K-Means Clustering

what is K-Means?

- given $K > 0$ (where K is the number of clusters)
- a set of N d-dimensional objects
  - Clustering is the process of grouping a set of N d-dimensional (2-dimensional, 3-dimensional, etc.) objects into K clusters of similar objects.
  - Objects should be similar to one another within the same cluster and dissimilar to those in other clusters.
- K-Means is a distance-based clustering algorithm.
- K-Means clustering (aka unsupervised learning) is a data mining algorithm to cluster, classify, or group your N objects based on their attributes or features into K number of groups (so-called clusters).

For example

- It can be used to find a group of consumers with common behaviors.

- To cluster documents based on the similarity of their contents

- Selection of K is specific to the application or problem domain.

K-Means algorithm

1. Partition *N* objects into *K* nonempty subsets.

2. Compute the centroids of the clusters in the current partition (the centroid is the center, or mean point, of the cluster). For all *i* = 1, 2, ..., *K*, compute ***μ****i* as:

$$\mu_i = \frac{1}{|c_i|} \sum_{j \in c_i} \underline{x}_j, \ \forall i$$

3. Assign each object to the cluster with the nearest centroid. This is simply attributing the closest cluster to each data point:

$$c_i = \{j : d(x_j, \mu_i) \le d(x_j, \mu_l), l \ne i, j = 1, \ldots, n\}$$

where *d*(*a, b*) is the distance function for two points: *a* and *b*.

4. Stop when there are no more new assignments. Otherwise, go back to step 2. Basically, we repeat steps 2 and 3 until convergence.

K-Means Distance Function

- Euclidean distance, Manhattan distance, Inner product space, Maximum norm, Your own custom function (any metric you define over the *d*-dimensional space)

Let:

$$X = (X_1, X_2, ..., X_d)$$

$$Y = (Y_1, Y_2, ..., Y_d)$$

then:

$$distance(X, Y) = \sqrt{(X_1 - Y_1)^2 + (X_2 - Y_2)^2 + \ . \ . \ . \ + (X_d - Y_d)^2}$$

The Euclidean distance function has some interesting properties:

- `distance(i,j)` $\geq$ `0`
- `distance(i,i) = 0`
- `distance(i,j) = distance(j,i)`
- `distance(i,j)` $\leq$ `distance(i,k)` + `distance(k,j)`

MapReduce Solution for K-Means Clustering

Example 12-1. K-Means clustering algorithm

```
// k = number of desired clusters
// delta = acceptable error for convergence
// data = input data
kmeans(k, delta, data) {
   // initialize the cluster centroids
   initial_centroids = pick(k, data);

   // this is how we broadcast centers to mappers
   writeToHDFS(initial_centroids);

   // iterate as long as necessary
   current_centroids = initial_centroids;
   while (true) {
        // theMapReduceJob() does 2 tasks:
        //      1. uses current_centroids in map()
        //      2. reduce() creates new_centroids and writes it to HDFS
        theMapReduceJob();
        new_centroids = readFromHDFS();
        if change(new_centroids, current_centroids) <= delta {
            // we are done, terminate loop-iteration
            break;
        }
        else {
            current_centroids = new_centroids;
        }
   }

   result = readFromHDFS();
   return result;
}
```

Example 12-2. K-Means clustering algorithm: change() method

```
change(new_centroids, current_centroids) {
   new_distance = [sum of squared distance in the new_centroids];
   current_distance = [sum of squared distance in the current_centroids];
   changed = absoulteValue(new_distance - current_distance);
   return changed;
}
```

MapReduce Solution: map()

- Read the cluster centroids into memory from a SequenceFile1 (note that we may also use Redis or memcached for persisting cluster centroids). In Hadoop's implementation, this will be done in the setup() method of the mapper class.
- Iterate over each cluster centroid for each input key-value pair. In Hadoop's implementation, for the map() function, the key is generated by Hadoop and ignored (not used).
- Compute the Euclidean distances and save the nearest center with the lowest distance to the input point (as a *d*-dimensional vector).
- Write the key-value pair to be consumed by reducers, where the key is the nearest cluster center to the input point (and the value is a *d*-dimensional vector). Both the key and the value are of the Vector data type.

MapReduce Solution: map()

Example 12-3. K-Means MapReduce: map() function

```
public class KmeansMapper ... {

   private List<Vector> centers = null;

   private List<Vector> readCentersFromSequenceFile() {
      // read cluster centroids from a SequenceFile,
      // which is a set of key-value pairs
      ...
   }

   // called once at the beginning of the map task
   public void setup(Context context) {
      this.centers = readCentersFromSequenceFile();
   }
    * @param key is MapReduce generated, ignored here
    * @param value is the d-dimensional Vector (V1, V2, ..., Vd)
    */
  map(Object key, Vector value) {
     Vector nearest = null;
     double nearestDistance = Double.MAX_VALUE;
     for (Vector center : centers) {
         double distance = EuclideanDistance.calculateDistance(center, value);
         if (nearest == null) {
             nearest = center;
             nearestDistance = distance;
         }
         else {
            if (nearestDistance > distance) {
                nearest = center;
                nearestDistance = distance;
            }
         }
     }

     // prepare key-value for reducers
     emit(nearest, value);
  }
```

MapReduce Solution: combine()

```
/**
 * @param key is the Centroid
 * @param values is a list of Vectors
 */
combine(Vector key, Iterable<Vector> values) {
  // all dimensions in sum Vector are initialized to 0.0
  Vector sum = new Vector();
  for (Vector value : values) {
     // note that value.length = d,
     // where d is the number of dimensions for input objects
     for (int i = 0; i < value.length; i++) {
        sum[i] += value[i];
     }

  }

  emit(key, sum);
}
```

MapReduce Solution: reduce()

- The main task of the reduce() function is to recenter.
- Each reducer iterates over each value vector and calculates the mean vector. Once you have found the mean, this is the new center; your final step is to save it into a persistent store (such as a SequenceFile).

```
/**
 * @param key is the Centroid
 * @param values is a list of Vectors
 */
reduce(Vector key, Iterable<Vector> values) {
  // all dimensions in newCenter are initialized to 0.0
  Vector newCenter = new Vector();
  int count = 0;
  for (Vector value : values) {
     count++;
     for (int i = 0; i < value.length; i++) {
        newCenter[i] += value[i];
     }
  }

  for (int i = 0; i < key.length; i++) {
     // set new mean for each dimension
     newCenter[i] = newCenter[i] / count;
  }

  emit(key.ID, newCenter);
}
```