OBJECTIVE

The Objective of our project is to use twitter data set to produce clusters. For getting dataset we are going to use python's tweepy module or Java's Twitter4j library. The dataset is going to be a single file with single tweet on each line.For clustering, we are using k-means algorithm. We wrote the Map-Reduce Job for the k-means clustering algorithm. To apply clustering algorithm on the data set, we need to convert twitter data into numeric feature vectors. We will use Apache Mahout to generate the tf-idf vectors.

PROCESS

1. To generate the tf-idf vectors using Apache Mahout -
    a. We have created a Java file called TwitterData.java. Compile and run this file.After this you will get a file list consisting of one file for each and every tweet.
    b. Create a directory in HDFS using - `hadoop fs -mkdir -p /DataMining/mahout`
    c. Use the command to upload the tweets files:
       `hadoop fs -put /home/shubham/Downloads/tweets /DataMining/mahout`
    d. Now we are going to generate mahout sequence files :

```
mahout seqdirectory \
-c UTF-8 \
-i /DataMining/mahout/tweets/ \
-o /DataMining/mahout/seqfiles
```

       Here tweets is the directory which contains one file for each tweet.

    e. Then we are going to generate tf-idf vectors from the sequence files we just generated.

```
mahout seq2sparse \
-i /DataMining/mahout/seqfiles/ \
-o /DataMining/mahout/vectors/ \
-ow -chunk 100 \
-x 50 \
-seq \
-ml 50 \
-n 2 \
-nv
```

       This uses the default analyzer and default TFIDF weighting, `-n 2` is good for cosine distance, which we are using , `-x 50` meaning that if a token appears in 50% of the docs it is considered a stop word, `-nv` to get named vectors.

Explaination -

tf–idf, short for **term frequency–inverse document frequency**, is a numerical statistic that is intended to reflect how important a word is to a document in a collection.
It consist of two terms, term frequency and inverse document frequency. Here is a paragraph from Wikipedia which explains the meaning and role of the two terms.

Term frequency
Suppose we have a set of English text documents and wish to determine which

document is most relevant to the query "the brown cow". A simple way to start out is by eliminating documents that do not contain all three words "the", "brown", and "cow", but this still leaves many documents. To further distinguish them, we might count the number of times each term occurs in each document and sum them all together; the number of times a term occurs in a document is called its *term frequency*.

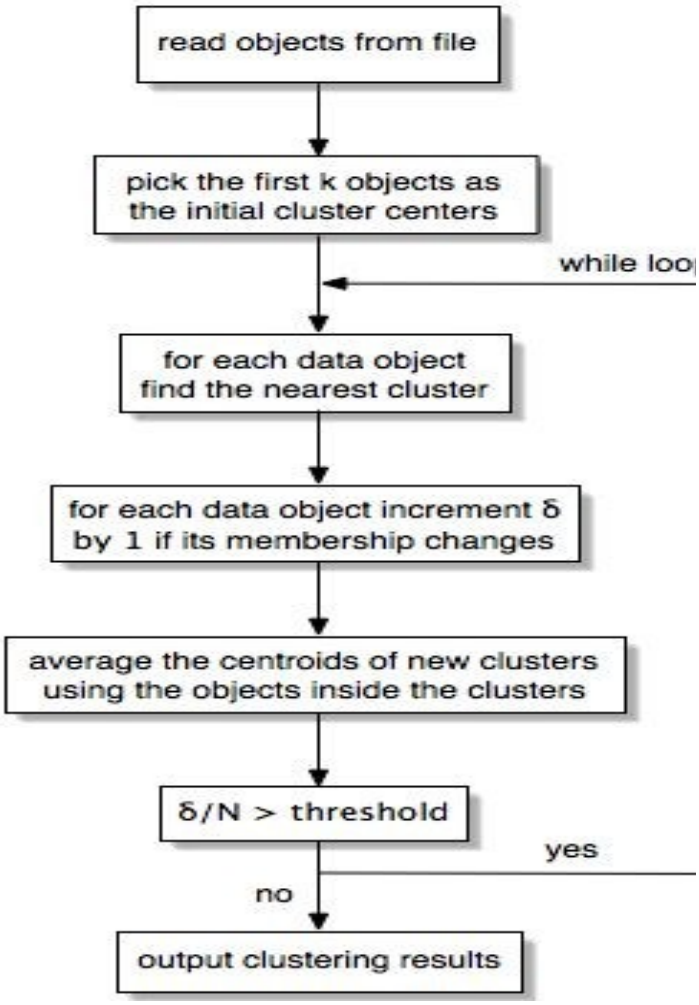
Inverse document frequency
However, because the term "the" is so common, this will tend to incorrectly t emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms "brown" and "cow". The term "the" is not a good keyword to distinguish relevant and non-relevant documents and terms, unlike the less common words "brown" and "cow". Hence an *inverse document frequency* factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely.

The tweetfile.txt and tfidf-vectors.txt are sample input and output files for giving a basic idea. During evaluation phase 3, we are going to use different files generated from tweepy module or Twitter4j.java.

2. Now we are going to apply the k-means algorithm to the tfidf file generated above.
There are two Java files which we are going to use for it. The Point.java and KmeansCluster.java.
Point class contains methods for calculating distance and sum for tfidf vectors. And the mapper, combiner and reducer class are in the KmeansCluster.java class.

Explaination :-

The K means clustering works in the following way as shown in the flowchart below :-



In MapReduce implementation we have three classes as follows :-

a. Mapper -

The Input is a point.
It compares the distance to all the centers and then find the nearest one.
Output is as follows -

key : centroid
value : point

b. Combiner -

It is the aggregator for the mapper result above, so the data to transfer between the nodes is less and thus decreases bandwidth.
The output for this file is :-

key : centroid
value : sum of point, count of the point for this key

c. Reducer -

Do the same work as Combiner but globally.
Generate new centroid from above step and compare it with the previous

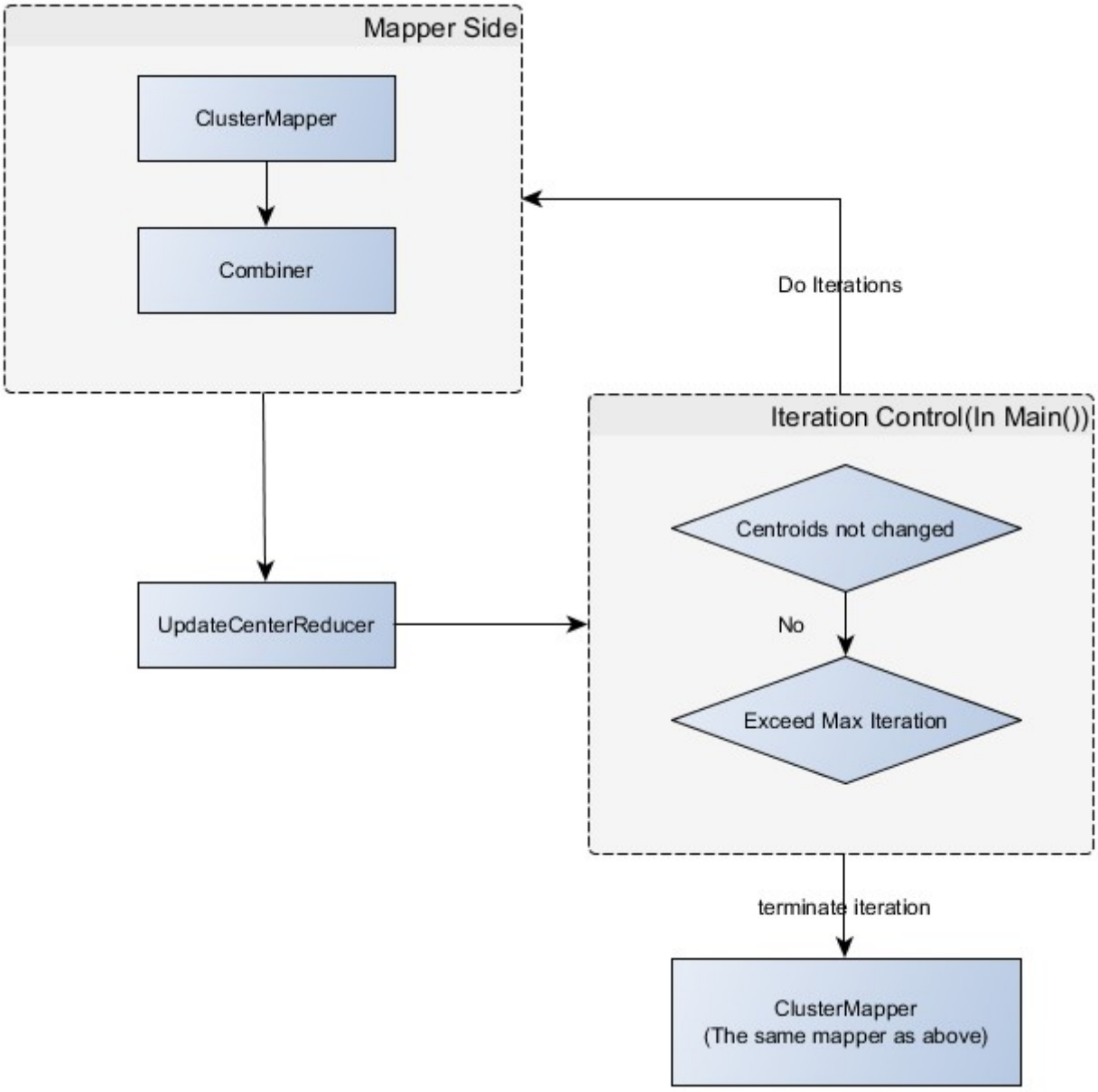
centroid.
Output for this is as follows:-

key : newCentroid

value : 1 if centroid is not changes or 0

The overall final output for this file is going to be new centroids. We can also get the clustered points with its centroids.

Its implementation in MapReduce framework is beautifully shown in the following flowchart :-



To get full details of how it works, see the code, it is well documented.

Detailed view -