# Project Report

## 1. Reflection

For the project, first I shall set up a working Spark environment for Windows OS, then I shall complete the provided four code templates for carrying out the two data analysis tasks: (a) hot zone analysis, (b) hot cell analysis.

Detailed seting-up document specific to Windows OS is provided in the Additional Resources: Hotspot Analysis Project webpage. I followed the step-by-step instructions to set up Java JDK 17, SBT, Apache Spark, hadoop.dll, winutils.exe.

To carry out the two data analysis tasks, there are the provided four code templates ( HotzoneUtils.scala, HotzoneAnalysis.scala, HotcellUtils.scala, HotcellAnalysis.scala ) for me to complete. Basically, Scala utility function code should be defined in ( HotzoneUtils.scala, HotcellUtils.scala ), and Spark query code should be defined in ( HotzoneAnalysis.scala, HotcellAnalysis.scala ).

Raw data are stored in the csv files ( zone-hotzone.csv, point-hotzone.csv ) and ( yellow_trip_sample_100000.csv ) for hot zone analysis and hot cell analysis respectively. Data are relatively small and can be easily analyzed using Pandas and Data-Frame (Python). For the project, I will implement the same data analysis but using Spark, Data-Frame and Temp-View.

## 2. Lessons Learned

When I tried to set up a Spark environment for Windows OS, I was redirected to the repository to download hadoop.dll and winutils.exe. One right way to download is to right-click on “raw” and then click “save link as”. I downloaded something incorrect (though name and extension appear OK) and did not realize that immediately, since they are not installers and no follow-up installation process.

In Spark, I can work with either Data-Frame or Temp-View. Query commands for Temp-View looks closer to PostgreSQL commands, so Temp-View would be easier for me to use at present.

## 3. Implementation

(a) for hot zone analysis, rectangles are stored in zone-hotzone.csv, and points are stored in point-hotzone.csv. Our goal is to compute how many points each rectangle contains.

1.

```
def ST_Contains(queryRectangle: String, pointString: String)
```

Given a rectangle represented by two corners (x1, y1, x2, y2) and a point (x, y), the utility function is to check if the point is located within the rectangle.

2.

```
val finalDf = spark.sql("select rectangle, count(point) from joinResult group by rectangle order
by rectangle").persist().coalesce(1)
```

joinResult Temp-View is already predefined in the code template, the query code is to compute how many points each rectangle contains.

(b) for hot cell analysis, the trips are stored in yellow_trip_sample_100000.csv. The trips are distributed into three-dimensional cells labeled by (x, y, z) where x, y represent a location and z represent a day. The space discretized by parameters (minx, maxX, minY, maxY, minZ, maxZ, numCells) are already predefined in the code template. Our goal is to compute how many points each cell contains, then what gScore each cell has.

1.

```
def countNeighbors(minX:Int, maxX:Int, minY:Int, maxY:Int, minZ:Int, maxZ:Int, X:Int, Y:Int, Z:Int)
```

Given a cell (X, Y, Z), the utility function is to determine how may neighbors the cell has. Internally, the code checks how many boundary-planes the cell can touch, and there are four situations in total to check.

(0 boundary-planes → 26 neighbors; 1 boundary planes → 17 neighbors
2 boundary-planes → 11 neighbors; 3 boundary-planes → 7 neighbors)

2.

```
val validCellsDf = spark.sql("select x, y, z from pickupInfoView where isValidCell(x, y, z) order by z, y, x").persist()
```

The query code is to firstly select all the valid cells. The function IsValidCell(x, y, z) is very simple that check the condition (x >= minX && x <= maxX && y >= minY && y <= maxY && z >= minZ && z <= maxZ)

3.

```
val attrxOfCellsDf = spark.sql("select x, y, z, count(*) as attrx from validCellsView group by z, y, x order by z, y, x").persist()
```

The query code is to compute how many points each cell contains. And it is named and saved as attrx for each cell.

4.

```
val muSigmaDf = spark.sql("select avg(attrx) as mu, stddev_pop(attrx) as sigma from attrxOfCellsView")
val mu = muSigmaDf.first().getAs[Double]("mu")
val sigma = muSigmaDf.first().getAs[Double]("sigma")
```

The query code is to compute average and standard deviation of attrx. And they are two useful parameters to compute gScore in the end.

5.

```
var localSumDf = spark.sql("""
   SELECT v1.x AS x, v1.y AS y, v1.z AS z, sum(v2.attrx) AS localSum
   FROM attrxOfCellsView AS v1, attrxOfCellsView AS v2
   WHERE ABS(v2.x - v1.x) <= 1 AND ABS(v2.y - v1.y) <= 1 AND ABS(v2.z - v1.z) <= 1
   GROUP BY v1.z, v1.y, v1.x
   ORDER BY v1.z, v1.y, v1.x
 """)
```

The query code is to compute the attrix sum of each cell and its neighbors. And it is named and saved as localSum for each cell.

6.

```
var localCountDf = spark.sql(s"""
   SELECT *, countNeighbors($minX, $maxX, $minY, $maxY, $minZ, $maxZ, x, y, z) AS localCount
   FROM localSumView
 """)
```

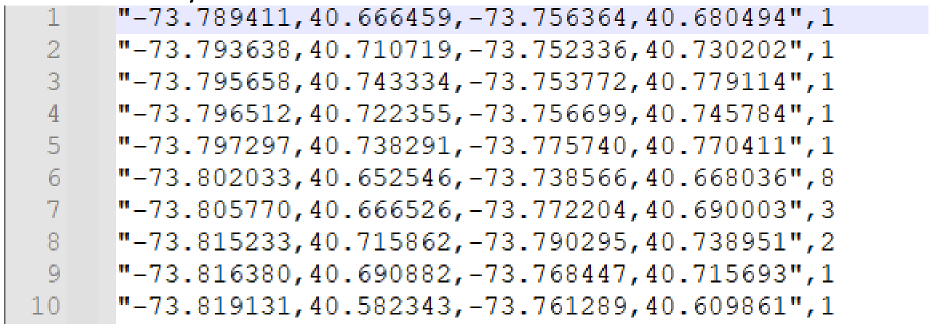
The query code is to compute the number of neighbors each cell has. And it is named and saved as localCount for each cell.
7.

```
val gScoreDf = spark.sql(s"""
   SELECT *, computeGScore(x, y, z, $numCells, $mu, $sigma, localCount, localSum) AS gScore
   FROM localCountView
   ORDER BY gScore DESC
   LIMIT 50
 """)
```

Finally, the query code is to compute what gScore each cell has using the above computed parameters.

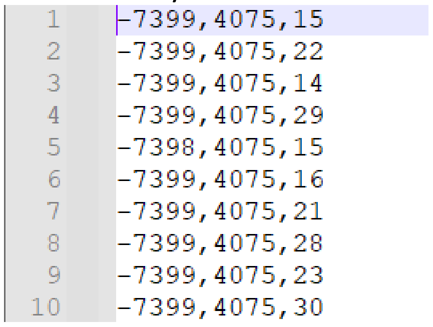
## 4. Output

Hot zone Analysis



```
"-73.789411,40.666459,-73.756364,40.680494",1
"-73.793638,40.710719,-73.752336,40.730202",1
"-73.795658,40.743334,-73.753772,40.779114",1
"-73.796512,40.722355,-73.756699,40.745784",1
"-73.797297,40.738291,-73.775740,40.770411",1
"-73.802033,40.652546,-73.738566,40.668036",8
"-73.805770,40.666526,-73.772204,40.690003",3
"-73.815233,40.715862,-73.790295,40.738951",2
"-73.816380,40.690882,-73.768447,40.715693",1
"-73.819131,40.582343,-73.761289,40.609861",1
```

Hot cell Analysis



```
-7399,4075,15
-7399,4075,22
-7399,4075,14
-7399,4075,29
-7398,4075,15
-7399,4075,16
-7399,4075,21
-7399,4075,28
-7399,4075,23
-7399,4075,30
```