COMP63301-Data Engineering Concepts

Workshop 1 Description for Students

1 Introduction

During this workshop, you will explore a real-world data engineering scenario centred around scooter rental activity in Albuquerque, California — a popular destination for summer holidays. With its scenic routes and vibrant tourist culture, Albuquerque attracts both locals and visitors who frequently rent electric scooters to explore the area. These rentals are facilitated through an online application, and each scooter is tracked via GPS, generating rich datasets that include trip start and end times, locations, and vehicle identifiers.

Despite the technological sophistication of GPS tracking, historical data reveals notable issues with data completeness and reliability. You will delve into these imperfections, learning how to identify some of the most common data quality challenges in large-scale, timeseries datasets. This use case not only offers technical depth but also connects with the seasonal habits of young people and tourists — making it both relevant and engaging.

The primary objective of this workshop is to equip you with the practical skills required to explore and analyse time-series datasets using Python — one of the most widely adopted programming languages in data science and engineering. Working with raw, unprocessed CSV files containing GPS-tracked scooter rental data from Albuquerque, California, you will learn how to handle real-world data that is often messy and incomplete. This hands-on experience will foster a deeper understanding of how to interrogate datasets, uncover their structure and distribution, and identify common defects such as missing or inconsistent entries.

2 Before Starting

To begin working on the exercises for this workshop, you should first download the two CSV files (associated with this workshop) provided on the course's Canvas space. These files contain the raw GPS-tracked scooter rental and geo-coded street location data that will be used throughout the session. It is essential that you ensure you have access to a Python environment — either by installing Python on you personal laptop or by using the University's "Apps Anywhere" platform, which provides remote access to the University's Python installation.

Once the files are downloaded and Python is accessible, you should verify that you can open and inspect the CSV files using basic Python commands. Familiarity with tools or any preferred Python IDE can be beneficial. This initial setup will ensure a smooth start to the workshop and allow you to focus on exploring and analysing the data effectively.

3 Exploratory Data Analysis

Before embarking on a data refinement process, it is essential to first understand the structure and characteristics of your dataset. This initial phase, known as **Exploratory Data Analysis (EDA)**, enables you to gain insight into the shape and distribution of your data, the number of rows and columns, the data types present, and the range of values within each column.

While more advanced statistical techniques such as distribution analysis and skewness evaluation are available, this workshop focuses on developing a rapid and practical understanding of the dataset. This foundational knowledge will prepare you for the subsequent data refinement tasks.

3.1 Reading csv files into DataFrames

To begin your exploration, you will need to import the `pandas` library and load the dataset from the provided CSV file. The following Python commands will help you get started:

Python Code

```
import pandas as pd
df = pd.read_csv('scooter.csv')
```

Once the data is loaded into a `DataFrame`, you will be able to inspect its contents, identify patterns, and begin your analysis. This hands-on approach will help you develop confidence in working with real-world datasets and prepare you for more advanced data engineering tasks.

3.2 Exploring the Data

Once your Python environment is ready and the dataset has been loaded, the first step is to begin exploring the structure of the data. Rather than printing the entire dataset immediately, it is advisable to first inspect the column names and their corresponding data types. This will provide a clear overview of the dataset's schema and help you understand the nature of the data you are working with.

Use the following commands to examine the columns and data types:

Inspecting DataFrame Structure

```
df.columns
Index(['month', 'trip_id', 'region_id', 'vehicle_id',
'started_at', 'ended_at', 'DURATION',
'start_location_name', 'end_location_name',
'user_id', 'trip_ledger_id'],
dtype='object'

df.dtypes
```

Student Task

Question: After running the command, carefully observe the output.

What do you notice about the rows that have been returned?

Now that you are familiar with the dataset's columns and data types, it is time to begin examining the actual data. A useful starting point is to view a sample of the records using the `head()` method, which displays the first few rows of the DataFrame.

Viewing Initial Records

```
df.head()
```

This command will return the first five rows by default, showing a snapshot of the dataset.

Student Task

Question: After running the command, carefully observe the output.

What do you notice about the rows that have been returned?

To view the last few rows instead, you can use the `tail()` method:

Viewing Final Records

```
df.tail()
```

Both `head()` and `tail()` accept an optional integer parameter to specify how many rows to display. For example, `df.head(10)` will show the first ten rows.

If you wish to display all columns without cropping, you can adjust the display settings using the following command:

Expanding Column Display

```
pd.set_option('display.max_columns', 500)
```

This will allow you to view all columns when using `head()` or `tail()`, although the output may wrap depending on your screen width.

If your interest lies in a specific column, such as `DURATION`, you can access it directly using dictionary-style indexing:

Accessing a Single Column

```
df['DURATION']
```

Student Task

Question: After running the command, carefully observe the output.

What do you notice about the rows that have been returned?

As you continue exploring the dataset, you may notice that when displaying a large number of rows, the output is truncated—represented by ellipses (`...`)—to conserve space. This behaviour is similar to what occurs with columns and is the result of combining the `head()` and `tail()` methods. Although it is possible to adjust the number of rows shown using the `display.max_rows` option, doing so is not necessary for our current exploration.

Just as you can view a single column, you may also retrieve a subset of columns by passing a list of column names using double square brackets. For example, to examine the trip ID, duration, and starting location, use the following command:

Selecting Multiple Columns

```
df[['trip_id', 'DURATION', 'start_location_name']]
```

At this stage, you may wish to examine a random subset of the dataset to gain a quick impression of its contents. This can be achieved using the `sample()` method, which allows you to specify the number of rows to retrieve.

Sampling Rows from the DataFrame

```
df.sample(n)
```

Replace `n` with the number of rows you wish to view. For example, `df.sample(5)`.

Student Task

Question: After running the `sample()` command, carefully observe the output.

What do you notice about the rows that have been returned?

3.3 Slicing and Selecting Data

In this section, you will learn how to extract specific rows from your dataset using slicing, indexing, and conditional filtering. These techniques are essential for narrowing down your analysis and working with targeted subsets of data.

To retrieve the first ten rows of the DataFrame, use the following slicing syntax:

Slicing the First Ten Rows

```
df[:10]
```

Student Task

Task: Run the command above and compare your output with the student(s) next to you.

Did you obtain the same result? If not, what might explain the difference?

To retrieve all rows from the tenth onward, use:

Slicing from Row 10 to the End

```
df[10:]
```

To extract rows starting from index 3 up to (but not including) index 9:

Slicing a Specific Range

```
df[3:9]
```

Student Task

Task: Compare your output with your peers.

Was the result consistent across different machines?

If you know the exact index of a row, you can retrieve it directly using the `loc()` method:

Selecting a Specific Row by Index

```
df.loc[34221]
```

Student Task

Task: Run the command and compare your result with others.

Did everyone get the same row?

To access a specific value from a given row and column, use the `at()` method:

Accessing a Single Value

```
df.at[2, 'DURATION']
```

3.4 Filtering Data Based on Conditions

You can also filter rows based on specific conditions. For example, to select all rows where the user ID is `8417864`, use:

Filtering with `where()`

```
user = df.where(df['user_id'] == 8417864)
user
```

Student Task

Task: Compare your output with your neighbour's.

What do you observe in the filtered rows?

To achieve a similar result without including rows with missing values, use this alternative notation:

Filtering Without NaNs

```
df[(df['user_id'] == 8417864)]
```

Student Task

Task: Compare this output with the previous one.

What is the key difference between the two methods?

You can also combine multiple conditions. For instance, to filter rows where the user ID is `8417864` and the trip ledger ID is `1488838`, use:

Combining Conditions with `where()`

```
one = df['user_id'] == 8417864
two = df['trip_ledger_id'] == 1488838
df.where(one & two)
```

Student Task

Task: Run the combined condition and compare your result with others.

Did you all retrieve the same filtered rows?

3.5 Analysing the Data

3.5.1 Analysing the Dataset with Descriptive Statistics

Now that you've familiarised yourself with the structure and contents of the dataset, it's time to begin analysing it. One of the most useful tools for initial statistical exploration is the `describe()` method. This function provides a summary of key statistics for each numeric column, including count, mean, standard deviation, minimum, and maximum values.

Generating Descriptive Statistics

```
df.describe()
```

Student Task

Task: Run the command above and examine the output.

How useful is the `describe()` method when applied to the data in `scooter.csv`?

Consider how this method would behave if applied to a column containing age data. What kinds of anomalies might it help you detect?

Sometimes, applying `describe()` to a single column can offer more targeted insights. For example, let's examine the `start_location_name` column:

Describing a Single Column

```
df['start_location_name'].describe()
```

Student Task

Task: Run the command and compare your output with the student(s) next to you.

Were the results identical? What patterns or anomalies can you observe in the output?

3.5.2 Exploring Value Frequencies

Another useful method for understanding your data is `value_counts()`, which returns the frequency of each unique value in a column. This is particularly helpful for identifying common or rare entries.

Let's apply it to the `DURATION` column:

Counting Unique Values

```
df['DURATION'].value_counts()
```

Student Task

Task: Run the command above and inspect the output.

What do you notice about the distribution of trip durations? Are there any durations that appear unusually often or seem implausible?

3.5.3 Identifying Missing Values

In real-world datasets, missing values are common and must be handled carefully. The `value_counts()` method, by default, excludes missing entries. However, you can include them by setting the `dropna` parameter to `False`. For example, to inspect the frequency of values in the `end_location_name` column, including missing entries:

Counting Values Including NaNs

```
df['end_location_name'].value_counts(dropna=False)
```

Student Task

Task: Run the command above and compare your output with the student(s) next to you.

Did you obtain the same result? What do you observe about the number of missing entries?

To get a broader view of missing data across the entire dataset, you can use the `isnull()` method in combination with `sum()` to count the number of missing values per column:

Summarising Missing Values

```
df.isnull().sum()
```

Student Task

Task: Compare this output with the previous one.

How do the results differ? What additional insights does this summary provide?

3.5.4 Grouping Data Using Bins

Another useful technique is grouping numeric data into intervals, or "bins". Although the scooter dataset may not contain ideal columns for binning, you can still experiment with a numeric column such as `trip_id`:

Creating Binned Groups

```
df['trip_id'].value_counts(bins=10)
```

Student Task

Task: Run the command and compare your output with your peers.

What do you observe from the grouped data? How might this technique be useful if applied to a column such as age?

3.6 Addressing Common Data Issues

3.6.1 Dropping Unnecessary Data

Before making any modifications to your dataset, it is important to consider which fields are actually relevant to your analysis. For instance, in the e-scooter dataset, the column `region_id` simply encodes the location "Albuquerque", which is already known. As such, it does not contribute meaningful information and can be removed.

To drop a column from the DataFrame, use the `drop()` method and specify the column name:

Removing a Column

```
df.drop(columns=['region_id'], inplace=True)
```

The `inplace=True` argument ensures that the original DataFrame is updated directly.

To remove a specific row, you can use the same method but specify the index instead:

Removing a Row

```
df.drop(index=[34225], inplace=True)
```

3.6.2 Dropping Data Based on Conditions

Sometimes, you may wish to remove rows or columns based on certain conditions—such as missing values. The `dropna()` method provides a flexible way to do this. It accepts several parameters:

- **axis**: Determines whether to drop rows (`0`) or columns (`1`). Defaults to rows.
- **how**: Specifies whether to drop if `any` or `all` values are missing. Defaults to `any`.
- **thresh**: Allows you to set a minimum number of non-null values required to retain the row or column.
- **subset**: Lets you target specific columns or rows for evaluation.
- **inplace**: If set to `True`, modifies the original DataFrame.

Here is an example of how to use `dropna()`:

Dropping Rows with Missing Values

```
df.dropna(axis=0, how='any', inplace=True)
```

Student Task

Task: Try dropping rows with missing values using the command above.

What effect does this have on your dataset? How many rows were removed?

3.7 Changing the Structure of Tabular Data

3.7.1 Filling Missing Values and Filtering Rows

Before applying more advanced filtering techniques, it's worth considering whether missing values should be removed or replaced. In many cases, retaining rows with missing data is preferable, especially if the gaps can be filled with a sensible default.

To replace missing entries in a column or across the DataFrame, use the `fillna()` method. For example, to fill all missing values with a default duration:

Filling Missing Values

```
df.fillna(value='00:00:00', axis='columns')
```

Student Task

Task: Run the command above.

What effect does this have on your dataset? Which columns were affected?

3.7.2 Filtering Rows Based on Conditions

Rather than manually iterating through the DataFrame to remove rows, you can apply filters and drop rows based on specific conditions. For instance, if you wish to exclude all records from the month of May, you can first isolate those rows:

Filtering Rows for May

```
may = df[df['month'] == 'May']
may
```

Then, remove those rows from the original DataFrame by referencing their index:

Dropping Filtered Rows

```
df.drop(index=may.index, inplace=True)
```

To confirm that the rows have been removed, you can inspect the distribution of months:

Checking Remaining Months

```
df['month'].value_counts()
```

Student Task

Task: Run the filtering and dropping commands above.

Is the month of May still present in your dataset? What months remain, and how many records are associated with each?

3.7.3 Standardising Column Values

Consistency in data formatting is essential, especially when working across multiple datasets or systems. In the scooter dataset, one column—`DURATION`—is written in uppercase, while others use lowercase. This inconsistency can lead to confusion and errors in queries.

To standardise the values in a column, you can apply string methods such as `upper()`, `lower()`, or `capitalize()`:

Converting Column Values to Uppercase

```
df['month'] = df['month'].str.upper()
df['month'].head()
```

3.7.4 Creating New Columns Based on Conditions

You can add new columns to a DataFrame using the assignment format `df['new_column'] = value`. This assigns the same value to all rows. However, to assign values conditionally, you can iterate through the DataFrame:

Creating a Column with Conditional Values

```
for i, r in df.head().iterrows():
if r['trip_id'] == 1613335:
df.at[i, 'new_column'] = 'Yes'
else:
df.at[i, 'new_column'] = 'No'
df[['trip_id', 'new_column']].head()
```

Student Task

Task: Run the code above and compare your output with your peers.

Did you obtain the same result? What patterns do you notice in the new column?

For better performance, use `loc()` to apply conditions directly:

Efficient Conditional Assignment

```
df.loc[df['trip_id'] == 1613335, 'new_column'] = '1613335'
df[['trip_id', 'new_column']].head()
```

Student Task

Task: Compare this output with the previous method.

Was the result the same? Which method was faster or more readable?

3.7.5 Converting Data Types

Some columns may appear as generic objects but actually contain structured data, such as dates. For example, the `started_at` column contains datetime values but is stored as an object. Filtering by date will not work correctly unless the column is converted.

Attempting to filter before conversion:

Filtering Without Conversion

```
when = '2019-05-23'
x = df[df['started_at'] > when]
len(x)   # Returns 34226 (all rows)
```

To fix this, convert the column using `to_datetime()`:

Converting to Datetime Format

```
df['started_at'] = pd.to_datetime(df['started_at'], format='%m/%d/%Y
%H:%M')
df.dtypes
```

Student Task

Task: Run the conversion and inspect the data types.

Was the column successfully converted? How does this affect filtering and analysis?

3.8 Basic Data Integration for Data Enrichment

3.8.1 Geocoding and Enriching Location Data

Although the e-scooter dataset contains location names, it lacks geographic coordinates, which are essential for mapping and spatial analysis. To address this, we can enrich the dataset by geocoding the location names.

Fortunately, the City of Albuquerque provides a public geocoding service. For this exercise, we'll begin by identifying the five most frequent starting locations:

Extracting Frequent Locations

```
new = pd.DataFrame(df['start_location_name'].value_counts().head())
new.reset_index(inplace=True)
new.columns = ['address', 'count']
new
```

Student Task

Task: Run the code above and compare your output with your peers.

Do the top five addresses match? What patterns do you notice in the frequency counts?

3.8.2 Cleaning Address Data for Geocoding

To prepare the addresses for geocoding, we need to simplify them. For example, intersections like "Central @ Tingley" should be rewritten using "and" instead of "@". We also only need the street portion of the address:

Cleaning Address Strings

```
n = new['address'].str.split(pat=',', n=1, expand=True)
replaced = n[0].str.replace("@", "and")
new['street'] = replaced
new
```

Student Task

Task: Inspect the cleaned street names.

Were all addresses correctly formatted for geocoding? What changes were made?

3.8.3 Joining Geocoded Data

We now enrich our dataset by joining it with a CSV file containing geocoded coordinates. Load the file using:

Loading Geocoded Data

```
geo = pd.read_csv('geocodedstreet.csv')
geo
```

Student Task

Task: Load the geocoded data and inspect the columns.

Do you see the expected street names and coordinate values?

To combine the datasets, you can use either a join or a merge. Here's how to perform a join:

Joining DataFrames

```
joined = new.join(other=geo, how='left', lsuffix='_new',
rsuffix='_geo')
joined[['street_new', 'street_geo', 'x', 'y']]
```

Student Task

Task: Perform the join and inspect the result.

Were the coordinates correctly matched to the street names?

Alternatively, use `merge()` to avoid duplicate columns:

Merging DataFrames

```
merged = pd.merge(new, geo, on='street')
merged.columns
```

Student Task

Task: Compare the output of `merge()` with that of `join()`.

Which method produced a cleaner result? Why might you choose one over the other?

4 Analysing Trip Patterns Using Queries

In the following, we provide two examples of how the data can be queried to obtain further understanding.

In this section, we present two example queries designed to deepen your understanding of the scooter trip dataset. Your task is to translate each query into Python code, execute it, and interpret the results.

Query 1: Shortest Trip per Location Pair

For each unique pair of starting and ending locations in the dataset (i.e., each distinct combination of `<start_location_name, end_location_name>`), identify the following:

- The user (`user_id`) who completed the trip with the shortest recorded duration.
- The corresponding trip identifier (`trip_id`) for that journey.
- The exact duration of the trip.
- The month in which this shortest trip occurred.

Student Task

Task: Write and run the Python code to implement Query 1.

What do you observe from the results? Are there any surprising patterns or anomalies?

Query 2: Longest Trip per Location Pair

For each unique pair of starting and ending locations in the dataset, determine:

- The user (`user_id`) who completed the trip with the longest recorded duration.
- The corresponding trip identifier (`trip_id`) for that journey.
- The exact duration of the trip.
- The month in which this longest trip occurred.

Student Task

Task: Write and run the Python code to implement Query 2.

Compare the results with those from Query 1. What differences do you notice in trip patterns and durations?