cleaning_code_report

December 9, 2025

[1]:
```python
import pandas as pd
import numpy as np
from minio import Minio
import io
import re
```

[2]:
```python
# minio client setup
# this data from docker-compose.yml file
client = Minio(
    "localhost:9000",
    access_key="admin",
    secret_key="admin123",
    secure=False
)
```

[3]:
```python
#function to read data from minio
def read_from_minio(bucket, filename):
    try:
        obj = client.get_object(bucket, filename)
        return pd.read_csv(obj) #return data as dataframe
    except Exception as e:
        print(f"Error reading {filename}: {e}")
        return pd.DataFrame() # return empty DataFrame on error
```

[4]:
```python
# function to upload parquet to minio
def upload_parquet(df, bucket, filename):
    try:
        # convert dataframe to parquet in memory
        parquet_buffer = io.BytesIO()
        df.to_parquet(parquet_buffer, index=False)
        parquet_buffer.seek(0)

        # upload to minio
        client.put_object(
            bucket,
            filename,
            data=parquet_buffer,
            length=parquet_buffer.getbuffer().nbytes
```

```
        )
        print(f"Uploaded clean file: {filename} to bucket: {bucket}")
    except Exception as e:
        print(f"Error uploading {filename}: {e} !!!!")
```

1 Cleanning_Code

[5]:
```
# function to parse text date
def parse_date_text(x):
    if pd.isna(x):
        return None
    x = str(x).strip()
    # convert time to standard format if needed
    if re.search(r'\d{1,2}[AP]M$', x):
        x = x[:-2] + ":00 " + x[-2:]
    # add space before AM/PM if missing
    x = re.sub(r'(\d)(AM|PM)', r'\1 \2', x)
    return x
```

[6]:
```
# function to clean date column
def clean_date_column(df, col_name='date_time', dataset_name="Data"):
    # print head of the column before cleaning
    print(f"\nCleaning '{col_name}' in {dataset_name} dataset...")

    # apply parsing function
    df[col_name] = df[col_name].apply(parse_date_text)

    # convert to datetime
    #utc=True to avoid warnings about timezone because i faced error in␣
 ↪previous runs
    df[col_name] = pd.to_datetime(df[col_name], errors='coerce',␣
 ↪dayfirst=True,utc=True)
    # convert timezone-aware → timezone-naive (remove timezone)
    df[col_name] = df[col_name].dt.tz_convert(None)
    # check for invalid dates with nat and future dates
    invalid_count = df[col_name].isna().sum()
    if invalid_count > 0:
        print(f" Found {invalid_count} invalid date entries!!! \nDropping these␣
 ↪rows...")
        df = df.dropna(subset=[col_name]) # drop rows with invalid dates
    else:
        print(" All dates are valid.")

    # check for future dates beyond 2025
    if len(df) > 0:
        future_dates = df[df[col_name].dt.year > 2025]
```

```
        if len(future_dates) > 0:
            print(f"Found {len(future_dates)} future date entries beyond 2025!!!
 ↪")
            print(f"Sample: {future_dates[col_name].head(3).tolist()}") # print␣
 ↪sample of future dates
            df = df[df[col_name].dt.year <= 2025] # Keep only logical years
        else:
            print(" No future dates beyond 2025 found.")

    return df
```

[7]:
```
# Read raw data from Bronze layer
try:
    df_weather = read_from_minio("bronze", "weather_raw.csv")
    df_traffic = read_from_minio("bronze", "traffic_raw.csv")
    print("Data read from Bronze layer successfully.")
except Exception as e:
    print(f"Error reading data: {e} !!!!")
    exit(1)
```

```
Data read from Bronze layer successfully.
```

[8]:
```
# Get raw counts before cleaning to use it for reporting in final
count_w_raw = len(df_weather)
count_t_raw = len(df_traffic)
```

Cleanning_Weather

[9]:
```
print(" Cleaning Weather Data...")
df_w = df_weather.copy()
print("\nInitial Weather Data Info:")
df_w.info()
```

```
 Cleaning Weather Data…

Initial Weather Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   weather_id        4736 non-null   float64
 1   date_time         4920 non-null   object
 2   city              4756 non-null   object
 3   season            4692 non-null   object
 4   temperature_c     4904 non-null   float64
 5   humidity          4912 non-null   float64
 6   rain_mm           4881 non-null   float64
 7   wind_speed_kmh    4867 non-null   float64
```

```
 8   visibility_m       4920 non-null   object
 9   weather_condition  4163 non-null   object
dtypes: float64(5), object(5)
memory usage: 390.8+ KB
```

[10]:
```
df_w.head()
```

[10]:

	weather_id	date_time	city	season	temperature_c	humidity	\
0	6092.0	08/05/2024 02AM	London	Summer	-0.422031	33.0	
1	8807.0	2024-07-10T00:51Z	London	Autumn	-2.764074	150.0	
2	9809.0	2024-07-18 11:30	London	Winter	-1.384667	24.0	
3	6888.0	2024-02-25T01:49Z	London	Autumn	33.687915	68.0	
4	6933.0	2024-07-31T18:37Z	London	NaN	0.490894	68.0	

	rain_mm	wind_speed_kmh	visibility_m	weather_condition
0	42.105282	54.274155	7528	NaN
1	49.246241	45.528274	4981	Snow
2	27.974057	78.584612	50000	Rain
3	31.628876	24.903601	6067	Snow
4	36.518187	0.642094	1881	Storm

[11]:
```
# check for duplicate weather IDs
print("Checking Weather IDs...")
df_w['weather_id'] = pd.to_numeric(df_w['weather_id'], errors='coerce') # unify␣
 ↪type
# delete fully duplicated rows first that means duplicated because of duplicate␣
 ↪weather IDs and all other columns
df_w_before = len(df_w)
df_w = df_w.drop_duplicates()
print(f"Removed {df_w_before - len(df_w)} fully duplicated rows.")

# find max weather_id for assigning new IDs for the duplicated IDs and␣
 ↪non_duplicated in other columns
max_id = df_w['weather_id'].dropna().max()
# define the problematic rows with duplicated or null weather IDs
problem_rows = df_w[df_w['weather_id'].isna() | df_w.
 ↪duplicated(subset=['weather_id'], keep=False)]
# iterate over problematic rows and assign new unique IDs
for idx in problem_rows.index:
    max_id += 1
    df_w.at[idx, 'weather_id'] = max_id # assign new unique ID
# confirm no more duplicates
df_w = df_w.drop_duplicates(subset=['weather_id'])
# convert weather_id to int
df_w['weather_id'] = df_w['weather_id'].astype(int)

print("Cleaning Done Successfully!")
```

```
print(f"Total rows after cleaning: {len(df_w)}")
```

```
Checking Weather IDs…
Removed 0 fully duplicated rows.
Cleaning Done Successfully!
Total rows after cleaning: 5000
Cleaning Done Successfully!
Total rows after cleaning: 5000
```

[12]:
```
# Clean date column in weather data
df_w = clean_date_column(df_w, 'date_time', dataset_name="Weather")
```

```
Cleaning 'date_time' in Weather dataset…
 Found 255 invalid date entries!!!
Dropping these rows…
 No future dates beyond 2025 found.
```

[13]:
```
# check and clean temperature_c column from -5 to 40
print("Checking Temperature...")
df_w['temperature_c'] = pd.to_numeric(df_w['temperature_c'], errors='coerce') #␣
 ↪unify type
bad_temp = df_w[(df_w['temperature_c'] < -6) | (df_w['temperature_c'] > 40)]

# Detect outliers
if len(bad_temp) > 0:
    print(f"Found {len(bad_temp)} outliers! (Values outside 5 to 40)\n")
    print(f"Sample: {bad_temp['temperature_c'].head().tolist()}")# print sample␣
 ↪of bad temps
else:
    print(" Temperature data is clean.")
# keep only valid temperature ranges
df_w = df_w[(df_w['temperature_c'] >= -6) & (df_w['temperature_c'] <= 40)]
df_w['temperature_c'] = df_w['temperature_c'].astype(float)
```

```
Checking Temperature…
Found 163 outliers! (Values outside 5 to 40)

Sample: [-30.0, 60.0, 60.0, 60.0, 60.0]
```

[14]:
```
# check and clean humidity between 0 and 100
print("Checking Humidity...")
df_w['humidity'] =pd.to_numeric(df_w['humidity'], errors='coerce') # unify type
df_w['humidity']= df_w['humidity'].abs()# make sure humidity is non-negative
bad_hum = df_w[(df_w['humidity'] < 0) | (df_w['humidity'] > 100)]

# Detect outliers
if len(bad_hum) > 0:
```

```
    print(f"Found {len(bad_hum)} outliers! (Values outside 0-100)\n")
    print(f"Sample: {bad_hum['humidity'].head().tolist()}")# print sample of␣
 ↪bad humidity
else:
    print(" Humidity data is clean.")
# keep only valid humidity ranges
df_w = df_w[(df_w['humidity'] >= 0) & (df_w['humidity'] <= 100)]
df_w['humidity']= df_w['humidity'].astype(int) # unify type
```

```
Checking Humidity…
Found 101 outliers! (Values outside 0-100)

Sample: [150.0, 150.0, 150.0, 150.0, 150.0]
```

[15]:
```
# check and clean wind_speed_kmh >200
print("Checking Wind Speed...")
# convert to numeric
df_w['wind_speed_kmh'] = pd.to_numeric(df_w['wind_speed_kmh'], errors='coerce')
# ensure non-negative
df_w['wind_speed_kmh'] = df_w['wind_speed_kmh'].abs()

# detect outliers (>200 km/h)
bad_wind = df_w[df_w['wind_speed_kmh'] >= 200]
if len(bad_wind) > 0:
    print(f"Found {len(bad_wind)} outliers! (Speed > 200 km/h)\n")
    print("Sample:", bad_wind['wind_speed_kmh'].head().tolist())
else:
    print("Wind Speed data is clean.")
# remove outliers and keep only valid speeds
df_w = df_w[df_w['wind_speed_kmh'] < 200]

# convert type properly
df_w['wind_speed_kmh'] = df_w['wind_speed_kmh'].astype(float)
print("Wind Speed cleaning completed successfully!")
```

```
Checking Wind Speed…
Found 96 outliers! (Speed > 200 km/h)

Sample: [200.0, 200.0, 200.0, 200.0, 200.0]
Wind Speed cleaning completed successfully!
```

[16]:
```
# check and clean visibility_m 50 - 10000
print("Checking Visibility...")
df_w['visibility_m'] =pd.to_numeric(df_w['visibility_m'], errors='coerce') #␣
 ↪unify type
df_w['visibility_m'] = df_w['visibility_m'].abs()# make sure visibility is␣
 ↪non-negative
```

```
df_w['visibility_m'] = pd.to_numeric(df_w['visibility_m'], errors='coerce') #␣
 ↪unify type
bad_vis = df_w[(df_w['visibility_m'] < 50) | (df_w['visibility_m'] > 10000)]

# Detect outliers
if len(bad_vis) > 0:
    print(f"Found {len(bad_vis)} outliers! (Visibility >= 40,000m)")
    print(f"Sample: {bad_vis['visibility_m'].head().tolist()}") # print sample␣
 ↪of bad visibility
else:
    print("Visibility data is clean.")
# keep only valid visibility ranges
df_w = df_w[(df_w['visibility_m'] <= 10000) | (df_w['visibility_m'] >= 50)]
df_w['visibility_m'] =df_w['visibility_m'].astype(int)
```

```
Checking Visibility…
Found 71 outliers! (Visibility >= 40,000m)
Sample: [50000.0, 50000.0, 50000.0, 50000.0, 50000.0]
```

[17]:
```
# Weather Condition Imputation and Standardization
print("\nStandardizing Weather Conditions...")
# convert all existing values to lowercase first
df_w['weather_condition'] = df_w['weather_condition'].str.lower()
# define accepted weather conditions from PDF (lowercase)
accepted_conditions = {'clear', 'rain', 'fog', 'storm', 'snow'}
# check for non-standard values
non_standard_mask = ~df_w['weather_condition'].isin(accepted_conditions)

if non_standard_mask.any():
    non_standard_count = non_standard_mask.sum()
    non_standard_values = df_w[non_standard_mask]['weather_condition'].unique()

    print(f"   Found {non_standard_count} non-standard weather conditions")
    print(f"   Non-standard values: {list(non_standard_values)}")

    # Replace non-standard values with 'unknown'
    df_w.loc[non_standard_mask, 'weather_condition'] = 'unknown'
    print(f"   Replaced {non_standard_count} non-standard values with␣
 ↪'unknown'")

# handle missing values
nan_count = df_w['weather_condition'].isna().sum()
if nan_count > 0:
    print(f"Found {nan_count} missing weather conditions!!!! \nfilling with␣
 ↪'unknown'...")
    df_w['weather_condition'] = df_w['weather_condition'].fillna('unknown')
```

```
# final validation
final_values = df_w['weather_condition'].unique()
final_counts = df_w['weather_condition'].value_counts()

print(f"   Weather conditions standardized to accepted values:␣
 ↪{list(final_values)}")
print(f"   Counts: {dict(final_counts)}")
```

```
Standardizing Weather Conditions…
   Found 682 non-standard weather conditions
   Non-standard values: [nan]
   Replaced 682 non-standard values with 'unknown'
   Weather conditions standardized to accepted values: ['unknown', 'rain',
'snow', 'storm', 'clear', 'fog']
   Counts: {'rain': 686, 'unknown': 682, 'storm': 682, 'fog': 672, 'snow': 650,
'clear': 608}
```

[18]:
```
# Rainfall Imputation
print("Handling missing rain values...")
df_w['rain_mm'] =pd.to_numeric(df_w['rain_mm'], errors='coerce') # unify type
df_w['rain_mm'] = df_w['rain_mm'].abs()# make sure rain is non-negative
if df_w['rain_mm'].isna().sum() > 0:
    print("Filling missing rain values with median...") # print only if there␣
 ↪are missing values
    rain_median = df_w['rain_mm'].median()
    df_w['rain_mm'] = df_w['rain_mm'].fillna(rain_median) # impute missing rain␣
 ↪with median
else:
    print("No missing rain values found.")
df_w['rain_mm'] = df_w['rain_mm'].astype(float) # unify type
```

```
Handling missing rain values…
Filling missing rain values with median…
```

[19]:
```
#function to get season from date
def get_season(date):
# get month from date
    month = date.month
    if month in [12, 1, 2]:
        return "winter"
    elif month in [3, 4, 5]:
        return "spring"
    elif month in [6, 7, 8]:
        return "summer"
    else:
        return "autumn"
```

[20]:
```
# Season Imputation and Unification
print("\nHandling missing season values and unifying season names...")

# count missing before fixing
missing_season_before = df_w['season'].isna().sum()

# convert existing season values to lowercase
if 'season' in df_w.columns:
    df_w['season'] = df_w['season'].str.lower()

# fill missing season values based on date_time
df_w['season'] = df_w.apply(
    lambda row: get_season(row['date_time']) if pd.isna(row['season']) else␣
 ↪row['season'],
    axis=1)

# check for any values that don't match our standard seasons
standard_seasons = {'winter', 'spring', 'summer', 'autumn'}
non_standard_mask = ~df_w['season'].isin(standard_seasons)

# fix non-standard values using date_time
if non_standard_mask.any():
    non_standard_count = non_standard_mask.sum()
    print(f"Found {non_standard_count} non-standard season values!!!")
    print(f"Sample of non-standard values: {df_w[non_standard_mask]['season'].
 ↪unique()[:5].tolist()}")

    # Fix non-standard values using date_time
    df_w.loc[non_standard_mask, 'season'] = df_w[non_standard_mask].apply(
        lambda row: get_season(row['date_time']),
        axis=1)

    print(f"Replaced {non_standard_count} non-standard values with season from␣
 ↪date_time")

# convert all season values to lowercase again to ensure consistency
df_w['season'] = df_w['season'].str.lower()
# Step 7: Count final missing values
missing_season_after = df_w['season'].isna().sum()

print(f"\n   Fixed {missing_season_before - missing_season_after} missing␣
 ↪season values")
print(f"   Season values unified to lowercase: {sorted(df_w['season'].unique().
 ↪tolist())}")
print(f"   Season column now has {len(df_w) - missing_season_after} valid␣
 ↪entries out of {len(df_w)} total")
```

```
Handling missing season values and unifying season names…
Found 231 non-standard season values!!!
Sample of non-standard values: ['rainy']
Replaced 231 non-standard values with season from date_time

   Fixed 246 missing season values
Found 231 non-standard season values!!!
Sample of non-standard values: ['rainy']
Replaced 231 non-standard values with season from date_time

   Fixed 246 missing season values
   Season values unified to lowercase: ['autumn', 'spring', 'summer', 'winter']
   Season column now has 3980 valid entries out of 3980 total
```

[21]:
```
# City Imputation
print("Fixing missing cities...")
df_w['city'] = df_w['city'].fillna('London') # impute missing city with London␣
 ↪because most data is from London
```

```
Fixing missing cities…
```

[22]:
```
# Air Pressure Cleaning if added!!
if 'air_pressure_hpa' in df_w.columns:
    print("\nCleaning Air Pressure Data...")
    df_w['air_pressure_hpa'] =pd.numeric(df_w['air_pressure_hpa'],␣
 ↪errors='coerce') # unify type
    df_w['air_pressure_hpa'] = df_w['air_pressure_hpa'].abs()# make sure air␣
 ↪pressure is non-negative
    # remove outliers outside 950-1050 hPa
    before_count = len(df_w)
    outlier_mask = (df_w['air_pressure_hpa'] < 950) | (df_w['air_pressure_hpa']␣
 ↪> 1050)
    outlier_count = outlier_mask.sum()
    # handle missing values
    missing_before = df_w['air_pressure_hpa'].isna().sum()
    if missing_before > 0:
        print(f"Found {missing_before} missing pressure values!!!!")

        # fill with median (better for pressure data)
        pressure_median = df_w['air_pressure_hpa'].median()
        df_w['air_pressure_hpa'] = df_w['air_pressure_hpa'].
 ↪fillna(pressure_median)

        print(f"Filled {missing_before} missing values with median:␣
 ↪{pressure_median:.1f} hPa")
        missing_after = df_w['air_pressure_hpa'].isna().sum()
        print(f"Missing values after imputation: {missing_after}")
```

```
        df_w['air_pressure_hpa'] = df_w['air_pressure_hpa'].astype(float,␣
↪errors='coerce')# unify type
```

Cleanning_Traffic

[23]:
```
print(" Cleaning traffic Data...")
df_t = df_traffic.copy()
print("\nInitial traffic Data Info:")
df_t.info()
```

```
 Cleaning traffic Data…

Initial traffic Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   traffic_id        4754 non-null   float64
 1   date_time         4905 non-null   object
 2   city              4747 non-null   object
 3   area              4191 non-null   object
 4   vehicle_count     4869 non-null   float64
 5   avg_speed_kmh     4890 non-null   float64
 6   accident_count    4879 non-null   float64
 7   congestion_level  3981 non-null   object
 8   road_condition    4031 non-null   object
 9   visibility_m      4864 non-null   float64
dtypes: float64(5), object(5)
memory usage: 390.8+ KB
```

[24]:
```
df_t.head()
```

[24]:

	traffic_id	date_time	city	area	vehicle_count	\
0	10938.0	2024-02-02T15:54Z	London	Chelsea	776.0	
1	10283.0	03/09/2024 12PM	London	Islington	2021.0	
2	12293.0	2024-05-25T17:13Z	London	Camden	2816.0	
3	14068.0	2024-12-09 20:54	London	NaN	178.0	
4	10606.0	2024-01-02T00:39Z	London	NaN	3148.0	

	avg_speed_kmh	accident_count	congestion_level	road_condition	visibility_m
0	58.704429	6.0	Medium	NaN	8384.0
1	15.023026	8.0	High	Dry	2890.0
2	NaN	9.0	Severe	Wet	3000.0
3	98.951356	3.0	Medium	Dry	50000.0
4	43.926189	3.0	Medium	NaN	8336.0

[25]:
```
# check for duplicate traffic IDs
print("Checking traffic IDs...")
df_t['traffic_id'] = pd.to_numeric(df_t['traffic_id'], errors='coerce') # unify␣
 ↪type
# delete fully duplicated rows first that means duplicated because of duplicate␣
 ↪traffic IDs and all other columns
df_t_before = len(df_t)
df_t = df_t.drop_duplicates()
print(f"Removed {df_t_before - len(df_t)} fully duplicated rows.")

# find max traffic_id for assigning new IDs for the duplicated IDs and␣
 ↪non_duplicated in other columns
max_id = df_t['traffic_id'].dropna().max()
# define the problematic rows with duplicated or null traffic IDs
problem_rows = df_t[df_t['traffic_id'].isna() | df_t.
 ↪duplicated(subset=['traffic_id'], keep=False)]
# iterate over problematic rows and assign new unique IDs
for idx in problem_rows.index:
    max_id += 1
    df_t.at[idx, 'traffic_id'] = max_id # assign new unique ID
# confirm no more duplicates
df_t = df_t.drop_duplicates(subset=['traffic_id'])
# convert traffic_id to int
df_t['traffic_id'] = df_t['traffic_id'].astype(int)

print("Cleaning Done Successfully!")
print(f"Total rows after cleaning: {len(df_t)}")
```

```
Checking traffic IDs…
Removed 0 fully duplicated rows.
Cleaning Done Successfully!
Total rows after cleaning: 5000
Cleaning Done Successfully!
Total rows after cleaning: 5000
```

[26]:
```
# clean traffic date column
df_t = clean_date_column(df_t, 'date_time', dataset_name="Traffic")
```

```
Cleaning 'date_time' in Traffic dataset…
 Found 268 invalid date entries!!!
Dropping these rows…
 No future dates beyond 2025 found.
```

[27]:
```
# Fix Missing City (Standardize to London as per PDF)
print("Fixing missing cities...")
df_t['city'] = df_t['city'].fillna('London')
```

Fixing missing cities…

[28]:
```
# impute missing area values with 'Unknown'
print("\nHandling missing area values...")
# check for missing values
missing_area = df_t['area'].isna().sum()
if missing_area > 0:
    print(f"Found {missing_area} missing areas!!!! \nFilling with 'unknown'...")

    # fill missing values with 'unknown'
    df_t['area'] = df_t['area'].fillna('unknown')

# convert all area values to lowercase
df_t['area'] = df_t['area'].str.lower()

# show sample of unique areas (first 10)
unique_areas = df_t['area'].unique()
print(f"Sample of unique areas: {sorted(unique_areas[:10]) if len(unique_areas)␣
 ↪> 10 else sorted(unique_areas)}")
print(f"Total unique area values: {len(unique_areas)}")
```

```
Handling missing area values…
Found 771 missing areas!!!!
Filling with 'unknown'…
Sample of unique areas: ['camden', 'chelsea', 'islington', 'kensington',
'southwark', 'unknown']
Total unique area values: 6
```

[29]:
```
# check and clean avg_speed_kmh
print("Checking Speed...")
df_t['avg_speed_kmh'] = pd.to_numeric(df_t['avg_speed_kmh'], errors='coerce') #␣
 ↪unify type
# correct  negatives
df_t['avg_speed_kmh'] = df_t['avg_speed_kmh'].abs() # take absolute value
# check for unrealistic high speeds
bad_speed = df_t[df_t['avg_speed_kmh'] > 150]
if len(bad_speed) > 0:
    print(f"Found {len(bad_speed)} excessive speeds (> 150 km/h)!") # print␣
 ↪sample of bad speeds
    df_t = df_t[df_t['avg_speed_kmh'] <= 150] # Keep only valid speeds
    df_t['avg_speed_kmh'] = df_t['avg_speed_kmh'].astype(float,␣
 ↪errors='coerce') # unify type
else:
    print("Speed data is clean.")
```

```
Checking Speed…
Speed data is clean.
```

[30]:
```
# standardize speed values
print("\nChecking Speed...")
df_t['avg_speed_kmh'] = pd.to_numeric(df_t['avg_speed_kmh'], errors='coerce') #␣
 ↪unify type
df_t['avg_speed_kmh'] = df_t['avg_speed_kmh'].abs() # Fix negatives
df_t = df_t[df_t['avg_speed_kmh'] <= 150] # Remove outliers
# Impute missing speed with Mean (Average)
speed_mean = df_t['avg_speed_kmh'].mean()
df_t['avg_speed_kmh'] = df_t['avg_speed_kmh'].fillna(speed_mean)
print(f"Filled missing speeds with mean: {speed_mean:.2f} km/h")
df_t['avg_speed_kmh'] = df_t['avg_speed_kmh'].astype(float) # unify type
```

```
Checking Speed...
Filled missing speeds with mean: 60.78 km/h
```

[31]:
```
# standardize vehicle_count values
print("Checking Vehicle Counts...")
df_t['vehicle_count'] = pd.to_numeric(df_t['vehicle_count'], errors='coerce') #␣
 ↪unify type
df_t['vehicle_count'] = df_t['vehicle_count'].abs() # Fix negatives
df_t = df_t[df_t['vehicle_count'] <= 5000] # Remove outliers that are too high
# fill missing vehicle counts with median (safer for counts)
veh_median = df_t['vehicle_count'].median()
df_t['vehicle_count'] = df_t['vehicle_count'].fillna(veh_median)
df_t['vehicle_count'] = df_t['vehicle_count'].astype(int) # unify type
print("Vehicle Counts cleaned successfully!")
```

```
Checking Vehicle Counts...
Vehicle Counts cleaned successfully!
```

[32]:
```
# standardize accident_count values
print("Checking Accidents...")
df_t['accident_count'] = pd.to_numeric(df_t['accident_count'], errors='coerce')␣
 ↪# unify type
df_t['accident_count'] = df_t['accident_count'].abs() # Fix negatives
df_t['accident_count'] = df_t['accident_count'].fillna(0)# fill missing␣
 ↪accidents with 0
bad_acc = df_t[df_t['accident_count'] >= 11] # detect extreme accident counts
if len(bad_acc) > 0:
    print(f" Found {len(bad_acc)} extreme accident counts (> 10)!!!! \nRemoving␣
 ↪them...")
    df_t = df_t[df_t['accident_count'] <= 10] # Keep only valid accident counts␣
 ↪and delete outliers
else:
    print("Accident data is reasonable.")
df_t['accident_count'] = df_t['accident_count'].astype(int) # unify type
```

```
Checking Accidents…
 Found 108 extreme accident counts (> 10)!!!!
Removing them…

 Found 108 extreme accident counts (> 10)!!!!
Removing them…
```

[33]:
```python
# standardize visibility_m values
print("Checking Visibility...")
# Ensure it's numeric first
df_t['visibility_m'] = pd.to_numeric(df_t['visibility_m'], errors='coerce') # unify type
df_t['visibility_m'] = df_t['visibility_m'].abs() # convert negative values to positive

bad_acc = df_t[(df_t['visibility_m'] < 50) | (df_t['visibility_m'] > 10000)] # detect extreme accident counts
if len(bad_acc) > 0:
    print(f" Found {len(bad_acc)} extreme visibility!!!! \nRemoving them...")
    df_t = df_t[(df_t['visibility_m'] >= 50) | (df_t['visibility_m'] <= 10000)] # Keep only valid accident counts and delete outliers
else:
    print("visibility data is reasonable.")
# Impute missing visibility with Mean
vis_mean = df_t['visibility_m'].mean()
df_t['visibility_m'] = df_t['visibility_m'].fillna(vis_mean)
print(f"Filled missing visibility with mean: {vis_mean:.2f} m")
df_t['visibility_m'] = df_t['visibility_m'].astype(int) # unify type
```

```
Checking Visibility…
 Found 107 extreme visibility!!!!
Removing them…
Filled missing visibility with mean: 6198.06 m
 Found 107 extreme visibility!!!!
Removing them…
Filled missing visibility with mean: 6198.06 m
```

[34]:
```python
# standardize congestion_level values
print("\nStandardizing Congestion Levels...")
# convert all existing values to lowercase first
df_t['congestion_level'] = df_t['congestion_level'].str.lower()
# define our standard congestion levels (lowercase)
standard_levels = {'low', 'medium', 'high'}
#  check for non-standard values
non_standard_mask = ~df_t['congestion_level'].isin(standard_levels)

if non_standard_mask.any():
    non_standard_count = non_standard_mask.sum()
```

```
    non_standard_values = df_t[non_standard_mask]['congestion_level'].unique()

    print(f"Found {non_standard_count} non-standard congestion levels!!!")
    print(f"Non-standard values: {list(non_standard_values)}")

    # replace non-standard values with 'unknown'
    df_t.loc[non_standard_mask, 'congestion_level'] = 'unknown'
    print(f"   Replaced {non_standard_count} non-standard values with␣
 ↪'unknown'")

# fill any remaining NaN values with 'unknown'
nan_count = df_t['congestion_level'].isna().sum()
if nan_count > 0:
    print(f"   Found {nan_count} NaN values, filling with 'unknown'")
    df_t['congestion_level'] = df_t['congestion_level'].fillna('unknown')

#  final validation
final_values = df_t['congestion_level'].unique()
final_counts = df_t['congestion_level'].value_counts()

print(f"   Congestion levels standardized to: {list(final_values)}")
print(f"   Counts: {dict(final_counts)}")
```

```
Standardizing Congestion Levels…
Found 1703 non-standard congestion levels!!!
Non-standard values: [nan, 'severe']
   Replaced 1703 non-standard values with 'unknown'
   Congestion levels standardized to: ['medium', 'high', 'low', 'unknown']
   Counts: {'unknown': 1703, 'low': 875, 'high': 807, 'medium': 766}
```

[35]:
```
# standardize road_condition values
print("Standardizing Road Conditions...")

df_t['road_condition'] = df_t['road_condition'].str.lower()

# define our standard road conditions (lowercase)
standard_conditions = {'dry', 'wet', 'snowy', 'damaged'}

#  check for non-standard values
non_standard_mask = ~df_t['road_condition'].isin(standard_conditions)

if non_standard_mask.any():
    non_standard_count = non_standard_mask.sum()
    non_standard_values = df_t[non_standard_mask]['road_condition'].unique()
```

```
    print(f"Found {non_standard_count} non-standard road conditions!!!")# print␣
 ↪only if non-standard values found
    print(f"Non-standard values: {list(non_standard_values)}")# print only if␣
 ↪non-standard values found

    #  replace non-standard values with 'unknown'
    df_t.loc[non_standard_mask, 'road_condition'] = 'unknown'
    print(f"Replaced {non_standard_count} non-standard values with 'unknown'")

#  fill any remaining NaN values with 'unknown'
nan_count = df_t['road_condition'].isna().sum()
if nan_count > 0:
    print(f"   Found {nan_count} NaN values, filling with 'unknown'")
    df_t['road_condition'] = df_t['road_condition'].fillna('unknown')

#  Final validation
final_values = df_t['road_condition'].unique()
final_counts = df_t['road_condition'].value_counts()

print(f"   Road conditions standardized to: {list(final_values)}")
print(f"   Counts: {dict(final_counts)}")
```

```
Standardizing Road Conditions…
Found 817 non-standard road conditions!!!
Non-standard values: [nan]
Replaced 817 non-standard values with 'unknown'
   Road conditions standardized to: ['unknown', 'dry', 'wet', 'damaged',
'snowy']
   Counts: {'snowy': 848, 'wet': 842, 'dry': 829, 'unknown': 817, 'damaged':
815}
```

[36]:
```
# the report of the cleaning phase
print("DATA QUALITY REPORT (PHASE 2)\n")
print(f"Weather Data:")
print(f" - Rows Before: {count_w_raw}")
print(f" - Rows After:  {len(df_w)}\n\n\n\n")


print(f"Traffic Data:")
print(f" - Rows Before: {count_t_raw}")
print(f" - Rows After:  {len(df_t)}")
```

```
DATA QUALITY REPORT (PHASE 2)

Weather Data:
 - Rows Before: 5000
 - Rows After:  3980
```

```
Traffic Data:
 - Rows Before: 5000
 - Rows After:  4151
```

[37]:
```
df_t.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4151 entries, 0 to 4999
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   traffic_id        4151 non-null   int32
 1   date_time         4151 non-null   datetime64[ns]
 2   city              4151 non-null   object
 3   area              4151 non-null   object
 4   vehicle_count     4151 non-null   int32
 5   avg_speed_kmh     4151 non-null   float64
 6   accident_count    4151 non-null   int32
 7   congestion_level  4151 non-null   object
 8   road_condition    4151 non-null   object
 9   visibility_m      4151 non-null   int32
dtypes: datetime64[ns](1), float64(1), int32(4), object(4)
memory usage: 291.9+ KB
```

[38]:
```
df_w.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3980 entries, 0 to 4998
Data columns (total 10 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   weather_id         3980 non-null   int32
 1   date_time          3980 non-null   datetime64[ns]
 2   city               3980 non-null   object
 3   season             3980 non-null   object
 4   temperature_c      3980 non-null   float64
 5   humidity           3980 non-null   int32
 6   rain_mm            3980 non-null   float64
 7   wind_speed_kmh     3980 non-null   float64
 8   visibility_m       3980 non-null   int32
 9   weather_condition  3980 non-null   object
dtypes: datetime64[ns](1), float64(3), int32(3), object(3)
memory usage: 295.4+ KB
```

[39]:
```
# Upload cleaned data to Silver layer in parquet format
upload_parquet(df_w, "silver", "weather_cleaned.parquet")
upload_parquet(df_t, "silver", "traffic_cleaned.parquet")

print("Done!!  \nCheck MinIO silver bucket.") #YES!!!!!!
```

```
Uploaded clean file: weather_cleaned.parquet to bucket: silver
Uploaded clean file: traffic_cleaned.parquet to bucket: silver
Done!!
Check MinIO silver bucket.
```

[ ]: