US Tornadoes 1950-2021 - Technical Writeup

Overview

This technical write-up serves as a comprehensive guide to the development, deployment, and impact of our "US Tornadoes 1950-2021" project.

The "US Tornadoes 1950-2021" project aims to analyze trends and patterns in tornado occurrences across the United States. The dataset contains 67,558 entries, providing details on magnitude, injuries, fatalities, and location data. To ensure data integrity and reliability, we performed data cleaning and transformation before utilizing it for visualization and analysis.

Data Cleaning and Transformation

Data Preparation:

The dataset was first loaded into a Pandas DataFrame (`data`), allowing for initial data cleaning and preprocessing. This step ensured data consistency and helped handle any missing or erroneous values before storage.

Issues Identified:

1. Inconsistent date data types
2. Duplicate records
3. Invalid tornado magnitudes (-9)
4. Unclear column names

Here is a screenshot of the original dataset (source):

	yr	mo	dy	date	st	mag	inj	fat	slat	slon	elat	elon	len	wid
0	1950	1	3	1950-01-03	IL	3	3	0	39.10	-89.30	39.12	-89.23	3.6	130
1	1950	1	3	1950-01-03	MO	3	3	0	38.77	-90.22	38.83	-90.03	9.5	150
2	1950	1	3	1950-01-03	OH	1	1	0	40.88	-84.58	0.00	0.00	0.1	10
3	1950	1	13	1950-01-13	AR	3	1	1	34.40	-94.37	0.00	0.00	0.6	17
4	1950	1	25	1950-01-25	IL	2	0	0	41.17	-87.33	0.00	0.00	0.1	100

Changed the Date Datatype:

```
data_unclean["date"] = pd.to_datetime(data_unclean["date"])
```

Removed Duplicates:

```
duplicates_to_remove = data_unclean[data_unclean.duplicated(keep='first')]

duplicates_to_remove.info()
```

```
data_unclean = data_unclean.drop_duplicates(keep='first')

data_unclean.info()
```

Removed Invalid Magnitudes:

```
magnitudes_to_remove = data_unclean[data_unclean["mag"] == -9]

magnitudes_to_remove.info()
```

Renamed Columns:

```
data_unclean = data_unclean.rename(columns={
    "yr": "year",
    "mo": "month",
    "dy": "day",
    "st": "state",
    "mag": "tornado_magnitude",
    "inj": "injuries",
    "fat": "fatalities",
    "slat": "start_latitude",
    "slon": "start_longitude",
    "elat": "end_latitude",
    "elon": "end_longitude",
    "len": "tornado_length",
    "wid": "tornado_width"
})

data_unclean.head()
```

After these cleaning steps, we had a refined dataset to load into sqlite database. Here is a sample of the clean dataset.

	year	month	day	date	state	tornado_magnitude	injuries	fatalities	start_latitude	start_longitude	end_latitude	end_longitude
0	1950	1	3	1950-01-03	IL	3	3	0	39.10	-89.30	39.12	-89.23
1	1950	1	3	1950-01-03	MO	3	3	0	38.77	-90.22	38.83	-90.03
2	1950	1	3	1950-01-03	OH	1	1	0	40.88	-84.58	0.00	0.00
3	1950	1	13	1950-01-13	AR	3	1	1	34.40	-94.37	0.00	0.00
4	1950	1	25	1950-01-25	IL	2	0	0	41.17	-87.33	0.00	0.00

Database Design, Data Loading and Database Integration

In our "US Tornadoes 1950-2021" project, we utilized SQLite as our relational database management system to efficiently store and manage the extensive dataset. SQLite is lightweight in nature and seamless integration with Python made it an ideal choice for our data storage needs.

Database Connection:

Using **SQLAlchemy**,We established a connection to an SQLite database (`database.db`). SQLAlchemy allows seamless interaction between Python code and the SQL database.

Data Insertion:

The prepared DataFrame was then inserted into the database using the `to_sql` method:

```
data.to_sql("tornado", con=engine, if_exists="replace", index=False)
```

1. The `if_exists="replace"` parameter ensured that any existing table named **tornado** would be overwritten, maintaining an up-to-date dataset with each run.
2. **Verification of Data Load:**
   To verify successful data insertion, we employed **SQLAlchemy's** inspection tools. By querying the **tornado** table and printing the first few rows, we confirmed that the data was correctly stored, with the appropriate columns and data types.

```python
# Create the connection engine
engine = create_engine("sqlite:///data/database.db")
```

```python
# Create the inspector and connect it to the engine
inspector = inspect(engine)

# Collect the names of tables within the database
tables = inspector.get_table_names()

# Using the inspector to print the column names within the 'dow' table and its types
for table in tables:
    print(table)
    print("--------")
    columns = inspector.get_columns(table)
    for column in columns:
        print(column["name"], column["type"])

    print()
```

```
tornado
--------
year INTEGER
month INTEGER
day INTEGER
date TEXT
state TEXT
tornado_magnitude INTEGER
injuries INTEGER
fatalities INTEGER
start_latitude REAL
start_longitude REAL
end_latitude REAL
end_longitude REAL
tornado_length REAL
tornado_width INTEGER
```

Executing SQL Queries:

To gain insights from the data, we used raw SQL queries with the help of **SQLAlchemy's** `text()` function:

Sample query is executed to check whether the database is running fine and giving correct results.

```
query = text("""
                SELECT
                    year,
                    sum(fatalities) AS 'Total_Fatalities'
                FROM
                    tornado
                GROUP BY
                    year
                """)
df = pd.read_sql(query, con=conn)
df
```

Why did we use a Database?

- **Scalability:** Easier to handle large datasets compared to in-memory storage.
- **Query Flexibility:** SQL provides powerful tools for data aggregation, filtering, and joining.
- **Data Integrity:** The database ensures data is stored in a structured and reliable manner.

Integration with Web Application and backend setup with Flask

Our project is deployed as an interactive web application, enabling users to explore tornado data dynamically. We utilized Flask, a lightweight web framework for Python, to handle HTTP requests and serve the application. The integration between Flask and SQLite was achieved using the `sqlite3` module in Python, allowing seamless interaction between the web interface and the database.

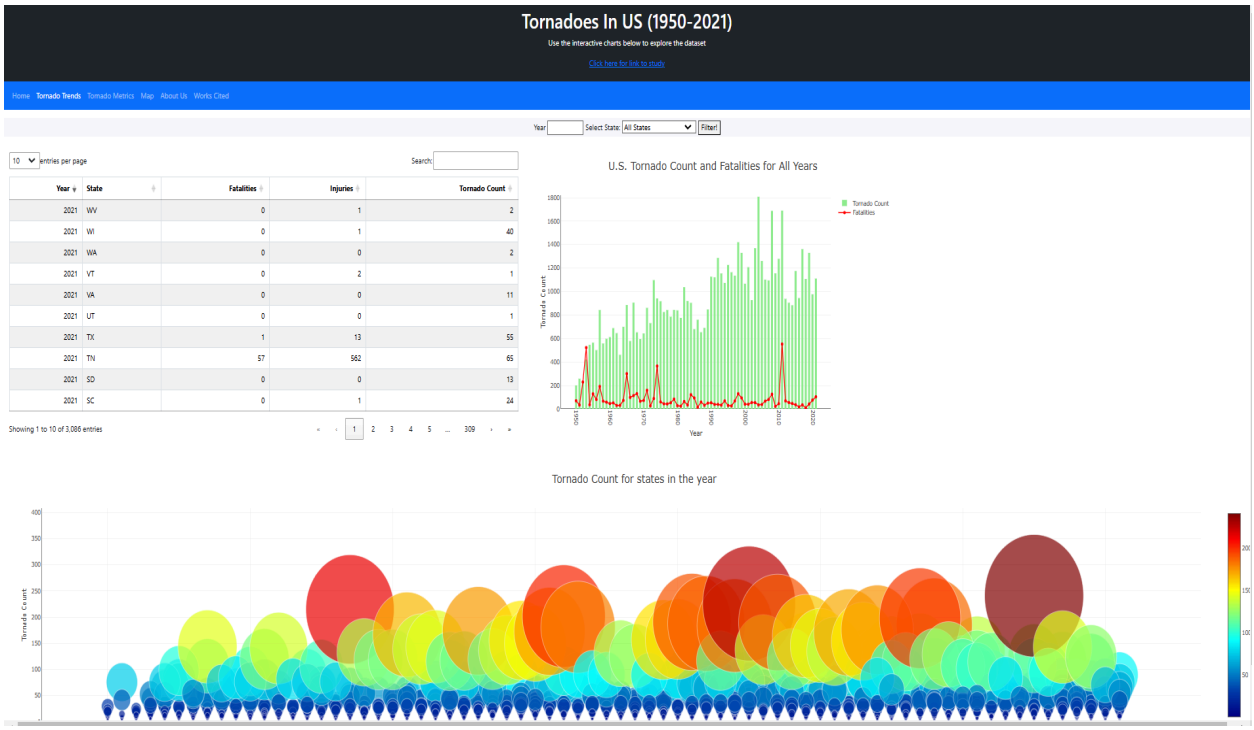
To ensure efficient data retrieval and display, we employed server-side caching mechanisms for frequently accessed queries, such as the leaderboard. This approach minimizes database load and enhances the user experience by reducing page load times.

- **Routing:** The Flask app defines routes for different pages (`home`, `dashboard1`, `map`, etc.) and API endpoints (`/api/v1.0/table`, `/api/v1.0/linechart_data`, etc.).
- **Data Handling:** Using a custom **SQLHelper** class, I executed SQL queries to fetch data for visualizations. The queried data was converted to JSON format using Pandas' `to_dict` method, ensuring smooth integration with frontend charts.
- **API Endpoints:** The API provides data for various visualizations, including line charts, bubble charts, histograms, and interactive maps.
- **Caching Control:** Implemented caching headers to ensure the latest data and visualizations are always displayed.

Frontend Functionality and Data Visualization

The **JavaScript** code utilizes **D3.js** and **Plotly** libraries to create an interactive and dynamic dashboard:

- **Data Loading and Filtering:** When the page loads or the filter button is clicked, the `handleDataUpdate` function fetches data from multiple API endpoints (`/api/v1.0/table`, `/api/v1.0/linechart_data`, `/api/v1.0/bubblechart_data etc.`) and applies user-selected filters (e.g., year, state).
- **Dynamic Table and Charts:** Filtered data is displayed in a dynamic **DataTable**, and visualizations are updated using **Plotly**:
- **Interactivity:** The dashboard automatically updates when new filters are applied, ensuring a smooth and engaging user experience.
- **Dashboard1:**
    - **Line Chart:** Displays tornado counts and fatalities over time, using both bar and line traces.
    - **Bubble Chart:** Visualizes tornado counts with bubble size and color, providing insights into the frequency of tornadoes across different years.

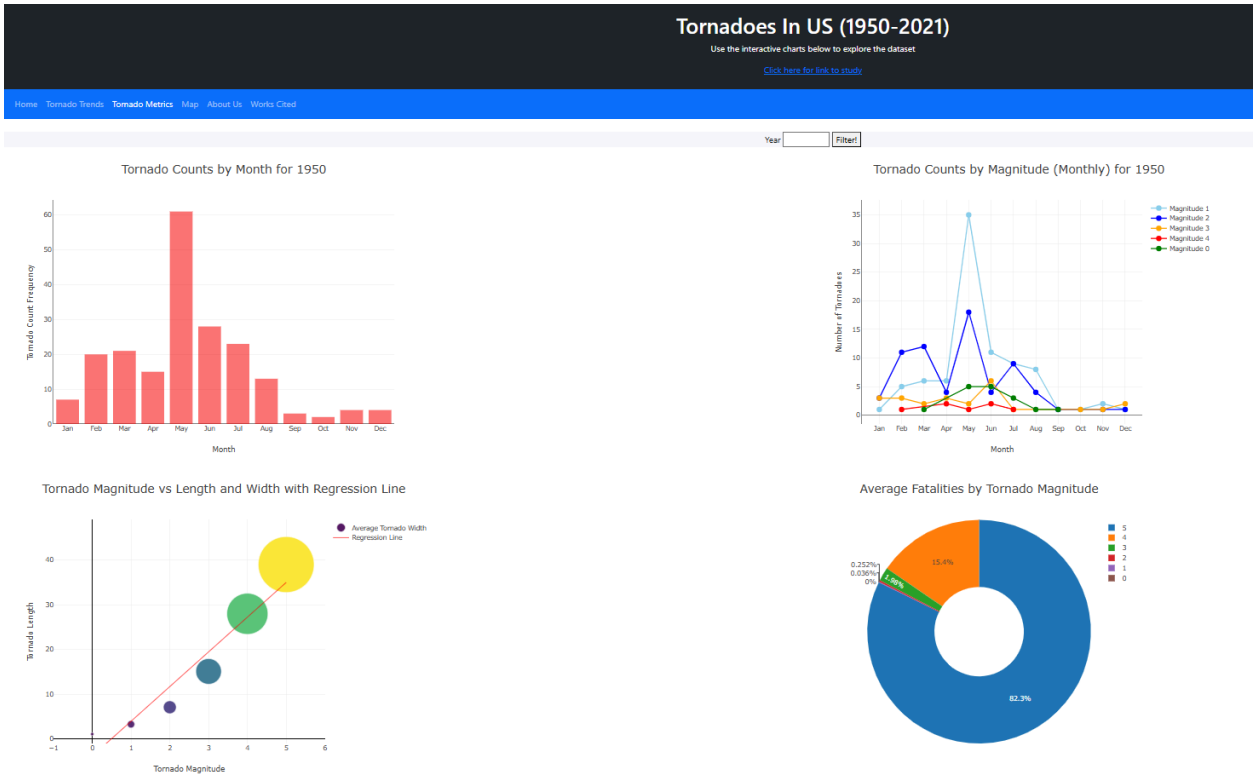


Dashboard2:

- **Histogram:** Visualizes tornado counts by month, showing frequency distribution for a selected year.
- **Time Series Chart:** Displays monthly tornado counts by magnitude, with colored lines for each magnitude.
- **Scatter Chart:** Shows tornado magnitude vs. average tornado length and width, with a regression line for trends.
- **Pie Chart:** Represents average fatalities by tornado magnitude with a donut style.

Additional Features:

- **Data Filtering by Year:** `filterDataByYear()` is used to dynamically update the visualizations.
- **Regression Line Calculation:** The `computeRegression()` function adds a trendline to the scatter plot.
- **Color Management:** `getColorForMagnitude()` function assigns colors based on tornado magnitudes.

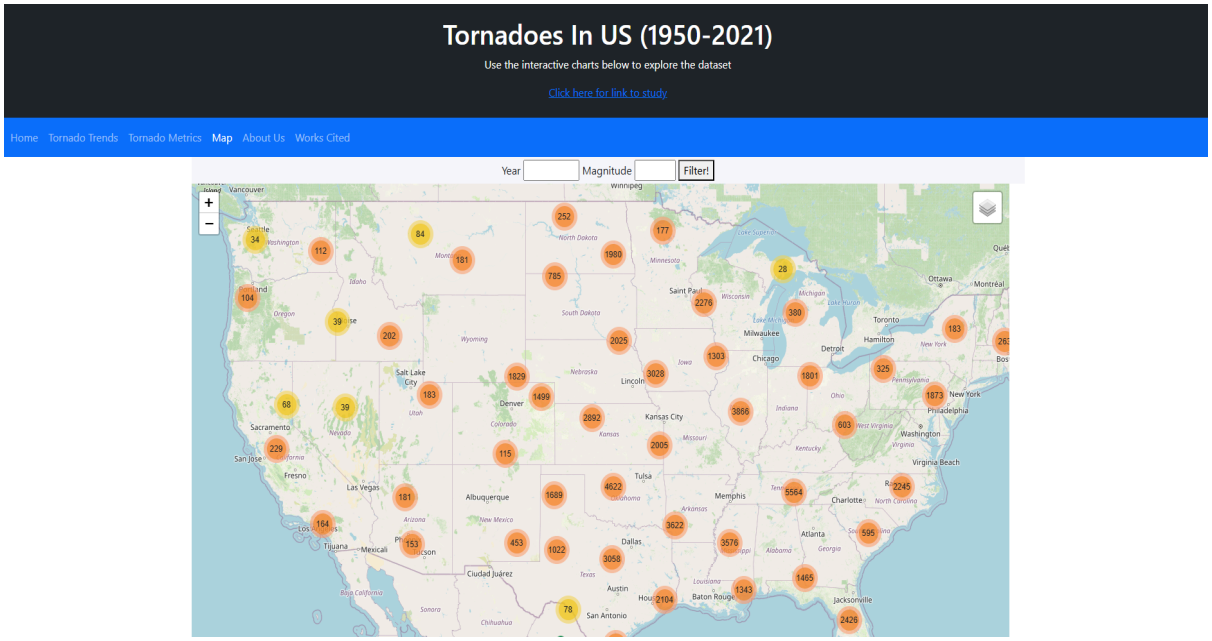


Map Dashboard:

The provided code sets up an interactive map visualization of tornado data using **Leaflet.js**, incorporating both **marker clusters** and **heatmap layers** for an enhanced user experience.

Key Features:

- **Dynamic Filtering:** Users can filter tornado data by **year** and **magnitude** using input fields and a filter button. This triggers the `handleMapUpdate` function, which fetches and filters data before updating the map.
- **Marker Coloring:** The `chooseColor` function maps tornado magnitudes to colors based on the **Enhanced Fujita (EF) Scale**, improving visual clarity of tornado severity.
- **Interactive Map Elements:** The map includes **popup information** for each tornado marker, showing relevant details like latitude, longitude, year, magnitude, and state.
- **Map Layers and Controls:** Supports **street view** and **topography view** via **OpenStreetMap** and **OpenTopoMap** tiles, with a control panel to toggle between **heatmap** and **marker clusters**.
- **Automatic Initialization:** On page load, the map is populated with default data.
- **Clustered Markers:** Enhances performance and presentation by grouping nearby tornado data points.
- **Heatmap Visualization:** Shows tornado density, highlighting regions with higher tornado activity.



Key Takeaway: Interactive Experience

- The dashboard's interactivity empowers users to **explore patterns**, **identify trends**, and **gain insights** based on their particular areas of interest.
- The ability to filter by **year** and **state** makes it an effective tool for researchers, emergency planners, and the general public to **analyze tornado impacts** and **prepare for future events**.

- **Explore the dashboards:** Adjust filters, analyze trends, and gain valuable insights that can contribute to saving lives and improving safety strategies!

Few Research questions answered through the dashboard:

1. **Geographical Distribution:** Where do tornadoes most frequently occur in the U.S.?
2. **Regional Impact:** Which states experience the highest fatalities and injuries due to tornadoes?
3. **Tornado Hotspots:** Are there specific regions that consistently experience high tornado counts over the years?
4. **State-Level Trends:** How do tornado occurrences vary by state when filtered by specific years or tornado magnitudes?
5. **Mapping Severity:** Can the map visually depict the severity of tornadoes (e.g., using color or size to indicate magnitude or fatality rates)?

Libraries and Technologies Used

1. Back-End Libraries:

- **Flask:** Lightweight web framework for building the backend and API endpoints.
- **Pandas:** For data manipulation, cleaning, and processing of tornado datasets.
- **SQLAlchemy:** Object-Relational Mapping (ORM) tool for database management and queries.
- **SQLite:** Lightweight database used to store and manage tornado data.
- **Numpy:** Supports numerical operations and data handling.
- **Gunicorn:** WSGI HTTP server for deploying the Flask application in a production environment.

2. Front-End Technologies/ Libraries:

- **HTML/CSS:** Provides the structure and styling for the dashboard layout.
- **JavaScript:** Powers interactivity, data processing, and chart rendering.
- **D3.js (Data-Driven Documents)**
  - **Purpose:** Data fetching, manipulation, and dynamic DOM updates.
  - **Usage in Dashboard:**
    - Fetching data via API calls (`d3.json()`).
    - Handling UI interactions, such as button clicks (`d3.select("#filter-btn").on("click", ...)`).
    - Managing input values (`d3.select("#year").property("value")`).
- **Plotly.js**

Ethical Considerations

- Data privacy and ethical handling of sensitive information were prioritized.
- Bias and misinterpretation risks were assessed to ensure fair representation of findings.

Conclusion

- The Tornado Data Visualization Project seamlessly integrates a robust **Flask** backend with a dynamic **D3.js** and **Plotly.js** frontend to deliver an interactive and insightful dashboard.
- The backend efficiently handles data processing and API management, while the frontend transforms raw tornado data into compelling visualizations.
- This holistic approach ensures a smooth user experience, enabling deep exploration of tornado trends and impacts.
- Future enhancements could include real-time data integration, predictive analytics, and advanced filtering options to further enrich the analytical potential of the dashboard.

Limitations and Bias:

- **Data Limitations:** The accuracy of visualizations depends on the completeness and quality of the tornado dataset. Missing or inconsistent data could affect insights.
- **Temporal Bias:** Older tornado records might be less accurate due to limited technology and inconsistent reporting practices in earlier decades.
- **Geographical Bias:** Some regions may have more comprehensive data collection, potentially skewing analysis toward areas with better reporting infrastructure.