Examination of Initialization Techniques for Nonnegative Matrix Factorization

John Frederic

EXAMINATION OF INITIALIZATION TECHNIQUES FOR NONNEGATIVE MATRIX FACTORIZATION

by

JOHN FREDERIC

Under the Direction of Marina Arav

ABSTRACT

While much research has been done regarding different Nonnegative Matrix Factorization (NMF) algorithms, less time has been spent looking at initialization techniques. In this thesis, four different initializations are considered. After a brief discussion of NMF, the four initializations are described and each one is independently examined, followed by a comparison of the techniques. Next, each initialization's performance is investigated with respect to the changes in the size of the data set. Finally, a method by which smaller data sets may be used to determine how to treat larger data sets is examined.

INDEX WORDS: Nonnegative matrix factorization, Initialization, Spherical K-means, Compression ratio, Percent error, Random Acol, Random C

EXAMINATION OF INITIALIZATION TECHNIQUES FOR NONNEGATIVE MATRIX FACTORIZATION

by

JOHN FREDERIC

A Thesis Presented in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in College of Arts and Sciences

Georgia State University

2008

EXAMINATION OF INITIALIZATION TECHNIQUES FOR NONNEGATIVE MATRIX FACTORIZATION

by

JOHN FREDERIC

ACKNOWLEDGEMENTS

The author wishes to thank Dr. Marina Arav for her assistance throughout his entire Master's program. I would also like to acknowledge the administration and staff of American Heritage Academy for their constant support. Finally, thanks to my friends and family who helped me maintain my motivation.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

With every advance in technology comes huge growth in the availability of information. Unfortunately, those same advancements do not always allow for storage capacity of that information. As a result, much work has been done to devise ways of compressing information to decrease the demand on storage capacity or to improve performance when processing the data. Techniques such as Principal Component Analysis, Singular Value Decomposition, and QR-Factorization have been extensively studied. Unfortunately, these techniques may cause a loss of some subtle, underlying information in the data. Most notably, much of the data generated today is inherently nonnegative due to physical reality. The compression techniques above often generate compressed data containing negative values which have no real-world significance. So what is lost?

To answer that, we look at a motivating example [5]. Today, there are countless objects orbiting the earth. It is important to not only know where these objects are but also what they are. Sometimes, optical technology is not sufficient to identify small objects so we must rely on spectral analysis. A sensor sensitive to a wide range of EM radiation is pointed toward the unknown object and samples of reflected radiation are taken over time. In this manner, we may generate a matrix, A, whose columns correspond to different times and whose rows correspond to different spectral wavelengths. Thus the (i, j) entry in the matrix represents the amount of spectral energy reflected at wavelength i and time j. Assuming we take n time samples over m wavelengths, A will be an $m \times n$ array of data.

We know that the object is composed of different substances and each substance will contribute to the reflected spectral energy at a given wavelength. If we knew what the object was made of and what percentage of the total object was composed of each substance, we could determine what the spectrum would be by linearly combining the contributions of the substances weighted by their percentages. Let $S = [s_{ij}]$ be a matrix whose columns represent the individual substances and whose rows represent different wavelengths. Let $X = [x_{ij}]$ be a matrix whose columns represent time and whose rows represent the percentage present of each component substance. So we can interpret s_{ij} as the j^{th} substance's reflectivity at wavelength i and x_{ij} as the percentage of substance i present at time j. Then we can write that $A = SX + N$ where N is a noise vector.

However, we are usually presented with data matrix A and would like to determine S and X. Clearly, matrices S and X will be nonnegative. Using the factorization above may reduce our storage needs but may also produce an S and/or X that contain negative values. We would like to use a method that will preserve the inherent nonnegativity of the data. By preserving nonnegativity, we hope to be able to factor the spectral data into its individual parts. Recent research has produced such a method called Nonnegative Matrix Factorization (NMF).

2. Nonnegative Matrix Factorization

2.1 Definitions

Let $\mathbb{R}^{m\times n}$ be the set of all $m \times n$ matrices with real number entries. The matrix $X \in \mathbb{R}^{m\times n}$ is *nonnegative* if $x_{ij} \geq 0$ for all $1 \leq i \leq m,\ 1 \leq j \leq n$ where x_{ij} represents the element in the i^{th} row and j^{th} column. The NMF problem attempts to minimize the Frobenius norm of the error matrix. The *Frobenius norm* of a matrix $A = [a_{ij}]$, $1 \leq i \leq m,\ 1 \leq j \leq n$ is given by

$$\|A\|_F \equiv \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}. \tag{1}$$

Also, when discussing vectors, their lengths are often given as a type of vector norm called the *2-norm*. The 2-norm of a vector x is given by

$$\|x\|_2 \equiv \sqrt{\sum_{i=1}^{n} |x_i|^2}. \tag{2}$$

When discussing large data sets, *clustering* is often used to partition the data into smaller sets such that the members of each set share some similarity. With vectors, it is often desirable to cluster by location. The *centroid* of a set of vectors is just the average of all the vectors in the set. Thus, a large set of vectors may be clustered into smaller sets where each set has a centroid, and membership in a set is determined by a vector's distance from the centroid.

2.2 NMF Background

The NMF problem can be summarized as follows.

[NMF Problem] Let A be our $m \times n$ data matrix and let k be a positive integer such that $k < \min\{m, n\}$. Find a nonnegative $m \times k$ matrix S and a nonnegative

$k \times n$ matrix X that will minimize the functional

$$\frac{1}{2}\|A - SX\|_F^2. \tag{3}$$

A flurry of research began on this topic when Lee and Seung published their first algorithm for performing an NMF on real data [3]. Since then many variations of their original algorithm have been studied. A summary of much of the work done in this area is given by Berry et al. [1]. Lee and Seung have shown that their algorithm will produce monotonically non-increasing values in (3). Additionally, repeated and varied experiments have shown that the algorithm is consistently effective in practice. We will therefore use it as our NMF algorithm in the experiments to follow. The algorithm is given as follows:

Algorithm 1: Nonnegative Matrix Factorization

```
Given: $A \in \mathbb{R}^{m \times n}$ with $A \geq 0$ and $k > 0$ such that $k < \min\{m, n\}$
  1. Initialize $S \in \mathbb{R}^{m \times k}$ and $X \in \mathbb{R}^{k \times n}$ with random nonnegative values.
  2. Scale columns of $S$ to sum to one.
  3. Until a set number of iterations is reached do
     3a. $X_{dj} \leftarrow X_{dj} \frac{(S^T A)_{dj}}{(S^T SX)_{dj} + \varepsilon}, (1 \leq d \leq k, 1 \leq j \leq n)$
     3b. $S_{id} \leftarrow S_{id} \frac{(AX^T)_{id}}{(SXX^T)_{id} + \varepsilon}, (1 \leq i \leq m, 1 \leq d \leq k)$
     3c. Scale columns of $S$ to sum to one.
  4. End loop.
```

3. Initialization Techniques

3.1 Introduction

While there has been a very large amount of work done to refine and improve NMF algorithms, much less work has been done on how to initialize the factor matrices. Originally, the factor matrices were initialized with random nonnegative entries. However, researchers have since begun to study different ways of initializing the factors. Just as NMF can preserve some of the information inherent in the data, perhaps there is a way to extract information from the data matrix in order to more effectively initialize the factors. We now look at three basic techniques that have been studied so far.

3.2 Technique Descriptions

The first two techniques, *random Acol initialization* and *random C initialization*, were introduced by Langville et al. [2]. In *random Acol initialization*, each column of S is initialized by averaging p randomly chosen columns of A. This will help maintain any sparsity in A which would be lost with pure random initialization with dense vectors. This method is also very computationally inexpensive and easy to implement. *Random C initialization* is similar to *random Acol initialization* with one significant difference. In *random C initialization* we first select q of the longest (in the 2-norm sense) columns of A and then average p randomly chosen columns from the q longest in order to initialize each column of S. Thus, we end up using the densest vectors for initialization which are more likely to be near the centroid centers. This method is also fairly inexpensive and easy to implement and is summarized in Algorithm 2.

Algorithm 2: Random C Initialization

```
Given: $A \in \mathbb{R}^{m \times n}$ with $A \geq 0$
    1. Find $q$ of the longest (in the 2-norm sense) columns of $A$.
    2. For each column of $S$ do
        2a. Average $p$ randomly chosen columns out of the $q$ longest of $A$.
    3. End loop.
```

The third technique, *spherical k-means clustering*, was presented by Wild [7]. We summarize the technique here using notation consistent with [7]. The goal of this technique is to initialize the columns of S with the centroids of A. We wish to find k centroids $\{c_j\}$ that represent k disjoint subsets of the columns of A. Each subset contains all the vectors in A that are closest to their respective centroid. The centroid itself is calculated as the average of all the vectors in the subset.

To find the centroids, we need a way to determine the distance between two vectors. In order to assign equal weight to each vector, the column vectors of A are normalized to be of unit length in the Euclidean norm. In this way, the direction of each vector becomes the important characteristic. This normalization along with the nonnegativity of the data allow us to use the Cosine Similarity measure to compare vectors

$$\cos(\theta_{x,y}) = \|x\|_2 \, \|y\|_2 \cos(\theta_{x,y}) = x^T y, \tag{4}$$

where the first equality comes from the normalization of x and y, and the second equality is the standard definition of the inner product. Using the Cosine Similarity measure, values near one indicate vectors that are closer together. As long as we also normalize the centroids c_j, we may use this measure to find the centroids and their associated subsets. The algorithm is summarized as follows:

Algorithm 3: Spherical k-means Clustering

```
Given: $X \in \mathbb{R}^{m \times n}$ with $X \geq 0$
    1. Initialize $k$ centroids $c_j, 1 \leq j \leq k$.
    2. While the clusters change from $t$ to $t+1$ do
        2a. Compute $d_{ij}^{(t)} = x_i^T c_j^{(t)}$ $(1 \leq i \leq n, 1 \leq j \leq k)$.
        2b. Define the new partition of clusters
            $\pi_j^{(t+1)} = \left\{ x_i | j = \text{argmax}_l \left( d_{il}^{(t)} \right) \right\}$.
        2c. Recompute each centroid
            $c_j^{(t+1)} = \frac{\sum_{x_i \in \pi_j^{(t+1)}} x_i}{\left\| \sum_{x_i \in \pi_j^{(t+1)}} x_i \right\|}$.
    3. End loop.
```

4. Numerical Results

4.1 Data Sets

Ultimately, we would like to examine how the size of the data set affects the performance of each of the four initialization techniques: random, random Acol, random C, and spherical k-means. To begin we create 5 data matrices. Each matrix will have 70 rows and 100, 200, 400, 800, 1600 columns, respectively. As we are not considering any specific applications, each matrix was generated by filling each entry with a random number selected from a uniform distribution between 0 and 30. Thus, we can expect each matrix to be densely populated.

4.2 Comparison of the Four Techniques

We will first look at the performance of each technique independently by considering the effect of the parameter k. For this, we used the 70×100 matrix as our data set. For each run of the NMF algorithm we let the Frobenius norm of the error matrix, $\|A - SX\|_F$, be our performance criterion. We began by considering the effect of k on the performance by running the NMF algorithm for $k = 5, 25, 50$. The results are shown in Figure 1. All of the initialization techniques show similar results in both convergence time and overall error performance. Additionally, in all four cases, a decrease in k resulted in worse overall performance. This indicates that the more we try to reduce the rank of the factor matrices, the greater the data loss we will experience. Reduced rank factor matrices may not only reduce storage requirements, but may also increase the speed of post-processing of the data. For any application it will be necessary to consider what is an acceptable trade off between minimizing error and storage, or between minimizing error and data processing speed.

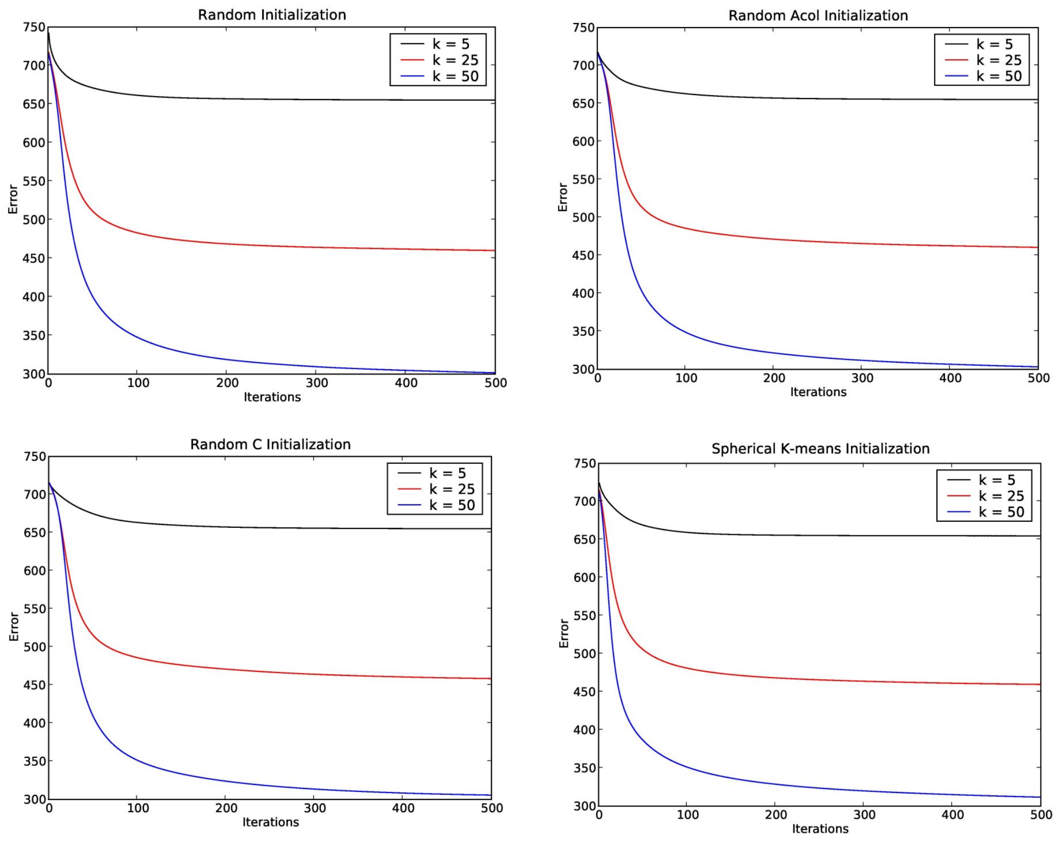

Figure 1: Frobenius error for all four initialization techniques for k=5, 25, 50

Random initialization as well as spherical k-means require only the selection of k. Random Acol and random C, however, introduce other parameters. For random Acol, we must choose how many columns of the data matrix to average for each initialization of a column of S. Using the same 70×100 data matrix as before, the NMF algorithm was run using random Acol for different values of p (the number of columns averaged) while holding k constant. For this we chose $k = 25$. Results indicated very similar performance for all values of p during the first 100 iterations when the approximation error experiences its greatest improvement. After 400-500 iterations, when the error changes very little, there were only small differences in performance between the three cases. Overall, it seems that the selection of p has a minor impact on algorithm performance. The results are shown in Figure 2.

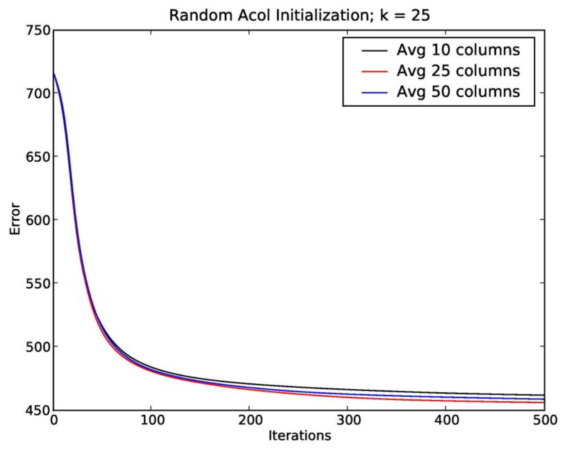

Figure 2: Random Acol initialization for $k = 25$ and $p = 10$, 25, and 50

Random C initialization also requires the choice of more than just k. In random C, we average columns of A to initialize S, but we only select from q of the longest (in the 2-norm sense) columns of A. Thus, we must select both p and q. Considering the previous results from random Acol that indicated the limited impact of p, we let p be a constant and we varied our choice of q. Again using the same 70×100 data matrix as before, we ran the NMF algorithm with $k = 25, p = 10$, and $q = 15$, 25, and 50. As with p, there was very little difference in performance between the three values of q. Very similar performance was seen at both the early iterations as well as the later iterations. Results can be seen in Figure 3.

It is also interesting to compare the performance of the various initialization techniques with each other. Figure 4 shows a comparison for $k = 25$. While all four techniques show similar performance, we can note small differences. Spherical k-means had the best performance initially while random Acol showed the best reduction in error by the final iteration. Overall, it seems all four techniques, at least with this data set, would be effective choices for an initialization technique. For specific applications, however, different techniques could show greater differences in performance.

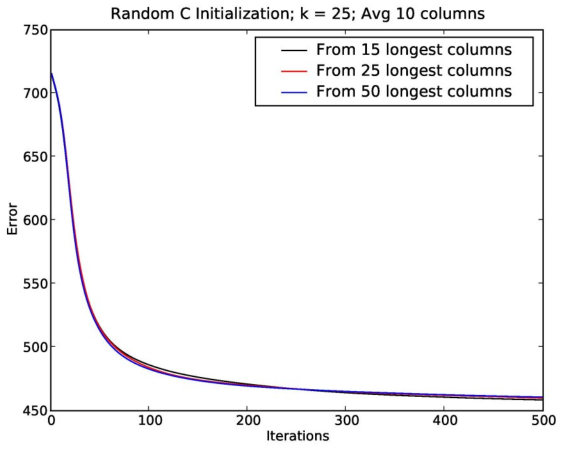

Figure 3: Random C initialization for $k = 25$, $p = 10$, and $q = 15$, 25, and 50

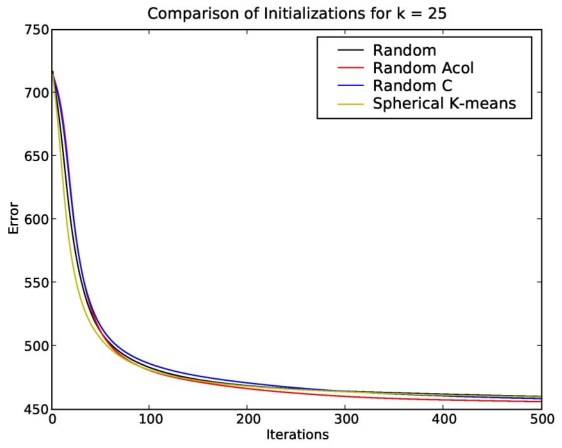

Figure 4: All four initializations with k = 25

4.3 Effect of Data Set Size

If we look at all four figures above, we see that the Frobenius norm of the error matrix ranges between 300 and 700. All results to this point were obtained by beginning with a 70×100 data matrix. By increasing the size of the data matrix, we increase the number of elements in the data matrix as well as the error matrix. As a natural consequence of having more elements, we expect the Frobenius norm of the error matrix to increase. In order to examine the effect of data set size, we introduce a new measure which we will call percent error. We define *percent error* as

$$\frac{\|A - SX\|_F}{\|A\|_F}, \tag{5}$$

which is simply the ratio of the Frobenius norm of the error matrix to the Frobenius norm of the data matrix. Essentially, we look at $\|\cdot\|_F$ as a measure of the amount of information in each respective matrix. Thus, it makes more sense to consider the amount of data lost relative to the amount of data present rather than strictly the magnitude of the error.

We ran 5 simulations for each initialization technique in which k was held constant at 25. For random Acol, 10 columns were averaged and for random C, 10 columns were averaged from the 15 longest. A simulation was conducted using data matrices with 70 rows and 100, 200, 400, 800, 1600 columns, respectively. The results, shown in Figure 5, were as expected. As the size of the data set increased, the percent error also increased. This is consistent with the data from Figure 1, because increasing data set size while holding k constant is very similar to holding the data set size constant while decreasing k. In both cases we are varying the amount of rank reduction we are attempting to accomplish. It is interesting to note that in all 4 cases, the difference in percent error between successive data set sizes decreased as data set size increased. The percent error change from 800 to 1600 columns was less than 1%, while it was approximately 4% from 100 to 200 columns. This may indicate an upper limit on the percent error vs. data set size.

This leads to the question of whether or not, given a set of requirements for percent error, we can effectively choose k based on data set size so as to met those requirements. To answer this we introduce another metric which we will call the

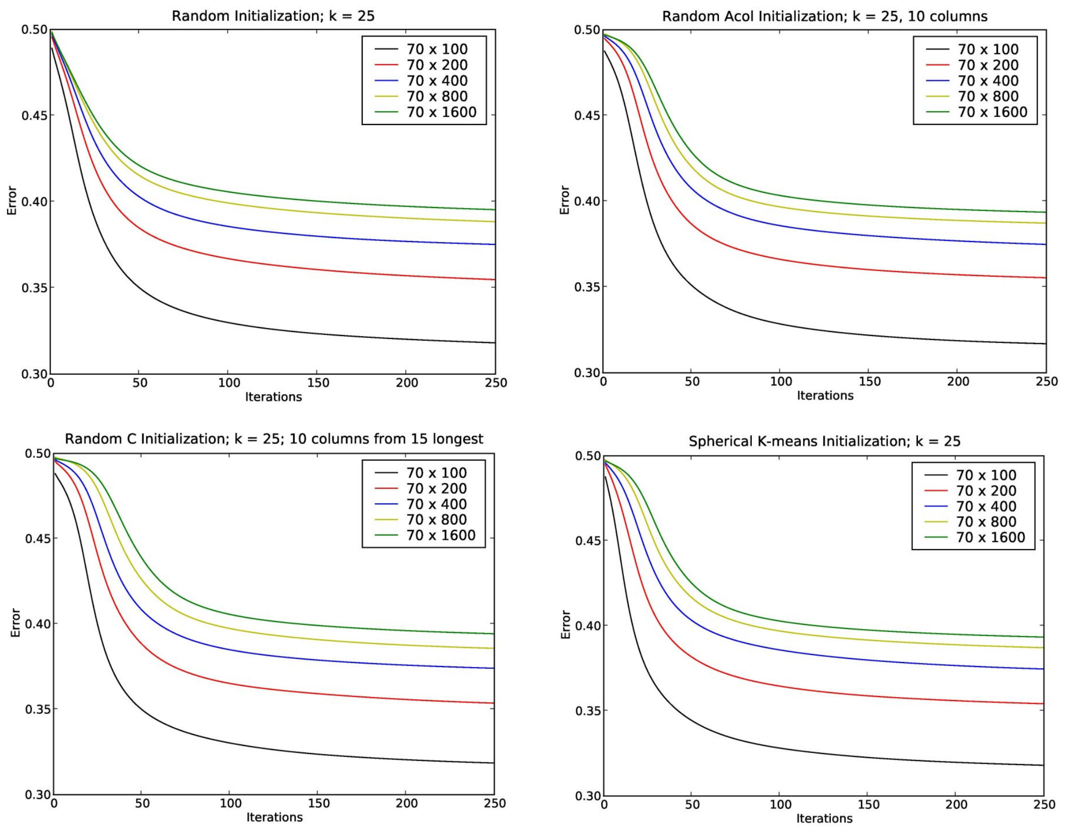

Figure 5: Percent error for each of the four initialization techniques vs. data set size.

compression ratio and define it to be

$$\frac{\text{size}(S) + \text{size}(X)}{\text{size}(A)}, \tag{6}$$

where *size(Z)* is the number of elements in matrix Z. If we assume that each element in a matrix will require 1 unit of storage space (or more generally that the size of the matrix is directly proportional to the storage space required), then (6) essentially computes the savings in storage space gained by the NMF process.

Using random Acol initialization and a 70×200 data matrix, we ran the NMF algorithm for $k = 29$, 32, and 35. The results, Figure 6, again showed that error increased as k decreased. More importantly, comparing Figure 6 with random Acol in Figure 5, the percent error for $k = 25$ and a 70×100 data matrix is very close to the percent error for $k = 32$ and a 70×200 data matrix. Both of these cases correspond to approximately the same compression ratio. This indicates that as data set size grows, if we choose k so as to maintain the same compression ratio as a smaller data set, we can maintain the same percent error seen in the smaller data set. Thus, if we have a very large data set which will require considerable processing time, we can experiment with different values of k for a much smaller data set size and then use the compression ratio to translate our k to the larger data set.

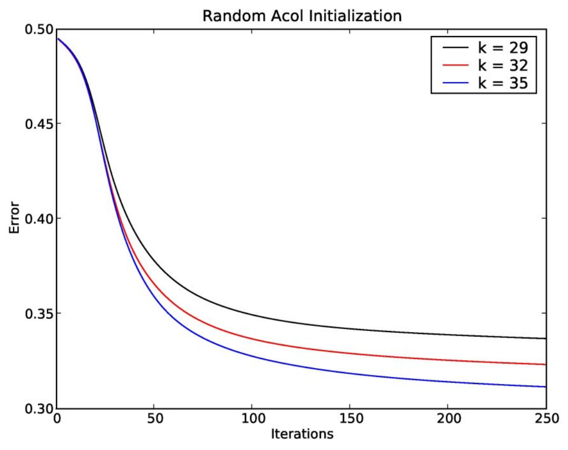

Figure 6: Percent error vs. k for a 70×200 data matrix

4.4 Software Implementation

All simulations were run using SAGE [6] which is an open source alternative to programs such as *Matlab* and *Mathematica*; it is based on the Python programming language. We include in this section the source code for the implementation of the simulations in SAGE. As the author is not a seasoned programmer, these implementations may not be the most efficient methods of accomplishing the required tasks and do not necessarily adhere to any programming guidelines.

Procedure Definitions

```
eps=0.000000001

def normal(MatToNorm, cols):
    Mat=zeros((len(MatToNorm),len(MatToNorm.T)))
    for t in range(cols):
        Mat[:,t]=MatToNorm[:,t]/sum(MatToNorm.T[t])
    return (Mat)

def nmf (y, w, h, iter, ncols):
    norms=zeros((1,iter+1))
    norms[0,0]=fnorm(y-dot(w,h))
    for x in range(iter):
        h=h*(dot(w.T,y)/(dot(dot(w.T,w),h)+eps))
        w=w*(dot(y,h.T)/(dot(dot(w,h),h.T)+eps))
        w=normal(w, ncols)
        norms[0,x+1]=fnorm(y-dot(w,h))
    return(w, h, norms)

def fnorm(a):
    return ((sum(a*a))**.5)

def AvgRandomCols(wcols, NumToAvg, Y):
    winit=zeros((len(Y),wcols))
    for s in range(wcols):
        index=zeros((1,NumToAvg))-1
        for p in range(NumToAvg):
            q=int((random.rand()*len(Y.T)))
            while q in index:
                q=int((random.rand()*len(Y.T)))
            index[0,p]=q
        for r in range(NumToAvg):
            winit[:,s]=winit[:,s]+Y[:,index[0,r]]
        winit[:,s]=winit[:,s]/NumToAvg
    return (winit)
```

```
def AvgLongCols(NumLong, NumColsToAvg, Y, Compress):
    ColNorms=zeros((1,len(Y.T)))
    for j in range(len(Y.T)):
        ColNorms[0,j]=fnorm(Y[:,j])
    Sorted=argsort(ColNorms)
    LongCols=zeros((len(Y),NumLong))
    for j in range(NumLong):
        LongCols[:,j]=Y[:,Sorted[0,len(Y.T)-j-1]]
    W=AvgRandomCols(Compress, NumColsToAvg, LongCols)
    return(W)

def EucNorm(B):
    cols=len(B.T)
    for x in range(cols):
        B[:,x]=B[:,x]/fnorm(B[:,x])
    return (B)

def initKmeans(wrows, k, Y):
    CM=zeros((wrows,k))
    temp=zeros((1,k))-1
    for i in range(k):
        q=int((random.rand()*len(Y.T)))
        while q in temp:
            q=int((random.rand()*len(Y.T)))
        CM[:,i]=Y[:,q]
        temp[0,i]=q
    return(CM)

def NewClusters(D,tempI):
    tempI[tempI¿0]=0
    for i in range(len(tempI)):
        q=int(where(D[i,:]==max(D[i,:]))[0])
        tempI[i,q]=1
    return(tempI)
```

```
def NewCent(tempCent, Indicator, Y):
    tempCent[tempCent¿0]=0
    rows=len(Indicator)
    cols=len(tempCent.T)
    for j in range(cols):
        for i in range(rows):
            if Indicator[i,j]==1:
                tempCent[:,j]=tempCent[:,j]+Y[:,i]
    for k in range(cols):
        tempCent[:,k]=tempCent[:,k]/fnorm(tempCent[:,k])
    return (tempCent)

def Converge(A,B):
    return (all(A==B))

def Kmeans(wrows, Y, NumClusters):
    Y0=copy(Y)
    Y0=EucNorm(Y0)
    CentMat=initKmeans(wrows,NumClusters,Y0)
    PrevIndicator=zeros((len(Y0.T),NumClusters))
    CurrentIndicator=zeros((len(Y0.T),NumClusters))
    D=dot(Y0.T,CentMat)
    CurrentIndicator=NewClusters(D,CurrentIndicator)
    while not Converge(CurrentIndicator, PrevIndicator):
        CentMat=NewCent(CentMat, CurrentIndicator, Y0)
        PrevIndicator=copy(CurrentIndicator)
        D=dot(Y0.T,CentMat)
        CurrentIndicator=NewClusters(D,CurrentIndicator)
    return (CentMat)
```

Example Simulation Executions

Data matrix y, k-means initialization, 250 iterations, $k = 25$

```
w=Kmeans(70,y,25)
w=normal(w,25)
h=random.rand(25,100)
ww,hh,norm=nmf(y,w,h,250,25)
```

Data matrix y, Random C initialization, 250 iterations, $k = 25, p = 10, q = 15$

```
w=AvgLongCols(15,10,y,25)
w=normal(w,len(w.T))
h=random.rand(25,100)
ww,hh,norms=nmf(y,w,h,250,25)
```

5. Conclusion

We have examined four initialization techniques for the NMF algorithm: random, random Acol, random C, and spherical k-means. For our given data set consisting of densely populated random matrices, all four initialization techniques produced similar results in the performance. They also all demonstrated that the more we try to reduce the rank of the factor matrices, the greater the error we create. In random Acol, the choice of how many columns to average had only a minor impact on the performance. Likewise, changing the number of the longest columns, from which we select the columns that are then averaged, had a small impact on the performance of random C. As the size of the data matrix increased and k remained constant, the magnitude of the degradation in the performance decreased. We also introduced the compression ratio which allows the use of smaller data sets to determine acceptable choices of k for large data sets.

REFERENCES

[1] Michael W. Berry, Murray Browne, Amy N. Langville, V. Paul Pauca, Robert J. Plemmons, Algorithms and Applications for Approximate Nonnegative Matrix Factorization. Preprint. 2006

[2] Amy N. Langville, Carl D. Meyer, Russell Albright, James Cox, and David Duling. Algorithms, Initializations, and Convergence for the Nonnegative Matrix Factorization. Preprint.

[3] D. Lee, H. Seung, Algorithms for Nonnegative Matrix Factorization, *Advances in Neural Information Processing Systems*, pp. 556-562, 2000.

[4] Steven J. Leon, *Linear Algebra with Applications*, Pearson Education Inc., 2006.

[5] V. Paul Pauca, J. Piper, Robert J. Plemmons, Nonnegative matrix factorization for spectral data analysis, *Linear Algebra and its Applications*, Vol. 416, pp. 29-47, 2006.

[6] [SAGE], SAGE Mathematical Software, Version 3.0.1, http://www.sagemath.org

[7] S. Wild, Seeding Nonnegative Matrix Factorization with the Spherical K-Means Clustering, M.S. Thesis, University of Colorado, 2002.