12/29/2016

Parallelization of KNN Algorithm

K Nearest Neighbour Classification

in APACHE PYSPARK using files saved on

HDFS (Hadoop Distributed File System)

Hira Fatima | Graduate Student
DS8001 | Master of Data Science and Analytics | RYERSON UNIVERSITY

Table of Contents

INTRODUCTION

This report analyzes K Nearest Neighbour algorithm run in a parallel setting for big data computing. To parallelize the algorithm, Apache PYSPARK was used with test and training data files saved on HDFS (Hadoop Distributed File System). Euclidean distance is used to find the nearest neighbours. KNN inherently requires intensive computation. Its parallelization can be very beneficial as it is one of the simplest, most commonly used classification algorithm.

SPARK's MLlib (Machine Learning Library) includes a number of algorithms that can be used for Machine Learning analysis, however, it does not have anything for K-Nearest Neighbour.[1] My project here is important because my code allows for KNN classification on big data files saved on distributed file systems. My algorithm can work with any number of features and class labels as long as the features data is numeric so that the distances can be computed. If some features are categorical, then these can be factorized to be used with this algorithm.

CLASSIC K-NN ALGORITHM

Per Wikipedia, "In *k-NN classification*, the output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its *k* nearest neighbors (*k* is a positive integer, typically small). If *k* = 1, then the object is simply assigned to the class of that single nearest neighbor…. *k*-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification.[2]

As defined above, since k-NN is instance based learning, the need for computation arises only when a new point needs to be classified. When a point needs to be classified, its distance is calculated against all other points in the training set. If the number of k=3, then the 3 nearest points will be shortlisted and their majority class vote will be used to label the new point.

Pseudo Code:

- Points in dataset A = {1, 2, 3} are new points that needs to be classified based on K=3
- Points in dataset B = {1,2,3,4,5,6,7,8,9} are provided with labels that will be used to classify dataset A
- Compute the distance (Euclidean) between each point in set A with set B
- For each point in dataset A, pick 3 points from dataset B with minimum distance
- Use the majority vote from those 3 points to classify each point in dataset A

GOAL

K-NN algorithm can be run on datasets using many languages such as R. However, when run in its simplified form, the calculations are done sequentially and would take much longer. For instance, using the previous pseudo code example, the distance of point 1 from set A will first be calculated against all points in set B one by one, then the distance of 2 will be calculated against all points in set B and so on and so forth.

The goal of my algorithm is to find out the classification for test data points using their K nearest neighbours' classification (majority vote) with a parallelized approach. SPARK seemed to be the best option as it "provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance."[3]

The data files were saved on HDFS to mimic the situation of working with big data saved on clusters. Spark and HDFS together maximize data locality to increase parallelism allowing for the calculations to be performed in parallel as opposed to sequentially.

[1] https://spark.apache.org/mllib/
[2] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm
[3] https://en.wikipedia.org/wiki/Apache_Spark

RELATED LITERATURE

KNN algorithm requires a join between both, the test and training dataset points followed by calculating all the distances and then sorting to find the least k distances. This is an expensive operation that needs to be performed as there is no heuristic approach that can be applied in finding K nearest neighbours. As data is becoming big data, there is a dire need to find efficient solutions for this problem. Currently there are some articles that explain efficient ways of performing KNN algorithm on vast amounts of data.

Zhang et al in their article called "Efficient parallel kNN joins for large data in MapReduce" explain ways in which KNN can be applied efficiently using MapReduce (Hadoop) technology on “tens or hundreds of millions of records” in both the test and training datasets with up to “30 dimensions”.[4]

Another article called “A CUDA-based parallel implementation of K-nearest neighbor algorithm.” discusses how the authors were able to utilize “CUDA multi-thread model” along with “Graphics Processing Units (GPUs)” to compute data elements in parallel. Per the article, “CUKNN outperforms the serial KNN on an HP xw8600 workstation significantly, achieving up to 46.71X speedup including I/O time. It also shows good scalability when varying the dimension of the reference dataset, the number of records in the reference dataset, and the number of records in the query dataset.”[5] A similar method was also tested in another article called “A practical GPU based kNN algorithm.” where the authors were able to obtain high increase in performance compared to ordinary CPU version while capitalizing on “recent developments in programmable, highly paralleled Graphics Processing Units (GPU)” that “have opened a new era of parallel computing which deliver tremendous computational horsepower in a single chip.”[6]

Other than the first article mentioned here which used Map Reduce (Hadoop), most of the other authors used GPU technology to process KNN in parallel. I am glad to have tried KNN in Apache Spark and see that it works seamlessly using RDD data structure.

[4] Zhang, Chi, Feifei Li, and Jeffrey Jestes. "Efficient parallel kNN joins for large data in MapReduce." Proceedings of the 15th International Conference on Extending Database Technology. ACM, 2012.
[5] Liang, Shenshen, et al. "A CUDA-based parallel implementation of K-nearest neighbor algorithm." Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC'09. International Conference on. IEEE, 2009.
[6] Kuang, Quansheng, and Lei Zhao. "A practical GPU based kNN algorithm." International symposium on computer science and computational technology (ISCSCT). 2009.

METHOD

The test and training datasets are saved on HDFS and read into SPARK (pyspark) as RDDs. Transformation and action functions for paired RDDs are then used to manipulate these RDDs to get the classification based on k nearest neighbours. SPARK manages most of the parallelization under the hood.

- Training Datasets
- Test Datasets

Classified Test Points

WHY SPARK & HDFS?

Christo Wilson, on this website explains in very simple words why HDFS and SPARK together can be a power combination to compute large amounts of data in parallel with very simple code rather quickly.[7]

From working with different programs in various assignments in other courses, it was obvious that SPARK was easy to use, required minimal code lines and the process of transforming RDDs was quite simple, smooth and fast. There was no need to define schema of tables like in PIG and HIVE, and, also no need to write streaming commands like in MAPREDUCE. These advantages allowed to make the code work for any number of rows/records and features. I could paste the entire code in PYSPARK for any dataset and it would provide instant results.

ASSUMPTIONS

- The training dataset is in the following format where all the features are listed first and the last field contains the class:

  {Feature1, Feature2, Feature3, ….. FeatureD, ClassLabel}

- The test dataset contains the features in the same order as the training set without class labels:

  {Feature1, Feature2, Feature3, ….. FeatureD}

- All the features contain numeric data; any categorical attributes are factorized
- Data has been cleaned and missing/null records are taken care of

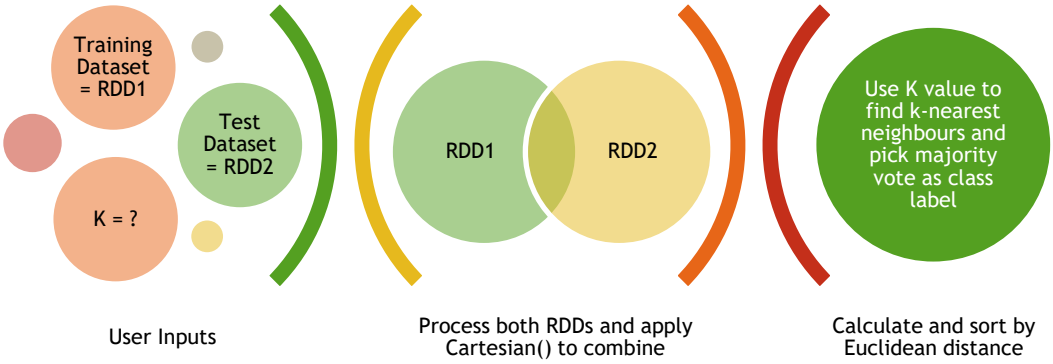
[7] http://www.ccs.neu.edu/home/cbw/spark.html

DATASET USED FOR ANALYSIS

For the purpose of testing and analysis, Iris dataset was used in this exercise and is available here. This dataset has 4 features, but the parallelized algorithm can take any number of features. This dataset has 3 classes as shown below but the algorithm can work with any number of classes.

-- Iris Setosa
-- Iris Versicolour
-- Iris Virginica

HIGH-LEVEL PSEUDO CODE



RESULTS

The results obtained via parallelized algorithm in SPARK were exactly as expected. The algorithm classified the points the same way as they were classified in R Programming hence the accuracy remains intact. The goal of parallelizing the computation was also achieved as expected. As opposed to the simple KNN algorithm run in R using KNN package which runs sequentially, my algorithm could process the data in parallel hence making the process faster.

SUMMARY

Learning how to parallelize KNN was a great exercise. It allowed me to not only learn how KNN exactly works so I could parallelize it, but it also helped in putting together concepts from different courses and programs to come up with my final product, a parallelized version of KNN.

KNN is an instance based algorithm, and to classify any point, all calculations need to be done to get the final answer – there is no heuristic approach that can be applied. SPARK+HDFS made it easier to run these calculations in parallel optimizing its use for big data (large datasets).

CODE

The code is provided below with detailed explanation of each step.

Parallelized Code (with detailed explanation/comments)

Below is the code with detailed explanation of each step along with examples of how the RDD looked at different stages during transformations.

Code	Explanation
import math	To calculate Euclidean distance
from pyspark.mllib.linalg import Vectors	To convert the tuple of features into a vector
# User-defined Information	
k=7	Input the value of K, # of neighbours
trainingset = sc.textFile('/user/root/knn/knn2/IrisDataset_Training.txt').zipWithIndex()	Define the path of training set which is imported as an RDD with Index number for each element attached
testset = sc.textFile('/user/root/knn/knn2/IrisDataset_Testing.txt').zipWithIndex()	Define the path for test set which is imported as an RDD with Index number for each element attached
# Code for Test Point Classification	
trainingseta = trainingset.map(lambda x : (x[1],x[0]))	Reverse the record so that Index # becomes the key
trainingsetb = trainingseta.mapValues(lambda x: x.split(","))	Split the fields in file
trainingsetc = trainingsetb.mapValues(lambda x: (tuple(x[0:-1]),x[-1]))	Split the values into a tuple of features and labels
testseta = testset.map(lambda x : (x[1],x[0]))	Reverse the record so that Index # becomes the key
testsetb = testseta.mapValues(lambda x: tuple(x.split(",")))	Split the fields in file
combinedRDD = trainingsetc.cartesian(testsetb)	Combine the RDDs using Cartesian function so that each element in training set is combined with each element in test set
# Example of Output: ((0, ((u'5.1', u'3.8', u'1.6', u'0.2'), u'Iris-setosa')), (0, (u'6.2', u'3.4', u'5.4', u'2.3')))	This is how the RDD looks at this point
# Example of Output: ((trainingindex, ((trainingvector), traininglabel)), (testindex, (testvector)))	This is the information contained in the RDD
step1 = combinedRDD.map(lambda ((trainingindex, ((trainingvector), traininglabel)), (testindex, (testvector))): ((testindex, trainingindex,	Convert the feature tuples into vectors and rearrange the fields in RDD such that the two vectors

trainginglabel),(Vectors.dense(trainingvector), Vectors.dense(testvector))))	become values and the remaining fields become keys
step2 = step1.mapValues(lambda (x,y) : (x-y))	On values (feature vectors), apply these functions to calculate the Euclidean distance starting with difference
step3 = step2.mapValues(lambda x: x*x)	Then square
step4 = step3.mapValues(lambda x: sum(x))	Followed by sum
step5 = step4.mapValues(lambda x: math.sqrt(x))	And then square root
# Example of Output: ((1, 7, u'Iris-versicolor'), 1.4106735979665888)	Here is an example of the RDD at this step
# Example of Output: ((testindex, trainingindex, trainglabel), distance)	And this is the data it contains
step6 = step5.map(lambda ((testindex, trainingindex, trainglabel), distance): ((testindex, trainingindex, trainglabel, distance))).sortBy((lambda x: (x[0],x[3])))	Now transform the RDD so it can be sorted by test index and distance in ascending order
step7 = step6.groupBy(lambda x: x[0]).mapValues(lambda value: tuple(value))	Group by test index so that for each test point, it groups the data into a list sorted ascending
step8 = step7.mapValues(lambda x: x[0:k])	Pick the first k values which would be the smallest distance points
step9 = step8.values()	Since the values contain all necessary information, drop the key and use values only for further processing
step10 = step9.flatMap(lambda (d): list(d))	Apply the list function to split grouped data into key/value pairs
# Example of Output: (0, 1, u'Iris-setosa', 4.7968739820845823)	This is how the RDD looks at this stage
# Example of Output: (testindex, trainingindex, trainglabel, distance)	This is the information it contains
step11 = step10.map(lambda (testindex, trainingindex, trainglabel, distance): ((testindex, trainglabel) , 1))	Now similar to the word count example, add one to value to do a count of each label
step12 = step11.reduceByKey(lambda v1,v2:v1+v2)	Reduce to find the total count
# Example of Output: ((1, u'Iris-versicolor'), 1)	This is how the RDD looks at this stage
# Example of Output: ((testindex, trainglabel), count)	This is the information it contains
step13 = step12.map(lambda ((testindex, trainglabel), count):(testindex, trainglabel, count)).sortBy((lambda x: (x[0],x[2])),ascending=False)	Now sort the data again, first by test index and then by count of labels in descending order
step14 = step13.groupBy(lambda x: x[0]).mapValues(lambda value: tuple(value)).mapValues(lambda x: x[0:1])	Pick the first record in each group

step15 = step14.values().flatMap(lambda (d): list(d)).map(lambda (testindex,testlabel,count):(testindex,testlabel))	Using only the values (as all info is in values), convert the grouped data into a list and transform the RDD to show test index and label only.
step15.collect()	Collect results

Parallelized Code (without explanation/comments)

Below is the code without extra lines for explanation/comments. Turquoise highlighted lines are where user inputs are defined. The first line is the value of k for how many neighbours need to be assessed for classification. The second and third contain file location of training and test data. The rest of the algorithm would process the points and provide class labels in step 15.

```
import math
from pyspark.mllib.linalg import Vectors
k=7
trainingset = sc.textFile('/user/root/knn/knn2/IrisDataset_Training.txt').zipWithIndex()
testset = sc.textFile('/user/root/knn/knn2/IrisDataset_Testing.txt').zipWithIndex()
trainingseta = trainingset.map(lambda x : (x[1],x[0]))
trainingsetb = trainingseta.mapValues(lambda x: x.split(","))
trainingsetc = trainingsetb.mapValues(lambda x: (tuple(x[0:-1]),x[-1]))
testseta = testset.map(lambda x : (x[1],x[0]))
testsetb = testseta.mapValues(lambda x: tuple(x.split(",")))
combinedRDD = trainingsetc.cartesian(testsetb)
step1 = combinedRDD.map(lambda ((trainingindex, ((trainingvector), traininglabel)), (testindex, (testvector))): ((testindex, trainingindex, traininglabel),(Vectors.dense(trainingvector), Vectors.dense(testvector))))
step2 = step1.mapValues(lambda (x,y) : (x-y))
step3 = step2.mapValues(lambda x: x*x)
step4 = step3.mapValues(lambda x: sum(x))
step5 = step4.mapValues(lambda x: math.sqrt(x))
step6 = step5.map(lambda ((testindex, trainingindex, traininglabel), distance): ((testindex, trainingindex, traininglabel, distance))).sortBy((lambda x: (x[0],x[3])))
step7 = step6.groupBy(lambda x: x[0]).mapValues(lambda value: tuple(value))
step8 = step7.mapValues(lambda x: x[0:k])
step9 = step8.values()
step10 = step9.flatMap(lambda (d): list(d))
step11 = step10.map(lambda (testindex, trainingindex, traininglabel, distance): ((testindex, traininglabel) , 1))
step12 = step11.reduceByKey(lambda v1,v2:v1+v2)
step13 = step12.map(lambda ((testindex, traininglabel), count):(testindex, traininglabel, count)).sortBy((lambda x: (x[0],x[2])),ascending=False)
step14 = step13.groupBy(lambda x: x[0]).mapValues(lambda value: tuple(value)).mapValues(lambda x: x[0:1])
step15 = step14.values().flatMap(lambda (d): list(d)).map(lambda (testindex,testlabel,count):(testindex,testlabel))
step15.collect()
```