Introduction

With advances in database technology and tools used to analyze databases, companies have become able to analyze new parts of their businesses. One area of analysis that has gained a lot of attention is geospatial analytics. Companies specializing in driver management, like delivery networks and taxi services, can use software like Apache Spark and Scala, to find patterns in the patterns of their customers. For this project, I am analyzing geographic data gathered from a taxi company database of pickup locations across multiple years in order to calculate popularity of certain locations. This project will be made up of two algorithms: hotzoneanalysis, which calculates the number of visits for each zone, and hotcellanalysis, which calculates the popularity of a zone over a time period using the Getis-Ord (Gi*) statistic.

Reflection and Implementation

In order to start my project, I had to gain an understanding of the tools required for this project, the taxi company geo-spatial data, and the template code provided to load in the specific data used by the taxi company. Due to the ability to optimally load and manipulate data, I utilized Apache Spark and Scala to analyze the taxi company's data. This project was the first time I had utilized these tools, so I started by analyzing the template code in order to focus on the correct areas required to complete this project. The starter code uses Apache Spark's read and SQL select statements to extract only the required data for each subtask. This is necessary, because the data provided by the taxi company includes unnecessary data like the price of each trip. The template code is split into two mutually exclusive sections: HotZoneAnalysis and HotCellAnalysis. For this project, I chose to start with the HotZoneAnalysis.

The goal of the HotZoneAnalysis algorithm was to calculate the number of taxi pickups occur in specific zones and return this information in a csv file. The HotZoneAnalysis code takes in two inputs with the spark session information: pointPath and rectanglePath. The pointPath is the file location of the data for each pickup spot point, and the rectanglePath is the file location of the data for each zone, represented in rectangle form through two of the zone's diagonal data points. In order to select all of the data points in each rectangle, the HotZoneAnalysis code uses a function in HotZoneUtils called ST_Contains. The goal of ST_Contains is that, using a point string and a rectangle string as input, it can determine whether a point is contained within the rectangle. To implement this function, I split the code into three parts. Firstly, the point string and rectangle string are split using the split command on each comma in the string. The point string is split into point_x and point_y, and the rectangle string is split into corner_one_x, corner_one_y, corner_two_x, and corner_two_y values. Secondly, I use the Math library's max and min functions to select the rectangles maximum and minimum coordinate values and assign them to variables: max_rect_x, min_rect_x, max_rect_y, and min_rect_y. Finally, I checked for each point coordinate values if it was greater than the maximum value of the coordinate or less than the minimum value of the coordinate. If either of those conditions were met for the point's x and y values, ST_Contains returned false. If none of the conditions were met, ST_Contains returned true. After selecting for only the points in a zone, I made sure that the result csv for the HotZoneAnalysis algorithm fit the required specifications by grouping points by their rectangle string and coalescing all of the partitions into one file.

The goal of the HotCellAnalysis code was to determine the top 50 cells for popularity by using the G* statistic to represent popularity. Unlike the HotZoneAnalysis code, the HotCellAnalysis code adds in the z dimension, which corresponds to the pick up time. The Gi* statistic formula uses neighboring cell values in order to calculate the popularity of certain cells, with the highest neighboring cell values resulting in the largest Gi* values. To run this formula, I created a helper function in HotCellUtils called calculateGetisOrd. This function takes in the neighbor sum, the neighbor count, the mean, standard deviation, and n value to calculate the formula $Gi^* = (\Sigma(wij * xj) - \bar{X} * \Sigma(wij)) / (S * sqrt((n*\Sigma(wij^2) - (\Sigma(wij))^2) / (n-1)))$. In order to use this function I had to create code to calculate all of these values. Firstly, I gathered all of the valid cells from the data and calculated the number of pickup points in that cell. Once I got the cell counts, I calculated the global mean, standard deviation, and n value. Secondly, I calculated the sequence of values to connect the neighboring cells. This spanned 27 cell values from (-1,-1,-1) to (1,1,1) and were transformed into a dataframe called offsetsDF. With all of these values, I joined the cell count data on the offsets dataframe by selecting values from each coordinate values added to the neighboring cell values. After gathering all of the cells with their neighboring cells, I calculated the neighborCount and neighborSums and used all of the information to add a column called gistar corresponding to the values found from running the rows in calculateGetisOrd. After each Gi* value were added, I ordered the data using gistar, and selected to x, y, and z columns from the top 50 cells.

Lessons Learned

This was my first time using Scala and I was impressed by the good library functions for running joins and other large data calculations. It was also my first time using Apache Spark to package scala files into a jar file, and then run the jar files using spark-submit. This process was much more difficult than it had to be for me due to the naming of the test file names switching from dash to underscore, and the inability to make spark-submit work without temporarily adding the path to the terminal window. My other classes in my master program covered calculating and using getis ord statistic but it was useful creating code to calculate on realistic data. The calculation is much more complex when using a z dimension, and I had issues verifying my formula was incorrect because I didn't include the z score in the test files to see that I was clearly way off of my calculation. I ended up not including the cell itself and only included the neighbors, so I had to do awkward joins to make the neighbor joins work well. The concept for this project ended up being more challenging and interesting than the NoSQL project, due to the use of a larger data set, more complicated input data, and the raw data being provided that needed to be modified. Compared to undergraduate projects, this class provided more realistic scenarios for the data provided in the real world (especially with the growth of sensor data).