Bo Zhang
01063214

1 Create a blog-term matrix

Create a blog-term matrix. Start by grabbing 100 blogs; include:
http://f-measure.blogspot.com/
http://ws-dl.blogspot.com/
and grab 98 more as per the method shown in class.

Use the blog title as the identifier for each blog. 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" 1000 terms, this is *after* the criteria on p. 32 (slide 7) has been satisfied. Remember that blogs are paginated.

Algorithm:

1. Open the URL of "Next Blog" 500 times to get 500 Blog URLs.
2. For each time opening the URL, get a BeautifulSoup object, extract the Atom link, save it to "feedlist.txt" and open it.
3. Get a BeautifulSoup object with the new page, extract the link of "Next Page"(if any), save it to "feedlist.txt" and open it.
4. Repeat Step 3 again and again until there is no "Next Page".
5. Open the URLs of Blog "f-measure" and "ws-dl", finish the Steps 2-4 of each Blog but use "feedlistReady.txt" as the output file.
6. Open "feedlist.txt" to read all the URLs to a list.
7. Use "set" method to remove duplicated URLs.
8. Open the remaining URLs 1 by 1, use the Blog title and the package "langid" to remove non-English Blogs.
9. Save the first 98 remaining URLs to "feedlistReady.txt" and the others to "feedlistMore.txt".
10. Use the script from "Programming Collective Intelligence" to open "feedlistReady.txt" and count all the words from the 100 Blogs.
11. Edit the script to add the frequency to the word list and sort the word list by frequency.
12. Generate the blogdata with top 1000 words in the word list and save it as "blogdata(allFilter).txt".

Source code:

Listing 1: The content of BlogURLdownload.py

```
import urllib.request
from bs4 import BeautifulSoup

def getNextPage(url, file):
    with urllib.request.urlopen(url) as res:
        html = res.read()
    soup = BeautifulSoup(html, "html.parser")
    for link in soup.find_all('link'):
        if link['rel']==['next']:
            f = open(file, 'a', encoding='utf-8')
            f.write(link['href']+'\n')
            f.close()
            getNextPage(link['href'],file)
            break
    return


startLink='https://www.blogger.com/next-blog?navBar=true&blogID=3471633091411211117'
for i in range(500):
    print (i)
    try:
        with urllib.request.urlopen(startLink) as res:
            html = res.read()
        soup = BeautifulSoup(html, "html.parser")
        print (soup.title)

        for link in soup.find_all('link'):
            if link['rel']==['alternate'] and link['type']=='application/atom+xml':
                f = open('feedlist.txt','a',encoding='utf-8')
                f.write(link['href']+'\n')
                f.close()
                getNextPage(link['href'],'feedlist.txt')
                break
    except:
        continue


with urllib.request.urlopen('http://f-measure.blogspot.com/') as res:
    html = res.read()
soup = BeautifulSoup(html, "html.parser")
for link in soup.find_all('link'):
    if link['rel']==['alternate'] and link['type']=='application/atom+xml':
        f = open('feedlistReady.txt','a',encoding='utf-8')
        f.write(link['href']+'\n')
        f.close()
        getNextPage(link['href'],'feedlistReady.txt')
        break

with urllib.request.urlopen('http://ws-dl.blogspot.com/') as res:
    html = res.read()
soup = BeautifulSoup(html, "html.parser")
for link in soup.find_all('link'):
    if link['rel']==['alternate'] and link['type']=='application/atom+xml':
        f = open('feedlistReady.txt','a',encoding='utf-8')
        f.write(link['href']+'\n')
        f.close()
        getNextPage(link['href'],'feedlistReady.txt')
        break
```

Results:

feedlist.txt
feedlistReady.txt
feedlistMore.txt
blogdata(allFilter).txt

2 Hierarchical Clustering

Create an ASCII and JPEG dendrogram that clusters the most similar blogs. Include the JPEG in your report and upload the ascii file to github.

Algorithm:

1. Use the script from “Programming Collective Intelligence” to open “blogdata(allFilter).txt”.
2. Run the Hierarchical Clustering.
3. Save the ASCII dendrogram to “ASCII dendrogram(Q2).txt”.
4. Save the JPEG dendrogram to “blogclust(Q2).jpg”.

Source code:

Listing 2: The content of Q2.py

```
from math import sqrt
from PIL import Image,ImageDraw

def readfile(filename):
    f = open(filename, 'r')
    lines=f.readlines()
    f.close()
    #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
```

```
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=1200
    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))


blognames,words,data=readfile('blogdata(allFilter).txt')
clust=hcluster(data)
printclust(clust,labels=blognames)
drawdendrogram(clust,blognames,jpeg='blogclust(Q2).jpg')
```

Results:

ASCII dendrogram(Q2).txt

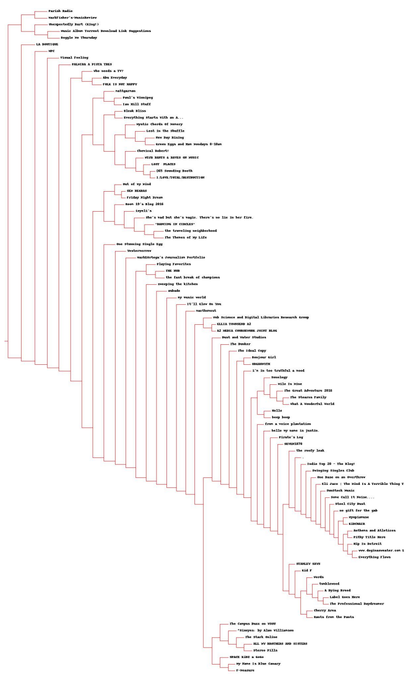

Figure 1: blogclust(Q2)

3 K-Means Clustering

Cluster the blogs using K-Means, using k=5,10,20. Print the values in each centroid, for each value of k. How many interations were required for each value of k?

Algorithm:
1. Use the script from "Programming Collective Intelligence" to open "blogdata(allFilter).txt".
2. Run the K-Means Clustering with k=5,10,20.
3. Print all the centroid.

Source code:

Listing 3: The content of Q3.py

```
from math import sqrt
import random

def readfile(filename):
    f = open(filename, 'r')
    lines=f.readlines()
    f.close()
    #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


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


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


blognames,words,data=readfile('blogdata(allFilter).txt')

print ('k=5:')
kclust=kcluster(data,k=5)
for i in range(5):
    print ([blognames[r] for r in kclust[i]])
    print ('\n')

print ('\nk=10:')
kclust=kcluster(data,k=10)
for i in range(10):
    print ([blognames[r] for r in kclust[i]])
    print ('\n')

print ('\nk=20:')
kclust=kcluster(data,k=20)
for i in range(20):
    print ([blognames[r] for r in kclust[i]])
    print ('\n')
```

Results: K-Means.txt

4 Multidimensional Scaling

Use MDS to create a JPEG of the blogs similar to slide 29 of the week 12 lecture. How many iterations were required?

Algorithm:

1. Use the script from "Programming Collective Intelligence" to open "blogdata(allFilter).txt".
2. Edit the "scaledown" function to make output both step number and error.
3. Do the Multidimensional Scaling and draw the two-dimensional chart as "blogs2d.jpg".

Source code:

Listing 4: The content of Q4.py

```
from math import sqrt
import random
from PIL import Image,ImageDraw

def readfile(filename):
    f = open(filename, 'r')
    lines=f.readlines()
    f.close()
    #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


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


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)]

    outersum=0.0
    # 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 ('{}'.format(m+1)+'\t{}'.format(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')


blognames,words,data=readfile('blogdata(allFilter).txt')
coords=scaledown(data)
draw2d(coords,blognames,jpeg='blogs2d.jpg')
```

Results: Total iterations are 255(Q4.txt)

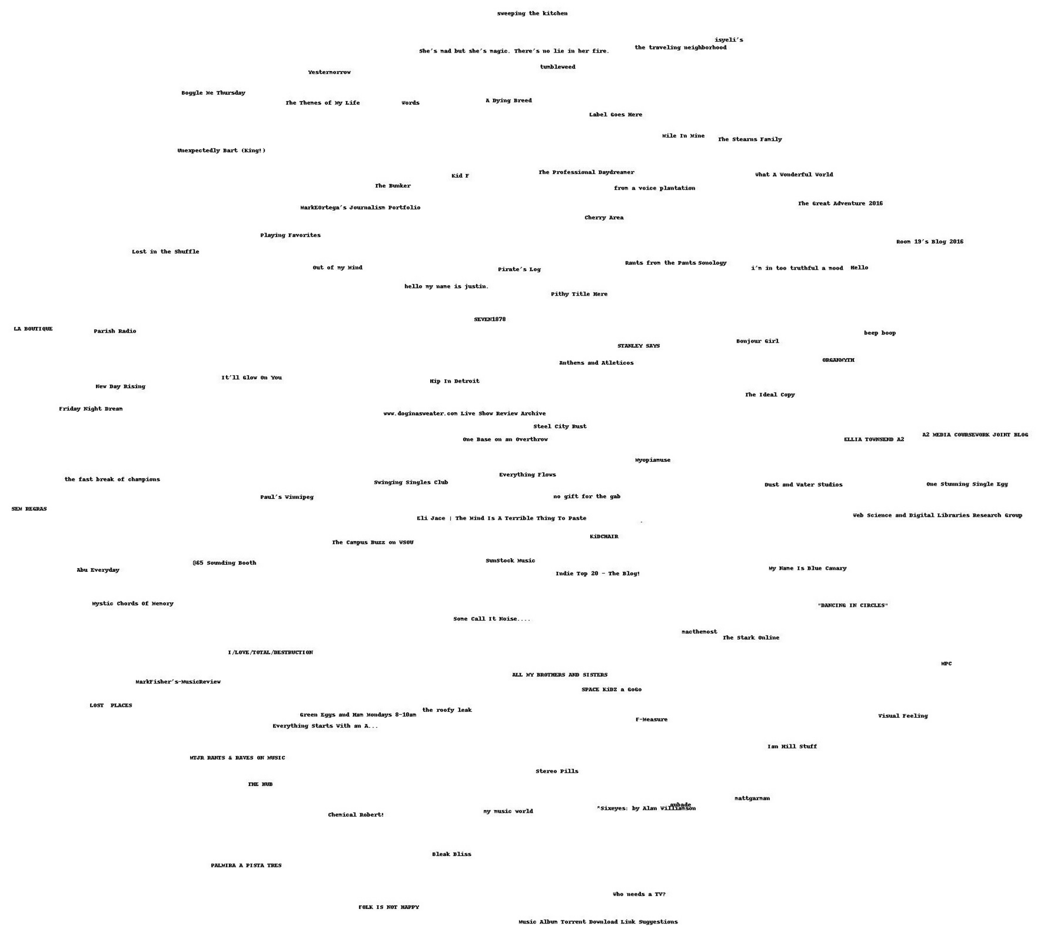

Figure 2: blogs2d

5 TFIDF

Re-run question 2, but this time with proper TFIDF calculations instead of the hack discussed on slide 7 (p. 32). Use the same 1000 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.

Algorithm:

1. Use the script from "Programming Collective Intelligence" to open "blogdata(allFilter).txt".
2. Compute the TFIDF of each term of each Blog and save them to "blogdata(Q5).txt".
3. Run the Hierarchical Clustering.
4. Save the ASCII dendrogram to "ASCII dendrogram(Q5).txt".
5. Save the JPEG dendrogram to "blogclust(Q5).jpg".

Source code:

Listing 5: The content of TFIDF.py

```
import math

def readfile(filename):
    f = open(filename, 'r', encoding='utf-8')
    lines=f.readlines()
    f.close()

    # 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


blognames,words,data=readfile('blogdata(allFilter).txt')
sumRow = [0 for i in range(len(blognames))]
countCol = [0 for i in range(len(words))]

for i in range(len(blognames)):
    for j in range(len(words)):
        sumRow[i] = sumRow[i] + data[i][j]
        if data[i][j]>0:
            countCol[j] = countCol[j] + 1

TFIDF=[]
for i in range(len(blognames)):
    line=[]
    for j in range(len(words)):
        line.append(data[i][j]/sumRow[i]*math.log(len(blognames)/countCol[j], 2))
    TFIDF.append(line)

f = open('blogdata(Q5).txt', 'w', encoding='utf-8')
f.write('Blog')
for word in words:
    f.write('\t'+word)
f.write('\n')
for i in range(len(blognames)):
    f.write(blognames[i])
    for j in range(len(words)):
        f.write('\t{}'.format(TFIDF[i][j]))
    f.write('\n')
f.close()
```

Listing 6: The content of Q5.py

```
from math import sqrt
from PIL import Image,ImageDraw

def readfile(filename):
    f = open(filename, 'r')
    lines=f.readlines()
    f.close()
    #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


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=1200
    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))


blognames,words,data=readfile('blogdata(Q5).txt')
clust=hcluster(data)
printclust(clust,labels=blognames)
drawdendrogram(clust,blognames,jpeg='blogclust(Q5).jpg')
```

Results:

ASCII dendrogram(Q5).txt

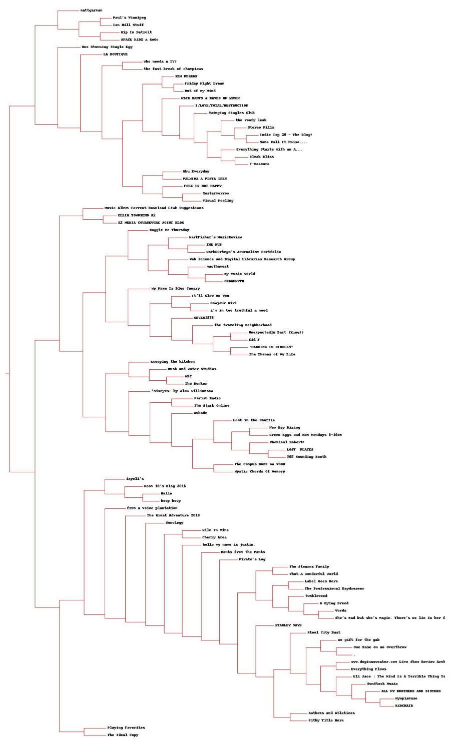

Figure 3: blogclust(Q5)

According to the result from Q2, if we split all Blogs into 2 groups, there are only 5 Blogs in the top group. According to the result from Q5, if we do the same split, there are 27 groups in the top group. The split seems more balanced with this method.