Big Data Apache Spark Multivariate Regression

Gakuo Patrick

January 29, 2019

1 Introduction

In this project we demonstrate the use of Apache Scala Spark in a multivariate regression analysis of tweets. We will analyze features of a tweet that determine the number of likes the tweet is likely to get. To this end, we use a tweets dataset which we divide into a training set and a test set. The dependent variable is the number of likes per tweet. The independent variables include the length of the text, the number of hashtags, the number of followers the user has, the number of accumulated likes by the user, the number of friends, the user's listed count, and user statuses count. In this project we use Scala Spark to implement the required three steps. These steps include parsing each tweet into a Tweet case class, standardizing each feature of the tweet (independent and dependent variables), and finally calculating the regression parameters (denoted as Theta). We have two implementations of this project, one using RDDs and the other using Datasets.

2 Parsing Tweets

In the RDD implementation, we read in the tweets inside our main function as an RDD and map our tweet parsing function over each RDD line. This transforms each RDD row into a Tweet object with eight fields. The Tweet case class represents the seven independent and the dependent variable(likes). For the Dataset implementation, we read the tweets and parse them just like for the RDD implementation. However, we use the method $toDS()$ **to transform the RDD into a dataset.**

In the $parseTweet(tweet : String)$**, we use the json4s library to parse from a JSON string into a Tweet case class. To get interesting tweets with likes, we look for embedded tweets with the fields "retweeted status" and "quoted status". The other tweets were freshly tweeted**

and were likely going to bias the analysis due to their lack of likes. We give such tweets dummy values and then filter them out.

3 Standardizing the features

Once we parse in a tweet, we have to standardize it for efficient multivariate regression analysis. This is a process of obtaining the Z-score for each observed feature. This required us to calculate the mean and standard deviation for each tweet feature. We use $calculateFeatureMean(rdd : RDD[Tuple], sampleSize : Long) : Tuple$ **and** $calculateFeatureSTD(rdd : RDD[Tuple], means : Tuple, sampleSize : Long) : Tuple$ **to calculate the mean and standard deviation respectively. They both return a Tuple containing the mean and standard deviation respectively for each feature. The standard deviation calculation takes the results of the mean calculation as one of its input.**

We then use the mean and standard deviation Tuples to standardize the features in the function $standardizeFeature(rdd : RDD[Tuple], means : Tuple, std : Tuple) : RDD[(Tuple)]$**. This function computes the Z-score for each observed feature. It produces an RDD/dataset whose every row is a Tuple.**

4 Calculating the gradient descent

To calculate the gradient descent we used the function $gradientDescent(rdd : RDD[Tuple], sampleSize : Long) : Theta$ shown in $codelisting - 1$. This function has a couple of helper functions namely:

I genRandom(): Double - generates a random number between -2 and 2 (Arca, 2017).

II $expandEntries(rdd : RDD[Tuple], theta : Theta) : RDD[ExtendedTuple]$ - it takes a theta and calculates the error for each tweet (predicted likes - actual likes). It returns an ExtendedTuple containing the error and all the independent features and an extra field containing just the value 1. See $codelisting - 2$.

III cost(rdd:RDD[ExtendedTuple],sampleSize:Long):Double - Helps calculate the error of the multivariate regression. This is our cost function. It helps us determine when the predicted parameters are close enough to predicting the actual observed likes value. See $codelisting - 3$.

IV updateTheta(theta:Theta,rdd:RDD[ExtendedTuple],alpha:Double, countExamples:Long): Theta - This function updates the theta after each iteration of the gradient descent. See $codelisting - 4$

```scala
 def gradientDescent(rdd:RDD[Tuple],sampleSize:Long): Theta ={
   val alpha = 0.001 //learning rate
   val sigma = 0.001
   var theta = Theta(genRandom(),genRandom(),genRandom(),genRandom()
   ,genRandom(),genRandom(),genRandom(),genRandom())
   //initial guessed coefficients
   var rddWithErrorEntries = expandEntries(rdd,theta).persist()
   var error = cost(rddWithErrorEntries,sampleSize)
   var delta: Double =0
   do{
     theta= updateTheta(theta,rddWithErrorEntries,alpha,sampleSize)
     //update coefficients
     rddWithErrorEntries = expandEntries(rdd,theta).persist()
     val cost1 = cost(rddWithErrorEntries,sampleSize)
     delta = error - cost1
     error = cost1 //update error for the next round

   }while(delta>sigma)
   theta
 }
```

Code Listing-1: Gradient Descent function

```scala
 def expandEntries(rdd:RDD[Tuple],theta:Theta):RDD[ExtendedTuple]={
   //calculate predicted value and calculate error
   rdd.map(elem=>ExtendedTuple(((theta.errCoeff+elem.text*theta.txtCoef + elem.tags *theta.tagCoeff + elem.followers *theta.follCoef  +elem.friends *theta.frndCoeff +elem.favorites *theta.favCoeff + elem.statuses *theta.statusCoef +elem.listed *theta.listedCoef) - elem.likes),1,elem.text,elem.tags,elem.followers,elem.friends,elem.favorites,elem.statuses,elem.listed))
 }
```

Code Listing-2: Expand Entries function

```scala
def cost(rdd:RDD[ExtendedTuple],sampleSize:Long):Double={
   //for each row, square error then sum all entries and divide by 2 * sampleSize
   val accumulatedError=rdd.map(elem=>(math.pow(elem.error,2 ))).fold(0)((acc,elem)=>acc+elem)
   val cost =((1.toDouble/(2*sampleSize))*accumulatedError)
   cost
 }
```

Code Listing-3: Cost function

```scala
def updateTheta(theta:Theta,rdd:RDD[ExtendedTuple],alpha:Double,countExamples:Long): Theta={
    val factor = alpha*(1.toDouble/countExamples)
    //calculate coefficients deltas
    val intermediateDifferences= rdd.map(elem=>(elem.error*elem.errCoeff, elem.error*elem.text, elem.error*elem.tags, elem.error*elem.followers, elem.error*elem.friends, elem.error*elem.favorites, elem.error*elem.statuses,elem.error*elem.listed))
    .fold((0,0,0,0,0,0,0,0))((acc,elem) => (acc._1+elem._1,acc._2
        +elem._2,acc._3+elem._3,acc._4+elem._4,acc._5
    +elem._5,acc._6+elem._6,acc._7+elem._7,acc._8+elem._8))
   //update coefficients
   Theta(theta.errCoeff - intermediateDifferences._1 *factor,theta
       .txtCoef - intermediateDifferences._2 *factor,theta.
       tagCoeff -intermediateDifferences._3 *factor,theta.follCoef
        -intermediateDifferences._4 *factor,theta.frndCoeff -
       intermediateDifferences._5*factor,theta.favCoeff -
       intermediateDifferences._6 *factor,theta.statusCoef -
       intermediateDifferences._7 *factor,theta.listedCoef -
       intermediateDifferences._8 *factor)


 }
```

Code Listing-4: Update Theta function

The gradient descent algorithm starts with guessing a random Theta with each parameter falling between -2 and 2. We then pass the standardized RDD and the guessed Theta to the *expandEntries* function which is described in (II) above. From this expanded RDD, we can determine the cost using the function *cost* described in (III) above. We then get into the while loop where we check whether the difference between the previous error and the current error is below the maximum error allowed. Each iteration first updates the Theta, then expands the standardized RDD with the updated Theta. We then use the expanded RDD to calculate the current cost which is subtracted from the cost of the previous iteration. We only break out of the while loop if the error difference is below a predetermined amount (*sigma*). We then output the current Theta as the result of the multivariate regression.

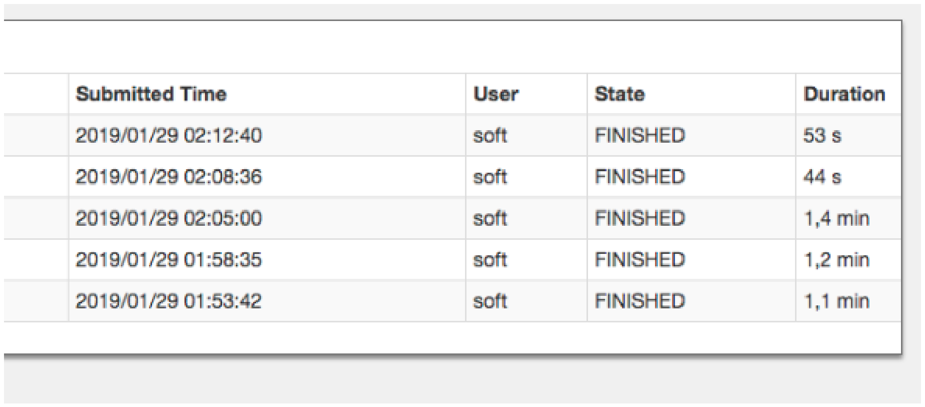
5 ANSWERING QUESTIONS

5.0.1 Have you persisted some of your intermediate results? Can you think of why persisting your data in memory may be helpful for this algorithm?

We persisted every RDD or Dataset that was to be acted upon by more than one action. Transformations used here include filter and map. These are lazy but calling actions such as count and fold which are eager causes the RDD/Dataset to be computed. To avoid multiple computations we persist each RDD/Dataset which actions will be require more than once. This is especially so in the gradient descent loop where many re-computations may result into slow processing.

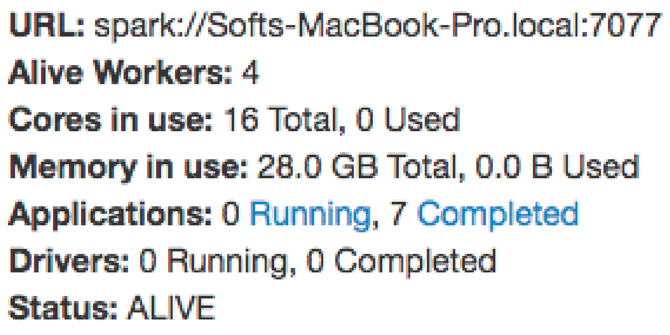
5.0.2 In which parts of your implementation have you used partitioning?

We used the partitioning provided by "sc.textFile(path:String,numPartions:Int) .map(parseTweet).persist()". We tried various numbers of partitions and got results shown in the screen-shot below: From top to bottom 1,2,4,8,and 16 partitions. We wanted to input data to be read in parallel and the partitions persisted. These results were obtained from a corei7 macBook Pro with 4 cores, and 8 GB RAM. We set up a standalone Apache Spark with 4 workers each having 7 GB of memory and 4 cores. It appears that partitioning in this case causes shuffling which reduces performance. It is recommended to create partitions equal to 2 or 3 times the number of available cores partition. The benefits of this partitioning would have been evident with a large dataset.



Submitted Time	User	State	Duration
2019/01/29 02:12:40	soft	FINISHED	53 s
2019/01/29 02:08:36	soft	FINISHED	44 s
2019/01/29 02:05:00	soft	FINISHED	1,4 min
2019/01/29 01:58:35	soft	FINISHED	1,2 min
2019/01/29 01:53:42	soft	FINISHED	1,1 min

Image 1: results of using partitioning- From top to bottom 1,2,4,8,and 16 partitions



URL: spark://Softs-MacBook-Pro.local:7077
Alive Workers: 4
Cores in use: 16 Total, 0 Used
Memory in use: 28.0 GB Total, 0.0 B Used
Applications: 0 Running, 7 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Image 2: The Standalone Apache Spark Setup

5.0.3 Did you use RDDs, DataFrames or Datasets for your implementation?Why?

We implemented this project using both RDDs and Datasets. RDDs allow for the use of unstructured data giving us more freedom to operate on our data with functional programming as opposed to domain specific expressions (Damji, 2016). On the other hand, Dataframes and Datasets impose a schema on the data and are built on top of the Spark SQL engine (Damji, 2016). They offer high level abstractions which although are easy to use, they do not give as much freedom as RDDs do. However, Datasets and Dataframes offer optimization to queries which make them much faster than RDDs (Damji, 2016). We chose Datasets because they additionally impose strong-typing which ensures runtime type-safety (Damji, 2016). The dataset implementation ran locally but it failed to run on Isabelle and on the standalone setup. We suspect the definition of the spark session outside the main as the cause of errors. Moving the spark session inside the main meant that we did not have the spark-session implicits required by the functions outside main. Thus we could not run our Dataset implementation.

6 Results

We divided our dataset into a training and a test set. The training set resulted into the followoing Theta value as one of its results: (NoiseValue: 0.3798297578049597, LengthofTExt: 0.7181256623765928 , Tags: -0.1129920131678076 , Followers: 2.4219107633096986 , Friends: 0.22972193416344092 ,TotalUserLikes: 0.2696693537870232 , Total Statuses: 0.014312531466087514 , ListedCount: -2.3556630472098212). We then used this theta on the test set and calculated its cost. This gave us a low cost of 1.0756589044064777.

References

Arca, V. (2017, February). Linear regression 4 – learning rate and initial weights. *http://www.vitaarca.net/linear-regression-4-learning-rate-and-initial-weights/ Linear Regression 4 – Learning Rate and Initial Weights.*

Damji, J. (2016, July). A tale of three apache spark apis: Rdds vs dataframes and datasets when to use them and why. *https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html.*