NYC Taxi Trips Dashboard: Technical Documentation

Team Members:

- Samuel NIYONKURU
- David NGARAMBE
- Attorney Valois NIYIGABA
- Prudence Browns

1. Problem Framing and Dataset Analysis

Dataset Description

We analyzed the New York City Taxi Trip Dataset, which includes detailed records of taxi trips such as timestamps, pickup/dropoff locations, trip durations, distances, and fare information. This dataset is essential for understanding urban mobility patterns in NYC.

Data Challenges

- Missing Values: Many records lacked critical fields like timestamps or coordinates.
- Invalid Data: Some records had illogical values (e.g., dropoff times before pickup times or coordinates outside NYC).
- Outliers: A few trips had extremely long durations or distances, likely due to data errors.
- Duplicates: Duplicate records needed removal to ensure data integrity.

Assumptions

- Trips with coordinates outside NYC bounds were considered invalid and excluded.
- Trips with dropoff times before pickup times were treated as data errors and removed.
- Coordinates were rounded to six decimal places for consistency.

Unexpected Observation

We noticed a significant number of trips with very short durations (less than 1 minute) and distances (less than 0.1 km). This led us to implement minimum thresholds for trip duration and distance to filter out likely data errors or canceled trips.

2. System Architecture and Design Decisions

System Architecture Diagram

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                          NYC Taxi Trips Dashboard
│
└───────────────┬───────────────────────────────────────────────┬──────────────┘
                 │                                               │
                 ▼                                               ▼
┌───────────────────────┐                          ┌───────────────────────┐
│       Frontend         │                          │       Backend          │
│  (HTML, CSS, JS)      │                          │  (Flask, Python)      │
└───────────────┬───────┘                          └───────────┬───────────┘
                │                                             │
                ▼                                             ▼
┌───────────────────────┐                          ┌───────────────────────┐
│   User Interaction     │                          │   Data Processing      │
│  (Filtering, Charts)   │                          │  (Cleaning, Queries)  │
└───────────────────────┘                          └───────────┬───────────┘
                                                              │
                                                              ▼
                                                      ┌───────────────────────┐
                                                      │       Database        │
                                                      │    (MySQL, Tables)    │
                                                      └───────────────────────┘
```

Architecture Description

Our system follows a three-tier architecture:

1. Frontend: Single-page application with dynamic filtering and visualization (HTML, CSS, JavaScript, Chart.js, Leaflet.js).
2. Backend: RESTful API with endpoints for data retrieval (Flask, Python).
3. Database: Normalized relational schema with tables for trips, vendors, and locations (MySQL).

Stack Choices Justification

- Frontend: Vanilla JavaScript for simplicity and broad compatibility.
- Backend: Flask for its lightweight nature and ease of integration with MySQL.
- Database: MySQL for reliability, performance, and widespread use.

Trade-offs

- We prioritized simplicity and maintainability over complex frameworks.
- Focused on core functionality to meet the project timeline.

3. Algorithmic Logic and Data Structures

Problem

We needed to efficiently filter and clean a large dataset of NYC taxi trips, removing invalid records based on missing fields, invalid timestamps, and out-of-bound coordinates without relying on built-in libraries.

Custom Implementation

We manually implemented a data cleaning algorithm in `data_cleaning.py`:

- Custom Datetime Parsing: Parses datetime strings into structured tuples.
- Custom Datetime Comparison: Validates that pickup times are before dropoff times.
- Custom Coordinate Validation: Ensures coordinates fall within NYC bounds.
- Custom CSV Reading/Writing: Handles file I/O without libraries.
- Custom Data Filtering: Filters and cleans data based on validation rules.

Pseudo-code

```
function parse_datetime(datetime_str):
    split datetime_str into date and time parts
    split date into year, month, day
    split time into hour, minute, second
    validate all components are within valid ranges
    return tuple of (year, month, day, hour, minute, second) or None if invalid

function compare_datetimes(dt1, dt2):
    compare year, month, day, hour, minute, second components
    return True if dt1 <= dt2, False otherwise

function is_valid_coordinate(lat, lon, nyc_bounds):
    convert lat and lon to float
    check if they are within NYC geographic bounds
    return True if valid, False otherwise

function read_csv(file_path):
    open file and read all lines
    extract headers from first line
    for each subsequent line, split into values and create a dictionary
    return headers and list of rows

function write_csv(file_path, headers, rows):
    open file for writing
    write headers as first line
    for each row, write values as comma-separated line

function clean_data(input_file, output_file, excluded_file):
    define NYC geographic bounds
    read input CSV file
    for each row:
        parse pickup and dropoff datetimes
        check for missing critical fields
        if missing fields, add to excluded_rows and continue
        check for invalid timestamps
        if invalid timestamps, add to excluded_rows and continue
        parse pickup and dropoff coordinates
        check for invalid coordinates
        if invalid coordinates, add to excluded_rows and continue
        round coordinates to 6 decimal places
        calculate trip_duration_min
        add row to cleaned_rows
    add 'trip_duration_min' to headers if not present
    write cleaned data to output file
    write excluded records to output file
```

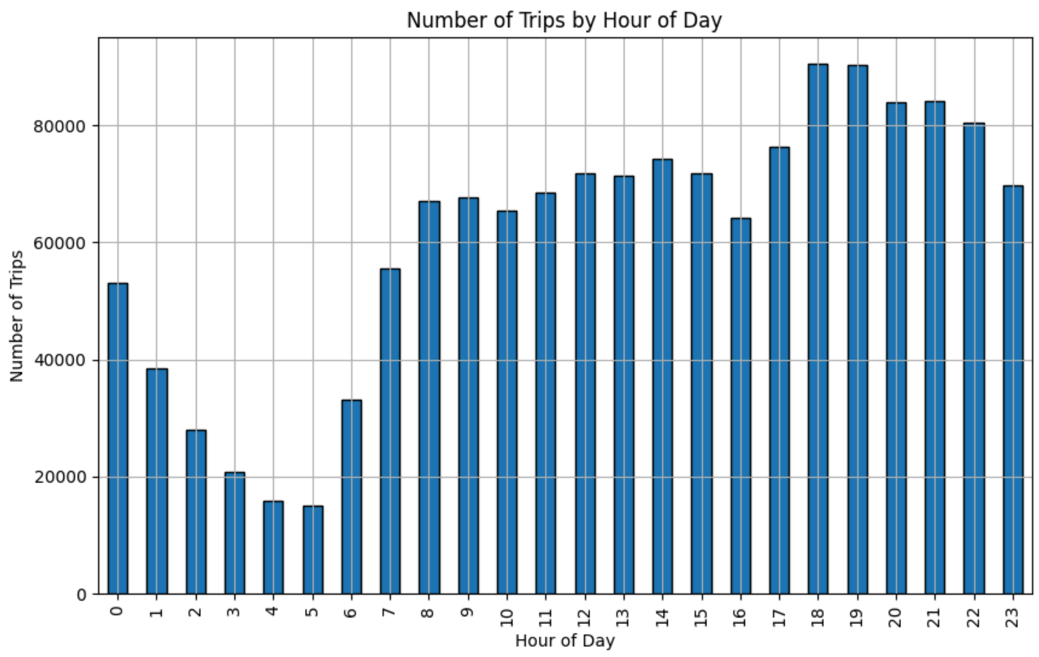
Complexity Analysis

- Time Complexity: O(n), where n is the number of rows. Each row is processed once.
- Space Complexity: O(n), where n is the number of rows. Both cleaned and excluded rows are stored in memory.

4. Insights and Interpretation

We used Peak Travel Times as our insight and interpretation

- Derivation: We analyzed trip counts by hour of the day.
- Visual:



- Interpretation: Peak travel times are 7-9 AM and 4-6 PM, corresponding to rush hours. This suggests high demand for taxis during commuting times.

5. Reflection and Future Work

Challenges

- Data Cleaning: Handling missing and invalid data was time-consuming but crucial for data quality.
- Performance: Processing large datasets required careful optimization.
- Collaboration: Coordinating between team members with different strengths was challenging but rewarding.

Improvements

- Scalability: Implement batch processing for very large datasets.
- Real-time Processing: Add support for streaming data to enable real-time analytics.
- Advanced Visualizations: Incorporate more interactive and dynamic visualizations.

Future Work

- Predictive Analytics: Implement machine learning models to predict demand and optimize taxi allocation.
- Integration: Connect with other urban mobility datasets (e.g., subway, bike-sharing) for a comprehensive view.
- Mobile App: Develop a mobile application for real-time trip tracking and analytics.

Technical Descriptions (Architecture Designs)

Frontend

- Structure: Single-page application with dynamic filtering and visualization.
- Technologies: HTML, CSS, JavaScript, Chart.js, Leaflet.js.
- Features: Interactive filters, dynamic charts, and maps.

Backend

- Structure: RESTful API with endpoints for data retrieval.
- Technologies: Flask, Python.
- Features: Data filtering, aggregation, and serving.

Database

- Structure: Normalized relational schema with tables for trips, vendors, and locations.
- Technologies: MySQL.
- Features: Indexed columns for fast querying, constraints for data integrity.

Explanation of Architecture and Choices

Frontend

We chose a single-page application to provide a seamless user experience without page reloads. Chart.js and Leaflet.js were selected for their ease of use and powerful visualization capabilities.

Backend

Flask was chosen for its simplicity and ease of integration with MySQL. We implemented custom data processing logic to ensure flexibility and control over data handling.

Database

We used MySQL for its reliability and performance. The schema was designed to be normalized, ensuring data integrity and efficient querying. Indexes were added to frequently queried columns to optimize performance.

Demonstration of Working Features

1. Data Filtering:
    - Users can filter trips by date range, duration, distance, and passenger count.
    - The frontend dynamically updates visualizations based on selected filters.
2. Interactive Visualizations:
    - Charts show distributions of trip durations, distances, and passenger counts.
    - Maps display pickup and dropoff locations.
3. Data Export:
    - Users can export filtered data for further analysis.