✅ Methodology for Step 1: Reading Raw Monthly Data

🔍 Objective: Efficiently load each monthly Yellow Taxi dataset in Parquet format into a Spark DataFrame.

🧰 Tools Used:

Apache Spark: for distributed DataFrame-based processing.

Parquet format: for efficient I/O.

📋 What Happens in This Step:

The script looks into the folder defined by RAW\_DATA\_PATH for files ending in .parquet (e.g., yellow\_tripdata\_2024-01.parquet).

For each file:

It uses Spark’s DataFrame API to read the Parquet file:

python

Copy

Edit

df = spark.read.parquet(file\_path)

It counts the number of rows and logs the result:

python

Copy

Edit

print(f"📦 Loaded {df.count()} raw rows from {filename}")

🧪 Example Output (from your logs):

sql

Copy

Edit

Loaded 2964624 raw rows from yellow\_tripdata\_2024-01.parquet

📦 Why Parquet is Ideal:

It's a compressed, columnar format.

Spark reads only the necessary columns for transformations, speeding up processing and saving memory.

🧾 Output of Step 1:

A Spark DataFrame named df that holds all raw rows for a month, ready to be cleaned and enriched in the next steps.

✅ STEP 2: Data Preprocessing Methodology

This step transforms raw or enriched trip data into a cleaner, feature-rich format suitable for analysis or modeling.

🧱 Inputs

Chunked CSVs or a full Parquet file from the ingestion step (e.g. full\_trip\_data\_part\_\*.csv or full\_trip\_data.parquet).

These contain fields such as tpep\_pickup\_datetime, pickup/dropoff zone, trip\_distance, fare\_amount, and location coordinates.

🔄 Overall Preprocessing Pipeline

1. 📥 Load Data

Accepts either a file path (CSV or Parquet) or a Pandas DataFrame.

Uses:

python

Copy

Edit

pd.read\_parquet() or pd.read\_csv()

2. 🧪 Downsampling (For Efficiency)

Randomly samples 10% of the data using:

python

Copy

Edit

df.sample(frac=0.1, random\_state=42)

Purpose: Speed up clustering and memory savings.

3. 🧹 Clean Data

Filters out:

trips with distance ≤ 0

negative fare\_amounts

records where pickup time == dropoff time

any row with missing (NaN) values

python

Copy

Edit

df = df[df['trip\_distance'] > 0]

df = df[df['fare\_amount'] >= 0]

df = df[df['tpep\_pickup\_datetime'] != df['tpep\_dropoff\_datetime']]

df.dropna(inplace=True)

4. 📐 Derived Fields

Calculates new fields to enrich the dataset:

Duration of trip in minutes:

python

Copy

Edit

duration = (dropoff - pickup).total\_seconds() / 60

Speed (in mph or kmph depending on distance unit):

python

Copy

Edit

speed = trip\_distance / (duration / 60)

5. ⏱ Time Features Extraction

Converts pickup datetime into:

hour of day (pickup\_hour)

day of month (pickup\_day)

weekday (pickup\_weekday)

is\_weekend (binary)

cyclical time features for ML models:

python

Copy

Edit

hour\_sin = sin(2π × hour / 24)

hour\_cos = cos(2π × hour / 24)

6. 🌍 Geospatial Clustering (MiniBatchKMeans)

Clusters pickup points (lon/lat) into zones using unsupervised learning.

Uses MiniBatchKMeans for efficiency on large data:

python

Copy

Edit

MiniBatchKMeans(n\_clusters=10, batch\_size=1000)

Clusters are stored in a new column zone.

🗃 Output

Final DataFrame with original + derived + time + cluster zone features.

Saved as:

Copy

Edit

full\_trip\_data\_cleaned.parquet

🧠 Key Strengths

Scalable: Supports both full and sampled datasets.

Memory-efficient: Downsampling + batch clustering.

ML-ready: Extracted features + clustering zones can improve model quality.

✅ STEP 3: Feature Engineering & Dimensionality Reduction (PCA)

This step transforms the cleaned dataset into a set of enriched, numerically optimized features that are reduced in dimension for efficient downstream ML or clustering tasks.

🔹 INPUT

Cleaned data: full\_trip\_data\_cleaned.parquet

Output of Step 2

🔧 Methodology

1. 📥 Load Data

python

Copy

Edit

df = pd.read\_parquet(input\_path)

Input file: full\_trip\_data\_cleaned.parquet

2. 🛠 Feature Engineering

(a) 🌍 Geographical Features

If trip\_distance is missing and coordinates exist, calculate it using the Haversine formula.

Calculate pickup/dropoff zone density (count of trips starting or ending in a zone).

Calculate zone-to-zone trip frequency (zone\_flow).

python

Copy

Edit

df['trip\_distance'] = calculate\_haversine\_distance(...)

df['pickup\_zone\_density'] = df['pickup\_zone'].map(value\_counts)

df['zone\_flow'] = merged from grouped pickup/dropoff pairs

(b) ⏰ Time Features

Extract from tpep\_pickup\_datetime:

hour, day, month, weekday

cyclical encoding (sin/cos of hour, weekday, month)

is\_weekend, is\_rush\_hour (7–10 AM & 4–7 PM)

time\_of\_day bucket (night, morning, afternoon, evening)

(c) 🚕 Trip Features

fare\_per\_mile = fare\_amount / trip\_distance

tip\_percentage = (tip\_amount / fare\_amount) \* 100

passenger\_group = bins of 1, 2, 3–4, 5+

speed\_category = binned from speed (very\_slow, slow, normal, fast)

💾 Result of Feature Engineering

Many raw features transformed into structured ones.

Resulting number of features before PCA: 40

3. 🧪 Prepare for PCA

Select numeric columns only

Drop identifiers (VendorID, zone, etc.)

Fill NA with column mean

Apply StandardScaler to normalize values

python

Copy

Edit

scaled\_data = scaler.fit\_transform(df\_numeric)

4. 🔻 Dimensionality Reduction using PCA

python

Copy

Edit

pca = PCA(n\_components=0.95)

pca\_transformed = pca.fit\_transform(scaled\_data)

PCA retains enough components to preserve 95% variance

In this case: 27 components retained out of 40

Result:

matlab

Copy

Edit

PCA complete. Retained 27 components explaining 95.63% of variance.

5. 📊 Feature Importance

Importance is derived from absolute sum of PCA loadings across all components.

Top 10 features:

markdown

Copy

Edit

1. DOLocationID

2. extra

3. pickup\_lat

4. hour\_cos

5. is\_rush\_hour

6. fare\_per\_mile

7. mta\_tax

8. improvement\_surcharge

9. PULocationID

10. pickup\_zone\_density

6. 💾 Save Outputs

Final dataset with new features + PCA components: full\_trip\_data\_with\_features.parquet

PCA model: pca\_model.pkl

Scaler model: scaler\_model.pkl

📈 Summary

Phase Description

Load Cleaned Data Read full\_trip\_data\_cleaned.parquet

Feature Engineering Add geo, time, trip features with transformations

Scaling Normalize features with StandardScaler

PCA Reduce dimensionality (retain 95% variance) to 27 components

Output Save processed data + models for reuse

Would you like a CSV or PNG export of the top 20 feature importances or a visual of the PCA variance plot?

✅ Detailed Methodology for Step 4: Clustering and Zone Labeling

Step 4 of the project involves clustering high-traffic zones and assigning labels (red, yellow, green) based on traffic congestion. Below is a detailed methodology derived from your clustering pipeline:

1. Data Loading (PCA Features)

Input: The clustering process starts by loading a pre-processed dataset containing PCA features.

File Path:

kotlin

Copy code

C:\Users\VaishnaviM\Desktop\BIG\_DATA\data\processed\full\_trip\_data\_with\_features.parquet

Code Snippet:

python

Copy code

def load\_feature\_data(input\_path):

print(f"Loading PCA features from {input\_path}")

return pd.read\_parquet(input\_path)

Explanation:

Uses the pandas library to read the Parquet file containing PCA-transformed data.

Efficient data loading due to the use of Parquet format.

Logs the file path for debugging purposes.

2. Clustering Using K-Means

Objective: Identify high-traffic zones by clustering similar areas based on PCA features.

Number of Clusters: 10 (configurable)

Clustering Algorithm: K-Means from sklearn.cluster

Code Snippet:

python

Copy code

def run\_kmeans\_clustering(df, n\_clusters=10, pca\_cols\_prefix='pca\_component\_'):

print(f"Running KMeans clustering with {n\_clusters} clusters...")

pca\_cols = [col for col in df.columns if col.startswith(pca\_cols\_prefix)]

kmeans = KMeans(n\_clusters=n\_clusters, random\_state=42)

df['zone\_cluster'] = kmeans.fit\_predict(df[pca\_cols])

return df, kmeans

Explanation:

Extracts PCA columns dynamically from the dataset.

Initializes the K-Means algorithm with a fixed random state for reproducibility.

Fits the model to the PCA features and assigns cluster labels to each data point.

Appends the cluster assignments to the DataFrame as a new column, zone\_cluster.

3. Zone Labeling Based on Traffic Speed

Objective: Categorize clusters into traffic zones (red, yellow, green) based on average speed.

Code Snippet:

python

Copy code

def assign\_zone\_labels(df):

print("Assigning red/yellow/green labels based on average speed...")

cluster\_speed = df.groupby('zone\_cluster')['speed'].mean().sort\_values()

cluster\_labels = {}

total\_clusters = len(cluster\_speed)

for i, cluster\_id in enumerate(cluster\_speed.index):

if i < total\_clusters / 3:

cluster\_labels[cluster\_id] = 'red'

elif i < 2 \* total\_clusters / 3:

cluster\_labels[cluster\_id] = 'yellow'

else:

cluster\_labels[cluster\_id] = 'green'

df['traffic\_zone'] = df['zone\_cluster'].map(cluster\_labels)

return df

Explanation:

Groups data by cluster and calculates the average speed for each cluster.

Sorts the clusters by speed in ascending order.

Divides the clusters into three equal parts and assigns labels:

Red: Lowest third (high congestion)

Yellow: Middle third (moderate congestion)

Green: Highest third (low congestion)

Maps these labels to the DataFrame as a new column, traffic\_zone.

4. Saving Clustered Data

Objective: Persist the results for further analysis and visualization.

Code Snippet:

python

Copy code

def save\_clustered\_data(df, output\_path):

os.makedirs(os.path.dirname(output\_path), exist\_ok=True)

df.to\_parquet(output\_path, index=False)

print(f"Clustered data saved to {output\_path}")

Explanation:

Ensures the output directory exists, creating it if necessary.

Saves the clustered and labeled data as a Parquet file for efficient storage.

Logs the save path for confirmation.

File Path:

kotlin

Copy code

C:\Users\VaishnaviM\Desktop\BIG\_DATA\data\processed\clustered\_trip\_data.parquet

5. Clustering Pipeline Execution

Objective: Execute the clustering and labeling steps in a streamlined pipeline.

Code Snippet:

python

Copy code

def run\_clustering\_pipeline(input\_path, output\_path, n\_clusters=10):

df = load\_feature\_data(input\_path)

df, \_ = run\_kmeans\_clustering(df, n\_clusters=n\_clusters)

df = assign\_zone\_labels(df)

save\_clustered\_data(df, output\_path)

return df

Explanation:

Loads the input data.

Performs clustering and zone labeling.

Saves the final clustered data to the specified path.

Returns the processed DataFrame for further use.

Key Outputs:

Clustered data with zone assignments saved at:

kotlin

Copy code

C:\Users\VaishnaviM\Desktop\BIG\_DATA\data\processed\clustered\_trip\_data.parquet

Log messages for each step for easy tracking and debugging.

Advantages of the Clustering Approach:

Efficiency: Uses PCA for dimensionality reduction and faster clustering.

Scalability: K-Means efficiently handles large datasets, suitable for NYC traffic data.

Interpretability: Clustering results are easily understandable with labeled zones.

Flexibility: The number of clusters can be easily adjusted for experimentation.

Would you like assistance with visualizing the clustered zones or further analysis?

✅ Detailed Methodology for Step 5: Time Series Prediction

Step 5 involves predicting traffic speed for different zones using time series analysis and XGBoost regression. Below is a detailed breakdown derived from your code:

1. Data Loading (Clustered Data)

Input: Clustered data from the previous step.

File Path:

kotlin

Copy code

C:\Users\VaishnaviM\Desktop\BIG\_DATA\data\processed\clustered\_trip\_data.parquet

Code Snippet:

python

Copy code

def load\_clustered\_data(file\_path):

print(f"Loading clustered data from {file\_path}")

return pd.read\_parquet(file\_path)

Explanation:

Uses pandas to load the clustered trip data.

Efficiently loads the file using the Parquet format.

Logs the file path for verification.

2. Preparing Time Series Features

Objective: Extract meaningful features from the timestamp data for prediction.

Steps:

Datetime Parsing: Convert timestamp fields to a consistent format.

Time-Based Features: Extract hour and weekday from the timestamp.

Grouping: Aggregate data by zone, date, and hour to calculate average speed.

Code Snippet:

python

Copy code

def prepare\_time\_series\_features(df):

print("Preparing features for time series modeling...")

if 'pickup\_datetime' in df.columns:

df['pickup\_datetime'] = pd.to\_datetime(df['pickup\_datetime'])

df['hour'] = df['pickup\_datetime'].dt.hour

df['weekday'] = df['pickup\_datetime'].dt.weekday

grouped = df.groupby(['zone\_cluster', df['pickup\_datetime'].dt.date, 'hour']).agg({

'speed': 'mean'

}).reset\_index().rename(columns={

'pickup\_datetime': 'date',

'speed': 'avg\_speed'

})

grouped['date'] = pd.to\_datetime(grouped['date'])

grouped = grouped.sort\_values(['zone\_cluster', 'date', 'hour'])

return grouped

Explanation:

Ensures proper parsing of datetime fields.

Creates time-based features like hour and weekday for capturing temporal patterns.

Groups data to calculate hourly average speed per zone.

Sorts the grouped data to maintain temporal consistency.

3. Creating Lag Features

Objective: Enhance time series modeling by adding lagged speed values as features.

Code Snippet:

python

Copy code

def create\_lag\_features(df, lag\_hours=[1, 2, 3]):

print(f"Creating lag features: {lag\_hours}")

for lag in lag\_hours:

df[f'lag\_{lag}'] = df.groupby('zone\_cluster')['avg\_speed'].shift(lag)

df.dropna(inplace=True)

return df

Explanation:

Uses a sliding window approach to create lag features (e.g., speed values from the last 1, 2, and 3 hours).

Groups by zone\_cluster to maintain context within each zone.

Drops rows with NaN values caused by lagging.

Lag Features Created:

lag\_1, lag\_2, lag\_3

4. Training the XGBoost Model

Objective: Train a regression model to predict average speed.

Code Snippet:

python

Copy code

def train\_xgboost\_model(df):

print("Training XGBoost model...")

feature\_cols = ['hour', 'lag\_1', 'lag\_2', 'lag\_3']

target\_col = 'avg\_speed'

X = df[feature\_cols]

y = df[target\_col]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

model = xgb.XGBRegressor(objective='reg:squarederror', n\_estimators=100)

model.fit(X\_train, y\_train)

preds = model.predict(X\_test)

rmse = np.sqrt(mean\_squared\_error(y\_test, preds))

mae = mean\_absolute\_error(y\_test, preds)

r2 = r2\_score(y\_test, preds)

print(f"XGBoost RMSE: {rmse:.4f}")

print(f"MAE: {mae:.4f}")

print(f"R²: {r2:.4f}")

return model

Explanation:

Uses the lag features and hour as predictors.

Splits the data into training (80%) and testing (20%) sets.

Trains an XGBoost regressor with 100 trees.

Evaluates the model using RMSE, MAE, and R² metrics.

Output Metrics:

makefile

Copy code

XGBoost RMSE: 8.1273

MAE: 2.1432

R²: 0.7342

5. Validating the Prediction

Objective: Test the model's ability to predict speed for a given zone at a specific hour.

Code Snippet:

python

Copy code

def validate\_prediction(model, df, hour=15, zone\_id=0):

print(f"\nValidating prediction for zone {zone\_id} at hour {hour}...")

actual\_speed = df[(df['zone\_cluster'] == zone\_id) & (df['hour'] == hour)]['speed'].mean()

sample = pd.DataFrame({'hour': [hour], 'lag\_1': [18.5], 'lag\_2': [19.2], 'lag\_3': [17.8]})

predicted\_speed = model.predict(sample)[0]

print(f"Predicted speed: {predicted\_speed:.2f} mph")

print(f"Actual average speed: {actual\_speed:.2f} mph")

print(f"Absolute error: {abs(predicted\_speed - actual\_speed):.2f} mph")

Explanation:

Uses lagged speed values from recent hours as inputs.

Predicts the speed for the specified hour and zone.

Compares the predicted speed to the actual average speed.

Output:

yaml

Copy code

Predicted speed: 16.44 mph

Actual average speed: 6.45 mph

Absolute error: 9.99 mph

6. Saving the Trained Model

Objective: Save the trained XGBoost model for future use.

Code Snippet:

python

Copy code

def save\_model(model, model\_path):

os.makedirs(os.path.dirname(model\_path), exist\_ok=True)

joblib.dump(model, model\_path)

print(f"Model saved to {model\_path}")

File Path:

makefile

Copy code

C:\Users\VaishnaviM\Desktop\BIG\_DATA\models\xgboost\_model.pkl

7. Prediction Pipeline Execution

Objective: Execute all the above steps in a streamlined pipeline.

Code Snippet:

python

Copy code

def run\_prediction\_pipeline(input\_path, model\_output\_path, validate=True):

df = load\_clustered\_data(input\_path)

ts\_df = prepare\_time\_series\_features(df)

ts\_df = create\_lag\_features(ts\_df)

model = train\_xgboost\_model(ts\_df)

save\_model(model, model\_output\_path)

if validate:

validate\_prediction(model, ts\_df, hour=15, zone\_id=0)

Explanation:

Runs the complete pipeline from loading data to saving the trained model.

Includes a validation step to check prediction accuracy.

Advantages of the Time Series Prediction Approach:

Efficiency: Uses lag features to capture temporal dependencies.

Accuracy: XGBoost handles non-linear patterns effectively.

Model Persistence: Saves the trained model for future predictions.

Validation: Immediate feedback on model performance.

Would you like help with visualizing the prediction results or further tuning the model?

Detailed Methodology for Step 6: Map Visualization of Traffic Clusters

✅ Step 6 involves visualizing the clustered traffic data on a map using the Folium library. This step helps in spatial analysis of traffic congestion zones. Below is a comprehensive breakdown derived from your code:

1. Data Loading (Clustered Trip Data)

Objective: Load the clustered traffic data generated from previous steps.

File Path:

bash

Copy code

../data/processed/clustered\_trip\_data.parquet

Code Snippet:

python

Copy code

def load\_clustered\_data(file\_path):

print(f"Loading clustered trip data from {file\_path}")

return pd.read\_parquet(file\_path)

Explanation:

Uses the pandas library to read the Parquet file.

Logs the file path for tracking purposes.

Returns the loaded DataFrame for visualization.

2. Creating a Color Map for Traffic Zones

Objective: Assign colors to zones based on traffic congestion levels (red, yellow, green).

Code Snippet:

python

Copy code

def create\_color\_map(zone\_label):

color\_map = {

'red': 'red',

'yellow': 'orange',

'green': 'green'

}

return color\_map.get(zone\_label, 'blue')

Explanation:

Defines a mapping between traffic zones and colors:

Red: High congestion

Yellow: Moderate congestion

Green: Low congestion

Uses a default color ("blue") for any unclassified zone.

3. Plotting Clusters on a Map

Objective: Display traffic clusters on a map with interactive markers.

Code Snippet:

python

Copy code

def plot\_clusters\_on\_map(df, output\_html):

print("Plotting traffic zones on NYC map...")

nyc\_center = [40.7128, -74.0060]

folium\_map = folium.Map(location=nyc\_center, zoom\_start=11)

# Clustered markers for better visualization

marker\_cluster = MarkerCluster().add\_to(folium\_map)

for \_, row in df.iterrows():

lat = row['pickup\_lat']

lon = row['pickup\_lon']

label = row['traffic\_zone']

color = create\_color\_map(label)

folium.CircleMarker(

location=[lat, lon],

radius=5,

color=color,

fill=True,

fill\_color=color,

fill\_opacity=0.7,

popup=f"Zone: {label} | Speed: {row['speed']:.2f} mph"

).add\_to(marker\_cluster)

# Save the map as an HTML file

os.makedirs(os.path.dirname(output\_html), exist\_ok=True)

folium\_map.save(output\_html)

print(f"Map saved to {output\_html}")

Explanation:

Initializes a Folium map centered on NYC with a zoom level of 11.

Uses MarkerCluster to group markers for improved performance.

Iterates over each row to add a circle marker:

Location: Uses pickup latitude and longitude.

Color: Based on traffic zone.

Popup: Displays traffic zone and average speed.

Saves the interactive map to an HTML file.

Sample Map Path:

bash

Copy code

../outputs/maps/nyc\_clustered\_traffic\_map.html

4. Generating the Interactive Map

Objective: Execute the visualization pipeline, including data sampling for better performance.

Code Snippet:

python

Copy code

def generate\_map(file\_path, output\_html, sample\_frac=0.01):

df = load\_clustered\_data(file\_path)

# Downsample for performance

if sample\_frac < 1.0:

original\_count = len(df)

df = df.sample(frac=sample\_frac, random\_state=42)

print(f"Sampled {len(df)} records from {original\_count} (fraction={sample\_frac})")

plot\_clusters\_on\_map(df, output\_html)

Explanation:

Loads the clustered data.

Sampling:

Uses a sample fraction (1% by default) to limit the number of plotted points.

Reduces the dataset size to improve performance during rendering.

Calls the plot\_clusters\_on\_map function to create the visualization.

Map Output:

bash

Copy code

../outputs/maps/nyc\_clustered\_traffic\_map.html

5. Running the Visualization Pipeline

Code Snippet:

python

Copy code

if \_\_name\_\_ == "\_\_main\_\_":

input\_file = "data/processed/clustered\_trip\_data.parquet"

output\_map = "outputs/maps/nyc\_traffic\_zones\_map.html"

generate\_map(input\_file, output\_map, sample\_frac=0.10)

Explanation:

Calls the generate\_map function with specified file paths.

Uses a higher sample fraction (10%) for better coverage in the final output.

Console Output:

javascript

Copy code

Creating Folium map with MarkerCluster...

Sampling 10000 rows from 1980062 total rows...

Map saved to: ../outputs/maps/nyc\_clustered\_traffic\_map.html

6. Key Visualization Features:

Interactive Markers: Clickable points display congestion level and speed.

Color-Coded Zones:

Red: High congestion

Yellow: Moderate congestion

Green: Low congestion

Efficient Plotting: Uses clustering and sampling for optimal rendering.

Easy Navigation: Map centers around NYC for a clear view.

7. Use of HERE API Key:

Purpose: Enhance map visualization with additional geographic data or custom tiles (not directly shown in the script).

Integration: The code uses Folium, but integrating HERE Maps can further improve visualization (e.g., traffic layers, heatmaps).

Configuration: Add API key to the map initialization if using HERE tiles.

8. Advantages of the Visualization Approach:

Scalability: Downsampling ensures performance even with large datasets.

Interactivity: Allows users to explore traffic clusters interactively.

Customization: Easily extendable with additional features like heatmaps or route plotting.

Location Accuracy: Clustering markers reduce clutter and highlight key zones.

Would you like assistance with improving the map visualization or adding dynamic layers?

✅ Detailed Methodology for Step 7: Live Traffic Speed Comparison

Step 7 involves comparing the predicted traffic speed from the clustering and modeling steps with live traffic data fetched using the OpenRouteService API. This step helps in evaluating the accuracy of the model predictions against real-world conditions.

1. Loading Random Trip Samples

Objective: Load a random sample of clustered trip data for comparison.

File Path:

kotlin

Copy code

C:\Users\VaishnaviM\Desktop\BIG\_DATA\data\processed\clustered\_trip\_data.parquet

Code Snippet:

python

Copy code

def load\_trip\_samples(parquet\_path, n\_samples=10):

print(f"Loading {n\_samples} random trips from {parquet\_path}")

df = pd.read\_parquet(parquet\_path)

sample\_df = df[[

'pickup\_lat', 'pickup\_lon', 'dropoff\_lat', 'dropoff\_lon',

'speed', 'pickup\_zone', 'dropoff\_zone'

]].dropna().sample(n=n\_samples, random\_state=42)

return sample\_df

Explanation:

Reads the clustered data using the pandas library.

Extracts relevant columns like pickup/dropoff latitude and longitude, speed, and zones.

Drops rows with missing values and takes a random sample of trips for comparison.

Sample Size: 400 trips

2. Fetching Live Speed from the OpenRouteService API

Objective: Get the real-time speed between the pickup and dropoff locations.

API Details:

Base URL:

bash

Copy code

https://api.openrouteservice.org/v2/matrix/driving-car

Headers:

python

Copy code

headers = {

'Accept': 'application/json, application/geo+json, application/gpx+xml, img/png; charset=utf-8',

'Authorization': HEIGIT\_API\_KEY,

'Content-Type': 'application/json; charset=utf-8'

}

Code Snippet:

python

Copy code

def get\_live\_speed(pickup, dropoff):

body = {

"locations": [pickup, dropoff],

"metrics": ["distance", "duration"],

"units": "m"

}

try:

response = requests.post(HEIGIT\_API\_URL, json=body, headers=headers)

response.raise\_for\_status()

data = response.json()

distance\_m = data['distances'][0][1]

duration\_s = data['durations'][0][1]

if duration\_s == 0:

return None

speed\_mps = distance\_m / duration\_s

speed\_mph = speed\_mps \* 2.23694

return speed\_mph

except Exception as e:

print(f"API error: {e}")

return None

Explanation:

Sends a POST request to the OpenRouteService API with pickup and dropoff coordinates.

Retrieves the distance and duration between the two points.

Calculates speed as:

speed (mps)

=

distance (m)

duration (s)

speed (mps)=

duration (s)

distance (m)

​

speed (mph)

=

speed (mps)

×

2.23694

speed (mph)=speed (mps)×2.23694

Returns the speed if successfully retrieved, otherwise returns None.

Challenges:

Handles API errors such as connection issues and response timeouts.

Errors Encountered:

vbnet

Copy code

API error: HTTPSConnectionPool(host='api.openrouteservice.org', port=443): Read timed out.

API error: ('Connection aborted.', ConnectionAbortedError(10053, 'An established connection was aborted by the software in your host machine'))

3. Comparing Predicted vs. Live Data

Objective: Calculate the difference between predicted and live speeds.

Code Snippet:

python

Copy code

def compare\_with\_live\_data(df):

print("Comparing predicted vs live traffic speed...")

results = []

for \_, row in df.iterrows():

pickup = [row['pickup\_lon'], row['pickup\_lat']]

dropoff = [row['dropoff\_lon'], row['dropoff\_lat']]

predicted\_speed = row['speed']

live\_speed = get\_live\_speed(pickup, dropoff)

time.sleep(1) # API rate limiting

if live\_speed is not None:

error = abs(predicted\_speed - live\_speed)

results.append({

'predicted\_speed': predicted\_speed,

'live\_speed': live\_speed,

'error': error,

'lat': pickup[1],

'lon': pickup[0]

})

return pd.DataFrame(results)

Explanation:

Iterates through each row in the sampled DataFrame.

Calls get\_live\_speed() to fetch the actual speed.

Computes the absolute error between predicted and live speeds.

Appends results to a list for later conversion to a DataFrame.

Handling Rate Limiting:

Uses time.sleep(1) to avoid API rate limit errors.

4. Saving Comparison Results

Objective: Store the comparison results for analysis.

Code Snippet:

python

Copy code

def run\_live\_comparison(input\_path, output\_csv="outputs/predictions/live\_vs\_predicted.csv", n\_samples=10):

df = load\_trip\_samples(input\_path, n\_samples=n\_samples)

df\_live = compare\_with\_live\_data(df)

os.makedirs(os.path.dirname(output\_csv), exist\_ok=True)

df\_live.to\_csv(output\_csv, index=False)

print(f"Results saved to {output\_csv}")

print(df\_live.describe())

Explanation:

Loads the sampled data.

Compares predicted and live speeds.

Saves the comparison result as a CSV file.

Output File Path:

makefile

Copy code

C:\Users\VaishnaviM\Desktop\BIG\_DATA\outputs\predictions\live\_vs\_predicted.csv

Summary of Results:

matlab

Copy code

predicted\_speed live\_speed error lat lon

count 394.000000 394.000000 394.000000 394.000000 394.000000

mean 10.204612 19.216576 9.451823 40.748448 -73.964155

std 5.590909 4.918373 4.870066 0.033989 0.053021

min 0.401753 3.035847 0.149014 40.639832 -74.015999

25% 6.732983 15.658580 6.160394 40.736126 -73.991798

50% 8.898445 18.564393 9.106165 40.754398 -73.981499

75% 12.027108 21.455751 12.525565 40.767479 -73.956703

max 43.609756 35.627333 26.868918 40.913944 -73.778740

5. Running the Live Comparison Pipeline

Code Snippet:

python

Copy code

if \_\_name\_\_ == "\_\_main\_\_":

input\_file = r"C:\Users\VaishnaviM\Desktop\BIG\_DATA\data\processed\clustered\_trip\_data.parquet"

run\_live\_comparison(input\_file, n\_samples=400)

Explanation:

Loads the trip data and performs live speed comparison for 400 samples.

Saves the final output for further analysis.

Advantages of the Live Comparison Approach:

Accuracy Check: Directly compares predicted speeds with real-time data.

Error Handling: Graceful handling of API errors ensures pipeline robustness.

Data Storage: Results are stored in a structured format for post-analysis.

Performance Metrics: Descriptive statistics provide a quick summary.

Would you like assistance with further data analysis or visualization of the prediction errors?

✅ Detailed Methodology for Step 8: Web-Based Dashboard for NYC Traffic Analysis

Step 8 involves building an interactive dashboard using Streamlit and Folium to visualize predicted and live traffic data. This dashboard allows users to analyze, compare, and visualize traffic conditions in NYC.

1. Setting Up the Web Application

Framework: Streamlit

Map Library: Folium

Deployment: Streamlit application hosted locally or on a server.

File Path:

bash

Copy code

dashboard/app.py

2. Loading Data Efficiently

Objective: Load preprocessed traffic data and live comparison data efficiently.

Code Snippet:

python

Copy code

@st.cache\_data

def load\_clustered\_data():

return pd.read\_parquet(CLUSTERED\_DATA\_PATH)

@st.cache\_data

def load\_comparison\_data():

if os.path.exists(LIVE\_RESULTS\_PATH):

return pd.read\_csv(LIVE\_RESULTS\_PATH)

else:

return pd.DataFrame()

Explanation:

Uses Streamlit's @st.cache\_data to cache loaded data for faster reloading.

Loads clustered data and live comparison data from predefined file paths.

Returns an empty DataFrame if live data is unavailable.

Data Paths:

ini

Copy code

CLUSTERED\_DATA\_PATH = "data/processed/clustered\_trip\_data.parquet"

LIVE\_RESULTS\_PATH = "outputs/predictions/live\_vs\_predicted.csv"

3. Interactive Map Generation

Objective: Visualize traffic zones interactively with color-coded markers.

Code Snippet:

python

Copy code

def create\_interactive\_map(df, selected\_point=None, map\_title="Predicted", is\_actual=False):

color\_map = {"red": "red", "yellow": "orange", "green": "green"}

nyc\_center = [40.7128, -74.0060]

folium\_map = folium.Map(location=nyc\_center, zoom\_start=11)

for \_, row in df.iterrows():

lat = row.get("pickup\_lat") if not is\_actual else row.get("lat")

lon = row.get("pickup\_lon") if not is\_actual else row.get("lon")

popup\_text = f"Speed: {row.get('speed', 0):.2f} mph" if not is\_actual else f"Actual: {row['live\_speed']:.2f} mph"

color = color\_map.get(row.get("traffic\_zone"), "purple") if not is\_actual else "purple"

radius = 4

if selected\_point and abs(lat - selected\_point['lat']) < 0.0001 and abs(lon - selected\_point['lng']) < 0.0001:

radius = 6

color = "black"

folium.CircleMarker(

location=[lat, lon],

radius=radius,

color=color,

fill=True,

fill\_opacity=0.7,

popup=popup\_text

).add\_to(folium\_map)

return folium\_map

Explanation:

Generates a Folium map centered on NYC.

Uses different colors to indicate traffic congestion levels (red, yellow, green).

Highlights the selected point with a black marker.

Uses Streamlit-Folium for interactive map embedding.

Map Features:

Interactive markers displaying traffic zone and speed.

Dynamic updates based on user selection.

Separate maps for predicted and actual speeds.

4. Building the Streamlit Dashboard

Objective: Create an interactive and user-friendly dashboard.

Layout Configuration:

python

Copy code

st.set\_page\_config(page\_title="NYC Traffic Zone Dashboard", layout="wide")

st.title("🚦 NYC Traffic Analysis Dashboard")

Sidebar for User Input:

python

Copy code

st.sidebar.header("Filters")

selected\_zone = st.sidebar.selectbox("Select Traffic Zone", options=["All", "red", "yellow", "green"])

sample\_size = st.sidebar.slider("Number of Points", 100, 2000, 500)

Explanation:

Uses Streamlit's sidebar to provide filtering options.

Allows users to select the traffic zone and the number of points to display.

Default sample size set to 500 points.

5. Displaying Predicted and Actual Maps

Objective: Side-by-side comparison of predicted and actual traffic speeds.

Code Snippet:

python

Copy code

st.subheader("🗺️ Live vs Predicted Traffic Comparison")

col1, col2 = st.columns(2)

with col2:

st.markdown("### 🟣 Actual Traffic Map")

if not comp\_df.empty:

st\_folium(create\_interactive\_map(comp\_df, map\_title="Actual", is\_actual=True), width=450, height=500)

with col1:

st.markdown("### 🔵 Predicted Traffic Map")

st\_folium(create\_interactive\_map(df\_sample), width=450, height=500)

Explanation:

Uses Streamlit columns to display the predicted and actual maps side by side.

Interactive maps update based on user selection.

Shows both predicted and live speeds with color-coded markers.

6. Speed Lookup by Pickup and Dropoff Zones

Objective: Enable users to check the predicted and actual speed between specific zones.

Code Snippet:

python

Copy code

selected\_pickup = st.selectbox("Select Pickup Zone", sorted(available\_pickups))

selected\_dropoff = st.selectbox("Select Dropoff Zone", sorted(available\_dropoffs))

match\_rows = df[(df["pickup\_zone"] == selected\_pickup) & (df["dropoff\_zone"] == selected\_dropoff)]

if not match\_rows.empty:

avg\_predicted = match\_rows["speed"].mean()

st.success(f"🚗 Predicted Average Speed: {avg\_predicted:.2f} mph")

Explanation:

Dropdowns to select pickup and dropoff zones.

Displays average speed for the selected route.

Shows both predicted and actual speeds when available.

7. Running the Web Application

Entry Point:

python

Copy code

if \_\_name\_\_ == "\_\_main\_\_":

main()

Execution:

bash

Copy code

streamlit run app.py

Output:

Interactive dashboard with maps and traffic speed analysis.

User can dynamically compare predicted and actual traffic speeds.

Advantages of the Web Dashboard Approach:

User-Friendly: Provides an interactive interface for non-technical users.

Real-Time Visualization: Dynamically updates maps and speed comparisons.

Side-by-Side Comparison: Helps in visually analyzing the difference between predicted and live speeds.

Customization: Users can filter by traffic zones and check specific routes.

Performance: Caching mechanisms reduce data loading time.