Public Transport Route Recommendation Using NYC's Taxi Trip Data

Wan-Yu Lin **Charvi Gupta** **Priyanka Narain**

Graduate School of Arts & Science, New York University

wl2484@nyu.edu cg4177@nyu.edu pn2182@nyu.edu

Abstract

This project introduces a pragmatic approach for recommending public transport routes in New York City (NYC) using the Taxi and Limousine Commission (TLC) Trip Record Data. Our approach involves computing the frequency of taxi trip pairs and identifying the top K pairs to establish prevalent routes. Given the absence of explicit travel paths in the dataset, we propose a simple method to employ Dijkstra's Algorithm to approximate coarse-grained routes based on neighboring relations between taxi zones. Our top recommended routes expose a deficit in convenient or direct public transport options for journeys from the upper right to the lower left of Manhattan, and the top 10 routes collectively account for 9.845% of the 2 billion taxi trips in the past 3 years. This study contributes valuable insights into optimizing public transport recommendations and has implications for enhancing transportation infrastructure planning in urban areas.

1. Introduction

New York City (NYC), the most populated city in the United States, boasts a multifaceted transportation network. This includes a vast subway system, one of the largest globally, comprehensive bus services across all five boroughs, ferry services, and a significant presence of borough taxis distributed throughout the city.

Taxis are one of the major modes of transportation in New York City. Taxi cabs are operated by private companies in New York and are licensed by the New York City Taxi and Limousine Commission (TLC). TLC, created in 1971, is the agency responsible for licensing and regulating New York City's medallion (yellow) taxis and street hail livery (green) taxis, for-hire vehicles (FHVs), High Volume For Hire Services (HVFHV).

Despite a robust public transport system, the number of taxi trips has increased steadily over the years. In 2020, the number of yearly trips was 45046979, while in 2023, 193098701 trips were recorded, which is an increase of a whopping 328%.

In our project, we leveraged the rich repository of NYC's taxi trip data, TLC Trip Record Data [2] (introduced in section 2.2), to conduct a comprehensive analysis of taxi trips. This study aimed to discover common patterns in the taxi trip pick-up and drop-off zones. The existence of prevalent starting and ending location pairs suggests a high demand for transportation support between the pair locations and thus makes them good endpoints or intermediate stops for public transport routes.

We need a feasible approach to plan an actual route linking each prevalent pair of locations. We apply a simple method utilizing the neighboring relations between taxi zones and Dijkstra's algorithm to infer the shortest path between two locations as the recommended route. By mapping these routes and comparing them against the existing public transport network, we were able to suggest strategic additions to the public transport system. These recommendations are not just theoretical; we quantified the impact by estimating the total number of taxi trips that could be accommodated by these newly proposed routes.

2. Data Sources

We used 3 datasets for our analysis. Using the NYC Taxi Zones [1] dataset (introduced in section 2.1), we inferred the geometric (geo) coordinates of each taxi zone from the geo-coordinates of its borderlines. Then, with the TLC Trip Record Data [2] (introduced in section 2.2), we computed the occurrence of each pick-up and drop-off pair from taxi trips over the past 3 years. Lastly, by combining our results with the Taxi Zone Lookup Table [8] (introduced in section 2.3), we transform our route recommendation results to provide human-readable zone names as the outcome of our analysis.

2.1 NYC Taxi Zones

The NYC Taxi Zones [1] dataset is a CSV-formatted dataset of ~4MB size that provides geo-information of each taxi zone used in TLC Trip Record Data [2]. Table 1 shows a row of sample data.

Column	Value
OBJECTID	186

Shape_Leng	0.0246963902234
the_geom	"MULTIPOLYGON (((-73.99096832799995 40.74542088999988, -73.99141858599994 40.74480268199988, -73.99426258599996 40.74600398099987, -73.99709902899994 40.74720510199991, -73.99664704399989 40.74782272899996, -73.99619655499993 40.74843932799997, -73.99574855599987 40.74905219599989, -73.99529821499979 40.74967168299992, -73.99484363799989 40.75029330899991, -73.99437946899982 40.75093038999993, -73.99394825699984 40.75152222199995, -73.99346417699985 40.752190054999886, -73.99211708899992 40.75162207799989, -73.99062219999986 40.75099175399994, -73.98793687499989 40.749854966999926, -73.98776905899994 40.749787028999926, -73.98797357699985 40.74949077899986, -73.98823761499983 40.74913016899993, -73.9887140509998 40.748513273999954, -73.98916651699989 40.747895434999904, -73.98961683100002 40.74727642299989, -73.99006749199991 40.746659282999936, -73.99051761299985 40.746038637999895, -73.99096832799995 40.74542088999988)))"
Shape_Area	0.000037072941695
zone	Penn Station/Madison Sq West
LocationID	186
borough	Manhattan

Table 1. Raw data sample of NYC Taxi Zones [1]

We leveraged this dataset to derive the following regarding the boundary geo-coordinates of each taxi zone:

1. The representative geo coordinate (in latitude and longitude) of each location, i.e. average of the boundary geo-coordinates of the largest sub-region (i.e. the sub-region with the most coordinates) within the location
2. The big circle distances [11] (in meters) between each location and its neighbors
3. The neighbor zone graph

2.1.1 Data cleansing

```
case class BoroughTaxiZone(borough: String, location_id: Long, boundary: Array[Array[(Double, Double)]])

Wan-Yu Lin *
def cleanRawData(spark: SparkSession, rawDF: DataFrame): Dataset[BoroughTaxiZone] = {
  import spark.implicits._

  val pattern = """([1-9][0-9]*),MULTIPOLYGON \(\(\(\(.*)\)\)\),([a-zA-Z ]+)$""".r

  rawDF.select( col = "LocationID",  cols = "the_geom", "borough")
    .withColumnRenamed( existingName = "LocationID",  newName = "location_id")
    .map(r => {
      val rStr = r.mkString( sep = ",")
      val pattern(locationId, geom, borough) = rStr
      val boundary = geom.split( regex = "\)\)\), \(\(\(")
        .map(subRegionGeomStr => subRegionGeomStr.split( regex = ", ")
          .map(coordinateStr => {
            val coordinate = coordinateStr.split( regex = " ").map(_.toDouble)
            (coordinate(0), coordinate(1))
          }))
        .sortWith(_.length > _.length)
      BoroughTaxiZone(borough, locationId.toLong, boundary)
    })
}
```

Figure 1. Code snippet for cleaning NYC Taxi Zones [1]

To fulfill the 3 aim usages, we cleaned the NYC Taxi Zones [1] dataset by keeping only 3 columns: (1) *borough*, (2) *LocationID* (renamed as *location_id*), and (3) *the_geom*, which consists of a list of geo-coordinates where each internal list corresponds to the boundary geo coordinates of a taxi zone's sub-region. Then split the third column by ")), ((" and transform each into a list of `(Double, Double)` to get the boundary geo-coordinates of each sub-region in the zone, and sort the list descendingly by the number of coordinates in the sub-region. Figure 1 shows the corresponding code snippet.

```
def saveSummarizedCleanData(cleanDS: Dataset[BoroughTaxiZone], path: String, boroughs: Seq[String]): Unit = {
  // define udfs
  val avgLatOfTheLargestSubRegion: Array[Array[(Double, Double)]] => Double = { boundary =>
    val (count, latSum) = boundary.head.foldLeft((0, 0.0)) {
      case (acc, coordinate) => (acc._1 + 1, acc._2 + coordinate._1)
    }
    latSum / count
  }
  val avgLatOfTheLargestSubRegionUDF = udf(avgLatOfTheLargestSubRegion)
  val avgLonOfTheLargestSubRegion: Array[Array[(Double, Double)]] => Double = { boundary =>
    val (count, latSum) = boundary.head.foldLeft((0, 0.0)) {
      case (acc, coordinate) => (acc._1 + 1, acc._2 + coordinate._2)
    }
    latSum / count
  }
  val avgLonOfTheLargestSubRegionUDF = udf(avgLonOfTheLargestSubRegion)

  val summarizedDS = cleanDS.withColumn( colName = "avg_lat", avgLatOfTheLargestSubRegionUDF(col( colName = "boundary")))
    .withColumn( colName = "avg_lon", avgLonOfTheLargestSubRegionUDF(col( colName = "boundary")))
    .drop( colName = "boundary")
    .orderBy(asc( columnName = "borough"), asc( columnName = "location_id"))
    .coalesce( numPartitions = 1)

  boroughs.foreach(b =>
    summarizedDS.filter( conditionExpr = f"borough = '$b'")
      .write
      .mode(SaveMode.Overwrite) // workaround for abnormal path-already-exists error
      .option("header", true)
      .csv( path = f"$path/$b")
  )
}
```

Figure 2. Code snippet for profiling NYC Taxi Zones [1]

After that, we computed the average latitude and longitude of the largest sub-region as *avg_lat* and *avg_lon* to be the geo coordinate of the zone, and finally saved *avg_lat* and *avg_lon* along with *borough* and *location_id* as the cleaned data. Note that representing each zone with

its largest sub-region keeps the average coordinate close to the center of the main region and makes the choice of the shortest path in the later part of the project more reasonable. Figure 2 shows the corresponding code snippet.

2.1.2 Data profiling

We profiled the cleaned NYC Taxi Zones [1] dataset with the following statistics:

1. Total coordinate count per location
2. Total sub-region count per location
3. Max sub-region coordinate count per location
4. All sub-region coordinate counts in a location

Figure 3 shows the corresponding profiling scripts.

```
def profileBoroughLocationByBoundary(cleanDS: Dataset[BoroughTaxiZone], path: String, boroughs: Seq[String]): Unit = {
  // define udfs
  val totalCoordinateCount: Array[Array[(Double, Double)]] => Int = _.map(_.length).sum
  val totalCoordinateCountUDF = udf(totalCoordinateCount)
  val maxSubRegionCoordinateCount: Array[Array[(Double, Double)]] => Int = _.map(_.length).max
  val maxSubRegionCoordinateCountUDF = udf(maxSubRegionCoordinateCount)
  val subRegionCoordinateCounts: Array[Array[(Double, Double)]] => String = _.map(_.length).mkString(" ")
  val subRegionCoordinateCountsUDF = udf(subRegionCoordinateCounts)

  val statsDS = cleanDS.withColumn( colName = "total_coordinate_count", totalCoordinateCountUDF(col( colName = "boundary")))
    .withColumn( colName = "total_sub_region_count", size(col( colName = "boundary")))
    .withColumn( colName = "max_sub_region_coordinate_count", maxSubRegionCoordinateCountUDF(col( colName = "boundary")))
    .withColumn( colName = "sub_region_coordinate_counts", subRegionCoordinateCountsUDF(col( colName = "boundary")))
    .drop( colName = "boundary")
    .orderBy(asc( columnName = "borough"), desc( columnName = "total_sub_region_count"))
    .coalesce( numPartitions = 1)

  boroughs.foreach(b =>
    statsDS.filter( conditionExpr = f"borough = '$b'")
      .write
      .mode(SaveMode.Overwrite) // workaround for abnormal path-already-exists error
      .option("header", true)
      .csv( path = f"$path/borough_location_boundary_stats/$b")
  )
}
```

Figure 3. Code snippet for profiling NYC Taxi Zones [1]

2.2 TLC Trip Record Data

The TLC Trip Record Data [2] is a parquet-formatted dataset that consists of 4 types of taxi ride differentiated by license, i.e. Yellow Taxi [50MB/Month], Green Taxi [2MB/Month], For-Hire Vehicle (FHV) [10MB/Month], and High Volume For-Hire Vehicle (HVFHV) [500MB/Month]. Figures 4, 5, 6, and 7 record the data schema of each according to the {Yellow [4], Green [5], FHV [6], HVFHV [7]} Trips Data Dictionary included in the dataset.

All taxi trips in between NYC Taxi Zones [1] of the 5+1 boroughs in NYC, i.e. Bronx, Brooklyn, Manhattan, Queens, Staten Island, and EWR (the airport area) are covered in this dataset, however, to simplify the analysis we focused on trips within the Manhattan borough, and utilized only the past 3 years of data spanning from Oct 2020 to September 2023. Note that the taxi zones defined in this dataset are roughly based on the NYC Department of City Planning's Neighborhood Tabulation Areas (NTAs), and are meant to approximate neighborhoods.

Field Name	Description
VendorID	A code indicating the TPEP provider that provided the record. **1= Creative Mobile Technologies, LLC; 2= VeriFone Inc.**
tpep_pickup_datetime	The date and time when the meter was engaged.
tpep_dropoff_datetime	The date and time when the meter was disengaged.
Passenger_count	The number of passengers in the vehicle. This is a driver-entered value.
Trip_distance	The elapsed trip distance in miles reported by the taximeter.
PULocationID	TLC Taxi Zone in which the taximeter was engaged
DOLocationID	TLC Taxi Zone in which the taximeter was disengaged
RateCodeID	The final rate code in effect at the end of the trip. **1= Standard rate** **2=JFK** **3=Newark** **4=Nassau or Westchester** **5=Negotiated fare** **6=Group ride**
Store_and_fwd_flag	This flag indicates whether the trip record was held in vehicle memory before sending to the vendor, aka "store and forward," because the vehicle did not have a connection to the server. **Y= store and forward trip** **N= not a store and forward trip**
Payment_type	A numeric code signifying how the passenger paid for the trip. **1= Credit card** **2= Cash** **3= No charge** **4= Dispute** **5= Unknown** **6= Voided trip**
Fare_amount	The time-and-distance fare calculated by the meter.
Extra	Miscellaneous extras and surcharges. Currently, this only includes the $0.50 and $1 rush hour and overnight charges.
MTA_tax	$0.50 MTA tax that is automatically triggered based on the metered rate in use.
Improvement_surcharge	$0.30 improvement surcharge assessed trips at the flag drop. The improvement surcharge began being levied in 2015.
Tip_amount	Tip amount – This field is automatically populated for credit card tips. Cash tips are not included.
Tolls_amount	Total amount of all tolls paid in trip.
Total_amount	The total amount charged to passengers. Does not include cash tips.
Congestion_Surcharge	Total amount collected in trip for NYS congestion surcharge.
Airport_fee	$1.25 for pick up only at LaGuardia and John F. Kennedy Airports

Figure 4: Yellow Trips Data Dictionary [4]

Field Name	Description
VendorID	A code indicating the LPEP provider that provided the record. **1= Creative Mobile Technologies, LLC; 2= VeriFone Inc.**
lpep_pickup_datetime	The date and time when the meter was engaged.
lpep_dropoff_datetime	The date and time when the meter was disengaged.
Passenger_count	The number of passengers in the vehicle. This is a driver-entered value.
Trip_distance	The elapsed trip distance in miles reported by the taximeter.
PULocationID	TLC Taxi Zone in which the taximeter was engaged
DOLocationID	TLC Taxi Zone in which the taximeter was disengaged
RateCodeID	The final rate code in effect at the end of the trip. **1= Standard rate** **2=JFK** **3=Newark** **4=Nassau or Westchester** **5=Negotiated fare** **6=Group ride**
Store_and_fwd_flag	This flag indicates whether the trip record was held in vehicle memory before sending to the vendor, aka "store and forward," because the vehicle did not have a connection to the server. **Y= store and forward trip** **N= not a store and forward trip**
Payment_type	A numeric code signifying how the passenger paid for the trip. **1= Credit card** **2= Cash** **3= No charge** **4= Dispute** **5= Unknown** **6= Voided trip**
Fare_amount	The time-and-distance fare calculated by the meter.
Extra	Miscellaneous extras and surcharges. Currently, this only includes the $0.50 and $1 rush hour and overnight charges.
MTA_tax	$0.50 MTA tax that is automatically triggered based on the metered rate in use.
Improvement_surcharge	$0.30 improvement surcharge assessed on hailed trips at the flag drop. The improvement surcharge began being levied in 2015.
Tip_amount	Tip amount – This field is automatically populated for credit card tips. Cash tips are not included.
Tolls_amount	Total amount of all tolls paid in trip.
Total_amount	The total amount charged to passengers. Does not include cash tips.

Figure 5: Green Trips Data Dictionary [5]

Field Name	Description
Dispatching_base_num	The TLC Base License Number of the base that dispatched the trip
Pickup_datetime	The date and time of the trip pick-up
DropOff_datetime	The date and time of the trip dropoff
PULocationID	TLC Taxi Zone in which the trip began
DOLocationID	TLC Taxi Zone in which the trip ended
SR_Flag	Indicates if the trip was a part of a shared ride chain offered by a High Volume FHV company (e.g. Uber Pool, Lyft Line). For shared trips, the value is **1**. For non-shared rides, this field is **null**. **NOTE:** For most High Volume FHV companies, only shared rides that were requested AND matched to another shared-ride request over the course of the journey are flagged. However, Lyft (base license numbers **B02510** + **B02844**) also flags rides for which a shared ride was requested but another passenger was not successfully matched to share the trip—therefore, trips records with SR_Flag=1 from those two bases could indicate EITHER a first trip in a shared trip chain OR a trip for which a shared ride was requested but never matched. Users should anticipate an overcount of successfully shared trips completed by Lyft.

Figure 6: FHV Trips Data Dictionary [6]

Field Name	Description
Hvfhs_license_num	The TLC license number of the HVFHS base or business As of September 2019, the HVFHS licensees are the following: • HV0002: Juno • HV0003: Uber • HV0004: Via • HV0005: Lyft
Dispatching_base_num	The TLC Base License Number of the base that dispatched the trip
Pickup_datetime	The date and time of the trip pick-up
DropOff_datetime	The date and time of the trip drop-off
PULocationID	TLC Taxi Zone in which the trip began
DOLocationID	TLC Taxi Zone in which the trip ended
originating_base_num	base number of the base that received the original trip request
request_datetime	date/time when passenger requested to be picked up
on_scene_datetime	date/time when driver arrived at the pick-up location (Accessible Vehicles-only)
trip_miles	total miles for passenger trip
trip_time	total time in seconds for passenger trip
base_passenger_fare	base passenger fare before tolls, tips, taxes, and fees
tolls	total amount of all tolls paid in trip
bcf	total amount collected in trip for Black Car Fund
sales_tax	total amount collected in trip for NYS sales tax
congestion_surcharge	total amount collected in trip for NYS congestion surcharge
airport_fee	$2.50 for both drop off and pick up at LaGuardia, Newark, and John F. Kennedy airports
tips	total amount of tips received from passenger
driver_pay	total driver pay (not including tolls or tips and net of commission, surcharges, or taxes)
shared_request_flag	Did the passenger agree to a shared/pooled ride, regardless of whether they were matched? (Y/N)
shared_match_flag	Did the passenger share the vehicle with another passenger who booked separately at any point during the trip? (Y/N)
access_a_ride_flag	Was the trip administered on behalf of the Metropolitan Transportation Authority (MTA)? (Y/N)
wav_request_flag	Did the passenger request a wheelchair-accessible vehicle (WAV)? (Y/N)
wav_match_flag	Did the trip occur in a wheelchair-accessible vehicle (WAV)? (Y/N)

Figure 7: High Volume FHV Trips Data Dictionary [7]

The main usage of this dataset in our analysis is as follows:

1. Identify common or prevalent pairs of pick-up and drop-off locations, where each pair of locations make good endpoints or intermediate stops for public transport routes.
2. Explore and discover meaningful insights into taxi trips.

2.2.1 Data cleansing

To achieve the first usage described above, we essentially need the two columns, *PULocationID* and *DOLocationID*. However, to allow possible explorations along the timeline when cleaning the 4 sub-datasets together as a whole, we attempted to keep the corresponding columns of pick-up and drop-off timestamps in each sub-dataset accordingly.

The cleaning steps we applied were quite straightforward: (1) select the 4 columns of pick-up and drop-off location IDs and timestamps in each sub-dataset, (2) drop rows containing null values in any of these 4 columns, (3) rename the 4 selected columns to a set of names in common, i.e. pu_datetime, do_datetime, pu_location_id, and do_location_id, and then (4) save the result as the cleaned data. Figure 8 shows the code snippet to clean the TLC Trip Record Data [2] as a whole, assuming each sub-dataset is stored under the same folder named `tlc`, followed by the name of the sub-dataset, and then by year and month. Note that there exists a schema change in *PULocationID* and *DOLocationID* columns from Feb 2023 in all sub-datasets, either from `Double` to `Long` or from `Long` to `Integer`, and thus to work around the type casting error encountered, we leveraged the multipath feature supported by the `spark.read.parquet API` to load data before the schema change as a Dataset and data after the schema change as another, and union the two at the end after working around the type casting error using a case class.

```scala
case class TaxiTrip(tlc_type: String, pu_datetime: Timestamp, do_datetime: Timestamp, pu_location_id: Long, do_location_id: Long)

val DEFAULT_TOTAL_PARTITION_COUNT = 100
val DEFAULT_SINGLE_PARTITION_COUNT = 10

val dataColsByType = Seq(
  ("fhv", Seq("pickup_datetime", "dropOff_datetime", "PULocationID", "DOLocationID")),
  ("fhvhv", Seq("pickup_datetime", "dropoff_datetime", "PULocationID", "DOLocationID")),
  ("green", Seq("lpep_pickup_datetime", "lpep_dropoff_datetime", "PULocationID", "DOLocationID")),
  ("yellow", Seq("tpep_pickup_datetime", "tpep_dropoff_datetime", "PULocationID", "DOLocationID"))
)
val newColNames = Seq("tlc_type", "pu_datetime", "do_datetime", "pu_location_id", "do_location_id")

// Wan-Yu Lin
def loadAndCleanRawData(spark: SparkSession, basePath: String): Dataset[TaxiTrip] = {
  import spark.implicits._

  def cleanRawData(rawDF: DataFrame, tlcType: String, oldColNames: Seq[String]): Dataset[TaxiTrip] = {
    rawDF.select(oldColNames.head, oldColNames.tail: _*)
      .na.drop() // drop rows with null value in any selected columns
      .withColumn(newColNames(0), lit(tlcType))
      .withColumnRenamed(oldColNames(0), newColNames(1))
      .withColumnRenamed(oldColNames(1), newColNames(2))
      .withColumnRenamed(oldColNames(2), newColNames(3))
      .withColumnRenamed(oldColNames(3), newColNames(4))
      .withColumn( colName = "pu_location_id", $"pu_location_id".cast(LongType))
      .withColumn( colName = "do_location_id", $"do_location_id".cast(LongType))
      .select(newColNames.head, newColNames.tail: _*)
      .as[TaxiTrip]
  }

  dataColsByType.map {
    case (tlcType, oldColNames) => {
      val by202301DS = cleanRawData(spark.read.parquet( paths = f"$basePath/$tlcType/202[0-2]/*/*", f"$basePath/$tlcType/2023/01/*"), tlcType, oldColNames)
      val from202302DS = cleanRawData(spark.read.parquet( path = f"$basePath/$tlcType/2023/0[2-9]/*"), tlcType, oldColNames)
      by202301DS.union(from202302DS)
    }
  }.reduce((x, y) => x union y)
}
```

Figure 8: Code snippet for cleaning TLC Trip Record Data [2] as a whole

After cleaning the dataset, we have 218722937 (roughly 2 billion) data points for our analysis, as shown in Figure 9.

	Filter-1	Filter-2	Filter-3	Filter-4	Total Trip Count
Raw Trips					598034968
	To or from Unknown (3.6%)				21539554
	To and from Known (96.4%)				576495414
		To or from outside Manhattan (60.35%)			347898299
		To and from inside Manhattan (39.65%)			228597115
			To or from isolated locations (0.32%)		724172
			To and from connected locations (99.68%)		227872943
				To and from the same location (4.02%)	9150006
				To and from different locations (95.98%)	218722937

Figure 9: Total trip count for each filtering stage when cleaning the TLC Trip Record Data [2] as a whole

Nevertheless, for detailed profiling of each sub-dataset owned by different members, we applied custom cleaning rules as well. We removed some invalid data from the Yellow [4] and Green [5] Trip Records Data by applying the checks mentioned below:

1. Remove rows where columns [PUlocationID, DOlocationID, total_amount, fare_amount, payment_type, RatecodeID] are NULL
2. Rename columns for readability
3. Remove trips where there are no passengers
4. Remove trips where the pick-up and drop-off are in the same area.
5. Remove trips that are dated outside October 2020 - September 2023.
6. Keep trips with payments in Cash or Credit Card or were taken free of cost. Remove the disputed, unknown, or voided trips.
7. Remove trips to airports EWR and JFK, and trips to Nassau or Westchester (based on the RateCode applied to the trip)
8. Remove trips where fare_amount was 0. This could be due to a faulty meter reading
9. Remove trips that were charged too low or too high (above 3 standard deviations of the mean total_amount)
10. Save the result as the cleaned data

2.2.2 Data profiling

As shown in Figure 10, we profiled the cleaned Yellow [4] and Green [5] dataset with the following:

1. The top-5 pick-up zones
2. The most frequent route

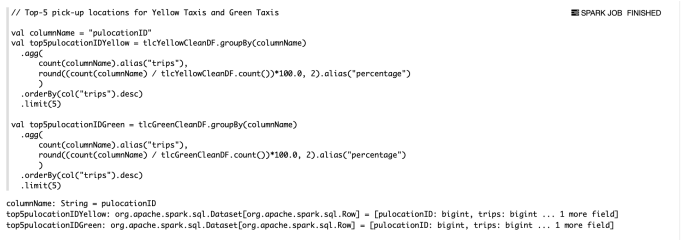

Figure 10. Code snippet for cleaning and profiling Yellow [4] & Green [5] Taxi record

As shown in Figure 11, we profiled the cleaned FHV [6] and HVFHV [7] dataset with the following:

1. The highest outgoing trips
2. The busiest routes among all the boroughs based on the pulocationId and dolocationId pair frequency counts

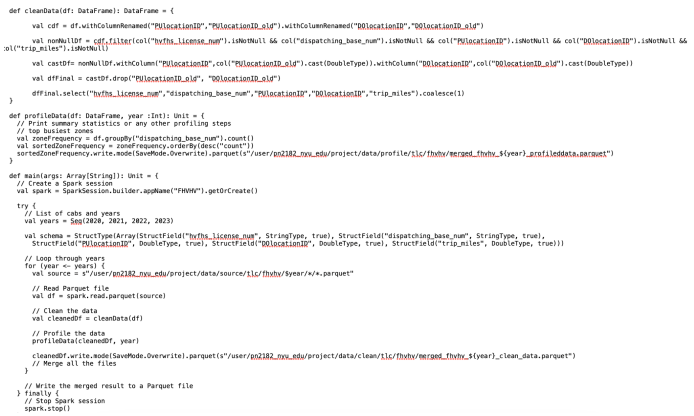

Figure 11. Code snippet for cleaning and profiling FHV [6] & HVHFV [7] Taxi record

2.3 Taxi Zone Lookup Table

The Taxi Zone Lookup Table [8] is a sub-dataset of the TLC Trip Record Data [2] dataset introduced in section 2.2 which provides the metadata, i.e. *LocationID*, *Borough*, *Zone*, and *service_zone*, of each taxi zone. Figure 12 shows its data snippet.

```
"LocationID","Borough","Zone","service_zone"
1,"EWR","Newark Airport","EWR"
2,"Queens","Jamaica Bay","Boro Zone"
3,"Bronx","Allerton/Pelham Gardens","Boro Zone"
4,"Manhattan","Alphabet City","Yellow Zone"
5,"Staten Island","Arden Heights","Boro Zone"
6,"Staten Island","Arrochar/Fort Wadsworth","Boro Zone"
7,"Queens","Astoria","Boro Zone"
8,"Queens","Astoria Park","Boro Zone"
9,"Queens","Auburndale","Boro Zone"
10,"Queens","Baisley Park","Boro Zone"
```

Figure 12: Data snippet of Taxi Zone Lookup Table [8]

In essence, this dataset acts as the lookup table for inferring taxi zone or location names.

2.3.1 Data cleansing

To fulfill the aim usage, we cleaned the data in two steps: (1) keep the former 3 columns, and rename each to a name in lowercase that aligns with other datasets, i.e. rename *LocationID* to *location_id*, *Borough* to *borough*, and *Zone* to *zone*, and (2) exclude rows where its borough value equals to “Unknown”. Figure 13 shows the code snippet for cleaning Note that this is a small table with no more than 300 rows, and hence examined by human eyes and searches in the Vim editor that there are no other null or unexpected values in the data.

```
def cleanRawData(spark: SparkSession, rawDF: DataFrame): Dataset[TaxiZoneLookupTable] = {
  import spark.implicits._

  rawDF.drop( colName = "service_zone")
    .select( col = "LocationID",  cols = "Zone", "Borough")
    .withColumnRenamed( existingName = "LocationID",  newName = "location_id")
    .withColumnRenamed( existingName = "Zone",  newName = "zone")
    .withColumnRenamed( existingName = "Borough",  newName = "borough")
    .as[TaxiZoneLookupTable]
    .filter(r => r.borough != "Unknown")
    .coalesce( numPartitions = 1)
}
```

Figure 13. Code snippet for cleaning Taxi Zone Lookup Table [8]

2.3.2 Data profiling

Table 2 shows the resulting location count per borough.

borough	location_count
Manhattan	69
Queens	69
Brooklyn	61
Bronx	43
Staten Island	20
EWR	1

Table 2. Location count per borough

Given this is a small dataset consisting only of metadata type of info, we simply profiled the dataset by the borough to compute the number of locations per borough, as shown in Figure 14.

```
def profileBoroughByLocationCount(cleanDS: Dataset[TaxiZoneLookupTable], path: String): Unit = {
  cleanDS.groupBy( col1 = "borough")
    .count()
    .coalesce( numPartitions = 1)
    .withColumnRenamed( existingName = "count",  newName = "location_count")
    .orderBy(desc( columnName = "location_count"))
    .write
    .mode(SaveMode.Overwrite) // workaround for abnormal path-already-exists error
    .option("header", true)
    .csv( path = f"$path/borough_location_count")
}
```

Figure 14. Code snippet for profiling Taxi Zone Lookup Table [8]

3. Methodology

When there exist frequent taxi trips going from location-X to location-Y, both location-X and location-Y could be considered good starting and ending stops, or 2 intermediate stops, of a public transport route.

With the TLC Trip Record Data [2], we can compute the frequency of each pick-up and drop-off pair from the taxi trips, and identify the top K pairs that appear most frequently. Then, we need to figure out an applicable route from the pick-up location to the drop-off location for each of the top K pairs.

As the explicit travel way of each taxi trip is not given in the dataset, we propose a simple approach to approximate a coarse-grained route using the neighboring relation between taxi zones (i.e. locations).

Inherently there exist multiple possible paths going from location-X to location-Y. For each possible path of length N-1, e.g. location-L_{1 -> ... -> i -> ... -> N}, where N >= 2 and 1 <= i <= N, L_1 corresponds to X, L_N corresponds to Y, and for all i each edge connecting L_i and L_(i+1) implies a neighboring relation and its weight represents the distance between L_i and L_(i+1). By adopting this setting, we can then apply Dijkstra's Algorithm to find the shortest (distance) path, i.e. an applicable route, going from location-X to location-Y.

Following are the assumptions we made:

1. Each taxi zone is a small enough area where no public transport route recommendation is needed, i.e. taxi trips starting and ending at the same location are excluded from the datasets.
2. Every taxi zone has at least 1 neighbor, i.e. isolated locations like islands are excluded from the datasets.
3. For all locations, there exists at least one viable way going directly from itself to each of its neighbors.
4. Each taxi trip going from location-X to location-Y took the shortest distance path between the two locations.
5. Given the geo-coordinates of the taxi zone boundaries provided in dataset-1, the coordinate of each location is represented by the average of the borderline of its main region or neighborhood (i.e. the sub-region with the most number of geo-coordinates).
6. The distance between two locations corresponds to the big circle distance [11].

After that, the implementation for this project could be broken down into 6 steps, where section 3.1 presents step 1: how the neighbor zone graph was constructed, 3.2 explains step 2: the computation of taxi trip frequency, 3.3 illustrates step 3: shortest path inference from the neighbor zone graph for each pick-up and drop-off location pair, 3.4 elaborates step 4: how the taxi trip coverage count of each shortest path could be calculated using all-pairs of location subsequence in the path, 3.5 details step 5: how the recommended routes for public transport were chosen from the result, and finally, 3.6 depicts step 6: how we evaluated the overall efficacy of our recommendations. Note that all the project steps, and data cleaning and profiling included, are implemented using Apache Spark in Scala, and leveraged NYU's Dataproc cluster to run.

3.1 Construct the neighbor zone graph

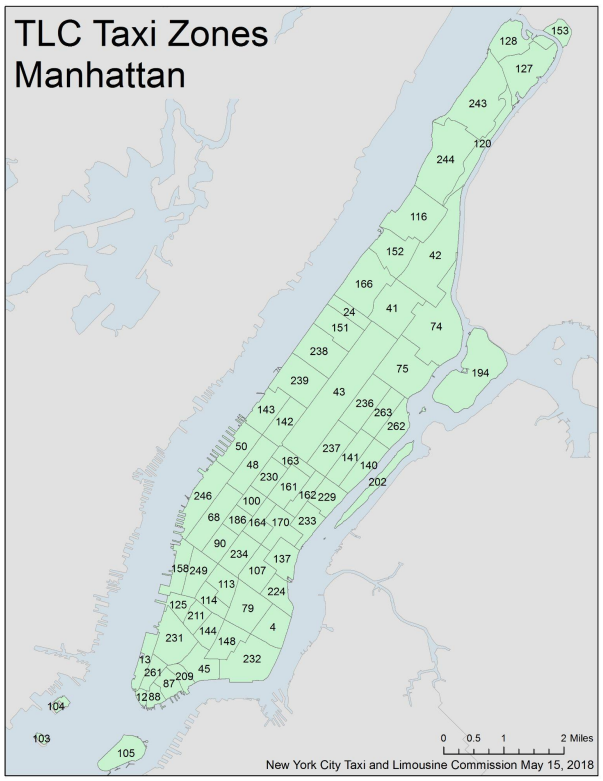

Figure 15. Taxi zone map of the Manhattan borough

In this step, originally we aimed to transform the NYC Taxi Zones [1] dataset into a connected neighbor zone graph with non-isolated zones in Manhattan, i.e. zones in Manhattan with location ID other than 103, 104, 105, 153, 194, and 202, and represent the graph by an adjacency list using distance as edge weight. Unfortunately, we found no common coordinates in the dataset between zones sharing the same boundary and thus were forced to create the neighbor zone mapping manually using Manhattan's Taxi Zone Map [9]. Figure 15 shows the taxi zone map of the Manhattan borough, and Figure 16 is a snippet of the manually created adjacency list.

```
"Manhattan" -> Map(
  4 -> List(224, 232, 79),
  24 -> List(166, 41, 43, 151),
  12 -> List(13, 261, 88),
  13 -> List(231, 261, 12),
  41 -> List(42, 74, 75, 43, 24, 166),
  45 -> List(144, 148, 232, 209, 231),
  42 -> List(120, 74, 41, 152, 116),
  43 -> List(41, 75, 236, 237, 163, 142, 239, 238, 151, 24),
  48 -> List(142, 163, 230, 100, 68, 246, 50),
```

Figure 16. A snippet of the manually created adjacency list

With the manually created neighbor zone mapping, and the representative geo coordinate of each zone we stored in the cleaned NYC Taxi Zones [1] data, we were able to compute the big circle distance [11] of each location with its neighbors, as shown in Figure 17.

```
val EARTH_RADIUS_M = 6378137d // Equatorial radius (WGS84) in meters

new *
def calcGeoDistanceInMeter(startLat: Double, startLon: Double, endLat: Double, endLon: Double): Int = {
  val latDiff = math.toRadians(endLat - startLat)
  val lonDiff = math.toRadians(endLon - startLon)
  val lat1 = math.toRadians(startLat)
  val lat2 = math.toRadians(endLat)

  val a = math.sin(latDiff / 2) * math.sin(latDiff / 2) +
    math.sin(lonDiff / 2) * math.sin(lonDiff / 2) * math.cos(lat1) * math.cos(lat2)
  val c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

  (EARTH_RADIUS_M * c).toInt
}

new *
case class TaxiZoneNeighborDistance(location_id: Long, neighbor_location_id: Long, distance: Int)

Wan-Yu Lin
def saveLocationNeighborsDistance(spark: SparkSession, sourcePath: String, outputPath: String): Unit = {
  import spark.implicits._
  val sc = spark.sparkContext

  val boroughs = connected.keys.toSeq.filter(_ == "Manhattan") // Manhattan only for the project
  boroughs.foreach(b => {
    val zones = TaxiZones.loadCleanDataByBorough(spark, sourcePath, borough = b)  : Dataset[TaxiZones.TaxiZoneGeo]
      .rdd                                                                        : RDD[TaxiZones.TaxiZoneGeo]
      .map(z => z.location_id -> (z.avg_lat, z.avg_lon))                          : RDD[(Long, (Double, Double))]
      .collect                                                                    : Array[(Long, (Double, Double))]
      .toMap                                                                      : Map[Long, (Double, Double)]
    val zonesBroadcast = sc.broadcast(zones)

    val zoneNeighborDisDS = sc.parallelize(connected(b).toSeq.flatMap {
      case (startLoc, neighbors) => {
        val (startLat, startLon) = zonesBroadcast.value(startLoc)
        neighbors.map(endLoc => {
          val (endLat, endLon) = zonesBroadcast.value(endLoc)
          TaxiZoneNeighborDistance(startLoc, endLoc, calcGeoDistanceInMeter(startLat, startLon, endLat, endLon))
        })
      }
    }).toDS

    zoneNeighborDisDS.coalesce( numPartitions = 1)
      .write
      .mode(SaveMode.Overwrite) // workaround for abnormal path-already-exists error
      .option("header", true)
      .csv( path = f"$outputPath/location_neighbors_distance/$b")
  })
}
```

Figure 17. Code snippet for computing distance with neighbors for zones in Manhattan

After that, we referred to [10] to convert the adjacency list with neighbor distances as edge weights into a WeightedGraph as Figure 18 shows, and had it stored in a broadcast variable to improve computation efficiency in the succeeding parts of the project.

```
val borough = "Manhattan" // target borough

// step-1
Wan-Yu Lin
def buildZoneNeighboringGraph(spark: SparkSession, neighborsDistanceInputPath: String): WeightedGraph[Long] = {
  import spark.implicits._

  val zoneNeighborDisDS = loadLocationNeighborsDistanceByBorough(spark, neighborsDistanceInputPath, borough = borough)
  new WeightedGraph[Long](zoneNeighborDisDS.map(r =>
    (r.location_id, WeightedEdge(r.neighbor_location_id, r.distance))).rdd
    .groupByKey()
    .mapValues(_.toList)
    .collect
    .toMap)
}
```

Figure 18. Code snippet for converting the adjacency list with neighbor distances as edge weights into a WeightedGraph

3.2 Compute taxi trip frequency

In this step, we aimed to identify the prevalent pick-up and drop-off location pairs within the Manhattan borough, using the cleaned TLC Trip Record Data [2], and then compute the occurrence or frequency of each with the `groupBy` operation and `count` aggregation. Note that trips traveling to and from the same location are excluded due to the lack of need for taking public transport within a neighborhood; trips that start or end in an isolated zone were also kept out of the calculation because we're unable to infer a path for zones without neighbors. Figure 19 shows our filtering and computation scripts.

```
// step-2
Wan-Yu Lin *
def getTaxiTripFreqDS(spark: SparkSession, rawTrips: Dataset[TaxiTrip], conLocList: List[Long],
                      isoLocList: List[Long]): Dataset[TaxiTripFreq] = {
  import spark.implicits._

  rawTrips.select( col = "pu_location_id",  cols = "do_location_id")
    .groupBy( col1 = "pu_location_id",  cols = "do_location_id")
    .agg(count( columnName = "*") as "frequency")
    .filter( condition = $"pu_location_id" =!= $"do_location_id")
    .filter($"pu_location_id".isin(conLocList: _*) && $"do_location_id".isin(conLocList: _*))
    .filter(!$"pu_location_id".isin(isoLocList: _*) && !$"do_location_id".isin(isoLocList: _*))
    .as[TaxiTripFreq]
}
```

Figure 19: Code snippet for computing taxi trip frequency

3.3 Infer the shortest path

In this step, by applying Dijkstra's Algorithm (as implemented in [10] shown in Figure 20) on the WeightedGraph built in section 3.1, we aimed to infer a logical path for each pick-up and drop-off pair in the frequency RDD computed in section 3.2.

```
  def findShortestPaths[N](source: N, graph: WeightedGraph[N]): ShortStep[N] = {

    // set distances to all nodes to infinity, distance to source node is zero
    val sDistances: Map[N, Int] = graph.nodes.map(_ -> Int.MaxValue).toMap + (source -> 0)

    def shortestPath(step: ShortStep[N]): ShortStep[N] = {
      // get nodes with minimal value
      step.extractMin().map {
        case (node, currentDistance) =>
          val newDist = graph.neighboursWithWeights(node).collect {
            case WeightedEdge(neighbour, neighbourDistance) if step.distances.get(neighbour).exists(_ > currentDistance + neig
              neighbour -> (currentDistance + neighbourDistance)
          }

          val newParents = newDist.map { case (neighbour, _) => neighbour -> node }

          shortestPath(ShortStep(
            step.parents ++ newParents,
            step.unprocessed - node, // current node is processed
            step.distances ++ newDist))
      }.getOrElse(step)
    }

    shortestPath(ShortStep(Map(), graph.nodes.toSet, sDistances))
  }

  private def findPathRec[N](node: N, parents: Map[N, N]): List[N] =
    parents.get(node).map(parent => node +: findPathRec(parent, parents)).getOrElse(List(node))

  /**
   * Function return the way from source node to given destination node.
   *
   * @param destination destination node
   * @param parents     parent nodes
   * @tparam N
   * @return path from source to destination
   */
  def findPath[N](destination: N, parents: Map[N, N]): List[N] = {
    findPathRec(destination, parents).reverse
  }
```

Figure 20. The code snippet for Dijkstra's Algorithm

Code snippet in Figure 21 demonstrates the scripts we used to transform each pick-up and drop-off pair in the frequency RDD `tripDreqDS` into the shortest path string (separated by ",") inferred from the neighbor zone graph.

```
// step-3
Wan-Yu Lin +1
def getTripPathFreqDS(spark: SparkSession, tripFreqDS: Dataset[TaxiTripFreq],
                      graphBroadcast: Broadcast[WeightedGraph[Long]]): Dataset[TripPathFreq] = {
  import spark.implicits._

  tripFreqDS.map(tf => {
    val result = Dijkstra.findShortestPaths(tf.pu_location_id, graphBroadcast.value)
    val path = Dijkstra.findPath(tf.do_location_id, result.parents)
    TripPathFreq(tf.frequency, path.mkString(","))
  })
}
```

Figure 21: Code snippet for transforming each pick-up and drop-off pair in the frequency RDD into the shortest path inferred

After that, we then sorted the taxi trip paths descendingly by frequency, and exported it to a TSV file for later usage.

Figure 22 is a data snippet of the resulting taxi trip path frequency.

frequency	trip_path
954337	237,236
853949	236,237
483353	237,162,161
428745	161,162,237
395692	239,142
394864	141,236
386968	236,237,162,161
382741	41,42
380419	75,74

Figure 22: Data snippet of the resulting taxi trip path frequency

3.4 Compute the coverage count for each trip path

The primary objective of this step is to determine the coverage count for each potential public transport route. The coverage count mentioned here is a measurement of how many taxi trips could be covered by a particular public transport route, thereby indicating the route's potential utility and effectiveness.

For example, as shown in Figure 23, if the input taxi trip path frequency looks like [("1,2", 2), ("2,3", 2), ("3,4", 2), ("1,2,3", 2), ("2,3,4", 2), ("1,2,3,4", 2), ("3,4", 1), ("4,5", 1)], the coverage count of path "1,2,3,4" should be 12, because the trips of paths "1,2", "2,3", "3,4", "1,2,3", "2,3,4", and "1,2,3,4" can all be covered.

```
Input: RDD[("1,2,3,4", 12), ("1,2,3", 6), ("2,3,4", 6), ("3,4,5", 4), ("1,2", 2), ("2,3", 2),

Output:
coverage_count\ttrip_path
12\t1,2,3,4
6\t1,2,3
6\t2,3,4
4\t3,4,5
2\t1,2
2\t2,3
2\t3,4
1\t4,5
```

Figure 23: Coverage count for each pair of locations

A higher coverage count suggests that a route could accommodate a large number of taxi trips, making it a prime candidate for public transport. Thus, through this step, we analyzed the taxi trip data to extract meaningful insights about potential public transport routes.

3.5 Recommend candidate routes

The primary goal here is to translate our analytical findings into practical, actionable recommendations for public transport routes in the Manhattan borough. This step revolves around selecting the most effective routes based on the coverage count calculated in step 4.

We begin by identifying the top-k trip paths that have the highest coverage counts and a minimum length of m for each path. Here m indicates the number of stops along the path. This ensures that the recommended routes are

substantial enough to make a profitable and efficient transportation network. Once the top paths are identified, the next crucial task is to assign human-readable real-world location names to these paths from strings of location IDs (e.g., "1,2,3,4"). This conversion is done by joining the selected paths with dataset-3, which contains detailed information about the names of each location, allowing us to represent each route in terms of actual place names or landmarks.

```
# select the top k paths of at least length m according to coverage count
Input: k = 5, m = 3, RDD[("1,2,3,4", 12), ("1,2,3", 6), ("2,3,4", 6), ("3,4,5", 4), ("1,2", 2), ("2,
Output: [("1,2,3,4", 12), ("1,2,3", 6), ("2,3,4", 6)]

# join candidate paths with dataset-3 to provide human-readable routes
Input: [("1,2,3,4", 12), ("1,2,3", 6), ("2,3,4", 6)]
Output:
RDD[
    ("Penn Station/Madison Sq West,Flatiron,West Village,Hudson Sq", 12),
    ("Penn Station/Madison Sq West,Flatiron,West Village", 6),
    ("Flatiron,West Village,Hudson Sq", 6),
    ("West Village,Hudson Sq,SoHo", 4)
]
```

Figure 24: Recommended human-readable routes

Figure 24 shows sample input and output of the final step that involves exporting the human-readable routes as a TSV (Tab-Separated Values) file. This file is sorted in descending order by coverage count, followed by the route length.

3.6 Compute the coverage of taxi trips by the top-k recommended routes and each route

We also compute two metrics to assess the effectiveness of our analyses. Firstly, we compute the percentage of taxi trips covered by each recommended route. This metric is useful for assessing the effectiveness of the recommended routes. Table 3 shows the top 3 routes with actual location/landmark names along with the percentage of the 2 billion taxi trips that each route can cover. Next, we compute the same metric for the top-k routes that are the output of step-3.5. This metric gives an insight into how many possible recommendations should be considered for actual route planning to achieve a target coverage percentage of the total taxi trips used for the analysis. Table 4 shows the results for top-5, 10, 20, and 50 trips that are at least 10 stops long.

4. Results

Our project presents an approach to find the top recommended public transport routes in Manhattan, derived from a comprehensive analysis of 2 billion taxi trips. Our focus was on identifying routes that are at least 10 stops long and evaluating their coverage percentage concerning the total number of trips. The insights gained from this analysis are pivotal for enhancing urban mobility and guiding future public transportation planning. From Table 3, we see that the top 3 routes can individually cover roughly 4% of the total trips in consideration. Each of these is a good candidate for potential routes. We also see that the first and the third routes are opposite of each other, indicating that there is a need for transport in both directions.

The coverage analysis of the top-K routes underscores the importance of a well-planned route network that can significantly enhance the coverage and efficiency of the public transportation system. From Table 4, we see that the top-20 routes cover roughly 9.85% of the total trips. This indicates that the unique location IDs in the top-20 routes are all areas with high taxi traffic and there is a need to have a wider range of routes in those areas.

Route	**% of all trips covered**
East Harlem North, East Harlem South, Upper East Side North, Upper East Side South, Midtown East, Midtown Center, Midtown South, Union Square, Greenwich Village North, Greenwich Village South, SoHo, TriBeCa, Battery Park City.	4.25
Yorkville West, Upper East Side North, Upper East Side South, Midtown East, Midtown Center, Midtown South, Union Square, Greenwich Village North, Greenwich Village South, SoHo, TriBeCa, Battery Park City.	4.22
Battery Park City, TriBeCa, SoHo, Greenwich Village South, Greenwich Village North, Union Square, Midtown South, Midtown Center, Midtown East, Upper East Side South, Upper East Side North, East Harlem South, East Harlem North.	4.18

Table 3. Top-3 recommended routes with their coverage percentage

Top-K Routes	**% of all trips covered**
Top-5	9.122
Top-10	9.845
Top-20	9.851
Top-50	17.706

Table 4. Cleaned data schema of TLC Trip Record Data [2]

As we can observe in Figure 25 - Current subway routes in Manhattan often require multiple transitions and changes when traveling from the Upper East Side to the Lower Left Side (such as the neighborhoods of TriBeCa and Battery Park City). This can result in longer travel

times and potentially less convenient commutes for residents and visitors.

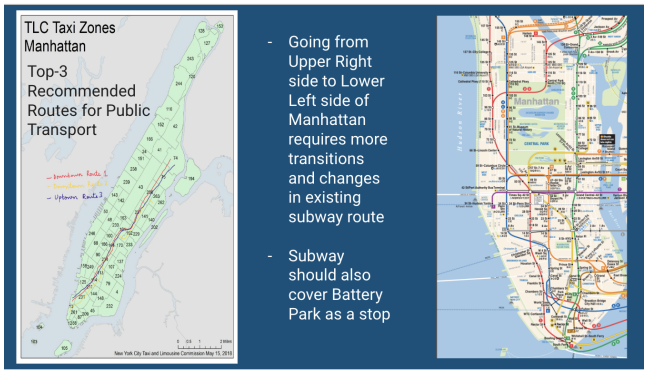

Figure 25: Recommended human readable routes

The analysis conducted in our project revealed that the top three most frequented taxi routes in Manhattan span from the Upper East Side to the Lower West Side. Based on these findings, we recommend the implementation of new subway routes mirroring these high-demand paths.

5. Discussions

We faced several challenges and findings during this work. We discuss them briefly here:

1. While constructing the neighbor graph using the geometric coordinates of each zone boundary, there were no common coordinates in the dataset even between the neighbor locations that share the same borderline. We decided to manually build up the neighbor mapping according to the map.
2. Since we are using Dijkstra's algorithm to approximate a coarse-grained route, we exclude the isolated locations without neighbors from our computations. We also do not include trips that begin and end at the same location ID, as they are too short and do not make sense for our goal. This would ensure the correctness of our results as well, as we are excluding corner cases for the sake of simplicity.
3. We also experimented with the Union-Find algorithm[19] to cluster overlapping trips. Since the recommended trips can go over 12 stops long, it became difficult to verify the results manually. Hence we proceed by storing overlapping trips in a Map.

To ensure accurate results we have written additional scripts to verify the correctness of the zone neighboring graph and performed a manual check from the map as well. We have verified the aggregation results with different queries and ensured that identical results are being produced.

6. Conclusions

In conclusion, this research project has successfully demonstrated the potential of utilizing big data, specifically the TLC Trip Record Data [2], in conjunction with the NYC taxi zones and Taxi Zone Lookup datasets, to enhance public transportation planning in the Manhattan borough. This analysis is also useful to plan for public transit routes in cities/zones where it doesn't exist yet.

Our approach involved several phases: initial data cleansing, the construction of a neighbor zone graph, computation of taxi trip frequencies, the transformation of these trips into the shortest paths using Dijkstra's Algorithm, and finally, the determination of coverage counts for each path.

The effectiveness of our methodology was particularly evident in identifying and recommending public transport routes. By prioritizing paths with the highest coverage counts, we ensure that the suggested routes are not only the most frequented but also the most beneficial for the maximum number of commuters (in terms of shortest distance). The transformation of these routes into a human-readable format was a critical step toward making our findings accessible and practical for urban transportation planning.

7. References

[1] NYC Taxi Zones: https://catalog.data.gov/dataset/nyc-taxi-zones
[2] TLC Trip Record Data: https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[3] Trip Record User Guide: https://www.nyc.gov/assets/tlc/downloads/pdf/trip_record_user_guide.pdf
[4] Yellow Trips Data Dictionary: https://www.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf
[5] Green Trips Data Dictionary: https://www.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_green.pdf
[6] FHV Trips Data Dictionary: https://www.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_fhv.pdf
[7] High Volume FHV Trips Data Dictionary: https://www.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_hvfhs.pdf
[8] Taxi Zone Lookup Table: https://d37ci6vzurychx.cloudfront.net/misc/taxi+_zone_lookup.csv

[9] Taxi Zone Map - Manhattan: https://www.nyc.gov/assets/tlc/images/content/pages/about/taxi_zone_map_manhattan.jpg
[10] Graph algorithms in Scala: https://github.com/Arminea/scala-graphs
[11] Scala: Calculating the Distance Between Two Locations: https://dzone.com/articles/scala-calculating-distance-between-two-locations
[12] GeoTrellis: Haversine.scala: https://github.com/locationtech/geotrellis/blob/master/util/src/main/scala/geotrellis/util/Haversine.scala
[13] RDD vs Dataframe vs Dataset: https://www.databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html
[14] Spark SQL: API Doc (Scala): https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/index.html
[15] Reading Command-Line Arguments in Scala: https://www.baeldung.com/scala/read-command-line-arguments
[16] Spark Scala: Operations on One Column: https://haya-toumy.gitbook.io/spark-notes/spark-scala/spark-scala/operations-on-one-column
[17] Spark SQL UDFs: https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-udfs.html
[18] Sum a list of tuples by foldLeft: https://stackoverflow.com/questions/30154736/how-to-sum-a-list-of-tuples
[19] Union-Find implementation in Scala: https://github.com/scala-ide/scala-refactoring/blob/master/src/main/scala/scala/tools/refactoring/util/UnionFind.scala

8. Division Of Work

1. **Data Ingestion**
   - Wan-Yu Lin, wl2484
2. **Data Cleaning & Profiling**
   - NYC Taxi Zones [1]
     - Wan-Yu Lin, wl2484
   - TLC Trip Record Data - Yellow Trips Data [4]
     - Charvi Gupta, cg4177
   - TLC Trip Record Data - Green Trips Data [5]
     - Charvi Gupta, cg4177
   - TLC Trip Record Data - FHV Trips Data [6]
     - Priyanka Narain, pn2182
   - TLC Trip Record Data - HVFHV Trips Data [7]
     - Priyanka Narain, pn2182
   - TLC Trip Record Data [2] - As a whole
     - Wan-Yu Lin, wl2484
   - Taxi Zone Lookup Table [8]
     - Wan-Yu Lin, wl2484
3. **Methodology**
   - Step 1: Wan-Yu Lin, wl2484
   - Step 2:
     - Priyanka Narain, pn2182, Wan-Yu Lin, wl2484
   - Step 3: Priyanka Narain, pn2182
   - Step 4: Charvi Gupta, cg4177
   - Step 5: Charvi Gupta, cg4177
   - Step 6: Charvi Gupta, cg4177
4. **Presentation**
   - Slide:
     - Wan-Yu Lin, wl2484
     - Charvi Gupta, cg4177
     - Priyanka Narain, pn2182
   - Presentation:
     - Wan-Yu Lin, wl2484
     - Charvi Gupta, cg4177
5. **Report**
   - Wan-Yu Lin, wl2484
   - Priyanka Narain, pn2182
   - Charvi Gupta, cg4177