Project Report

Team Members:

Ronit Sharma,

Utkarsh Pujari

GitHub Link: - https://github.ccs.neu.edu/utkarshpujari/SparkProject

https://github.ccs.neu.edu/ronnie294/Project-SSSP

Project Overview:

Our motivation for this project has been to learn how a machine learning model can be applied to data in a parallel environment. In the first task we have tried to run linear regression parallelly to predict house prices. We are working on data with around 80 features. We have implemented the Linear Regression and Random Forest and calculating accuracy on training data.

Input Data:

We are working with housing data, that has 80 features that describes the house, like LotArea, YearBuilt, etc., which we will use to predict the selling price of the house. We took the data from a Kaggle competition and it is already divided into training and testing data. The training data has 1461 rows which was multiplied enough times so that it is suitable for parallel execution. Task 1 is to run prediction on this dataset using Linear Regression and Random Forest.

Overview

This task is focused on building a Machine Learning model (Linear Regression) in a parallel computing environment and predicting house process from that model.

Pseudo Code

- val inputDataRDDTest = test.csv
- val inputDataRDDTrain = train.csv
- val trainMap= inputDataRDDTrain.map(s=>parseRecordObject(s))
- val testMap= inputDataRDDTest.map(s=>parseRecordObject(s))
- val testDF = testMap.toDF
- val trainDF = trainMap.toDF
- val assembler = new Assembler().setInputCols(“Selected Features”).setOutputCols(features)
- val testDF2=assembler.transform(testDF1)
- val trainDF2=assembler.transform(testDF2)
- val lr = new LinearRegression()
- lr.fit(testDF2)
- lr.predict(trainDF2)

- val regression = new RandomForestRegressor().setImpurity("variance")
  .setMaxDepth(10).setNumTrees(100).setFeatureSubsetStrategy("auto")
  .setSeed(5043)
- val model = regression.fit(trainingData)
- val predictions = model.transform(ecommDF2)

Algorithm and Program Analysis

We have used Linear Regression model and Random Forest. Out of 80 features we are using about 36 of those which are numeric and ran our prediction on test data. Few 'NA' values were converted to 0. We have converted non-numeric features into classes. We then convert the input to objects of records and then convert that to dataframes. Then we extract features that will be used for training and then apply models based on it to produce Root Mean Square Errors using Linear Regression and Random Forest Regression.

Experiments

To start with we have trained our model based on all numeric features which are about 36 features on training data. There were many numeric columns with 'NA' as values, which for now we are converting into a 0. We have submitted our initial prediction at Kaggle, with a Root Mean Squared Logarithmic error as 0.44.

We started off with Linear regression on 1461 rows which ran for 6 seconds, later we multiplied the same data manifolds to make this a big data problem. We ran on about 600k records which took roughly 10 mins on aws. We also tried to change the hyperparameters like max depth of the tree, and number of trees. In general, as the number of trees increases, the execution time also increases and the accuracy increases as well.

No. of rows	Machine	#of Trees	Max depth	Execution Time	Accuracy(RMSE)
300k	5	500	10	10m	6380
600k	5	100	10	13m	6326
600k	5	500	10	16m	6394
600k	10	500	10	12m	6394

Speedup

As much of the algorithm for machine learning is implemented in the MlLib api, which provides a good speedup.

No. Of rows	Machine	Execution
600k	5	16m
600k	10	12m

Scaleup

We started out with small amount of data (1461 rows) which finished in seconds. As we multiplied the data the execution time did not change proportionally. For example, when we duplicated the data 400 times the time only increased by 10 times. So, it has a good scale up.

Results

Predictions are sorted by id. These are the predictions.

[119983.98489053274]

[156626.08678829944]

[174710.8999169882]

[201434.72255561638]

[196320.55155298635]

[182283.4858034216]

[198298.52568115888]

[169742.7563206591]

[208876.54115646138]

[115883.96844387526]

[203689.49634292786]

[99331.08854661725]

[76336.97871382482]

We are not storing the predictions as output this time, rather we are appending the log with below information.

```
No of trees in Random Forest: 500
Max depth for Random Forest: 10
Random Forest RMSE:  6409.748629676631
Linear Regression RMSE: 34088.8516180331
LinearMSE: 1.162049804636278E9
r2: 0.8163538147327166
```

Task2- Single Source Shortest Path in Spark

Single source shortest path is a widely used algorithm. It can be used to solve a wide gamut of problems like finding the shortest route from a starting point. It can also be used to solve many social network analysis problems. Our motivation for this task has been to learn how we can run such a widely used algorithm in parallel.

Input Data:

We are working with the twitter dataset that we have widely used, only one modification has been made, that we add a field to indicate the source vertex in the edges.csv file, for example- (1,S) would mean that we are using the first node as the source.

Overview

This task is focused on finding Single Source Shortest Path on twitter dataset.

Psuedo Code: Task2

- val threshold=1000000
- val textFile = sc.textFile(args(0))
- val rawD = textFile.map(word => (word.split(",")(0),(1.0,word.split(",")(1)))).
  filter(s=>(Integer.valueOf(s._1)<=threshold)&&(if (s._2._2.equals("S")) {true} else
  Integer.valueOf(s._2._2)<=threshold))
- val graph=rawD.groupByKey().mapValues(_.toList)
- graph.persist()
- var distances = graph.mapValues(s=> if (s.contains((1.0,"S"))) 0.0 else Double.PositiveInfinity)
- val accum=sc.doubleAccumulator
- while (accum.isZero){
- val temp=distances
- distances =
  graph.join(distances).flatMap(s=>helper(s._2._1,s._2._2,s._1)).reduceByKey((x,y)=>min(x,y))
- var count = temp.subtract(distances).count()
- if (count==0){
- accum.add(1.0)
- }}
- distances.saveAsTextFile(args(1))}
- Helper function to emit distances
- def helper(values:List[(Double,String)],distance:Double,id:String) :List[(String,Double)] ={
- val list= values.map(s=>(s._2,s._1+distance))
- val toReturn =(id,distance)::list
- toReturn}

Algorithm and Program Analysis

The program starts by reading the input and creating the adjacency list which serves as the graph, we persist the graph to restrict the network flow and use mapValues to avoid shuffling. We have used an accumulator to detect the convergence of the algorithm. The program exhibits good speedup and scaleup as shown below. The biggest challenge for us was to detect when to stop the iterations. We ran into a few problems initially which are mentioned below.

Experiments

Initially while testing, we were testing for convergence by storing the previous distance as a list using the collect method, which makes a list. This approach was causing our program to take a humongous amount of time. We solved this by not converting to a list and instead using the rdd.subtract method. We subtract the current distance rdd from the previous one and if the count of the records is zero, it means that we have reached our solution. We used m4.large machines and the size of our cluster was 5(1 Master, 4 Workers) for the small cluster, and 10(1 Master, 9 Workers) for the large cluster. In the Dataset, we used k = 100000. We also ran a case where we run SSSP on the entire dataset, which is documented below.

Speedup

Our program achieved good speedup, as a result of the decreased network traffic, and reduced shuffling.

Run Data

Configuration	Small Cluster Result	Large Cluster Result
SSSP in SPARK K=100000	Time: 6 Minutes	Time: 4 Minutes
SSSP in SPARK (Whole Dataset)	N/A	Time: 15 Minutes (M5.xlarge)

The reason for good speedup is that as we increase the number of workers available, work among each iteration can be distributed even further, as work within each iteration does not have any dependency and thus is inherently parallelizable. Also because of the graph is persisted, we don't increase network traffic while adding new nodes.

Scaleup

Our program showed good scaleup, initially we were running for k=5000, our program took less than a minute, after raising the k to 70000, the program took only 2 minutes. Even though we had increased k by a factor of more than 10, the time only doubled. When from k =70000 we raised it to 100000 the time only tripled. On running it for the entire dataset, we encountered out of memory problem, as the graph was so big that it could not be persisted in memory. To overcome this we ran the entire dataset on a large cluster using m5.large machines, which overcomes the memory problem. The results are reported.

Results(Partial)

Task 2:

(557825,3.0)
(706775,Infinity)
(573151,3.0)
(270715,Infinity)
(488543,Infinity)
(584140,3.0)
(510826,Infinity)
(766814,Infinity)
(253895,Infinity)

(606811,3.0)
(553928,3.0)
(462712,3.0)
(10117,3.0)
(119687,3.0)
(527413,3.0)
(48657,3.0)
(881078,Infinity)
(899510,3.0)
(301440,3.0)
(137731,Infinity)
(695237,Infinity)
(670018,3.0)
(462271,Infinity)
(770260,Infinity)
(648383,4.0)
(213844,3.0)
(234832,3.0)
(967343,4.0)
(932422,3.0)

Conclusion

Our main learning from the both tasks, and probably from the entire course has been how to distribute tasks over different machines efficiently. Coming to the details, from the Single Source Shortest Path task, our main learning was how to reduce shuffling in a task by using mapValues instead of map and using the persist option to reduce network traffic. In the future we could work on solving real world problems like traffic navigation using this algorithm as a starting point.
Machine Learning in Spark allows parallel model computation using ensemble techniques. This not only increases the accuracy but also parallelism by running multiple models concurrently. MLlib provides great tools to run your models so that users don't have to worry much. Learning how to run machine learning models in a distributed manner is a great learning that can have great implications for the future, as we can make better and bigger predictions.