

Assignment 8

Fall 2014

CS595 Web Science

Dr. Michael Nelson

Mathew Chaney

December 1, 2014

Contents

1	Question 1	4
1.1	Question	4
1.2	Answer	4
2	Question 2	9
2.1	Question	9
2.2	Answer	9
3	Question 3	12
3.1	Question	12
3.2	Answer	12
4	Question 4	14
4.1	Question	14
4.2	Answer	14
5	Question 5	16
5.1	Question	16
5.2	Answer	16
6	Appendix A	17
7	References	22

Listings

1	main for get_uris.py	4
2	referenced variables in get_uris.py	4
3	get_atom function	5
4	add_uri function	5
5	looping over the URIs	5
6	processing each blog	6
7	creating the blog data matrix	6
8	loading the data	7
9	building the master wordlist	7
10	writing the data	7
11	building the histogram	7
12	creating the dendrograms	9
13	creating the dendrograms	9
14	hcluster function	9
15	printclust function	10
16	drawdendrogram function	10
17	kclustering main	12
18	kcluster function	12
19	output of kclustering algorithm	13
20	main for scaledown	14
21	scaledown function	14
22	draw2d function	15
23	get_uris.py	17
24	matrix.py	18

25 scaledown output 20

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, which can be found in Appendix A, Listing 23. Two default blogs, F-Measure and the Old Dominion Web Science and Digital Libraries blogs, were added as defaults to the initial URI list and then, using the seed URI provided (Listing 2), the remaining 98 URIs from random blogs within the blogger.com family were added.

```
27 if __name__ == '__main__':  
28     uris = set()  
29     with open('blog_uris', 'a') as outfile:  
30         if len(sys.argv) > 1 and sys.argv[1] == 'new':  
31             for must_have in must_haves:  
32                 uri = get_atom(must_have)  
33                 add_uri(uri, uris, outfile)  
34         else:  
35             with open('blog_uris') as infile:  
36                 [uris.add(line.strip()) for line in infile]  
37         while len(uris) < 100:  
38             uri = get_atom(default)  
39             add_uri(uri, uris, outfile)
```

Listing 1: main for `get_uris.py`

```
7 default = 'http://www.blogger.com/next-blog?navBar=true&blogID=3471633091411211117'  
8 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 [1] URIs from each blog and add them to the set of URIs with the `add_uri` function, shown in Listing 4.

```

10 def get_atom(uri):
11     try:
12         r = requests.get(uri)
13     except Exception, e:
14         return None
15     soup = BeautifulSoup(r.text)
16     links = soup.find_all('link', {'type': 'application/atom+xml'})
17     if links:
18         return str(links[0]['href'])
19     return None

```

Listing 3: get_atom function

```

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

```

Listing 4: add_uri function

After the full list of 100 URIs was obtained page counts for each blog were extracted and saved to a file called `pagecounts` using the `matrix.py` script. This script is a modified version of `generatefeedvectors.py` from the book *Programming Collective Intelligence* [2] and can be found in full in Appendix A, Listing 24.

The code responsible for downloading the blogs and counting the words in each is shown in Listing 5, which calls the `get_titles`, `get_words` and `get_next` functions found in Listing 6. This code loops over the list of URIs that was created with the `get_uris.py` script (Listing 23), parses each entry, and extracts all the words in each entry's title.

```

95 if __name__ == '__main__':
96     with open('blog_uris') as infile:
97         uris = [line.strip() for line in infile if line.strip()]
98         if len(sys.argv) == 2 and sys.argv[1] == 'get':
99             with futures.ThreadPoolExecutor(max_workers=8) as executor:
100                 uri_futures = [executor.submit(get_titles, uri) for uri in uris]
101                 for future in futures.as_completed(uri_futures):
102                     uri, title, subtitle, pages, wc = future.result()
103                     with open('wcs/' + md5.new(uri).hexdigest() + '.w') as out:
104                         out.write(title + ': ' + subtitle + '\t' + str(pages) + '\t')
105                     json.dump(wc, out)

```

Listing 5: looping over the URIs

```

9 def get_next(d):
10     for item in d.feed.links:
11         if item['rel'] == u'next':
12             return item['href']
13     return None
14
15 def get_words(text):
16     txt = re.compile(r'<[>]+>').sub('', text)
17     words = re.compile(r'^A-Z^a-z+').split(txt)
18     return [word.lower() for word in words if word != '']
19
20 def get_titles(uri):
21     print('processing {}'.format(uri))
22     next = uri
23     wc = {}
24     pages = 0
25     while next is not None:
26         d = feedparser.parse(next)
27         for e in d.entries:
28             words = get_words(e.title.encode('utf-8'))
29             for word in words:
30                 wc.setdefault(word, 0)
31                 wc[word] += 1
32             pages += 1
33             next = get_next(d)
34             print('next {}'.format(next))
35     title = d.feed.title.encode('utf-8')
36     subtitle = d.feed.subtitle[:50].encode('utf-8')
37     print('finished: {}: {}'.format(title, subtitle))
38     return uri, title, subtitle, pages, wc

```

Listing 6: processing each blog

The parsed results were then read by the code in Listing 7. This code used the `load_data` functions in Listing 8 and 9 to read each of the blog word counts and then created four collections to organize them all:

1. **apcount**: A dictionary containing the count for all words combined
2. **wordcounts**: A dictionary containing each blog's individual word count
3. **pagecounts**: A dictionary containing each blog's page count
4. **wordlist**: A list containing all of the words found in each blog

```

107     apcount, wordcounts, pagecounts = load_data(uris)
108     wordlist = build_wordlist(apcount, uris)
109     if len(sys.argv) == 2 and sys.argv[1] == 'pages':
110         with open('pagecounts', 'w') as outfile:
111             outfile.write('blog\tpages\n')
112             for blog, pagecount in pagecounts.iteritems():
113                 outfile.write("\"\"\" + blog.replace("\"\"", "") + "\"\"\" + '\t' + str(
114                     pagecount) + '\n')
115     elif len(sys.argv) == 2 and sys.argv[1] == 'wc':
116         write_data('blogdata1.txt', wordlist, wordcounts)

```

Listing 7: creating the blog data matrix

```

50 def load_data(uris):
51     apcount = {}
52     wordcounts = {}
53     pagecounts = {}
54     for uri in uris:
55         with open('wcs/' + md5.new(uri).hexdigest()) as infile:
56             try:
57                 lines = infile.read().split('\t')
58                 title = lines[0]
59                 pages = int(lines[1])
60                 wc = json.loads(lines[2])
61             except Exception, e:
62                 print('*** {} generated an exception: {}'.format(uri, e))
63                 continue
64             wordcounts[title] = wc
65             pagecounts[title] = pages
66             for word, count in wc.items():
67                 apcount.setdefault(word, 0)
68                 apcount[word] += count
69     return apcount, wordcounts, pagecounts

```

Listing 8: loading the data

```

71 def build_wordlist(apcount, uris):
72     wordlist = []
73     for w, bc in sorted(apcount.items(), key=lambda x: x[1], reverse=True):
74         frac = float(bc) / len(uris)
75         if frac > 0.1 and frac < 0.5:
76             wordlist.append(w)
77     return wordlist

```

Listing 9: building the master wordlist

The code in Listing 10 then created the matrix using the `write_data` function.

```

79 def write_data(filename, wordlist, wordcounts, form=lambda wc, word, wordcounts: wc[word]):
80     with open(filename, 'w') as out:
81         out.write('Blog')
82         for word in wordlist[:500]:
83             out.write('\t%s' % word)
84         out.write('\n')
85         for blog, wc in wordcounts.items():
86             print blog
87             out.write(blog)
88             for word in wordlist[:500]:
89                 if word in wc:
90                     out.write('\t{}'.format(form(wc, word, wordcounts)))
91                 else:
92                     out.write('\t0')
93             out.write('\n')

```

Listing 10: writing the data

To build a histogram showing the blog page counts, the `pagecounts` file was parsed by the R script in Listing 11 and saved as a pdf, which is shown in Figure 1.

```

1 #! /usr/bin/Rscript
2
3 data <- read.table("pagecounts", sep="\t", header=TRUE, comment.char="")
4 counts <- table(data$pages)
5 pdf("hist.pdf")
6 barplot(counts, ylab="Number of Blogs", xlab="Page Count", main="Page Count per Blog")
7 dev.off()

```

Listing 11: building the histogram

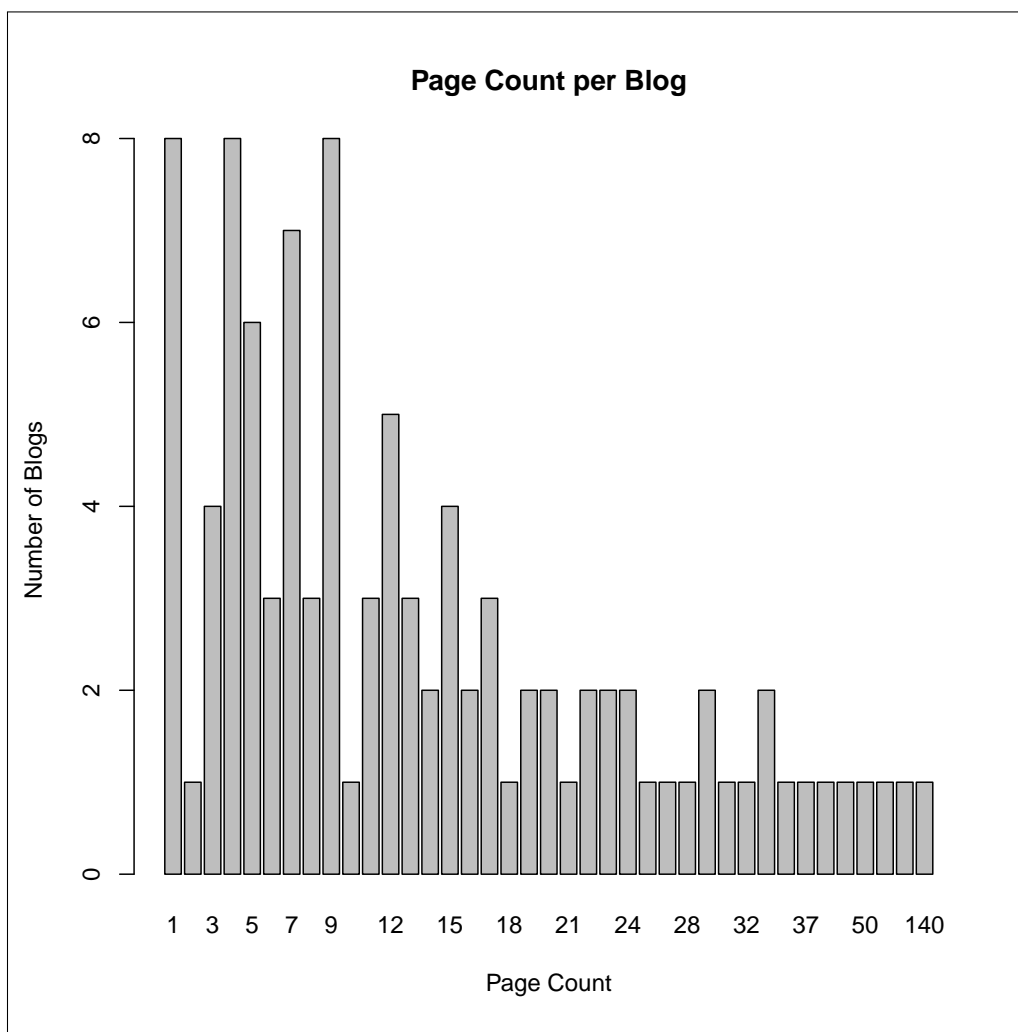


Figure 1: Page Count per Blog

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 12, which is modeled after the example from class.

```
286 blognames, words, data = readfile('blogdata1.txt')
287 clust = hcluster(data)
288 with open('dendrogram.txt', 'w') as outfile:
289     stdout = sys.stdout
290     sys.stdout = outfile
291     printclust(clust, labels=blognames)
292     sys.stdout = stdout
293 drawdendrogram(clust, blognames, jpeg='blogclust.jpg')
```

Listing 12: creating the dendrograms

This uses the `readfile` function shown in Listing 13 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 14 to produce the clustered representation of the blogs.

```
3 def readfile(filename):
4     lines=[line for line in file(filename)]
5
6     # First line is the column titles
7     colnames=lines[0].strip().split('\t')[1:]
8     rownames=[]
9     data=[]
10    for line in lines[1:]:
11        p=line.strip().split('\t')
12        # First column in each row is the rowname
13        rownames.append(p[0])
14        # The data for this row is the remainder of the row
15        data.append([float(x) for x in p[1:]])
16    return rownames, colnames, data
```

Listing 13: creating the dendrograms

```
48 def hcluster(rows, distance=pearson):
49     distances={}
50     currentclustid=-1
51
52     # Clusters are initially just the rows
53     clust=[bicluster(rows[i], id=i) for i in range(len(rows))]
54
55     while len(clust)>1:
56         lowestpair=(0,1)
57         closest=distance(clust[0].vec, clust[1].vec)
58
59         # loop through every pair looking for the smallest distance
60         for i in range(len(clust)):
61             for j in range(i+1, len(clust)):
62                 # distances is the cache of distance calculations
63                 if (clust[i].id, clust[j].id) not in distances:
64                     distances[(clust[i].id, clust[j].id)]=distance(clust[i].vec, clust[j].vec)
65
66                 d=distances[(clust[i].id, clust[j].id)]
67
68                 if d<closest:
69                     closest=d
```

```

70         lowestpair=(i,j)
71
72     # calculate the average of the two clusters
73     mergevec=[
74         (clust[lowestpair[0]].vec[i]+clust[lowestpair[1]].vec[i])/2.0
75         for i in range(len(clust[0].vec))
76     ]
77     # create the new cluster
78     newcluster=biclusterm(mergevec,left=clust[lowestpair[0]],
79                           right=clust[lowestpair[1]],
80                           distance=closest,id=currentclustid)
81
82     # cluster ids that weren't in the original set are negative
83     currentclustid-=1
84     del clust[lowestpair[1]]
85     del clust[lowestpair[0]]
86     clust.append(newcluster)
87
88     return clust[0]

```

Listing 14: hcluster function

The `printclust` function from Listing 15 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 12.

```

90 def printclust(clust, labels=None, n=0):
91     # indent to make a hierarchy layout
92     for i in range(n): print ' ',
93     if clust.id<0:
94         # negative id means that this is branch
95         print '- ',
96     else:
97         # positive id means that this is an endpoint
98         if labels==None: print clust.id
99         else: print labels[clust.id]
100
101     # now print the right and left branches
102     if clust.left!=None: printclust(clust.left, labels=labels, n=n+1)
103     if clust.right!=None: printclust(clust.right, labels=labels, n=n+1)

```

Listing 15: printclust function

The `drawdendrogram` function from Listing 16 creates a jpeg image of the cluster, which is shown in Figure 2.

```

122 def drawdendrogram(clust, labels, jpeg='clusters.jpg'):
123     # height and width
124     h=getheight(clust)*20
125     w=1200
126     depth=getdepth(clust)
127
128     # width is fixed, so scale distances accordingly
129     scaling=float(w-150)/depth
130
131     # Create a new image with a white background
132     img=Image.new('RGB',(w,h),(255,255,255))
133     draw=ImageDraw.Draw(img)
134
135     draw.line((0,h/2,10,h/2),fill=(255,0,0))
136
137     # Draw the first node
138     drawnode(draw,clust,10,(h/2),scaling,labels)
139     img.save(jpeg,'JPEG')

```

Listing 16: drawdendrogram function

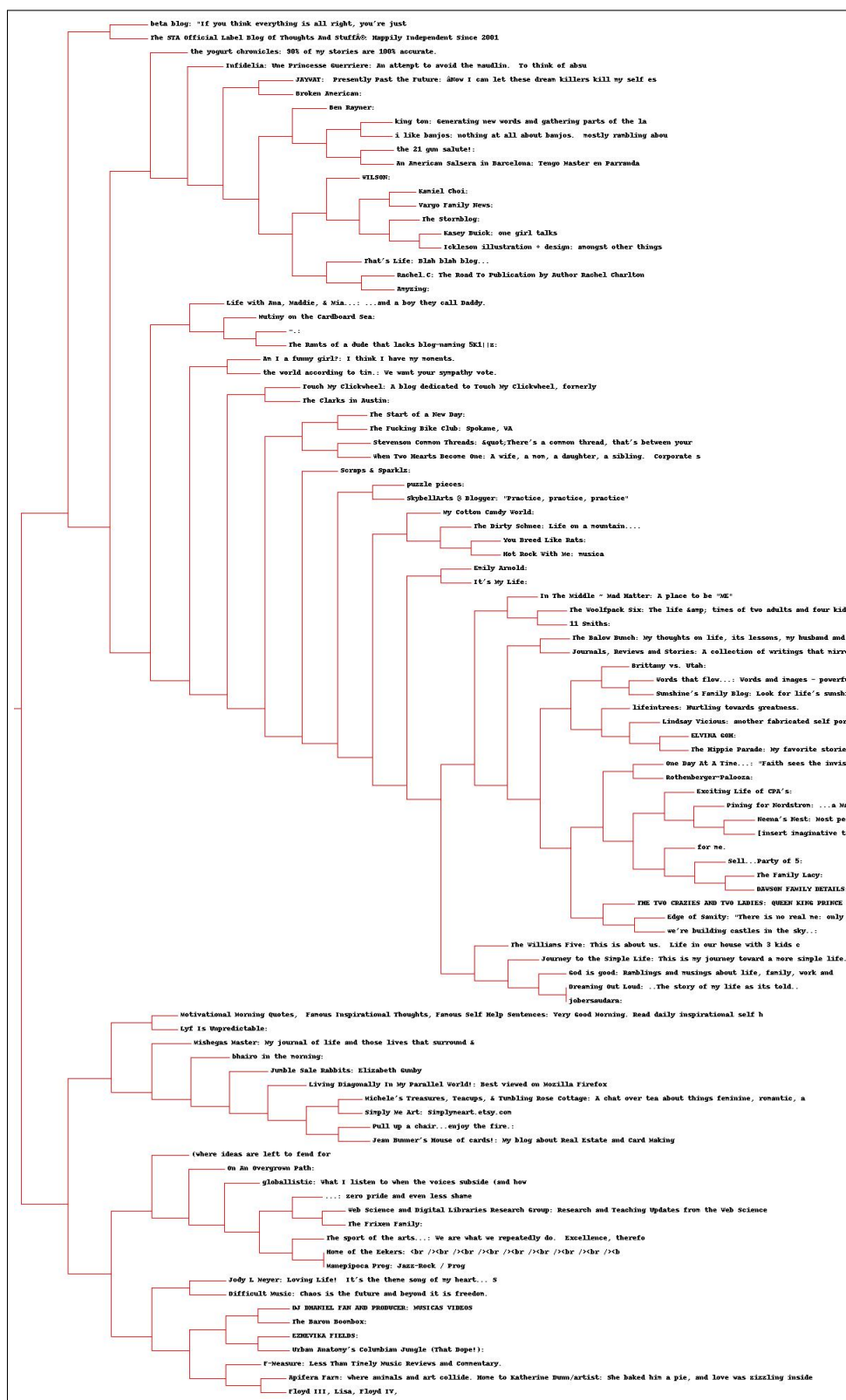


Figure 2: dendrogram

3 Question 3

3.1 Question

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

3.2 Answer

Using the code in Listing 17 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 18.

```
295 print "K=5"
296 kclust=kcluster(data, k=5)
297 print "K=10"
298 kclust=kcluster(data, k=10)
299 print "K=20"
300 kclust=kcluster(data, k=20)
```

Listing 17: kclustering main

```
174 def kcluster(rows,distance=pearson,k=4):
175     # Determine the minimum and maximum values for each point
176     ranges=[(min([row[i] for row in rows]),max([row[i] for row in rows]))
177             for i in range(len(rows[0]))]
178
179     # Create k randomly placed centroids
180     clusters=[[random.random()*(ranges[i][1]-ranges[i][0])+ranges[i][0]
181               for i in range(len(rows[0]))] for j in range(k)]
182
183     lastmatches=None
184     for t in range(100):
185         print 'Iteration %d' % t
186         bestmatches=[] for i in range(k)]
187
188         # Find which centroid is the closest for each row
189         for j in range(len(rows)):
190             row=rows[j]
191             bestmatch=0
192             for i in range(k):
193                 d=distance(clusters[i],row)
194                 if d<distance(clusters[bestmatch],row): bestmatch=i
195             bestmatches[bestmatch].append(j)
196
197         # If the results are the same as last time, this is complete
198         if bestmatches==lastmatches: break
199         lastmatches=bestmatches
200
201         # Move the centroids to the average of their members
202         for i in range(k):
203             avgs=[0.0]*len(rows[0])
204             if len(bestmatches[i])>0:
205                 for rowid in bestmatches[i]:
206                     for m in range(len(rows[rowid])):
207                         avgs[m]+=rows[rowid][m]
208                 for j in range(len(avgs)):
209                     avgs[j]/=len(bestmatches[i])
210                 clusters[i]=avgs
211
212     return bestmatches
```

Listing 18: kcluster function

The output is shown in Listing 19. As the output reads, a kcluster with $n = 5$ required nine iterations, $n = 10$ required four iterations and $n = 20$ also required four iterations.

```
1 Done with dendrograms
2 K=5
3 Iteration 0
4 Iteration 1
5 Iteration 2
6 Iteration 3
7 Iteration 4
8 Iteration 5
9 Iteration 6
10 Iteration 7
11 Iteration 8
12 K=10
13 Iteration 0
14 Iteration 1
15 Iteration 2
16 Iteration 3
17 K=20
18 Iteration 0
19 Iteration 1
20 Iteration 2
21 Iteration 3
```

Listing 19: 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 20, multidimensional scaling (MDS) was used to create a two-dimensional visualization of the blog distance graph.

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

Listing 20: main for scaledown

This main code calls the scaledown function, which is shown in Listing 21. The algorithm continues until the error factor stops decreasing, as shown in the output in Appendix A, Listing 25.

```
224 def scaledown(data,distance=pearson,rate=0.01):
225     n=len(data)
226
227     # The real distances between every pair of items
228     realdist=[[distance(data[i],data[j]) for j in range(n)]
229               for i in range(0,n)]
230
231     # Randomly initialize the starting points of the locations in 2D
232     loc=[[random.random(),random.random()] for i in range(n)]
233     fakedist=[[0.0 for j in range(n)] for i in range(n)]
234
235     lasterror=None
236     for m in range(0,1000):
237         # Find projected distances
238         for i in range(n):
239             for j in range(n):
240                 fakedist[i][j]=sqrt(sum([pow(loc[i][x]-loc[j][x],2)
241                                           for x in range(len(loc[i]))]))
242
243         # Move points
244         grad=[[0.0,0.0] for i in range(n)]
245
246         totalerror=0
247         for k in range(n):
248             for j in range(n):
249                 if j==k: continue
250                 # The error is percent difference between the distances
251                 if realdist[j][k] != 0:
252                     errorterm=(fakedist[j][k]-realdist[j][k])/realdist[j][k]
253
254                 # Each point needs to be moved away from or towards the other
255                 # point in proportion to how much error it has
256                 grad[k][0]+=((loc[k][0]-loc[j][0])/fakedist[j][k])*errorterm
257                 grad[k][1]+=((loc[k][1]-loc[j][1])/fakedist[j][k])*errorterm
258
259                 # Keep track of the total error
260                 totalerror+=abs(errorterm)
261         print totalerror
262
263         # If the answer got worse by moving the points, we are done
264         if lasterror and lasterror<totalerror: break
265         lasterror=totalerror
266
267         # Move each of the points by the learning rate times the gradient
268         for k in range(n):
269             loc[k][0]-=rate*grad[k][0]
270             loc[k][1]-=rate*grad[k][1]
271
272     return loc
```

Listing 21: 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 22, which produced the two-dimensional visualization created from the MDS algorithm, as shown in Figure 3.

```

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

```

Listing 22: 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

```
1 #!/usr/bin/env python
2
3 import requests
4 import sys
5 from bs4 import BeautifulSoup
6
7 default = 'http://www.blogger.com/next-blog?navBar=true&blogID=3471633091411211117'
8 must_haves = ['http://f-measure.blogspot.com/', 'http://ws-dl.blogspot.com/']
9
10 def get_atom(uri):
11     try:
12         r = requests.get(uri)
13     except Exception, e:
14         return None
15     soup = BeautifulSoup(r.text)
16     links = soup.find_all('link', {'type': 'application/atom+xml'})
17     if links:
18         return str(links[0]['href'])
19     return None
20
21 def add_uri(uri, uris, outfile):
22     if uri and uri not in uris:
23         uris.add(uri)
24         outfile.write(uri + '\n')
25         print len(uris), uri
26
27 if __name__ == '__main__':
28     uris = set()
29     with open('blog_uris', 'a') as outfile:
30         if len(sys.argv) > 1 and sys.argv[1] == 'new':
31             for must_have in must_haves:
32                 uri = get_atom(must_have)
33                 add_uri(uri, uris, outfile)
34         else:
35             with open('blog_uris') as infile:
36                 [uris.add(line.strip()) for line in infile]
37     while len(uris) < 100:
38         uri = get_atom(default)
39         add_uri(uri, uris, outfile)
```

Listing 23: get_uris.py

```

1 import feedparser
2 import futures
3 import math
4 import md5
5 import re
6 import sys
7 import json
8
9 def get_next(d):
10     for item in d.feed.links:
11         if item['rel'] == u'next':
12             return item['href']
13     return None
14
15 def get_words(text):
16     txt = re.compile(r'<[^>]+>').sub('', text)
17     words = re.compile(r'^A-Z^a-z+').split(txt)
18     return [word.lower() for word in words if word != '']
19
20 def get_titles(uri):
21     print('processing {}'.format(uri))
22     next = uri
23     wc = {}
24     pages = 0
25     while next is not None:
26         d = feedparser.parse(next)
27         for e in d.entries:
28             words = get_words(e.title.encode('utf-8'))
29             for word in words:
30                 wc.setdefault(word, 0)
31                 wc[word] += 1
32             pages += 1
33             next = get_next(d)
34             print('next {}'.format(next))
35         title = d.feed.title.encode('utf-8')
36         subtitle = d.feed.subtitle[:50].encode('utf-8')
37         print('finished: {}: {}'.format(title, subtitle))
38         return uri, title, subtitle, pages, wc
39
40 def tf(wc, word):
41     return float(wc[word]) / float(sum(wc.values()))
42
43 def idf(wordcounts, word):
44     present = 0
45     for wc in wordcounts.values():
46         if word in wc:
47             present += 1
48     return math.log(len(wordcounts) / present, 2)
49
50 def load_data(uris):
51     apcount = {}
52     wordcounts = {}
53     pagecounts = {}
54     for uri in uris:
55         with open('wcs/' + md5.new(uri).hexdigest()) as infile:
56             try:
57                 lines = infile.read().split('\n')
58                 title = lines[0]
59                 pages = int(lines[1])
60                 wc = json.loads(lines[2])
61             except Exception, e:
62                 print('*** {} generated an exception: {}'.format(uri, e))
63                 continue
64             wordcounts[title] = wc
65             pagecounts[title] = pages
66             for word, count in wc.items():
67                 apcount.setdefault(word, 0)
68                 apcount[word] += count
69     return apcount, wordcounts, pagecounts
70
71 def build_wordlist(apcount, uris):
72     wordlist = []
73     for w, bc in sorted(apcount.items(), key=lambda x: x[1], reverse=True):
74         frac = float(bc) / len(uris)
75         if frac > 0.1 and frac < 0.5:
76             wordlist.append(w)

```

```

77     return wordlist
78
79 def write_data(filename, wordlist, wordcounts, form=lambda wc, word, wordcounts: wc[word]):
80     with open(filename, 'w') as out:
81         out.write('Blog')
82         for word in wordlist[:500]:
83             out.write('\t%s' % word)
84         out.write('\n')
85         for blog, wc in wordcounts.items():
86             print blog
87             out.write(blog)
88             for word in wordlist[:500]:
89                 if word in wc:
90                     out.write('\t{}'.format(form(wc, word, wordcounts)))
91                 else:
92                     out.write('\t0')
93             out.write('\n')
94
95 if __name__ == '__main__':
96     with open('blog_uris') as infile:
97         uris = [line.strip() for line in infile if line.strip()]
98         if len(sys.argv) == 2 and sys.argv[1] == 'get':
99             with futures.ThreadPoolExecutor(max_workers=8) as executor:
100                 uri_futures = [executor.submit(get_titles, uri) for uri in uris]
101                 for future in futures.as_completed(uri_futures):
102                     uri, title, subtitle, pages, wc = future.result()
103                     with open('wcs/' + md5.new(uri).hexdigest(), 'w') as out:
104                         out.write(title + ': ' + subtitle + '\t' + str(pages) + '\t')
105                     json.dump(wc, out)
106             else:
107                 apcount, wordcounts, pagecounts = load_data(uris)
108                 wordlist = build_wordlist(apcount, uris)
109                 if len(sys.argv) == 2 and sys.argv[1] == 'pages':
110                     with open('pagecounts', 'w') as outfile:
111                         outfile.write('blog\tpages\n')
112                         for blog, pagecount in pagecounts.iteritems():
113                             outfile.write("'" + blog.replace("\n", "") + "'" + "\n" + '\t' + str(
114                                 pagecount) + '\n')
115                 elif len(sys.argv) == 2 and sys.argv[1] == 'wc':
116                     write_data('blogdata1.txt', wordlist, wordcounts)
117                 elif len(sys.argv) == 2 and sys.argv[1] == 'tfidf':
118                     write_data('blogdata2.txt', wordlist, wordcounts, form=lambda wc, word,
119                                 wordcounts: tf(wc, word) * idf(wordcounts, word))

```

Listing 24: matrix.py

1	5114.96392059
2	3378.46476941
3	3336.07313947
4	3321.07681636
5	3314.82206924
6	3311.9923119
7	3310.01886371
8	3308.22374881
9	3306.81320798
10	3305.60952179
11	3304.38029019
12	3303.24545568
13	3302.30781839
14	3301.243952
15	3299.95637566
16	3298.77418812
17	3297.63109703
18	3296.49161679
19	3295.11129947
20	3294.10589657
21	3293.18215656
22	3292.22035202
23	3291.35503982
24	3290.54419166
25	3289.68063935
26	3288.84471038
27	3287.93583439
28	3286.81371666
29	3285.75771263
30	3284.77259039
31	3283.77319553
32	3282.98356628
33	3282.34595444
34	3281.76444272
35	3281.14689344
36	3280.34170838
37	3279.58018102
38	3278.80573491
39	3277.9717399
40	3277.12547791
41	3276.22103528
42	3275.36339745
43	3274.49949723
44	3273.66834004
45	3272.88975764
46	3272.10968588
47	3271.20916637
48	3270.26089512
49	3269.2602024
50	3268.24022652
51	3267.24447156
52	3266.15717891
53	3265.11562966
54	3264.09645474
55	3263.10678415
56	3262.17405097
57	3261.24269022
58	3260.24197749
59	3259.21770939
60	3258.21872025
61	3257.21365814
62	3256.14783975
63	3255.07114015
64	3253.95694354
65	3252.82012061
66	3251.74546935
67	3250.68451973
68	3249.66129176
69	3248.70073351
70	3247.79195239
71	3246.95679073
72	3246.21969935
73	3245.51207638
74	3244.76825233
75	3244.01399188
76	3243.24587537

```
77 3242.48283094
78 3241.71482665
79 3240.90414524
80 3240.06884623
81 3239.2161162
82 3238.39444623
83 3237.54683299
84 3236.64086261
85 3235.71735019
86 3234.7338854
87 3233.77110747
88 3232.74569866
89 3231.69231302
90 3230.68223382
91 3229.65408498
92 3228.67416712
93 3227.73050549
94 3226.77238191
95 3225.84598007
96 3224.90789993
97 3223.93891712
98 3222.97023299
99 3222.00328229
100 3221.01682062
101 3219.99573444
102 3218.98233768
103 3217.96397433
104 3217.02014452
105 3216.10645137
106 3215.26301316
107 3214.46094201
108 3213.69225371
109 3212.97453282
110 3212.37486606
111 3211.88183289
112 3211.43154754
113 3210.97675943
114 3210.54908169
115 3210.13502517
116 3209.73531676
117 3209.36218171
118 3208.99514668
119 3208.66323696
120 3208.39128699
121 3208.12054437
122 3207.90388395
123 3207.68129653
124 3207.41962813
125 3207.23391386
126 3207.10962751
127 3207.00701309
128 3206.91310546
129 3206.76792077
130 3206.617873
131 3206.50512523
132 3206.40682189
133 3206.30562406
134 3206.15769083
135 3205.98896221
136 3205.793855
137 3205.59259045
138 3205.40111852
139 3205.14860068
140 3204.8188681
141 3204.40861263
142 3203.98272656
143 3203.70308847
144 3203.99074546
```

Listing 25: scaledown output

7 References

- [1] Internet Engineering Task Force (IETF). RFC-4287 The Atom Syndication Format. <https://tools.ietf.org/html/rfc4287>, 2005.
- [2] Toby Segaran. *Programming Collective Intelligence*. O'Reilly, first edition, 2007.