Part 1. Hot zone analysis

1. Reflection
   a. Reflection

For the part 1, I need to find the number of points in the rectangles in the given skeleton code and dataset.

```
spark.udf.register( name= "ST_Contains",(queryRectangle:String, pointString:String)=>(HotzoneUtils.ST_Contains(queryRectangle, pointString)))
```

User defined function and spark SQL for finding points in rectangles has been already defined.

```
val joinDf = spark.sql( sqlText= "select rectangle._c0 as rectangle, point._c5 as point from rectangle,point where ST_Contains(rectangle._c0,point._c5)")
joinDf.createOrReplaceTempView( viewName= "joinResult")
```

I only need to modify “ST_Contains” function to complete part 1 like below.

```
def ST_Contains(queryRectangle: String, pointString: String ): Boolean = {
  val rectangle = queryRectangle.split( regex = ",")
  val point = pointString.split( regex = ",")
  if (rectangle(0).toDouble <= point(0).toDouble && rectangle(2).toDouble >= point(0).toDouble
    && rectangle(1).toDouble <= point(1).toDouble && rectangle(3).toDouble >= point(1).toDouble)
  {
    return true
  }
  else {
    return false
  }
}
```

Only when the point is in the rectangle, the function returns true. Otherwise, the function returns false. Spark SQL only selects rectangles and points when where is true.

```
val resultDf = joinDf.groupBy( col1 = "rectangle").count().orderBy( sortCol = "rectangle")

return resultDf.coalesce( numPartitions = 1)
```

Then, using Group by and order by, I need to count and sort the joinDf dataframe.
Result:

1	-73.789411,40.666459,-73.756364,40.680494	1
2	-73.793638,40.710719,-73.752336,40.730202	1
3	-73.795658,40.743334,-73.753772,40.779114	1
4	-73.796512,40.722355,-73.756699,40.745784	1
5	-73.797297,40.738291,-73.775740,40.770411	1
6	-73.802033,40.652546,-73.738566,40.668036	8
7	-73.805770,40.666526,-73.772204,40.690003	3
8	-73.815233,40.715862,-73.790295,40.738951	2
9	-73.816380,40.690882,-73.768447,40.715693	1
10	-73.819131,40.582343,-73.761289,40.609861	1
11	-73.825921,40.702281,-73.790734,40.719217	2
12	-73.826577,40.757744,-73.790317,40.779587	1
13	-73.832707,40.620010,-73.746541,40.665414	200

   b. Implications for future learning

We may use user defined function in Spark SQL to manipulate and massage data for data preprocessing or ETL in MCS course as well as at work.

2. Analysis
   a. Lessons Learned

From part1 Hot zone analysis, I learned how to work Spark SQL with scalar. At starting this project, because it was my first time to use Spark and Scala language, I had to learn basic Spark SQL and Scala. After that, I understood the given skeleton code. I have been used Visual Studio Code and Pycharm, but IntelliJ tool was also great tool to debug the code.

   b. Assessment/Grading

For the first submission, I could not get full mark because I did not consider the border line of rectangles. After including equal signs (<= or >=) in if condition, finally I was able to get full marks for the part 1.

Part 2. Hot cell analysis

1. Reflection
   a. Reflection

For the part 2, This task will focus on applying spatial statistics to spatio-temporal big data in order to identify statistically significant spatial hot spots using Apache Spark.
I need to calculate z-score and sort in descending order for hot cell analysis.

```
val countedDf = spark.sql( sqlText = "select x,y,z,count(*) as c from pickupInfo group by x,y,z")
```

First, using spark.sql, the query select the cell coordinate and count pickups to get how many pickups in the cell.

```
val sum = spark.sql( sqlText = "select sum(c) from countedDf").first()(0).asInstanceOf[Long]
val mean = sum / numCells
println("mean= " + mean)
val sum2 = spark.sql( sqlText = "select sum(c*c) from countedDf").first()(0).asInstanceOf[Long]
val std = sqrt((sum2/numCells)-(mean*mean))
println("std = " + std)
```

Then, given the formulas, mean X and std S were calculated using spark.sql.
I used user defined function to calculate weight w based on cell boundaries.

```
def CalculateW(x : Int, y : Int, z : Int): Int =
{
  // Configuration variable:
  // Coordinate step is the size of each cell on x and y
  var result = 27
  if (x == -7450 || x == -7370 || y == 4050 || y == 4090 || z == 1 || z == 31) {
    result = 18
  }
  if ( ((x == -7450 || x == -7370) && (y == 4050 || y == 4090)) || ((y == 4050 || y == 4090) &&
    (z == 1 || z == 31)) || ((x == -7450 || x == -7370) && (z == 1 || z == 31)) ) {
    result = 12
  }
  if ((x == -7450 || x == -7370) && (y == 4050 || y == 4090) && (z == 1 || z == 31)) {
    result = 8
  }
  return result
}
```

Return 18 if point lies on x, y, or z-boundary
Return 12 if point lies on x and y boundary, y and z boundary, or x and z boundary
Return 8 if point lies on x, y, and z boundary

```
spark.udf.register( name = "CalculateW",(x: Int, y: Int, z: Int)=>((
  HotcellUtils.CalculateW(x, y, z)
  )))
val sumDf= spark.sql( sqlText = "select df1.x, df1.y, df1.z, CalculateW(df1.x, df1.y, df1.z) as w, sum(df2.c) as sum from countedDf as df1, countedDf as df2 " +
  "where df1.x<=df2.x + 1 and df1.x >= df2.x - 1 and df1.y<=df2.y + 1 and df1.y >= df2.y - 1 and df1.z<=df2.z + 1 and df1.z >= df2.z - 1 group by df1.x, df1.y, df1.z")
```

Using spark.sql, self-join, and UDF, I was able to create the dataframe which includes cell coordinates and sum of counts in the range of neighbor 27 cells. This query was a game changer and very important for part 2.

```
var resultDf= sumDf.withColumn( colName = "z-score", (col( colName = "sum") - (col( colName = "w") * mean)) / functions.sqrt(((col( colName = "w")*numCells)
  - (col( colName = "w")*col( colName = "w")))/(numCells-1)) / std )
```

Using withColumn method, it is easy to calculate “z-score”

```
resultDf = resultDf.orderBy(desc( columnName = "z-score"))
//resultDf.show()
resultDf = resultDf.drop( colNames = "w", "sum", "z-score")
```

After that, sort “z-score” in descending order and drop columns “w”, “sum”, and “z-score” to get the result below.

1	-7399	4075	15	79.39845
2	-7399	4075	22	77.06822
3	-7399	4075	14	76.25263
4	-7399	4075	29	76.05845
5	-7398	4075	15	75.61182
6	-7399	4075	16	75.2817

   b. Implications for future learning

I may use user defined function in Spark SQL to manipulate and massage data for data preprocessing or ETL in MCS course as well as at work. Using withColumn method in spark, I would be able to update all rows of a column easily and fast.

2. Analysis
   a. Lessons Learned

      From part2 Hot cell analysis, at first, I did not use spark sql and used “for-loop” with dataframe to obtain each row of dataframe. but, it was too slow for autograder to grade error with “time out” and it took so much time to calculate “z-score”. After modifying the code using sqark sql, I figured out that spark sql is much faster than “for-loop” because spark uses memory. Spark SQL is optimized for big data processing due to the speed, which make it easy to utilize deep learning with big data set in MCS course as well as at work.

   b. Assessment/Grading

      Even though the result is almost the same as the answer, I could not get full marks. After spending some time, I figured out the weights on boundaries would be different. In the most cases, weight is 27, but the weight on boundaries is different. Thus, I modified the code based on the following condition in the function.

```
def CalculateW(x : Int, y : Int, z : Int): Int =
{
  // Configuration variable:
  // Coordinate step is the size of each cell on x and y
  var result = 27
  if (x == -7450 || x == -7370 || y == 4050 || y == 4090 || z == 1 || z == 31) {
    result = 18
  }
  if ( ((x == -7450 || x == -7370) && (y == 4050 || y == 4090)) || ((y == 4050 || y == 4090) &&
    (z == 1 || z == 31)) || ((x == -7450 || x == -7370) && (z == 1 || z == 31)) ) {
    result = 12
  }
  if ((x == -7450 || x == -7370) && (y == 4050 || y == 4090) && (z == 1 || z == 31)) {
    result = 8
  }
  return result
}
```

      Return 18 if point lies on x, y, or z boundary. Return 12 if point lies on x and y boundary, y and z boundary, or x and z boundary. Return 8 if point lies on x, y, and z boundary. These solutions helped me to obtain full marks.

      Because Spark SQL is very fast for the queries, it is the great database system for big data processing. I could not believe it takes only few minutes to retrieve information from over 2GB CSV file using Spark SQL. Data keeps growing up and bigger. We need to deal with big data efficiently and effectively to use deep learning technologies which need big datasets.