# Final Project Report: Weather & College Football

SI206

Team: Big Steppas 2024

Members: Emily Jennett, Sriram Kumaran, Daniel Vega

**Original Goal**

Our project's objective is to identify relationships between weather conditions and college football game outcomes by integrating and analyzing data from three APIs. By collecting game statistics such as team performance, game locations, and scores from the College Football API and comparing this with weather data including temperature, wind speed, precipitation, and air quality from weather APIs, we aim to identify potential correlations. More specifically, we seek to determine if various weather conditions impact scoring, win-loss records, and overall team performance. Through statistical analysis and visualizations created with Matplotlib and Seaborn, we will uncover trends and insights that will develop our understanding of how environmental factors impact college football games.

**Achieved Goal**

We were able to successfully identify relationships between weather conditions and college football game outcomes. The team collected game statistics such as team performance, game locations, and scores from the College Football Data API and combined this with weather data including temperature, wind speed, and precipitation from historical weather APIs. Despite initial challenges with data inconsistencies due to mismatched longitude and latitude values, we resolved these issues and created a database linking football games and corresponding weather conditions. In the end, we uncovered trends such as the influence of extreme wind speeds on scoring and correlations between temperature ranges and total points scored. These insights supported our original hypothesis that environmental factors impact college football games.

**Problems Faced**

In our project, we encountered several challenges starting with the data collection step, especially when working with the weather API and creating the weather_data table. While we were able to successfully store 100 unique items in our database without duplication, many of the entries were either incomplete or incorrect. For example, only 63 out of 100 weather_id values were stored correctly, the rest showing up as "NULL." Additionally, the precipitation data (rain and snow conditions) was consistently recorded as 0, all temperature values were negative, and the wind speeds were unrealistically high. After a closer analysis of the data, we realized the longitude and latitude data were swapped, affecting the weather precipitation, temperature, and wind speeds.

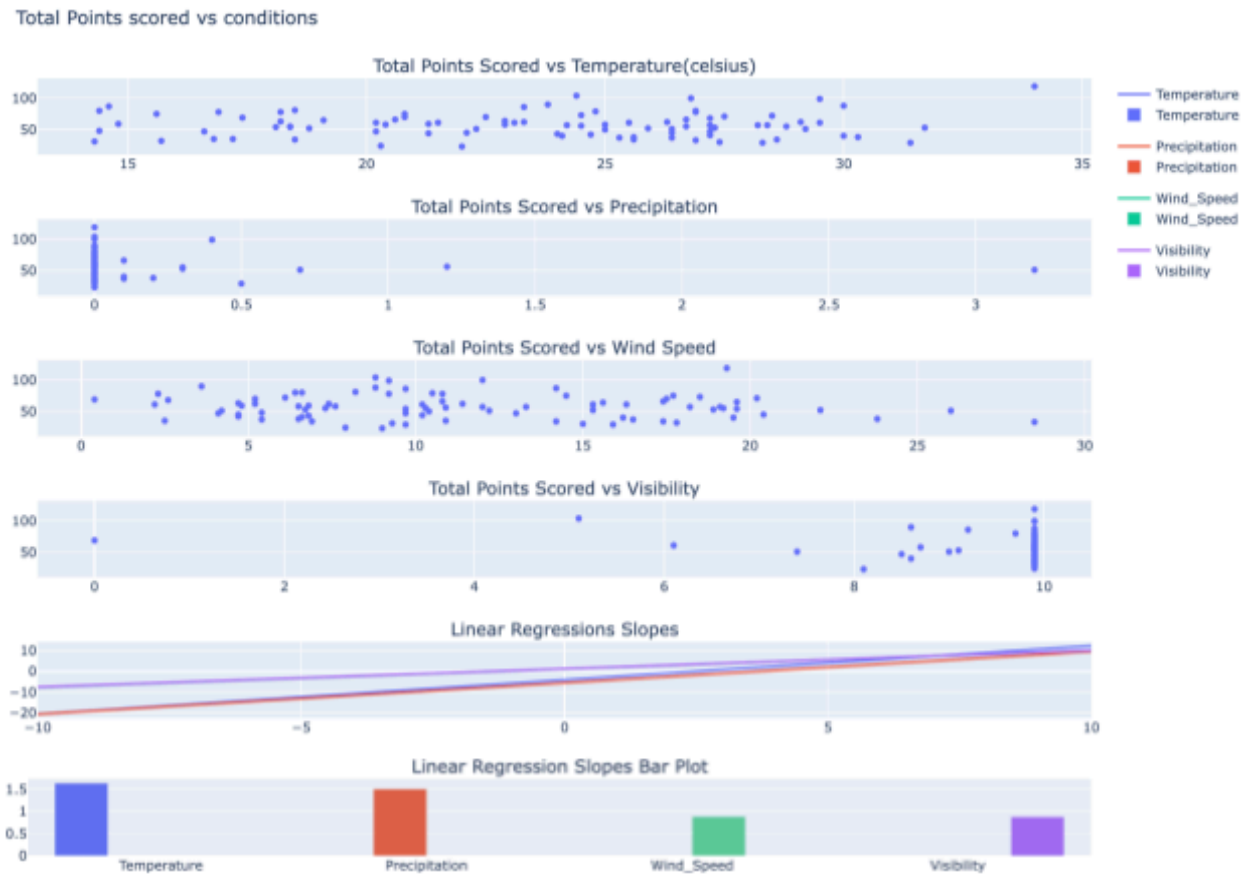
**Calculation File (Screenshot)**

```python
def fetch_football_weather_data():
    """
    Preform an INNER JOIN to retreive each football game's total points, temp, precipitation, and wind speed.
    """
    join_command = f"""SELECT football_games.total_points,
                          weather_data.temperature,
                          weather_data.precipitation,
                          weather_data.wind_speed,
                          weather_data.visibility
                   FROM football_games
                   INNER JOIN weather_data
                   ON football_games.weather_id = weather_data.weather_id"""
    conn = sqlite3.connect(FOOTBALL_DB_FILENAME)
    cursor = conn.cursor()
    cursor.execute(join_command)
    rows = cursor.fetchall()
    conn.close()
    return list(filter(lambda row: is_valid_row(row), rows))
```

```python
#calculate linear regression
def calc_linear_regression(rows, x_index, y_index):
    """
    Calculate the linear regression (slope and intercept) between two variables.

    Args:
        rows (list of tuples): the data retrieved from the database.
        x_index (int): the index of the independent variable (temp, precipitation, or wind_speed).
        y_index (int): the index of the dependent variable (total_points).

    Returns:
        dict: a dictionary with the slope and intercept.
    """
    # extracting x and y values
    x_values = [row[x_index] for row in rows]
    y_values = [row[y_index] for row in rows]
    # print(x_values)
    # print(y_values)

    # number of data points
    n = len(x_values)

    # calculate sum
    x_sum = sum(x_values)
    # print(x_sum)
    y_sum = sum(y_values)
    # print(y_sum)
    x_sum_squared = sum(x ** 2 for x in x_values)
    # print(x_sum_squared)
    xy_sum = sum(x * y for x, y in zip(x_values, y_values))
    print(xy_sum)

    # calculate slope (m) and intercept (b):
    denominator = (n * x_sum_squared - x_sum ** 2)
    if not denominator == 0:
        m = (n * xy_sum - x_sum * y_sum) / denominator
    else:
        m = 99999999
    # print(m)
    b = (y_sum - m * x_sum) / n
    # print(b)

    return {"slope": m, "intercept": b}

```

```
{
    "linear_regressions": {
        "temperature": {
            "slope": 0.18546456907775505,
            "intercept": 53.26037675200547
        },
        "precipitation": {
            "slope": -2.826855771168894,
            "intercept": 57.854726696351676
        },
        "wind_speed": {
            "slope": -0.316444484752332,
            "intercept": 61.227080608099826
        }
    },
    "data": [
        {
            "total_points": 88,
            "temperature": 30.0,
            "precipitation": 0.0,
            "wind_speed": 8.8,
            "visibility": 9.9
        },
        {
            "total_points": 53,
            "temperature": 27.3,
            "precipitation": 0.0,
            "wind_speed": 6.7,
            "visibility": 9.1
        },
        {
            "total_points": 61,
            "temperature": 23.1,
            "precipitation": 0.0,
            "wind_speed": 15.3,
            "visibility": 9.9
```


**Data Visualizations (5 total)**



**Instructions for running the code:**

1. Run the bash script run.sh ( it runs all the files in the order )
2. Terminal run command  chmod +x run.sh
3. Terminal run command ./run.sh

**Code Documentation (includes describing the input and output for each function)**

**GET DATA**

"weatherData.py" Functions

**fetch_weather() -** Uses the API Open-Meteo to retrieve hourly weather data for a given location and date range. The inputs are latitude, longitude, start_date, and end_date in YYYY-MM-DD format. It constructs the API request URL and fetches data for parameters such as temperature, humidity, precipitation, wind speed/direction, and cloud cover. The output is the parsed json containing the weather data or an empty list if there was an error.

**fetch_visibility() -** Fetches hourly visibility data for a specific location and date using the Visual Crossing Weather API. It takes latitude, longitude, date, and an apiKey as inputs. The function constructs the API request URL and retrieves hourly visibility data. It returns the parsed JSON response or an empty dictionary if an error occurs.

**create_weather_table() -** This function sets up the weather_data table in an SQLite database named football_data.db. It deletes any existing weather_data table to make sure we have a clean setup. The table sets up fields for weather ID, city, latitude, longitude, date, temperature, precipitation, wind speed/direction, humidity, cloud cover, and visibility. It doesn't take any inputs and only confirms successful table creation as its output.

**find_closest_time_index() -** This function identifies the index of the closest time in a list (time_list) to a target time (target_time). The input time_list consists of time strings in local time, while target_time is a UTC timestamp. The function compares parsed times to find the closest match and outputs the index of that time.

**fetch_football_data_from_db() -** This function retrieves game data from the football_games table in the SQLite database. It selects records where the weather_id field is NULL, indicating that weather data has not yet been associated with the game. It doesn't require any inputs and outputs a list of tuples, with each tuple containing game data such as game ID, date, time, city, latitude, and longitude.

**addWeatherDataFromDb()** - This function processes football game data in batches, fetches corresponding weather and visibility data, and inserts it into the weather_data table. It links weather data to games by updating the weather_id in the football_games table. Inputs include an optional batch_size (default is 25). It retrieves game data from the database, calls fetch_weather and fetch_visibility, and inserts data into the database. The output is a log of progress and a confirmation of successful batch processing.

## “footballData.py“ Functions

**fetch_games()** - This function retrieves game data for a specified year and season type from the College Football Data API. Inputs are year (int) and an optional season_type (defaults to "regular"). It constructs the API request using the given parameters and includes an authorization header. The output is a JSON object containing game data or an empty list if the request fails.

**fetch_venues()** - This function fetches data about football venues using the College Football Data API. It constructs an API request with an authorization header and retrieves venue details such as city and geographic coordinates. The function outputs a JSON object containing venue data or an empty list if the request fails.

**combine_data()** - This function combines game data and venue data into a unified format. Inputs are games and venues, which are lists. It maps each game to its venue using a dictionary for looking items up. The function outputs a list of dictionaries where each dictionary represents a game with additional venue details and derived metrics like total points and point difference.

**create_table()** - This function creates an SQLite database table named football_games to store football game and venue data. The table includes fields for game details (teams, scores, and results), venue information lake city, latitude, and longitude, and a foreign key for weather data.

**addFootBallDataToTable()** - This function inserts the combined game and venue data into the football_games table in batches. Inputs include combined_data (a list of dictionaries from combine_data) and an optional batch_size of 25 items. The function processes up to 100 records, preparing data for insertion and committing it to the database. It ensures that any remaining records are added after the main batching loop. The output is a log message indicating successful data insertion or an error message if an issue occurs.

“sqlfunctions.py“ Functions

Importing footballData and weatherData, these modules provide details for fetching football games, venues, weather data, and combining/storing the information into an SQLite database. The script calls fetch_venues() to retrieve venue information like city and geographic coordinates, then combine_data() with the fetched games and venues as inputs. This step matches each game with its matching venue. The output is a list of combined dictionaries, each representing a game with its associated venue details.

“createDatabase.py“ Functions

**create_football_games_table() -** This function is responsible for creating a football_games table in a SQLite database file. The function takes one input (db_name) which represents the name of the database file where the table will be created. The purpose of the table is to store information about football games, including game location, points, and performance statistics.

**PROCESS THE DATA**

“processData.py“ Functions

**fetch_football_weather_data() -** The fetch_football_weather_data() function retrieves data by doing an SQL INNER JOIN between the football_games and weather_data tables in the SQLite database made in the gathering data step. It combines columns from both tables based on the weather_id key, matching records. The function uses SQL queries to select total_points, temperature, precipitation, and wind_speed values from the joined tables. It then filters out rows with any None values using the helper function is_valid_row, making sure it completes and valid data is returned as a list of tuples.

**is_valid_row() -** This function checks if all elements in each row are valid, returning False if any value is None and True otherwise. It is a utility function used within fetch_football_weather_data() to ensure that rows containing incomplete data are excluded from the analysis for the visualization step.

**calc_linear_regression() -** This function computes the linear regression parameters, slope and intercept. It takes as input the list of data rows, the index of the independent variable (x), and the dependent variable (y). It takes x and y values, calculates the sums, and applies it to the linear regression formula. The function handles edge cases such as division by zero, by assigning a placeholder value for the slope if the denominator becomes zero. Lastly, results are returned as a dictionary containing the slope and intercept.

**save_to_json_file() -** Save_to_json_file() saves the joined data and calculated regression done previously and returns them into a structured JSON file. It formats the data as a list of dictionaries where each dictionary corresponds to a row containing total_points, temperature, precipitation, and wind_speed. The

regression results are included in the output JSON under the linear_regressions key. The formatted data is then written to the process_data.json file with indentation for readability, making sure the processed data is easily accessible for the visualization step.

**VISUALIZE THE DATA**

"DataVisualization.py" Functions

**load_data()** - Opens and reads the process_data.json file, containing game and weather data in JSON format. It uses Python's built-in json library to load the file's contents into a Python dictionary. This dictionary enables structured and efficient access to the data for further processing. The function ensures clean data loading and returns the parsed data to be used across other functions.

**initialize_dict()** - This function extracts attributes from the loaded data and organizes them into separate lists. The attributes are total_points, temperature_celsius, precipitation, wind_speed, and visibility. For each attribute, the function iterates through the JSON data, appending values to their respective lists. These lists are returned as a tuple.

**set_plot_data()** - Organizes the lists returned by initialize_dict() into a dictionary called plot_data. This dictionary maps each weather metric (temperature, precipitation, wind_speed) to its corresponding list of values, making the data more accessible for plotting. The function returns the original lists to ensure compatibility with other functions, such as linear regression calculations.

**linear_regression()** - Performs a linear regression analysis for temperature, precipitation, wind speed, and visibility metrics in relation to total_points. It uses calc_linear_regression() to compute the slope and intercept of the regression line for each category. The function also assigns descriptive labels to each regression line for clarity.

**make_plot()** - This function is the core of the script, combining data, the analysis, and visualizations. It starts by retrieving the processed data from set_plot_data() and computes linear regression results using the linear_regression function. To create regression line plots, it generates tailored data points using NumPy, which are then transformed into a Pandas DataFrame for compatibility with Plotly's plotting tools.

**Resource Documentation**

| Date | Issue Description | Location of Resource | Result (did it solve the issue?) |
|---|---|---|---|
| 12/06/2024 | Our team needed historical weather data (location, temperature, humidity, precipitation, and wind speed) for analysis of correlation between weather and football game scores. | Historical Weather API | Yes, it provided detailed historical weather data within the year (2020) which aligned with our football game data. |
| 12/07/2024 | We needed access to this college football game data to identify games' total scores and location, so we can align location of games to location of weather condition. | ESPN API List | Yes, it offered a list of potential endpoints for game data retrieval. |
| 12/09/2024 | We needed access to the visibility data from certain locations but were not provided by the initial weather API we used | Visibiltiy API | Yes, if offers a detailed response on the visibility of the location on a specific date |