Homework Writeup

## /\*Startup Reminders\*/

/\*

• Start Spark shell like this:

spark-shell --master local[\*] --driver-memory 4g --jars /proj/cse398-

498/course/aas/ch08-geotime/target/ch08-geotime-2.0.0-jar-with-dependencies.jar

• You’ll also need GEOJSON file for NYC boroughs:

/proj/cse398-498/course/AAS\_CH8/nyc-boroughs.geojson

\*/

# //--Geospatial and Temporal Data Analysis on New York City Taxi Trip Data--//

/\*

New york is widely known for its yellow taxis. The only consensus people say is

during 4 to 5, take the subway instead since thats the shift change. On March 4th

2014, New York City Taxi and Limousine Commission shared an infographic on its

Twitter account that displayed the number of taxis on the road and the numbers

occupied at a given time. This information supported the earlier claim since during

the shift change, taxi drivers have to drive back to there center to return the

taxi for the next shift. Chris Wong noticed this release and followed up asking

for more data if they had. Taxi commision agreed if he turned in a Freedom of

Information Law request. He then released the two 500 gb hardrives publicly online.

One of the important statistics within the taxi field would be utilization, or

amount of time/duration that a taxi is being used. During rush hour, drivers may

earn fares nonstop, whereas at 2am, it may take some time. To sort data like this,

there exist temporal data, dates/time, and geospatial information, like points

of longitutde and latitude, that will have to be analyzed. Sparks capability rose

from Spark 1 to 2 in terms of capability to manipulate temporal data due to the

release of Java 8's java.time package and the use of UDFS from Apache Hive

proj in SparkSQL. Geospatial data is still hard to manipulate and will require

3rd party and our own custom UDFs

\*/

# //--Getting the Data--//

//For this chapter, we used the fare data from January 2013, 2.5 gb

/\*

$ mkdir taxidata

$ cd taxidata

$ curl -O https://nyctaxitrips.blob.core.windows.net/data/trip\_data\_1.csv.zip

$ unzip trip\_data\_1.csv.zip

$ head -n 10 trip\_data\_1.csv

\*/

/\*

Each row of the file after the header represents a single taxi ride in CSV format.

For each ride, we have some attributes of the cab (a hashed version of the medallion

number) as well as the driver (a hashed version of the hack license, which is what

licenses to drive taxis are called), some temporal information about when the trip

started and ended, and the longitude/latitude coordinates for where the passenger(s)

were picked up and dropped off

The first row is our header, which shows our attributes:

medallion

hack\_license

vendor\_id

rate\_code

store\_and\_fwd\_flag

pickup\_datetime

dropoff\_datetime

passenger\_count

trip\_time\_in\_secs

trip\_distance

pickup\_longitude

pickup\_latitude dropoff\_longitude

dropoff\_latitude

\*/

# 

# //--Working with Third-Party Libraries in Spark--//

/\*

One of the major benefits of Scala being built on Java is that Java is such a

widely known and used language, that most code you may need is already available

or even open-source. However, the quality of each library differs drastically.

The library we choose we want it to keep working with the Serializable inter-

face, or be serialized using Kyro. We also want to make sure the libraries we use

have the least amount of dependencies (due to size and differ Java variations).

Lastly, we don't want our APIs that extensivly use Java-oriented design patterns

like abstract factories and visitors. Some even have Scala wrappers to increase

scalability and reduce boilerplate code.

\*/

# //--Geospatial Data with the Esri Geometry API and Spray--//

/\*

When working with geospatial data, there is 2 major kinds: vector and raster.

Within the dataset, we have longitude/latitude and vector data stored in GeoJSON

format, which represents boundaries of the different boroughs within New York.

Sadly, there is only a library to parse these GeoJSON into Java objects, but no

other library to do spatial analyzation. Due to this, we will be working with

ESRI Geometry API, but can only parse a subjset of GeoJSON (need to clean data).

However, we can make a new Scala function for parsing all of GeoJSON.

\*/

## //-Exploring the Esri Geometry API-//

/\*

Core data type is Geometry, holds shape and geolocation. Esri library can compute

are of geometry, whether two overlap, compute the geometry of overlap. In our case,

the geometry objects will represent dropoffs and boroughs. We want to know if a

point is in one of the boroughs. All these methods are within GeometryEngine,

inclduingcontains operation. The contains method takes three arguments:

two Geometry objects, one instance of the SpatialReference class, the coordinate system.

The SpatialReference will be using is WKID 4326, coord system for GPS.

Following the naming convention, we are going to add some helper methods.

\*/

import com.esri.core.geometry.{GeometryEngine, SpatialReference, Geometry}

import scala.language.implicitConversions

class RichGeometry(val geometry: Geometry,

val spatialReference: SpatialReference =

SpatialReference.create(4326)) {

def area2D() = geometry.calculateArea2D()

def contains(other: Geometry): Boolean = {

GeometryEngine.contains(geometry, other, spatialReference)

}

def distance(other: Geometry): Double = {

GeometryEngine.distance(geometry, other, spatialReference)

}

}

//make it so that it implicitly converts all Geometry to RichGeometry

object RichGeometry {

implicit def wrapRichGeo(g: Geometry) = {

new RichGeometry(g)

}

}

//import this implicit function

import RichGeometry.\_

## //-Intro to GeoJSON-//

/\*

The boundaries for boroughs are in GeoJSON format. The core object in GeoJSON is called a

feature, which is made up of a geometry instance and a set of key-value pairs called

properties.A set of features is called a FeatureCollection. Download the data:

$ curl -O https://nycdatastables.s3.amazonaws.com/2013-08-19T18:15:35.172Z/

nyc-borough-boundaries-polygon.geojson

$ mv nyc-borough-boundaries-polygon.geojson nyc-boroughs.geojson

Esri will parse the Geometry objects, but no the id nor properites. Use Spray to convert

any Scala object to a corresponding JsValue by calling an implicit toJson method.

Convert string to parseJson then converting it to scala.

We need to create a class to hold GeoJSON features. Each JSON object references its own

attribute.

\*/

//create a function to look up values within properites

import spray.json.\_

case class Feature(

val id: Option[JsValue],

val properties: Map[String, JsValue],

val geometry: RichGeometry) {

def apply(property: String) = properties(property)

def get(property: String) = properties.get(property)

}

//We need to also make a corresponding class for GeoJSON FeatureCollection.

case class FeatureCollection(features: Array[Feature])

extends IndexedSeq[Feature] {

def apply(index: Int) = features(index)

def length = features.length

}

/\*

After creating our case classes, we need a way to help Spray convert RichGeometry, Feature,

and FeatureCollection along with a JsValue.

\*/

implicit object FeatureJsonFormat extends

RootJsonFormat[Feature] {

def write(f: Feature) = {

val buf = scala.collection.mutable.ArrayBuffer(

"type" -> JsString("Feature"),

"properties" -> JsObject(f.properties),

"geometry" -> f.geometry.toJson)

f.id.foreach(v => { buf += "id" -> v})

JsObject(buf.toMap)

}

def read(value: JsValue) = {

val jso = value.asJsObject

val id = jso.fields.get("id")

val properties = jso.fields("properties").asJsObject.fields

val geometry = jso.fields("geometry").convertTo[RichGeometry]

Feature(id, properties, geometry)

}

}

//The rest of the code can be seen within the jar file we are using.

## /\*All imports\*/

import spark.implicits.\_

import java.text.SimpleDateFormat

import java.util.Locale

import java.util.concurrent.TimeUnit

import org.apache.spark.sql.{DataFrame, Row, SparkSession}

import org.apache.spark.sql.functions.\_

import com.esri.core.geometry.Point

import spray.json.\_

import com.cloudera.datascience.geotime.GeoJsonProtocol.\_

# //--Preparing the New York City Taxi Trip Data--//

val taxiRaw = spark.read.option("header", "true").csv("/proj/cse398-498/course/AAS\_CH8/taxidata")

taxiRaw.show()

/\*

When using the automatic converter, it takes 2 folds. Which is incredibly

ineffecient and will be wasted even more when attributes are dropped for

analysis. DO custom conversion ourselves. If we want to take advantage

of the speed and processing power of the Dataset class, we must stick to

small data types (int, string, double, long).

\*/

case class Trip(

license: String,

pickupTime: Long,

dropoffTime: Long,

pickupX: Double,

pickupY: Double,

dropoffX: Double,

dropoffY: Double

)

//As of now, time is long due to Unix epoch, and x&y will become a Point

//Create a method to parse information if null

class RichRow(row: org.apache.spark.sql.Row) {

def getAs[T](field: String): Option[T] = { //returns an Option[T] to

if (row.isNullAt(row.fieldIndex(field))) { //explicitly handle nulls

None

} else {

Some(row.getAs[T](field))

}

}

}

//Parse string to get time in miliseconds

def parseTaxiTime(rr: RichRow, timeField: String): Long = {

val formatter = new SimpleDateFormat(

"yyyy-MM-dd HH:mm:ss", Locale.ENGLISH)

val optDt = rr.getAs[String](timeField)

optDt.map(dt => formatter.parse(dt).getTime).getOrElse(0L)

}

//Convert pickup/dropoff locations from string to Doubles using implicit method

def parseTaxiLoc(rr: RichRow, locField: String): Double = {

rr.getAs[String](locField).map(\_.toDouble).getOrElse(0.0) //return 0 if null

}

//Combining all 3 methods into one:

def parse(row: org.apache.spark.sql.Row): Trip = {

val rr = new RichRow(row)

Trip(

license = rr.getAs[String]("hack\_license").orNull,

pickupTime = parseTaxiTime(rr, "pickup\_datetime"),

dropoffTime = parseTaxiTime(rr, "dropoff\_datetime"),

pickupX = parseTaxiLoc(rr, "pickup\_longitude"),

pickupY = parseTaxiLoc(rr, "pickup\_latitude"),

dropoffX = parseTaxiLoc(rr, "dropoff\_longitude"),

dropoffY = parseTaxiLoc(rr, "dropoff\_latitude")

)

}

## //-Handling Invalid Records at Scale-//

/\*

Many failures within the pipeline is due to data that doesn't confirm to

standard. Typically, this is a game of whack-a-mole; where the developer

is fixing one error and getting another one, and so on. One way is to use

try-catch blocks, and skip the handful of mistakes. In Spark, we can

adapt our parsing to even work with invalid entries.

There is 2 possible outcomes, success parsing or failure. If it is failure,

we want to catch the entrie and exception. Whenever an operation has 2

outcomes, we can use

Either[L (success), R (failure, a tuple of entrie and excpetion)]

\*/

def safe[S, T](f: S => T): S => Either[T, (S, Exception)] = {

new Function[S, Either[T, (S, Exception)]] with Serializable {

def apply(s: S): Either[T, (S, Exception)] = {

try {

Left(f(s))

} catch {

case e: Exception => Right((s, e))

}

}

}

}

//Apply safe wrapper to parser to prevent parsing issues

val safeParse = safe(parse)

val taxiParsed = taxiRaw.rdd.map(safeParse) //no direct due to Either not in Dataset API

taxiParsed.map(\_.isLeft). //print number correctly parsed

countByValue().

foreach(println)

//(true,14776615)

//Since none failed, convert parsed to Dataset

val taxiGood = taxiParsed.map(\_.left.get).toDS

taxiGood.cache()

/\*

Just because everything parsed properly doesnt mean there are discrpencies within the

data. One of the top of the head, is if dropoff time is earlier than pickup.

\*/

//Create a method to convert miliseconds to hours

val hours = (pickup: Long, dropoff: Long) => {

TimeUnit.HOURS.convert(dropoff - pickup, TimeUnit.MILLISECONDS)

}

//Wrap the hours in a UDF (UserDefinedFunction) to apply to both time columns

import org.apache.spark.sql.functions.udf

val hoursUDF = udf(hours)

taxiGood.

groupBy(hoursUDF($"pickupTime", $"dropoffTime").as("h")).

count().

sort("h").

show()

//returns a histogram of time and count (perfect use of DataSetAPI/SparkSQL)

/\*

+---+--------+

| h| count|

+---+--------+

| -8| 1| //one instance of negative 8 hours

| 0|14752326|

| 1| 22934|

| 2| 843|

| 3| 197|

| 4| 86|

| 5| 55|

| 6| 42|

| 7| 33|

| 8| 17|

| 9| 9|

| 10| 11|

| 11| 13|

| 12| 7|

| 13| 5|

| 14| 5|

| 15| 3|

| 16| 5|

| 17| 4|

| 19| 3|

+---+--------+

\*/

//Analyze odd instance

taxiGood.

where(hoursUDF($"pickupTime", $"dropoffTime") < 0).

collect().

foreach(println)

//Trip(4669D6DB6D5B6739B9194E999D907924,1359155305000,1359125716000,-73.952911,40.748318,-73.952835,40.748287)

//Analyzing histogram shows that most rides are no longer than 3 hours

spark.udf.register("hours", hours) //registering our hours function as an SparkSql function

val taxiClean = taxiGood.where(

"hours(pickupTime, dropoffTime) BETWEEN 0 AND 3"

)

//MAIN IDEA: Use Scala's Option[T] to deal with nulls and clean data using Sql

## //-Geospatial Analysis-//

/\*

Another instance we clean from the data are checking to see if trips start and end

long/lat are within NY Broughs.

\*/

//Read in our GeoJson file using the source class from scala.io

val geojson = scala.io.Source.

fromFile("/proj/cse398-498/course/aas/ch08-geotime/src/main/resources/nyc-boroughs.geojson").

mkString

//Using Esri and Spray to parse geojson to FeatureCollection

import com.cloudera.datascience.geotime.\_

import GeoJsonProtocol.\_

import spray.json.\_

val features = geojson.parseJson.convertTo[FeatureCollection]

//Lets try to test some random point and see where it may be

import com.esri.core.geometry.Point

val p = new Point(-73.994499, 40.75066)

val borough = features.find(f => f.geometry.contains(p))

// Some(Feature(Some(72),Map(boroughCode -> 1, borough -> "Manhattan", @id -> ...

/\*

To increase time efficency, we are going to take the boroughs that are largest

and move them to the top, that way, statistically, our most common points

will load faster since they are higher up the hierachy.

\*/

val areaSortedFeatures = features.sortBy(f => {

val borough = f("boroughCode").convertTo[Int] //switch boroughs to #1-5

(borough, -f.geometry.area2D())//based of 2d area

}) //scala auto sorts ascending order

//Write a function to to find which borough trips ended in

val bFeatures = sc.broadcast(areaSortedFeatures) //create copy to mess with

val bLookup = (x: Double, y: Double) => {

val feature: Option[Feature] = bFeatures.value.find(f => {

f.geometry.contains(new Point(x, y))

})

feature.map(f => {

f("borough").convertTo[String]

}).getOrElse("NA")

}

val boroughUDF = udf(bLookup) //convert to udf

//create a histogram of trips to borough

taxiClean.

groupBy(boroughUDF($"dropoffX", $"dropoffY")).

count().

show()

/\*

+-----------------------+--------+

|UDF(dropoffX, dropoffY)| count|

+-----------------------+--------+

| Queens| 672192|

| NA| 339037|

| Brooklyn| 715252|

| Staten Island| 3338|

| Manhattan|12979047|

| Bronx| 67434|

+-----------------------+--------+

\*/

/\*

Most are typically in Manhattan, which is not a suprised, but what is suprising

is the number of NA counts.

\*/

//Print out these NA points

taxiClean.

where(boroughUDF($"dropoffX", $"dropoffY") === "NA").

show()

/\*

|559F071794B721398...|1357615159000|1357616342000| 0.0| 0.0| 0.0| 0.0|

|A4B0B563E94A1C3AD...|1357593562000|1357593853000| 0.0| 0.0| 0.0| 0.0|

|559F071794B721398...|1357602162000|1357602593000| 0.0| 0.0| 0.0| 0.0|

|559F071794B721398...|1357607962000|1357608190000| 0.0| 0.0| 0.0| 0.0|

|4103ADCF50D18CFE2...|1358069880000|1358070240000| 0.0| 0.0| 0.0| 0.0|

|59BFB5C9B1E404F09...|1358093160000|1358093700000| 0.0| 0.0| 0.0| 0.0|

\*/

//Filter out all cases where start and stop are 0.0 using SparkSql

val taxiDone = taxiClean.where(

"dropoffX != 0 and dropoffY != 0 and pickupX != 0 and pickupY != 0"

).cache()

//Rerun histogram to show changes

taxiDone.

groupBy(boroughUDF($"dropoffX", $"dropoffY")).

count().

show()

/\*

+-----------------------+--------+

|UDF(dropoffX, dropoffY)| count|

+-----------------------+--------+

| Queens| 670912|

| NA| 62778|

| Brooklyn| 714659|

| Staten Island| 3333|

| Manhattan|12971314|

| Bronx| 67333|

+-----------------------+--------+

\*/

//Cut most of NA down, and some others entries in other boroughs

# //--Sessionization in Spark--//

/\*

Its been a while, but if we can recall, the point of this chapter was to find the

utilization of taxis in NY. For each driver, we need to sort each drive they had

per shift. This is called sessionization: the analysis on a single entity as it

executes a series of events over time.

Google does this for autocorrect. It tracks when an user types a query, stops,

goes back and types a modified version of the original query, and then googled it,

clicked on a new link, and didn't return back. This lets us build a better

autocorrect than any dictionary could ever. It can even idenitify mistakes even

when everything is spelt correctly: untied stats -> united states. This idea

is used for query suggestions to OneBox (so no clicks needed)

\*/

## //-Building Sessions: Secondary Sorts in Spark-//

/\*

The naive approach would be to use groupBy to create a session and then shift

the events around. This is not practical as it would require every entry to

be in memory and will not scale.

"In MapReduce, we can build sessions by performing a secondary sort, where we

create a composite key made up of an identifier and a timestamp value, sort all

of the records on the composite key, and then use a custom partitioner and grouping

function to ensure that all of the records for the same identifier appear in the

same output partition."

\*/

val sessions = taxiDone.

repartition($"license"). //make sure they have same license

sortWithinPartitions($"license", $"pickupTime") //then sort by pickupTime

sessions.cache()

//When working with large sets like this, it is useful to cache/export out

/\*

Create a method to calculate the amount of time from pickup to dropoff

and time inbetween to get next fare

\*/

case class Trip(

license: String,

pickupTime: Long,

dropoffTime: Long,

pickupX: Double,

pickupY: Double,

dropoffX: Double,

dropoffY: Double)

def boroughDuration(t1: Trip, t2: Trip): (String, Long) = {

val b = bLookup(t1.dropoffX, t1.dropoffY)

val d = (t2.pickupTime - t1.dropoffTime) / 1000

(b, d)

}

//Instead of using a loop to apply method to all sequential pairs, use sliding

val boroughDurations: DataFrame =

sessions.mapPartitions(trips => {

val iter: Iterator[Seq[Trip]] = trips.sliding(2)

val viter = iter.

filter(\_.size == 2). //ignore if there is only 2 trips

filter(p => p(0).license == p(1).license) //ignore if license not same

viter.map(p => boroughDuration(p(0), p(1)))

}).toDF("borough", "seconds") //returns as DF

//Show average and standard deviation of the pickup times by borough

boroughDurations.

where("seconds > 0 AND seconds < 60\*60\*4").

groupBy("borough").

agg(avg("seconds"), stddev("seconds")).

show()

/\*

+-------------+------------------+--------------------+

| borough| avg(seconds)|stddev\_samp(seconds)|

+-------------+------------------+--------------------+

| Queens|2380.6603554494727| 2206.6572799118035|

| NA| 2006.53571169866| 1997.0891370324784|

| Brooklyn| 1365.394576250576| 1612.9921698951398|

|Staten Island| 2723.5625| 2395.7745475546385|

| Manhattan| 631.8473780726746| 1042.919915477234|

| Bronx|1975.9209786770646| 1704.006452085683|

+-------------+------------------+--------------------+

\*/

/\*

As we expected Manhattan has the shortest time and somewhere

as far as Queens or Staten Island, it is a way longer time.

\*/

My Extension Writeup

# Cleaning the data even more

At the end of the chapter, it quickly mentioned that during the above data cleansing we removed instances where the X or Y of dropoff or pickup being 0, could be cleaned more. I feel like this ideology, while good and effective in removing numerous instances, could be optimized. Instead of finding out if the X or Y be 0, we should see if the distance traveled is 0. I realized this when I saw an instance where dropoff and pickup were the exact same location. We first start off by importing in scala sqrt, to calculate distance.

import scala.math.sqrt

//create a histogram of trips to borough

taxiClean.

groupBy(boroughUDF($"dropoffX", $"dropoffY")).

count().

show()

/\* Printed this off to show before and after

+-----------------------+--------+

|UDF(dropoffX, dropoffY)| count|

+-----------------------+--------+

| Queens| 672192|

| NA| 339037|

| Brooklyn| 715252|

| Staten Island| 3338|

| Manhattan|12979047|

| Bronx| 67434|

+-----------------------+--------+

\*/

val distance = (pickupX: Double, pickupY: Double, dropoffX: Double, dropoffY: Double) => {

sqrt((dropoffX - pickupX)\*(dropoffX - pickupX) + (dropoffY - pickupY)\*(dropoffY - pickupY))

}

spark.udf.register("distance", distance) //registering our hours function as an SparkSql function

val distanceUDF = udf(distance)

taxiClean.groupBy(distanceUDF($"pickupX", $"pickupY", $"dropoffX", $"dropoffY").as("h")).count().sort("h").show()

/\*

+--------------------+------+

| h| count|

+--------------------+------+

| 0.0|422606|

| 2.82842712474619E-6| 1|

|2.999999999531155...| 362|

|3.000000006636583E-6| 29|

|3.999999997006398E-6| 942|

|4.000000004111825...| 675|

|6.999999996537554E-6| 560|

|7.000000003642981E-6| 120|

|7.000000010748408E-6| 133|

|7.615773102496731...| 97|

|7.615773105295697E-6| 5|

|7.615773115558567E-6| 32|

|7.615773118357532E-6| 1|

|7.999999994012796E-6| 566|

|8.000000001118224E-6| 367|

|8.000000008223651E-6| 360|

|8.062257743807061E-6| 255|

|8.062257747332341E-6| 169|

| 8.06225775614554E-6| 78|

| 8.06225775967082E-6| 52|

+--------------------+------+

only showing top 20 rows

\*/

//Before removing all NA instances, there are over 400,000 0 distance entries in taxiClean

taxiDone.groupBy(distanceUDF($"pickupX", $"pickupY", $"dropoffX", $"dropoffY").as("h")).count().sort("h").show()

/\*

+--------------------+------+

| h| count|

+--------------------+------+

| 0.0|165564|

|2.999999999531155...| 361|

|3.000000006636583E-6| 29|

|3.999999997006398E-6| 940|

|4.000000004111825...| 673|

|6.999999996537554E-6| 560|

|7.000000003642981E-6| 119|

|7.000000010748408E-6| 133|

|7.615773102496731...| 97|

|7.615773105295697E-6| 5|

|7.615773115558567E-6| 32|

|7.615773118357532E-6| 1|

|7.999999994012796E-6| 566|

|8.000000001118224E-6| 365|

|8.000000008223651E-6| 360|

|8.062257743807061E-6| 255|

|8.062257747332341E-6| 169|

| 8.06225775614554E-6| 78|

| 8.06225775967082E-6| 52|

|8.544003739546916E-6| 120|

+--------------------+------+

\*/

//There is still over 150,000 entries... after removing all X and Y = 0. This proves my hypothesis that this methodology be optimized

//Rerun histogram to show before

taxiDone.

groupBy(boroughUDF($"dropoffX", $"dropoffY")).

count().

show()

/\*

+-----------------------+--------+

|UDF(dropoffX, dropoffY)| count|

+-----------------------+--------+

| Queens| 670912|

| NA| 62778|

| Brooklyn| 714659|

| Staten Island| 3333|

| Manhattan|12971314|

| Bronx| 67333|

+-----------------------+--------+

\*/

val taxiDone2 = taxiDone.where("distance(pickupX, pickupY, dropoffX, dropoffY) != 0") //apply conditional method statement to taxiDone

taxiDone2.groupBy(boroughUDF($"dropoffX", $"dropoffY")).count().show().cache()

/\*

+-----------------------+--------+

|UDF(dropoffX, dropoffY)| count|

+-----------------------+--------+

| Queens| 644722|

| NA| 55003|

| Brooklyn| 704557|

| Staten Island| 2963|

| Manhattan|12852982|

| Bronx| 64538|

+-----------------------+--------+

\*/

/\*

A bunch of entries disappeared from all boroughs. Seems like just

removing NA wasn't enough. Lets see if we can remove any instances

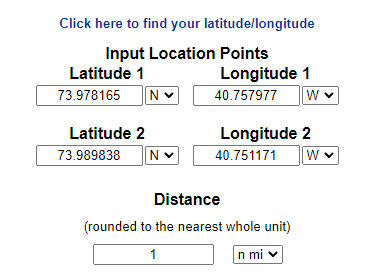
with less than one minute to clean the data even more.

\*/

taxiDone2.where(distanceUDF($"pickupX", $"pickupY", $"dropoffX", $"dropoffY") < 3).take(10).foreach(println) //example used to calculate distance

//Trip(BA96DE419E711691B9445D6A6307C170,1357071108000,1357071490000,-73.978165,40.757977,-73.989838,40.751171)

//3 distance (seen in histogram of distances) is less than one km



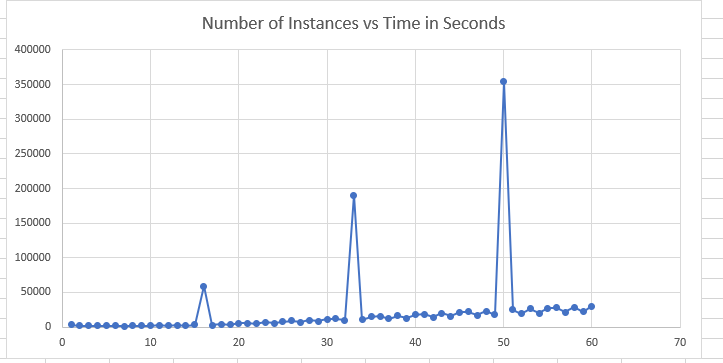
//Create seconds function

val seconds = (pickup: Long, dropoff: Long) => { TimeUnit.HOURS.convert(dropoff - pickup, TimeUnit.SECONDS)}

val secondsUDF = udf(seconds)

spark.udf.register("seconds", seconds) //registering our seconds function as an SparkSql function

taxiDone2.groupBy(secondsUDF($"pickupTime", $"dropoffTime").as("h")).count().sort("h").take(60).foreach(println)



val taxiDone3 = taxiDone2.where("seconds(pickupTime, dropoffTime) > 60 AND distance(pickupX, pickupY, dropoffX, dropoffY) < 3") //aproximately less than one minute and one km

taxiDone3.groupBy(boroughUDF($"dropoffX", $"dropoffY")).count().show

/\*taxiDone3 ←- MODIFIED VERSION

+-----------------------+--------+

|UDF(dropoffX, dropoffY)| count|

+-----------------------+--------+

| Queens| 619598|

| NA| 48708|

| Brooklyn| 681154|

| Staten Island| 2906|

| Manhattan|11630506|

| Bronx| 63620|

+-----------------------+--------+

\*/

/\*

scala> taxiDone3.count()

res62: Long = 13046492

REMOVED ADDITIONAL 1,443,837 bad instances

scala> taxiDone.count()

res61: Long = 14490329

\*/

/\*taxiDone ← ORIGINAL TEXTBOOK VERSION

+-----------------------+--------+

|UDF(dropoffX, dropoffY)| count|

+-----------------------+--------+

| Queens| 670912|

| NA| 62778|

| Brooklyn| 714659|

| Staten Island| 3333|

| Manhattan|12971314|

| Bronx| 67333|

+-----------------------+--------+

\*/

# Sessionization the newly cleaned taxiDone3

val sessions = taxiDone3.

repartition($"license").

sortWithinPartitions($"license", $"pickupTime").

cache()

def boroughDuration(t1: Trip, t2: Trip): (String, Long) = {

val b = bLookup(t1.dropoffX, t1.dropoffY)

val d = (t2.pickupTime - t1.dropoffTime) / 1000

(b, d)

}

val boroughDurations: DataFrame = sessions.mapPartitions(trips => {

val iter: Iterator[Seq[Trip]] = trips.sliding(2)

val viter = iter.filter(\_.size == 2).filter(p => p(0).license == p(1).license)

viter.map(p => boroughDuration(p(0), p(1)))

}).toDF("borough", "seconds")

boroughDurations.

where("seconds > 0").

groupBy("borough").

agg(avg("seconds"), stddev("seconds")).

show()

/\* taxidone (original)

+-------------+------------------+--------------------+

| borough| avg(seconds)|stddev\_samp(seconds)|

+-------------+------------------+--------------------+

| Queens| 15145.02921535893| 46184.65570022602|

| NA| 11145.50690421012| 41062.38476837451|

| Brooklyn|10924.258102953178| 40079.37390372924|

|Staten Island| 17012.34120171674| 41266.189555996105|

| Manhattan| 3441.172764592876| 22029.98741240281|

| Bronx|13846.641869522882| 41205.83813202717|

+-------------+------------------+--------------------+

\*/

/\* taxidone3

+-------------+------------------+--------------------+

| borough| avg(seconds)|stddev\_samp(seconds)|

+-------------+------------------+--------------------+

| Queens| 15987.87468371353| 46748.80682717808|

| NA|12499.481820445773| 44664.82118171125|

| Brooklyn|11703.938399555638| 41485.08013004814|

|Staten Island|18761.355064844025| 43003.091870988756|

| Manhattan| 3847.903044809918| 23382.97123334362|

| Bronx|14974.669308311672| 42677.088182751584|

+-------------+------------------+--------------------+

\*/

By removing over a million bad instances, our avg time has increased significantly, which makes mathematical sense. The new sessionization done on the newly cleaned set is a more accurate reflection of the actual duration of waiting. The data could technically be cleaned significantly more, if we reflect that most rides, are typically, over a mile (whereas here I only did anything less than a kilometer)

