Project Report: Twitter Dataset Embedding, Clustering and NL2SQL Tools

This project focuses on analyzing Twitter/X dataset provided by USC's HUMANS Lab. At the heart of asking questions and arriving at meaningful answers is a foundation of clean, well-organized data that can be accessed easily. By cleaning the dataset, storing it in a structured database, and making it quickly searchable, I enabled deeper explorations and insights into the data. Below is an overview of the main accomplishments, technical components, and how they work together.

---

1. Overview of the Project

The goal was to develop an end-to-end system for:

1. Organizing and exploring Twitter data in a structured way.
2. Generating text embeddings to enable semantic exploration.
3. Hierarchical Clustering these embeddings to discover topics or themes.
4. Performing dimensionality reduction and visualization in 3D.
5. Summarizing each discovered topic with an on-device Large Language Model (LLM).
6. Providing an NL2SQL (Natural Language to SQL) interface allows users to query the data in plain English.

By combining these techniques, I created an environment where anyone can gain quick insights into the dataset, explore emerging themes, and refine their research questions.

---

2. Data Preparation and Cleaning

2.1 Data Issues

The full report on statistics related to the raw and processed data can be found in the analysis notebook on GitHub.

Multiple CSV Partitions

The dataset is divided into multiple directories (e.g., `part_x`) each containing multiple gzipped CSV files. These files contain slightly different schemas or column orders, leading to difficulties when reading them consistently. Please visit Analysis notebook for details of the issue.

Python Objects Embedded in CSV

Some CSV columns included serialized Python objects or dictionaries, for instance:

- Python `datetime` objects such as `datetime.datetime(2024, 7, 5, tzinfo=datetime.timezone.utc)`.
- JSON-like structures with keys such as `{'count': '14', 'state': 'EnabledWithCount'}`.
- Strings storing nested user objects (e.g., location, username, etc.) but in inconsistent formats (some columns have partial JSON, and some have raw Python dictionary syntax).

Column Discrepancies

- The CSV chunks had mismatched columns, inconsistent naming, and, in some cases, missing columns.
- Some columns contained repeated or misplaced data (e.g., `epoch` vs. `timestamp` vs. Python datetimes).
- For user-related data, relevant fields (location, bio, username, followers count) were buried in the user column's JSON-like structure.

2.2 Data Cleaning Methods

To address these issues, I created a Python script that systematically:

1. **Uses chunk-based reading** to handle large CSV files in smaller segments (with a configurable chunk size, e.g., 50,000 rows). This reduces memory overhead and also makes error handling more granular.
2. **Replaces Python `datetime.datetime(...)`** patterns with ISO8601 strings. A custom regex-based function—`replace_python_datetimes_with_strings`—searches for any embedded Python datetime calls and replaces them with a valid string representation (e.g., `2024-07-05T12:34:56+00:00`).
3. **Converts textual boolean-like values** (`"true"`/`"false"`) into proper Python booleans.
4. **Extracts view counts** from nested dictionaries (e.g., `{"count": "14", "state": "EnabledWithCount"}`), storing them as integers.
5. **Safely interprets user metadata** by:
    - Parsing a JSON-like column (`user`) if possible.
    - Extracting `location`, `username`, `rawDescription` (bio), `followersCount`, `verified`.
    - Storing them in consistent columns in the output table (`location`, `bio`, `username`, etc.).

6. **Handles numeric conversions** (e.g., retweet count, reply count, like count) so that they become integers rather than strings.
7. **Preserves other fields** such as `lang`, `conversationId`, `hashtags`, etc., storing them as strings to maintain data integrity.
8. **Implements robust exception handling and retry logic**:
    - If a parse error occurs in a chunk, the code retries up to three times.
    - If columns are missing or data is malformed, the code logs the issues but continues with other rows when possible.

By cleaning the data in a controlled step-by-step process, I create a consistent schema suitable for querying and analysis.

---

3. Data Ingestion with DuckDB

I decided to use DuckDB for its ease of use, speed, and seamless integration into python apps.

3.1 Schema Design

After cleaning, each tweet is mapped to a stable schema in DuckDB:

```
CREATE TABLE IF NOT EXISTS tweets (
id VARCHAR,
text VARCHAR,
url VARCHAR,
timestamp TIMESTAMP,
media VARCHAR,
retweetedTweet BOOLEAN,
retweetedTweetID VARCHAR,
retweetedUserID VARCHAR,
lang VARCHAR,
replyCount BIGINT,
retweetCount BIGINT,
likeCount BIGINT,
quoteCount BIGINT,
conversationId VARCHAR,
conversationIdStr VARCHAR,
hashtags VARCHAR,
mentionedUsers VARCHAR,
links VARCHAR,
viewCount BIGINT,
quotedTweet BOOLEAN,
```

```
location VARCHAR,
cash_app_handle VARCHAR,
username VARCHAR,
bio VARCHAR,
followersCount BIGINT,
verified BOOLEAN );
```

3.2 Chunked Insertion

As the script reads each GZ-compressed CSV in chunks, it appends the cleaned rows to the DuckDB table via:

- `con.append(TABLE_NAME, insert_df)` within a transaction.
- Commits are performed after each chunk to maintain data integrity and allow partial results if a failure occurs mid-file.

This approach keeps memory usage reasonable and allows for incremental ingestion if any file fails.

3.3 Result

Once the entire set of CSV files is processed, we have a single DuckDB database (`tweets.duckdb`) containing a uniform, well-structured `tweets` table with consistent columns and types. This addresses the primary challenge of messy data distribution across multiple CSV files. The zipped database can be downloaded via here.

---

4. Analytic Pipeline: Embeddings, Clustering, and Summaries

This clustering work is inspired by "Clustering Tweets via Tweet Embeddings" by Daniel X. Sun.

4.1 End-to-End Workflow

The analytic pipeline is designed to transform raw textual tweets into **actionable insights** by:

1. **Loading & Cleaning** data from DuckDB into a Pandas DataFrame.
2. **Generating embeddings** that numerically represent the semantic content of each tweet.
3. **Hierarchical Clustering** the resulting embeddings into topical groups.
4. **Reducing the dimensionality** of the embedding space to visualize clusters in 3D.
5. **Summarizing** each group with a local Large Language Model (LLM).

The motivations for these steps are as follows:

- **Embedding** text captures semantic meaning in a numeric vector, making it easier to group, query, and compare tweets by topic or theme.
- **Clustering** helps discover hidden patterns or conversation topics that might otherwise be lost in a large unstructured dataset.
- **Dimensionality Reduction** (e.g., t-SNE or PCA) is crucial to reveal the structure of the data in a 2D/3D visualization—uncovering meaningful relationships that would be invisible in a high-dimensional embedding space.
- **Local LLM-based Summaries** provide human-readable "topic labels" for each cluster, turning large volumes of text into concise and interpretable categories.

Bringing these capabilities together provides an **interactive research environment** in which anyone—technical or not—can quickly glean insights about trending themes, major discussion points, or user behaviors in the underlying Twitter data.

4.2 Technical Foundations and Code Structure

Below is an outline of the key technical components, taken from the referenced Python script:

1. **Streamlit UI**
    - A user interface built in Streamlit guides the entire process, allowing researchers to select:
        - How many rows to load from the DuckDB table,
        - Which text column to embed (e.g., `text`, `bio`, or any other textual field),
        - Which embedding model to use from the Ollama local LLM server,
        - And the parameters for clustering (number of clusters, hierarchy levels) and dimensionality reduction.
    - This approach lowers the barrier to entry—no specialized command-line knowledge is required; researchers can explore the pipeline with easy sliders and drop-down menus.
2. **DuckDB Integration**
    - The pipeline begins by **loading cleaned data** from DuckDB using a simple `SELECT * FROM <table> LIMIT <N>` query.
    - The result is placed into a Pandas DataFrame, ensuring we can handle the data using Python's scientific libraries (NumPy, scikit-learn, etc.).
    - This step is critical: it gives a consistent, structured way to pull large datasets into memory for analysis, while still respecting memory constraints (because we can limit the loaded row count).
3. **Embedding Generation with Ollama**
    - Text embedding is performed locally via an Ollama server. Each tweet is passed to the `embeddings` endpoint:

```
response = requests.post( f'{ollama_url}/api/embeddings', json={'model': model, 'prompt': text_str} )
```

- Once returned, the embeddings are stored in a NumPy array.
- **Caching** is implemented via `.npz` files on disk (e.g., `embeddings_cache`). This ensures that if a user re-runs the pipeline with the same text data and model, the script can skip the (often time-consuming) re-embedding step.
- *Importance:* Embeddings capture semantic relationships between tweets. Tweets about the same topic or event typically cluster in the embedding space. By generating these numerical vectors, we enable downstream machine learning tasks like clustering and dimensionality reduction.

2. **K-Means Clustering and Hierarchical Grouping**
   - The code uses a top-down hierarchical approach, driven by repeated K-Means clustering:
     - We choose `K` (e.g., 3, 5, or 10) and a maximum depth (the `num_levels` slider).
     - At each level, we split the data into `K` clusters. We then recursively cluster each subgroup until we reach the specified depth (or run out of data).
   - This yields a tree-like structure of clusters, which can highlight both **broad** conversation themes (the top-level splits) and **more granular** sub-topics in deeper nodes.
   - *Importance:* Instead of a single flat clustering, we get a **multi-level topic hierarchy**, which can help isolate and drill down into finer detail. In large data sets, high-level clusters might be too broad (e.g., “politics”), whereas deep clusters give more nuanced subtopics (e.g., “discussion of gun control legislation in the Midwest”).
3. **LLM-based Summaries & Topic Naming**
   - Each cluster is passed to a local LLM model (also via Ollama) with a **prompt** that includes sample tweets. For leaf clusters, the LLM is asked to produce a succinct descriptive label; for parent clusters, the LLM uses summary info from child clusters.
   - The process is done in a **post-order** traversal: children are named before their parent so that the parent node summary can factor in child cluster names.
   - *Importance:* Rather than requiring the user to manually read hundreds or thousands of tweets to label a cluster, an LLM automatically provides a short descriptor. This significantly **reduces time** and helps non-experts in quickly understanding what the cluster is about (e.g., “Debates About Gun Policy” vs. “Energy Infrastructure Updates”).
4. **Dimensionality Reduction for 3D Visualization**
   - After clustering, the code can optionally run either **t-SNE** or **PCA** to project the embeddings down to 3D.
   - t-SNE is good for capturing local neighborhood structures but can be slow or sometimes produce “artifacts.” PCA is faster and more straightforward but may

lose some local cluster structure if the main variance in the data is overshadowed.
    - *Importance:* High-dimensional embeddings are impossible to interpret visually. By plotting them in 3D, we can see how the tweets form distinct groups and identify outliers or overlapping topics. This is a **key step** in letting researchers quickly sense-check whether the cluster boundaries make intuitive sense.
5. **3D Scatter Plot (Plotly)**
    - Each tweet is represented as a 3D point, colored by its assigned cluster.
    - Interactive hover tooltips show the tweet text, user metadata, and other relevant fields—allowing a deeper inspection of any point of interest in the cluster "cloud."
    - *Importance:* By providing an **interactive** 3D scatter plot, researchers can zoom, pan, and click on clusters to see what kind of tweets are grouped together. This merges the benefits of visual analytics with big data exploration.
6. **Treemap Visualization of the Hierarchical Taxonomy**
    - Beyond just 3D scatter plots, the code also builds a **Treemap** from the hierarchical cluster structure, using Plotly's treemap.
    - Each node (cluster) is represented as a rectangle; child clusters are nested within their parent's rectangle. The area or color can reflect the size of each cluster.
    - *Importance:* This provides a compact, bird's-eye view of the entire topic hierarchy. Especially useful if one wants to see the relative distribution of tweets across different branches (e.g., a massive cluster about "Elections" vs. a smaller but still important cluster about "Public Health Debates").

4.3 Why This Pipeline Matters

1. **Handles Large, Noisy Real-World Data**
    - Twitter data is infamous for its informality, abbreviations, misspellings, and fleeting context. By automatically embedding tweets and grouping them, we mitigate the chaos of raw text and surface genuine thematic structures.
2. **Speeds Up Discovery of Trends**
    - Researchers or analysts can load the dataset, see the main topics at a glance, and quickly form a hypothesis or ask deeper questions. For instance, an epidemiologist studying misinformation might instantly see clusters of tweets referencing conspiracy theories, enabling targeted examination.
3. **Human Interpretability via Summaries**
    - Purely algorithmic cluster labels (like "Cluster 1," "Cluster 2," etc.) are not human-friendly. The LLM-based naming helps end users figure out whether a cluster is about "Vaccine Mandates," "Political Campaign Ads," or "Football Scores," saving them from reading thousands of tweets manually.
4. **Flexible, Extensible Design**
    - The pipeline is modular: one can swap in a different embedding model (e.g., a domain-specific LLM for medical or legal text) or a different clustering approach.

The code is structured to facilitate incremental improvements or expansions (like time-series clustering, advanced merging/splitting strategies, or more elaborate LLM prompt engineering).

5. **Immediate Visual Feedback**
    - The 3D scatter plot and treemap let researchers visually confirm that clusters are formed in a sensible manner and quickly detect if something is off (e.g., a user's text ended up in the "wrong" cluster, or the cluster is too broad to be meaningful).

Overall, **Section 4** details a multi-step pipeline that drastically **shortens the feedback loop** from raw data ingestion to high-level insights. By leveraging embeddings, clustering, dimensionality reduction, and automated LLM-based summarization, it helps researchers manage and make sense of large Twitter datasets with minimal friction—unlocking quicker and more accurate thematic exploration.

---

5. NL2SQL Agent for Natural Language Queries

5.1 Motivation

Not all users want to write SQL or even use a clustering approach. Sometimes, the question is simply, "Which tweets have the highest like count?" or "How many verified users posted about `#Trump2024`?" To bridge that gap, I built an **NL2SQL Agent** that translates plain-English questions into SQL queries. Please note that this tool is in Beta version and prone to bugs (sorry its hard to get these LLMs to behave in a stable manner 🙂).

5.2 Implementation Details

1. **Ollama Integration**:
    - I use Streamlit and Python to provide the user's query as a prompt to a local LLM.
    - The LLM sees a "system prompt" that describes the schema of `tweets` (obtained by running `DESCRIBE tweets`).
2. **Function Calling**:
    - Ollama has a tool-calling interface. The LLM is instructed that if it needs to run a SQL query, it must return a JSON object:

```
{
"name": "execute_sql_query",
"arguments": {"query": "<SQL>"}
}
```

    - My Streamlit app intercepts this function call, runs the query against DuckDB, and returns the results to the LLM.
2. **Final Response**:
    - After receiving the query results, the LLM outputs a final explanation or analysis for the user.

5.3 Advantages and Considerations

- **Ease of Use**: Users can quickly ask open-ended questions about the dataset, bypassing SQL.
- **Safety**: I limit the LLM to a read-only connection. The system is constrained to the `tweets` table, preventing data leakage or injection beyond that scope.
- **Complexity & Accuracy**: For advanced queries (joins, subqueries, window functions), the LLM might produce imperfect SQL. I've built partial error handling—errors are displayed, and the user can refine their question.

---

6. Summary of Accomplishments

1. **Data Cleaning & Ingestion**:
    - Dealt with mismatched CSV columns, embedded Python objects, and inconsistent data formats.
    - Built a robust parsing system that fixes or transforms messy data into a cohesive schema.
    - Stored the cleaned data in DuckDB for fast, SQL-based access.
2. **Analytic Pipeline**:
    - Developed a Streamlit app to load data from DuckDB, generate text embeddings, hierarchical clustering of tweets, reduce dimensionality, and visualize the results.
    - Created an automated cluster summarization technique using a local LLM.
3. **Natural Language Query Interface (NL2SQL)**:
    - Leveraged Ollama's function-calling to convert user queries to SQL automatically.
    - Returned both the results of the query and a final explanation in natural language.
4. **Modularity & Extensibility**:
    - Organized code so that each step (cleaning, embedding, clustering, NL2SQL) is reusable or independently upgradable.
    - Incorporated fallback, caching, and error-handling strategies to handle large volumes of data and variable data quality.

---

7. Ongoing Challenges and Future Directions

- **Data Scale**: As the number of tweets grows beyond millions, memory constraints could require additional optimizations such as partial clustering or sampling-based summaries.
- **Advanced-Data Quality**:
    - Further refining the user column parsing (some user fields may still be missing or have irregular JSON structures).
    - Handling ephemeral or changed user data (like a username change).
- **Topic Evolution**: Extending the clustering approach to examine how conversation topics evolve over time would be valuable (e.g., a time-series-based or sliding-window approach).
- **NL2SQL Complexity**: Ensuring that complicated queries are generated accurately by the LLM might require more advanced prompt engineering or an incremental clarifying question approach (“What do you mean by X?”).
- **Cross-Platform Analysis**: If future data merges multiple social networks, integrating multiple tables (e.g. Telegram, TikTok datasets) in DuckDB and adjusting the LLM’s tool calling accordingly will be crucial.

---

8. Conclusion

This project successfully tackles a common challenge in large-scale social media research—namely, messy CSV data with inconsistent schemas and embedded Python/JSON objects. Through robust data cleaning, chunked ingestion into DuckDB, and a series of user-friendly tools (clustering, summarization, NL2SQL), I created an end-to-end pipeline that empowers researchers to explore and analyze election-related tweets more efficiently.

By merging advanced local LLM-based capabilities (embeddings, summarization, function-calling for SQL) with high-level visualization (Plotly, Streamlit), My workflow stands as a comprehensive, scalable approach to social media analysis. Future expansions can build upon these solid foundations, incorporating more advanced modeling or multi-platform data synchronization, but the current project already demonstrates significant progress and provides a powerful toolkit for immediate research use.

Acknowledgment

Generative AI tools such as ChatGPT, and Claude were used in creating the code and content of this report.

References

1. ***DuckDB*** – *https://duckdb.org/*
   Used for its speed, ease of integration with Python, and ability to handle large-scale SQL queries in-memory.
2. ***Streamlit*** – *https://streamlit.io/*
   Chosen for its simplicity and interactivity in building data apps, enabling rapid prototyping and user-friendly interfaces.
3. ***Plotly*** – *https://plotly.com/*
   Utilized for advanced and interactive visualizations, including 3D scatter plots and treemaps.
4. ***Ollama*** – *https://ollama.com/*
   Used as a local LLM server for generating text embeddings and natural language responses, enabling offline or self-hosted large language model operations.
5. ***scikit-learn*** – *https://scikit-learn.org/stable/*
   Relied upon for its robust machine learning algorithms, including K-Means clustering and dimensionality reduction techniques (PCA, t-SNE).
6. ***Embedding and Clustering Idea Inspiration*** – *Daniel X. Sun, "Clustering Tweets via Tweet Embeddings," (MIT Thesis) Link*
   Provided methodological insights for generating high-quality tweet embeddings and clustering approaches.
7. **Datalens** – *An NL2SQL agent* Code
   I created Datalens which Inspired the design of the NL2SQL interface, showing how natural language queries can be translated into SQL with minimal user friction.