CS 6240: Final Report

Team Members (MR001**)**
Aman Rayat
Surbhi Garg
Sushmita Chaudhari

Project Overview:
In this project, we explored more about Apache Spark using Scala by addressing different problems.
Our first goal was to count the number of distinct triangles in Twitter graph by using RS join and Rep join algorithms in Spark Scala. Our second goal was to apply Full pagerank algorithm on the dataset to find top 100 influential users. The third goal of this project was to predict the fare amount for a taxi ride in New York City with RandomForestRegressor algorithm using Apache Spark MLlib library

Our two major tasks involve using Twitter dataset. In the first task we achieved our goal of calculating unique social triangles for twitter followers. In the second task, we found the 100 most popular Twitter users. In the third task, we predicted the yellow taxi fare in New York City.

Input Data
Task 1: Revisit Scala Homework & Task 2: Full Page Rank
http://socialcomputing.asu.edu/datasets/Twitter
The input shows the friendship/followership network among the users. The friends/followers are represented using edges. Edges are directed.

Sample data: 1,2
This means the user with id "1" is following the user with id "2".

Task 3: Classification and Prediction in Spark MLlib :
https://www.kaggle.com/c/new-york-city-taxi-fare-prediction/data
This data has 55M rows

Sample data:
key,fare_amount,pickup_datetime,pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_latitude,passenger_count
2009-06-15 17:26:21.0000001,4.5,2009-06-15 17:26:21 UTC, -73.844311, 40.721319, -73.84161, 40.712278, 1

Task 1: Revisit Scala Homework

Overview
We want to count the number of distinct triangles in the Twitter graph or a social amplifier pattern. A triangle (X, Y, Z) is a triple of user IDs, such that there exist three edges (X, Y), (Y, Z), and (Z, X). We will solve this problem with RS-join and Rep-join in Spark Scala with RDD and DSET.

Pseudo-Code:

Triangle Count Partition and Broadcast - RDD
// Setting up the HashMap

```
val emptySet = mutable.HashSet.empty[String]
val addToSet = (s: mutable.HashSet[String], v: String) => s += v
val mergeSets = (p1: mutable.HashSet[String], p2: mutable.HashSet[String]) => p1 ++= p2

//  BroadCasting the Hashmap
val broadcastRDD = forwardEdges.aggregateByKey(emptySet)(addToSet, mergeSets)
val broadcast = sc.broadcast(broadcastRDD.collectAsMap())

//    Calculating the chevron
  val chevron = reverseEdges.mapPartitions(edge => {edge.flatMap { case (k, e1) =>
        broadcast.value.get(k) match {case None => Seq.empty[(String, String)]
          case Some(e2) => var emptySeq = Seq.empty[(String, String)]
            e2.foreach(e3 => emptySeq = emptySeq :+ (e3, e1))
            emptySeq}}}}, preservesPartitioning = true)

 val triangle = chevron.mapPartitions(edge => {edge.flatMap {
       case (k, e1) =>broadcast.value.get(k) match {case None => Seq.empty[(String, String)]
          case Some(e2) => if (e2.contains(e1)) Seq((e1, e1)) else Seq.empty[(String, String)]}}
    }, preservesPartitioning = true)
```

Algorithm and Program Analysis:

We have implemented the Hash Partitioning and Partition and Broadcast algorithm for both RDD and DSET. The above pseudo code is for Partition and Broadcast. If we increase the side of the MAX value for this program running time increased significantly since in this algorithm there is a memory constraint on the size of the broadcast. If we increase the number of machines it showed speedup.

Experiments :

We ran our program on 5 and 10 machines with Machine type M4 xLarge on twitter dataset by varying Maximum Follower id.

Triangle Count Hash and Shuffle - RDD

Sno	Nodes	Max Follower Id	Machine Type
1	1 Master, 5 nodes	30000,10000	M4 xLarge
2	1 Master, 10 nodes	30000,10000	M4 xLarge

Triangle Count Hash and Shuffle - Dataset

Sno	Nodes	Max Follower Id	Machine Type
1	1 Master, 5 nodes	75000, 30000	M4 xLarge
2	1 Master, 10 nodes	75000, 30000	M4 xLarge

Triangle Count Partition and Broadcast - Dataset

Sno	Nodes	Max Follower Id	Machine Type
1	1 Master, 5 nodes	75000,30000	M4 xLarge

2	1 Master, 10 nodes	75000,30000	M4 xLarge

Triangle Count Partition and Broadcast - RDD

Sno	**Nodes**	**Max Follower Id**	**Machine Type**
1	1 Master, 5 nodes	10000,5000	M4 xLarge
2	1 Master, 10 nodes	10000,5000	M4 xLarge

Speedup

To achieve speedup we ran our programs on two cluster sizes ie. one with 5 machines and other with 10 machines. In case of RDD with larger machine size, we are achieving speedup but the same is not always true for the dataset.

	M4.xlarge 5 nodes	M4.xlarge 10 nodes
Triangle Count Hash and Shuffle - RDD(30K)	25 Minutes	22 Minutes
Triangle Count Hash and Shuffle - Dataset	20 Minutes	15 Minutes
Triangle Count Partition and Broadcast - Dataset	15 Minutes	16 Minutes
Triangle Count Partition & Broadcast-Dataset No Threshold	11 Minutes	13 Minutes
Triangle Count Partition and Broadcast - Dataset -RDD	13 Minutes	12 Minutes

Scalability

On changing the input size running time is increasing as evident from the below tables

Triangle Count Hash and Shuffle - Dataset

	Large Data (70K)	Small Data (30K)
M4.xlarge 5 nodes	20 Minutes	15 Minutes
M4.xlarge 10 nodes	15 Minutes	13 Minutes

Triangle Count Partition and Broadcast - Dataset

	Large Data (70K)	Small Data (30K)
M4.xlarge 5 nodes	15 Minutes	11 Minutes
M4.xlarge 10 nodes	16 Minutes	12 Minutes

Triangle Count Hash and Shuffle - RDD

	Large Data (30K)	Small Data (10K)
M4.xlarge 5 nodes	25 Minutes	13 Minutes
M4.xlarge 10 nodes	22 Minutes	12 Minutes

Triangle Count Partition and Broadcast - RDD

	Large Data (10K)	Small Data (5K)
M4.xlarge 5 nodes	13 Minutes	2 Minutes
M4.xlarge 10 nodes	12 Minutes	3 Minutes

When we ran the partition and Broadcast for max value of 20K, it was taking more than 90 minutes, So we ran it for smaller max values.

Results:

The final result is the number of unique social triangles on twitter dataset

Triangle Count Hash and Shuffle - RDD(30K) Triangle count: 3087524

Triangle Count Hash and Shuffle - Dataset(75 K): Triangle count 102580605

Triangle Count Partition and Broadcast - Dataset (75k): Triangle count 11397845

Triangle Count Partition and Broadcast - Dataset - No Threshold (75k): Triangle count 11397845

Conclusion:

Using the different algorithms like breadth-first search in parallel we calculated the number of unique social triangles for Twitter followers. We can further use different algorithms to obtain complex relationships between Twitter followers and can obtain some interesting patterns.

Task 2: Full Page Rank

Overview:

We ran Full PageRank on the input to find out the 100 most popular users(whoever has the highest PageRank) Since the user with the highest PageRank will have the most inlinks, it means that he/she is the most popular or the easiest to find in the network.

Pseudo Code:

```
//Reading the input
 val inputFile = sc.textFile(args(0))

  //Pre-processing the input
  val inputData = inputFile.map(line => {
     (line.split(",")(0), line.split(",")(1))})

//collect 'from' nodes in one set and 'to' nodes in one set
val toList = inputData.map(x => x._1)
val fromList = inputData.map(x => x._2)
//nodes that don't have any incoming edges with pr 0
val sourceNodesPR = toList.subtract(fromList).map(ele => (ele, 0.0)).distinct()
```

```
//Dangling nodes with empty List as their adjacency lists
val danglingNodes = fromList.subtract(toList).map(node => (node, List[String]())).distinct()
 //Each node with its adjacency list
val graph = inputData.groupByKey().mapValues(_.toList)
 //creating rdd for all the nodes(including dangling nodes) with their adjacency lists
 val adjacencyListData = graph.union(danglingNodes)
    adjacencyListData.persist()

//count the number of vertices
val V = adjacencyListData.count()

//ranks of each node
 var ranks = adjacencyListData.map(x => (x._1, 1.0/V))
 val iters = 10
 for(i <- 1 to iters){

//node, (adjacency list, rank) e.g => (9, (List(8), 0.11))
val joinRDD: RDD[(String, (List[String], Double))] = adjacencyListData.join(ranks)
joinRDD.persist()

//ranks of the node, without the adjacency list, e.g (9, 0.11)
ranks = joinRDD.map(data => (data._1, data._2._2))

//delta -- the sum of the pr of all the nodes whose adjacency list are empty -- dangling nodes
val delta = joinRDD.filter(x => x._2._1.isEmpty).map(x => x._2._2).sum()

//calculate the outgoing pr
val outGoingContributions: RDD[(String, Double)] = joinRDD.flatMap({
            case (nodeId, data) =>
           //if adjacency list is not empty
           if(data._1.nonEmpty) {
           //calculate the size of the adjacency list
            val adjListSize = data._1.size

            //map each pr to new pr
  data._1.map(el => (el, data._2/adjListSize))
  } else {
 List((nodeId, data._2)) }})
val newPR = outGoingContributions.reduceByKey(_+_)
 ranks = newPR.union(sourceNodesPR)
 ranks.persist()}
 ranks.saveAsTextFile(args(1))
 println(ranks.values.sum())}
```

Algorithm and Program Analysis:

The project uses Random Surfer Model for Pagerank algorithm. This model considers the Twitter network to have a fixed set of users, with each user containing a fixed set of links, and each link is a reference to some other user. The project studies a random surfer in the network.

Program Analysis: The program was optimized by removing any operations like collect() which collects all data to the driver; this helped efficiency a lot(5 minutes for 1 iteration, 20 mins for 10 iters). This must be because Spark manages shuffling a lot better by loading data into memory, the power and fault tolerance of RDDs, and persist option.

Experiments :

The program was run for 15 iterations on AWS, this caused to the memory limits to exceed (5 GB). m4.large instance types were used for this task; using m4.xlarge instance type increases the speed. This gave the best performance in terms of efficiency, speedup and space.

	M4.xlarge 5 nodes	M4.xlarge 10 nodes
Full pagerank 15 iterations	26 minutes	14 minutes
Full pagerank 20 iterations	57 minutes	29 minutes

Speedup: Speedup was ~ 2 as shown below.

	M4.large 5 nodes	M4.large 10 nodes
Full pagerank 10 iterations	41 minutes	20 minutes

	M4.xlarge 5 nodes	M4.xlarge 10 nodes
Full pagerank 15 iterations	26 minutes	14 minutes

Scalability: On changing the input size running time is increasing as evident from table:

	Small Data 40M	Big data 85M
M4.large 5 nodes	14 minutes	41 minutes
M4.large 10 nodes	9 minutes	23 minutes

Result Samples: UserID, pagerank

(9771624,1.021480034480823E-7)
(10177310,1.17812252122532E-7)
(9442845,1.1349689892343565E-7)
(9580455,1.031840242040711E-7)
(9076248,1.2942380443969174E-7)
(9760635,3.730382168356753E-7)
(10015445,1.015942559949559E-7)
(743864,3.96903514405714E-8)
(10245620,9.722536210727777E-8)
(2827182,2.7146617678012E-9)

(7095951,7.754529470858491E-9)
(8396613,3.3124430293652727E-10)

Conclusion: The algorithm and the program worked well for the input data. The challenges of convergence for pagerank was met by running the program for finite number of iterations. A possible extension for this scala program will be to run until convergence(by checking the ranks don't change in further iterations).

Task 3: Classification and prediction in Spark MLlib

Overview: In this task, we are predicting the fare amount for a taxi ride in New York City with RandomForestRegressor algorithm using Apache Spark MLlib library.

Pseudo Code:

```
//Defining columns that will be used as features
val featureCols=Array ("pickup_longitude", "pickup_latitude", "dropoff_longitude", "dropoff_latitude",
"passenger_count")
//Vectorize the features for training the model
val assembler = new VectorAssembler().setInputCols(featureCols).setOutputCol("features")
val df2 = assembler.transform(finalDf)
val Array(trainingData, testData) = df2.randomSplit(Array(0.8, 0.2))
//Fitting Random Forest model on the training data
val rf = new RandomForestRegressor().setLabelCol("fare_amount").
.setMaxDepth(30)
setFeaturesCol("features")
.fit(trainingData)
//Predicting values
val prediction = rf.transform(testData)
val evaluator = new RegressionEvaluator() .setLabelCol("fare_amount")
   .setPredictionCol("fare_amount").setMetricName("rmse")
val rmse = evaluator.evaluate(prediction)
```

Algorithm and Program Analysis:

Features used to train the model:
"pickup_longitude","pickup_latitude","dropoff_longitude","dropoff_latitude","passenger_count"

After reading the data from the input file we filtered out the rows with empty values and also filtered out the fares with values < $2.5 and rest of the fields shouldn't be zero. In this task, we are solving prediction problem so we are using the regression model.
Initially, on the local machine, we tried with the Linear Regression and the results were pretty bad.Then we tried with Random Forest where we took advantage of parallel processing since we can generate base learners in parallel. The basic motivation of parallel methods is to exploit independence between the base learners since the error can be reduced dramatically by averaging. Each worker gets a split of the data file and works on a set of data to create label columns.

RandomForestRegressor fits a number of classifying decision trees on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement.

Experiments: With default parameters, this algorithm gives nice results. We experimented with the max-depth parameter.As we increase the max-depth of the random forest prediction results improved.
We tried with max-depth value 20 and 30.

Max-depth: 30 accuracy: 9.345
Max-depth 2- accuracy: 10.212

Speedup:

	M4.large 5 nodes	M4.large x10 nodes
50 million records	150 minutes	83 minutes

Scalability:

	M4.large 5 nodes	M4.large x10 nodes
50 million records	150 minutes	83 minutes
25 million records	80 minutes	40 minutes

Result Samples:

```
+-----------+---------------+--------------+----------------+---------------+--------------+-------------------+----
|fare_amount|pickup_longitude|pickup_latitude|dropoff_longitude|dropoff_latitude|passenger_count|
features|       prediction|
+-----------+---------------+--------------+----------------+---------------+--------------+-------------------+----
|        2.51|       -73.9597|     40.767033|       -73.99776|      40.729237|           1.0|[-73.959701538085...|
9.166306460861161|
|        2.54|     -73.962074|      40.76364|       -73.94599|      40.824646|
1.0|[-73.962074279785...|11.435792710225645|
|         2.9|      -74.24305|     40.732487|      -74.243034|       40.73249|
1.0|[-74.243049621582...|10.488641973279718|
```

RMSE for 50 million records: 9.345

Conclusion: Prediction results obtained are not very promising. We can further decrease the root mean square error by constructing useful feature crosses. Along with this we can prune outliers or try using neural networks.

References:

Apache Spark 2.4.0: MLlib Guide
Kaggle New York City Taxi Fare Prediction
ASU Social Computing

Final Conclusions:

We were successful in achieving the goals of this project. All three tasks were executed using Spark Scala, which allowed us to explore and learn more about it. We found unique social triangles for Twitter followers in task 1; in task 2, we found top 100 influential Twitter users with highest pagerank. In the third task we predicted the taxi fare for yellow can in NYC; this allowed us to explore Spark MLlib.

Logs

Although log size is not huge it is easy to share and keep track so we checked logs in our Github.
https://github.ccs.neu.edu/cs6240f18/MR001/tree/master/Logs

Output:

For Task 1 & Task 3, the output are at :
https://github.ccs.neu.edu/cs6240f18/MR001/tree/master/Logs
The output files are denoted by stdout.

For Task 2 , the output is at :
https://github.ccs.neu.edu/cs6240f18/MR001/tree/master/Task2-Output

Github

https://github.ccs.neu.edu/cs6240f18/MR001