**SI 206 W25 Final Project Report**

Sarah Chauloult, Zejun Li, Nicole Salinas

Link to GitHub repo: <https://github.com/zejun-l/206-Final-Project>

**Project Goals**

1. The goal of our project was to find out how weather affects performance in ultimate frisbee by gathering game data from women’s D1 college frisbee tournaments in 2024.
   1. We chose to only include tournaments from 2024 to ensure other variables (changes in roster and rules) wouldn’t severely impact our results.
2. We used the following website and APIs:
   1. USAU: <https://play.usaultimate.org/events/tournament/>
      1. We gathered several tournament event pages and game schedules to scrape game data, including the date, location, names of the winner and loser, and the final score.
   2. Geocoding API: <https://openweathermap.org/api/geocoding-api>
      1. We collected the exact coordinates of the location of each game to use with the weather API.
   3. Weather API: <https://open-meteo.com/en/docs/historical-weather-api>
      1. We collected the historical weather (wind speed, temperature, and precipitation) of the location of each game using the latitude and longitude coordinates obtained from the Geocoding API.

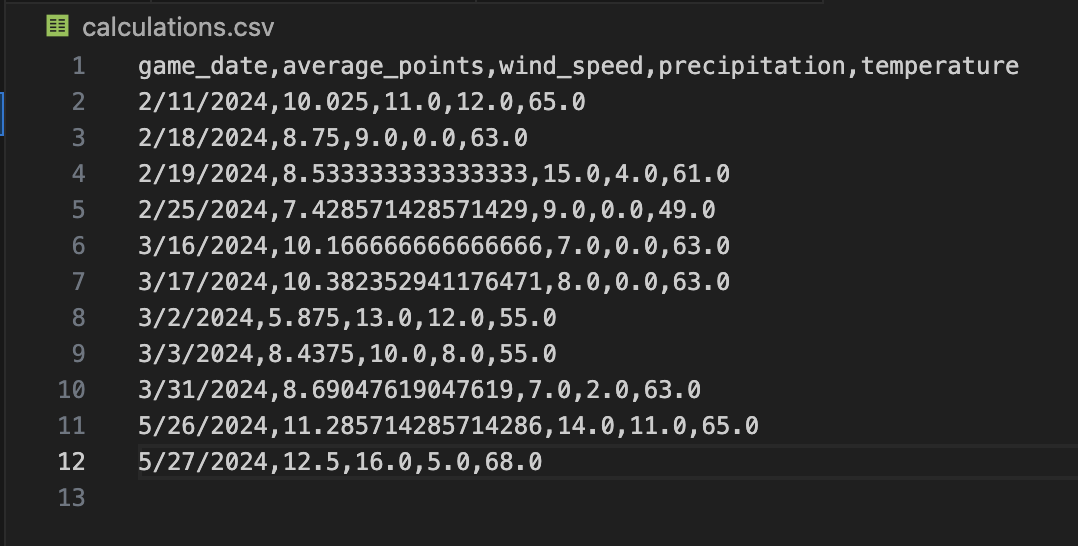
**Goals Achieved**

1. We used the website and two APIs listed above to achieve our goal of gathering game data to find out how weather affects frisbee performance.
   1. We scraped two pages for each tournament: the event page for the tournament location, and the schedule page for game dates and other data, including the names of the winner, loser, and the final score.
   2. After obtaining the game data, we used the geocoding API to get the latitude and longitude of the location of each game to use with the weather API to obtain the wind speed, temperature, and precipitation of the location of the game on that day.
2. The Weather API doesn’t require an API key, so it is easier to gather information, so it allows for flexible customization of weather data, including temperature, wind speed, and precipitation, as well as control over units and timezone.
3. Using the game’s latitude, longitude, and date from the existing database (games.db), the script successfully pulled the exact weather from Open-Meteo’s historical weather archive for each game.
4. The weather data was then stored in a new table called weather, linked to each game by game\_id
5. The weather values are stored in appropriate formats (integers, not blobs) and follow proper database design.

**Problems Faced**

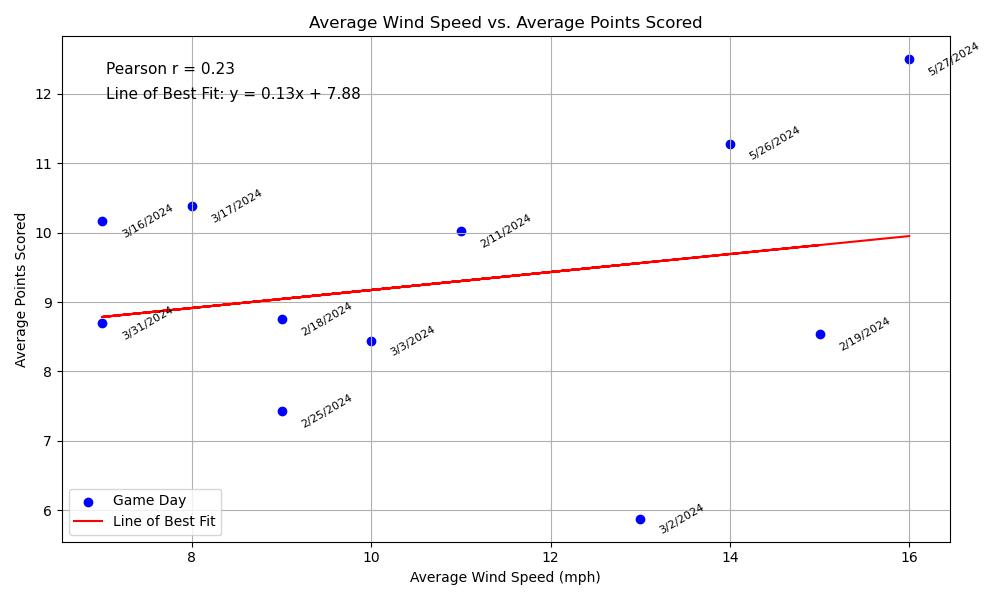
1. One of the biggest problems we faced was figuring out what HTML classes to target to get all the game data we needed, and then having to clean the names of the winners and losers, final score, and skipping games that had missing information. It took a lot of trial and error and debugging to get it to where we wanted.
2. Another problem was when we were testing our code and running it over and over, the pages we were scraping to fill our database would get unhappy and stop responding. We’d have to wait a while before trying again, which was frustrating.
3. We also realized halfway through that a lot of our data contained duplicate strings, so we had to break them down into different tables that shared keys, which took a little bit to code and make it run smoothly.
4. One problem we faced with the geocoding API was that one of the location names wasn’t returning any matches in their database. We found out that the location name was actually two different towns, so we had to separate the name using split and found the coordinates for one of the locations.
5. One of the biggest challenges with the Weather API was getting the weather values to store properly in SQLite. Initially, the weather data was returned as NumPy float values, which were mistakenly stored in the database as BLOBs (binary data).
6. This was fixed by explicitly casting each weather value to an integer before inserting it into the database using int().

**Calculations**

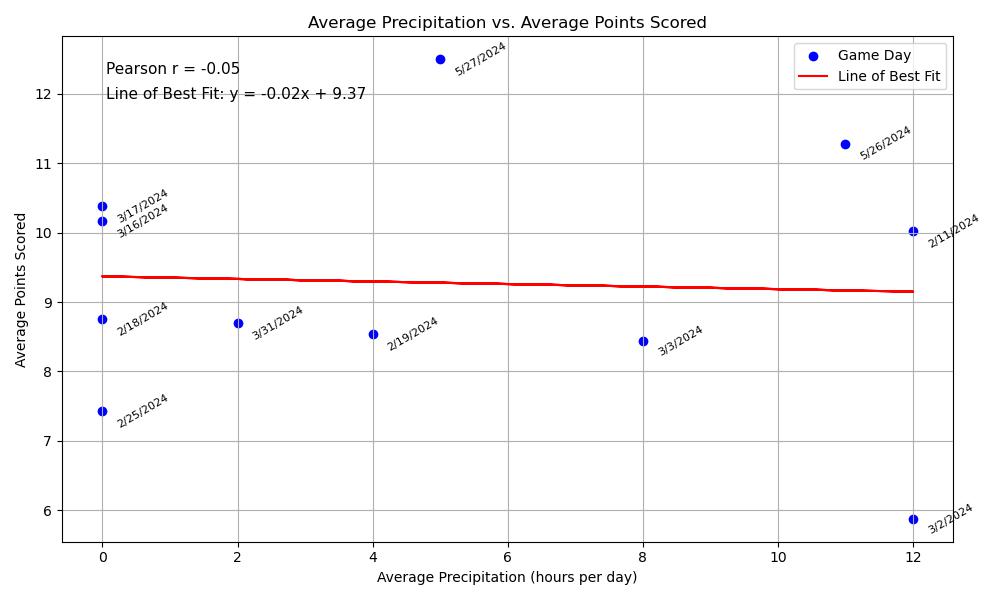


**Visualizations**

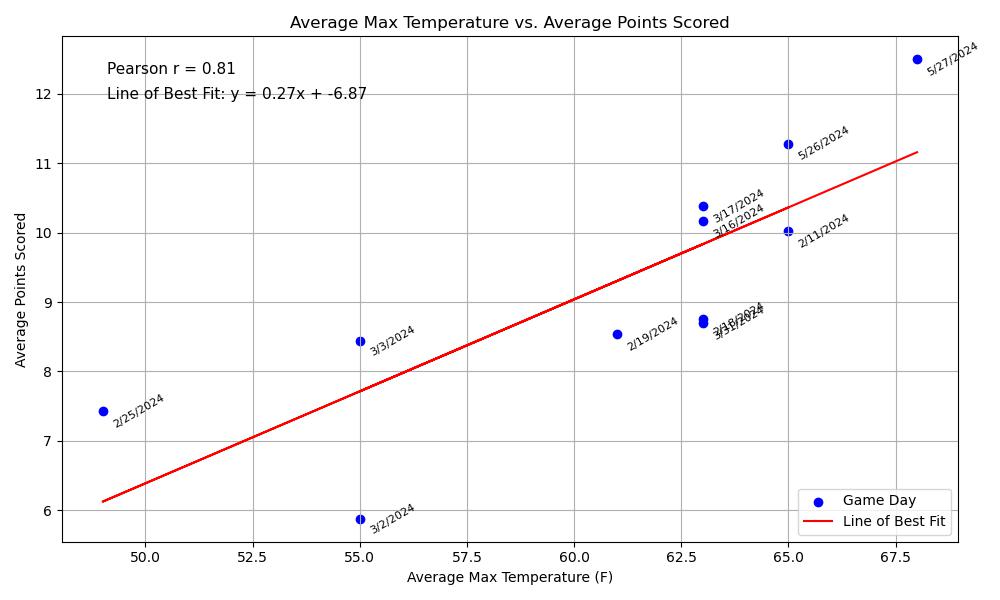
1. Average Wind Speed vs. Average Points Scored



1. Average Precipitation vs. Average Points



1. Average Max Temperature vs. Average Points Scored



**Instructions for Running Code**

1. Installations:
   1. Weather API: Open your terminal and type in the following two lines to install the weather API:
      1. pip install openmeteo-requests
      2. pip install requests-cache retry-requests numpy pandas
2. Running the Code:
   1. Database:
      1. Open the file “frisbee\_scraping.py” and run it.
      2. Each time you run the file, you will be asked if you want to wipe the database or not. Type “Yes/yes” if you want to clear the database and “No/no” if you want to keep filling up the database.
      3. Open the file “geocoding.py” and run it.
         1. You will have to run this 7 times in order to fully fill the geocoding table in the database.
   2. Calculations and Visualizations:
      1. Open the file “calculations.py” and run it.
         1. This should create a new CSV file called “calculations.csv” that contains the calculations that the visualizations will use.
      2. Open the file “visualizations.py” and run it.
      3. You will have to close the visualizations as they come up on your screen to see all of them.

**Function Documentation**

1. **read\_urls():** Reads event and schedule URLs from two text files and returns them as lists.
   1. Inputs: None
   2. Outputs: event\_urls (list): URLs from event\_pages.txt; schedule\_urls (list): URLs from schedule\_pages.txt
2. **setup\_database():** Sets up the SQLite database schema with four tables: locations, teams, dates, and games.
   1. Inputs: None
   2. Outputs: conn; cur
3. **wipe\_database(cur, conn):** Prompts the user to optionally wipe existing data from all tables and reset their auto-increment counters.
   1. Inputs: cur; conn
   2. Outputs: None
4. **insert\_location(cur, conn, city, state):** Inserts a city-state pair into the locations table if it doesn't already exist.
   1. Inputs: cur; conn; city (city name, str); state (state abbreviation, str)
   2. Outputs: location\_id (int): ID of the inserted or existing location
5. **insert\_team(cur, conn, team\_name):** Inserts a team name into the teams table if it doesn't already exist.
   1. Inputs: cur; conn; team\_name (team name, str)
   2. Outputs: team\_id (int): ID of the inserted or existing team
6. **insert\_game\_date(cur, conn, game\_date):** Inserts a game date into the dates table if it doesn't already exist.
   1. Inputs: cur; conn; game\_date (date in MM/DD/YYY format, str)
   2. Outputs: date\_id (int): ID of the inserted or existing date
7. **clean\_team\_name(name):** Cleans raw team names by removing seed numbers and discarding placeholder names like “W of Semifinal 1”.
   1. Inputs: name (raw team name, str)
   2. Outputs: cleaned\_name (cleaned team name or None if invalid, str)
8. **is\_valid\_score(score):** Checks if a score is valid (i.e., not a 0-0 game or a forfeit).
   1. Inputs: score (score string, str)
   2. Outputs: valid (bool)
9. **clean\_score(score):** Parses a score string and converts it to integer values for the winner and loser.
   1. Inputs: score (score string, str)
   2. Outputs: winner\_score (int); loser\_score (int); Returns (None, None) if parsing fails
10. **scrape\_event\_and\_schedule(event\_url, schedule\_url, cur, conn, max\_new\_games, new\_games\_count):** Scrapes location info and game results from two webpages, cleans the data, and inserts valid results into the database.
    1. Inputs: event\_url (URL for event metadata, str); schedule\_url (URL for game bracket, str); cur; conn; max\_new\_games (max number of games to add, int); new\_games\_count (games added so far, int)
    2. Outputs: new\_games\_count (updated count of new games, int): Updated count of new games; all\_games\_data (list of game summaries, list of dicts);
11. **print\_summary(cur, new\_games\_count):** Prints summary statistics about the total games in the database and how many were added during this run.
    1. Inputs: cur; new\_games\_count (games added this run, int)
    2. Outputs: None (prints to console)
12. **print\_games(all\_games\_data):** Prints a readable summary of each game added during the session.
    1. Inputs: all\_games\_data (game details, list of dicts)
    2. Outputs: None (prints to console)
13. **main():** Runs the full scraping and database update process.
    1. Inputs: None
    2. Outputs: None (calls other functions and prints output)
14. **create\_visualization(x, y)**: creates a visualization using the provided data for the x and y values.
    1. Inputs: x, y (data to be used for x and y values)
    2. Outputs: None (shows plot and saves plot)

**Resources Used**

1. You must also clearly document all resources you used. The documentation should be of the following form (20 points)

| Date | Issue Description | Location of Resource | Result (did it solve the issue?) |
| --- | --- | --- | --- |
| 4/10/25 | One location wasn’t returning any matches from the geocoding API | chatgpt.com | Yes, ChatGPT helped me realize that the location name was actually two different towns and the name had to be split up. |
| 4/11/25 | Couldn’t figure out how to get the table IDs to increment properly | chatgpt.com | Yes, ChatGPT helped me implement code to autoincrement and create unique IDs when new rows are added |
| 4/11/25 | Clarification of API parameters and variable ordering for the Weather API | chatgpt.com | Yes, confirmed correct usage |
| 4/12/25 | Weather data values were being stored as BLOBs values due to NumPy types | chatgpt.com | Yes, converted the values into integers using int() |