, and a shape tuple that tracks the rows x columns of the dataframe. The dataframe is column oriented, where the columns are stored as dictionary keys and the dictionary values are the column data in python lists. I have 2 helper functions (__getitem__, __str__) for the dataframe class that allow me to have column access (bracket notation) to the data and allow me to print the dataframe. Like we discussed in class, using the strategy with direct dictionary lookup is very efficient. I chose to have all the SQL operations be methods of the class. For the SELECT operation, I just looped through the columns and checked if the requested columns existed. Then if they do exist, I add the data from the specific columns to a new dataframe and then return the new dataframe that has just the selected columns. For the WHERE operation, I first checked to see if the dataframe is empty or not. Then I make a row dictionary that's temporary but makes accessing the values easier since I can access them by column in this new row dictionary. I have 2 condition types for the filtering: lambda function and dictionary matching. If the input is callable aka lambda, then the row dictionary gets passed to the function and the rows that return true are kept. If the input is a dictionary, then I loop through each key-value pair in the dictionary checking if the values match. Like select function, the result of matched rows are returned as a new dataframe. For the aggregation, I need the GROUP BY function first to use in all the different operations. The group by function starts with confirming all the columns to be grouped exist and are in a python list. Then I loop through each row in the dataframe, getting the values from the grouping column for that row. Next a group key is made and if the function is grouping by a single column, the key is just that value. If the function is grouping by multiple columns, the key is a tuple of those values (tuples are hashable so they can be dictionary keys). If the key doesn't exist I make a new list for it and if it does exist then it gets appended to the existing key list. Finally the group by function returns a dictionary of all the groups. It doesn't actually return any data, just row indices. This results in more efficiency since the aggregation functions only pull the required indices the need. The AVG, SUM, MAX, MIN, and COUNT functions are all pretty similar. In each of the functions, they first run the group by function to determine the chosen groups. Then for each group key and row index it extracts the relevant values, performs the respective aggregation, and returns a new dataframe with the aggregation complete. Now finally the last operation I did was the JOIN operation. My join implementation was an inner join so it only joined on matches found in both datasets. First my join function starts with validating the given join keys exist. Then I initialize an empty list for the resulting columns and empty dictionaries for the result data and the changed columns (conflicting column names). I build the result list of columns with the left and right dataset columns (skipping the right key for no duplicates). If there's a column in the right dataset that already exists in the left dataset, I added a "_right" to its name to avoid this name collision. Next I have a nested loop to find the matches. The outer loop goes through each row in the left dataset, getting the key value for the row, then iterates through the right dataset in the inner loop checking each row for matching values. If a match is found, then the data from both rows is put into the result dataframe and this result dataframe is returned.

Dataframe and SQL operation functions:

```python
# ////////////////////  Dataframe class  ////////////////////////////////////
class dataFrame:

    #Initalizing dataframe
    def __init__(self, data, column_names = None):
        self.data = data
        self.column_names = column_names
        self.shape = (len(data[self.column_names[0]]) if self.column_names else 0, len(self.column_names))


    #Getting a column from the dataframe
    def __getitem__(self, key):
        if key in self.data:
            return self.data[key]
        else:
            raise KeyError(f"Column '{key}' not found!")


    #Printing the dataframe
    def __str__(self):
        if not self.column_names:
            return "Empty DataFrame"

        # Create a simple table representation
        result = []

        # Header row(limit to first 5 columns)
        header_row = " | ".join(f"{h:<20}" for h in self.column_names[:5])
        result.append(header_row)
        result.append("-" * len(header_row))

        # Data rows (first 10 rows)
        num_rows = min(10, self.shape[0])
        for i in range(num_rows):
            row_data = []
            # Limit first 5 columns
            for header in self.column_names[:5]:
                value = str(self.data[header][i])
                row_data.append(f"{value:<20}")
            result.append(" | ".join(row_data))

        if self.shape[0] > 10:
            result.append(f"... ({self.shape[0] - 10} more rows)")

        result.append(f"\nDataFrame Shape: {self.shape}")
        return "\n".join(result)
```

:

```python
# #////////////////////   Select function aka projection   ////////////////////////////////////
def select(self, columns):
    # Validate that requested columns exist
    for col in columns:
        if col not in self.column_names:
            raise KeyError(f"Column '{col}' doesn't exist!")

    # New data dictionary with only selected columns
    new_data = {}
    for col in columns:
        new_data[col] = self.data[col]

    # Return new dataFrame
    return dataFrame(new_data, columns)



# #////////////////////   Where function aka filtering   ////////////////////////////////////
def where(self, condition):
    if not self.column_names or self.shape[0] == 0:
        return dataFrame({}, [])

    # Determine which rows to keep
    rows_to_keep = []

    for i in range(self.shape[0]):
        # Build a row dictionary for this index
        row = {col: self.data[col][i] for col in self.column_names}

        # Check condition
        if callable(condition):
            # If condition is a function, call it with the row
            if condition(row):
                rows_to_keep.append(i)
        elif isinstance(condition, dict):
            # If condition is a dict, check all key-value pairs match
            match = True
            for col, val in condition.items():
                if col not in self.data:
                    raise KeyError(f"Column '{col}' not found!")
                if self.data[col][i] != val:
                    match = False
                    break
            if match:
                rows_to_keep.append(i)
        else:
            raise ValueError("Condition must be a function or dictionary")

    # Build new data with only the rows we're keeping
    new_data = {}
    for col in self.column_names:
        new_data[col] = [self.data[col][i] for i in rows_to_keep]

    return dataFrame(new_data, self.column_names)
```

```python
#////////////////////   Group by function   ////////////////////////////////////
def group_by(self, columns):
    if isinstance(columns, str):
        columns = [columns]

# Validate columns exist
    for col in columns:
        if col not in self.column_names:
            raise KeyError(f"Column '{col}' not found!")

# Create groups
    groups = {}

    for i in range(self.shape[0]):
        # Create group key from values in group columns
        key_parts = []
        for col in columns:
            key_parts.append(self.data[col][i])

        # Use tuple as key (hashable)
        group_key = tuple(key_parts) if len(key_parts) > 1 else key_parts[0]

        # Add row index to this group
        if group_key not in groups:
            groups[group_key] = []
        groups[group_key].append(i)

    return groups



##////////////////////   Aggregation functions   ////////////////////////////////////
def avg(self, group_column, average_column):
    #Grouping
    groups = self.group_by(group_column)

    result_data = {
        group_column: [],
        f'{average_column}_avg': []
    }

    for group_key, row_indices in groups.items():
        result_data[group_column].append(group_key)

        #Getting the values to average
        values = [self.data[average_column][i] for i in row_indices]
        values = [i for i in values if i is not None]

        # Taking the average of the values
        avg_val = sum(values) / len(values) if values else 0
        result_data[f'{average_column}_avg'].append(avg_val)

    return dataFrame(result_data, [group_column, f'{average_column}_avg'])
```

```python
def sum(self, group_column, sum_column):
    #Grouping
    groups = self.group_by(group_column)

    result_data = {
        group_column: [],
        f'{sum_column}_sum': []
    }

    for group_key, row_indices in groups.items():
        result_data[group_column].append(group_key)

        #Getting values to sum, removes None
        values = [self.data[sum_column][i] for i in row_indices]
        values = [i for i in values if i is not None]

        #"built ins" because there is a name conflict with pythons sum
        total = sum(values) if values else 0
        result_data[f'{sum_column}_sum'].append(total)

    return dataFrame(result_data, [group_column, f'{sum_column}_sum'])


def max(self, group_column, max_column):
    #Grouping
    groups = self.group_by(group_column)

    result_data = {
        group_column: [],
        f'{max_column}_max': []
    }

    for group_key, row_indices in groups.items():
        result_data[group_column].append(group_key)

        #Getting values to find max, removes none
        values = [self.data[max_column][i] for i in row_indices]
        values = [i for i in values if i is not None]

        # Selecting the Maximum out of all the values
        max_val =  max(values) if values else None
        result_data[f'{max_column}_max'].append(max_val)

    return dataFrame(result_data, [group_column, f'{max_column}_max'])
```

```python
def min(self, group_column, min_column):
    #Grouping
    groups = self.group_by(group_column)

    result_data = {
        group_column: [],
        f'{min_column}_min': []
    }

    for group_key, row_indices in groups.items():
        result_data[group_column].append(group_key)

        #Getting values to find min, removes none
        values = [self.data[min_column][i] for i in row_indices]
        values = [v for v in values if v is not None]

        # Selecting the minimum out of all the values
        min_val = min(values) if values else None
        result_data[f'{min_column}_min'].append(min_val)

    return dataFrame(result_data, [group_column, f'{min_column}_min'])


def count(self, group_column):
    #Grouping
    groups = self.group_by(group_column)

    result_data = {
        group_column: [],
        'count': []
    }

    for group_key, row_indices in groups.items():
        result_data[group_column].append(group_key)

        # Counting number of rows
        count_val = len(row_indices)
        result_data['count'].append(count_val)

    return dataFrame(result_data, [group_column, 'count'])
```

```python
#////////////////////   Join functions   ////////////////////////////////////
def join(self, df_to_join, left_key, right_key):
    # Validate keys exist
    if left_key not in self.column_names:
        raise KeyError(f"Left key '{left_key}' not found in left DataFrame!")
    if right_key not in df_to_join.column_names:
        raise KeyError(f"Right key '{right_key}' not found in right DataFrame!")

    # Initialize result columns
    result_data = {}
    result_columns = []
    changed_columns = {}

    # Add all columns from left DataFrame
    for col in self.column_names:
        result_data[col] = []
        result_columns.append(col)

    # Add columns from right DataFrame (skip join key to avoid duplication)
    for col in df_to_join.column_names:
        if col != right_key:
            # Check for matching columns that isn't key
            if col in result_columns:
                new_col_name = f"{col}_right"
                result_data[new_col_name] = []
                result_columns.append(new_col_name)
                changed_columns[col] = new_col_name

            # No matching columns found
            else:
                result_data[col] = []
                result_columns.append(col)


    for i in range(self.shape[0]):
        left_value = self.data[left_key][i]

        # Find matching rows in df_to_join
        for j in range(df_to_join.shape[0]):
            right_value = df_to_join.data[right_key][j]

            # Match found in both tables
            if left_value == right_value:
                for col in self.column_names:
                    result_data[col].append(self.data[col][i])

                for col in df_to_join.column_names:
                    if col != right_key:
                        if col in changed_columns:
                            result_data[changed_columns[col]].append(df_to_join.data[col][j])
                        else:
                            result_data[col].append(df_to_join.data[col][j])

    return dataFrame(result_data, result_columns)
```

What I've learned: Throughout this project I got to learn a few things. I first gained a lot of respect and appreciation for the people who create python libraries. Not being able to just import what I need was challenging and I can imagine all the work and complexity that goes into creating robust libraries. I also learned that the pandas and csv libraries must have very complex and strong type conversion and error handling. Another thing I've learned is that I have a new least favorite SQL operation which is join. Doing the join operation was the most challenging throughout the project. I had a hard time trying to figure out how to handle conflicting columns during a join operation, ultimately deciding to add a "_right" column to the resulting data from the column that was conflicting originating in the right dataset. I also found it easier to separate the group by functionality into its own function. At first, I tried to do the grouping code in each aggregation method itself, but I found it made more sense and felt cleaner to have a separate function for the grouping code and just call that function in each aggregation method. Dealing with real world data isn't perfect which I knew but I got a reminder during this project with my dataset having inaccuracies with the player IDs. When trying to solve the join operation, I had a lot of extra time spent debugging thinking my code was wrong, but the data was inconsistent. I was getting incorrect matching because the player csv from Kaggle had incorrect player ids for some of the warriors' players. So I learned that even if the data is from a reputable source, it can still be messy/incomplete. Another thing I learned is that doing UX/UI isn't as easy as it seems. I was stressing over how to make my app looks, how complex/simple it should be, and how to present it in the most informative and effective way. I learned many things from this project and with my future hopes in ML, I know this experience of parsing and handling data will help me down the line.