Final Project Report (CS6240 – Map Reduce)
Identifying Spam Accounts in Twitter

Akshay Hathwar| Harish Rajagopal | Kosha Shah | Prashanth Ramanna

1. Introduction

Social networking sites have become very popular in recent years and in an important part of our modern life. There are several kinds of social networking websites, among these, Twitter is fastest growing site. Twitter offers several free services to users which attracts many companies and spammers to profit out of advertisements and schemes. This pollutes the search results and may cause inconvenience to genuine users. Removing such spammers is important for the site's reputation. Hence Twitter bans the creation of serial or bulk accounts to "artificially inflate the popularity of users seeking to promote themselves on Twitter." Identifying such accounts will provide advertisers with more accurate reports with respect to twitter usage and avoid users with reduced "annoying" tweets.

For this project, we have used APIs provided by Twitter to collect live Twitter feed which we analyze and evaluate to label all the unique users that we have collected. Our project involves identifying Spam accounts after learning and creating a model for accounts which were identified by twitter as fake. We do this by using different classification algorithms which will be parallelized to make them more efficient when used in Map Reduce Framework. The different classification algorithms used are k-Nearest Neighbors, Naive Bayesian with Binning, Decision Trees, and K-means classification algorithms.

2. DataSet

We are using Twitter's REST API to collect user profile information. After querying back Twitter and getting status of each of these users, we have labeled our data. The current size of our data set is around 12GB. We will make this data set available once we have completed the collection process.

The format of the dataset is line separated, JSON, twitter feed. For the sake of our problem set we filter out the feed based on the creation date of the user. The probability of a spam user is higher if his profile was created in the recent past, if the user has been around for a while, the information needed to identify such a user as a spam user might be more complex and we ignore such cases for now. Here we take users created in the year of 2014[i.e. The data spans over the last 4 months].

2.1.Sample Data Example

The feed below has been truncated for brevity, the original feed has over 65 fields. We mainly concentrate on the attributes of the user. (We have converted all the boolean attributes into numerical – true as 1 and false as 0).

default_profile, profile_background_tile, statuses_count, favorites_count, protected, contributors_enabled, description, verified, default_profile_image, followers_count, geo_enabled, friends_count, profile_use_background_image, is_translator, listed_count, label

```
{"contributors": null, "truncated": false, "text": "I am on visit to my lovely Ingridpus &lt;3", "in_reply_to_status_id": null, "id": 448184195240972288, "favorite_count": 0, "source": "<a href=\"http://twitter.com/download/android\" rel=\"nofollow\">Twitter for Android</a>", "retweeted": false, "coordinates": null, "entities": {"symbols": [], "user_mentions": [], "hashtags": [], "urls": []}, "in_reply_to_screen_name": null, "id_str": "448184195240972288", "retweet_count": 0, "in_reply_to_user_id": null, "favorited": false, "user": {"lang": "en", "utc_offset": null, "statuses_count": 2, "follow_request_sent": null, "friends_count": 0, "profile_use_background_image": true, "contributors_enabled": false, "profile_link_color": "0084B4", "profile_image_url": "http://pbs.twimg.com/profile_images/448184135853805568/a4JWIGVX_normal.jpeg", "time_zone": null, "notifications": null, "screen_name": "tiffinyjzwis", "favourites_count": 0, "profile_background_image_url_https": "https://abs.twimg.com/images/themes/theme1/bg.png", "profile_background_color": "C0DEED", "id": 2433562121, "profile_background_image_url": "http://abs.twimg.com/images/themes/theme1/bg.png", "description": null, "is_translation_enabled": false, "default_profile": true, "profile_background_tile": false, "verified": false, "name": "Tiffiny Conley", "url": null, "profile_image_url_https": "https://pbs.twimg.com/profile_images/448184135853805568/a4JWIGVX_normal.jpeg", "profile_sidebar_fill_color": "DDEEF6", "location": "\u0130stanbul", "is_translator": false, "geo_enabled": false, "profile_text_color": "333333", "followers_count": 0, "profile_sidebar_border_color": "C0DEED", "id_str": "2433562121", "default_profile_image": false, "following": null, "protected": false, "created_at": "Mon Mar 24 15:43:38 +0000 2014", "listed_count": 0}, "geo": null, "in_reply_to_user_id_str": null, "lang": "en", "created_at": "Mon Mar 24 19:46:53 +0000 2014", "filter_level": "medium", "in_reply_to_status_id_str": null, "place": null}^M
```



```
{"userId": "2433512322", "userAttr": {"is_translation_enabled": false, "profile_background_tile": false, "favourites_count": 2, "description": "the one i stan is the one who i've never met and they are \u6771\u65b9\u795e\u8d77", "friends_count": 46, "profile_use_background_image": true, "verified": false, "contributors_enabled": false, "utc_offset": null, "default_profile_image": false, "notifications": null, "followers_count": 24, "protected": false, "default_profile": true, "following": null, "statuses_count": 190, "geo_enabled": false, "listed_count": 0, "is_translator": false, "follow_request_sent": null}, "label": false, "tweetAttr": {"replied": 1, "contributors": 0, "numberOfHashTags": 0, "numberOfUrls": 0, "retweetCounts": 0, "numberOfUserMentions": 0, "favCount": 0}, "tweets": ["kapam kurs dollar turun ya fufufu."], "screenName": "cassiopeiannn"}
{"userId": "2383645013", "userAttr": {"is_translation_enabled": false, "profile_background_tile": false, "favourites_count": 23, "description": null, "friends_count": 106, "profile_use_background_image": true, "verified": false, "contributors_enabled": false, "utc_offset": null, "default_profile_image": false, "notifications": null, "followers_count": 107, "protected": false, "default_profile": false, "following": null, "statuses_count": 1575, "geo_enabled": false, "listed_count": 0, "is_translator": false, "follow_request_sent": null}, "label": false, "tweetAttr": {"replied": 10, "contributors": 0, "numberOfHashTags": 1, "numberOfUrls": 0, "retweetCounts": 0, "numberOfUserMentions": 12, "favCount": 1}, "tweets": ["RT @Mlle_Wiilson: Stacy Mon Bebe d amour", "RT @AliciaYaRien: @StacyEndy242 mdddddddddr tu mcherche ???", "RT @Mlle_Wiilson: @StacyEndy242 ouiii jc", "RT @NinaTanyaR: La meilleure des r\u00e9ponses.. LE SILENCE.. Ne rentre pas dans leurs jeux fait preuve d'intelligence.. http://t.co/1yEg5LBjP2", "@Mlle_Wiilson Kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk Toi Mm Nan Mais Je R\u00eave", "RT @tichabiina971: @StacyEndy242 Le collage dune parti de votre visagee Mdrrr", "RT @ClaraDalwear: #PP http://t.co/qU3Le9U1Ek", "Sal pute", "@Mlle_Wiilson Demande a Christian please", "RT @NinaTanyaR: DITES MOI C'EST QU'ELLE FDP QUI A FAIT CE MONTAGE ?? PTN \ud83d\ude02 \ud83d\ude02 http://t.co/YbJW4OodgI"], "screenName": "StacyEndy242"}
```


Sample of our final dataset

part-r-00001

```
0, 1, 237, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0
1, 0, 23, 1, 0, 0, 1, 0, 0, 16, 0, 29, 1, 0, 0, 0
1, 0, 1682, 2238, 0, 0, 1, 0, 0, 134, 1, 238, 1, 0, 2, 0
1, 0, 194, 2952, 0, 0, 1, 0, 0, 7, 0, 41, 1, 0, 0, 0
1, 0, 28, 17, 0, 0, 1, 0, 0, 7, 0, 138, 1, 0, 0, 0
0, 1, 222, 0, 0, 0, 1, 0, 0, 116, 0, 92, 1, 0, 0, 0
1, 0, 1308, 390, 0, 0, 1, 0, 0, 20, 0, 101, 1, 0, 0, 0
0, 1, 434, 0, 0, 0, 1, 0, 0, 794, 0, 339, 1, 0, 0, 0
1, 0, 6, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0
0, 1, 14557, 0, 0, 0, 0, 0, 0, 4873, 0, 2279, 1, 0, 8, 0
1, 0, 270, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0
0, 1, 2917, 667, 0, 0, 1, 0, 0, 157, 0, 110, 1, 0, 0, 0
1, 0, 446, 132, 0, 0, 1, 0, 0, 116, 0, 213, 1, 0, 0, 0
1, 0, 225, 30, 0, 0, 1, 0, 0, 92, 0, 72, 1, 0, 0, 0
1, 0, 47, 0, 0, 0, 1, 0, 0, 2, 0, 0, 1, 0, 0, 0
1, 0, 1743, 0, 0, 0, 1, 0, 0, 82, 0, 11, 1, 0, 0, 0
1, 0, 89, 6, 0, 0, 0, 0, 0, 37, 0, 39, 1, 0, 0, 0
1, 0, 744, 249, 0, 0, 1, 0, 0, 118, 0, 199, 1, 0, 1, 0
0, 1, 99, 12, 0, 0, 1, 0, 0, 2, 0, 0, 1, 0, 0, 0
```

3. Technical Discussion

3.1.Data Collection and Data Preprocessing

3.1.1. Purpose

Collect the live tweets and user information from the Twitter data and process the data to get rid of the unwanted attributes. The data is also transformed into the required format

3.1.2. General Information

- Tweets collected : 12 Million
- Unique Users : 2.7 Million
- Filters : Tweets of users from 2014 only to be collected

3.1.3. Data Collection

We have used tweepy and Twitter API to collect data. Tweepy is an abstraction layer over Twitter API for python. Tweet Object containing information about the tweet and an user object containing information about the user who had sent out that tweet

3.1.4. Data Transformation

Since the data given out by twitter is of user tweets, we had to transform it to get data where feature w.r.t one particular user could be accessed.

Feature Aggregation

Transformed and aggregated features from tweet object

Feature Extraction

Extracted features from the existing ones to merge and provide features with more information

Dimension Reduction / Feature Selection

We then selected the features we thought would be of most value and filtered the rest

3.1.5. Data Labeling

Once we had the filtered data, we had to classify them as spam/not spam, this was done by calling on the Twitters API to see if the userID existed. If the userID existed, we conclude the user is not a spammer else we assume he is one.

3.2.kNN Classification

3.2.1. Purpose

Classifying the spam accounts in Twitter

3.2.2. Main Idea

k-Nearest Neighbors is a classification algorithm which checks class membership of the top K neighbors of the test records to classify them. This can be achieved by performing following steps for every record in test dataset:

- Calculate the distance (Euclidean distance or Cosine similarity) of test record from all the training records
- Sort the training set based on the distances calculated above
- Consider the top k records and predict the majority class among the records.

For implementing the above steps, I have used map-only join and value-to-key-conversion design patterns.

Map-Only Join

Test dataset is stored in Distributed Cache. This makes it available to all the map task. Now each map call reads the training dataset from its input and joins each test record with each of the training record. Here the joining is done by creating a new object of type *DistanceNode* and assigning the value of test record, the training record and the distance between them to the class members of each object. This map calls emits the node it creates as the key and the class to which the training record belongs.

Value-to-key-conversion

After getting all the distances for a particular test record, sorting the training tuples based in distance is required. For this, I have implemented secondary sort. This requires a custom partitioner, key comparator, and a grouping comparator.

- My custom partitioner partitions based on the test records.
- The Key comparator sorts all the training records joined with the same test record based on distances.
- Finally, grouping comparator makes sure that all the records of the same test record goes to the same reduce call.

Hence now in my reducer, key would be the *DistanceNode* of a particular test record and values would be the list of labels of training records sorted based on distances.

Now, in reducer I will check the top k tuples and take majority voting for the classes.

3.2.3. Algorithm and Pseudo code

Input to the Algorithm:

Input is the final file which we get from the data preprocessing step. Each training record has 16 attributes including the label. We have converted all the attributes to numerical ones to simplify the similarity/distance calculation. There were no categorical attributes in original data, but the boolean attributes are converted into numerical by substituting 1 for true and 0 for false or null.

Driver: (kNNDriver.java)

```
main():
   create a configuration object
   add a cache file in DistributedCache
   Create a Job object
   set the other job details
```

DistanceNode: (DistanceNode.java)

```
   class members: String testPoint, String trainingPoint, double
                  distance
   euclideonDistance(String testPoint, String trainingPoint):
      testPoint[] <- testPoint.split(",")
```

```
      trainingPoint[] <- trainingPoint.split(",");
      for(each value in testPoint):
          apply the following formula for euclidean distance:
               double d = ((x1 - y1)^2 + (x2-y2)^2 + ...)
      distance(X,Y) = sqrt(d)
      return the distance

    cosineSimilarity(String testPoint, String trainingPoint):
      testPoint[] <- testPoint.split(",")
      trainingPoint[] <- trainingPoint.split(",");
      for(each value in testPoint):
          apply the following formula for euclidean distance:
             similarity(X,Y) = Σ(Xi * Yi) ÷ (√(Σ(Xi * Xi)) * √(Σ(Yi * Yi))))
      return the similarity
```

Mapper:(kNNMapper.java)

```
    setup():
          read the test data set from the DsitributedCache
    map():
          Create a new DistanceNode and calculate the distance
          between testPoint and trainingPoint
          context.write(node, label from the trainingPoint);
```

Partitioner: (kNNPartitioner.java)

```
   getPartition():
          return Math.abs(node.gettestPoint().hashcode()*127) %
                          numOfReducers
```

KeyComparator: (kNNKeyComparator.java)

```
   compare():
          sort based on the distances for the records belonging
          to the same testPoint
```

GroupingComparator: (kNNGroupComparator.java)

```
     compare():
         group all the test records in one reduce call
```

Reducer: (kNNReducer.java)

```
     reduce():
         Iterate through the top k values
         vote for the majority class
```

3.2.4. Concrete Results

Confusion Matrix

	Predicted Class	
Actual Class	**Yes**	**No**
Yes	TP	FN
No	FP	TN

True Positive: A fake record detected as fake

Cosine Similarity:

Data statistics: *# test records = 95, #training records = 20420, positives=15, negatives=80*

	Predicted Class	
Actual Class	**Yes**	**No**
Yes	6	5
No	9	75

Accuracy: (TP + TN)/ (P + N)
cosine similarity = 100*(6+75)/(95) = 85.26%

Euclidean distance

Data Statistics: *# test records = 95, #training records = 20420, positives=40 negatives=54*

	Predicted Class	
Actual Class	**Yes**	**No**
Yes	10	1
No	30	54

Accuracy: (TP + TN)/ (P + N)
Euclidean distance = 100*(10+54)/(95) = 67.36%

3.2.5. Analysis

Since kNN algorithm requires joining of Training and Testing dataset, I can either run small dataset on m1.small instance types or medium sized datasets on large instances.

Following are the few of the many runs performed on AWS:

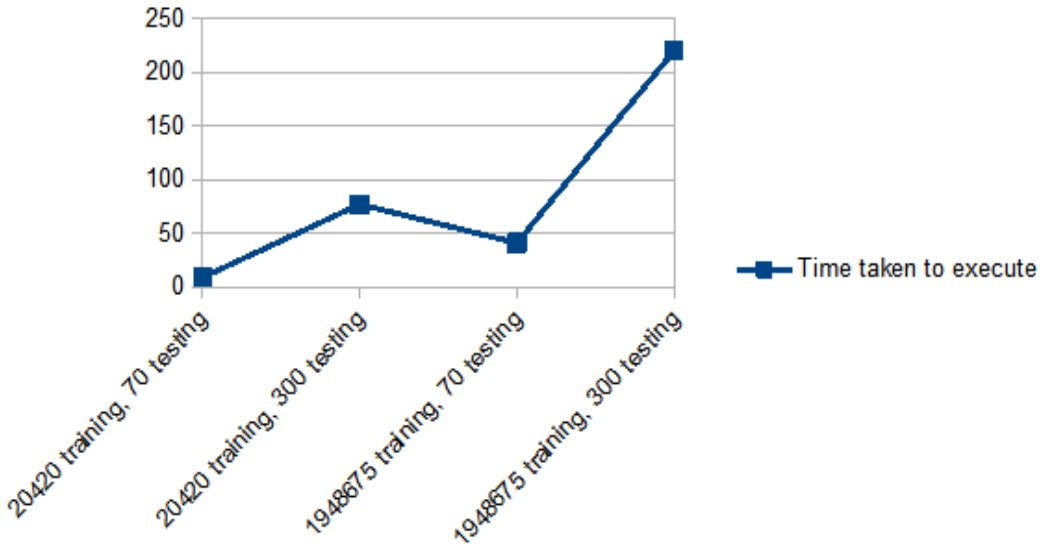
Type of instances	**Dataset properties**	**Execution time**
M1.small(1 master and10 cores)	20420 training records and 70 testing records	22 minutes
M1.small(1 master and 10 cores)	20420 training records and 20420 testing records	**Did not proceed from map 0% reduce 0% for almost 45 mins
M1.small(1 master) and m1.large(5 cores)	20420 training records and 70 testing records	9 minutes

**If the algorithm is executed on large testing datasets on large machines, then it does not proceed from map 0% reduce 0%, because it has to first read such a huge dataset from DistributedCache and secondly it needs to join both the datasets in mapper

As you can see in the above table, map only join fails when both the datasets is large enough, though it passes and runs without any error when one of the dataset is smaller.

Type of instances	Dataset properties	Execution time
M1.small(1 master) and m1.large(5 cores)	20420 training records and 70 testing records	9 minutes
M1.small(1 master) and m1.large(5 cores)	20420 training records and 300testing records	77 minutes
M1.small(1 master) and m1.large(5 cores)	1948675training records and 70 testing records	41 minutes
M1.small(1 master) and m1.large(5 cores)	1948675training records and 300 testing records	220 minutes

Graph:



There are efficient alternatives to implement map only joins like Semi-join and advanced map only join. Here Semi join tries to find the unique joining key shared between the two datasets and then perform the map only join [1]. This cannot be implemented for kNN since each test record needs to be joined with every other training record (assuming there are no duplicate training records in the dataset).

The above table and the graph shows that kNN algorithm implementation is not so scalable against the data size. As pointed out previously, the bottleneck of this algorithm is the joining performed. Map-only joins are better performant than the default join algorithm as it requires reducers, which involves the data transfer from mappers to reducers. Map-only joins seems to be affecting the scalability of the kNN algorithm. This is mainly because the logic of the algorithm itself which requires each test record to be joined against the training records.

As seen in the syslog file (submitted in the <foldername>), the map output records are approximately #records in training * #records in test. Hence this number grows exponentially with the size of test dataset.

3.2.6. Challenges

Distributed Cache:
Distributed Cache is used in kNN implementation to store the test dataset and make it available to all the map tasks. Hence the test dataset file was added to the distributed cache in the driver program using the configuration object but it was not reflecting in the mappers. After debugging and searching on the web, I realized that while creating the job object, it makes a copy of conf object passed to it and then proceeds further. Hence any changes made to the conf object will not be reflected in the job. Hence all the configuration related parameters should be set before creating the job object.

Joining:
KNN algorithm requires the information of distances between each training and testing record. This was implemented using map-only joining. The major bottleneck for the performance of this algorithm is the map-only join. This is because, it joins each test record with every other training record. This makes the computation quite slow. While running on AWS, it took lot of time and made debugging difficult and tedious. Hence to get more information, I installed EMR CLI which makes debugging easier.

3.3.Decision Trees

Decision tree is one of the frequently used classification algorithm, in this task I implemented a parallel version of decision tree, in which all the nodes at the lowest level(except leaf nodes) are expanded in one iteration of a Map reduce job. This was motivated from the Google's implementation of PLANET and other such papers[1][2][5].

3.3.1. Datasets :

- **Twitter** : 2.6M records, 16 features , Binary , Twitter API
- **HIGGS** : 11M records, 28features, Binary, UCI ML Repo

3.3.2. Algorithm:

InitializationJob : Runs once to find K thresholds per attribute
Input : Dataset , K-number of thresholds per attribute
Output : Attribute Index, Attribute value

Mapper
h: new HashMap<Integer,HashMap<Double,<Integer>>

map(record)
for every attribute in record
    h(attrIndex) ->(attrValue->freqCount)

cleanup()
for every record in h
    emit((h.attrIndex,h.attrValue),freqCount)

Reducer

```
interimSum=0
Reduce(CustomKey key, Iterable[Int] FreqCount)
        Find totalNumber of records [Use dummykey]
        attrIndx = key.indx
        valuesPerBucket = total/K
        Sum up FreqCount and at every valuePerBucket set threshold
        Emit(Null,Split(attrIndx,threshold))
```

FindBestSplitJob : For Each eligible Node selects the best Split to expand it
Input : Global Splits, Set of Nodes to expand, Current model, Dataset
Output : Node,Split, ChildEntropyInfo(left & right)

Mapper

```
setup()
N = get set of Nodes from HDFS
M = get Model from HDFS
H = HashMap<Node+Split,ChildEntropyInfo>

map(Dataset d)
 Splits = get file handle for global threshold file
 For every split in splits
        For every datapoint in dataset
                For valid node the datapoint percolates to:
                        Get child Entropy Info by splitting datapoint w.r.t split
                        Update h

cleanup()
For k,v in h
 Emit(k,v)
```

reducer :

minSplit : Maintain overall minimum IG and corresponding split

```
reduce(Key node+split , Values Iterable[ChildEntropyInfo])
        IG = calculate Info gain for the split
        Update minSplit if IG is better

cleanup()
        Emit (node,minsplit + childEntropyInfo)
```

3.3.3. Design Decisions

Build Decision tree in a breadth first :

Design: The primary reason behind this as explained in [1][2][5] is to enable faster exapansion of nodes. Also this can be leveraged to build random forest / bagging very easily. Breadth first approach to building decision trees enables us to expand all the nodes on a single level in one iteration. As we grow the tree theoretically we should be able to improve parallelism exponentially.

Implications: While growing trees in breadth first manner is better than Depth first when running in a distributed manner[5], there is a inherent upper bound in this approach i.e. the number of leaf nodes being expanded. In an ideal scenario with a balanced tree our parallelism increases exponentially. But, this might not always be the case and hence might be a possible bottleneck in cases where the tree is unbalanced. Also, the bigger the tree gets, more complex it becomes more likely it is going to overfit.

Observations:
Please refer the folder builder/* for the data being used to make the below observations.
The bottleneck w.r.t performance seems to be the first iteration which goes painfully slow. The iterations take place at a good rate.

- Reasons
  - Input file size: 2 input files, of size 30 mb and 70 mb are too small for Hadoop to handle.
  - Confirmed by splitting the files and found relatively improved performance.
  - Map task of FindBestSplitJob is computationally intensive.

Finding thresholds to split a node on:

Design: I find K thresholds per Attribute and store it in a file as mentioned in PLANET paper[1]. This serves as a global provider of thresholds, using which all the nodes are expanded. There are other ways to find thresholds , but partly for avoiding complexity and partly because from

Implications: Distibuted cache is used to provide the filename to find the global thresholds for every task. If the file is small enough there wont be too much overhead in accessing data. But, as we increase K, the size of the file increases and this might be a bottleneck.

Observations:

K	*5 Machines*	10 Machines	*15 Machine*
100	*1.5 mins*	*1.2 mins*	*2 mins*
1000	*1.7mins*	*1.8 mins*	*1.3 mins*

Finding optimality of each of the thresholds:

Design: This is a textbook secondary sort application, through which we ensure every node is allocated a reduce task and every reduce call calculates the best threshold per attribute

and during the cleanup we find the best threshold to apply. Other approaches explored needed an additional iteration, which we wanted to avoid.

Implications : Sending out SplitInfo[Entropy for left and right child] from each map after applying in mapper design, ensures reduced network chatter.

Observations: The reduce tasks finish there work at a good rate, where every task takes around 30-40 secs. But, the map task as mentioned before, takes most of the computation time.

Runtime:

#Machines	5	10	15
Runtime	43mins	47mins	48mins

Accuracy

	4	8
100	87%	90%
1000	92%	93%
10000**	93%	93%

**: This was run on a local for a smaller dataset

3.3.4. Conclusions :

- Calculating thresholds per feature globally, doesn't give us too much of an advantage as hoped while designing. The amount of time taken for this as mentioned above, means we can design jobs which find thresholds for features on each iteration. This would be scalable[map and reduce tasks take 30-40 seconds]
- Due to the data size, its hard to rule conclusively whether map task is coarsely grained, but from the results we have, it does seem the map task can be broken down to finer grained tasks.
- To add on, with larger number of K, we would need to read larger number of files in mapper which would add on to the file read operations which are currently around 20-100 mb

3.3.5. Challenges :

1. Serialization of model and extracting global thresholds from HDFS was initially tricky, but AWS documentation came handy.

2. Heap space and GC issues with larger global threshold file when testing with larger datasets. EMR CLI came in handy in debugging and getting to the possible root causes for the issue. It provides all the info there is about the cluster and very helpful tool when debugging on distributed systems.

3. Due to the high amount of time taken, made it a very long debugging lifecycle.

3.4.Naive Bayes

In Naive Bayes classifier we approximate an unknown function $f: A \rightarrow L$, where $A = A1 * \ldots * Am$ is the attribute space and L is the label space. For our dataset L is binary (0,1).
If $X = (X1, \ldots, Xm)$ is a random variable taking values in F and Y is a random variable taking valuesin L, then $f(x)$ can be estimated by computing $Py = P(Y = y \mid X = x)$
By Bayes' theorem we have:

$$P(Y = y \mid X = x) = \frac{P(Y = y).P(X = x \mid Y = y)}{P(X = x \mid Y = y)}$$

Now considering the naive conditional independence assumption:
assume that each feature Fi is conditionally independent of all other Fj features in the same category C.
Simply putting the whole thing the probability of class Cj given that we have d is:

$$P(cj \mid d) = \frac{P(d \mid cj)\, P(cj)}{P(d)}$$

and to simplify this we have:

$$P(d \mid cj) = P(d1|cj) * P(d2 \mid cj) * \ldots * P(dn|cj)$$

3.3.6. TRAINING

The most intensive calculation is to calculate the conditional probabilities, which involves the frequency of the label or the combination of the label, attribute name and attribute value.

To learn about users in the training dataset, we need to generate a data structure that not only provides for Naive Bayes' needs but also to help in our binning.
Here we maintain a HashMap where the key is the Attribute index and the value is a another HashMap.
This second HashMap has keys which is a string formed by concatenating the attribute value along with the label that is related to this and the value is basically 1.This basically helps us record the number of instances of unique value, label pairs for each attribute.

Since we also implement binning, we maintain the min and max value of all records in each of these attributes.

The map function parses the combination of the attribute index ,the value and the label along with the frequency aggregated so far.

We do not use global variables here, instead we just pass the MIN and MAX values towards the end with a special key such that when Hadoop sorts the intermediate keys, this entry will always be the first record to reach a reduce.

3.3.7. TRAINING MAP ALGORITHM

Input: the training dataset
Output: < key' , value' >, where key' is the combination of attribute index, value and label, and value' the frequency.

```
for each sample do
    - for each attribute value do
        - parse the label and value of each attribute
        - maintain the MIN and MAX for each attribute
          • construct a string as the combination of
          attribute index, attribute value and label which
          will be key'
          • maintain a frequency count for the string
          attribute value + label which will be value'
          • output <key' , value' > pair
          • construct a string as the combination of
          attribute index, and 2 unique values which will
          be key'
          • and value' would be the combination of MIN
          and MAX
          • output <key' , value' > pair
- end for
end for
```

Since our main intention is to have reduce tasks equal to the number of attributes, we partition based on number of attributes. Hence each reduce task will be in charge of processing the data of just one attribute.

In the reduce function, with the first <key', value'> we construct a histogram for each attribute and then perform binning on each of these histograms based on the MIN and MAX values that are passed in value'.
We then iterate over the intermediate keys, and update the respective histograms with their frequency counts based on which bin they fall under.
We maintain counters which denote label specific counts in each bucket as well.
we finally emit the attribute index as key and the value is a flattened out histogram for that respective attribute index.

3.3.8. ALGORITHM REDUCE TRAINING

Input: key and value (the key' and value' output from map)
Output: < key", value"> where key" is the Attribute index and value" is the flattened out version of the array representation of the histogram.

```
for each value do
   if key string has unique label and attribute index
        -set up histogram and bin thresholds
   else
        -fit the attribute value into a bin
        -increment respective counters for that bin
```

```
  end for

  set key" as the attribute index
  set value" as the flattened histogram
  output <key", value">
```

When training the model, the *map* function parses the label or the combination of the label, attribute name and attribute value, and the *value* of the *key* is 1.
For the *reduce* function, we count the frequency of each *key*.

3.3.9. TESTING

So far, the parameters required for the naive Bayesian classifier have been or calculated, including cj for P(cj) and Ai for P(Ai|cj), where cj denotes the total data point in the j-th bin of the histogram, Ai the number for labelled data points for i-th attribute.

When testing, the map function first indexes the key in the results produced by the training step, and reads the histograms for the corresponding attribute indexes. This is done using the distributed cache that has been made available in Hadoop.

Once this is set up, we iterate over the same training data records and then calculates the probability of the record based on each of the attributes by placing them in appropriate bins and calculating their probabilities.
So the label can be predicted according to the maximum posterior and a pair is output for each sample which takes a correctness factor of 1(correct) or 0 (wrong) as the key, and their respective counts as the value.

3.3.10. ALGORITHM TESTING MAP

Input: the testing dataset and the reduce result of the training step
Output: < Key', Value' > where key' is the correctness factor and value' the frequency

```
1. parse the reduce result of training step to get histogram
indexed by attribute index and load file onto distributed cache
2. for each record do
     for each attribute do
          fit the attribute value into a bin(i) in the
histogram
          prob_spam *= spam_count(i)/total_count(i)
          prob_non_spam *=  non_spam_count(i)/total_count(i)
     end for
     if(prob_spam > prob_non_spam)
          if (record_label == 1)
               correct_counter++
          else
               incorrect_counter++
end for
```

```
set key' as 1 and value' as correct_count
output <key', value'>
set key' as 0 and value' as incorrect_counter
output <key', value'>
```

For the reduce function , the number of the correctly or wrongly predicted samples can be calculated by just summing up the list of correct_count values and incorrect_count values in respective reduce tasks as we use a partitioner to send them to 2 different ones.
Therefore, the correct rate and error rate can be further calculated

3.3.11. Design Considerations

While designing this algorithm, we started off with a histogram containing 5 bins, and to experiment with the data distribution we increased the number of bins to 10. We found that the data did contain come attributes which did have values that would mainly fall under the lower or higher bins leaving the middle bins sparse.

We have hence restricted the experiments to just 5 and 10 bins

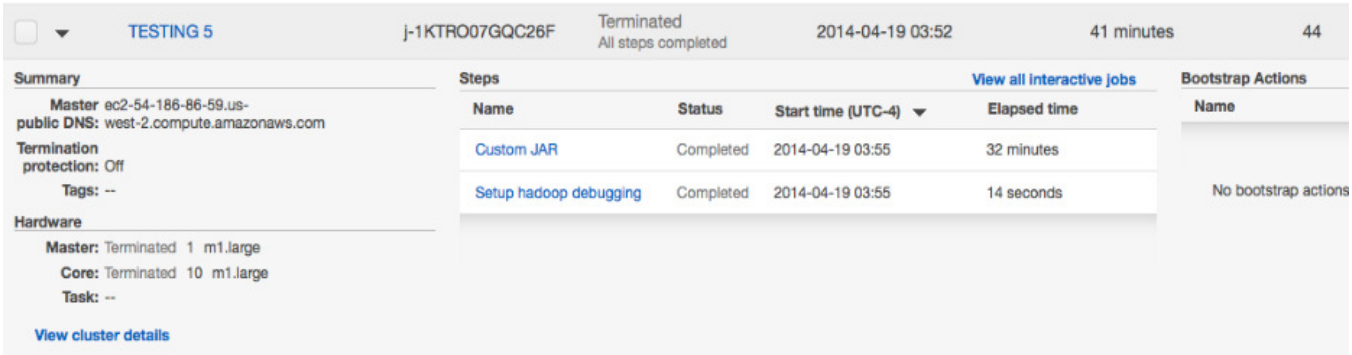
3.3.12. Twitter Dataset

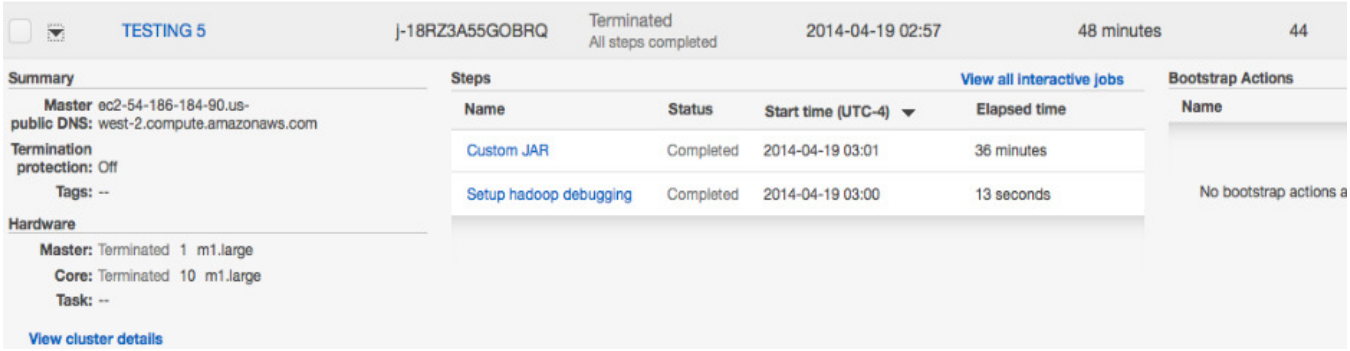
Number of buckets in histogram = 5

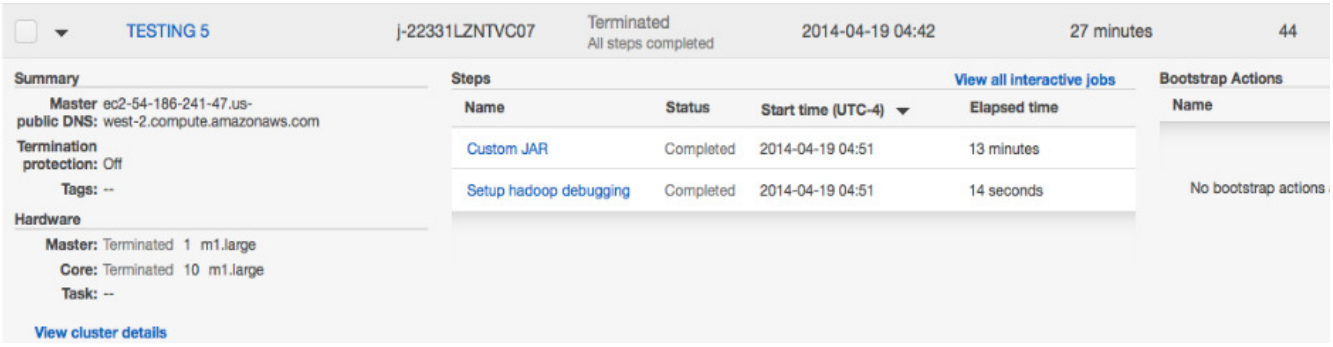
Machines used:
master : m1.large **1**
slaves : m1.large **10**

Dataset Size	Time taken	#Users processed	Accuracy
93MB	36mins	2,668,675	88.80%
68MB	32mins	1,948,675	88.78%
25MB	13mins	720,000	88.62%

SCREEN SHOTS





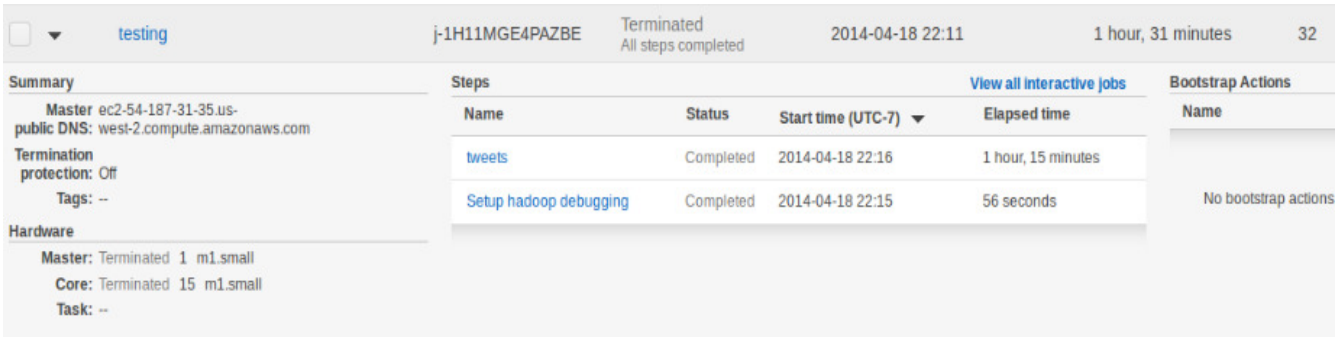


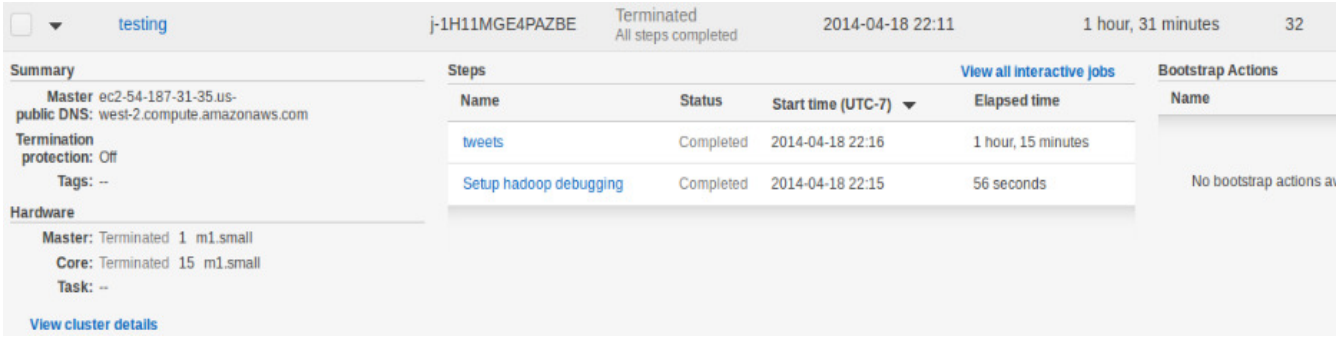
Number of buckets in histogram = **10**

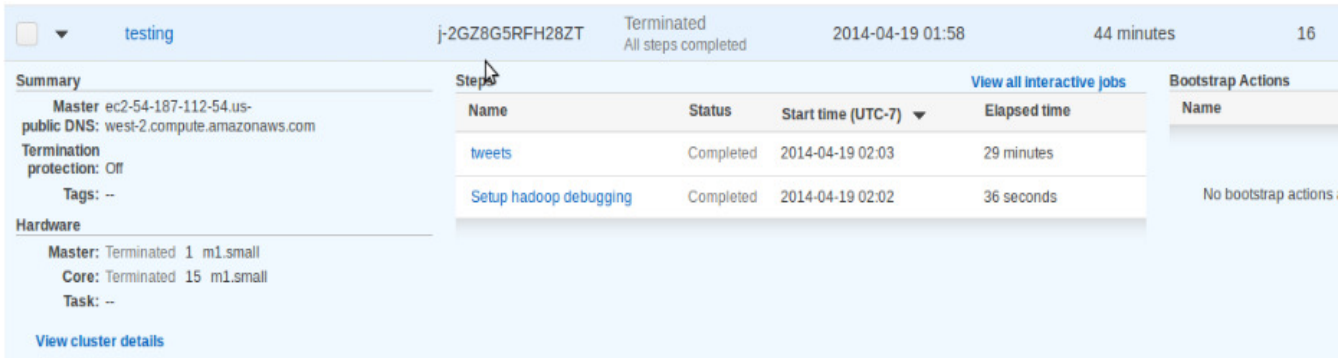
<u>Machines used:</u>
master : m1.small **1**
slaves : m1.small **15**

Dataset Size	Time taken	#Users processed	Accuracy
93MB	75mins	2,668,675	87.96%
68MB	77mins	1,948,675	87.82%
25MB	29mins	720,000	87.76%

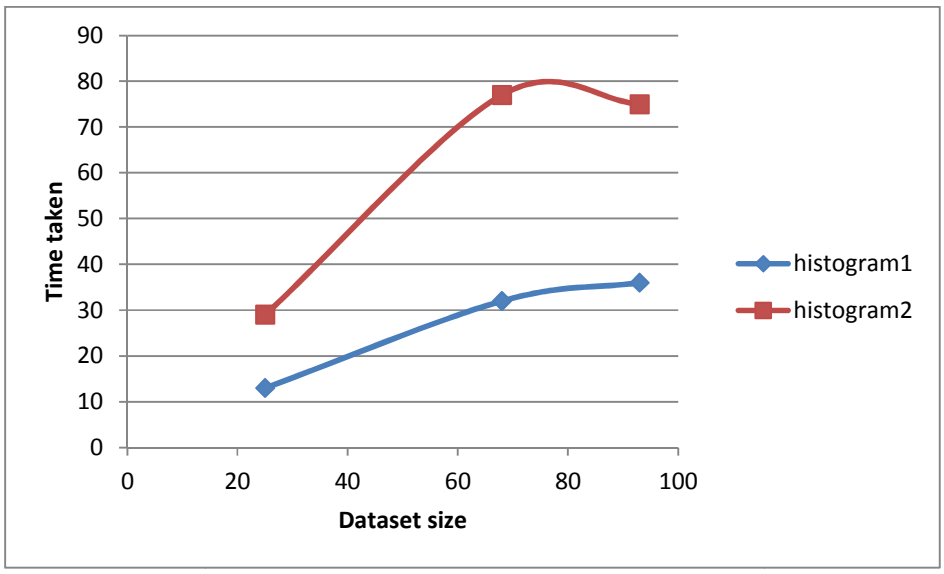
<u>SCREEN SHOTS</u>







3.3.13. Twitter data analysis



As we can empirically find in the graph above, the execution time increases almost linearly with the size of dataset, which is important for a parallel version of Naive Bayes.
As we know that MapReduce jobs are to be run on large dataset or data with large number of records, we haven't considered very small datasets or lesser number of records as the time taken for communication and assigning tasks is almost the same as the executing time of the

algorithm. Our Twitter dataset is very dense as we have pre processed it to be in the CSV format. Hence a file that is 1MB large would consist of around 10,000 records.
Since this model proves to work well and efficient with larger datasets it reveals promising scalability.

Additionally we observe that the accuracy is not really affected by either the type of machines or by the number of machines, which is as expected as the histogram built must be robust and independent of these factors.

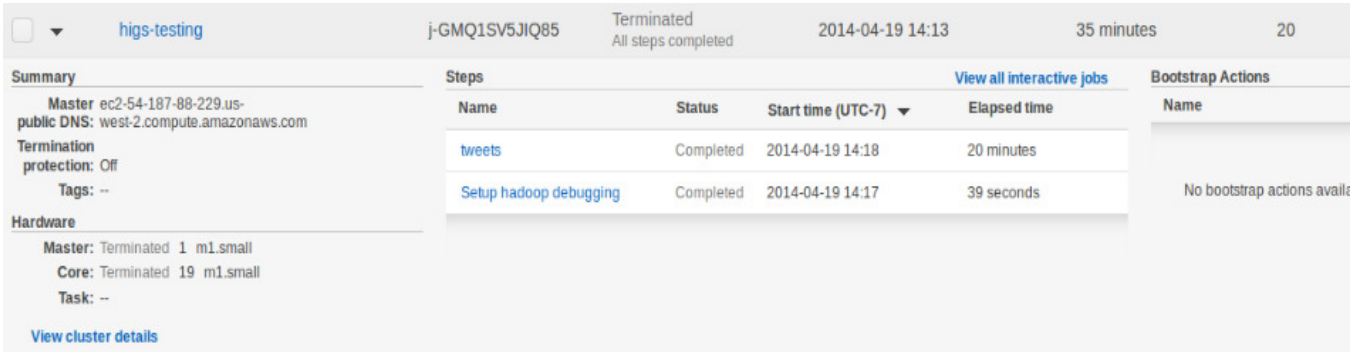
3.3.14. Higgs Dataset

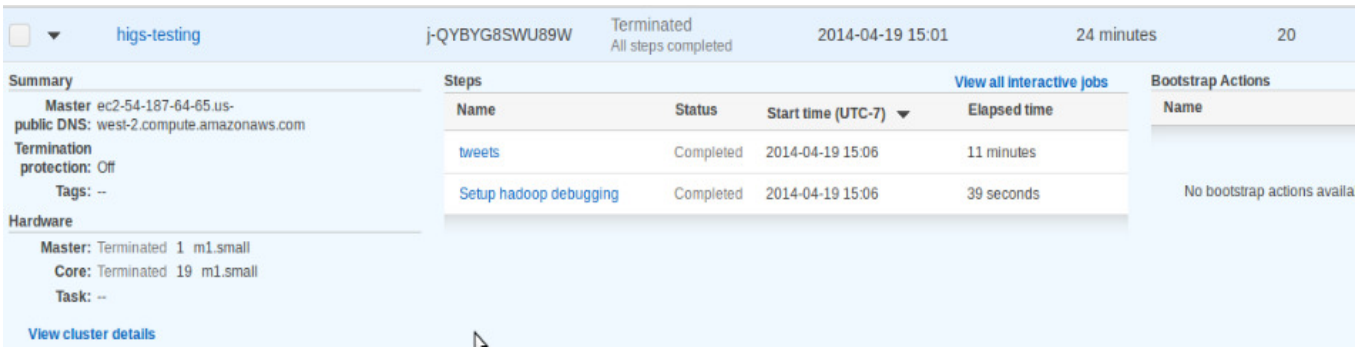
Number of buckets in histogram = 5

Machines used:
master : m1.small 1
slaves : m1.small 19

Dataset Size	Time taken	#Records processed	Accuracy
1400MB	20mins	2,000,220	58.36%
700MB	11mins	1,000,110	58.35%

SCREEN SHOTS



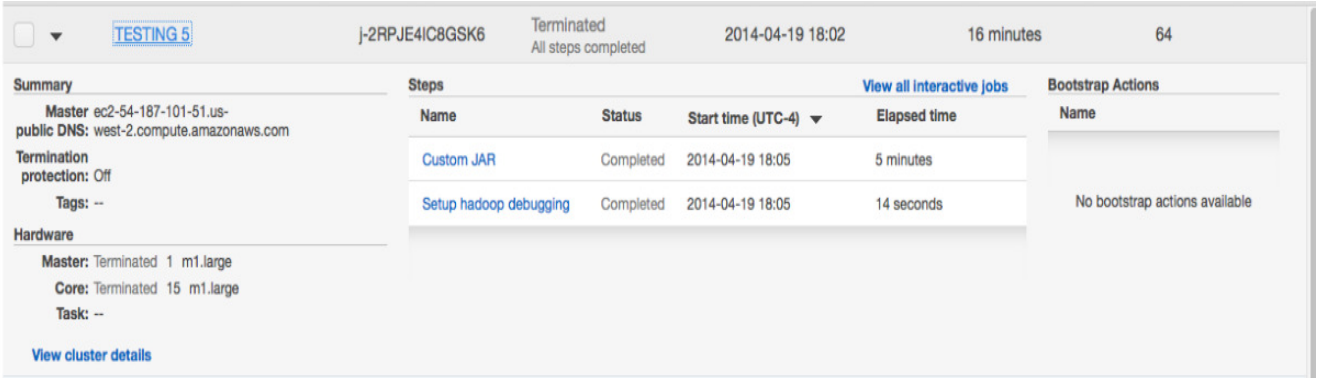


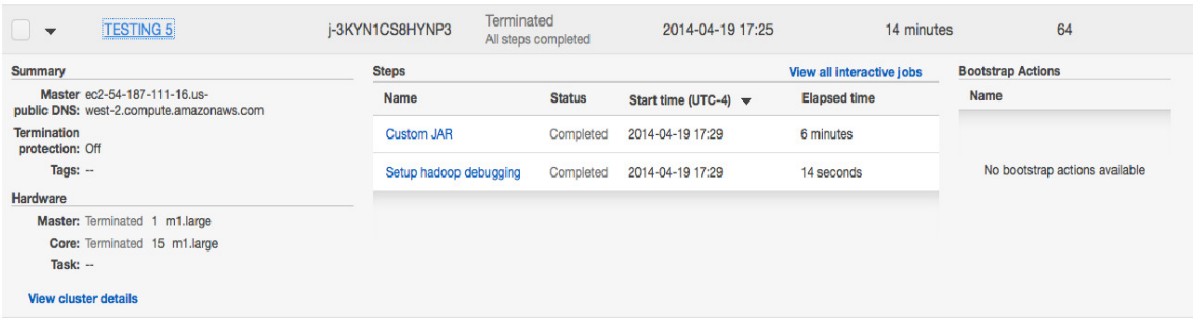
Number of buckets in histogram = 10

Machines used:
master : m1.large 1
slaves : m1.large 15

Dataset Size	Time taken	# Records processed	Accuracy
1400MB	6mins	2,000,220	53.68%
700MB	5mins	1,000,110	53.70%

SCREEN SHOTS





3.5.K-Means

***k*-means clustering** is a method generally used to classify semi-structured or unstructured datasets. This is one of the most commonly and effective methods to classify data because of its simplicity and ability to handle voluminous data sets. Given a set of observations ($\mathbf{x}1, \mathbf{x}2, \ldots, \mathbf{x}n$), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into k sets ($k \leq n$) $\mathbf{S} = \{S1, S2, \ldots, Sk\}$ so as to minimize the within-cluster sum of squares (WCSS):

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

where μi is the mean of points in Si.

It accepts the number of clusters and the initial set of centroids as parameters. The distance of each item in the data set is calculated with each of the centroids of the respective cluster. The item is then assigned to the cluster with which the distance of the item is the least. The centroid of the cluster to

which the item was assigned is recalculated. The process is repeated to obtain minimal distance between centroid of the cluster and item in the dataset.

Kmeans clustering using mapreduce:
Our implementation in mapreduce maintains two files- one file that contains the centroids and another file which contains the datapoints to be clustered.

The initial set of clusters are stored in input directory of HDFS and each centroid is represented as the 'key'. The mapper routine computes the distance between the datapoint and the centroid and the list of all vectors to which the centroid is the closest in terms of distance.

The recalculation(or restructuring) is done in the reduce subroutine. The centroid of each particular cluster is recalculated. The new set of centroids are updated and rewritten to HDFS.

Kmeans using mapreduce for our implementation: Clustering similarity of users on twitter The user data can be clustered based on all dimensions of users on twitter using distance measures as explained above. The initial centroids for clustering were chosen by random sampling. The results are as shown below.

```
Map:
 centers ← load from file
 minDist ← Double.MAXVALUE
 index ← -1
 for every center in centers do
      distance ← getDistance(datapoint, center)
     if distance < minDist{
       minDist ← distance
       index ← i
     }
 end for
 emit ( index, instance)
```

```
Reduce:
new listOfClusters
sum ← 0
count ← 0
for each value in values
   sum + = value
   ++count
newCentroid = sum/count
emit(newCentroid, value)
Driver:
If initial_iteration
  load cluster from file
else
 while(!stopping criteria)
  load cluster from previous iteration
  Create new JOB
  Set MAPPER
  Set REDUCER
  Submit JOB
```

Experiment and evaluation:

Experimented for values k = {3,4,5..15}. The following are the observations.

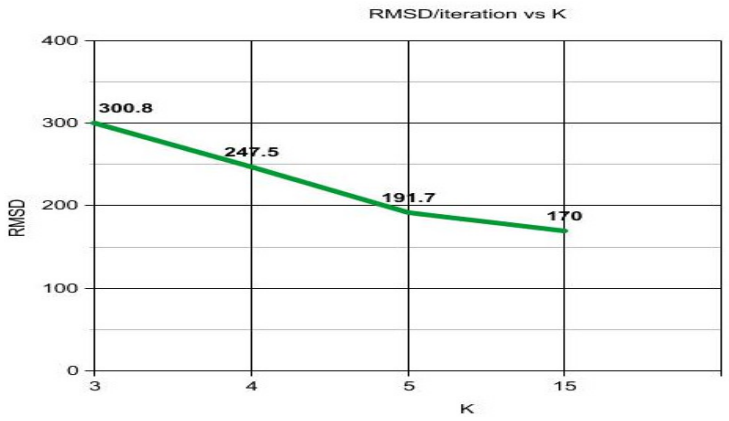
K	#Machines	Machine type	Time(mins)	Number of datapoints per cluster (in millions)	RMSD
3	6	M1 small	48	1.5 0.2 1	300.8
3	12	M1 small	32	1.5 0.2 1	
4	6	M1 small	47	1.1 0.8 0.6 0.2	247.5
4	12	M1 small	30	1.1 0.8 0.6 0.2	
5	6	M1 small	49	0.8 0.5 0.4 0.6 0.4	**191.7**
5	12	M1 small	33	0.8 0.5 0.4 0.6 0.4	
15	6	M1 small	47	The distribution is descent and has almost similar RMSD values as k = 5	**170.0**

RMSD vs K :

Since we store the intermediate set of centroids after every job run, we can compute the number the difference in centroid allocation after nth iteration for a given k. This allows us to find the RMSD for every iteration. The elbow method can be adopted to find the best possible variations after n successive iterations. As shown in the graph below, the RMSD for iterations does not dip heavily from k = 5 and k = 15, hence we can conclude a point between 5 and 15 as elbow an elbow point for the given subset of data.

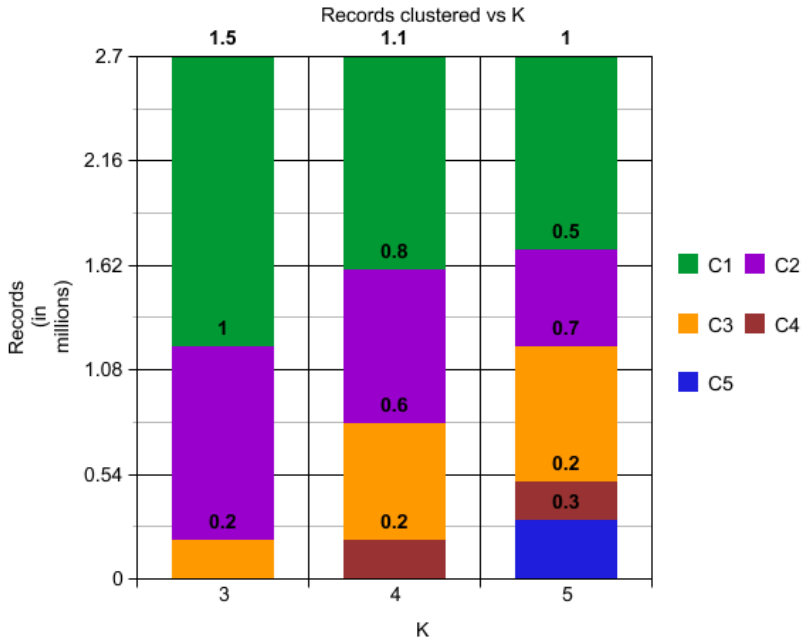
$$\mathrm{RMSD} = \sqrt{\frac{\sum_{t=1}^{n}(\hat{y}_t - y_t)^2}{n}}.$$

For every iteration, the rmsd values are calculated as the square root of the differences between centroids created for every attribute in the data point. When we plot k vs mean RMSD for every iteration, we see that the **elbow measure** clustering falls between 5 and 15.



Distribution of datapoints after n iterations:

Keeping the data constant, and by changing the value of k, the following results were observed. As we can see, the datapoints are clustered better when we choose k value greater than 5 and then the clusters remain consistent.



Conclusions:

K means clustering has a huge dependency on the value of 'k'. We could observe unrelated values for a few badly picked initial centroids. The results might sometimes get way too generalized to be effective. However, the simple nature of the algorithm makes it easy to implement using mapreduce.

The centroids obtained for a small chunk of data(ran locally) was significantly different when we run on the entire dataset of 2.7 million users. The data we have collected has users who have tweeted over a period of weeks, so the similarity between users need not be entirely true for a larger dataset.

4. References

[1]http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/36296.pdf
[2]http://www.stanford.edu/class/cs246/slides/14-dt.pdf
[3]http://www.ccs.neu.edu/home/vip/teach/MLcourse/lectures/mitchell_decision_tree.pdf
[4]http://www.ccs.neu.edu/home/vip/teach/MLcourse/lectures/decision_tree.pdf
[5]http://link.springer.com/chapter/10.1007%2F978-3-642-30217-6_12
[6]http://archive.ics.uci.edu/ml/datasets/HIGGS
[7]http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-plan-input-distributed-cache.html
[8]https://en.wikipedia.org/wiki/Decision_tree_learning
[9] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6579448

Extensions:

We have taken 1 days extension.
Harish- 1day
Akshay- 1day
Kosha- 1day
Prashanth - 1day