PROJECT 2 – HOT SPOT ANALYSIS INDIVIDUAL REPORT

- NAVEEN KUMAR MANOKARAN STUDENT ID -1232482864

The Project was very useful and taught me a lot about how to work on a dataset containing geospatial data. The project helped in learning about handling geospatial data, how to read the geospatial data from a csv file, how to find if a given point in space is a part of a rectangle when the coordinates of the rectangle and the point is given using that finding the hot zones the rectangle with the most points and find the hot cells using the Getis-Ord statistic.

REFLECTION:

I initially started the project by making a list of the technology stacks that are required for the project and getting a detailed knowledge about the technology stacks. Technology stacks that are used in the project include Apache Spark, SparkSQL, Scala, Java and Hadoop.

Once I gathered enough knowledge on the technology stacks that are required for the project I started to install the required tech stacks for the project. I am using a windows machine, so I installed all the required tech stacks as listed in the windows installer.

```
PS C:\spark-local\spark\bin> java -version
openjdk version "1.8.0_412"
OpenJDK Runtime Environment Corretto-8.412.08.1 (build 1.8.0_412-b08)
OpenJDK 64-Bit Server VM Corretto-8.412.08.1 (build 25.412-b08, mixed mode)
PS C:\spark-local\spark\bin> .\spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://host.docker.internal:4040
Spark context available as 'sc' (master = local[*], app id = local-1714368496796).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.7
      /_/

Using Scala version 2.11.12 (OpenJDK 64-Bit Server VM, Java 1.8.0_412)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

Once the required tech stacks are installed and the IDE is set up. I started to code. There are 2 parts of the project Hot Zone Analysis and Hot Cell Analysis. First, I started with Hot Zone Analysis. For this part we need to change the **HotzoneUtils** and

HotzoneAnalysis files. The **HotzoneUtils** contains the ST_Contains function that will take the in the rectangle and point as string inputs and calculate if the given point is a part of the given rectangle.

```
object HotzoneUtils {

  def ST_Contains(queryRectangle: String, pointString: String ): Boolean = {

    // check if the inputs strings are empty, if true then return false
    if (queryRectangle == null || queryRectangle.isEmpty() || pointString == null || pointString.isEmpty())
        return false

    val rectangleArr = queryRectangle.split(",")
    var x1 = rectangleArr(0).toDouble
    var y1 = rectangleArr(1).toDouble
    var x2 = rectangleArr(2).toDouble
    var y2 = rectangleArr(3).toDouble

    val pointArr = pointString.split(",")
    var x = pointArr(0).toDouble
    var y = pointArr(1).toDouble

    if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
        return true
    else if (x >= x2 && x <= x1 && y >= y2 && y <= y1)
        return true
    else
        return false
  }
}
```

The **HotzoneAnalysis** file will use this ST_Contains function to check if the point is a part of the rectangle and then count the points based on the rectangle using group by function and then sort the counts in descending order to get the hot zones with the most points.

```
// Join two datasets
spark.udf.register("ST_Contains",(queryRectangle:String, pointString:String)=>(HotzoneUtils.ST_Contains(queryRectangle, pointString)))
val joinDf = spark.sql("select rectangle._c0 as rectangle, point._c5 as point from rectangle,point where ST_Contains(rectangle._c0,point._c5)")
joinDf.createOrReplaceTempView("joinResult")

val countDataframe = joinDf.groupBy("rectangle").count()
val outputDataframe = countDataframe.sort("rectangle").coalesce(1)

outputDataframe
```

The next part is Hot Cell Analysis in this part we need to change the files **HotcellAnalysis** and **HotcellUtils**. The HotcellUtils file contains 2 functions. The computeAdjacentcell which calculates the total number of neighbors for each cell in a 3*3 matrix that is considered for this project. The GScore calculates the Getis-Ord statistic for each cell using the mean, standard deviation, number of cells and adjacent hot cell values. Using the formulae given below.

$$G_i^* = \frac{\sum_{j=1}^{n} w_{i,j} x_j - \bar{X} \sum_{j=1}^{n} w_{i,j}}{S \sqrt{\frac{\left[n \sum_{j=1}^{n} w_{i,j}^2 - \left(\sum_{j=1}^{n} w_{i,j} \right)^2 \right]}{n-1}}}$$

The code for the computeAdjacentcell and GScore functions are given below.

```
def computeAdjacentcell( minX: Double, minY: Double, minZ: Double, maxX: Double, maxY: Double, maxZ: Double, X: Double, Y: Double, Z: Double): Double = {
  var count = 0

  if (X == minX || X == maxX) {
      count += 1
  }
  if (Y == minY || Y == maxY) {
    count += 1
  }
  if (Z == minZ || Z == maxZ) {
    count += 1
  }

  count match {
    case 1 => 18
    case 2 => 12
    case 3 => 8
    case _ => 27
  }
}

def GScore(numCells: Int, x: Int, y: Int, z: Int, adjacentHotcell: Int, cellNumber: Int , avg: Double, stdDev: Double): Double = {
  var adjHotCell: Double = adjacentHotcell.toDouble
  var numOfCells: Double = numCells.toDouble
  (cellNumber.toDouble - (avg * adjHotCell)) / (stdDev * math.sqrt((( adjHotCell * numOfCells) - (adjHotCell * adjHotCell)) / (numOfCells - 1.0)))
}
```

The **HotCellAnalysis** page is then altered to calculate the average, standard deviation and adjacent hot cell values which are then substituted to the Getis Ord formulae to get the GScore values. The finals rows are then sorted based on the descending GScore values. The final x,y and z values of the sorted rows are passed as the output of the file. The formulae to calculate the average and standard deviation are given below. S is standard deviation and X is the mean.

$$S = \sqrt{\frac{\sum_{j=1}^{n} x_j^2}{n} - (\bar{X})^2} \qquad \bar{X} = \frac{\sum_{j=1}^{n} x_j}{n}$$

The code for **HotCellAnalysis** is given below.

```
pickupInfo = pickupInfo.select("x", "y", "z").where("x >= " + minX + " AND y >= " + minY + " AND z >= " + minZ + " AND x <= " + maxX + " AND y <= " + maxY + " AND z <= " + maxZ).orderBy("z", "y", "x")
var hotCellDataFrame = pickupInfo.groupBy("z", "y", "x").count().withColumnRenamed("count", "hot_cell").orderBy("z", "y", "x")
hotCellDataFrame.createOrReplaceTempView("Hotcell")

// Calculate Averge
val avg = (hotCellDataFrame.select("hot_cell").agg(sum("hot_cell")).first().getLong(0).toDouble) / numCells

// Calculate Standard Deviation
val stdDev = scala.math.sqrt((hotCellDataFrame.withColumn("sqr_cell", pow(col("hot_cell"), 2)).select("sqr_cell").agg(sum("sqr_cell")).first().getDouble(0) / numCells) - scala.math.pow(avg, 2))

var adHotCell = spark.sql("SELECT h1.x AS x, h1.y AS y, h1.z AS z, "
+ "sum(h2.hot_cell) AS cellNumber "
+ "FROM Hotcell AS h1, Hotcell AS h2 "
+ "WHERE (h2.y = h1.y+1 OR h2.y = h1.y OR h2.y = h1.y-1) AND (h2.x = h1.x+1 OR h2.x = h1.x OR h2.x = h1.x-1) AND (h2.z = h1.z+1 OR h2.z = h1.z OR h2.z = h1.z-1)"
+ "GROUP BY h1.z, h1.y, h1.x "
+ "ORDER BY h1.z, h1.y, h1.x" )

//calculate adjacent cell number
var calculateNumberAdjFunc = udf((minX: Double, minY: Double, minZ: Double, maxX: Double, maxY: Double, maxZ: Double, X: Double, Y: Double, Z: Double) => HotcellUtils.computeAdjacentcell(minX, minY, minZ, maxX, maxY, maxZ, X, Y, Z))
var adjacentHotcell = adHotCell.withColumn("adjacentHotcell", calculateNumberAdjFunc(lit(minX), lit(minY), lit(minZ), lit(maxX), lit(maxY), lit(maxZ), col("x"), col("y"), col("z")))

//calculate Getis-Ord
var gScoreFunc = udf((numCells: Int , x: Int, y: Int, z: Int, adjacentHotcell: Int, cellNumber: Int, avg: Double, stdDev: Double) => HotcellUtils.GScore(numCells, x, y, z, adjacentHotcell, cellNumber, avg, stdDev))
var gScoreHotCell = adjacentHotcell.withColumn("gScore", gScoreFunc(lit(numCells), col("x"), col("y"), col("z"), col("adjacentHotcell"), col("cellNumber"), lit(avg), lit(stdDev))).orderBy(desc("gScore")).limit(50)

pickupInfo = gScoreHotCell.select(col("x"), col("y"), col("z"))
pickupInfo
```

LESSONS LEARNED:

1. How to work on geospatial dataset
2. How to setup and work on Apache Spark and Scala.
3. The difference between Hot Zone Analysis and Hot Cell Analysis
4. Getis-Ord statistics to find the hot cells.
5. How to split the geospatial data into zones and analyze the results for each zones.

OUTPUT:

In this project I learned how to process geospatial data using SQL and Scala in the Apache Spark environment. Finding the most important zones and cells (hot zones/cells) is the biggest goal in the processing of geographic information. We used G* statistics to find statistically significant locations (hot cells). This metric helps us to determine where attributes with either high or low values are grouped spatially. The higher G-score (z-score) for a specific cell shows that cell contains more points than average points in the other cells.

The outputs are like the output testcases provided in the input the code is submitted using the code below.

```
PS C:\BigDataAtScale\CSE511-Project-Hotspot-Analysis-20200804T194129Z\CSE511-Project-Hotspot-Analysis> C:\spark-local\spark\bin\spark-submit target
\scala-2.11\CSE512-Hotspot-Analysis-Template-assembly-0.1.0.jar test\output hotcellanalysis src\resources\yellow_trip_sample_100000.csv
PS C:\BigDataAtScale\CSE511-Project-Hotspot-Analysis-20200804T194129Z\CSE511-Project-Hotspot-Analysis>
```

REFERENCES:

1. http://sigspatial2016.sigspatial.org/giscup2016/problem - ACM SIGSPATIAL GISCUP 2016