Assignment 8

Fall 2014
CS595 Web Science
Dr. Michael Nelson

Mathew Chaney

December 1, 2014

Contents

Listings

1 Question 1

1.1 Question

```
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 (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.

Create a histogram of how many pages each blog has (e.g., 30
blogs with just one page, 27 with two pages, 29 with 3 pages and
so on).
```

1.2 Answer

To complete this assignment, a blog word count dataset was required. To start off, a list of blog URIs was obtained using the method described in class, implemented as the `get_uris.py` script. Two default blogs, F-Measure and the Old Dominion Web Science and Digital Libraries blogs, were added to the list and then, using the seed URI provided (Listing 2), the remaining 98 URIs from random blogs within the blogger.com family were added. Then, using the `matrix.py` script, the page counts for each blog were extracted and saved to a file called `pagecounts`. The `matrix.py` script is a modified version of `generatefeedvectors.py` from the book *Programming Collective Intelligence* [1].

```python
if __name__ == '__main__':
    uris = set()
    with open('blog_uris', 'a') as outfile:
        if len(sys.argv) > 1 and sys.argv[1] == 'new':
            for must_have in must_haves:
                uri = get_atom(must_have)
                add_uri(uri, uris, outfile)
        else:
            with open('blog_uris') as infile:
                [uris.add(line.strip()) for line in infile]
        while len(uris) < 100:
            uri = get_atom(default)
            add_uri(uri, uris, outfile)
```

Listing 1: main for get_uris.py

```python
default = 'http://www.blogger.com/next-blog?navBar=true&blogID=3471633091411211117'
must_haves = ['http://f-measure.blogspot.com/', 'http://ws-dl.blogspot.com/']
```

Listing 2: referenced variables in get_uris.py

The `get_uris main` function in Listing 1 was the driver that called the `get_atom` function (shown in Listing 3) to extract the atom [2] URIs from each blog and add them to the set of URIs with the `add_uri` function, shown in Listing 4.

```python
def get_atom(uri):
    try:
        r = requests.get(uri)
    except Exception, e:
        return None
    soup = BeautifulSoup(r.text)
    links = soup.find_all('link', {'type':'application/atom+xml'})
    if links:
        return str(links[0]['href'])
    return None
```

Listing 3: get_atom function

```python
def add_uri(uri, uris, outfile):
    if uri and uri not in uris:
        uris.add(uri)
        outfile.write(uri + '\n')
        print len(uris), uri
```

Listing 4: add_uri function

The contents of each blog were downloaded and processed by the code shown in Listing 5 and the `get_titles`, `get_words` and `get_next` functions found in Listing 6. This code loops over the URIs that were downloaded with the `get_uris.py` script, parses each entry and extracts all the words in each entry's title. These words were then compiled into a master list for all 100 blogs, with the top 500 words that fit into the range bounded by the code in Listing 7 being used for the final word count.

```python
    with open('blog_uris') as infile:
        uris = [line.strip() for line in infile]
    if len(sys.argv) == 2 and sys.argv[1] == 'get':
        with futures.ThreadPoolExecutor(max_workers=8) as executor:
            uri_futures = [executor.submit(get_titles, uri) for uri in uris]
            for future in futures.as_completed(uri_futures):
                uri, title, subtitle, pages, wc = future.result()
                with open('wcs/' + md5.new(uri).hexdigest(), 'w') as out:
                    out.write(title + ': ' + subtitle + '\t' + str(pages) + '\t')
                    json.dump(wc, out)
```

Listing 5: looping over the URIs

```python
def get_next(d):
    for item in d.feed.links:
        if item['rel'] == u'next':
            return item['href']
    return None

def getwords(text):
    txt = re.compile(r'<[^>]+>').sub('', text)
    words = re.compile(r'[^A-Z^a-z]+').split(txt)
    return [word.lower() for word in words if word != '']

def get_titles(uri):
    print('processing {}'.format(uri))
    next = uri
    wc = {}
    pages = 0
    while next is not None:
        d = feedparser.parse(next)
        for e in d.entries:
            words = getwords(e.title.encode('utf-8'))
            for word in words:
                wc.setdefault(word, 0)
                wc[word] += 1
        pages += 1
        next = get_next(d)
        print('next {}'.format(next))
    title = d.feed.title.encode('utf-8')
    subtitle = d.feed.subtitle[:50].encode('utf-8')
    print('finished: {}: {}'.format(title, subtitle))
    return uri, title, subtitle, pages, wc
```

Listing 6: processing each blog

```python
        for w, bc in sorted(apcount.items(), key=lambda x: x[1], reverse=True):
            frac = float(bc) / len(uris)
            if frac > 0.1 and frac < 0.5:
                wordlist.append(w)
```

Listing 7: bounding the terms

To build a histogram showing the blog page counts, the `pagecounts` file was parsed by the R script in Listing 8 and saved as a pdf, which is shown in Figure 1.

```r
#! /usr/bin/Rscript

data <- read.table("pagecounts", sep="\t", header=TRUE, comment.char="")
counts <- table(data$pages)
pdf("hist.pdf")
barplot(counts, ylab="Number of Blogs", xlab="Page Count", main="Page Count per Blog")
dev.off()
```

Listing 8: building the histogram

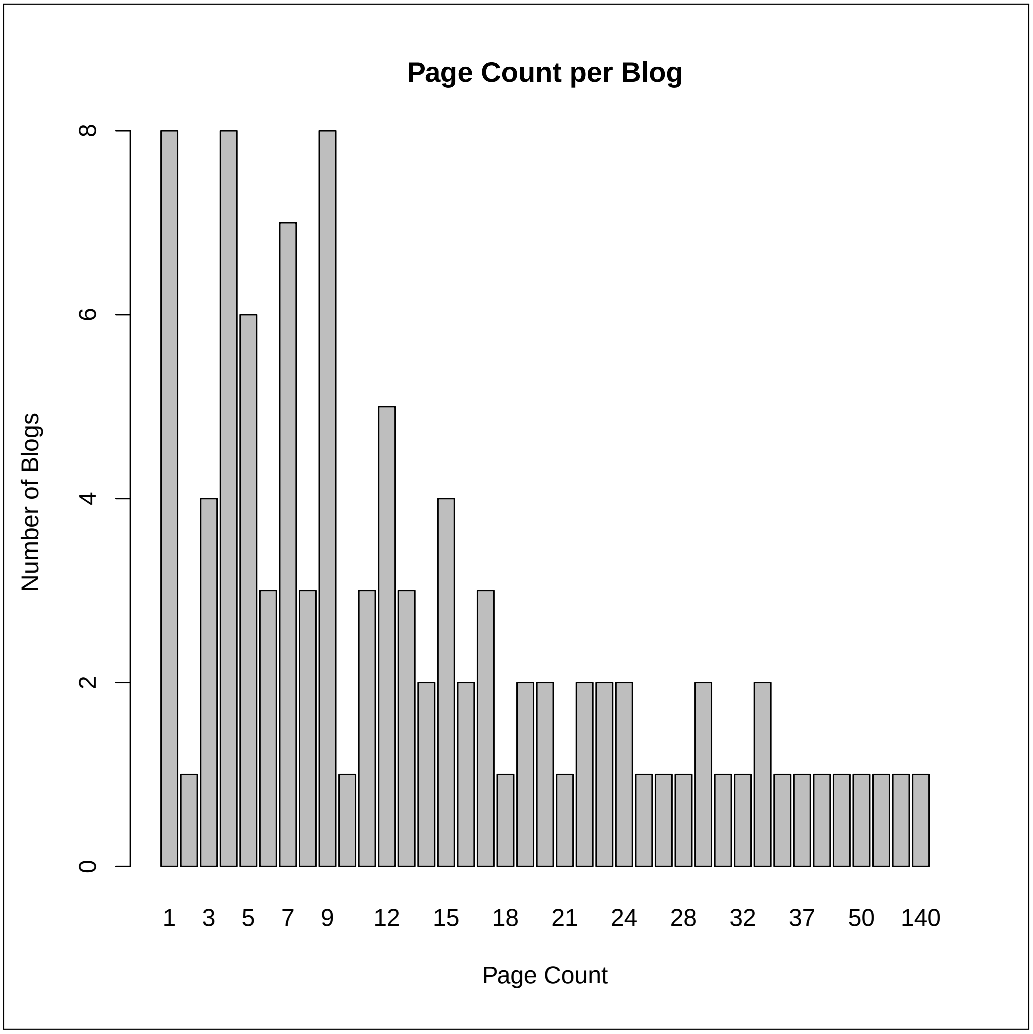

Figure 1: Page Count per Blog

2 Question 2

2.1 Question

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 Answer

The ascii and jpeg dendrograms were created using the method shown in Listing 9, which is modeled after the example from class.

```
    blognames, words, data = readfile('blogdata1.txt')
    clust = hcluster(data)
    with open('dendrogram.txt', 'w') as outfile:
        stdout = sys.stdout
        sys.stdout = outfile
        printclust(clust, labels=blognames)
        sys.stdout = stdout
    drawdendrogram(clust, blognames, jpeg='blogclust.jpg')
```

Listing 9: creating the dendrograms

This uses the `readfile` function shown in Listing 10 to read the data that was compiled from Question 1 into the script where it is then processed by the `hcluster` function found in Listing 11 to produce the clustered representation of the blogs.

```
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
```

Listing 10: creating the dendrograms

```
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]
```

Listing 11: hcluster function

The `printclust` function from Listing 12 prints the ascii dendrogram of the cluster object parameter to sys.stdout, which is redirected to write to a file with the code in Listing 9.

```
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)
```

Listing 12: printclust function

The `drawdendrogram` function from Listing 13 creates a jpeg image of the cluster, which is shown in Figure 2.

```
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')
```

Listing 13: drawdendrogram function

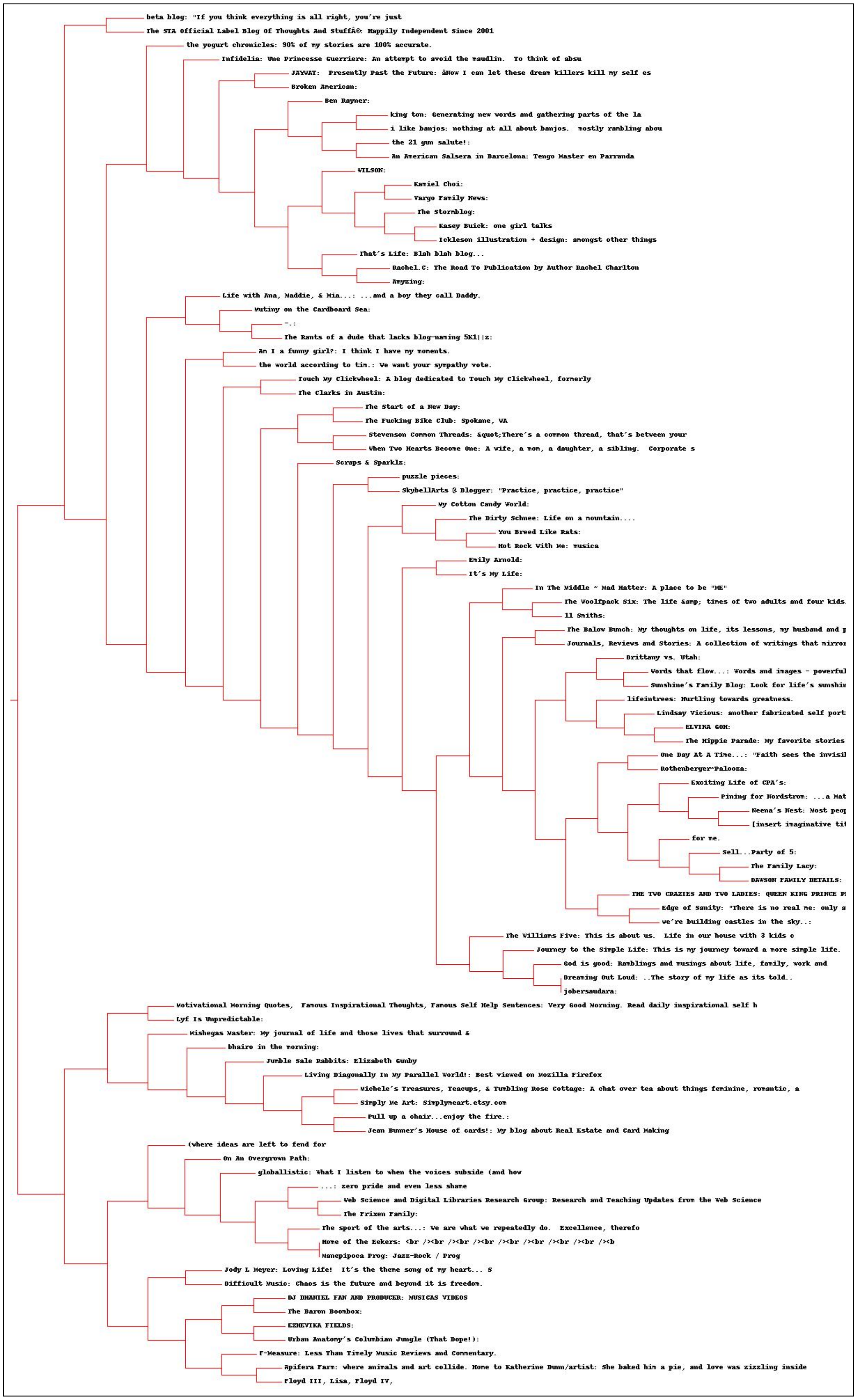

Figure 2: dendrogram

3 Question 3

3.1 Question

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 Answer

Using the code in Listing 14 kclustering was performed with values for $n = 5$, $n = 10$ and $n = 20$. The main function calls the kcluster function, which is shown in Listing 15.

```
    print "K=5"
    kclust=kcluster(data, k=5)
    print "K=10"
    kclust=kcluster(data, k=10)
    print "K=20"
    kclust=kcluster(data, k=20)
```

Listing 14: kclustering main

```
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
```

Listing 15: kcluster function

The output is shown in Listing 16. As the output reads, a kcluster with $n = 5$ required nine iterations, $n = 10$ required four iterations and $n = 20$ also required four iterations.

```
Done with dendrograms
K=5
 Iteration 0
 Iteration 1
 Iteration 2
 Iteration 3
 Iteration 4
 Iteration 5
 Iteration 6
 Iteration 7
 Iteration 8
K=10
 Iteration 0
 Iteration 1
 Iteration 2
 Iteration 3
K=20
 Iteration 0
 Iteration 1
 Iteration 2
 Iteration 3
```

Listing 16: output of kclustering algorithm

4 Question 4

4.1 Question

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

4.2 Answer

With the code in Listing 17, multidimensional scaling (MDS) was used to create a two-dimensional visualization of the blog distance graph.

```
    coords=scaledown(data)
    draw2d(coords, blognames, jpeg='blogs2d.jpg')
```

Listing 17: main for scaledown

This main code calls the scaledown function, which is shown in Listing 18. The algorithm continues until the error factor stops decreasing, as shown in the output in Appendix A, Listing 22.

```
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
        if realdist[j][k] != 0:
            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
```

Listing 18: scaledown function

The scaledown function returns the coordinates for each of the blogs in 2D space. This data was then used with the `draw2d` function in Listing 19, which produced the two-dimensional visualation created from the MDS algorithm, as shown in Figure 3.

```python
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')
```

Listing 19: draw2d function

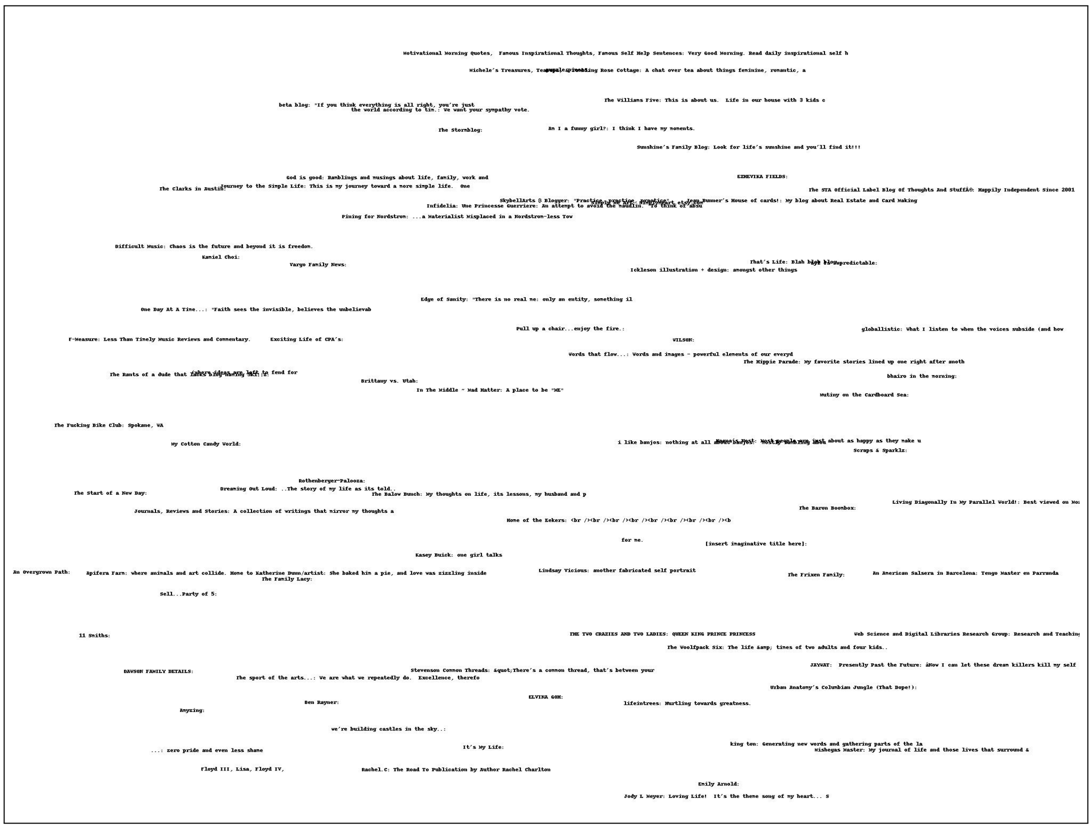

Figure 3: MDS 2d visualization

5 Question 5

5.1 Question

```
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.

Note: ideally you would not reuse the same 500 terms and instead
come up with TFIDF scores for all the terms and then choose the top
500 from that list, but I'm trying to limit the amount of work
necessary.
```

5.2 Answer

6 Appendix A

```python
#! /usr/bin/env python

import requests
import sys
from bs4 import BeautifulSoup

default = 'http://www.blogger.com/next-blog?navBar=true&blogID=3471633091411211117'
must_haves = ['http://f-measure.blogspot.com/', 'http://ws-dl.blogspot.com/']

def get_atom(uri):
    try:
        r = requests.get(uri)
    except Exception, e:
        return None
    soup = BeautifulSoup(r.text)
    links = soup.find_all('link', {'type':'application/atom+xml'})
    if links:
        return str(links[0]['href'])
    return None

def add_uri(uri, uris, outfile):
    if uri and uri not in uris:
        uris.add(uri)
        outfile.write(uri + '\n')
        print len(uris), uri

if __name__ == '__main__':
    uris = set()
    with open('blog_uris', 'a') as outfile:
        if len(sys.argv) > 1 and sys.argv[1] == 'new':
            for must_have in must_haves:
                uri = get_atom(must_have)
                add_uri(uri, uris, outfile)
        else:
            with open('blog_uris') as infile:
                [uris.add(line.strip()) for line in infile]
        while len(uris) < 100:
            uri = get_atom(default)
            add_uri(uri, uris, outfile)
```

Listing 20: get_uris.py

```python
import feedparser
import futures
import md5
import re
import sys
import json

def get_next(d):
    for item in d.feed.links:
        if item['rel'] == u'next':
            return item['href']
    return None

def getwords(text):
    txt = re.compile(r'<[^>]+>').sub('', text)
    words = re.compile(r'[^A-Z^a-z]+').split(txt)
    return [word.lower() for word in words if word != '']

def get_titles(uri):
    print('processing {}'.format(uri))
    next = uri
    wc = {}
    pages = 0
    while next is not None:
        d = feedparser.parse(next)
        for e in d.entries:
            words = getwords(e.title.encode('utf-8'))
            for word in words:
                wc.setdefault(word, 0)
                wc[word] += 1
        pages += 1
        next = get_next(d)
        print('next {}'.format(next))
    title = d.feed.title.encode('utf-8')
    subtitle = d.feed.subtitle[:50].encode('utf-8')
    print('finished: {}: {}'.format(title, subtitle))
    return uri, title, subtitle, pages, wc

if __name__ == '__main__':
    with open('blog_uris') as infile:
        uris = [line.strip() for line in infile]
    if len(sys.argv) == 2 and sys.argv[1] == 'get':
        with futures.ThreadPoolExecutor(max_workers=8) as executor:
            uri_futures = [executor.submit(get_titles, uri) for uri in uris]
            for future in futures.as_completed(uri_futures):
                uri, title, subtitle, pages, wc = future.result()
                with open('wcs/' + md5.new(uri).hexdigest(), 'w') as out:
                    out.write(title + ': ' + subtitle + '\t' + str(pages) + '\t')
                    json.dump(wc, out)
    else:
        apcount = {}
        wordcounts = {}
        pagecounts = {}
        for uri in uris:
            with open('wcs/' + md5.new(uri).hexdigest()) as infile:
                try:
                    lines = infile.read().split('\t')
                    title = lines[0]
                    pages = int(lines[1])
                    wc = json.loads(lines[2])
                except Exception, e:
                    print('*** {} generated an exception: {}'.format(uri, e))
                    continue
            wordcounts[title] = wc
            pagecounts[title] = pages
            for word, count in wc.items():
                apcount.setdefault(word, 0)
                apcount[word] += count
        wordlist = []
        for w, bc in sorted(apcount.items(), key=lambda x: x[1], reverse=True):
            frac = float(bc) / len(uris)
            if frac > 0.1 and frac < 0.5:
                wordlist.append(w)
        if len(sys.argv) == 2 and sys.argv[1] == 'pages':
            with open('pagecounts', 'w') as outfile:
                outfile.write('blog\tpages\n')
```

```python
                for blog, pagecount in pagecounts.iteritems():
                    outfile.write("\"" + blog.replace("\"", "") + "\"" + '\t' + str(
                        pagecount) + '\n')
        if len(sys.argv) == 2 and sys.argv[1] == 'wc':
            with open('blogdata1.txt', 'w') as out:
                out.write('Blog')
                for word in wordlist[:500]:
                    out.write('\t%s' % word)
                out.write('\n')
                for blog, wc in wordcounts.items():
                    print blog
                    out.write(blog)
                    for word in wordlist[:500]:
                        if word in wc:
                            out.write('\t{}'.format(wc[word]))
                        else: out.write('\t0')
                    out.write('\n')
```

Listing 21: matrix.py

```
5114.96392059
3378.46476941
3336.07313947
3321.07681636
3314.82206924
3311.9923119
3310.01886371
3308.22374881
3306.81320798
3305.60952179
3304.38029019
3303.24545568
3302.30781839
3301.243952
3299.95637566
3298.77418812
3297.63109703
3296.49161679
3295.11129947
3294.10589657
3293.18215656
3292.22035202
3291.35503982
3290.54419166
3289.68063935
3288.84471038
3287.93583439
3286.81371666
3285.75771263
3284.77259039
3283.77319553
3282.98356628
3282.34595444
3281.76444272
3281.14689344
3280.34170838
3279.58018102
3278.80573491
3277.9717399
3277.12547791
3276.22103528
3275.36339745
3274.49949723
3273.66834004
3272.88975764
3272.10968588
3271.20916637
3270.26089512
3269.2602024
3268.24022652
3267.24447156
3266.15717891
3265.11562966
3264.09645474
3263.10678415
3262.17405097
3261.24269022
3260.24197749
3259.21770939
3258.21872025
3257.21365814
3256.14783975
3255.07114015
3253.95694354
3252.82012061
3251.74546935
3250.68451973
3249.66129176
3248.70073351
3247.79195239
3246.95679073
3246.21969935
3245.51207638
3244.76825233
3244.01399188
3243.24587537
```

```
3242.48283094
3241.71482665
3240.90414524
3240.06884623
3239.2161162
3238.39444623
3237.54683299
3236.64086261
3235.71735019
3234.7338854
3233.77110747
3232.74569866
3231.69231302
3230.68223382
3229.65408498
3228.67416712
3227.73050549
3226.77238191
3225.84598007
3224.90789993
3223.93891712
3222.97023299
3222.00328229
3221.01682062
3219.99573444
3218.98233768
3217.96397433
3217.02014452
3216.10645137
3215.26301316
3214.46094201
3213.69225371
3212.97453282
3212.37486606
3211.88183289
3211.43154754
3210.97675943
3210.54908169
3210.13502517
3209.73531676
3209.36218171
3208.99514668
3208.66323696
3208.39128699
3208.12054437
3207.90388395
3207.68129653
3207.41962813
3207.23391386
3207.10962751
3207.00701309
3206.91310546
3206.76792077
3206.617873
3206.50512523
3206.40682189
3206.30562406
3206.15769083
3205.98896221
3205.793855
3205.59259045
3205.40111852
3205.14860068
3204.8188681
3204.40861263
3203.98272656
3203.70308847
3203.99074546
```

Listing 22: scaledown output

7 References

[1] Toby Segaran. *Programming Collective Intelligence.* O'Reilly, first edition, 2007.

[2] Internet Engineering Task Force (IETF). RFC-4287 The Atom Syndication Format. https://tools.ietf.org/html/rfc4287, 2005.