BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI

FOUNDATIONS OF DATA SCIENCE ASSIGNMENT
2019-20

Project Report
on
KNN and Bisecting K-Means using Hadoop and Spark

Submitted to:
Prof. Navneet Goyal

Submitted by:
2017A7PS276P Kavya Gupta
2017A7PS0001P Bhoomi Sawant
2017A7PS0160P Aditi Mandloi

Introduction

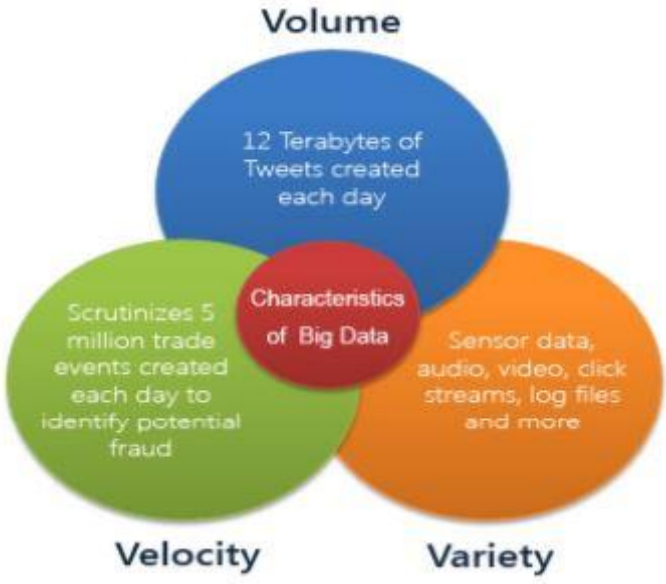
This is the era of Big Data. It is driving the way we live our everyday life. It has the power to affect our lives in positive ways and make Earth a better place to live. Due to insufficient memory and incapability of traditional data management techniques, there is a huge demand for more innovative latest technology. Hadoop and Spark are frameworks that attempts to provide solution for challenges of Big Data by providing facility to store the dataset on distributed environments and perform the computations in parallel. In this project, we have implemented KNN Classification and Bisecting K-Means Clustering Algorithms over hadoop and spark frameworks and compared their execution time. The dataset provided for KNN Classification has 10 attributes where the last one corresponds to class. Other attributes are integer values. Dataset for K-Means Clustering comprises of 13 attributes each of which is integer value. The algorithms have been implemented in Java without using MLlib and mhout. Flow charts explaining the architecture of the program have been provided. It is also followed by easy to comprehend pseudo-codes and code snippets for mapper and reducer classes in case of Hadoop.

Big Data

The term “Big Data” refers to a collection of data that is large in size and yet growing exponentially with time. It is so large and complex that it is beyond the ability of traditional data management tools to capture, store, manage and analyze it efficiently. The incapability of already existing technologies to handle big data poses a great challenge and requirement for more innovative thinking and high performance tools. Big data however, also provides opportunities in terms of what we can do with it to enrich the lives of everyone and make this Earth a better place to live. Big Data solutions should be capable for analyzing not only raw structured data but also the semi-structured and unstructured data from a wide variety of sources.

Big Data is not only defined by large **Volume** but also diverse in terms of **Variety** and **Velocity**.

Volume: It refers to the scale of the data as well as processing needs. The data scales from Terabytes to Petabytes to Zetabytes.

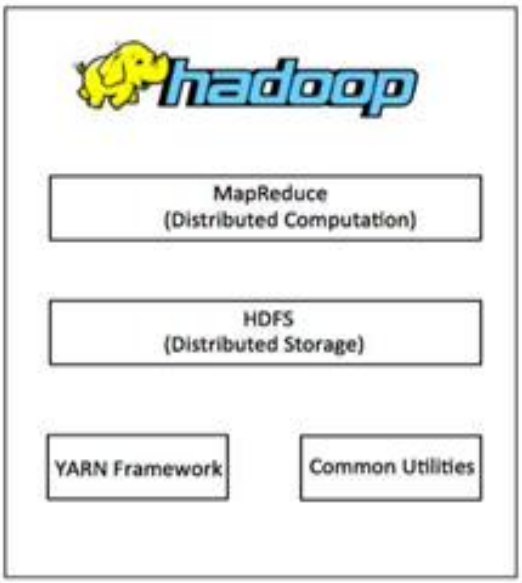


Variety: It refers to the different forms of data like images, documents, videos, streams etc. Here data is complex because of it includes not only structured traditional data, but also semi-structured and unstructured. Managing the data in different structures ranging from relational to logs to raw text etc. is a difficult task.

Velocity: It refers to the speed of the data processing. The speed of data arrival, storage and retrieval varies. The management of streaming data and large volume data movement requires more advanced tools.

Hadoop and Big Data

Big data offers great opportunity to discover non-trivial, previously unknown, meaningful patterns but it comes with a lot of challenges. The major problems include storing exponentially growing dataset, increasing heterogeneity of data, accessing and processing speed, insufficient disk space and incapability of traditional algorithms. Hadoop is an open source framework written in Java that allows users to deal with big data problems by providing facility to store the dataset on distributed environments and perform the computations in parallel. It provides massive scalability from single machine to thousands of clusters on commodity hardware, each offering local computation and storage. The RDBMS focuses mostly on structured data whereas Hadoop gives the flexibility to work with semi-structured and unstructured data as well.



Components of Hadoop

There are primarily two layers-

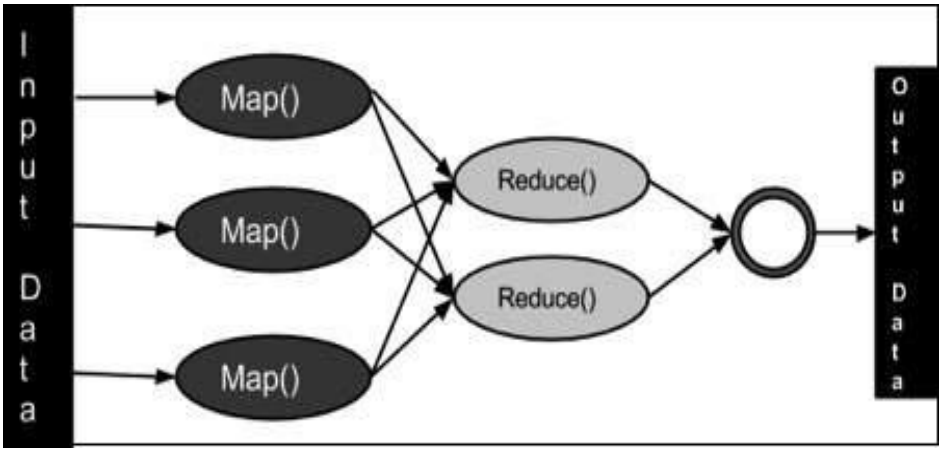
1. **MapReduce** – Processing and computation layer

2. **Hadoop Distributed File System** – storage layer

MapReduce

MapReduce is programming model for distributed computing suitable for huge data processing. Hadoop has the capability of running the MapReduce program in languages like Java, Ruby, Python, and C++. Multiple machines known as nodes run in parallel to produce the desired result. The term MapReduce represents two separate and distinct tasks- Map and Reduce. There are two different functions that need to be provided by the programmer. The complete process can be divided into four phases namely, splitting, mapping, shuffling, and reducing.

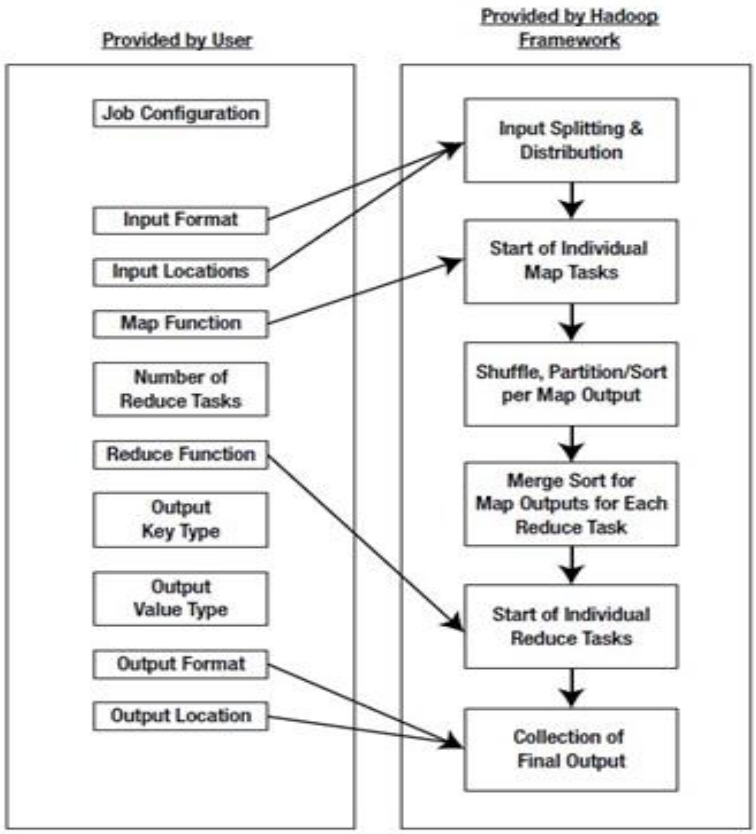
An input to a MapReduce job is divided into fixed-size pieces called input splits. It is a chunk of the input that is taken by a single map. Data in each split is passed to a mapper function which processes it to produce output values. In shuffling phase, the output of mapping phase is consumed. Its task is to collect the relevant records from Mapping phase output. The values are then combined and then reducing phase performs its task as specified by the programmer.



Map Reduce Architecture

JobConf is the framework that used to provide various parameters of a MapReduce job to the Hadoop for execution. The configuration set by JobConf is used by Hadoop platform to execute the program. The parameters are Map Function, Reduce Function, combiner, partitioning function, Input and Output format. Partitioner controls the shuffling of the tuples when sent from Mapper node to Reducer node. The total number of partitions done

in the tuples is equal to the number of reduce functions written by the user. Input Format describes the format of the input data for a MapReduce job. Input location specifies the location of the data file. Map takes an input pair and produces a set of *intermediate* key/value pairs. Intermediate values are grouped together based on same intermediate key I and passed to the Reduce function. The Reduce function accepts an intermediate key I and a set of values for that key. A possibly smaller set of values is formed. The intermediate values are supplied to the reduce function with the help of an iterator. Typically, the Hadoop framework consists of a single master and many slaves. Each master has a JobTracker and each slave has TaskTracker. Master distributes the program and data to the slaves. Task tracker, as the name suggests keep track of the task directed to it and gives back the information to the JobTracker. The JobTracker monitors and manages all the status reports and re-initiates the failed tasks, if any. Combiner classes are run on map task nodes. It takes the tuples emitted by Map nodes as input and performs reduce operation on them.



Hadoop Distributed File System (HDFS)

It is based on Google File System. It is designed for the storing of very large files with streaming data and capable of running on clusters of commodity hardware. It provides advantages like highly fault-tolerant, low-cost hardware, high throughput, suitable for applications having large datasets. It is designed in the form of master and slave pattern.

Organization of work by Hadoop

Hadoop divides the job into following 2 tasks:

1. Map tasks (Splits and Mapping)

2. Reduce tasks (Shuffling and Reducing)

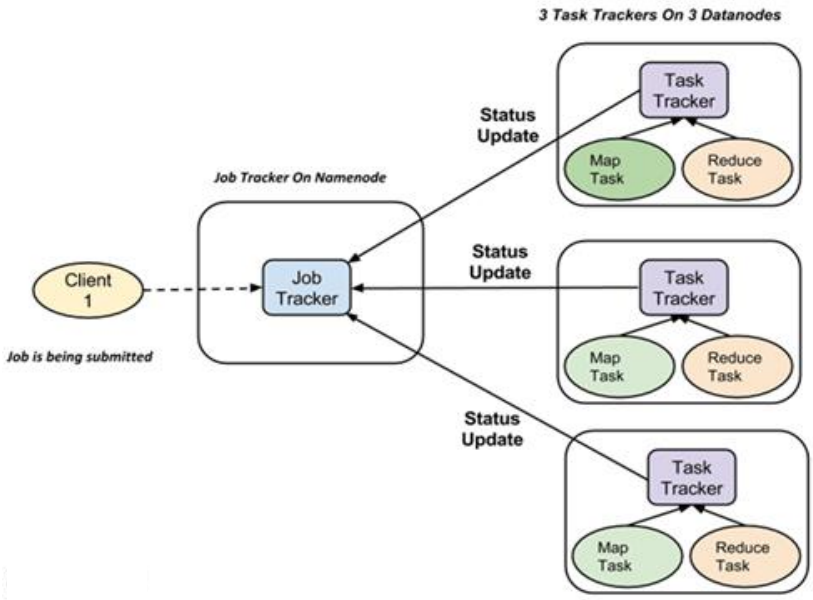
The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a

1. **Jobtracker** that acts like a master. It is responsible for complete execution of the submitted job.

2. Multiple Task Trackers that acts like slaves, each of them performs the job as specified by user.

For every job submitted for execution in the system, there is one Jobtracker that resides on **Namenode** and there are multiple tasktrackers which reside on **Datanode**.

Namenode is a machine that contains GNU/Linux operating system and the namenode software. It acts as a master server. It manages the file system namespace, regulates client's access to files and also executes renaming, closing, and opening files and directories and other file system operations. Datanode should have the datanode software. These nodes are responsible for managing the data storage of their system, performing read-write operation, creating, replicating and deleting blocks.



Spark

Spark is an open source, fast cluster computing platform with wide range data processing engine. Furthermore, Apache Spark extends Hadoop MapReduce to the next level. That also includes iterative queries and stream processing. Spark has in-memory cluster computation capability. Also increases the processing speed of an application. Spark run an application in Hadoop cluster, up to 100 times faster in memory and 10 times faster when running on disk. This is possible by reducing the number of read/write operations to disk. It stores the intermediate processing data in memory. Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.The main components of Spark include Spark Core, Spark SQL, Spark streaming, MLlib, GraphX, SparkR etc. The main component and key abstraction of Spark is RDD, Resilient Distributed Dataset. Also performs parallel operations. Moreover, Spark RDDs are immutable in nature. But, it can generate new RDD by transforming existing Spark RDD. There are 2 types of operations supported by Spark RDDs. **a) Transformation Operations**- It creates a new Spark RDD from the existing one. Moreover, it passes the dataset to the function and returns new dataset. **b) Action Operations-** Action returns final result to driver program or write it to the external data store.

The dataset

The Shuttle dataset- the dataset is multivariate with integer attributes (9). The dataset has 10 columns, the first column being time and the last being the class. The class has been coded as below-

1 Rad Flow

2 Fpv Close

3 Fpv Open

4 High

5 Bypass

6 Bpv Close

7 Bpv Open

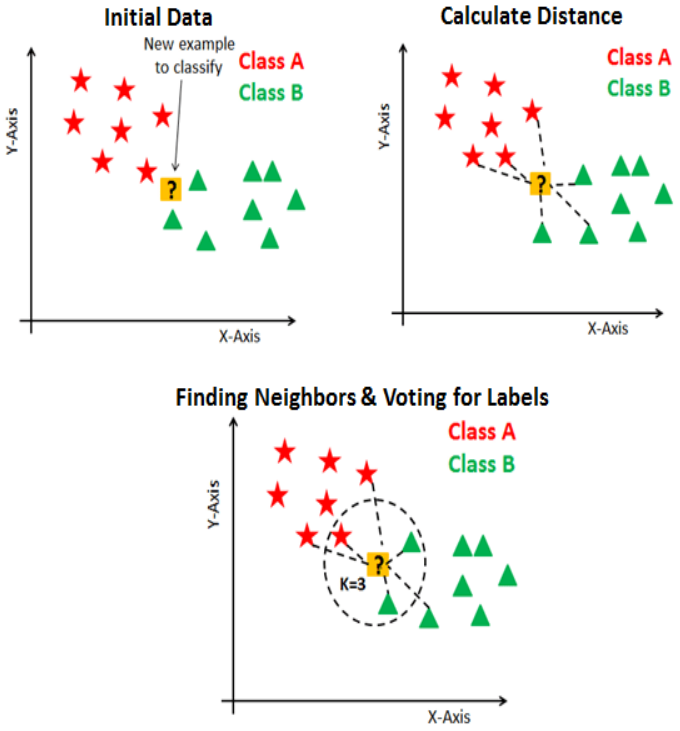
The above dataset is used for KNN whereas the dataset that is used for Bisecting K-Means contains 13 attributes, each having continuous values.

Problem statement

Implementing kNN-classifier algorithm and Bisecting-K means algorithm over MapReduce and Spark (using the Shuttle dataset) and comparing the execution time of each algorithm over both platforms by deploying in-built functions.

K- Nearest Neighbors (KNN)

Classification is a data analysis technique, the process of finding a model that is capable of assigning a class to each instance in testing data. It is the problem of predicting the category of a new test data point based on its features.K-Nearest Neighbors is one of the most basic yet essential classification algorithms in Machine Learning. KNN is an instance-based learning where the final computations are deferred until classification.



Bisecting K means

Clustering is the process of dividing the entire data into groups (also known as clusters) based on the patterns in the data such that the inter-cluster distance is maximum and intra-cluster distance is minimum. Bisecting K means is a modification of K means where each iteration increases the number of clusters giving convenience to stop the algorithm at required number of clusters. The centers of clusters and the data points in each cluster are adjusted by an iterative algorithm to finally define k clusters. Bisecting K means.

Comparing the KNN Execution in Hadoop & Spark

KNN using Map-Reduce Hadoop:

Generalizing the k-Nearest Neighbor:

1. Determine the value of ‘k’ (input)
2. Prepare the training data set by storing the coordinates and class labels of the data points.
3. Load the data point from the testing data set.
4. Conduct a majority vote amongst the ‘k’ closest neighbors of the testing data point from the training data set based on a distance metric.
5. Assign the class label of the majority vote winner to the new data point from the testing data set.
6. Repeat this until all the Data points in the testing phase are classified

The Mapper Design for k-NN

1. Load the file containing the test data set.
2. Load the file containing the training data set.

Repeat for all testing points: (creating separate temporary files for each testing data point)

Procedure KnnMapper:

1. Load testing data point one at a time.
2. Compute distance with every training data point for the test point
3. Insert these values into a map in ascending order and maintain the map size to k. (Tree-Map of size k)
4. Write (key, value) pair (distance, training_class) into the context.
5. Call reducer function.
6. end procedure.

The Reducer Design for K-NN

Procedure KnnReducer :

1. Load the value of k.

2. Open testFile created(tempfile) for reading the (key,value) pairs written earlier in mapper function.

3. Extract list of values from key:value pair.

    For each value in values:

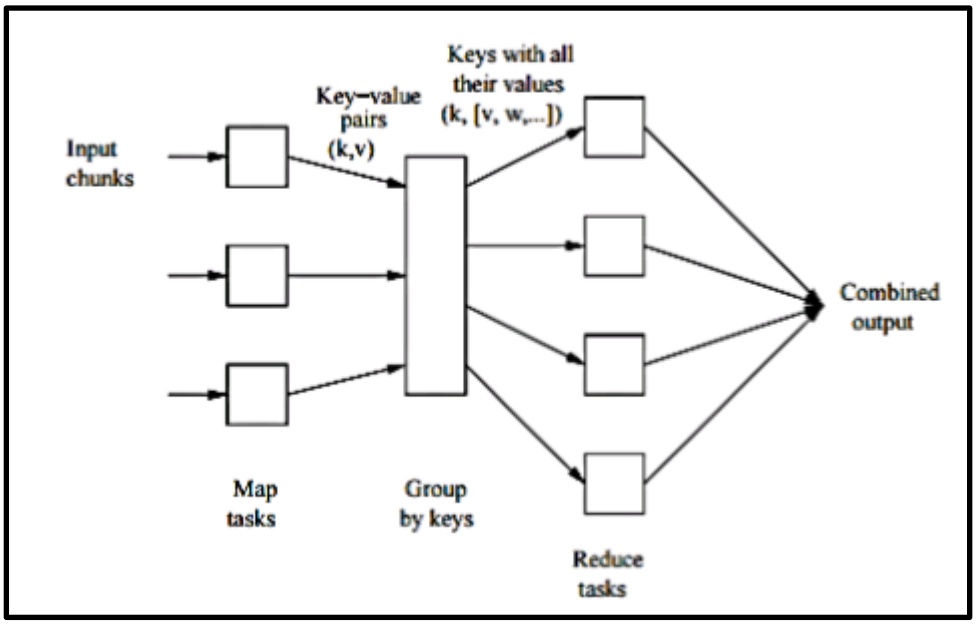
    Increment freq[value]

    Assign the test point the value of class label with highest frequency.

4. Update the output file with test point and assigned class label.

5. end procedure()

FLOW CHART



Input Chunks- testing data

Map Tasks- Mapping for each testing data point (k {Key: Value} pairs generated)

Reduce tasks- Calculating the majority class label among the k (key:value)pairs for each testing data point.

Combined output - Combine the result for all the testing points into a single file

```
public void reduce(NullWritable key, Iterable<DoubleDouble> values, Context context) throws IOException, InterruptedException
{
    for (DoubleDouble val : values)
    {
        double rModel = val.getModel();
        double tDist = val.getDistance();

        KnnMap.put(tDist, rModel);
        if (KnnMap.size() > K)
        {
            KnnMap.remove(KnnMap.lastKey());
        }
    }

        List<Double> knnList = new ArrayList<Double>(KnnMap.values());

        Map<Double, Integer> freqMap = new HashMap<Double, Integer>();

        // Add the members of the list to the HashMap as keys and the number of times each occurs
        // (frequency) as values
        for(int i=0; i< knnList.size(); i++)
        {
            Integer frequency = freqMap.get(knnList.get(i));
            if(frequency == null)
            {
                freqMap.put(knnList.get(i), 1);
            } else
            {
                freqMap.put(knnList.get(i), frequency+1);
            }
        }
        // Examine the HashMap to determine which key (model) has the highest value (frequency)
        Double mostCommonModel = null;
        int maxFrequency = -1;
        for(Map.Entry<Double, Integer> entry: freqMap.entrySet())
        {
            if(entry.getValue() > maxFrequency)
            {
                mostCommonModel = entry.getKey();
                maxFrequency = entry.getValue();
            }
        }

    // Finally write to context another NullWritable as key and the most common model just counted as value.
    context.write(NullWritable.get(), new Text(mostCommonModel)); // Use this line to produce a single classification
    context.write(NullWritable.get(), new Text(KnnMap.toString())); // Use this line to see all K nearest neighbours and distances
```

```
public void map(Object key, Text value, Context context) throws IOException, InterruptedException
{
    // Tokenize the input line (presented as 'value' by MapReduce) from the csv file
    // This is the training dataset, R
    String rLine = value.toString();
    StringTokenizer st = new StringTokenizer(rLine, " " );

        double a1 = normalisedDouble(st.nextToken(),minA1,maxA1) ;
        double a2 = normalisedDouble(st.nextToken(),minA2,maxA2);
        double a3 = normalisedDouble(st.nextToken(),minA3,maxA3);
        double a4 = normalisedDouble((st.nextToken()),minA4,maxA4) ;
        double a5 = normalisedDouble((st.nextToken()),minA5,maxA5) ;
        double a6 = normalisedDouble((st.nextToken()),minA6,maxA6) ;
        double a7 = normalisedDouble((st.nextToken()),minA7,maxA7) ;
        double a8 = normalisedDouble((st.nextToken()),minA8,maxA8) ;
        double a9 = normalisedDouble((st.nextToken()),minA9,maxA9) ;
        // double normalisedA10 = normalisedDouble((st.nextToken()),minA10,maxA10) ;
        // double normalisedA11 = normalisedDouble((st.nextToken()),minA11,maxA11) ;
        // double normalisedA12 = normalisedDouble((st.nextToken()),minA12,maxA12) ;
    // double normalisedRIncome = normalisedDouble(st.nextToken(), minIncome, maxIncome);
    // String rStatus = st.nextToken();
    // String rGender = st.nextToken();
    // double normalisedRChildren = normalisedDouble(st.nextToken(), minChildren, maxChildren);
    double rModel = Double.parseDouble(st.nextToken());

    // Using these row specific values and the unchanging S dataset values, calculate a total squared
    // distance between each pair of corresponding values.
    double tDist = totalSquaredDistance(normalisedA1,normalisedA2,normalisedA3,normalisedA4,normalisedA5,normalisedA6,normalisedA7,normalisedA8,normalisedA9,

    // Add the total distance and corresponding car model for this row into the TreeMap with distance
    // as key and model as value.
    KnnMap.put(tDist, rModel);
    // Only K distances are required, so if the TreeMap contains over K entries, remove the last one
    // which will be the highest distance number.
    if (KnnMap.size() > K)
    {
        KnnMap.remove(KnnMap.lastKey());
    }
}
```

KNN using RDD transformations SPARK:

Handle input parameters and create a spark session object.
Broadcast k and d as global shared objects where k = neighbors and d= no. pf attributes

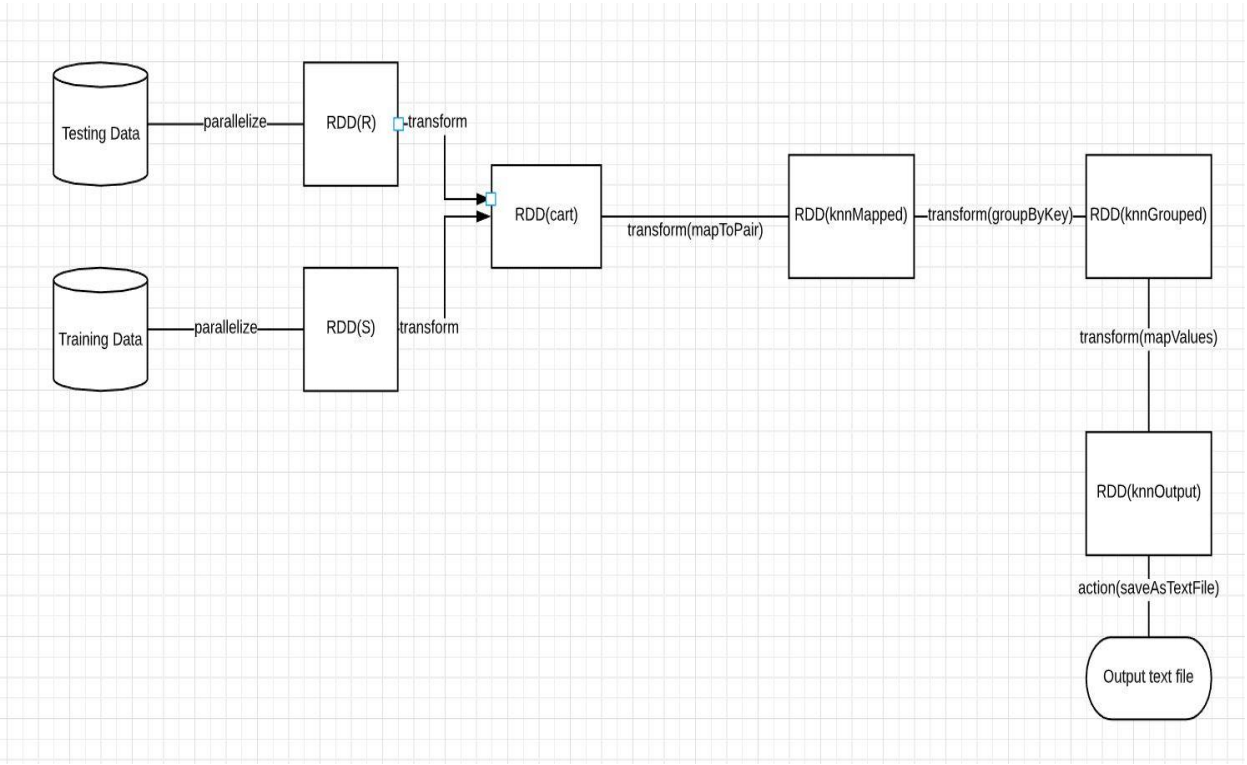
Step 1. Create RDD for Test(R) and Training(S) Data Sets
Step 2. Create new RDD “cart” which is cartesian product of the RDDs R and S.
Step 3. Obtain RDD “knnMapped”(i.e. an RDD consisting of key and value pairs) by performing transformation mapToPair() on RDD “cart”.“knnMapped” RDD has(Key,Value), where Key = unique-record-id-of-R, Value=Tuple2(distance, classificationID)
Step 4.Group distances by r in R ,grouping the results by r.recordID to obtain RDD knnGrouped using transformation groupByKey().
Step 5. Pass each value in the key-value pair knnGrouped RDD through a mapValues() transformation without changing the keys. Generate (K,V) pairs where K=r.recordID, classificationID in the RDD knnOutput.
Step 6.RDD knnOutput holds testing points and their respective classification. Final action saveAsTextFile() is applied onto this RDD.

```
JavaPairRDD<String, String> knnOutput =
    knnGrouped.mapValues(new Function<Iterable<Tuple2<Double,String>>, // input
                                      String                           // output (classification)
                                     >() {
  @Override
  public String call(Iterable<Tuple2<Double,String>> neighbors) {
      Integer k = broadcastK.value();
      // keep only k-nearest-neighbors
      SortedMap<Double, String> nearestK = Util.findNearestK(neighbors, k);

      // now we have the k-nearest-neighbors in nearestK
      // we need to find out the classification by majority
      // count classifications
      Map<String, Integer> majority = Util.buildClassificationCount(nearestK);

      // find a classificationID with majority of vote
      String selectedClassification = Util.classifyByMajority(majority);
      return selectedClassification;
  }
});
//
knnOutput.saveAsTextFile(outputPath+"/knnOutput");
```

Flow chart for kNN classifier on Spark



Comparing the Bisecting K-Means Execution in Hadoop & Spark

Bisecting K-means Using Map Reduce Hadoop:

Input: K = No. Of clusters; D: data points

Ouput: K clusters

```
S = {}  // empty set of clusters
Create single cluster C for all d points.
Add C to S.
For i = 1 to k-1 do
    Remove random cluster T from set S // Or one with lowest SSE.
    Call KmeansMapper on cluster T
    Call Kmeans Reducer on output of previous step
    Add resultant clusters to S.
End For
```

KmeansMapper-

Input: data points, K=2, two random points from data as centroids in the array centres

Output: pair, where key is data point and value is assigned cluster

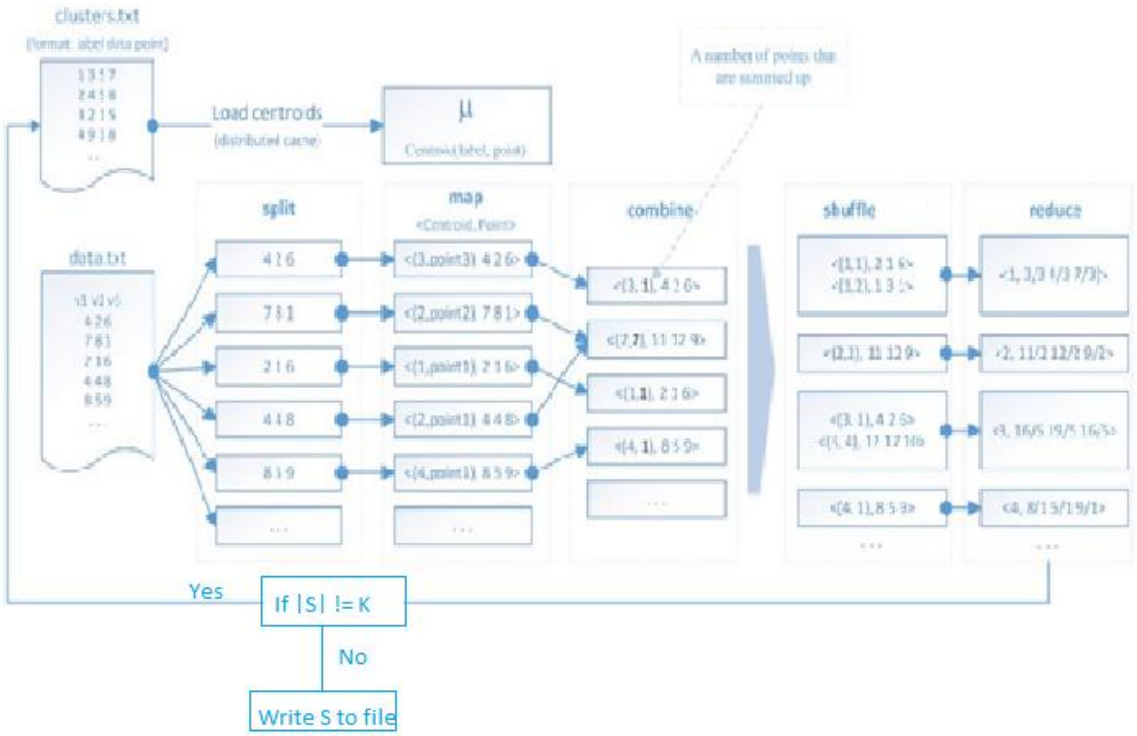
1. For each data point
2. minDis = Double.MAX VALUE;
3. index = -1;
4. For i=0 to centers.length do
    dis= ComputeDist(instance, centers[i]);
    If dis < minDis { minDis = dis; index = i; }
5. End For
6. Take datapoint as key';
7. Take value as index of cluster. (0/1)
8. output < key , value > pair;
9. End

KmeansReduce-

Input: output of mapper

Output: < key , value > pair, where the key' is the index of the cluster, value' is new centroid coordinates.

1. For each value (basically for each cluster) in list of values:
   Record the sum of all datapoints assigned to this value/index of cluster
   And maintain the count of these datapoints.
2. For each cluster calculate new centroid by dividing sum by count
3. Take key as cluster index
4. Construct value' as a string comprise of the center's coordinates.
5. output < key , value > pair;
6. End



Bisecting K-means Using RDD transformations SPARK:

Handle input parameters and create a spark session object.
Broadcast k as global shared objects where k = number of clusters.

Step 1. Construct a RDD with all data points as a single cluster : S (paired RDD <centroid,list(datapoints)>)
Step 2. Remove a random cluster from S into a new RDD T.
Step 3. Apply K means on RDD T to get two paired RDD R with <centroid,list(datapoints)> entries for each of the two centroids.

For applying K means on RDD T:

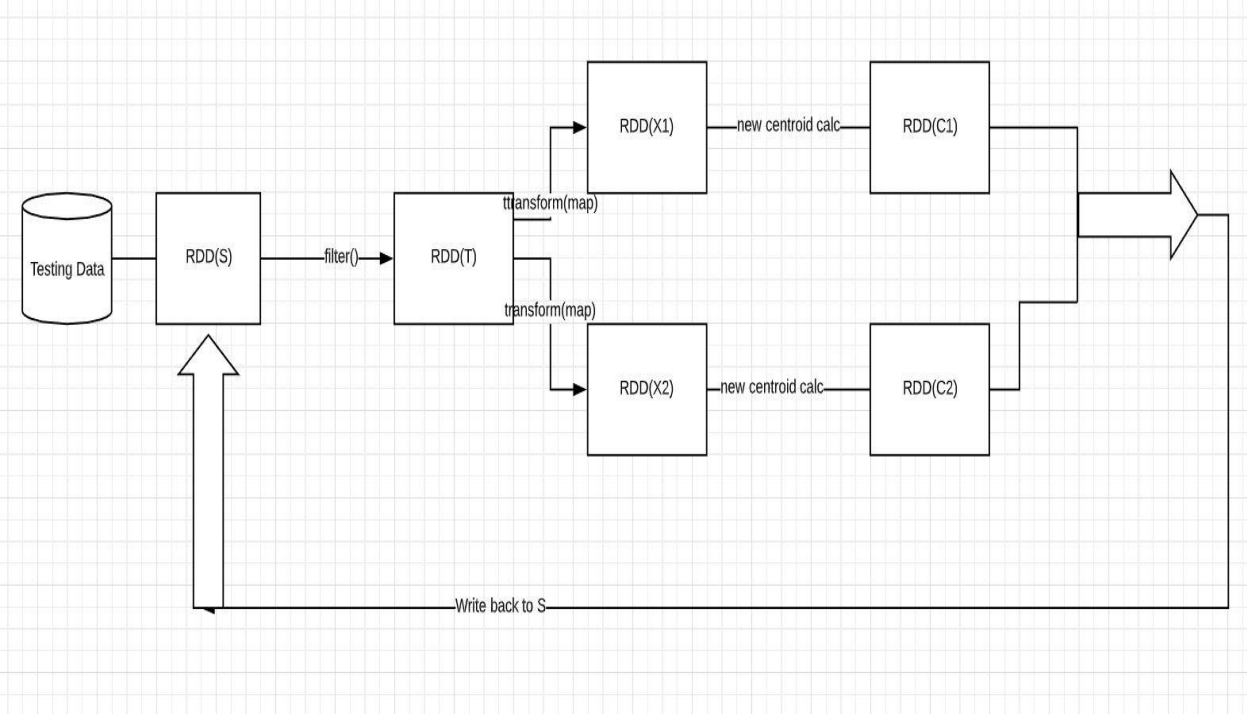
1) Choose two random data points from the list of datapoints as centroids.

2) Apply map transformation on this RDD to get (datapoint ,cluster) pair RDD after cluster assignment

3) Apply filter transformations to get two new pair RDD (centroid,list(datapoints)) for each cluster

4) Calculate new clusters and transform above two RDDs into new RDDs with updated centroids.

Step 4. Add the contents of these RDD to S.
Step 5. Repeat the above steps until number of entries in S equals K.



Conclusion

We found from our observations that running algorithms over spark takes less time than Hadoop. This is because Spark can do it in-memory, while Hadoop MapReduce has to read from and write to a disk. Spark passes the data directly without writing to persistent storage(disk). Spark has an in-memory caching abstraction. This makes spark efficient in iterative operations. Spark can therefore launch tasks much faster because it keeps an executer JVM running on each node.

References used

1. https://www.oracle.com/big-data/guide/what-is-big-data.html

2. https://hadoop.apache.org/docs/

3. http://ijiset.com/vol3/v3s10/IJISET_V3_I10_04.pdf

4. https://data-flair.training/blogs/spark-tutorial/

5. https://www.oreilly.com/library/view/learning-spark/9781449359034/ch04.html

6. https://www.programcreek.com/java-api-examples/index.php?api=org.apache.spark.api.java.function.Function

7. https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761

8. https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a

9. https://www.tutorialspoint.com/hadoop/hadoop_big_data_overview.htm

10. https://www.guru99.com/introduction-to-mapreduce.html

11. https://www.edureka.co/blog/mapreduce-tutorial/

12. https://spark.apache.org/docs/2.3.0/