CS896 Introduction to Web Science
Fall 2013
Report for Assignment 9

Corren G. McCoy

December 5, 2013

Contents

List of Figures

List of Tables

1 Question 1

1.1 Problem

Create a blog-term matrix. Use the blog title as the identifier for each blog (and row of the matrix). Use the terms from every item/title (RSS) or entry/title (Atom) for the columns of the matrix. The values are the frequency of occurrence. Essentially you are replicating the format of the "blogdata.txt" file included with the PCI book code. Limit the number of terms to the most "popular" (i.e., frequent) 500 terms, this is *after* the criteria on p. 32 (slide 7) has been satisfied.

- `http://f-measure.blogspot.com/`
- `http://ws-dl.blogspot.com/`

1.2 Response

We will use the techniques described in Segaran, 2007 [1] to cluster a set of 100 blogs based on the number of times a particular word appears in the title of each blog. To generate the complete dataset needed for this problem, we used the two recommended blogs along with a prebuilt, supplementary dataset provided with the textbook material. The supplementary dataset contains 100 RSS URLs which represent the "feeds for all of the most highly referenced blogs" according to Segaran. The modified Python function *generatefeedvector.py*, shown in Appendix A, performs the following tasks.

- Parse the XML for the RSS or ATOM feed using the Universal Feed Parser (`http://code.google.com/p/feedparser/`).
- Identify the blog entries using either the *summary* or *description* tag.
- Strip the HTML and returns a list of words. Sort the list in descending order based on the frequency.
- Eliminates common words based on a minimum and maximum frequency percentage (10 to 50%) based on the accumulated word count in the feed list. We also applied a filter to ensure that only significant words, with a length of at least three characters, remain in the list.
- Extract the 500 most popular terms from the word list.
- Use the resulting list of words and the blog names to create the blog-term matrix (i.e.,blogdata.txt). The text file is included in the github supporting files for this assignment (`https://github.com/correnm/cs595-f13/tree/master/SupportingFiles`.

2 Question 2

2.1 Problem

Create an ASCII and JPEG dendrogram that clusters (i.e., HAC) the most similar blogs (see slides 12 & 13). Include the JPEG in your report and upload the ascii file to github (it will be too unwieldy for inclusion in the report).

2.2 Response

We slightly modified the *clusters.py* found in Segaran [1] to adjust the dimensions and redirect the output when producing the dendrogram. The updated source code is shown in Appendix A. The complete ASCII version, *ascii-dendrogram.txt*, is included in the github supporting files for this assignment (`https://github.com/correnm/cs595-f13/tree/master/SupportingFiles`. An example of the clusters found in our set of blogs is shown below:

```
-
  -
    FOXNews.com
    -
      -
        Online Marketing Report
        -
          Bloggers Blog
          -
            ShoeMoney
            NewsBusters - Exposing Liberal Media Bias
      -
        flagrantdisregard
        SpikedHumor - Today's Videos and Pictures
  -
    Michelle Malkin
```

The graphical version of the dendrogram is shown in Figure 1. ****** Note from PCI: The dashes represent a cluster of two or more merged items. Here you see a great example of finding a group; its also interesting to see that there is such a large chunk of search-related blogs in the most popular feeds. Looking through, you also should be able to spot clusters of political blogs, technology blogs, and blogs about blogging. Youll also probably notice some anomalies. These writers may not have written on the same themes, but the clustering algorithm says that their word frequencies are correlated. This might be a reflection of their writing style or could simply be a coincidence based on the day that the data was downloaded.*****

3 Question 3

3.1 Problem

Cluster the blogs using K-Means, using k=5, 10, 20. (see slide 18). How many interations were required for each value of k?

3.2 Response

We used functions in *clusters.py* to apply k-means to hierarchical cluster for our set of blogs. We obtained the following results using various values for the number of clusters.

- When k=5, iterations=5
- When k=10, iterations=5
- When k=20, iterations=4

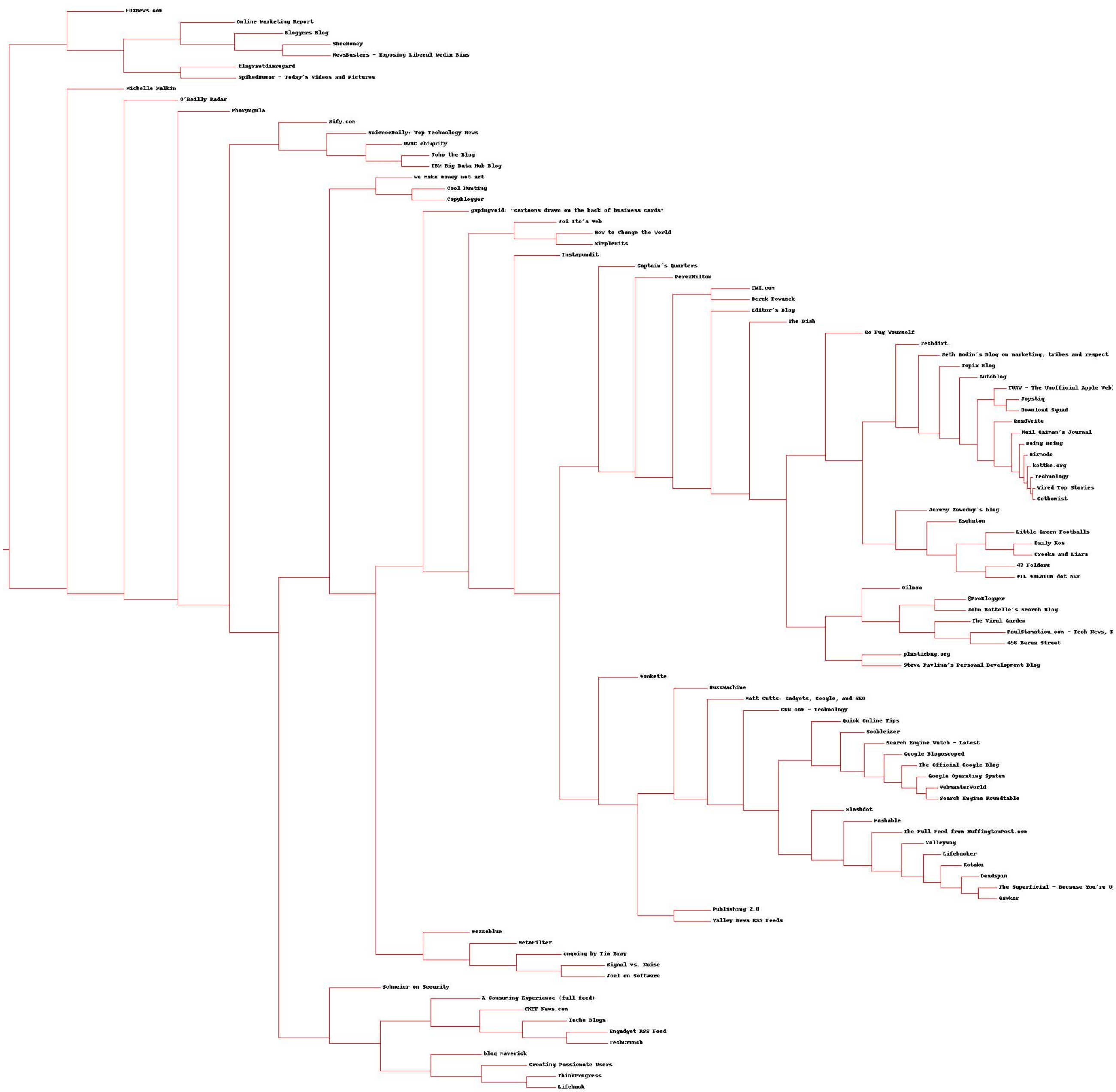

Figure 1: JPEG Dendrogram

****PCI Note: K-means clustering begins with k randomly placed centroids (points in space that represent the center of the cluster), and assigns every item to the nearest one. After the assignment, the centroids are moved to the average location of all the nodes assigned to them, and the assignments are redone. This process repeats until the assignments stop****

4 Question 4

4.1 Problem

Use MDS to create a JPEG of the blogs similar to slide 29. How many iterations were required?

4.2 Response

We used functions in *clusters.py* to perform MDS. 156 iterations were required to produce the JPEG shown in Figure 2. ****PCI note: Figure 3-10 shows the outcome of the multidimensional scaling algorithm. The clusters dont break out quite as well as they do on the dendrogram, but theres still clearly some topical grouping, such as the search-engine-related set near the top. These ended up very far away from the political and celebrity blogs. ***

5 Question 5 Extra Credit

5.1 Problem

Re-run question 2, but this time with proper TFIDF calculations instead of the hack discussed on slide 7 (p. 32). Use the same 500 words, but this time replace their frequency count with TFIDF scores as computed in assignment #3. Document the code, techniques, methods, etc. used to generate these TFIDF values. Upload the new data file to github. Compare and contrast the resulting dendrogram with the dendrogram from question #2.

5.2 Response

Not attempted.

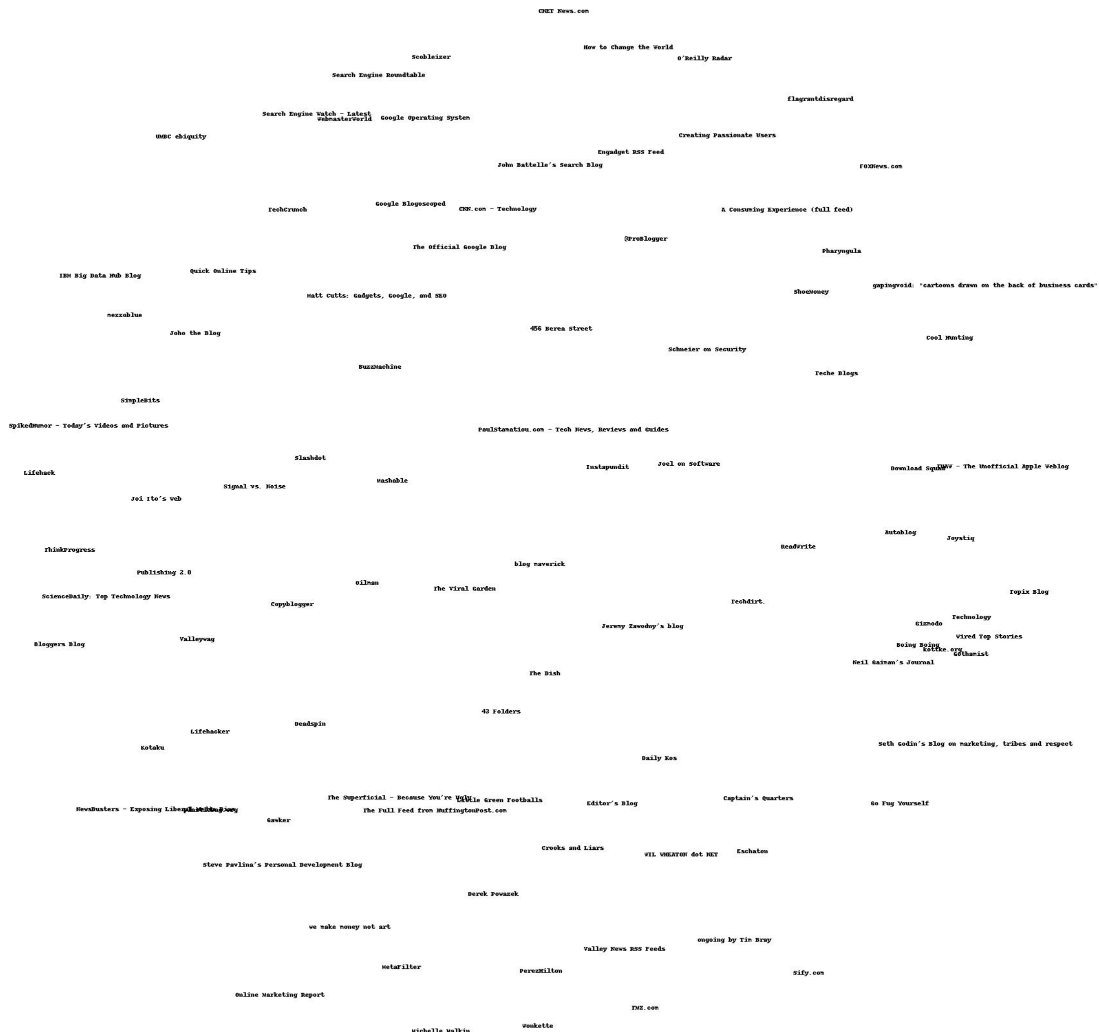

Figure 2: MDS for Blogs

Bibliography

[1] T. Segaran. *Programming collective intelligence: building smart web 2.0 applications*. O'Reilly Media, 2007.

Appendix A

Python Source Code

```
import feedparser
import re
import sys
from operator import itemgetter

# Returns title and dictionary of word counts for an RSS feed
def getwordcounts(url):
  # Parse the feed
  d=feedparser.parse(url)
  wc={}

  # Loop over all the entries
  for e in d.entries:
    if 'summary' in e: summary=e.summary
    else: summary=e.description

    # Extract a list of words
    words=getwords(e.title+' '+summary)
    for word in words:
      wc.setdefault(word,0)
      wc[word]+=1
  return d.feed.title,wc

def getwords(html):
  # Remove all the HTML tags
  txt=re.compile(r'<[^>]+>').sub('',html)

  # Split words by all non-alpha characters
  words=re.compile(r'[^A-Z^a-z]+').split(txt)

  # Convert to lowercase
  return [word.lower() for word in words if word!='']

## The first part of the code loops
## over every line in feedlist.txt and generates the word
## counts for each blog, as well as the number of blogs each
```

```
## word appeared in (apcount).
apcount={}
wordcounts={}
feedlist=[line for line in file('C:/Python27/myFiles/Assignment 9/feedlist.txt')]
for feedurl in feedlist:
  try:
    title,wc=getwordcounts(feedurl)
    wordcounts[title]=wc
    for word,count in wc.items():
      apcount.setdefault(word,0)
      if count>1:
        apcount[word]+=1
  except:
    print 'Failed to parse feed %s' % feedurl
    e = sys.exc_info()
    print "Error:" , e


wordlist=[]
# Sort the word, blogcount in descending order.
# When we apply the filtering criteria, the words
# will already be in order by frequency
for w,bc in sorted(apcount.items(), key=itemgetter(1), reverse=True): #apcount.items():
  frac=float(bc)/len(feedlist)
  ## you can reduce the total number of words
  ## included  by  selecting  only  those  words
  ## that  are  within  maximum  and  minimum
  ## percentages. In this case, you can start with 10 percent
  ## as the lower bound and 50 percent as the upper bound.
  ## Also, filter out single letter words.
  if frac>0.1 and frac<0.5 and len(w) >=3:
    wordlist.append(w)

## The final step is to use the list of words and the list of blogs
## to create a text file containing a big matrix of all the word
## counts for each of the blogs.
out=file('C:/Python27/myFiles/Assignment 9/blogdata.txt','w')
out.write('Blog')
# Top 500 words only
wordlist500=wordlist[0:500]
for word in wordlist500: out.write('\t%s' % word)
out.write('\n')
for blog,wc in wordcounts.items():
  print blog
  out.write(blog)
  for word in wordlist500:
    if word in wc: out.write('\t%d' % wc[word])
    else: out.write('\t0')
  out.write('\n')
```

```
import sys
from PIL import Image,ImageDraw

def readfile(filename):
  lines=[line for line in file(filename)]

  # First line is the column titles
  colnames=lines[0].strip().split('\t')[1:]
  rownames=[]
  data=[]
  for line in lines[1:]:
    p=line.strip().split('\t')
    # First column in each row is the rowname
    rownames.append(p[0])
    # The data for this row is the remainder of the row
    data.append([float(x) for x in p[1:]])
  return rownames,colnames,data


from math import sqrt

def pearson(v1,v2):
  # Simple sums
  sum1=sum(v1)
  sum2=sum(v2)

  # Sums of the squares
  sum1Sq=sum([pow(v,2) for v in v1])
  sum2Sq=sum([pow(v,2) for v in v2])

  # Sum of the products
  pSum=sum([v1[i]*v2[i] for i in range(len(v1))])

  # Calculate r (Pearson score)
  num=pSum-(sum1*sum2/len(v1))
  den=sqrt((sum1Sq-pow(sum1,2)/len(v1))*(sum2Sq-pow(sum2,2)/len(v1)))
  if den==0: return 0

  return 1.0-num/den

class bicluster:
  def __init__(self,vec,left=None,right=None,distance=0.0,id=None):
    self.left=left
    self.right=right
    self.vec=vec
    self.id=id
    self.distance=distance

def hcluster(rows,distance=pearson):
```

```
  distances={}
  currentclustid=-1

  # Clusters are initially just the rows
  clust=[bicluster(rows[i],id=i) for i in range(len(rows))]

  while len(clust)>1:
    lowestpair=(0,1)
    closest=distance(clust[0].vec,clust[1].vec)

    # loop through every pair looking for the smallest distance
    for i in range(len(clust)):
      for j in range(i+1,len(clust)):
        # distances is the cache of distance calculations
        if (clust[i].id,clust[j].id) not in distances:
          distances[(clust[i].id,clust[j].id)]=distance(clust[i].vec,clust[j].vec)

        d=distances[(clust[i].id,clust[j].id)]

        if d<closest:
          closest=d
          lowestpair=(i,j)

    # calculate the average of the two clusters
    mergevec=[
    (clust[lowestpair[0]].vec[i]+clust[lowestpair[1]].vec[i])/2.0
    for i in range(len(clust[0].vec))]

    # create the new cluster
    newcluster=bicluster(mergevec,left=clust[lowestpair[0]],
                         right=clust[lowestpair[1]],
                         distance=closest,id=currentclustid)

    # cluster ids that weren't in the original set are negative
    currentclustid-=1
    del clust[lowestpair[1]]
    del clust[lowestpair[0]]
    clust.append(newcluster)

  return clust[0]

def printclust(clust,labels=None,n=0):
  # indent to make a hierarchy layout
  for i in range(n): print ' ',
  if clust.id<0:
    # negative id means that this is branch
    print '-'
  else:
    # positive id means that this is an endpoint
```

```
    if labels==None: print clust.id
    else: print labels[clust.id]

  # now print the right and left branches
  if clust.left!=None: printclust(clust.left,labels=labels,n=n+1)
  if clust.right!=None: printclust(clust.right,labels=labels,n=n+1)

def getheight(clust):
  # Is this an endpoint? Then the height is just 1
  if clust.left==None and clust.right==None: return 1

  # Otherwise the height is the same of the heights of
  # each branch
  return getheight(clust.left)+getheight(clust.right)

def getdepth(clust):
  # The distance of an endpoint is 0.0
  if clust.left==None and clust.right==None: return 0

  # The distance of a branch is the greater of its two sides
  # plus its own distance
  return max(getdepth(clust.left),getdepth(clust.right))+clust.distance


def drawdendrogram(clust,labels,jpeg='clusters.jpg'):
  # height and width
  h=getheight(clust)*20
  w=2000
  depth=getdepth(clust)

  # width is fixed, so scale distances accordingly
  scaling=float(w-150)/depth

  # Create a new image with a white background
  img=Image.new('RGB',(w,h),(255,255,255))
  draw=ImageDraw.Draw(img)

  draw.line((0,h/2,10,h/2),fill=(255,0,0))

  # Draw the first node
  drawnode(draw,clust,10,(h/2),scaling,labels)
  img.save(jpeg,'JPEG')

def drawnode(draw,clust,x,y,scaling,labels):
  if clust.id<0:
    h1=getheight(clust.left)*20
    h2=getheight(clust.right)*20
    top=y-(h1+h2)/2
    bottom=y+(h1+h2)/2
```

```
    # Line length
    ll=clust.distance*scaling
    # Vertical line from this cluster to children
    draw.line((x,top+h1/2,x,bottom-h2/2),fill=(255,0,0))

    # Horizontal line to left item
    draw.line((x,top+h1/2,x+ll,top+h1/2),fill=(255,0,0))

    # Horizontal line to right item
    draw.line((x,bottom-h2/2,x+ll,bottom-h2/2),fill=(255,0,0))

    # Call the function to draw the left and right nodes
    drawnode(draw,clust.left,x+ll,top+h1/2,scaling,labels)
    drawnode(draw,clust.right,x+ll,bottom-h2/2,scaling,labels)
  else:
    # If this is an endpoint, draw the item label
    draw.text((x+5,y-7),labels[clust.id],(0,0,0))

def rotatematrix(data):
  newdata=[]
  for i in range(len(data[0])):
    newrow=[data[j][i] for j in range(len(data))]
    newdata.append(newrow)
  return newdata

import random

def kcluster(rows,distance=pearson,k=4):
  # Determine the minimum and maximum values for each point
  ranges=[(min([row[i] for row in rows]),max([row[i] for row in rows]))
  for i in range(len(rows[0]))]

  # Create k randomly placed centroids
  clusters=[[random.random()*(ranges[i][1]-ranges[i][0])+ranges[i][0]
  for i in range(len(rows[0]))] for j in range(k)]

  lastmatches=None
  for t in range(100):
    print 'Iteration %d' % t
    bestmatches=[[] for i in range(k)]

    # Find which centroid is the closest for each row
    for j in range(len(rows)):
      row=rows[j]
      bestmatch=0
      for i in range(k):
        d=distance(clusters[i],row)
        if d<distance(clusters[bestmatch],row): bestmatch=i
      bestmatches[bestmatch].append(j)
```

```
    # If the results are the same as last time, this is complete
    if bestmatches==lastmatches: break
    lastmatches=bestmatches

    # Move the centroids to the average of their members
    for i in range(k):
      avgs=[0.0]*len(rows[0])
      if len(bestmatches[i])>0:
        for rowid in bestmatches[i]:
          for m in range(len(rows[rowid])):
            avgs[m]+=rows[rowid][m]
        for j in range(len(avgs)):
          avgs[j]/=len(bestmatches[i])
        clusters[i]=avgs

  return bestmatches

def tanamoto(v1,v2):
  c1,c2,shr=0,0,0

  for i in range(len(v1)):
    if v1[i]!=0: c1+=1 # in v1
    if v2[i]!=0: c2+=1 # in v2
    if v1[i]!=0 and v2[i]!=0: shr+=1 # in both

  return 1.0-(float(shr)/(c1+c2-shr))

def scaledown(data,distance=pearson,rate=0.01):
  n=len(data)

  # The real distances between every pair of items
  realdist=[[distance(data[i],data[j]) for j in range(n)]
             for i in range(0,n)]

  # Randomly initialize the starting points of the locations in 2D
  loc=[[random.random(),random.random()] for i in range(n)]
  fakedist=[[0.0 for j in range(n)] for i in range(n)]

  lasterror=None
  for m in range(0,1000):
    # Find projected distances
    for i in range(n):
      for j in range(n):
        fakedist[i][j]=sqrt(sum([pow(loc[i][x]-loc[j][x],2)
                                 for x in range(len(loc[i]))]))

    # Move points
    grad=[[0.0,0.0] for i in range(n)]
```

```
    totalerror=0
    for k in range(n):
      for j in range(n):
        if j==k: continue
        # The error is percent difference between the distances
        errorterm=(fakedist[j][k]-realdist[j][k])/realdist[j][k]

        # Each point needs to be moved away from or towards the other
        # point in proportion to how much error it has
        grad[k][0]+=((loc[k][0]-loc[j][0])/fakedist[j][k])*errorterm
        grad[k][1]+=((loc[k][1]-loc[j][1])/fakedist[j][k])*errorterm

        # Keep track of the total error
        totalerror+=abs(errorterm)
    print totalerror

    # If the answer got worse by moving the points, we are done
    if lasterror and lasterror<totalerror: break
    lasterror=totalerror

    # Move each of the points by the learning rate times the gradient
    for k in range(n):
      loc[k][0]-=rate*grad[k][0]
      loc[k][1]-=rate*grad[k][1]

  return loc

def draw2d(data,labels,jpeg='mds2d.jpg'):
  img=Image.new('RGB',(2000,2000),(255,255,255))
  draw=ImageDraw.Draw(img)
  for i in range(len(data)):
    x=(data[i][0]+0.5)*1000
    y=(data[i][1]+0.5)*1000
    draw.text((x,y),labels[i],(0,0,0))
  img.save(jpeg,'JPEG')

## Main driver added
if __name__ == "__main__":
    import clusters
    # hierarchical clustering
    blognames,words,data=clusters.readfile('blogdata.txt')
    clust=clusters.hcluster(data)
    # ASCII dendrodram
    out=file('C:/Python27/myFiles/Assignment 9/ASCII-Dendrogram.txt','w')
    # redirect standard output to our file
    orig_stdout = sys.stdout
    sys.stdout = out
    clusters.printclust(clust,labels=blognames)
```

```
out.close()
sys.stdout = orig_stdout
# JPEG dendrogram
clusters.drawdendrogram(clust,blognames,jpeg='blogclust.jpg')
print "Dendrodrams complete."
# K-Means Clustering
print "K=5"
kclust=clusters.kcluster(data,k=5)
print "\n"
print "K=10"
kclust=clusters.kcluster(data,k=10)
print "\n"
print "K=20"
kclust=clusters.kcluster(data,k=20)
# Multidimensional scaling
coords=clusters.scaledown(data)
clusters.draw2d(coords,blognames,jpeg='blogs2d.jpg')
```