Katherine Kell and Maggie Scott

Van Lent

SI 206

22 April 2025

Connecting Event Attendance to Weather Conditions via Python

For this project, we had a lot of goals and expectations going into it, and the results were more challenging and different than expected—but we learned a lot in the process. Our initial plan was to use Ticketmaster's API to collect data about upcoming events in Detroit venues, a weather API to get temperature, precipitation, and weather conditions around said Detroit events, and EventBrite's API to cross-check our data from the Ticketmaster API to make sure the data was accurate, helping us improve the reliability of our analysis. However, what ended up happening is the Ticketmaster API wasn't cooperating, so we ended up using an API from a free website called PredictHQ. We also weren't able to cross-check our data with another API due to time constraints.

Some problems we encountered were that we were unsure how to limit the amount of data to a max of 25 items stored in the database each time a file is run. We figured this problem out by keeping track of how many rows were already in the database and then using that number as an offset when calling the API. That way, every time we ran the file, it would only add 25 new events instead of duplicating or overloading the database. We also used INSERT OR IGNORE statements to avoid duplicate entries and protect our tables from crashing. It took a lot of trial and error, especially since PredictHQ's documentation wasn't always super clear, but by the end, we had a functional database with multiple tables: including weather, events, addresses, and

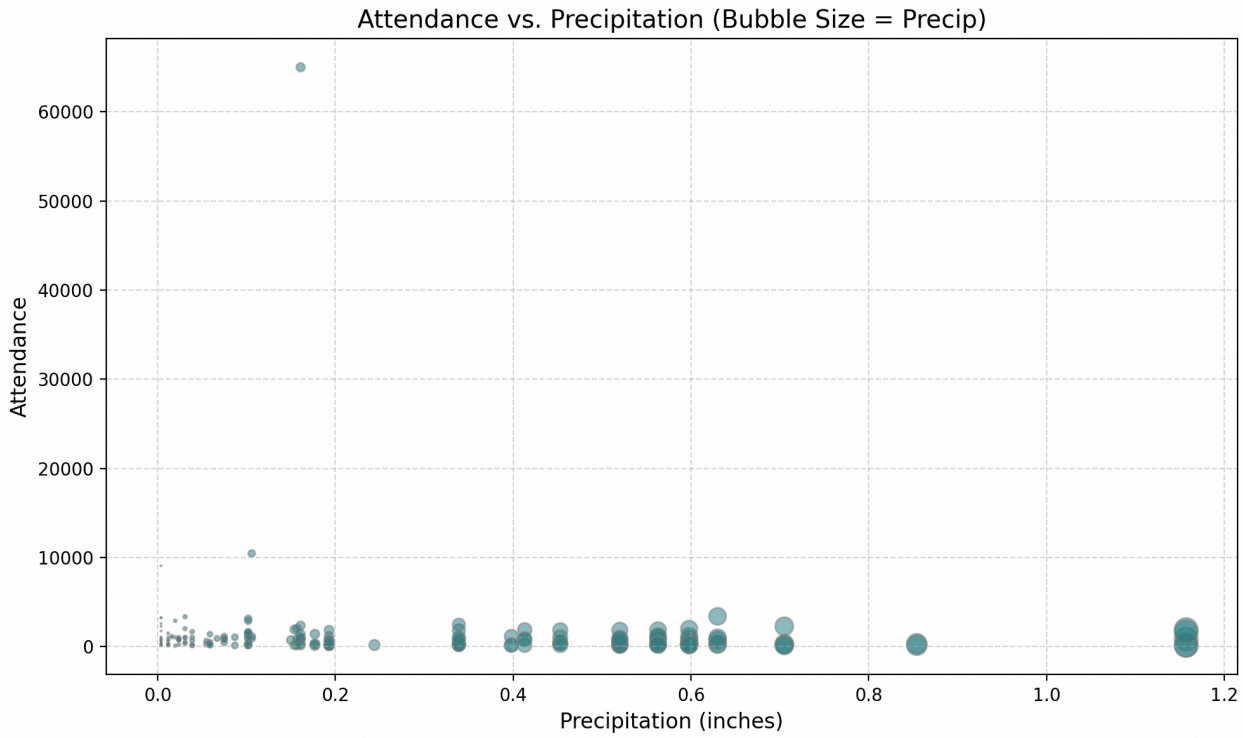
combined_events_weather. The reason why there is an addresses table is because since there are only so many concert venues in Detroit, we had a lot of duplicate string data in the table. We fixed this issue by creating an addresses table and assigning each unique venue location its own address_id. Then, instead of storing the full address string every time in the events table, we just referenced the address_id as a foreign key. This helped us save space and made joins across tables cleaner and more efficient. Overall, even though we had to pivot from our original plan, we ended up with a system that was much more thoughtfully structured and reflective of

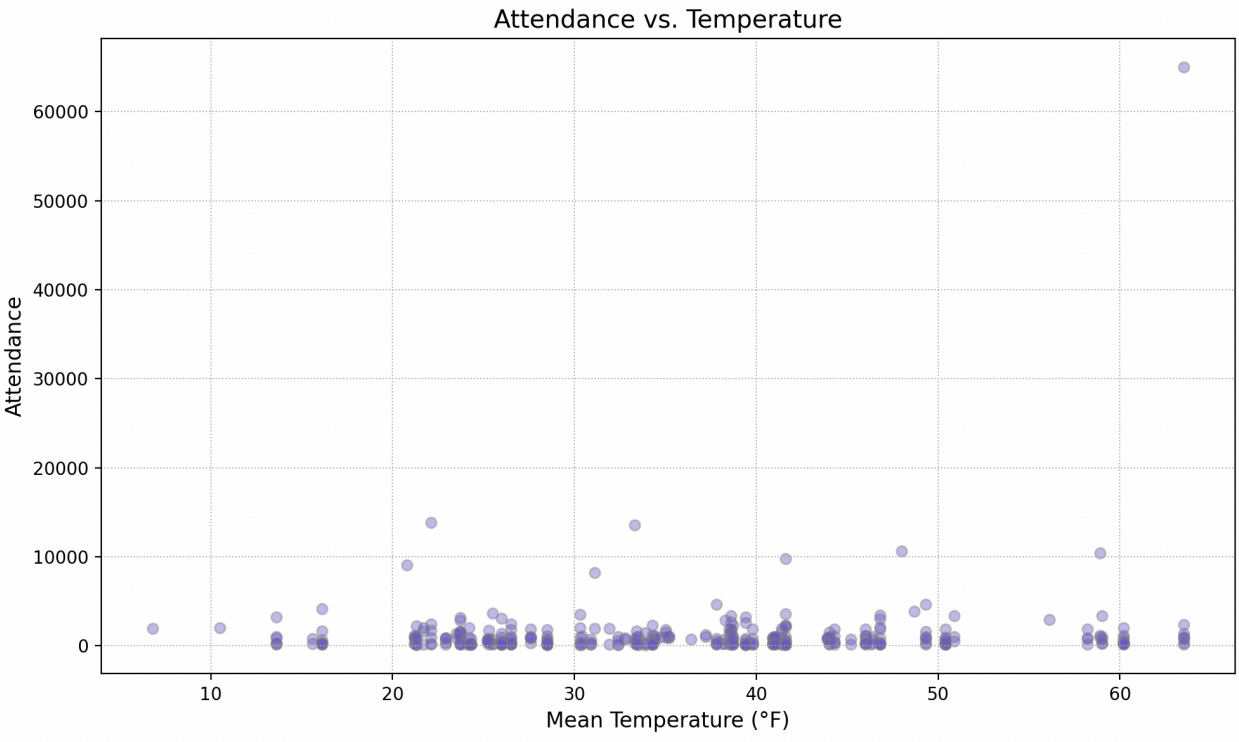
```
# Calculate averages [had chatgpt help me understand how to do this part]
categories = list(temp_bins.keys())
averages = [np.mean(temp_bins[cat]) if temp_bins[cat] else 0 for cat in categories]
```

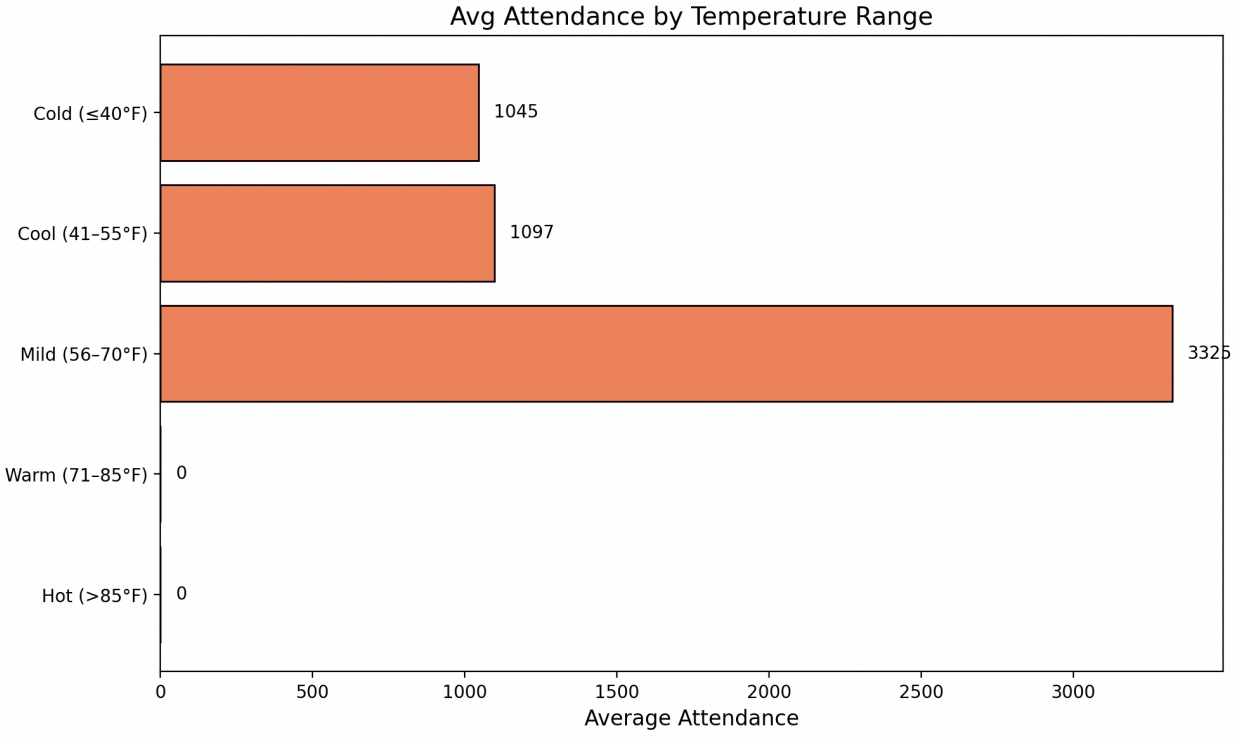
real-world data management practices.

After setting up and populating our database, we moved on to the data visualization and analysis part of our project. We wrote a separate Python file that pulled data from the combined_events_weather table and visualized potential correlations between weather conditions and attendance. We started with simple scatter plots—one comparing attendance to mean temperature and another comparing attendance to total precipitation, with each data point varying in size depending on the amount of precipitation. From there, we wanted to go a step further, so we created a horizontal bar chart that grouped average attendance by temperature ranges like Cold, Cool, Mild, Warm, and Hot to see if certain weather categories consistently impacted how many people showed up to events. Based on our final visualizations, we learned that there is no strong correlation between weather and concert attendance in Detroit-based venues.

To get these visualizations working, we used matplotlib and numpy. ChatGPT helped us figure out how to calculate averages for each temperature range and how to group temperatures into readable bins. Once the graphs were generated, we exported them as PNG files so they could be referenced in our report or used in a presentation. The visualizations are stored in the local directory as styled_attendance_vs_temp.png, bubble_attendance_vs_precip.png, and horizontal_bar_avg_attendance.png.




Attendance vs. Temperature
Attendance
60000
50000
40000
30000
20000
10000
0
10
20
30
40
50
60
Mean Temperature (°F)


Avg Attendance by Temperature Range
Cold (≤40°F)
1045
Cool (41–55°F)
1097
Mild (56–70°F)
3325
Warm (71–85°F)
0
Hot (>85°F)
0
0
500
1000
1500
2000
2500
3000
Average Attendance

In addition to creating the charts, we also wrote documentation for all of our functions, so it's clear what each one does, what inputs it takes, and what output it returns. We also made sure our code could be run by anyone by organizing it into two files: one for setting up and populating the database (main.py), and one for generating visualizations (datavis.py). We included comments, example outputs, and printed statements to help walk the user through what each part is doing.

To run the project successfully, users should first install the necessary Python libraries: requests, matplotlib, and numpy. These can be installed using pip. Running main.py will create and populate the SQLite database (final.db) by fetching event data from the PredictHQ API and corresponding weather data from the Open-Meteo API. This script automatically handles duplicates, table structure, and foreign key constraints. After the data is inserted and the database is complete, running datavis.py will generate the visualizations and save them as .png files in the same directory.

Throughout this project, we used several tools and resources to build and refine our work. We relied on the PredictHQ API to retrieve information about upcoming events in Detroit and the Open-Meteo API for historical weather data, including temperature and precipitation. Our project was developed in Python using the libraries sqlite3 for database handling, requests for API calls, matplotlib for data visualization, and numpy for numerical calculations and logic. We used DB Browser for SQLite to inspect and test database content and relationships. Additionally, we used ChatGPT to help debug our code, structure our logic, and assist with syntax and data visualization strategies. We also referred to API documentation from PredictHQ and Open-Meteo, as well as Stack Overflow and other online resources when solving errors or adjusting formatting.

Date	Issue Description	Location of Resource	Result (did it solve the issue?)
04-06-25	Understanding API parameters	PredictHQ API Documentation	The documentation was crucial for understanding how to narrow down the scope of the data. The API documentation explained how to narrow down both the location and dates.
04-18-25	Understanding weather parameters	Open-Meteo API Documentation	Like the events API, the weather API documentation explained the differences between each of the different classifiers of data. Figuring out how to access historical data was key to our project, as we needed historical weather in order to analyze the correlation with attendance.
04-19-25	Implementing functions	Python libraries: requests, sqlite3, matplotlib, numpy, csv	Over the course of the project, we implemented different Python libraries to run all of the necessary functions such as interpreting APIs, creating visualizations, creating databases, and performing calculations.
04-19-25	Debugging	ChatGPT	Generative AI was extremely helpful when finalizing our code and debugging. Applying various advanced concepts in one file was challenging, and ChatGPT helped clean up syntax errors as well as providing general logical outlines for different tasks.
04-21-25	Debugging	Stack Overflow	Paired with ChatGPT, online forums like Stack Overflow helped with the debugging process. In our case, we

			turned to this source to reference the functions necessary to create our calculations, as this was a newer concept for us

In our main function, we created seven functions. The first was “setup_database”. The function takes in the parameter “db_name”, which in this case is “combined_data1.db”. This function connects to an sqlite database and creates 4 tables: events, weather, addresses, and combined_events_weather. The addresses table creates address IDs to eliminate the duplicate address strings in the events table. Next, the function “fetch_events” takes in parameters that define the limit as 25 and the offset as 0 in order to limit the data intake to 25 rows at a time. This function creates a json object from the PredictHQ events API using events in downtown Detroit from January 1, 2025 to April 16, 2025. The return object is a list of event data that includes the event ID, date, title, location, and attendance. Next, the function “insert_event_data” takes in this event data as well as the sqlite database as parameters in order to insert the parsed data into the events table in the combined_data1.db database. A similar process is repeated with the weather API in the next two functions. The “fetch_weather” function takes no parameters, but it creates a json object from the weather API in order to return a list of weather data including dates, temperature means, precipitation sums, and apparent temperature means. The next function, “insert_weather_data” takes in this weather data as well as the sqlite database and inserts the data into the weather table. Next, the “combine_data” function takes in the sqlite database as a parameter and joins the weather and events tables into one combined_events_weather table, joined by date. Finally, the main function calls each of these functions. The main function helps limit each run to 25 new rows at a time by tracking the

number of current entries and adding 25 to the offset with each run. By adding in print statements, the main function helps the user track the counts in each table run by run.

In our other python file, “datavis.py”, we created four functions. The first function, “calculate_averages”, takes no parameters, but it connects to a sqlite database, and selects data from the events and weather tables using JOIN. It then defines the variables, attendance, temp_mean, and precipitation using data from these tables, and returns these three variables as a tuple. The next function is “create_visualizations”. This takes in the attendance, temp_mean, and precipitation variables from the last function as parameters, and it uses these to create three visualizations: a scatter plot for attendance v. temperature, a bubble plot for precipitation v. temperature, and a horizontal bar graph for average attendance by temperature range. The categories and averages for this final graph were calculated and returned. Our next function, “write_csv” writes these calculations into a csv file. Taking in the categories and averages from the last function as well as a new file name, the function writes a csv file with the headers “Temperature Range” and “Average Attendance”, filling in rows with the corresponding data. Finally, the main function in this file calls the previous three functions.

This project pushed us to think more critically about how real-world data is messy and formatted in inconsistent ways, and how important it is to build systems that can account for that. It also taught us how to problem-solve when things don’t go as planned—like when an API breaks or when you get duplicate data crashing your tables—and helped us learn how to adapt quickly and creatively.