Project 1 Analyzing New York City Taxi Data

Juan Gonzalo Quiroz Cadavid[1], Priit Peterson[1], Shivam Maheshwari[1], and Venkata Narayana Bommanaboina[1]

[1]University of Tartu
juangonzalo@ut.ee, email, email, email

March 9, 2025

Master plan 1

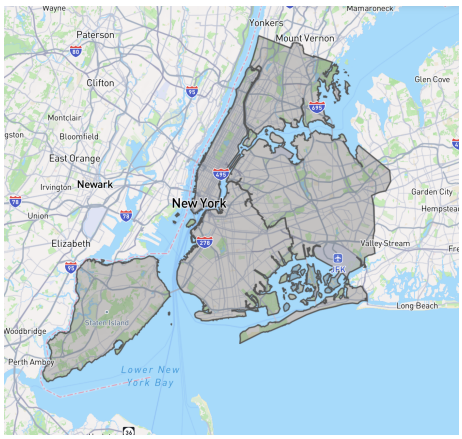
- **Geospatial preparation**: Load into memory the geo-json, store in decremental order base on the area of each borough, broadcast it to be read-only across all worker nodes under the spark context.
- **Data features**:
  - **Borough** name for each tuple of lat lon on pickup and drop off, using the previous data generated on geospatial preparation and a UDF to add a new field.
  - **Duration** using the delta time between the date time feature for both pick-up and drop-off.
- **Cleansing data**: Remove rows where lat/lon parameters are 0; duration is negative and where its duration is bigger than 4 hours (which means there could be an error as it is a long trip)
- **Window data**: Aggregate over each drive per same taxi driver, calculate the idle when this is less than 4 hours, more than 4 would be considered as a recession.

Geospatial preparation 2

The Geojson is a json file coded following the RFC 7946 standard. It contains 5 borough across New york city, represented by 104 different polygons, whose area are in the range [0.027193754466707243 Km, $5.094442942481346e^9$ Km].

The GeoJson file was read using json, each feature were loaded into a shapely shape object. For each feature a dict element were created containing the shape, borough name and code and the shape instance. This Dictionary will we later sorted and broadcasted across nodes.

Figure 1 shows the data store on the geojson, code 1 illustrate the process of extracting features, and loading them into the dictionary. The resulting dictionary is sorted on code 2 using python sort function, and broadcast using pyspak sparkContext broadcast function on code 3.



(a) nyc-boroughs.geojson represend using geojson

Figure 1

Listing 1: data geometrics creation

```
goeJson =(
    open("./data/nyc-boroughs.geojson", "r")
    .read()
)
y = json.loads(goeJson)
data_geomtries = []

for feature in y['features']:
    boroughCode = feature['properties']['boroughCode']
    borough = feature['properties']['borough']
    poly = feature['geometry']
    poly_shape = shape(poly)
    data_geomtries.append({
        "boroughCode": boroughCode,
        "borough": borough,
        "shape": poly_shape.area,
        "thing": poly_shape,
    })
```

Listing 2: Sort geometries by shape

```
sorted_geoms = sorted(
    data_geomtries,
    key=lambda poly: poly['shape'],
    reverse=True,
)
```

Listing 3: Broadcast dict across worker nodes using sparkContext broadcast feature.

```
broadcast_geoms = spark.sparkContext.broadcast(sorted_geoms)
```

Data features 3

3.1 Borough name

A UDF function were created. The function received lat and lon (See code 4), converted them into a shapely Point, iterated over all the geoms checking if the point is contained by any polygon, if so the name of the polygon would be returned. The method is registered as a UDF function (See code 5) to be used later to create two columns for pickup and dropoff locations (See code 6)

Listing 4: UDF method

```
def getBorough(lat:float, lon:float) -> str:
    geoms = broadcast_geoms.value
    p = Point(lon, lat)
    for geom in geoms:
        if geom["thing"].covers(p):
            return geom["borough"]
    return None
```

Listing 5: UDF creation

```
getBoroughUDF = F.udf(getBorough, StringType())
```

Listing 6: UDF usage

```
trip_df = ( trip_df
    .withColumn(
        "pickup_borough",
        getBoroughUDF(
            F.col("pickup_latitude"),
            F.col("pickup_longitude")
        )
    )
    .withColumn(
        "dropoff_borough",
        getBoroughUDF(
            F.col("dropoff_latitude"),
            F.col("dropoff_longitude")
        )
    )
)
```

3.2 Duration name

Duration (See Code 7) is calculated by transforming the dropoff and pickup datetime into seconds using unix_timestamp function from PySpark following the format "dd-MM-yy HH:mm". The duration is calculated by subtracting pickup to dropoff the once transformed into seconds.

Listing 7: Duration creation

```
trip_df = (
    .withColumn(
        "duration",
        F.unix_timestamp(
            F.col("dropoff_datetime"), "dd-MM-yy HH:mm"
        )
        - F.unix_timestamp(
            F.col("pickup_datetime"), "dd-MM-yy HH:mm"
        )
    )
)
```

Cleansing data 4

First, it will be removed all rows that does not contain a TaxiID, after it will remove rows where the lat and lon attributes are 0, and finally all rows whose duration is less than 0 would be removed. See code 8

Listing 8: Duration creation

```
trip_df = (
    trip_df.filter(
        ~F.isnull(F.col("taxiId"))
    )
    .filter(
        (F.col("pickup_latitude") != 0) &
        (F.col("pickup_longitude") != 0) &
        (F.col("dropoff_longitude") != 0) &
        (F.col("dropoff_latitude") != 0)
    )
    .filter(
        F.col("duration") > 0
    )
)
```

Query one: Utilization 5

For query one, we partitioned the data using the taxiID, ordered by their pickup ts, this will allow us to work per taxi ID; Then a new column is added for the previous pickup time for each row using Lag function. Later on we compute the idle for the trip, we summarized the total ride time, total idle time and computed utilization as $\frac{\text{total ride time}}{\text{total ride time+total idle time}}$. Code 9 Shows the process. Table 1 shows the first 10 rows of the resulting operation. Image 2 plot the distribution of the utilization across all taxi drivers, the top 10 and the buttom 10.

Listing 9: Query 1. Utilization

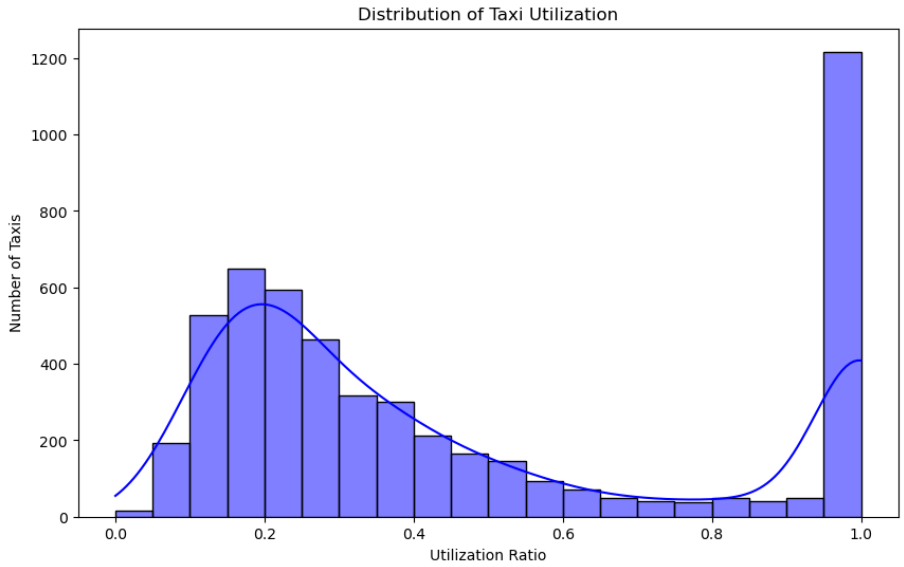
```
# Window partitioned by taxi_id, ordered by pickup_ts
w = Window.partitionBy("taxi_id").orderBy("pickup_ts")

trip_df = trip_df.withColumn("prev_dropoff_ts", F.lag("dropoff_ts").over(w))

# Idle time between consecutive trips (if gap <= 4 hours)
idle_expr = F.when(
    (F.col("pickup_ts") - F.col("prev_dropoff_ts") <= 4 * 3600) &
    (F.col("pickup_ts") - F.col("prev_dropoff_ts") >= 0),
    F.col("pickup_ts") - F.col("prev_dropoff_ts")
).otherwise(F.lit(0))

trip_df = trip_df.withColumn("idle_time", idle_expr)

util_df = trip_df.groupBy("taxi_id").agg(
    F.sum("Duration").alias("total_ride_time"),
    F.sum("idle_time").alias("total_idle_time")
)

util_df = util_df.withColumn(
    "utilization",
    F.col("total_ride_time") /
    (F.col("total_ride_time") + F.col("total_idle_time"))
)

print("=== Utilization per Taxi ===")
util_df.show(10, truncate=False)
```

taxi_id	total_ride_time	total_idle_time	utilization
000318C2E3E6381580E5C99910A60668	1560	2640	0.3714
002B4CFC5B8920A87065FC131F9732D1	1140	7020	0.1397
002E3B405B6ABEA23B6305D3766140F1	2760	2040	0.5750
0030AD2648D81EE87796445DB61FCF20	1980	720	0.7333
0035520A854E4F2769B37DAF5357426F	3060	5160	0.3723
0036961468659D0BFC7241D92E8ED865	1920	2160	0.4706
0038EF45118925A510975FD0CCD67192	1440	6240	0.1875
003D87DB553C6F00F774C8575BC8444A	420	0	1.0000
003EEA559FA61800874D4F6805C4A084	1020	12600	0.0749
0053334C798EC6C8E637657962030F99	420	0	1.0000

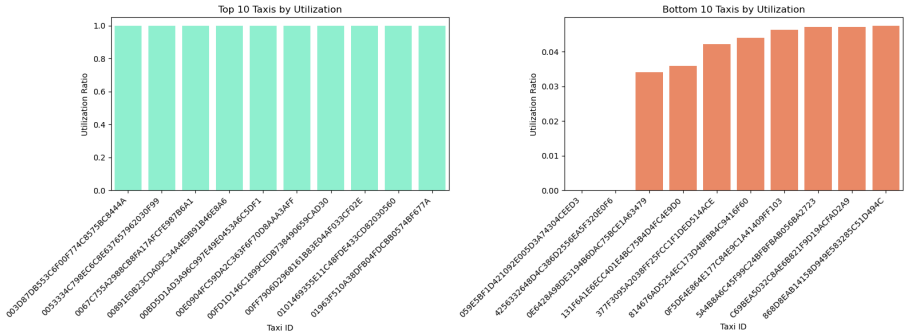
Table 1: Taxi Utilization Data

Query 2: Average Time to Find Next Fare per Destination Borough 6

Taxi drivers are spited by their respective ID using Window (See code 10), Compute $time_to_next_fare = next_pickup_ts - dropoff_ts$ if the delta is less than 4 hours, finally we group them by $dropoff_borough$ to get the average. Table 2 presents the results for all 5



(a) Distribution of taxi utilization



(b) Top 10 vs buttom 10

Figure 2: Newyork taxi utilization

borough in New york, Unknown are the points whose lat lon attributes does not fit on the geojson features. The same data is represented on figure 3.

Listing 10: Duration creation

```
w2 = Window.partitionBy("taxi_id").orderBy("pickup_ts")

trip_df = trip_df.withColumn("next_pickup_ts", F.lead("pickup_ts").over(w2))

trip_df = trip_df.withColumn(
    "time_to_next_fare",
    F.when(
        (F.col("next_pickup_ts") - F.col("dropoff_ts") <= 4*3600) &
        (F.col("next_pickup_ts") - F.col("dropoff_ts") >= 0),
        F.col("next_pickup_ts") - F.col("dropoff_ts")
    ).otherwise(F.lit(None))
)

wait_time_df = trip_df.groupBy("dropoff_borough") \
    .agg(F.avg("time_to_next_fare").alias("avg_time_to_next_fare"))

print("=== Average Time to Find Next Fare per Destination Borough ===")
wait_time_df.show(10, truncate=False)
```

Dropoff Borough	Avg. Time to Next Fare (s)
Manhattan	2677.15
Unknown	3298.60
Queens	4611.89
Brooklyn	4214.11
Bronx	3926.67
Staten Island	9630.00

Table 2: Average Time to Find Next Fare per Destination Borough

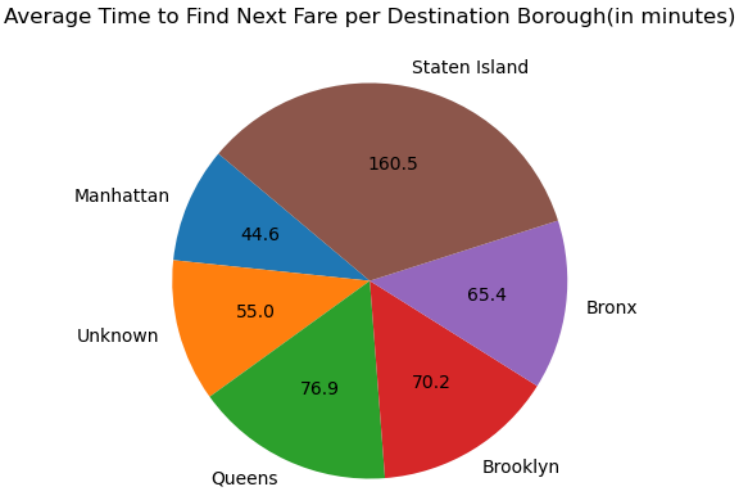


Figure 3: Average Time to Find Next Fare per Destination Borough(in minutes)

Query 3: Number of Trips that Started and Ended in the Same Borough 7

There is a total of **16821** trips that start and end in the same borough, code 12 filter by comparing pickup and dropoff borough feature, then using count we could extract the number of occurrences.

Listing 11: Same borough

```
same_borough_count = trip_df.filter(
    F.col("pickup_borough") == F.col("dropoff_borough")
).count()

print(
    "Number of trips that start and end in the same borough:",
    same_borough_count
)
```

Query 4: Number of Trips that Started in One Borough and Ended in Another 8

There is a total of **3179** trips that start and end in different borough, code 12 filter by comparing pickup and dropoff borough feature, then using count we could extract the number of occurrences.

Listing 12: Different borough

```
diff_borough_count = trip_df.filter(
    F.col("pickup_borough") != F.col("dropoff_borough")
).count()
```

Conclusion 9

We have:

1. Loaded and cleansed NYC Taxi data
2. Enriched it with borough information (pickup and dropoff)
3. Computed key metrics:
    - Utilization per taxi
    - Average wait time for next fare
    - Same vs. different borough trip counts
4. Created basic visualizations (bar chart, histogram)