CS 246: Problem Set 2

Tony Hyun Kim

February 6, 2012

1 Recommendation systems

1.1 Similarity Matrix

We can consider the cosine similarity to be a dot product between two normalized vectors $u/\,||u||$ and $v/\,||v||$.

Note that the items are represented as the columns of R. The norm squared of each column is given by the diagonal entries of Q. Hence, the normalized version of R (with respect to the items) may be written $R' = R \cdot Q^{-1/2}$.

Hence, the item similarity matrix S_I can be found as:

$$S_I = R'^T \cdot R' = Q^{-1/2} \cdot (R^T R) \cdot Q^{-1/2},$$

where we have made use of the symmetry of $Q^{-1/2}$.

Likewise, the users are represented as the rows of R. The norm square of each row is given by the diagonal entries of P. The normalized version of R (this time with respect to the users) is $R'' = P^{-1/2}R$. Hence:

$$S_U = R'' \cdot R''^T = P^{-1/2} \cdot (RR^T) \cdot P^{-1/2}.$$

1.2 Recommendation matrix

The collaborative filtering recommendation for user u and item s is defined as:

$$\begin{aligned} \text{User} - \text{user}: \qquad r_{u,s} &= \sum_{x \in \text{Users}} \cos(u,x) \cdot R(x,s), \\ \text{Item} - \text{item}: \qquad r_{u,s} &= \sum_{x \in \text{Items}} R(u,x) \cdot \cos(x,s). \end{aligned}$$

It is clear that the above definitions are equivalent to that of matrix multiplication. Therefore,

$$\Gamma_{\text{User}-\text{User}} = S_{\text{U}} \cdot R = P^{-1/2} \cdot (RR^T) \cdot P^{-1/2} \cdot R$$

and

$$\Gamma_{\text{Item}-\text{Item}} = R \cdot S_{\text{I}} = R \cdot Q^{-1/2} \cdot (R^T R) \cdot Q^{-1/2}.$$

1.3 Graphical representation of the recommendation matrix

The entries of the recommendation matrix $\Gamma_{u,s}$ represents a (weighted) sum over the indirect paths between user u and item s of the form:

$$u \leftrightarrow s' \leftrightarrow u' \leftrightarrow s$$

where u and u' are users, and s and s' are items.

To arrive at this interpretation, consider the expression for user-user collaborative filtering (the interpretation is the same for item-item CF, though the "weights" of each path will be different):

$$r_{u,s} = \sum_{u' \in \text{Users}} \cos(u, u') \cdot R(u', s). \tag{1}$$

Fix u and s. Then Eq. 1 is a sum that involves terms of the form $\cos(u, u') \cdot R(u', s)$ where

- The latter term $R(u', s)$ is nonzero only when there is a direct connection between u' and s.
- The cosine similarity $\cos(u, u')$ is nonzero when users u and u' share some items s' in common.

Hence, the product is nonzero when there exists a path $u \leftrightarrow s' \leftrightarrow u' \leftrightarrow s$ between user u and item s. Note also that Eq. 1 also involves a term of the form $\cos(u, u) \cdot R(u, s) = R(u, s)$ which adds a value of 1 to $r_{u,s}$ if there exists a direct path between user u and item s.

1.4 TV show collaborative filtering

1.4.1 Top 5 user-user collaborative filtering results

```
Rank: Title (Score)
1: FOX 28 News at 10pm (908.480)
2: Family Guy (861.176)
3: 2009 NCAA Basketball Tournament (827.601)
4: NBC 4 at Eleven (784.782)
5: Two and a Half Men (757.601)
```

1.4.2 Top 5 movie-movie collaborative filtering results

```
Rank: Title (Score)
1: FOX 28 News at 10pm (31.365)
2: Family Guy (30.001)
3: NBC 4 at Eleven (29.397)
4: 2009 NCAA Basketball Tournament (29.227)
5: Access Hollywood (28.971)
```

1.4.3 Precision in top-k predictions for user-user and item-item collaborative filtering

Fig. 1 shows the graph of the precision at top-k. The collaborative filtering performance of some 40% precision is above the baseline of 20% that one would get by random guessing.

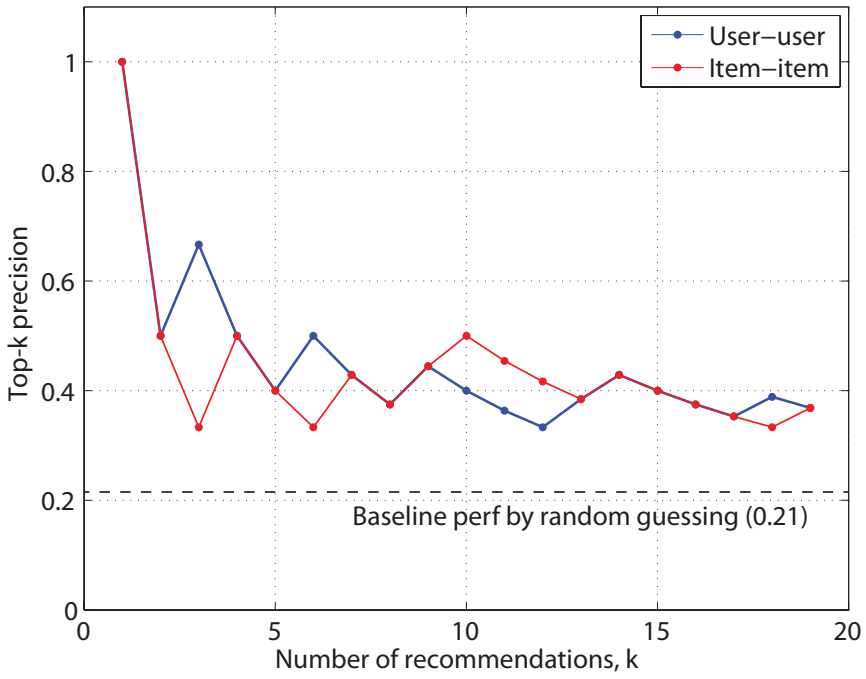

Figure 1: Precision in top-k predictions for Alex by user-user and item-item collaborative filtering. The CF performance exceeds the 21%-baseline that would be obtained by random guessing. Presumably, if we had implemented a richer rating system (*e.g.* with different levels of "like", etc.) then we could improve on this performance.

2 Singular value decomposition

2.1 Finding the SVD using AA^T or A^TA

2.1.1 Use AA^T or A^TA?

In this problem, the matrix A is taken to be 100×10000. Hence, AA^T is 100×100 while A^TA is 10000×10000. So, we'll take AA^T for sure!

2.1.2 Computing U and V

Since $A = USV^T$, we have

$$\begin{aligned} AA^T &= (USV^T) \cdot (VS^TU^T) = U(SS^T)U^T, \text{ and} \\ A^TA &= (VS^TU^T) \cdot (USV^T) = V(S^TS)V^T. \end{aligned}$$

Hence, U and V may be computed by diagonalizing AA^T and A^TA respectively.

2.1.3 Prove that $A^T u_i = \lambda_i v_i$

Let $x_i^{(m)}$ be the i-th standard basis vector in $\mathbb{R}^m$ and $x_i^{(n)}$ likewise in $\mathbb{R}^n$. We have

$$\begin{aligned} A^T u_i &= (VS^T U^T) \cdot u_i = VS^T (U^T \cdot u_i) \\ &= VS^T x_i^{(m)} = V(S^T x_i^{(m)}) \\ &= V\lambda_i x_i^{(n)} = \lambda_i (V x_i^{(n)}) \\ &= \lambda_i v_i \end{aligned}$$

2.1.4 Prove that U and V may be computed by decomposing AA^T only

Suppose we have decomposed AA^T. We have then computed U as well as the diagonal eigenvalue matrix Λ corresponding to AA^T.

Note that the singular values λ_i in S (which we may take to be positive without loss of generality) are related to the entries of Λ as: $\lambda_i = \sqrt{\Lambda_i}$. Hence, by decomposing AA^T, we have determined both U and S.

Then, by applying the results of Section 2.1.3, we may compute the corresponding components v_i from the known u_i's.

Remark: the vectors u_i and v_i corresponding to $\lambda_i = 0$ may be arbitrarily paired without affecting the SVD decomposition result.

2.1.5 Prove that U and V may be computed by decomposing $A^T A$ only

In Section 2.1.4, reorder the roles of u_i and v_i and replace AA^T with $A^T A$. The proof is identical.

2.2 Finding the SVD using M

2.2.1 Relationship between M and M^T

We have

$$M^T = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}^T = \begin{bmatrix} 0 & (A)^T \\ (A^T)^T & 0 \end{bmatrix} = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} = M.$$

2.2.2 Eigenvectors of M

Let u_i and v_i be corresponding columns in U and V. Then, u_i and v_i satisfy the relation proven in Section 2.1.3: $A^T u_i = \lambda_i v_i$ (and likewise $A v_i = \lambda_i u_i$).

Consider $\begin{bmatrix} v_i & -u_i \end{bmatrix}^T$, we have

$$M \cdot \begin{bmatrix} v_i \\ -u_i \end{bmatrix} = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} \cdot \begin{bmatrix} v_i \\ -u_i \end{bmatrix} = \begin{bmatrix} -A^T u_i \\ A v_i \end{bmatrix} = \begin{bmatrix} -\lambda_i v_i \\ \lambda_i u_i \end{bmatrix} = -\lambda_i \begin{bmatrix} v_i \\ -u_i \end{bmatrix}$$

and likewise

$$M \cdot \begin{bmatrix} v_i \\ u_i \end{bmatrix} = \begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} \cdot \begin{bmatrix} v_i \\ u_i \end{bmatrix} = \begin{bmatrix} A^T u_i \\ A v_i \end{bmatrix} = \begin{bmatrix} \lambda_i v_i \\ \lambda_i u_i \end{bmatrix} = \lambda_i \begin{bmatrix} v_i \\ u_i \end{bmatrix}.$$

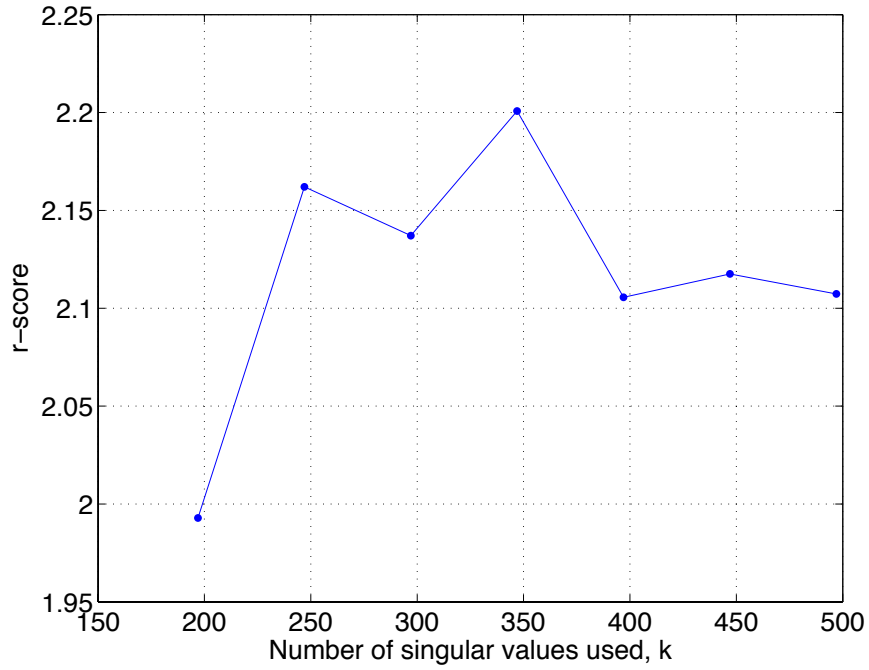

Figure 2: The r-score showing the mean similiarity of the "baseball documents" compared to the entire dataset as a function of SVD compression. Here, k denotes the number of singular values preserved in the compression, and $k = 497$ denotes the uncompressed matrix.

Hence, the vectors $\begin{bmatrix} v_i & -u_i \end{bmatrix}^T$ and $\begin{bmatrix} v_i & u_i \end{bmatrix}^T$ are eigenvectors of M with eigenvalues $-\lambda_i$ and λ_i respectively.

2.2.3 Prove that the eigenvalues are $-\lambda_i$ and λ_i

The eigenvalues were derived in the previous section. Furthermore, we have shown that M is symmetric and hence may be diagonalized. This way, we can obtain the SVD of A: the vectors u_i and v_i are parsed from the eigenvectors of M, and the singular values correspond to the positive eigenvalues of M.

2.3 Document similarity using SVD

The variation of the r-score as a function of the compression parameter k is shown in Fig. 2. The Matlab code for computing the r-score is attached in the following page.

```
% Tony Hyun Kim
% CS 246, PS 2, Problem 2(c)
function driver

clear all;
source = 'Matrix.txt';
A0 = load(source);

[U S V] = svd(A0,'econ');

for k = [197 247 297 347 397 447 497]
% for k = fliplr(297:10:497)
    % Apply compression
    mask = (1:497)<=k; mask = mask';
    Sk = diag(mask.*diag(S));
    A = U*Sk*V';

    % Compute the cosine similarity for the baseball docs
    sum_baseball = 0;
    for i = 1:99
        for j = (i+1):100
            sum_baseball = sum_baseball + cosim(A(i,:)',A(j,:)');
        end
    end
    avg_baseball = sum_baseball/nchoosek(100,2);

    % Compute the similarity among all documents
    sum_all = 0;
    for i = 1:496
        for j = (i+1):497
            sum_all = sum_all + cosim(A(i,:)',A(j,:)');
        end
    end
    avg_all = sum_all/nchoosek(497,2);

    % The 'r' score
    r = avg_baseball/avg_all;
    fprintf('%3d %.4f\n',k,r); % Dump output
end

% Expects u and v to be column vectors
function s = cosim(u,v)

s = (u'*v)/...
    (sqrt(u'*u)*sqrt(v'*v));
```

3 Theory of k-means

3.1 An identity involving the center of mass

Let S be an arbitrary set of points with center of mass $c(S) = \left(\sum_{x \in S} x\right)/|S|$ and z be an arbitrary point. We wish to show that

$$\sum_{x \in S} ||x - z||^2 - \sum_{x \in S} ||x - c(S)||^2 = |S| \cdot ||c(S) - z||^2 . \tag{2}$$

Consider the LHS

$$\begin{aligned}
\sum_{x \in S} ||x - z||^2 - \sum_{x \in S} ||x - c(S)||^2 &= \sum_{x \in S} \left(||x - z||^2 - ||x - c(S)||^2 \right) \\
&= \sum_{x \in S} x^2 - 2xz + z^2 - x^2 + 2xc(S) - c(S)^2 \\
&= \sum_{x \in S} 2x \cdot (c(S) - z) + z^2 - c(S)^2 \\
&= |S| \cdot \left[2c(S) \cdot (c(S) - z) + z^2 - c(S)^2 \right] \\
&= |S| \cdot \left[2c(S)^2 - 2c(S) \cdot z + z^2 - c(S)^2 \right] \\
&= |S| \cdot \left[c(S)^2 - 2c(S) \cdot z + z^2 \right] = |S| \cdot ||c(S) - z||^2 .
\end{aligned}$$

In particular, this result shows that the vector that minimizes the norm squared distance to a set of points S is the center of mass $c(S)$.

3.2 Prove that each iteration of k-means decreases the cost

The cost function is

$$\phi(\mathcal{C}, \{C_i\}) = \sum_{x \in \mathcal{X}} \min_{c \in \mathcal{C}} ||x - c||^2 , \tag{3}$$

which, for a given dataset $\mathcal{X}$, can be considered a function of the position of the centers as well as the assignment of the data to those centers.

We consider the two steps of each k-means iteration (denoted step 2 and 3 in the problem set).

- In Step 2, the centers $c \in \mathcal{C}$ are considered to be fixed and the data $x \in \mathcal{X}$ is assigned to the nearest center. This is clearly optimizing Eq. 3 with respect to $\{C_i\}$ (with $\mathcal{C}$ fixed).
- In Step 3, the assignment of the data to the centers is fixed, and the centers are re-computed to be the center of mass of the corresponding group. This can be considered an optimization of Eq. 3 with respect to $\mathcal{C}$ (with $\{C_i\}$ fixed). Note that we can rewrite Eq. 3 as

  $$\phi(\mathcal{C}, \{C_i\}) = \sum_{x \in C_1} ||x - c_1||^2 + \sum_{x \in C_2} ||x - c_2||^2 + \cdots + \sum_{x \in C_k} ||x - c_k||^2$$

  where each term takes the form that we analyzed in Section 3.1. Hence we are indeed optimizing ϕ with respect to $\mathcal{C}$ by assigning $c_i = c(C_i)$.

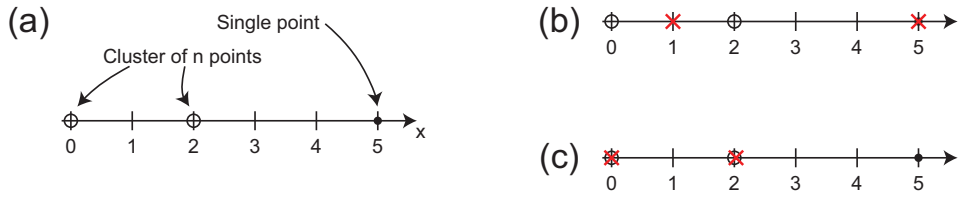

Figure 3: Demonstration of unfortunate k-means convergence due to bad initialization of centroids. (a) The dataset consists of $2n+1$ points with n overlapping points at $x=0$, n points at $x=2$, and a single point at $x=5$. We assume $n \gg 1$. (b) Stationary clusters (red crossmarks) where the left cluster owns $2n$ points at $x=0,2$, and the right cluster is assigned the single point at $x=5$. (c) A better solution where the two clusters are assigned to $x=0$ and $x \approx 2$.

3.3 Convergence of the cost function

We have shown that with each iteration of the k-means algorithm, the cost function ϕ must decrease (or stay the same). Now, the cost function is clearly bounded from below: for instance, we have the trivial result $\phi \geq 0$. It then follows that the cost function must converge.

3.4 Bad initialization example

Here is my pathological case that, with a bad initialization, may lead to a stationary set of clusters with a sum squared error (SSE) cost that is r times larger than the optimal cost.

Consider $2n+1$ data points in one dimension, laid out as shown in Fig. 3(a). There are n overlapping points at $x=0$, n points at $x=2$, and a single point at $x=5$. We assume $n \gg 1$. Fig. 3(b) shows a terrible stationary cluster assignment (red marks) where the left cluster is assigned the $2n$ points at $x=0,2$ and the right cluster corresponds to the single point at $x=5$. The corresponding SSE is $2n$. On the other hand, Fig. 3(c) shows a much better assignment where the two clusters own each of the n large clusters. (The right cluster is slightly displaced due to the point at $x=5$.) The SSE' is approximately $3^2=9$.

We now choose n such that SSE/SSE'$= 2n/9 > r$. The result is proven.

4 k-means on MapReduce

4.1 k-means implementation

The two steps per iteration of k-means maps rather perfectly to MapReduce. Basically, the mapper reads a chunk of data and, for each datum x, compute the index i of the nearest centroid c_i. The mapper then produces the key-value pair (i,x). Subsequently, the reducer re-calculates the center-of-mass of the points assigned to a particular center.

I used `DistributedCache` to initialize the list of centers for every mapper instance, and implemented a `VectorWritable` representation for the data. The code is attached.

```java
import java.io.*;

/*------------------------------------------------------------
 * Tony Hyun Kim
 * CS 246, PS 2, Problem 4
 * k-means on MapReduce
 *------------------------------------------------------------
 */
public class kmeans
{
    // Map assigns data to the k-centers
    public static class Map extends Mapper<LongWritable, Text, IntWritable, VectorWritable>
    {
        // Pulls data from DistributedCache into local memory
        ArrayList<VectorWritable> centers;
        void loadInitCenters(Path cachePath) throws IOException
        {
            BufferedReader cReader = new BufferedReader(new FileReader(cachePath.toString()));
            try
            {
                String line;
                this.centers = new ArrayList<VectorWritable>();
                while((line = cReader.readLine()) != null)
                    this.centers.add(new VectorWritable(line));
            }
            finally
            {
                cReader.close();
            }
        }
        void configure(Configuration conf)
        {
            try
            {
                Path[] cacheFiles = DistributedCache.getLocalCacheFiles(conf);
                if(cacheFiles != null && cacheFiles.length > 0)
                    for(Path cachePath : cacheFiles)
                        loadInitCenters(cachePath);
            }
            catch (IOException ioe)
            {
                System.err.println("IOException reading from distributed cache");
                System.err.println(ioe.toString());
            }
        }

        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
        {
            // Load the updated list of centers from the DistributedCache
            configure(context.getConfiguration());
```



```java

            // This is the input data point
            VectorWritable x = new VectorWritable(value.toString());

            // Let's find the center nearest to x
            int Ncenters = centers.size();
            int minIdx = 0;
            double minDist = x.minus(centers.get(minIdx)).magnitude();
            for(int i=1; i<Ncenters; i++)
            {
                double dist = x.minus(centers.get(i)).magnitude();
                if(dist<minDist)
                {
                    minIdx  = i;
                    minDist = dist;
                }
            }

            // Send datum to the new cluster
            context.write(new IntWritable(minIdx),x);

            // This is a special flag to compute the cost function
            context.write(new IntWritable(-1),new VectorWritable(minDist*minDist));
        }
    }

    // Reduce recomputes the center locations to be the respective center of mass
    public static class Reduce extends Reducer<IntWritable, VectorWritable, IntWritable, VectorWritable>
    {
        public void reduce(IntWritable key, Iterable<VectorWritable> values, Context context)
                throws IOException, InterruptedException
        {
            // Compute new center of mass
            Iterator<VectorWritable> iter = values.iterator();
            VectorWritable sum = new VectorWritable(iter.next());
            int count = 1;
            while(iter.hasNext())
            {
                sum.accumulate(iter.next());
                count++;
            }

            if(key.get() != -1) // Just a regular cluster
                context.write(null, sum.times(1.0/count));
            else // Compute the cost function
            {
                // We're going to write a 'cost.txt' directly to HDFS...
                String outDir  = FileOutputFormat.getOutputPath(context).toString();
                Path scorePath = new Path(outDir+"/cost.txt");
                FileSystem fs = FileSystem.get(context.getConfiguration());
```

```
113                 BufferedWriter writer = new BufferedWriter(new OutputStreamWriter
    (fs.create(scorePath,true)));
114                 writer.write(sum.toString()+"\n");
115                 writer.close();
116             }
117         }
118     }
119 
120     // THK: Trying out DistributedCache
121     //  Despite the excess initialization efforts, DC should be useful in the future
122     //  Adapted from: http://developer.yahoo.com/hadoop/tutorial/module5.html
123 
124     public static final String baseHdfsPath = "/user/cs246/";
125     public static String LOCAL_INITCENTERS_LIST = "/home/cs246/Desktop/dataset/
    c2.txt";
126 
127     public static void main(String[] args) throws Exception
128     {
129         Path inputPath = new Path(args[0]);
130         String outDir  = baseHdfsPath + args[1];
131         int numIter = Integer.valueOf(args[2]);
132 
133         for(int iter = 0; iter < numIter; iter++) // K-means iterations
134         {
135             Path centerPath = new Path(outDir+"/iter"+String.format("%03d",iter)+"/
    part-r-00000"); // Previous output
136             Path outputPath = new Path(outDir+"/iter"+String.format("%03d",iter+1));
137 
138             Configuration conf = new Configuration();
139 
140             if(iter==0) // On the first iteration, we'll copy the initial list of
    centroids
141             {
142                 FileSystem fs = FileSystem.get(conf);
143                 fs.copyFromLocalFile(false, true, new Path
    (LOCAL_INITCENTERS_LIST),centerPath);
144             }
145             DistributedCache.addCacheFile(centerPath.toUri(), conf);
146 
147             Job job = new Job(conf, "kmeans");
148             job.setJarByClass(kmeans.class);
149             job.setOutputKeyClass(IntWritable.class);
150             job.setOutputValueClass(VectorWritable.class);
151 
152             job.setMapperClass(Map.class);
153             job.setReducerClass(Reduce.class);
154 
155             job.setInputFormatClass(TextInputFormat.class);
156             job.setOutputFormatClass(TextOutputFormat.class);
157 
158             FileInputFormat.addInputPath(job, inputPath); // Input is always the
    dataset
159             FileOutputFormat.setOutputPath(job, outputPath);
```



```
160 
161             job.waitForCompletion(true);
162         }
163     }
164 }
165 
```

4.2 Computing the cost function

As suggested, I compute the cost function within the same MapReduce job. I achieve this with the following trick: Each mapper, in addition to assigning the datum x to a centroid index i with the key-value pair (i, x), will also emit the norm squared distance to a special reducer as follows $(-1, ||x - c_i||^2)$.

A reducer instance can then determine whether it is responsible for the cost function by the flag key $= -1$. This specialized reducer will then write the accumulated score directly to HDFS. Note that for a single MapReduce run only a single reducer (and its repeated instantiations) will attempt the direct write to HDFS, making write contention even in a large-scale run unlikely.

4.3 Comparison of the two initialization strategies

Fig. 4 shows the decrease in the cost function ϕ value with each iteration of k-means. I have normalized the cost function by the maximum observed cost. It is clear that the random initialization strategy (`c1.txt`) performs far worse than the initialization scheme where the centroids were separated as far as possible (`c2.txt`).

After 10 iterations, the cost function corresponding to `c1.txt` has decreased to 74% of its initial value. At the same iteration depth, the path corresponding to `c2.txt` has decreased to 25% of its initial value. Comparing `c1` and `c2` against each other, we find that after 10 iterations, the random initialization yields a SSE error that is about 320% higher than that of the distant initialization.

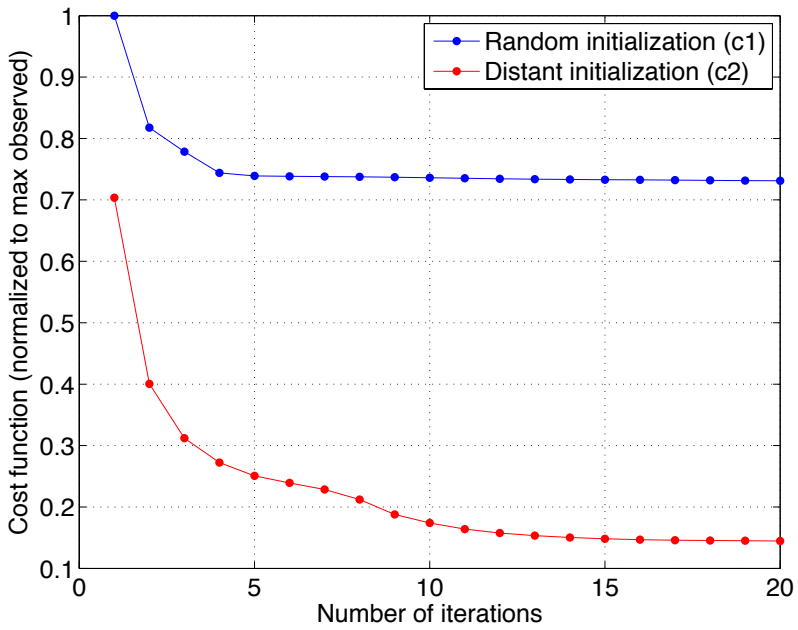

Figure 4: Decrease in the cost function ϕ (Eq. 3) with iterations of k-means. Values have been normalized by the maximum observed cost. The maximum-distance initialization of the centroids clearly outperforms random initialization.