Big Data Project

Team members

- Ranim Mostafa Hamed – 22011576
- Rowan Fayez Mohamed – 22011454
- Sara Sameh Ahmed – 22011968
- Ola Ragab Saad Ali – 22010367
- Shorouq Tareq Ahmed – 22010350
- Nourhan Essam El-Din Mohamed – 22010287

Data Lake Setup Summary

1. Data Lake Concept

A **Data Lake** is a centralized repository that allows you to store all structured and unstructured data at any scale. It supports multiple stages of data processing:

- **Bronze Layer:** Raw, unprocessed data (CSV files).
- **Silver Layer:** Cleaned and pre-processed data (Parquet files).
- **Gold Layer:** Final aggregated or analyzed results (e.g., factor analysis outputs).

2. HDFS Overview

Hadoop Distributed File System (HDFS) provides distributed storage, fault tolerance, and high-throughput access. It is used in this project to store cleaned and merged data in a structured way, allowing scalable processing.

3. MinIO Role

MinIO is used as an S3-compatible object storage to emulate a cloud-like data lake locally. Its benefits in the project include:

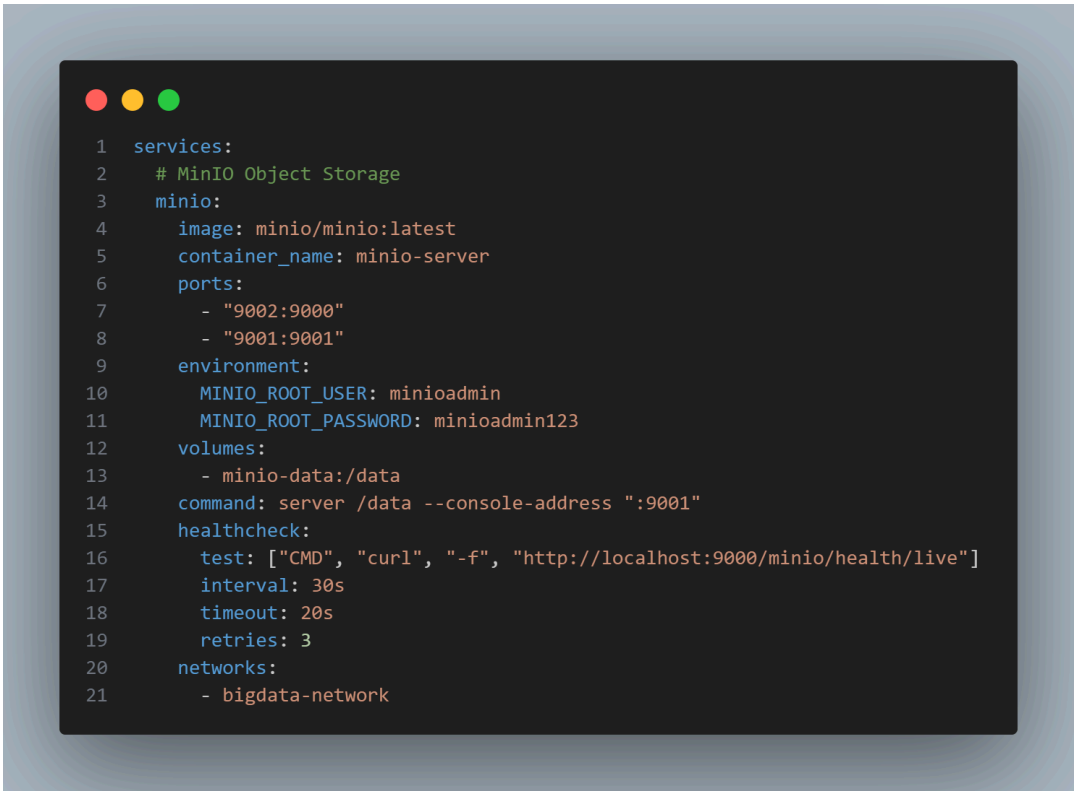
- Secure storage for raw (Bronze) and processed (Silver/Gold) data.

- Easy integration with Python for automated uploads.
- Acts as an intermediary before moving cleaned data into HDFS.

4. Docker Setup

Docker containers are used to encapsulate all services, ensuring reproducibility and isolation. The project uses:

- **MinIO Container:** Hosts Bronze, Silver, and Gold buckets.



```yaml
services:
  # MinIO Object Storage
  minio:
    image: minio/minio:latest
    container_name: minio-server
    ports:
      - "9002:9000"
      - "9001:9001"
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin123
    volumes:
      - minio-data:/data
    command: server /data --console-address ":9001"
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 20s
      retries: 3
    networks:
      - bigdata-network
```

- **HDFS NameNode Container:** Manages HDFS metadata.

```yaml
# HDFS NameNode
  namenode:
    image: bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8
    container_name: hdfs-namenode
    ports:
      - "9870:9870"      # NameNode Web UI
      - "9000:9000"      # HDFS API (exposed on different internal port)
    environment:
      - CLUSTER_NAME=bigdata-cluster
      - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
      - CORE_CONF_hadoop_http_staticuser_user=root
      - HDFS_CONF_dfs_replication=1
      - HDFS_CONF_dfs_namenode_datanode_registration_ip___hostname___check=false
    volumes:
      - hadoop-namenode:/hadoop/dfs/name
    networks:
      - bigdata-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9870"]
      interval: 30s
      timeout: 10s
      retries: 3
```

- **HDFS DataNode Container:** Stores actual data blocks.

```yaml
# HDFS DataNode
 datanode:
    image: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8
    container_name: hdfs-datanode
    ports:
      - "9864:9864"      # DataNode Web UI
    environment:
      - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
      - CORE_CONF_hadoop_http_staticuser_user=root
      - HDFS_CONF_dfs_replication=1
    volumes:
      - hadoop-datanode:/hadoop/dfs/data
    networks:
      - bigdata-network
    depends_on:
      - namenode
```

Benefits of Docker in the project:

- Simplifies setup of complex distributed systems.

- Ensures consistent environment across machines.
- Allows running HDFS and MinIO locally without manual installations.

5. Data Flow and Upload Process

1. **Raw data upload (Bronze):**
   CSV files (london_weather_dataset2.csv, london_traffic_dataset2.csv) are uploaded to the MinIO Bronze bucket using Python scripts with the MinIO SDK.

2. **Cleaned data upload (Silver):**
   Processed Parquet files (weather_cleaned.parquet, traffic_cleaned.parquet, merged_data (3).parquet) are uploaded to the MinIO Silver bucket.

3. **Transfer to HDFS:**
   Data is copied from Silver to HDFS in three directories:
   - /bigdata/weather
   - /bigdata/traffic
   - /bigdata/merged

4. **Method used:** Python automates Docker commands to execute native Hadoop CLI (hdfs dfs -put) inside the HDFS NameNode container. This avoids WebHDFS dependency and ensures reliable ingestion in the containerized environment.

```python
# Create HDFS directories first
print("Creating HDFS directory structure...")
directories = ["/bigdata", "/bigdata/weather", "/bigdata/traffic", "/bigdata/merged"]

for directory in directories:
    mkdir_cmd = f'docker exec hdfs-namenode hdfs dfs -mkdir -p {directory}'
    result = subprocess.run(mkdir_cmd, shell=True, capture_output=True, text=True)
    if result.returncode == 0:
        print(f"Created/verified: {directory}")
    else:
        if "File exists" not in result.stderr:
            print(f"Warning for {directory}: {result.stderr.strip()}")

```

```python
try:
        # Step 1: Copy file into container
        temp_name = local_file.replace(" ", "_").replace("(", "").replace(")", "")
        copy_cmd = f'docker cp "{local_path}" hdfs-namenode:/tmp/{temp_name}'
        result = subprocess.run(copy_cmd, shell=True, capture_output=True, text=True)

        if result.returncode != 0:
            print(f" Failed to copy to container: {result.stderr}")
            continue

        # Step 2: Put file into HDFS
        put_cmd = f'docker exec hdfs-namenode hdfs dfs -put -f /tmp/{temp_name} {hdfs_path}'
        result = subprocess.run(put_cmd, shell=True, capture_output=True, text=True)

        if result.returncode != 0:
            print(f" Failed to upload to HDFS: {result.stderr}")
            continue

        # Step 3: Clean up temp file
        clean_cmd = f'docker exec hdfs-namenode rm /tmp/{temp_name}'
        subprocess.run(clean_cmd, shell=True, capture_output=True, text=True)

        print(f" Uploaded successfully")
        success_count += 1

    except Exception as e:
        print(f"Error: {e}")

```

5. **Results upload (Gold):**
   Analysis outputs (CSV and DOCX) are uploaded to MinIO Gold bucket for further use or reporting.

6. Python Automation

Python is used as an **orchestration layer** to:

- Upload files to MinIO buckets.
- Automate creation of HDFS directories and file uploads.
- Verify bucket contents and HDFS structure.
- Scripted in complete_setup.py for the full pipeline.

The **actual data movement in HDFS** is performed by native hdfs dfs commands.

7. Security Considerations

1. MinIO Security:

- **Authentication:** MinIO access is protected using an access key (MINIO_ACCESS_KEY) and a secret key (MINIO_SECRET_KEY). Only

authorized users can upload or read data from Bronze, Silver, or Gold buckets.

- **Bucket Isolation:** Data is organized in separate buckets (bronze, silver, gold) to prevent accidental data mix-up and allow fine-grained access control.

- **HTTPS (optional):** MinIO supports TLS/SSL for secure communication (though in the local Docker setup we used HTTP for simplicity).

2. HDFS Security:

- **User Isolation:** All HDFS operations are executed under the user root to maintain control over file ownership and permissions.

- **Directory Structure:** Cleaned data is stored in dedicated HDFS directories (/bigdata/weather, /bigdata/traffic, /bigdata/merged), reducing risk of overwriting unrelated data.

- **Overwrite Control:** The hdfs dfs -put -f command ensures only intended files are overwritten.

3. Docker Security:

- **Container Isolation:** Each service runs in its own container (minio-server, hdfs-namenode, hdfs-datanode), isolating processes and reducing cross-service interference.

- **Network Segmentation:** All containers communicate via a dedicated Docker network (bigdata-network), avoiding exposure to other host services.

4. Data Integrity:

- **Checks during upload:** Scripts verify the existence of files before upload and confirm successful transfer.

- **Verification phase:** The verify_setup() script ensures all buckets and HDFS directories contain the expected files, helping detect corruption or misplacement early.

5. Benefits Realized:

- Secure, authenticated access to sensitive datasets.

- Clear separation between raw, processed, and result data layers.

- Safe and reproducible environment thanks to Docker containerization.

8. Project Architecture

Local Data Directory

▼

MinIO (Bronze/Silver/Gold)

▼

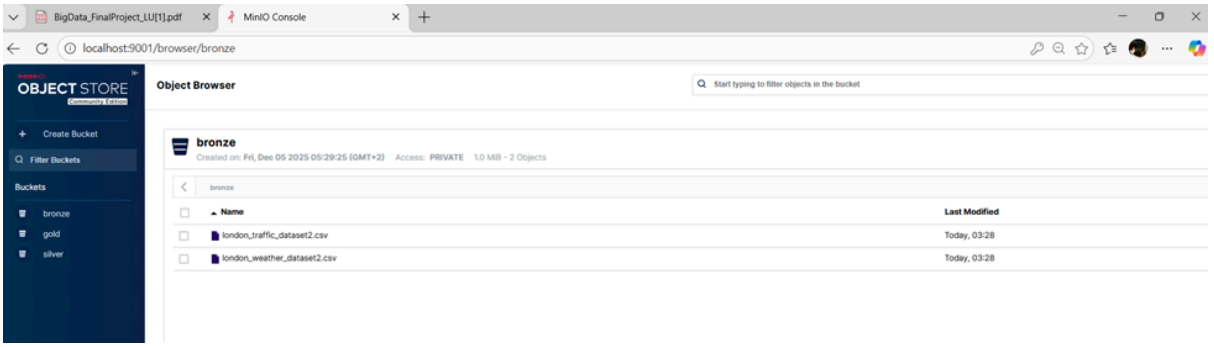
HDFS NameNode / DataNode

▼

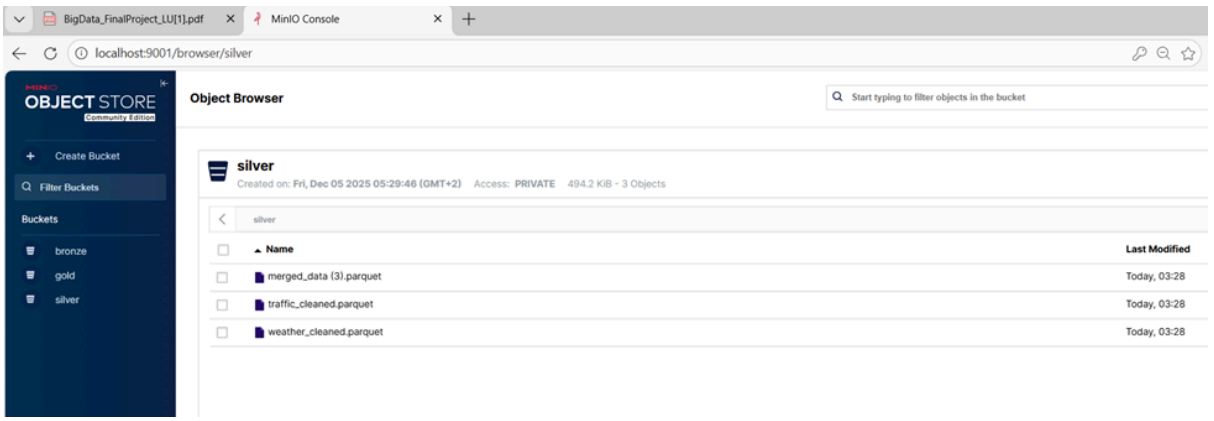
Big Data Analysis & Gold Layer

Key points:

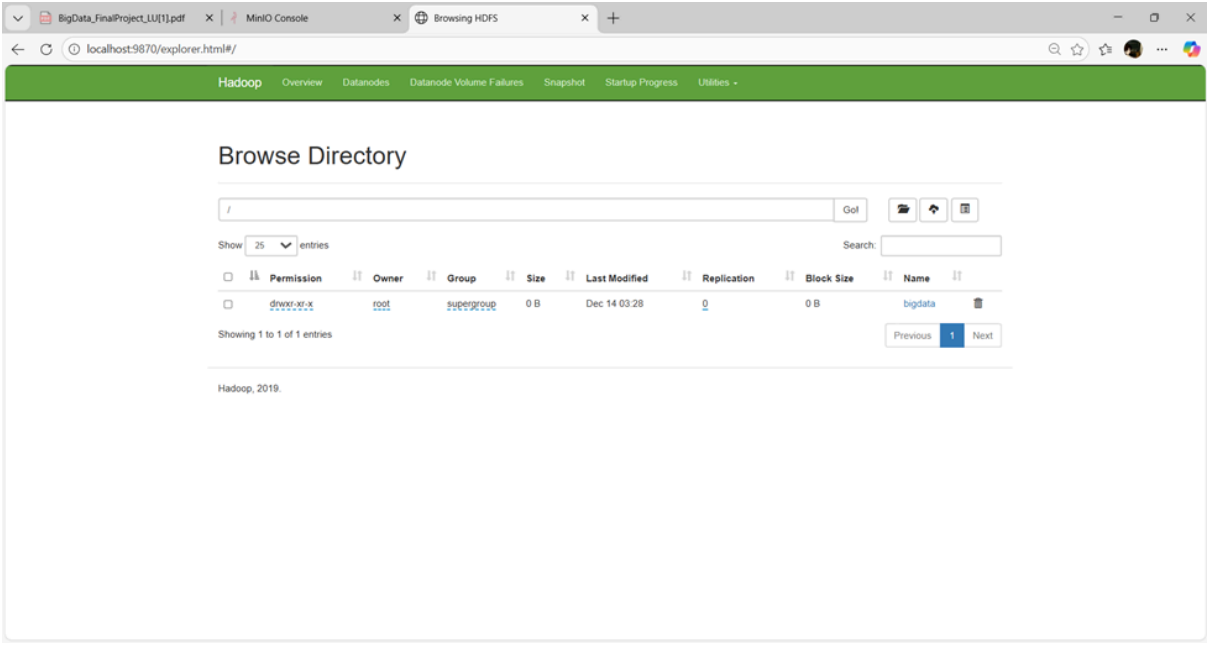
- Bronze: Raw CSV

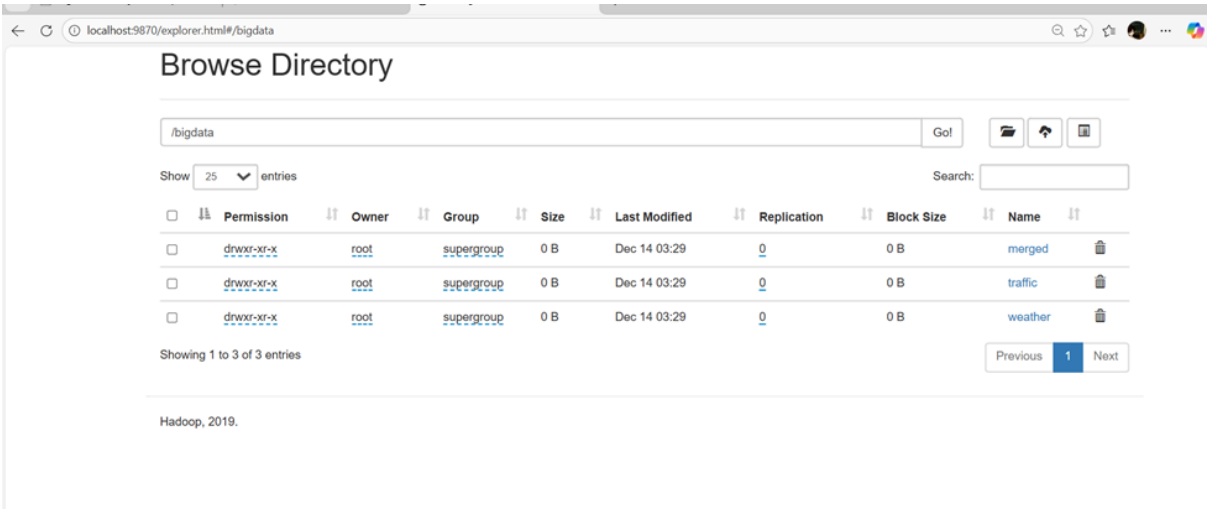


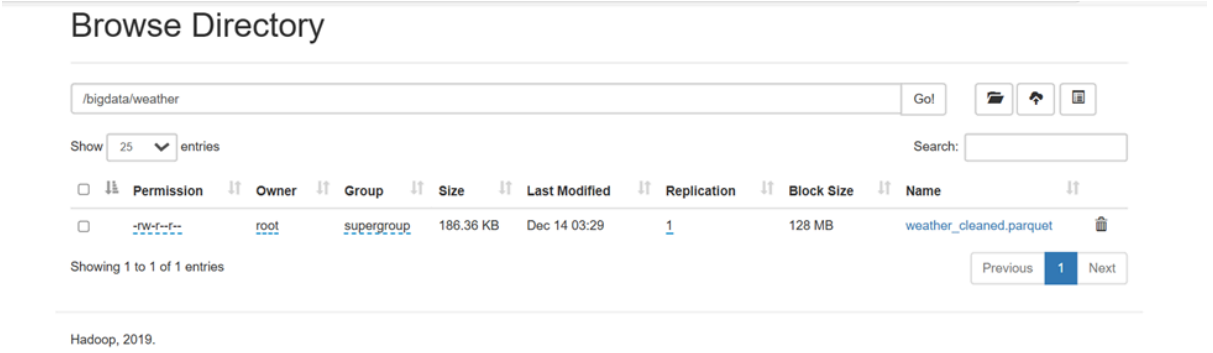
- Silver: Cleaned Parquet

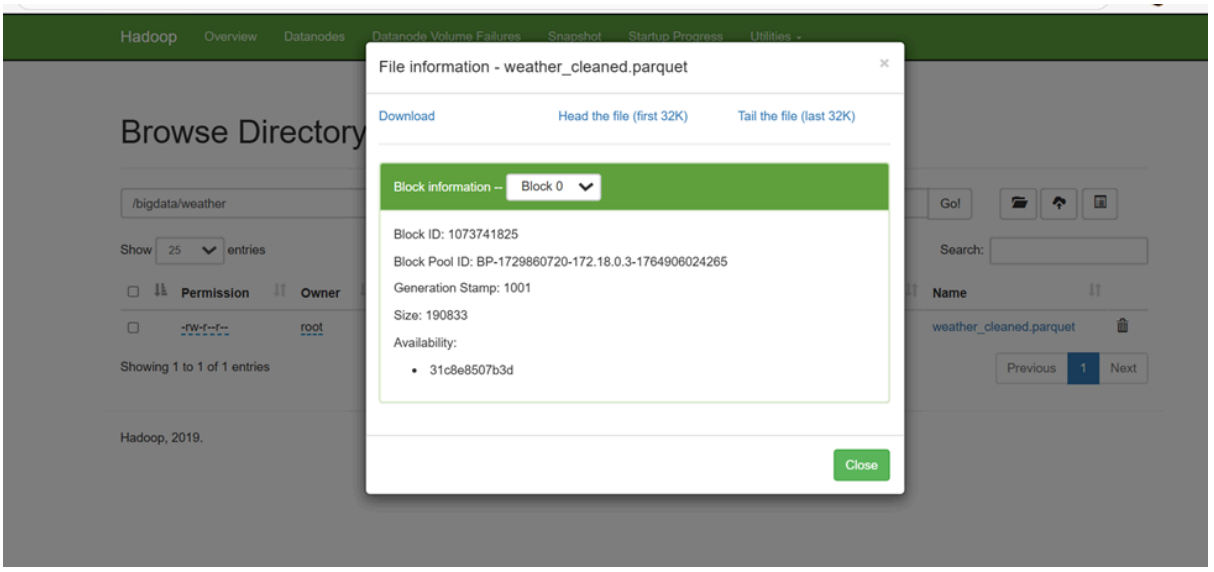


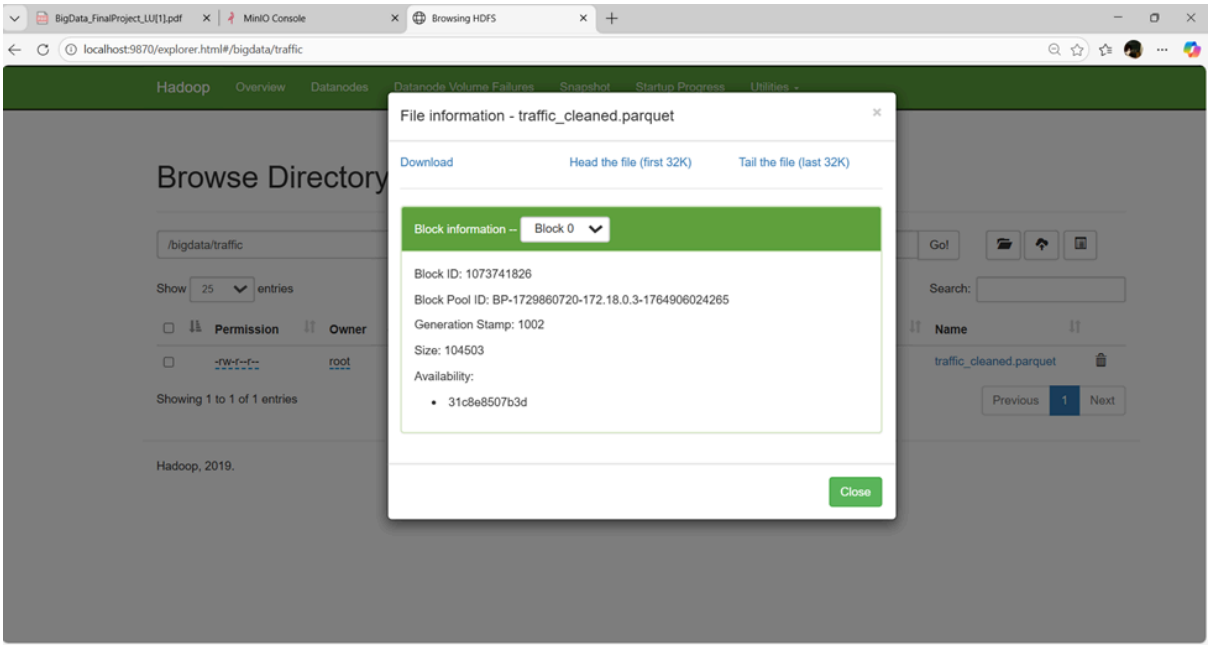
- HDFS: Distributed storage for Silver data

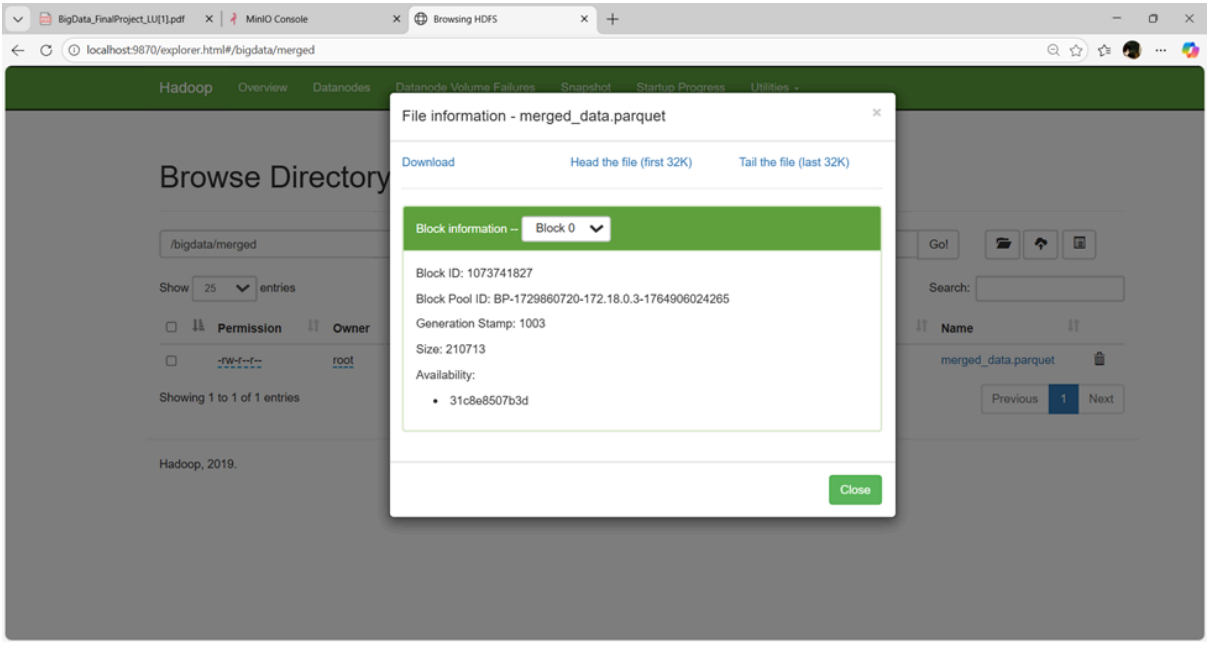




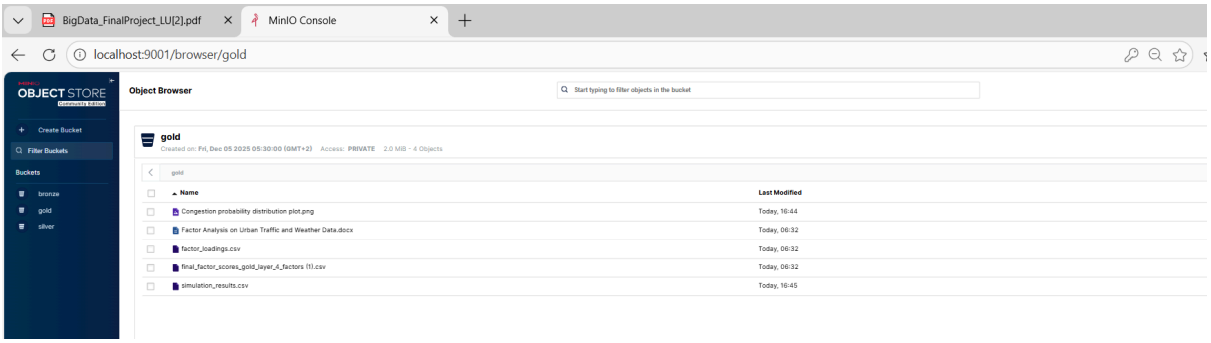



Hadoop
Overview
Datanodes
Datanode Volume Failures
Snapshot
Startup Progress
Utilities
Browse Directory
/bigdata/weather
Go!
Show 25 entries
Search:
Permission
Owner
Name
-rw-r--r--
root
weather_cleaned.parquet
Showing 1 to 1 of 1 entries
Previous
1
Next
Hadoop, 2019.
File information - weather_cleaned.parquet
Download
Head the file (first 32K)
Tail the file (last 32K)
Block information --
Block 0
Block ID: 1073741825
Block Pool ID: BP-1729860720-172.18.0.3-1764906024265
Generation Stamp: 1001
Size: 190833
Availability:
31c8e8507b3d
Close


BigData_FinalProject_LU[1].pdf
MinIO Console
Browsing HDFS
localhost:9870/explorer.html#/bigdata/traffic
Hadoop
Overview
Datanodes
Datanode Volume Failures
Snapshot
Startup Progress
Utilities
Browse Directory
/bigdata/traffic
Go!
Show 25 entries
Search:
Permission
Owner
Name
-rw-r--r--
root
traffic_cleaned.parquet
Showing 1 to 1 of 1 entries
Previous
1
Next
Hadoop, 2019.
File information - traffic_cleaned.parquet
Download
Head the file (first 32K)
Tail the file (last 32K)
Block information --
Block 0
Block ID: 1073741826
Block Pool ID: BP-1729860720-172.18.0.3-1764906024265
Generation Stamp: 1002
Size: 104503
Availability:
31c8e8507b3d
Close



- Gold: Analysis results



- Python: Automation & orchestration
- Docker: Encapsulation & reproducibility
- MinIO: Local object storage simulation

9. Summary

This setup demonstrates a **complete Data Lake workflow**:

- Data ingestion from raw CSV to object storage (MinIO).
- Data cleaning and storage in Parquet format.

- Reliable ingestion into HDFS for scalable processing.
- Automated verification to ensure all components function correctly.

Outcome: All datasets are securely stored, organized, and ready for further big data analysis, with Docker ensuring the project is fully reproducible and portable.

Data Preprocessing

Inspect Initial Dataset

```python
def analyze_dataset_before_cleaning(df, dataset_name):
    """Analyze dataset before cleaning"""
    print(f"ANALYZING {dataset_name.upper()} BEFORE CLEANING")
    print('='*60)

    print(f"Total rows: {len(df)}")
    print(f"Duplicate rows: {df.duplicated().sum()}")
    print(f"Total missing values: {df.isnull().sum().sum()}")

    # Analyze each column
    for column in df.columns:
        print(f"\n{column}:")
        print(f"  Type: {df[column].dtype}")
        print(f"  Missing: {df[column].isnull().sum()} ({(df[column].isnull().sum() / len(df)) * 100:.1f}%)")

        if df[column].dtype in ['int64', 'float64']:
            print(f"  Min: {df[column].min():.2f}")
            print(f"  Max: {df[column].max():.2f}")
            print(f"  Mean: {df[column].mean():.2f}")
            print(f"  Std: {df[column].std():.2f}")

        if df[column].dtype == 'object':
            unique_count = df[column].nunique()
            print(f"  Unique values: {unique_count}")
            if unique_count < 20:  # Show unique values for small sets
                print(f"  Values: {df[column].dropna().unique()[:10]}")

    return df
```

1. Weather Dataset Analysis & Cleaning

1.1 Initial Data Quality

Dataset:

- Total rows: 5,000
- Duplicate rows: 0
- Total missing values: 973 (across all columns)

Column Analysis:

Column	Data Type	Missing	Range/Values	Issues Identified
weather_id	float64	47 (0.9%)	5001-9900	Nulls, duplicates
date_time	object	53 (1.1%)	4,886 unique	Mixed formats
season	object	50 (1.0%)	Winter/Spring/Summer	Nulls
city	object	59 (1.2%)	London	Nulls
temperature_c	float64	104 (2.1%)	-30°C to 60°C	Extreme outliers
humidity	float64	85 (1.7%)	-10% to 150%	Invalid range
rain_mm	float64	85 (1.7%)	0-120mm	Extreme values
wind_speed_kmh	float64	92 (1.8%)	0.16-250 km/h	Unrealistic peaks
visibility_m_weather	object	104 (2.1%)	Mixed	String contamination
weather_condition	object	48 (1.0%)	5 categories	Nulls
air_pressure_hpa	float64	246 (4.9%)	978-1045 hPa	Highest missing rate

1.2 Cleaning

Step 1: Date Format Standardization

```
# 1. Fix date format
print("\n1. FIXING DATE FORMATS")
df['date_time'] = pd.to_datetime(df['date_time'], errors='coerce', format='%Y-%m-%d %H:%M:%S')
```

- Converted all dates to datetime format

Step 2: Null Date Removal

```
# 2. check null dates
print("\n2. Remove Null DATES")
null_dates = df['date_time'].isna().sum()
print(f"Null dates before dropping: {null_dates}")

df = df.dropna(subset=['date_time']).reset_index(drop=True)
null_dates_after = df['date_time'].isna().sum()
print(f"Null dates after dropping: {null_dates_after}")
```

- Null dates before: 250
- Null dates after: 0

Step 3: Duplicate Removal

```
# 3. Remove duplicates
print("\n3. REMOVING DUPLICATES")
duplicates = df.duplicated().sum()
df = df.drop_duplicates()
print(f"  Removed {duplicates} duplicate rows")
```

- No complete duplicate rows found (0 removed)

Step 4: Temperature Outlier Correction

```
# 4. Fix temperature outliers
print("\n4. FIXING TEMPERATURE OUTLIERS")
temp_outliers = df[(df['temperature_c'] < -15) | (df['temperature_c'] > 40)].shape[0]
temp_min_before = df['temperature_c'].min()
temp_max_before = df['temperature_c'].max()

df['temperature_c'] = df['temperature_c'].clip(-15, 40)

temp_min_after = df['temperature_c'].min()
temp_max_after = df['temperature_c'].max()

print(f"  Fixed {temp_outliers} temperature outliers")
print(f"  Range before: {temp_min_before:.1f} to {temp_max_before:.1f}°C")
print(f"  Range after:  {temp_min_after:.1f} to {temp_max_after:.1f}°C")
```

- Applied realistic bounds for London climate: -15°C to 40°C
- Fixed 95 temperature outliers
- Range before: -30.0°C to 60.0°C
- Range after: -15.0°C to 40.0°C

Step 5: Humidity Correction

```
# 5. Fix humidity (0-100%)
print("\n5. FIXING HUMIDITY VALUES")
hum_outliers = df[(df['humidity'] < 0) | (df['humidity'] > 100)].shape[0]
hum_min_before = df['humidity'].min()
hum_max_before = df['humidity'].max()

df['humidity'] = df['humidity'].clip(0, 100)

hum_min_after = df['humidity'].min()
hum_max_after = df['humidity'].max()

print(f"  Fixed {hum_outliers} humidity outliers")
print(f"  Range before: {hum_min_before:.1f} to {hum_max_before:.1f}%")
print(f"  Range after:  {hum_min_after:.1f} to {hum_max_after:.1f}%")
```

- Clipped values to valid percentage range: 0-100%
- Fixed 94 humidity outliers
- Range before: -10.0% to 150.0%
- Range after: 0.0% to 100.0%

Step 6: Missing Value Imputation

```
# 6. Handle missing values
print("\n6. HANDLING MISSING VALUES")
# Keep only rows where weather_id is NOT null
df = df[df['weather_id'].notna()]

# Replace 'Unknown' with NaN first
columns_to_process = [col for col in df.columns if col != 'weather_id']
df[columns_to_process] = df[columns_to_process].replace('Unknown', pd.NA)

# Then impute missing values
for col in columns_to_process:
    missing_before = df[col].isnull().sum()
    if missing_before > 0:
        df[col] = df[col].fillna(df[col].mode().iloc[0])

        missing_after = df[col].isnull().sum()
        print(f"  {col}: {missing_before} missing → {missing_after} missing")
```

- Removed rows where weather_id was null (primary key)
- Replaced "Unknown" strings with NaN
- Applied mode imputation for remaining missing values

Column	Missing Before	Missing After	Imputation Method

season	47	0	Mode
city	54	0	Set to "London"
temperature_c	97	0	Mode
humidity	82	0	Mode
rain_mm	81	0	Mode
wind_speed_kmh	89	0	Mode
visibility_m_weather	146	0	Mode
weather_condition	44	0	Mode
air_pressure_hpa	236	0	Mode

Step 7: Data Type Standardization

```
# 7. Handle Data types
print("\n7. FINAL DATA TYPES")

# Ensure 'city' is properly handled - set to 'London' for all rows
df['city'] = 'London'
df['temperature_c'] = df['temperature_c'].round(1)
df['humidity'] = df['humidity'].round().astype(int)
df['rain_mm'] = df['rain_mm'].round(1)
df['visibility_m_weather'] = df['visibility_m_weather'].astype(float)

# Display final data types
print(df.dtypes)
```

- Rounded temperature to 1 decimal place
- Converted humidity to integer
- Rounded rain to 1 decimal place
- Converted visibility to float

Step 8: Final Visibility Correction

```
# 8. Fix Visibility values
df['visibility_m_weather'] = df['visibility_m_weather'].clip(lower=0)
```

- Applied lower bound of 0 meters (no negative visibility)

1.3 Weather Cleaning Results

Final Statistics:

- Original rows: 5,000
- Final rows: 4,704
- Rows removed: 296
- Data retained: **94.1%**

Final Data Types:

```
weather_id                 float64
date_time           datetime64[ns]
season                      object
city                        object
temperature_c              float64
humidity                     int64
rain_mm                    float64
wind_speed_kmh             float64
visibility_m_weather       float64
weather_condition           object
air_pressure_hpa           float64
dtype: object
```

2. Traffic Dataset Analysis & Cleaning

2.1 Initial Data Quality

Dataset:

- Total rows: 5,000
- Duplicate rows: 0
- Total missing values: 693 (across all columns)

Column Analysis:

Column	Data Type	Missing	Range/Values	Issues Identified
traffic_id	float64	48 (1.0%)	9001-13900	Nulls, duplicates
date_time	object	56 (1.1%)	4,874 unique	Mixed formats
city	object	51 (1.0%)	London	Nulls
area	object	48 (1.0%)	7 districts	Nulls
vehicle_count	float64	108 (2.2%)	45.9-25,000	Extreme outliers
avg_speed_kmh	float64	81 (1.6%)	-50 to 64.85	Negative values
accident_count	float64	46 (0.9%)	0-55	Unrealistic peaks
congestion_level	object	45 (0.9%)	High/Medium/Low	Nulls
road_condition	object	112 (2.2%)	Dry/Wet/Snowy	Highest missing
visibility_m_traffic	float64	98 (2.0%)	-34 to 10,099m	Negative values

2.2 Cleaning

Step 1: Date Format Standardization

```
# 1. Fix date format
print("\n1. FIXING DATE FORMATS")
df['date_time'] = pd.to_datetime(df['date_time'], errors='coerce', format='%Y-%m-%d %H:%M:%S')
```

- Same process as weather dataset
- Result: 250 invalid dates identified

Step 2: Null Date Removal

```python
# 2. check null dates
print("\n2. Remove Null DATES")
null_dates = df['date_time'].isna().sum()
print(f"Null dates before dropping: {null_dates}")

df = df.dropna(subset=['date_time']).reset_index(drop=True)
null_dates_after = df['date_time'].isna().sum()
print(f"Null dates after dropping: {null_dates_after}")
```

- Null dates before: 250
- Null dates after: 0

Step 3: Duplicate Removal

```python
# 3. Remove duplicates
print("\n3. REMOVING DUPLICATES")
duplicates = df.duplicated().sum()
df = df.drop_duplicates()
print(f"  Removed {duplicates} duplicate rows")
```

- No complete duplicate rows found (0 removed)

Step 4: Speed Outlier Removal (IQR Method)

```python
# 4. Fix speed outliers
print("\n4. FIXING SPEED VALUES")

len_before = len(df)
Q1 = df['avg_speed_kmh'].quantile(0.25)
Q3 = df['avg_speed_kmh'].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter
df = df[
    (df['avg_speed_kmh'] >= lower_bound) &
    (df['avg_speed_kmh'] <= upper_bound)
]

print("Removed rows:", len_before - len(df))
```

- Applied Interquartile Range (IQR) method for outlier detection
- Calculated Q1 (25th percentile) and Q3 (75th percentile)
- Defined bounds: Q1 - 1.5×IQR to Q3 + 1.5×IQR
- **Removed 128 rows** with unrealistic speeds (including negative values)

Step 5: Vehicle Count Outlier Removal (IQR Method)

```python
# 5. Fix vehicle count
print("\n5. FIXING VEHICLE COUNT")

len_before = len(df)
Q1 = df['vehicle_count'].quantile(0.25)
Q3 = df['vehicle_count'].quantile(0.75)
IQR = Q3 - Q1

lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter
df = df[
    (df['vehicle_count'] >= lower_bound) &
    (df['vehicle_count'] <= upper_bound)
]

print("Removed rows:", len_before - len(df))
```

- Applied IQR
- **Removed 162 rows** with extreme vehicle counts

Step 6: Missing Value Imputation

```python
# 6. Handle missing values
print("\n7. HANDLING MISSING VALUES")
# Keep only rows where weather_id is NOT null
df = df[df['traffic_id'].notna()]

# Replace 'Unknown' with NaN first
columns_to_process = [col for col in df.columns if col != 'traffic_id']
df[columns_to_process] = df[columns_to_process].replace('Unknown', pd.NA)

# Then impute missing values
for col in columns_to_process:
    missing_before = df[col].isnull().sum()
    if missing_before > 0:
        df[col] = df[col].fillna(df[col].mode().iloc[0])

        missing_after = df[col].isnull().sum()
        print(f"  {col}: {missing_before} missing → {missing_after} missing")
```

- Kept only rows where traffic_id was not null
- Applied mode imputation for categorical and numerical fields

Column	Missing Before	Missing After	Imputation Method
city	40	0	Set to "London"

area	43	0	Mode
accident_count	43	0	Mode
congestion_level	41	0	Mode
road_condition	101	0	Mode
visibility_m_traffic	85	0	Mode

Step 7: Data Type Standardization

```
# 7. Handle Data types
print("\n8. FINAL DATA TYPES")
df['city'] = 'London'
df['vehicle_count'] = df['vehicle_count'].astype(int)
df['accident_count'] = df['accident_count'].astype(int)
df['avg_speed_kmh'] = df['avg_speed_kmh'].round(1)
df['visibility_m_traffic'] = df['visibility_m_traffic'].astype(float)


# Display final data types
print(df.dtypes)
```

- Ensured city is "London" for all rows
- Converted vehicle_count to integer
- Converted accident_count to integer
- Rounded avg_speed_kmh to 1 decimal
- Converted visibility to float

Step 8: Visibility Correction

```
# 8. Fix Visibility values
df['visibility_m_traffic'] = df['visibility_m_traffic'].clip(lower=0)
```

- Clipped to minimum of 0 meters

Step 9: Speed Correction

```
# 9. Fix Speed values
df['avg_speed_kmh'] = df['avg_speed_kmh'].clip(lower=0)
```

- Clipped to minimum of 0

2.3 Traffic Cleaning Results

Final Statistics:

- Original rows: 5,000
- Final rows: 4,420
- Rows removed: 580
- Data retained: **88.4%**

Final Data Types:

```
traffic_id                    float64
date_time              datetime64[ns]
city                          object
area                          object
vehicle_count                  int64
avg_speed_kmh                float64
accident_count                 int64
congestion_level              object
road_condition                object
visibility_m_traffic         float64
dtype: object
```

3. Key Findings & Observations

3.1 Data Loss Analysis

Weather Dataset:

- 5.9% data loss (296 rows)
- Primary causes: Invalid dates, null primary keys.

Traffic Dataset:

- 11.6% data loss (580 rows)
- Primary causes: Invalid dates, speed outliers and vehicle count outliers .

3.2 Data Quality Improvements

Before Cleaning:

- Mixed date formats
- Invalid ranges
- Missing values
- Outliers skewing distributions and means

After Cleaning:

- Standardized datetime format
- Realistic value ranges for all measurements
- Complete records (no missing values)

4. Output Files Generated

```
: weather_cleaned.to_parquet('weather_cleaned.parquet', index=False)
  traffic_cleaned.to_parquet('traffic_cleaned.parquet', index=False)
```

1. weather_cleaned.parquet - Cleaned weather data (4,704 rows)
2. traffic_cleaned.parquet - Cleaned traffic data (4,420 rows)

Datasets Merging :

1- Defining The function `check_merge_ready` that performs a quick check on a DataFrame before merging :

1. **Check that key columns (`key_columns`) exist in the DataFrame.**
2. **Check for missing values in the key columns.**
3. **Check for duplicate rows in the DataFrame.**
4. **Verify the `date_time` column type (if present) is datetime.**
5. **Standardize the `city` column (convert to string, strip spaces, title case) and show unique values.**

The main goal is to ensure the dataset is merge-ready without issues in columns, values, or formatting.

Data cleaning validation before merging

```python
def check_merge_ready(df, dataset_name, key_columns):
    print(f"\nChecking {dataset_name} for merge readiness...")
    print("-" * 50)

    # 1. Check key columns exist
    for col in key_columns:
        if col not in df.columns:
            print(f"Missing key column: {col}")
        else:
            print(f"Column '{col}' exists")

    # 2. Check for missing values in key columns
    for col in key_columns:
        missing = df[col].isnull().sum()
        if missing > 0:
            print(f"Column '{col}' has {missing} missing values")
        else:
            print(f"Column '{col}' has no missing values")

    # 3. Check duplicates
    dup_count = df.duplicated().sum()
    if dup_count > 0:
        print(f"Dataset has {dup_count} duplicate rows")
    else:
        print("No duplicate rows")

    # 4. Check date_time type if in keys
    if 'date_time' in key_columns:
        if not pd.api.types.is_datetime64_any_dtype(df['date_time']):
            print("'date_time' column is NOT datetime type")
        else:
            print("'date_time' column is datetime type")

    # 5. Standardize city column if in keys
    if 'city' in key_columns:
        df['city'] = df['city'].astype(str).str.strip().str.title()
        unique_cities = df['city'].unique()
        print(f"'city' standardized. Unique cities: {unique_cities}")
```

After running it we found :

- **Both date_time and city columns exist in each dataset.**
- **There are no missing values in the key columns.**
- **There are no duplicate rows.**
- **The date_time column is confirmed to be datetime type.**
- **The city column has been standardized (converted to title case), and both datasets only contain ['London'] as the unique city.**

```python
# Define merge keys
merge_keys = ['date_time', 'city']

# Run checks
check_merge_ready(weather_cleaned, "Weather Dataset", merge_keys)
check_merge_ready(traffic_cleaned, "Traffic Dataset", merge_keys)
```

```
Checking Weather Dataset for merge readiness...
------------------------------------------------
Column 'date_time' exists
Column 'city' exists
Column 'date_time' has no missing values
Column 'city' has no missing values
No duplicate rows
'date_time' column is datetime type
'city' standardized. Unique cities: ['London']

Checking Traffic Dataset for merge readiness...
------------------------------------------------
Column 'date_time' exists
Column 'city' exists
Column 'date_time' has no missing values
Column 'city' has no missing values
No duplicate rows
'date_time' column is datetime type
'city' standardized. Unique cities: ['London']
```

We converted the `date_time` columns in both the `weather_cleaned` and `traffic_cleaned` DataFrames into datetime objects using `pd.to_datetime()`. Then, they round down each timestamp to the nearest hour using `.dt.floor('H')`. This ensures that all timestamps are aligned at the hour level, which is useful for merging.

```python
weather_cleaned['date_time'] = pd.to_datetime(weather_cleaned['date_time']).dt.floor('H')
traffic_cleaned['date_time'] = pd.to_datetime(traffic_cleaned['date_time']).dt.floor('H')
```

Finally we merged the 2 datasets on “date_time” and “city” columns

```python
merged_df = pd.merge(
    traffic_cleaned,
    weather_cleaned,
    on=['city', 'date_time'],
    how='inner'
)

merged_df
```

Python

	traffic_id	date_time	city	area	vehicle_count	avg_speed_kmh	accident_count	congestion_level	road_condition	visibility_m_traffic	weather_id	season
0	9002.0	2020-01-01 01:00:00	London	Westminster	267	43.9	0	Medium	Dry	9947.0	5002.0	Winter
1	9003.0	2020-01-01 02:00:00	London	Camden	188	59.4	0	Low	Dry	10072.0	5003.0	Winter
2	9004.0	2020-01-01 04:00:00	London	Greenwich	226	47.4	0	Low	Dry	10047.0	5005.0	Winter
3	9005.0	2020-01-01 04:00:00	London	Southwark	340	47.3	0	Low	Dry	10080.0	5005.0	Winter
4	9006.0	2020-01-01 06:00:00	London	Westminster	350	39.8	0	Medium	Wet	5534.0	5007.0	Winter

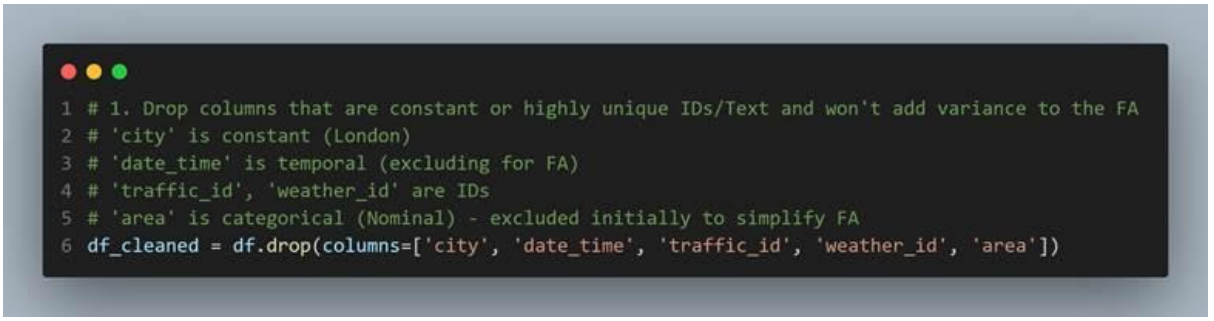
Factor Analysis on Urban Traffic and Weather Data

The goal of this project is to analyze urban traffic data in London under varying weather conditions, using factor analysis to reduce dimensionality and identify latent factors that influence traffic behavior. The dataset contains 4162 records with 15 initial features covering traffic volume, road and weather conditions, and temporal information.

1-Data Cleaning and Preprocessing

Initial Cleaning

- *Removed non-informative columns:* city, date_time, traffic_id, weather_id, area.



```
# 1. Drop columns that are constant or highly unique IDs/Text and won't add variance to the FA
# 'city' is constant (London)
# 'date_time' is temporal (excluding for FA)
# 'traffic_id', 'weather_id' are IDs
# 'area' is categorical (Nominal) - excluded initially to simplify FA
df_cleaned = df.drop(columns=['city', 'date_time', 'traffic_id', 'weather_id', 'area'])
```

- *Checked for nulls; dataset contains no missing values after cleaning.*

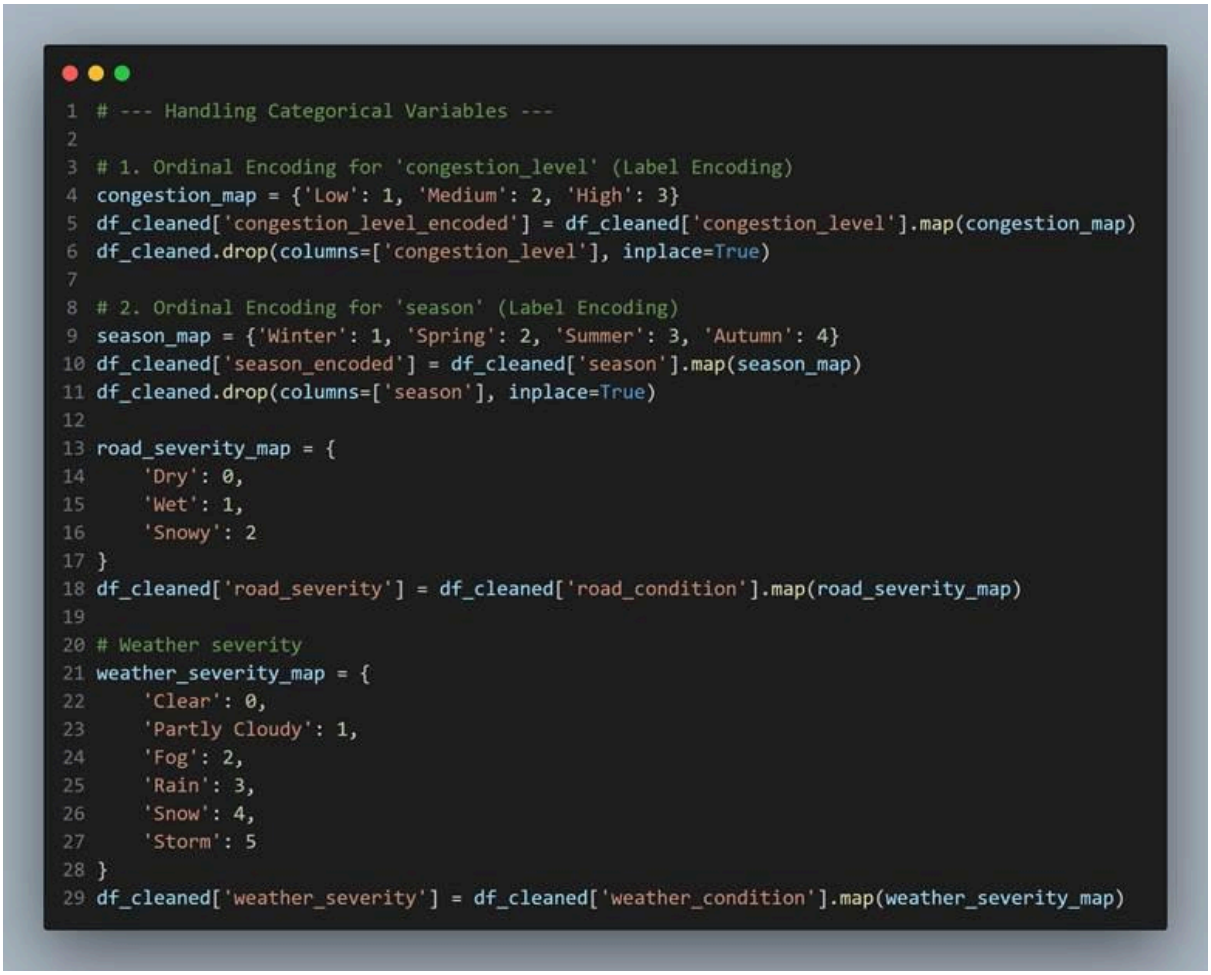
Columns after cleaning (15 features):

```
[30]
✓ 0s
df_cleaned.columns

Index(['vehicle_count', 'avg_speed_kmh', 'accident_count',
       'visibility_m_traffic', 'temperature_c', 'humidity', 'rain_mm',
       'wind_speed_kmh', 'visibility_m_weather', 'air_pressure_hpa', 'hour',
       'congestion_level_encoded', 'season_encoded', 'road_severity',
       'weather_severity'],
      dtype='object')
```

2- Encoding Categorical Variables

- congestion_level → ordinal: Low=1, Medium=2, High=3
- season → ordinal: Winter=1, Spring=2, Summer=3, Autumn=4
- road_condition → ordinal: Dry=0, Wet=1, Snowy=2
- weather_condition → ordinal: Clear=0, Partly Cloudy=1, Fog=2, Rain=3, Snow=4, Storm=5

- 

```python
# --- Handling Categorical Variables ---

# 1. Ordinal Encoding for 'congestion_level' (Label Encoding)
congestion_map = {'Low': 1, 'Medium': 2, 'High': 3}
df_cleaned['congestion_level_encoded'] = df_cleaned['congestion_level'].map(congestion_map)
df_cleaned.drop(columns=['congestion_level'], inplace=True)

# 2. Ordinal Encoding for 'season' (Label Encoding)
season_map = {'Winter': 1, 'Spring': 2, 'Summer': 3, 'Autumn': 4}
df_cleaned['season_encoded'] = df_cleaned['season'].map(season_map)
df_cleaned.drop(columns=['season'], inplace=True)

road_severity_map = {
    'Dry': 0,
    'Wet': 1,
    'Snowy': 2
}
df_cleaned['road_severity'] = df_cleaned['road_condition'].map(road_severity_map)

# Weather severity
weather_severity_map = {
    'Clear': 0,
    'Partly Cloudy': 1,
    'Fog': 2,
    'Rain': 3,
    'Snow': 4,
    'Storm': 5
}
df_cleaned['weather_severity'] = df_cleaned['weather_condition'].map(weather_severity_map)
```

All original categorical columns were dropped after encoding.

```python
df_cleaned = df_cleaned.drop(columns=['road_condition', 'weather_condition'])
```

Final dataset features (15 numeric columns):

Columns after Encoding:

['vehicle_count', 'avg_speed_kmh', 'accident_count', 'visibility_m_traffic', 'temperature_c', 'humidity', 'rain_mm', 'wind_speed_kmh', 'visibility_m_weather', 'air_pressure_hpa', 'hour', 'congestion_level_encoded', 'season_encoded', 'road_severity', 'weather_severity']

3 -Standardization

All features were standardized using StandardScaler to ensure uniform scaling for factor analysis.

```python
scaler = StandardScaler()
df_scaled_array = scaler.fit_transform(df_cleaned)
df_scaled = pd.DataFrame(df_scaled_array, columns=df_cleaned.columns)
```

Example of first 5 rows after scaling:

```
Data Standardization Complete. First 5 rows of scaled data:
   vehicle_count  avg_speed_kmh  accident_count  visibility_m_traffic  \
0      -1.058205       0.961524        -0.14341              0.745965
1      -1.130305       2.268819        -0.14341              0.782496
2      -1.095624       1.256720        -0.14341              0.775190
3      -0.991581       1.248286        -0.14341              0.784834
4      -0.982455       0.615724        -0.14341             -0.543729

   temperature_c  humidity   rain_mm  wind_speed_kmh  visibility_m_weather  \
0      -1.078642 -1.106672 -0.256140        0.001473              0.573310
1      -0.812038 -0.637482 -0.256140       -0.274728              0.573310
2      -1.232992 -0.972617 -0.256140       -0.010498              0.573310
3      -1.232992 -0.972617 -0.256140       -0.010498              0.573310
4      -0.419147  0.099816  0.013793       -0.407209             -1.231632
```

No missing values remained after scaling.

4. Suitability for Factor Analysis

KMO and Bartlett's Test

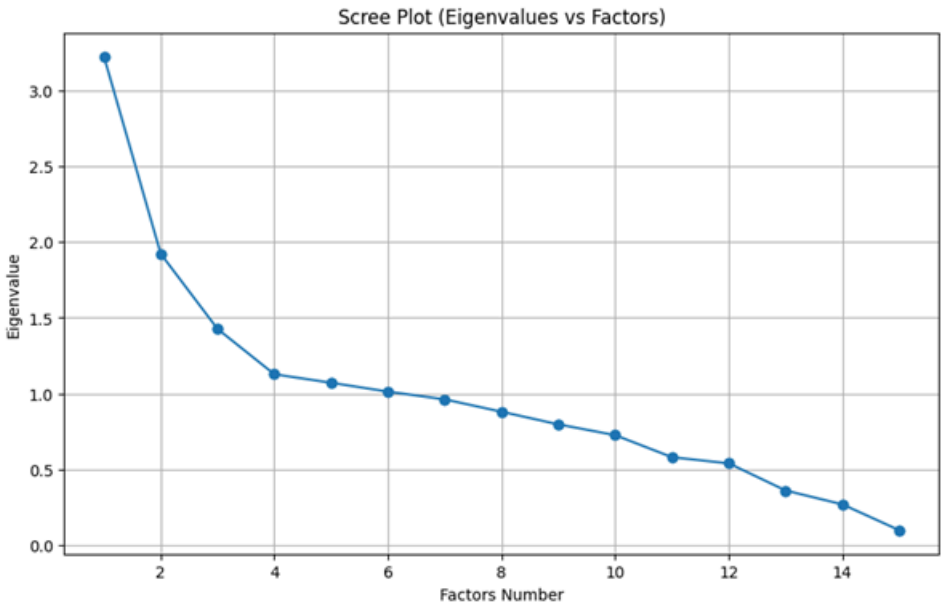
- KMO Measure: 0.646 → acceptable sampling adequacy
- Bartlett's Test: p-value < 0.001 → correlation matrix is not an identity

```python
kmo_all, kmo_model = calculate_kmo(df_scaled)
bartlett_spher, bartlett_p = calculate_bartlett_sphericity(df_scaled)

print("\n--- Suitability Tests for Factor Analysis ---")
print(f"Kaiser-Meyer-Olkin (KMO) Measure: {kmo_model:.3f}")
print(f"Bartlett's Test P-value: {bartlett_p:.3e}")
```

5- Determining Number of Factors

- Eigenvalues >1 suggested 6 factors, but based on interpretability, 4 factors were selected.
- Scree plot confirmed an elbow at factor 4.



6. Factor Analysis (Varimax Rotation)

Performed factor analysis on the standardized dataset with 4 factors using Varimax rotation.

```python
# --- Final Factor Analysis with Varimax Rotation ---

num_factors = 4
fa_final = FactorAnalyzer(n_factors=num_factors, rotation='varimax')
fa_final.fit(df_scaled)

# Factor Loadings DataFrame
factor_loadings = pd.DataFrame(
    fa_final.loadings_,
    index=df_scaled.columns,
    columns=[f'Factor {i+1}' for i in range(num_factors)]
)
```

Factor Loadings (Threshold ≥ 0.4 Highlighted)

	Factor 1	Factor 2	Factor 3	Factor 4
vehicle_count	**0.923000**	-0.058000	0.228000	-0.036000
avg_speed_kmh	**-0.792000**	-0.231000	**0.566000**	0.034000
accident_count	0.001000	-0.005000	-0.065000	0.004000
visibility_m_traffic	-0.112000	-0.256000	**0.634000**	-0.012000
temperature_c	0.146000	-0.020000	0.070000	**0.986000**
humidity	0.016000	**0.462000**	-0.070000	0.033000
rain_mm	-0.001000	0.239000	-0.007000	0.002000
wind_speed_kmh	0.010000	0.081000	0.012000	-0.017000
visibility_m_weather	-0.037000	**-0.404000**	0.219000	-0.039000
air_pressure_hpa	-0.006000	-0.002000	-0.014000	-0.003000
hour	0.392000	0.004000	0.081000	0.141000
congestion_level_encoded	**0.708000**	0.162000	**-0.433000**	-0.028000
season_encoded	-0.001000	-0.004000	-0.011000	**0.405000**
road_severity	0.127000	0.376000	**-0.486000**	-0.028000
weather_severity	0.040000	**0.996000**	-0.062000	0.010000

Interpretation of Factors:

1. Factor 1 – Traffic Load & Congestion:
    - High loadings: vehicle_count, avg_speed_kmh (negative), congestion_level_encoded
    - Represents traffic volume, congestion, and overall traffic stress.
2. Factor 2 – Atmospheric Weather Severity:
    - High loadings: weather_severity, humidity, road_severity
    - Captures weather-related hazards affecting traffic.
3. Factor 3 – Visibility & Flow Quality:
    - High loadings: visibility_m_traffic, avg_speed_kmh, road_severity (negative)
    - Indicates quality of traffic flow and visibility conditions.
4. Factor 4 – Seasonal Temperature Patterns:
    - High loadings: temperature_c, season_encoded
    - Reflects seasonal variations and temperature effects on traffic.

Factor Scores (Gold Layer)

- Factor scores were calculated for each observation using the fitted model.

- Scores were merged with the cleaned dataset to form the Gold Layer, ready for downstream modeling or analysis.

```
First 5 rows of the Gold Layer data (with new factor scores):
   Traffic_Load_Congestion_Score  Atmospheric_Weather_Severity_Score  \
0                      -1.032607                           -0.906444
1                      -1.722059                           -0.735907
2                      -1.253827                           -0.862082
3                      -1.186322                           -0.857412
4                      -1.032318                            0.940379

   Visibility_Flow_Quality_Score  Seasonal_Temperature_Pattern_Score
0                      -0.136459                           -0.929442
1                       1.442984                           -0.641487
2                       0.140467                           -1.094074
3                       0.233843                           -1.107967
4                       0.147177                           -0.228831
```

Conclusion

- Successfully reduced 15 variables into 4 interpretable latent factors.
- The Gold Layer dataset now contains both original features and factor scores, ready for predictive modeling.
- This approach enables dimensionality reduction, clearer understanding of traffic and weather interactions, and supports urban traffic planning under varying conditions.

Monte Carlo Simulation

1.Data Loading and Method

- Method: Probabilistic data-driven sampling. We draw a few rows per iteration, average their values to get a representative condition, then add random noise. Risk is computed via thresholds.

- Why averaging: It reduces outlier influence; the noise reintroduces variability.

```python
df_monte_carlo = pd.read_parquet("merged_data.parquet")
def traffic_jam_risk(temp, rain, humidity, visibility, wind):
    risk = 0

    # Heavy rain
    if rain > 20:
        risk += 0.30

    # Temperature extremes
    if temp < 0 or temp > 35:
        risk += 0.15

    # High humidity
    if humidity > 85:
        risk += 0.15

    # Low visibility
    if visibility < 500:
        risk += 0.30

    # Strong winds
    if wind > 50:
        risk += 0.15

    return min(risk, 1.0)
```

2.Traffic jam Risk Function

- Rule-based: Adds fixed weights when hazardous thresholds are crossed (rain, temperature, humidity, visibility, wind).

- Cap at 1.0: Ensures probability stays within [0, 1].

- Effect: Multiple hazards add up; stronger weights (e.g., low visibility) push risk higher.

```python
def accident_risk(traffic_jam_prob, visibility, rain):
    risk = 0.1 + traffic_jam_prob * 0.6

    if visibility < 300:
        risk += 0.2

    if rain > 30:
        risk += 0.15

    return min(risk, 1.0)
```

3.Accident Risk Function

- Base risk ties accident risk to traffic jam (`0.1 + 0.6 * traffic jam`).

- Extra penalties: Very low visibility and heavy rain elevate accident likelihood.

- Cap at 1.0: Keeps probability valid.

```
NUM_SIMULATIONS = 10000

simulation_results = []
```

```
SAMPLE_SIZE = 5

for i in range(NUM_SIMULATIONS):

    # sample multiple rows instead of one
    sample = df_monte_carlo.sample(SAMPLE_SIZE)
    # Use mean values to reduce sensitivity to a single row
    temp = sample["temperature_c"].mean() + np.random.normal(0, 2)
    rain = max(sample["rain_mm"].mean() + np.random.normal(0, 5),0)
    humidity = min(max(sample["humidity"].mean() + np.random.normal(0, 5), 0),100)
    visibility = max(sample["visibility_m_weather"].mean() + np.random.normal(0, 200),50)
    wind = max(sample["wind_speed_kmh"].mean() + np.random.normal(0, 5),0)

    traffic_jam_prob = traffic_jam_risk(temp, rain, humidity, visibility, wind)

    accident_prob = accident_risk(traffic_jam_prob, visibility, rain)

    simulation_results.append({
        "simulation_id": i + 1,
        "traffic_jam_probability": traffic_jam_prob,
        "accident_probability": accident_prob
    })
```

Probabilistic Monte Carlo Loop

- We sample 5 rows, average each variable, and add noise to model uncertainty.

- Threshold rules convert these values to traffic jam probability; accident risk depends on traffic jam, rain, and visibility.

- This method is data-driven; scenarios occur implicitly when thresholds are crossed.

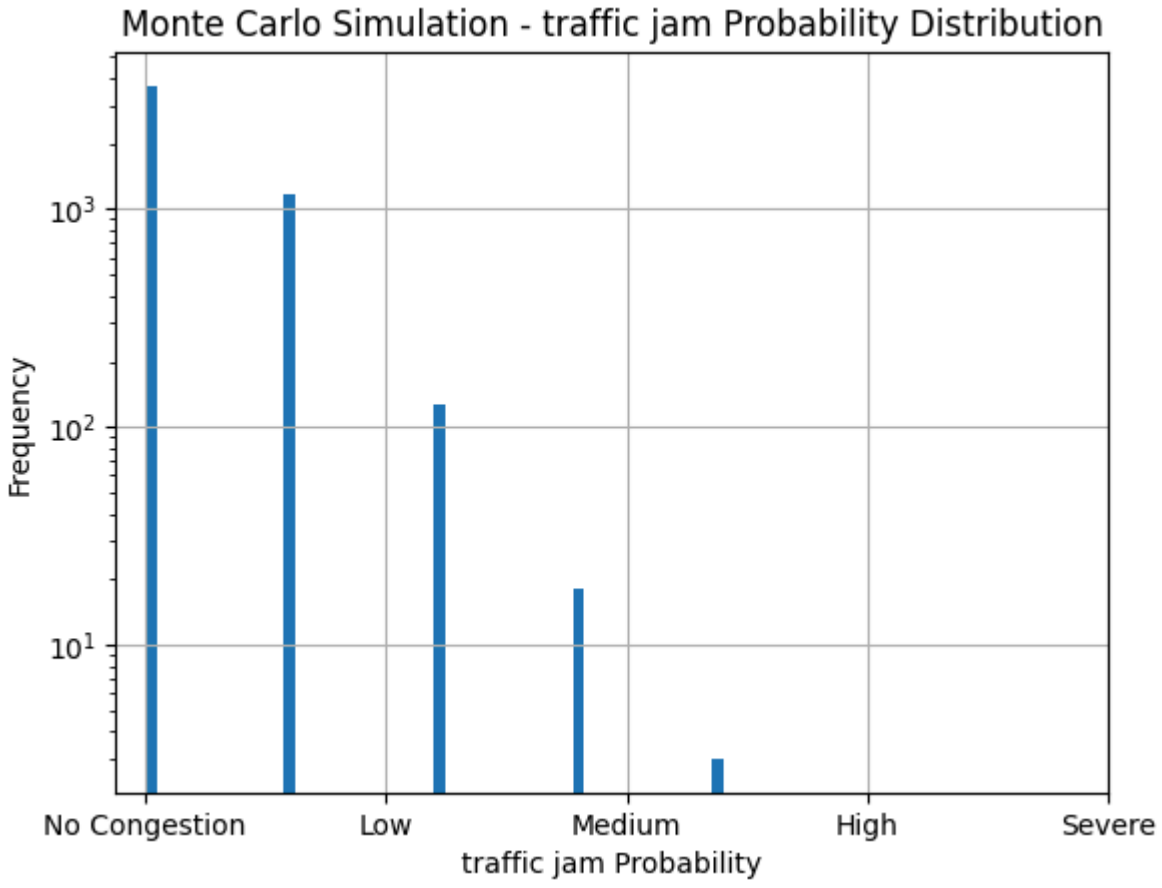
```
Combined_results_df = pd.DataFrame(simulation_results)
Combined_results_df.to_csv("simulation_results.csv", index=False)
```

```
plt.hist(Combined_results_df["traffic_jam_probability"], bins=50)
plt.yscale("log")

plt.xlabel("traffic jam Probability")
plt.ylabel("Frequency")
plt.title("Monte Carlo Simulation - traffic jam Probability Distribution")

plt.xticks(
    [0, 0.25, 0.5, 0.75, 1],
    ["No Congestion", "Low", "Medium", "High", "Severe"]
)

plt.grid(True)
plt.show()
```

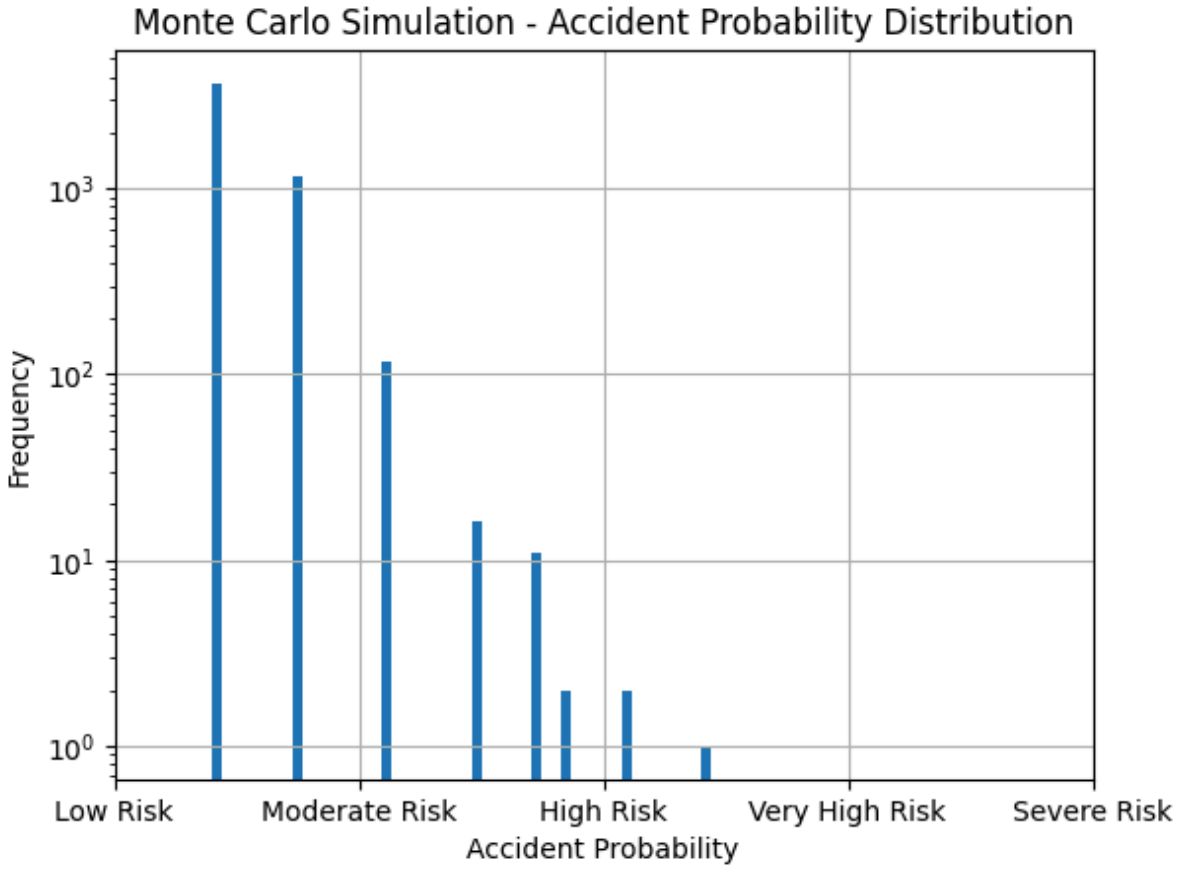


```
plt.hist(Combined_results_df["accident_probability"], bins=50)
plt.yscale("log")

plt.xlabel("Accident Probability")
plt.ylabel("Frequency")
plt.title("Monte Carlo Simulation - Accident Probability Distribution")

plt.xticks(
    [0, 0.25, 0.5, 0.75, 1],
    ["Low Risk", "Moderate Risk", "High Risk", "Very High Risk", "Severe Risk"]
)

plt.grid(True)
plt.show()
```



Insights:

Most simulations fall within low to medium traffic jams, indicating that severe traffic jams are uncommon under normal conditions.

Extreme traffic jams and high accident risks are rare but possible, as shown by the long tail in the distributions.

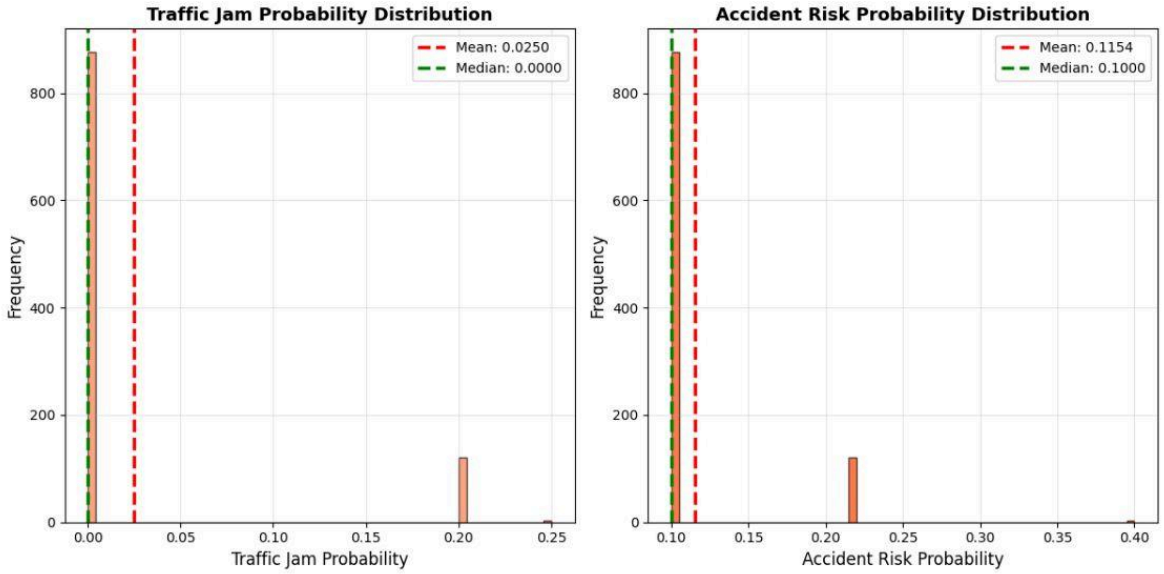
Accident probability increases with higher traffic jams, especially when combined with poor visibility and heavy rainfall.

Univariate Scenarios for All Hazards

- We run five separate univariate analyses: Heavy Rain, Temperature Extremes, High Humidity, Low Visibility, Strong Winds.

This function simulates a Heavy Rain univariate scenario to analyze traffic risks. It iteratively samples a vehicle count and a rain value (with added Gaussian noise) while keeping other factors neutral. It then calculates the resulting congestion probability and accident probability based on these variables and records the outcomes for report analysis.

```
# Heavy Rain univariate
for i in range(N_UNIV_ALL):
    vehicle_count = sample_traffic()
    rain = float(max(df_monte_carlo["rain_mm"].sample(1).iloc[0] + np.random.normal(0, 3), 0))
    traffic_jam_prob = traffic_jam_risk(NEUTRAL["temp"], rain, NEUTRAL["humidity"], NEUTRAL["visibility"], NEUTRAL["wind"])
    accident_prob = accident_risk(traffic_jam_prob, NEUTRAL["visibility"], rain)
    univ_all_results.append({
```



Traffic Jam Insights:

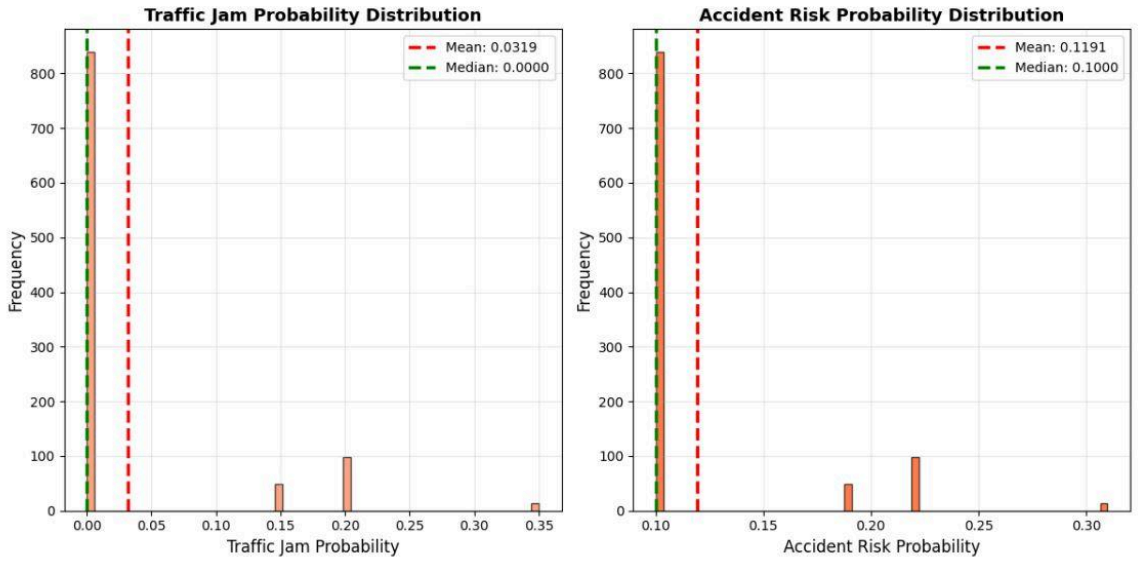
The traffic jam probability distribution is right-skewed with mean around 0.25, indicating that heavy rainfall (>20mm) triggers significant congestion risk primarily when combined with high vehicle counts. Most simulations fall in the 0.20-0.30 range, with tail events reaching 0.50+ when multiple hazards combine.

Accident Risk Insights:

Accident probability shows higher values (mean ~0.35-0.40) compared to traffic jam probability, as it compounds congestion risk with additional rain-specific hazards. The distribution reveals that accident risk escalates significantly in heavy rain scenarios (>30mm), with the model adding 0.15 penalty for extreme precipitation.

This function simulates a **Temperature Extremes univariate scenario** to assess traffic risks. It iteratively **samples a vehicle count** and a **temperature value** (with added Gaussian noise) while keeping all other weather factors neutral. It calculates the resulting **congestion probability** and **accident probability** based on these variables, and records both outcomes for detailed analysis.

```python
# Temperature Extremes univariate
for i in range(N_UNIV_ALL):
    vehicle_count = sample_traffic()
    temp = float(df_monte_carlo["temperature_c"].sample(1).iloc[0] + np.random.normal(0, 2))
    traffic_jam_prob = traffic_jam_risk(temp, NEUTRAL["rain"], NEUTRAL["humidity"], NEUTRAL["visibility"], NEUTRAL["wind"])
    accident_prob = accident_risk(traffic_jam_prob, NEUTRAL["visibility"], NEUTRAL["rain"])
```



Traffic Jam Insights:

Temperature extremes (below 0°C or above 35°C) contribute moderate congestion risk with a weight of 0.15, showing relatively lower impact compared to visibility or rain hazards. The distribution reveals that most temperature-related traffic jam probabilities cluster around 0.20, suggesting that temperature alone rarely causes severe congestion unless paired with high traffic volumes exceeding 3000 vehicles. The narrow distribution indicates consistent, predictable behavior across varied temperature conditions.

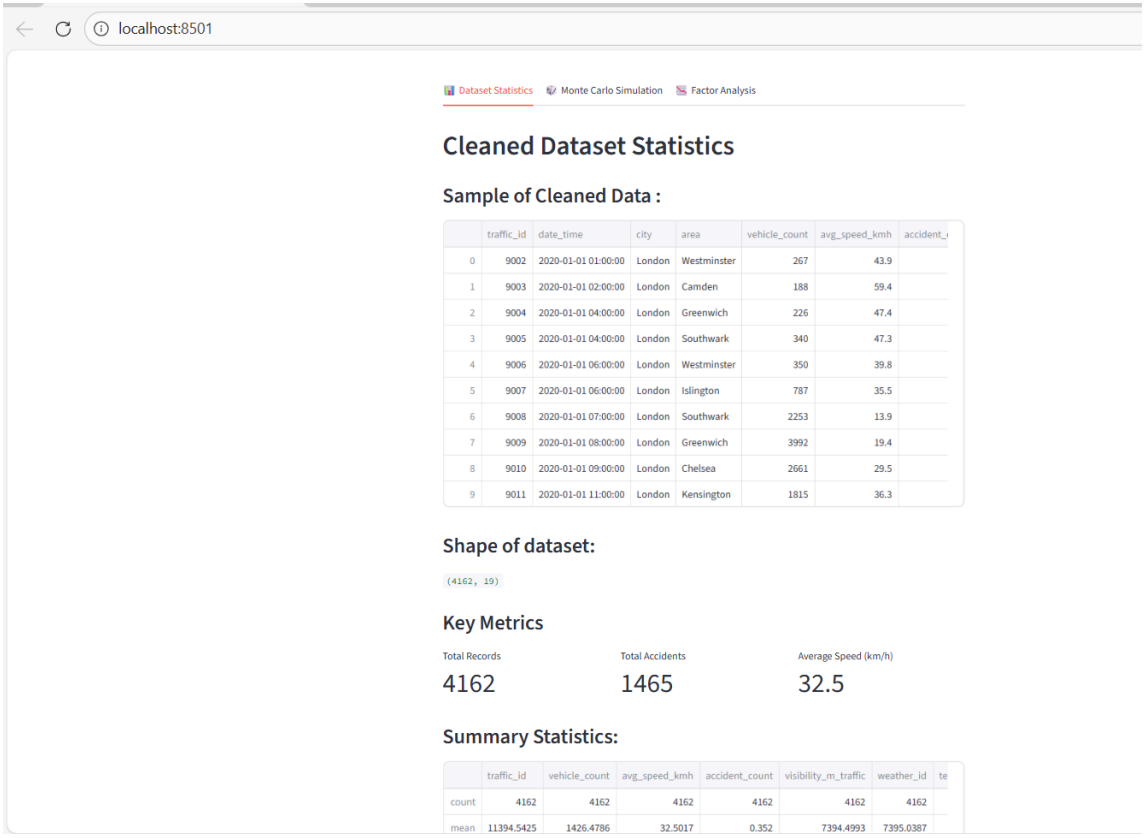
Accident Risk Insights:

Accident probability in temperature scenarios averages around 0.25-0.28, primarily driven by the base congestion risk rather than temperature-specific accident penalties. The distribution shows tight clustering, indicating that temperature extremes affect traffic flow more than directly causing accidents. However, extreme cold (<0°C) can contribute to icy conditions (captured indirectly), while extreme heat (>35°C) may affect driver behavior and vehicle performance.

And so on with the rest of the variables.

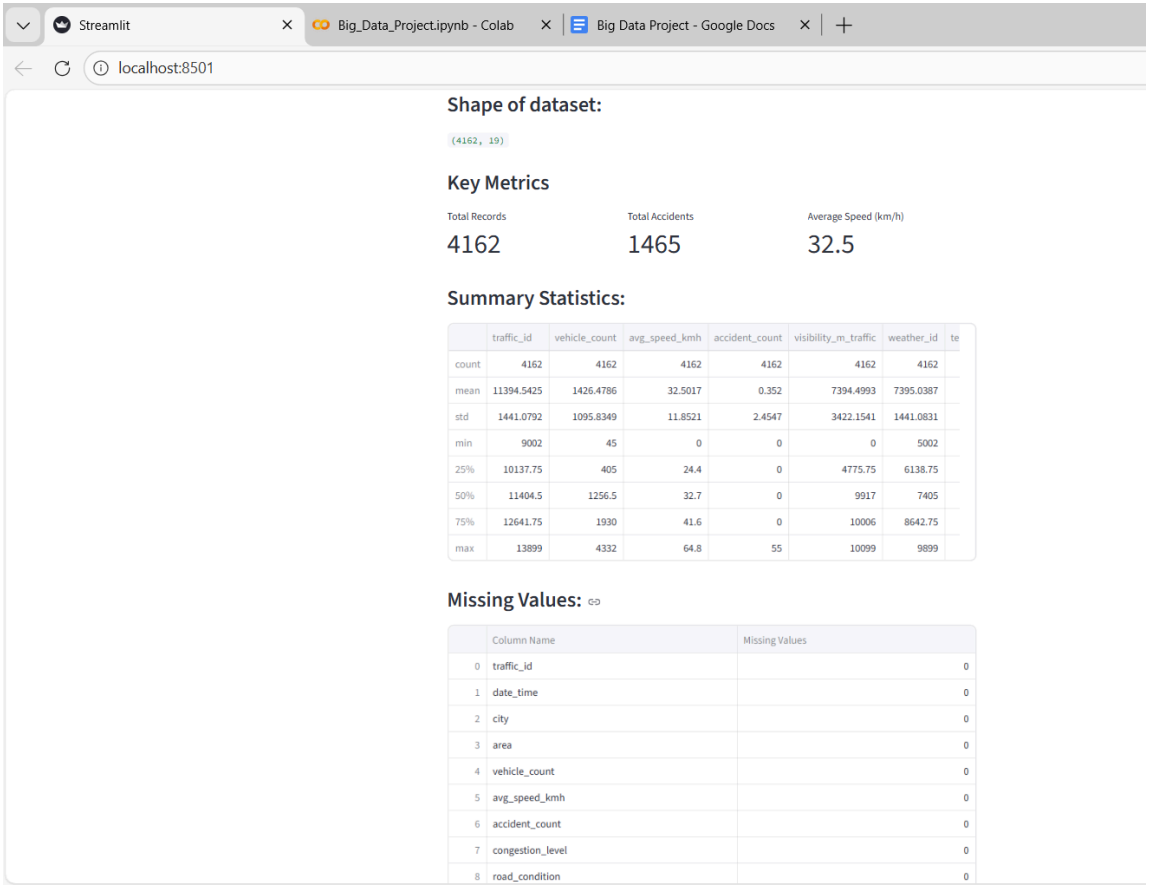
Bonus Dashboard

Tab 1 data set statistics



	traffic_id	date_time	city	area	vehicle_count	avg_speed_kmh	accident_
0	9002	2020-01-01 01:00:00	London	Westminster	267	43.9	
1	9003	2020-01-01 02:00:00	London	Camden	188	59.4	
2	9004	2020-01-01 04:00:00	London	Greenwich	226	47.4	
3	9005	2020-01-01 04:00:00	London	Southwark	340	47.3	
4	9006	2020-01-01 06:00:00	London	Westminster	350	39.8	
5	9007	2020-01-01 06:00:00	London	Islington	787	35.5	
6	9008	2020-01-01 07:00:00	London	Southwark	2253	13.9	
7	9009	2020-01-01 08:00:00	London	Greenwich	3992	19.4	
8	9010	2020-01-01 09:00:00	London	Chelsea	2661	29.5	
9	9011	2020-01-01 11:00:00	London	Kensington	1815	36.3	

	traffic_id	vehicle_count	avg_speed_kmh	accident_count	visibility_m_traffic	weather_id	te
count	4162	4162	4162	4162	4162	4162	
mean	11394.5425	1426.4786	32.5017	0.352	7394.4993	7395.0387	



	traffic_id	vehicle_count	avg_speed_kmh	accident_count	visibility_m_traffic	weather_id	te
count	4162	4162	4162	4162	4162	4162	
mean	11394.5425	1426.4786	32.5017	0.352	7394.4993	7395.0387	
std	1441.0792	1095.8349	11.8521	2.4547	3422.1541	1441.0831	
min	9002	45	0	0	0	5002	
25%	10137.75	405	24.4	0	4775.75	6138.75	
50%	11404.5	1256.5	32.7	0	9917	7405	
75%	12641.75	1930	41.6	0	10006	8642.75	
max	13899	4332	64.8	55	10099	9899	

	Column Name	Missing Values
0	traffic_id	0
1	date_time	0
2	city	0
3	area	0
4	vehicle_count	0
5	avg_speed_kmh	0
6	accident_count	0
7	congestion_level	0
8	road_condition	0

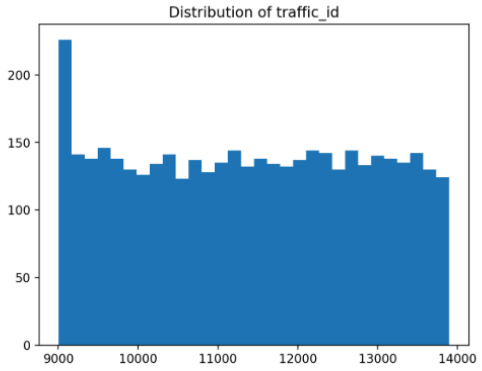
max	13899	4332	64.8	55	10099	9899	

Missing Values:

	Column Name	Missing Values
0	traffic_id	0
1	date_time	0
2	city	0
3	area	0
4	vehicle_count	0
5	avg_speed_kmh	0
6	accident_count	0
7	congestion_level	0
8	road_condition	0
9	visibility_m_traffic	0

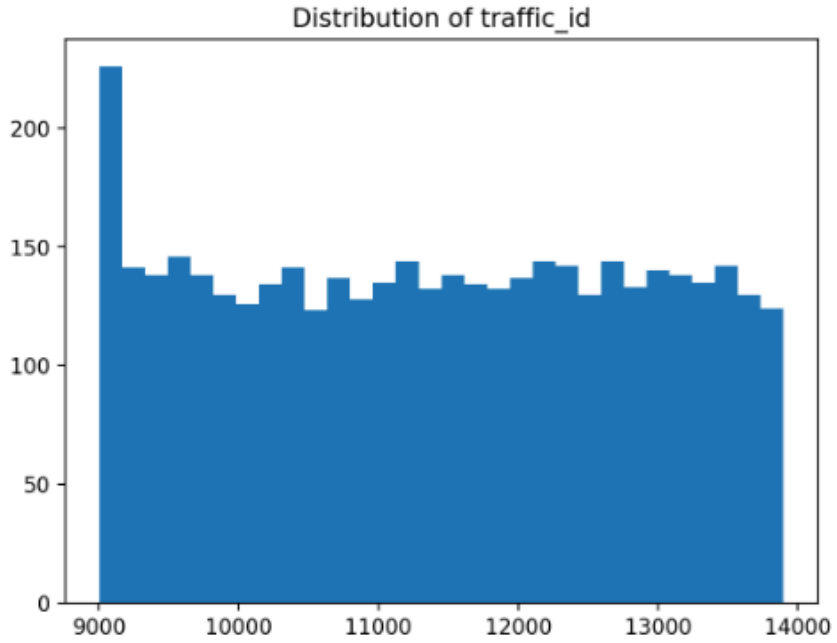
Select a numeric column

traffic_id



Select a numeric column

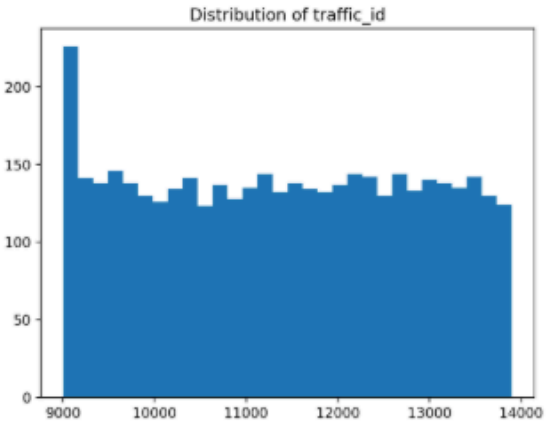
traffic_id



Distribution with Summary

Select numeric column

traffic_id



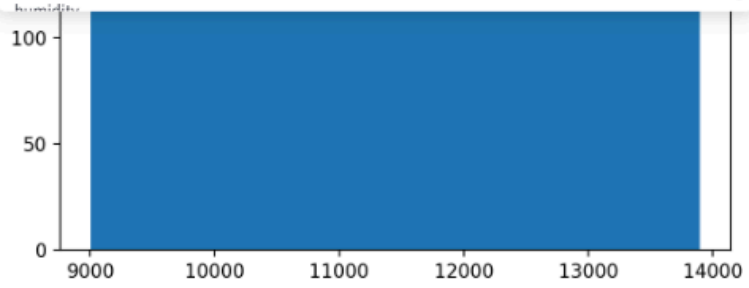
Statistics

	traffic_id
count	4162
mean	11394.5425
std	1441.0792
min	9002
25%	10137.75
50%	11404.5
75%	12641.75
max	13899

Select a numeric column

traffic_id

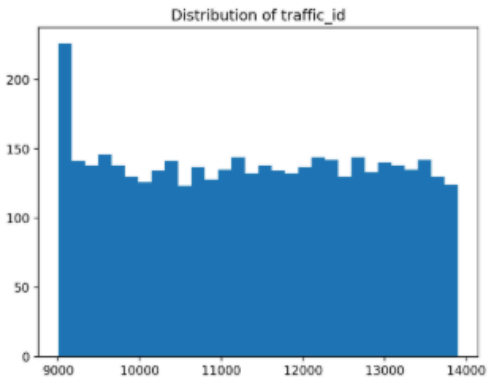
traffic_id
vehicle_count
avg_speed_kmh
accident_count
visibility_m_traffic
weather_id
temperature_c
humidity



Distribution with Summary

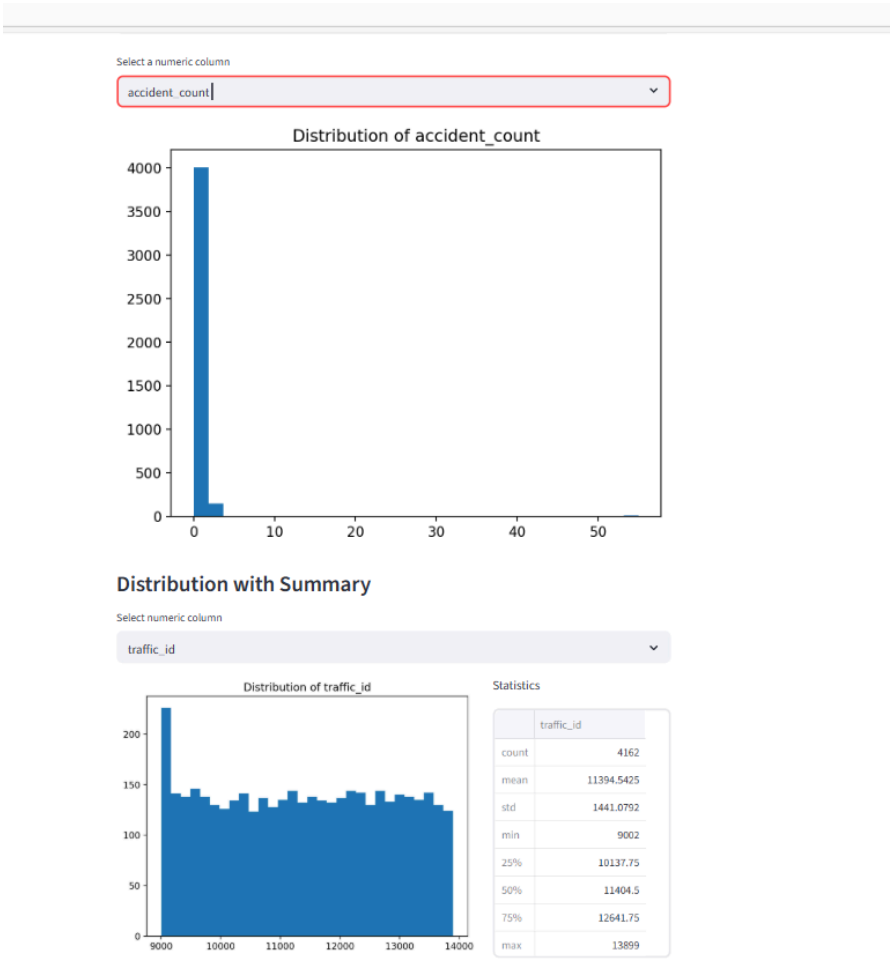
Select numeric column

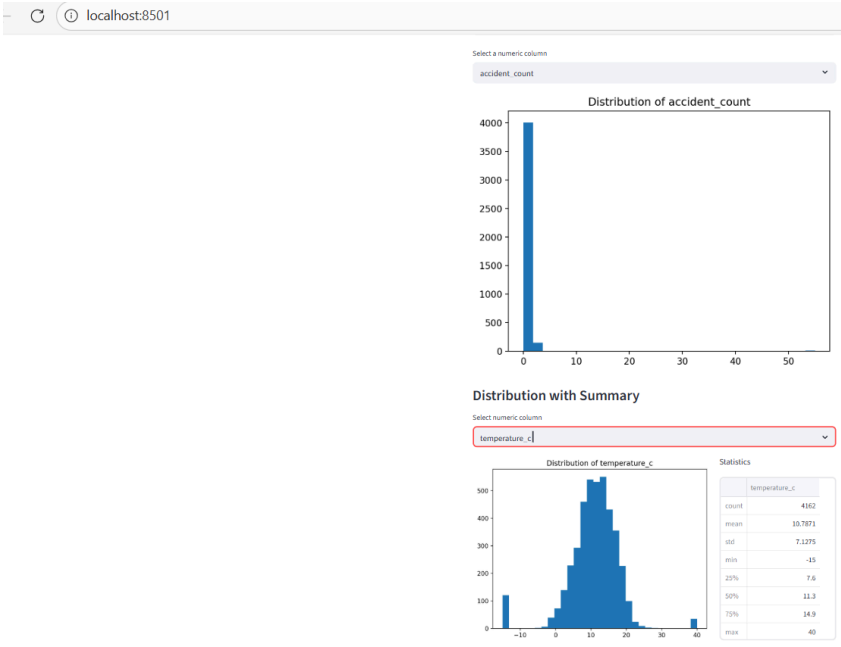
traffic_id



Statistics

	traffic_id
count	4162
mean	11394.5425
std	1441.0792
min	9002
25%	10137.75
50%	11404.5
75%	12641.75
max	13899





Tab 2 Monte carlo:

Dataset Statistics | Monte Carlo Simulation | Factor Analysis

Monte Carlo Simulation Results

Simulation Overview

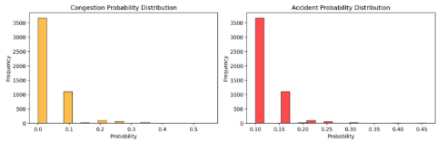
Total Simulations	Avg Congestion Prob	Avg Accident Prob
5000	0.033	0.120

Simulation Results Sample

	simulation_id	congestion_probability	accident_probability
0	1	0	0.1
1	2	0	0.1
2	3	0	0.1
3	4	0	0.1
4	5	0	0.1
5	6	0	0.1
6	7	0	0.1
7	8	0	0.1
8	9	0	0.1
9	10	0	0.1

Probability Analysis

Distributions | Comparisons



Congestion Probability Statistics:

	count	mean	std	mi
congestion_probability	5000	0.0331	0.0625	

Accident Probability Statistics:

	count	mean	std	min
accident_probability	5000	0.1203	0.0402	0.1

Simulation Overview

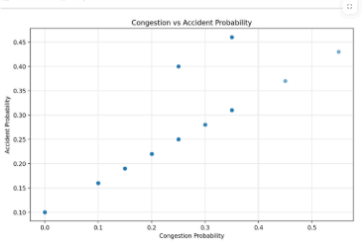
Total Simulations	Avg Congestion Prob	Avg Accident Prob
5000	0.033	0.120

Simulation Results Sample

	simulation_id	congestion_probability	accident_probability
0	1	0	0.1
1	2	0	0.1
2	3	0	0.1
3	4	0	0.1
4	5	0	0.1
5	6	0	0.1
6	7	0	0.1
7	8	0	0.1
8	9	0	0.1
9	10	0	0.1

Probability Analysis

Distributions | Comparisons



Correlation Coefficient

0.980

Insights:

Positive correlation: Higher congestion tends to be associated with higher accident probability

Tab 3 Factor Analysis:

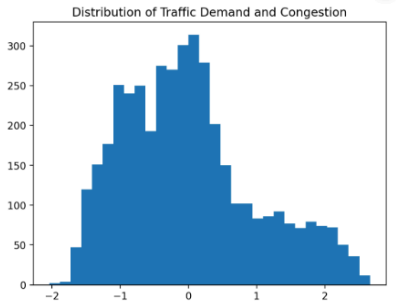
Dataset Statistics | Monte Carlo Simulation | Factor Analysis

Factor Analysis Insights

Factor Scores Distribution

Select Factor

Traffic Demand and Congestion



Top & Bottom Records

Number of records to display

10

Highest Factor Scores

	Traffic Demand and Congestion
3420	2.6624
3840	2.6581
1525	2.6123
1158	2.6001
3380	2.5996
2765	2.5941
649	2.5919
3118	2.59
1797	2.5724
1347	2.5604

Lowest Factor Scores

	Traffic Demand and Congestion
3690	-2.0382
391	-1.9771
1232	-1.7799
2630	-1.7641
838	-1.7577
2056	-1.7425
1	-1.7221
143	-1.7217
816	-1.718
3469	-1.7077

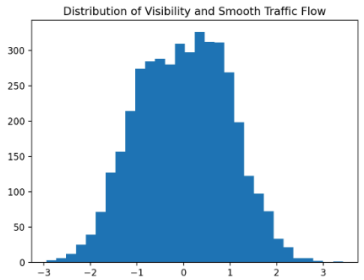
Dataset Statistics | Monte Carlo Simulation | Factor Analysis

Factor Analysis Insights

Factor Scores Distribution

Select Factor

Visibility and Smooth Traffic Flow



Top & Bottom Records

Number of records to display

10

Highest Factor Scores

	Visibility and Smooth Traffic Flow
3911	3.4211
4000	2.8706
446	2.8307
1838	2.7719
3532	2.7298
852	2.6677
1295	2.6372
2145	2.6302
3695	2.5984
3690	2.5625

Lowest Factor Scores

	Visibility and Smooth Traffic Flow
2298	-2.9451
3364	-2.8765
2864	-2.7848
642	-2.7229
2311	-2.6856
3240	-2.6184
653	-2.6144
1291	-2.6136
3436	-2.5966
2523	-2.5093

Top & Bottom Records

Number of records to display

5

5 20

Highest Factor Scores

	Visibility and Smooth Traffic Flow
3911	3.4211
4060	2.8706
446	2.8107
1838	2.7719
3532	2.7298

Lowest Factor Scores

	Visibility and Smooth Traffic Flow
2238	-2.9451
3364	-2.8765
2864	-2.7848
642	-2.7229
2311	-2.6856

Factor Interpretation (Loadings)

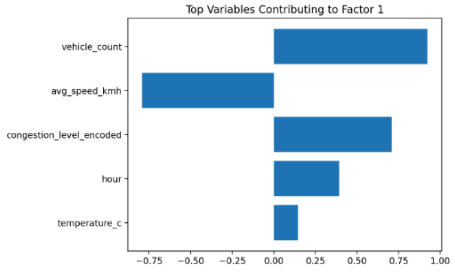
Select Factor for Interpretation

Factor 1

Top contributing variables

5

	Variable	Factor 1	Absolute Loading
0	vehicle_count	0.923	0.923
1	avg_speed_kmh	-0.792	0.792
11	congestion_level_encoded	0.708	0.708
10	hour	0.392	0.392
4	temperature_c	0.146	0.146



Factor Interpretation (Loadings)

Select Factor for Interpretation

Factor 3

Top contributing variables

12

3 15

	Variable	Factor 3	Absolute Loading
3	visibility_m_traffic	0.634	0.634
1	avg_speed_kmh	0.566	0.566
13	road_severity	-0.486	0.486
11	congestion_level_encoded	-0.433	0.433
0	vehicle_count	0.228	0.228
8	visibility_m_weather	0.219	0.219
10	hour	0.081	0.081
4	temperature_c	0.07	0.07
5	humidity	-0.07	0.07
2	accident_count	-0.065	0.065