# DataEng S24: Data Transformation In-Class Assignment

**Submit**: Make a copy of this document and use it to record your results. Store a PDF copy of the document in your git repository along with any needed code. Submit the in-class activity submission form by Friday at 10:00 pm.

## A. [MUST] Initial Discussion Questions

Discuss the following questions among your working group members at the beginning of the week and place your own response into this space. If desired, also include responses from your group members.

1. In the lecture we mentioned the benefits of Data Transformation, but can you think of any problems that might arise with Data Transformation?

Data Loss: Transformation processes, especially when modifying data types or aggregating data, can result in data loss.
Introduction of Errors: If the transformation logic is flawed, it may introduce errors or inconsistencies in the data.

2. Should data transformation occur before data validation in your data pipeline or after?

Before Data Validation: If we want to transform raw data into a more usable format for validation, we might perform data transformation before data validation. This approach can make the data more consistent and easier to validate.
After Data Validation: Typically, it's more common to perform data validation first, ensuring the data meets certain criteria and quality standards before transforming it. This way, we avoid propagating bad data through your transformation process. After the data is validated and found to be clean and accurate, we can proceed with transformation.

## B. [MUST] Small Sample of TriMet data

Here is sample data for one trip of one TriMet bus on one day (February 15, 2023): bc_trip259172515_230215.csv It's in .csv format not json format, but otherwise, the data is a typical subset of the data that you are using for your class project.

We recommend that you use google Colab or a Jupyter notebook for this assignment, though any python environment should suffice.

Use the [pandas.read_csv()](pandas.read_csv()) method to read the data into a DataFrame.

```
#Mount to drive
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
data = pd.read_csv("/content/drive/MyDrive/bc_trip259172515_230215.csv")
data
```

| | EVENT_NO_TRIP | EVENT_NO_STOP | OPD_DATE | VEHICLE_ID | METERS | ACT_TIME | GPS_LONGITUDE | GPS_LATITUDE | GPS_SATELLITES | GPS_HDOP |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 259172515 | 259172517 | 15FEB2023:00:00:00 | 4223 | 40 | 20469 | -122.648137 | 45.493082 | 12 | 0.7 |
| 1 | 259172515 | 259172517 | 15FEB2023:00:00:00 | 4223 | 48 | 20474 | -122.648240 | 45.493070 | 12 | 0.8 |
| 2 | 259172515 | 259172517 | 15FEB2023:00:00:00 | 4223 | 57 | 20479 | -122.648352 | 45.493123 | 12 | 0.8 |
| 3 | 259172515 | 259172517 | 15FEB2023:00:00:00 | 4223 | 73 | 20484 | -122.648385 | 45.493262 | 12 | 0.7 |
| 4 | 259172515 | 259172517 | 15FEB2023:00:00:00 | 4223 | 112 | 20489 | -122.648347 | 45.493582 | 12 | 0.8 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 156 | 259172515 | 259172530 | 15FEB2023:00:00:00 | 4223 | 5834 | 21389 | -122.677057 | 45.528040 | 11 | 0.8 |
| 157 | 259172515 | 259172531 | 15FEB2023:00:00:00 | 4223 | 5838 | 21424 | -122.677003 | 45.528037 | 6 | 2.0 |
| 158 | 259172515 | 259172531 | 15FEB2023:00:00:00 | 4223 | 5858 | 21429 | -122.676765 | 45.528043 | 7 | 1.1 |
| 159 | 259172515 | 259172531 | 15FEB2023:00:00:00 | 4223 | 5889 | 21434 | -122.676370 | 45.528047 | 10 | 1.1 |
| 160 | 259172515 | 259172531 | 15FEB2023:00:00:00 | 4223 | 5918 | 21439 | -122.675990 | 45.528065 | 11 | 0.8 |

161 rows × 10 columns

Next steps: View recommended plots

# C. [MUST] Filtering

Some of the columns in our TriMet data are not generally useful for our class project. For example, our contact at TriMet told us that the EVENT_NO_STOP column is not used and can be safely eliminated for any type of analysis of the data.

Use [pandas.DataFrame.drop()](pandas.DataFrame.drop()) to filter the EVENT_NO_STOP column.

For this in-class assignment we won't need the GPS_SATELLITES or GPS_HDOP columns, so drop them as well.

```
# Use DataFrame.drop() to remove the EVENT_NO_STOP, GPS_SATELLITES, and GPS_HDOP columns
columns_to_drop = ['EVENT_NO_STOP', 'GPS_SATELLITES', 'GPS_HDOP']
data_filtered = data.drop(columns=columns_to_drop)
```

```
[13] # Display the first few rows of the filtered DataFrame
print("\nFirst few rows of the DataFrame after dropping EVENT_NO_STOP, GPS_SATELLITES, and GPS_HDOP columns:")
print(data_filtered.head())
```

```
First few rows of the DataFrame after dropping EVENT_NO_STOP, GPS_SATELLITES, and GPS_HDOP columns:
   EVENT_NO_TRIP            OPD_DATE  VEHICLE_ID  METERS  ACT_TIME  \
0      259172515  15FEB2023:00:00:00        4223      40     20469
1      259172515  15FEB2023:00:00:00        4223      48     20474
2      259172515  15FEB2023:00:00:00        4223      57     20479
3      259172515  15FEB2023:00:00:00        4223      73     20484
4      259172515  15FEB2023:00:00:00        4223     112     20489

   GPS_LONGITUDE  GPS_LATITUDE
0    -122.648137     45.493082
1    -122.648240     45.493070
2    -122.648352     45.493123
3    -122.648385     45.493262
4    -122.648347     45.493582
```

Next, start over and this time try filtering these same columns using the usecols parameter of the read_csv() method.

```
[22] data_path = "/content/drive/MyDrive/bc_trip259172515_230215.csv"
```

```
[23] # Specify the columns to include in the DataFrame
columns_to_include = ['EVENT_NO_TRIP', 'OPD_DATE', 'VEHICLE_ID', 'METERS', 'ACT_TIME', 'GPS_LONGITUDE', 'GPS_LATITUDE']
```

```
[24] # Read the CSV file and include only the specified columns
df = pd.read_csv(data_path, usecols=columns_to_include)
```

```
[25] # Display the first few rows of the filtered DataFrame
print("First few rows of the filtered DataFrame:")
print(df.head())
```

```
First few rows of the filtered DataFrame:
   EVENT_NO_TRIP            OPD_DATE  VEHICLE_ID  METERS  ACT_TIME  \
0      259172515  15FEB2023:00:00:00        4223      40     20469
1      259172515  15FEB2023:00:00:00        4223      48     20474
2      259172515  15FEB2023:00:00:00        4223      57     20479
3      259172515  15FEB2023:00:00:00        4223      73     20484
4      259172515  15FEB2023:00:00:00        4223     112     20489

   GPS_LONGITUDE  GPS_LATITUDE
0    -122.648137     45.493082
1    -122.648240     45.493070
2    -122.648352     45.493123
3    -122.648385     45.493262
4    -122.648347     45.493582
```

Why might we want to filter columns this way instead of using drop()?

Using the usecols parameter in pandas.read_csv() to filter columns directly during the file reading process can offer several advantages over using DataFrame.drop() after reading the file:

Performance: When you specify columns to include using usecols during file reading, only the specified columns are read from the file and loaded into memory. This can significantly reduce memory usage and improve performance, especially when dealing with large datasets.

Efficiency: Since only the specified columns are read into memory, the dataset is loaded more efficiently, and you avoid the overhead of loading unnecessary columns and then dropping them later.
Faster Data Loading: By only loading the columns you need, the data reading process can be faster because there is less data to process.
Clarity and Maintainability: Specifying the columns to include in the data reading stage makes the code more explicit and easier to understand. It indicates which columns are relevant to the analysis, and the DataFrame is immediately ready for use with only the necessary columns.
Reduced I/O: When you specify the columns to include, the file reading process only retrieves the data for the specified columns, reducing the amount of input/output operations.
Minimize Data Transformation: When you drop columns using DataFrame.drop(), you are creating a new DataFrame with the unnecessary columns removed. This transformation may not be necessary if you can directly load only the required columns.

# D. [MUST] Decoding

Notice that the timestamp for each bread crumb record is encoded in an odd way that might make analysis difficult. The breadcrumb timestamps are represented by two columns, OPD_DATE and ACT_TIME. OPD_DATE merely represents the date on which the bus ran, and it should be constant, unchanging for all breadcrumb records for a single day. The ACT_TIME field indicates an offset, specifically the number of seconds elapsed since midnight on that day.

We're not sure why TriMet represents the breadcrumb timestamps this way. We do know that this encoding of the timestamps makes automated analysis difficult. So your job is to decode TriMet's representation and create a new "TIMESTAMP" column containing a [pandas.Timestamp](pandas.Timestamp) value for each breadcrumb.

Suggestions:

- Use DataFrame.apply() to apply a function to all rows of your DataFrame
- The applied function should input the two to-be-decoded columns, then it should:
    - create a datetime value from the OPD_DATE input using datetime.strptime()
    - create a timedelta value from the ACT_TIME
    - add the timedelta value to the datetime value to produce the resulting timestamp

```
[27] from datetime import datetime, timedelta
     # Function to decode OPD_DATE and ACT_TIME and create a pandas.Timestamp
     def create_timestamp(row):
         # Convert OPD_DATE to a datetime object
         date = datetime.strptime(row['OPD_DATE'], '%d%b%Y:%H:%M:%S')

         # Convert ACT_TIME to a timedelta object (from seconds since midnight)
         time_offset = timedelta(seconds=row['ACT_TIME'])

         # Combine date and time_offset to create a timestamp
         timestamp = date + time_offset

         return timestamp
```

```
[28] # Apply the function to each row and create the TIMESTAMP column
     df['TIMESTAMP'] = df.apply(create_timestamp, axis=1)
```

```
[29] # Display the first few rows of the DataFrame with the new TIMESTAMP column
     print("First few rows of the DataFrame with the new TIMESTAMP column:")
     print(df.head())
```

```
First few rows of the DataFrame with the new TIMESTAMP column:
   EVENT_NO_TRIP            OPD_DATE  VEHICLE_ID  METERS  ACT_TIME  \
0      259172515  15FEB2023:00:00:00        4223      40     20469
1      259172515  15FEB2023:00:00:00        4223      48     20474
2      259172515  15FEB2023:00:00:00        4223      57     20479
3      259172515  15FEB2023:00:00:00        4223      73     20484
4      259172515  15FEB2023:00:00:00        4223     112     20489

   GPS_LONGITUDE  GPS_LATITUDE           TIMESTAMP
0    -122.648137     45.493082 2023-02-15 05:41:09
1    -122.648240     45.493070 2023-02-15 05:41:14
2    -122.648352     45.493123 2023-02-15 05:41:19
3    -122.648385     45.493262 2023-02-15 05:41:24
4    -122.648347     45.493582 2023-02-15 05:41:29
```

# E. [MUST] More Filtering

Now that you have decoded the timestamp you no longer need the OPD_DATE and ACT_TIME columns. Delete them from the DataFrame.

```
[31] # Remove the OPD_DATE and ACT_TIME columns
     df = df.drop(columns=['OPD_DATE', 'ACT_TIME'])
```

```
[32] # Display the first few rows of the DataFrame with the new TIMESTAMP column
     print("First few rows of the DataFrame with the new TIMESTAMP column and without OPD_DATE and ACT_TIME columns:")
     print(df.head())

     First few rows of the DataFrame with the new TIMESTAMP column and without OPD_DATE and ACT_TIME columns:
        EVENT_NO_TRIP  VEHICLE_ID  METERS  GPS_LONGITUDE  GPS_LATITUDE  \
     0      259172515        4223      40    -122.648137     45.493082
     1      259172515        4223      48    -122.648240     45.493070
     2      259172515        4223      57    -122.648352     45.493123
     3      259172515        4223      73    -122.648385     45.493262
     4      259172515        4223     112    -122.648347     45.493582

                 TIMESTAMP
     0 2023-02-15 05:41:09
     1 2023-02-15 05:41:14
     2 2023-02-15 05:41:19
     3 2023-02-15 05:41:24
     4 2023-02-15 05:41:29
```

# F. [MUST] Enhance

Create a new column, called SPEED, that is a calculation of meters traveled per second. Calculate SPEED for each breadcrumb using the breadcrumb's METERS and TIMESTAMP values along with the METERS and TIMESTAMP values for the immediately preceding breadcrumb record.

Utilize the pandas.DataFrame.diff() method for this calculation. diff() allows you to calculate the difference between a cell value and the preceding row's value for that same column. Use diff() to create a new dMETERS column and then again to create a new dTIMESTAMP column. Then use apply() (with a lambda function) to calculate SPEED = dMETERS / dTIMESTAMP. Finally, drop the unneeded dMETERS And dTIMESTAMP columns.

**Question**: What is the minimum, maximum and average speed for this bus on this trip? (Suggestion: use the Dataframe.describe() method to find these statistics)

```
[35] # Calculate the difference between rows for METERS and TIMESTAMP
     df['dMETERS'] = df['METERS'].diff()
     df['dTIMESTAMP'] = df['TIMESTAMP'].diff().dt.total_seconds()  # Convert timedelta to seconds
```

```
[36] # Calculate SPEED as dMETERS / dTIMESTAMP
     df['SPEED'] = df['dMETERS'] / df['dTIMESTAMP']
```

```
[37] # Drop the unneeded dMETERS and dTIMESTAMP columns
     df.drop(columns=['dMETERS', 'dTIMESTAMP'], inplace=True)
```

```
[38] # Display the first few rows of the DataFrame with the SPEED column
     print("First few rows of the DataFrame with the SPEED column:")
     print(df.head())
```

```
First few rows of the DataFrame with the SPEED column:
   EVENT_NO_TRIP  VEHICLE_ID  METERS  GPS_LONGITUDE  GPS_LATITUDE  \
0      259172515        4223      40    -122.648137     45.493082
1      259172515        4223      48    -122.648240     45.493070
2      259172515        4223      57    -122.648352     45.493123
3      259172515        4223      73    -122.648385     45.493262
4      259172515        4223     112    -122.648347     45.493582

            TIMESTAMP  SPEED
0 2023-02-15 05:41:09    NaN
1 2023-02-15 05:41:14    1.6
2 2023-02-15 05:41:19    1.8
3 2023-02-15 05:41:24    3.2
4 2023-02-15 05:41:29    7.8
```

```
[39] df['SPEED'] = df['SPEED'].fillna(method='bfill')
```

```
[41] # Calculate and display the summary statistics for the SPEED column
     summary_statistics = df['SPEED'].describe()
     # Displaying the statistics
     print("Summary statistics for SPEED:")
     print(summary_statistics)
     # Extracting the minimum, maximum, and average speed
     min_speed = summary_statistics['min']
     max_speed = summary_statistics['max']
     average_speed = summary_statistics['mean']
     # Display the minimum, maximum, and average speed
     print("\nMinimum speed: {:.2f} m/s".format(min_speed))
     print("Maximum speed: {:.2f} m/s".format(max_speed))
     print("Average speed: {:.2f} m/s".format(average_speed))
```

```
Summary statistics for SPEED:
count    161.000000
mean       7.192254
std        4.429027
min        0.000000
25%        3.800000
50%        6.400000
75%       10.800000
max       17.400000
Name: SPEED, dtype: float64

Minimum speed: 0.00 m/s
Maximum speed: 17.40 m/s
Average speed: 7.19 m/s
```

# G. [SHOULD] Larger Data Set

Here is breadcrumb data for the same bus TriMet for the entire day (February 15, 2023): bc_veh4223_230215.csv

Do the same transformations (parts C through F) for this larger data set. Be careful, you might need to treat each trip separately. For example, you might need to find all of the unique values for the EVENT_NO_TRIP column and then do the transformations separately on each trip.

**Questions**:
What was the maximum speed for vehicle #4223 on February 15, 2023?

Where and when did this maximum speed occur?

What was the median speed for this vehicle on this day?

```
[45] # Drop the unneeded columns
     data.drop(columns=['OPD_DATE', 'ACT_TIME', 'GPS_SATELLITES', 'GPS_HDOP'], inplace=True)

[46] # Calculate dMETERS and dTIMESTAMP
     data['dMETERS'] = data['METERS'].diff()
     data['dTIMESTAMP'] = data['TIMESTAMP'].diff().dt.total_seconds()  # Convert timedelta to seconds

[47] # Calculate SPEED as dMETERS / dTIMESTAMP
     data['SPEED'] = data['dMETERS'] / data['dTIMESTAMP']

[48] # Drop the unneeded dMETERS and dTIMESTAMP columns
     data.drop(columns=['dMETERS', 'dTIMESTAMP'], inplace=True)

[49] # Calculate max speed, and median speed for each trip
     max_speed = data['SPEED'].max()
     median_speed = data['SPEED'].median()

[50] # Find the row with maximum speed
     max_speed_row = data[data['SPEED'] == max_speed].iloc[0]

[51] # Print results
     print(f"Maximum speed for vehicle #4223 on February 15, 2023: {max_speed:.2f} m/s")
     print(f"Where and when did this maximum speed occur:")
     print(f"  GPS Longitude: {max_speed_row['GPS_LONGITUDE']}")
     print(f"  GPS Latitude: {max_speed_row['GPS_LATITUDE']}")
     print(f"  Timestamp: {max_speed_row['TIMESTAMP']}")

     print(f"\nMedian speed for this vehicle on this day: {median_speed:.2f} m/s")

Maximum speed for vehicle #4223 on February 15, 2023: 17.40 m/s
Where and when did this maximum speed occur:
  GPS Longitude: -122.660822
  GPS Latitude: 45.505452
  Timestamp: 2023-02-15 05:44:49

Median speed for this vehicle on this day: 7.20 m/s
```

# H. [ASPIRE] Full Data Set

Here is breadcrumb data for all TriMet vehicles for the entire day (February 15, 2023):
bc_230215.csv

Do the same transformations (parts C through F) for the entire data set. Again, beware that simple transformations developed in parts C through F probably will need to be modified for the full data set which contains interleaved breadcrumbs from many vehicles.

**Questions**:
What was the maximum speed for any vehicle on February 15, 2023?

Where and when did this maximum speed occur?

Which vehicle had the fastest mean speed for any single trip on this day? Which vehicle and which trip achieved this fastest average speed?