

Assignment 9

CS 595: Introduction to Web Science

Fall 2013

Shawn M. Jones

Finished on December 5, 2013

1

Question

1. 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.

Answer

The script `fetchFeeds.py` shown in Listing 1 outputs a list of URIs that dereference to Atom feeds.

This script uses a canned URI on line 54 to acquire the next blog URI via the call on line 82. If it failed to acquire this URI, it sleeps for 5 seconds and tries again. If successful, it then dereferences the URI using the function defined on line 12. This function returns the representation of the resource, as well as its URI after all redirects have been followed.

If that is successful, it uses the function defined on line 21 to extract the Atom URI from the blog's HTML. If that is successful, it then tries to dereference the Atom feed URI, again acquiring the representation and the redirect-free version of the atom URI. Then it tests the URI using the function defined on line 33. This function checks to ensure that the blog has at least 25 entries. Attempts to ensure that the blog was English did not work well, as all Blogger pages use a character encoding of UTF-8, and detecting 'English' (maybe using a dictionary?) appeared to be outside the realm of this assignment.

Once the URI gets through all of those wickets, it is saved to a list, with the query string "`?max-results=1000`" appended. This ensures that we get at most 1000 entries from each blog, providing the maximum amount of data

for the following questions. Because all blogs come from Blogger, which has a consistent API, this query string works every time.

```
1  #!/usr/local/bin/python
2
3  import sys
4  import os
5  import os.path
6  import feedparser
7  import urllib2
8  import time
9  import chardet
10 from bs4 import BeautifulSoup
11
12 def dereferenceUri(uri):
13
14     pagehandle = urllib2.urlopen(uri)
15     pagedata = pagehandle.read()
16     derefurl = pagehandle.geturl()
17     pagehandle.close()
18
19     return pagedata, derefurl
20
21 def getAtomFeedUri(html):
22
23     soup = BeautifulSoup(html)
24
25     atomLinks = soup.find_all('link',
26                               attrs = { 'rel' : 'alternate', 'type' : 'application/
27                                         atom+xml' })
28
29     # we assume there is only one atom link
30     atomURI = atomLinks[0].attrs['href']
31
32     return atomURI
33
34 def meetsCriteria(feedText):
35
36     parsedData = feedparser.parse(feedText)
37
38     # assume we're good to go by default (fail optimistic?)
39     goodToGo = True
40
41     sys.stderr.write("blog has " + str(len(parsedData.entries))
42                     + " entries\n")
43
44     if (len(parsedData.entries) < 25):
45         goodToGo = False
```

```

45     #if (chardet.detect(feedText)['encoding'] != 'ascii'):
46     #     sys.stderr.write("blog charset is " + chardet.detect(
47         feedText)['encoding'] +
48         #         ", likely won't parse well for feed vector\n")
49     #     goodToGo = False
50
51     return goodToGo
52
53 def getNextUri():
54
55     uri = "http://www.blogger.com/next-blog?navBar=true&blogID
56         =3471633091411211117"
57
58     pagehandle = urllib2.urlopen(uri)
59     nexturi = pagehandle.geturl()
60     pagehandle.close()
61
62     return nexturi
63
64 if __name__ == "__main__":
65
66     feedlist = []
67     feedlist.append("http://f-measure.blogspot.com/feeds/posts/
68         default?max-results=200")
69     feedlist.append("http://ws-dl.blogspot.com/feeds/posts/
70         default?max-results=200")
71
72     while (len(feedlist) <= 100):
73
74         try:
75             # Tried these steps, but realized I would have to
76             # parse JavaScript
77             # * look for the iframe containing the next button
78             #     <iframe name='navbar-iframe'...
79             # iframeURI = getIframeUri(html)
80             # * dereference the link from that iframe
81             # iframeText = getIframeText(iframeURI)
82             # * extract the uri form
83             # <a class="b-link" href="...">Next Blog</a>
84             # uri = getNextUri(iframeText)
85             uri = getNextUri()
86         except urllib2.HTTPError as e:
87             sys.stderr.write("failed to acquire next uri,
88                 delaying 5 seconds\n")
89             time.sleep(5)
90         else:
91             sys.stderr.write("working on URI " + uri + "\n")

```

```

89         try:
90             # dereference the uri and get text
91             html,derefuri = dereferenceUri(uri)
92         except urllib2.HTTPError as e:
93             sys.stderr.write("failed to dereference " + uri
94                             +
95                             ", delaying 5 seconds\n")
96             time.sleep(5)
97         else:
98             try:
99                 # fetch the atom feed URI
100                 feedURI = getAtomFeedUri(html)
101                 sys.stderr.write("acquired feed URI " +
102                                 feedURI + "\n")
103             except IndexError as e:
104                 sys.stderr.write(
105                     "failed to acquire Atom feed from HTML,
106                     delaying 5 seconds\n")
107             else:
108                 try:
109                     # get the atom feed text
110                     feedText,feedURI = dereferenceUri(
111                         feedURI)
112                 except urllib2.HTTPError as e:
113                     sys.stderr.write("failed to dereference
114                                     " + feedURI +
115                                     ", delaying 5 seconds\n")
116                     time.sleep(5)
117                 else:
118                     # if it meets the criteria , save the
119                     # file
120                     if meetsCriteria(feedText):
121                         sys.stderr.write("Saving blog feed "
122                                         + feedURI + "?max-results=1000\n")
123                         feedlist.append(feedURI + "?max-
124                                         results=1000")
125
126                     # be nice to the site
127                     time.sleep(1)
128
129     for feed in feedlist:
130         print feed

```

Listing 1: Python script for fetching valid Atom feeds from Blogger

The script is run like so:

```
./fetchFeeds.py > feedlist.txt
```

Once `feedlist.txt` exists, it can be used by Toby Segaran's `generatefeedvector.py`[1]. I only modified that script on line 59 so that it could handle UTF-8 encodings for some of the blogs. It is captured in Listing 7 on page 22.

The script `eliminateWords.py` shown in Listing 2 takes in the `blogdata1.txt` file produced by `generatefeedvector.py` and removes the top N terms from it.

It does this by generating scores for each word by summing all of its frequencies across all blogs together. Once it has those scores, it gets the top n words. It then determines the index of each of those words in the list. Using these indices, it regenerates the format of `blogdata.txt`, only printing out each column if it is an “approved” index.

The final file is saved to github as `blogdata500.txt` and is used for each subsequent question.

```
1  #!/usr/local/bin/python
2
3  import sys
4
5  sys.path.insert(0, '../libs')
6
7  import clusters
8  import operator
9
10 import pprint
11
12 def getWordscores(words, data):
13
14     wordscores = {}
15
16     for i in range(len(words)):
17         sys.stderr.write('examining ' + words[i] + '\n')
18
19         for j in range(len(data)):
20
21             if words[i] in wordscores:
22                 wordscores[words[i]] += data[j][i]
23             else:
24                 wordscores[words[i]] = data[j][i]
25
26     return wordscores
27
28 def getTopNWords(wordscores, n):
29
30     topNWords = []
31
32     # thanks Stack Overflow:
```

```

33 # http://stackoverflow.com/questions/613183/python-sort-a-
    dictionary-by-value
34 reversedWordscores = sorted(
35     wordscores.iteritems(), key=operator.itemgetter(1),
        reverse=True
36 )
37
38 for i in range(n):
39     sys.stderr.write(
40         "adding " + reversedWordscores[i][0] + " with a
        score of "
41         + str(reversedWordscores[i][1]) + '\n'
42     )
43     topNWords.append(reversedWordscores[i][0])
44
45 return topNWords
46
47
48 if __name__ == '__main__':
49
50     n = int(sys.argv[1])
51
52     blognames, words, data = clusters.readfile('../q1/blogdata1.
        txt')
53
54     wordscores = getWordscores(words, data)
55
56     topNWords = getTopNWords(wordscores, n)
57
58     indexlist = []
59
60     for word in topNWords:
61         indexlist.append(words.index(word))
62
63     lines = []
64
65     line = []
66     line.append('Blog')
67
68     for i in range(len(words)):
69
70         if i in indexlist:
71             line.append(words[i])
72
73     lines.append(line)
74
75     for i in range(len(blognames)):
76         line = []
77         line.append(blognames[i])

```

```
78
79     for j in range(len(words)):
80         if j in indexlist:
81             line.append(str(int(data[i][j])))
82
83     lines.append(line)
84
85 for line in lines:
86     print "\t".join(line)
```

Listing 2: Python script for eliminating words from blog data

2

Question

2. 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).

Answer

The script `makeDendogram.py`, shown in Listing 3 uses Toby Segaran's *clusters.py* [1] shown on page 24.

```
1  #!/usr/local/bin/python
2
3  # all code here stolen shamelessly from
4  # "Programming Collective Intelligence, Chapter 3"
5
6  import sys
7
8  sys.path.insert(0, '../libs')
9
10 import clusters
11
12 blognames, words, data=clusters.readfile('../q1/blogdata500.txt')
13 clust = clusters.hcluster(data)
14
15 # print ASCII dendrogram
16 clusters.printclust(clust, labels=blognames)
17
18 # save JPEG dendrogram
19 clusters.drawdendrogram(clust, blognames, jpeg='blogclust.jpg')
```

Listing 3: Python script for generating dendograms from the blog data captured in question 1

It is run like so:

```
./makeDendogram.py > ascii-dendogram.txt
```

The `printclust` function on line 16 prints out the the dendrogram, so shell redirection is used to save it. The `drawdendrogram` function on line 19 saves a JPEG of the dendrogram, which can be seen in Figure 1.



Figure 1: Dendrogram produced by the *makeDendrogram.py* script

Unfortunately, it is difficult to see, but this dendogram shows that the blogs calculated to be most like *F-Measure* are *CRUCIAL CHANGES* and *The Poet You Never Were*. The blog calculated to be most like *Web Science and Digital Libraries Research Group* is *Providence Public Library Computer Classes*.

3

Question

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

Answer

The blog clustering is performed by the script shown in Listing 4, which makes use of Toby Segaran's *clusters.py* [1] on page 24., using the function *kcluster* on lines 15, 19, and 23.

```
1  #!/usr/local/bin/python
2
3  # all code here stolen shamelessly from
4  # "Programming Collective Intelligence, Chapter 3"
5
6  import sys
7
8  sys.path.insert(0, '../libs')
9
10 import clusters
11
12 blognames, words, data = clusters.readfile('../q1/blogdata500.txt')
13
14 print "For k=5"
15 kclust = clusters.kcluster(data, k=5)
16 print
17
18 print "For k=10"
19 kclust = clusters.kcluster(data, k=10)
20 print
21
22 print "For k=20"
23 kclust = clusters.kcluster(data, k=20)
24 print
```

Listing 4: Python script for clustering the blogs using K-means, using k=5, 10, and 20

This script is run like so:

```
./makeClusters.py
```

From its output, we see how many iterations each value of k produces.

```
For k=5
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4

For k=10
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7

For k=20
Iteration 0
Iteration 1
Iteration 2
Iteration 3
```

Thus, for $k = 5$ we get 5 iterations, for $k = 10$ we get 8 iterations, and for $k = 20$ we get 4 iterations.

4

Question

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

Answer

The blog space is generated using multidimensional scaling from the script `makeMDS.py` shown in Listing 5, which makes use of Toby Segaran's *clusters.py* [1] on page 24, using the functions *scaledown* on line 14 and *draw2d* on line 16.

```
1  #!/usr/local/bin/python
2
3  # all code here stolen shamelessly from
4  # "Programming Collective Intelligence, Chapter 3"
5
6  import sys
7
8  sys.path.insert(0, '../libs')
9
10 import clusters
11
12 blognames, words, data=clusters.readfile('../q1/blogdata500.txt')
13
14 coords = clusters.scaledown(data)
15
16 clusters.draw2d(coords, blognames, jpeg='blogs2d.jpg')
```

Listing 5: Python script for generating a MDS from the blog data

Again, unfortunately the blog space produced does not fit well on a letter-sized page shown in Figure 2. According to this two-dimensional representation, the blog closest to *F-Measure* is *Piazzolla on Video*. The blog closest to *Web Science and Digital Libraries Research Group* is *Christian Poet's Pen*.

Listing 6 shows the output from running this script, which took 253 iterations on this run.

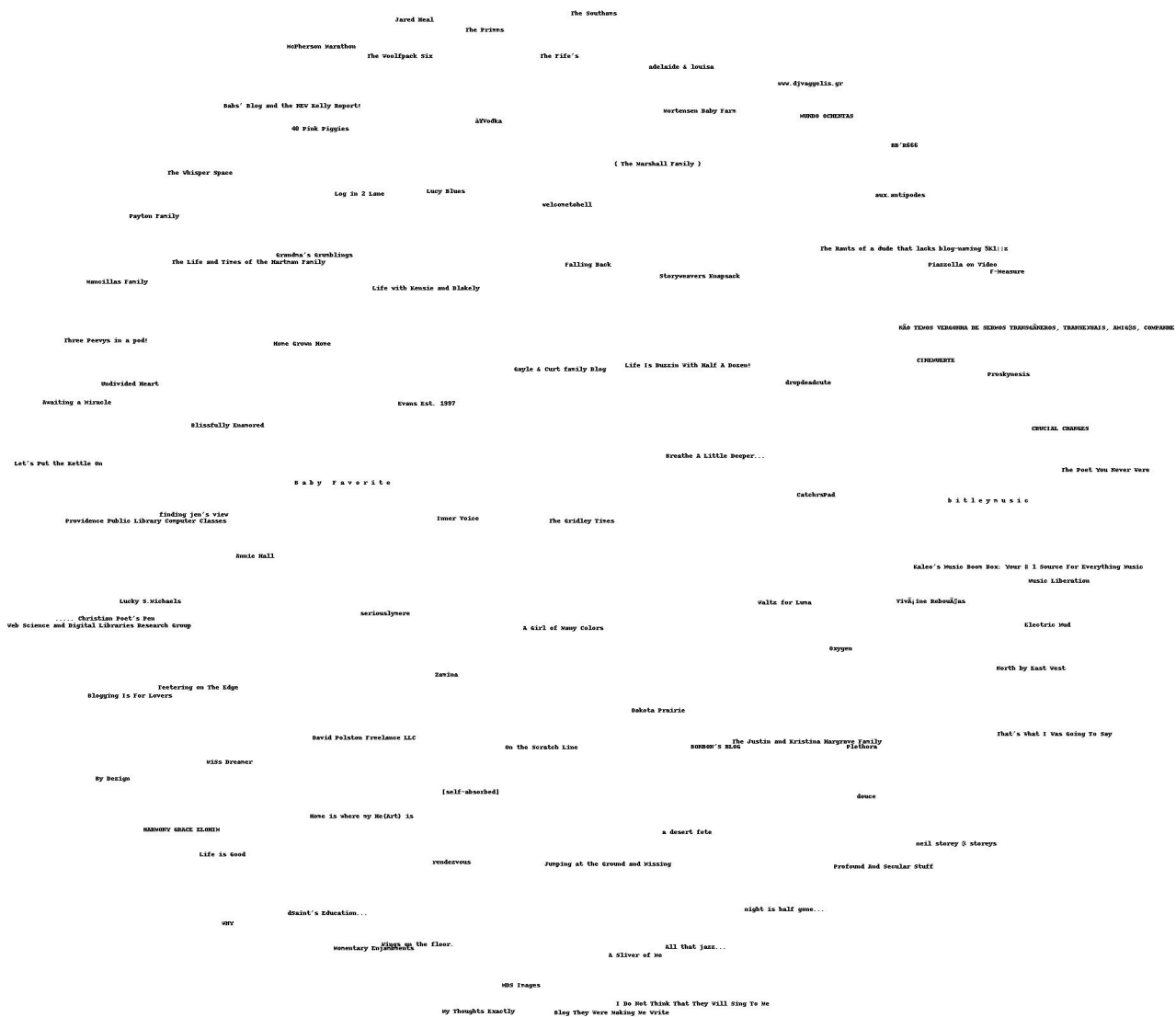


Figure 2: Blog space produced by the *makeMDS.py* script

1	4652.97312818
2	3444.54560754
3	3414.62730164
4	3400.67203169
5	3389.95858228
6	3381.16440101
7	3373.82386719
8	3367.21743327
9	3361.47597257
10	3356.87267707
11	3353.17006145
12	3349.91959467
13	3346.50445159
14	3342.7712176
15	3339.40810563
16	3336.18749192
17	3333.36565585
18	3331.08631674
19	3328.61026323
20	3326.17361035
21	3323.78500387
22	3321.22819411
23	3318.49250313
24	3315.69908383
25	3312.6086601
26	3309.26638909
27	3305.75947077
28	3302.29216809
29	3298.74957052
30	3295.19449667
31	3291.815478
32	3288.95965779
33	3286.11763062
34	3283.54487867
35	3281.2166054
36	3279.15760563
37	3277.20984653
38	3275.35728663
39	3273.44230762
40	3271.73667048
41	3270.01520471
42	3268.50170187
43	3267.07414959
44	3265.63691071
45	3264.18440046
46	3262.76691915
47	3261.51713336
48	3260.32089414

49	3259.26541528
50	3258.34744806
51	3257.40575722
52	3256.4426332
53	3255.57616937
54	3254.58979819
55	3253.56265378
56	3252.36826796
57	3251.06628649
58	3249.80678307
59	3248.57941193
60	3247.43634434
61	3246.32730461
62	3245.2078261
63	3244.22825496
64	3243.28404653
65	3242.22707054
66	3241.13521344
67	3239.93160408
68	3238.62004028
69	3237.28240704
70	3235.92647302
71	3234.52085483
72	3233.20005118
73	3231.89420911
74	3230.81633658
75	3229.80314621
76	3228.70925146
77	3227.59782686
78	3226.46072314
79	3225.38817305
80	3224.3636674
81	3223.38892093
82	3222.5284968
83	3221.63050878
84	3220.67993006
85	3219.77012171
86	3218.81533141
87	3217.79355282
88	3216.62299197
89	3215.46620878
90	3214.49279535
91	3213.4924339
92	3212.46917948
93	3211.40397312
94	3210.3524907
95	3209.28388818
96	3208.25258432
97	3207.26251653

98	3206.30297143
99	3205.37549009
100	3204.59350932
101	3203.80442945
102	3203.0451728
103	3202.25031254
104	3201.49270221
105	3200.81545163
106	3200.12176743
107	3199.31593576
108	3198.46898733
109	3197.61467205
110	3196.68991049
111	3195.75838424
112	3194.86055892
113	3193.94900304
114	3192.99211729
115	3192.08826043
116	3191.22663291
117	3190.30946002
118	3189.46708317
119	3188.72712052
120	3188.01598725
121	3187.20110648
122	3186.40735792
123	3185.62078163
124	3184.82698809
125	3184.01073388
126	3183.20102776
127	3182.45091721
128	3181.57524889
129	3180.68021674
130	3179.74876761
131	3178.86189047
132	3178.1074881
133	3177.3274539
134	3176.49131957
135	3175.68902688
136	3174.95331594
137	3174.25102647
138	3173.5521054
139	3172.82284795
140	3172.11385592
141	3171.37773358
142	3170.57934866
143	3169.83269526
144	3169.18172732
145	3168.52213221
146	3167.91064094

147	3167.22692835
148	3166.40703475
149	3165.58521134
150	3164.70843471
151	3163.92435141
152	3163.19300348
153	3162.47192079
154	3161.83928221
155	3161.243654
156	3160.64148572
157	3160.21633699
158	3159.86376582
159	3159.52913493
160	3159.24317861
161	3158.83178846
162	3158.35804978
163	3157.91175895
164	3157.40433565
165	3156.86932602
166	3156.33381637
167	3155.97404657
168	3155.62494362
169	3155.37246129
170	3155.19932712
171	3155.03096929
172	3154.83522848
173	3154.53390332
174	3154.18893389
175	3153.83183962
176	3153.40276254
177	3152.87435414
178	3152.31510187
179	3151.72852708
180	3151.10411497
181	3150.42107894
182	3149.79862346
183	3149.18989797
184	3148.65977191
185	3148.24907891
186	3147.79067567
187	3147.31040033
188	3146.86283412
189	3146.56378567
190	3146.39510759
191	3146.22896275
192	3146.02762133
193	3145.84238365
194	3145.6153811
195	3145.38227964

196	3145.22641677
197	3145.1172334
198	3144.9217065
199	3144.73765168
200	3144.58522027
201	3144.46017571
202	3144.2623508
203	3144.04046316
204	3143.88649814
205	3143.71472507
206	3143.49451935
207	3143.34495222
208	3143.21195249
209	3143.06595497
210	3142.86768577
211	3142.58333632
212	3142.22109297
213	3141.83157384
214	3141.44090212
215	3141.0104233
216	3140.65442177
217	3140.2682764
218	3139.837984
219	3139.37174459
220	3139.06353911
221	3138.78474898
222	3138.47854924
223	3138.10137388
224	3137.70469671
225	3137.32400673
226	3136.88169703
227	3136.41553364
228	3135.93689569
229	3135.48084958
230	3135.06273169
231	3134.66999331
232	3134.29178732
233	3133.9323377
234	3133.5619928
235	3133.16872034
236	3132.78158592
237	3132.39549534
238	3132.07350417
239	3131.75044192
240	3131.43851118
241	3131.11586757
242	3130.86995683
243	3130.63571349
244	3130.45459785

```
245 3130.3275778
246 3130.23509093
247 3130.13331005
248 3130.03621585
249 3129.9068499
250 3129.81537448
251 3129.77058799
252 3129.74265487
253 3129.77570722
```

Listing 6: Output from script *makeMDS.py*

5

Question

5. 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.

Answer

Not attempted.

A Segaran's *generatefeedvector.py*

```
1 import feedparser
2 import re
3
4 # Returns title and dictionary of word counts for an RSS feed
5 def getwordcounts(url):
6     # Parse the feed
7     d=feedparser.parse(url)
8     wc={}
9
10    # Loop over all the entries
11    for e in d.entries:
12        if 'summary' in e: summary=e.summary
13        else: summary=e.description
14
15        # Extract a list of words
16        words=getwords(e.title+' '+summary)
17        for word in words:
18            wc.setdefault(word,0)
19            wc[word]+=1
20    return d.feed.title,wc
21
22 def getwords(html):
23     # Remove all the HTML tags
24     txt=re.compile(r'<[^>]+>').sub('',html)
25
26     # Split words by all non-alpha characters
27     words=re.compile(r'[^A-Za-z]+').split(txt)
28
29     # Convert to lowercase
30     return [word.lower() for word in words if word!='']
31
32
33 apcount={}
34 wordcounts={}
35 feedlist=[line for line in file('feedlist.txt')]
36 for feedurl in feedlist:
37     try:
38         title,wc=getwordcounts(feedurl)
39         wordcounts[title]=wc
40         for word,count in wc.items():
41             apcount.setdefault(word,0)
42             if count>1:
43                 apcount[word]+=1
44     except:
45         print 'Failed to parse feed %s' % feedurl
46
```

```

47 wordlist=[]
48 for w,bc in apcount.items():
49     frac=float(bc)/len(feedlist)
50     if frac>0.1 and frac<0.5:
51         wordlist.append(w)
52
53 out=file('blogdata1.txt','w')
54 out.write('Blog')
55 for word in wordlist: out.write('\t%s' % word)
56 out.write('\n')
57 for blog,wc in wordcounts.items():
58     print blog
59     blog = blog.encode('UTF-8')
60     out.write(blog)
61     for word in wordlist:
62         if word in wc: out.write('\t%d' % wc[word])
63         else: out.write('\t0')
64     out.write('\n')

```

Listing 7: Segaran’s *generatefeedvector.py*, with small modification added on line 59 to handle UTF-8 encodings

B Segaran's *clusters.py*

```
1 from PIL import Image, ImageDraw
2
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
17
18
19 from math import sqrt
20
21 def pearson(v1, v2):
22     # Simple sums
23     sum1=sum(v1)
24     sum2=sum(v2)
25
26     # Sums of the squares
27     sum1Sq=sum([pow(v,2) for v in v1])
28     sum2Sq=sum([pow(v,2) for v in v2])
29
30     # Sum of the products
31     pSum=sum([v1[i]*v2[i] for i in range(len(v1))])
32
33     # Calculate r (Pearson score)
34     num=pSum-(sum1*sum2/len(v1))
35     den=sqrt((sum1Sq-pow(sum1,2)/len(v1))*(sum2Sq-pow(sum2,2)/len(v1)))
36     if den==0: return 0
37
38     return 1.0-num/den
39
40 class bicluster:
41     def __init__(self, vec, left=None, right=None, distance=0.0, id=None):
42         self.left=left
43         self.right=right
44         self.vec=vec
```

```

45     self.id=id
46     self.distance=distance
47
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]
65                                                                    ].vec,clust[j].vec)
66
67                 d=distances[(clust[i].id,clust[j].id)]
68
69                 if d<closest:
70                     closest=d
71                     lowestpair=(i,j)
72
73     # calculate the average of the two clusters
74     mergevec=[
75         (clust[lowestpair[0]].vec[i]+clust[lowestpair[1]].vec[i])
76         /2.0
77         for i in range(len(clust[0].vec))]
78
79     # create the new cluster
80     newcluster=bicluster(mergevec,left=clust[lowestpair[0]],
81                           right=clust[lowestpair[1]],
82                           distance=closest,id=currentclustid)
83
84     # cluster ids that weren't in the original set are negative
85     currentclustid-=1
86     del clust[lowestpair[1]]
87     del clust[lowestpair[0]]
88     clust.append(newcluster)
89
90     return clust[0]
91
92 def printclust(clust,labels=None,n=0):
93     # 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
103         +1)
104     if clust.right!=None: printclust(clust.right ,labels=labels ,n=n
105         +1)
106
107 def getheight(clust):
108     # Is this an endpoint? Then the height is just 1
109     if clust.left==None and clust.right==None: return 1
110
111     # Otherwise the height is the same of the heights of
112     # each branch
113     return getheight(clust.left)+getheight(clust.right)
114
115 def getdepth(clust):
116     # The distance of an endpoint is 0.0
117     if clust.left==None and clust.right==None: return 0
118
119     # The distance of a branch is the greater of its two sides
120     # plus its own distance
121     return max(getdepth(clust.left),getdepth(clust.right))+clust.
122         distance
123
124 def drawdendrogram(clust , labels ,jpeg='clusters.jpg') :
125     # height and width
126     h=getheight(clust)*20
127     w=1200
128     depth=getdepth(clust)
129
130     # width is fixed , so scale distances accordingly
131     scaling=float(w-150)/depth
132
133     # Create a new image with a white background
134     img=Image.new( 'RGB' ,(w,h) ,(255,255,255))
135     draw=ImageDraw.Draw(img)
136
137     draw.line((0,h/2,10,h/2) , fill=(255,0,0))
138
139     # Draw the first node

```

```

138 drawnode(draw, clust, 10, (h/2), scaling, labels)
139 img.save(jpeg, 'JPEG')
140
141 def drawnode(draw, clust, x, y, scaling, labels):
142     if clust.id < 0:
143         h1=getheight(clust.left)*20
144         h2=getheight(clust.right)*20
145         top=y-(h1+h2)/2
146         bottom=y+(h1+h2)/2
147         # Line length
148         ll=clust.distance*scaling
149         # Vertical line from this cluster to children
150         draw.line((x, top+h1/2, x, bottom-h2/2), fill=(255,0,0))
151
152         # Horizontal line to left item
153         draw.line((x, top+h1/2, x+ll, top+h1/2), fill=(255,0,0))
154
155         # Horizontal line to right item
156         draw.line((x, bottom-h2/2, x+ll, bottom-h2/2), fill=(255,0,0))
157
158         # Call the function to draw the left and right nodes
159         drawnode(draw, clust.left, x+ll, top+h1/2, scaling, labels)
160         drawnode(draw, clust.right, x+ll, bottom-h2/2, scaling, labels)
161     else:
162         # If this is an endpoint, draw the item label
163         draw.text((x+5, y-7), labels[clust.id], (0,0,0))
164
165 def rotatematrix(data):
166     newdata=[]
167     for i in range(len(data[0])):
168         newrow=[data[j][i] for j in range(len(data))]
169         newdata.append(newrow)
170     return newdata
171
172 import random
173
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
177         rows]))
178     for i in range(len(rows[0]))]
179
180     # Create k randomly placed centroids
181     clusters=[[random.random()*(ranges[i][1]-ranges[i][0])+ranges[
182         i][0]
183         for i in range(len(rows[0]))] for j in range(k)]
184
185     lastmatches=None
186     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
213
214 def tanamoto(v1,v2):
215     c1,c2,shr=0,0,0
216
217     for i in range(len(v1)):
218         if v1[i]!=0: c1+=1 # in v1
219         if v2[i]!=0: c2+=1 # in v2
220         if v1[i]!=0 and v2[i]!=0: shr+=1 # in both
221
222     return 1.0-(float(shr)/(c1+c2-shr))
223
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
232     2D
233     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             errorterm=(fakedist[j][k]-realdist[j][k])/realdist[j][k]
252
253             # Each point needs to be moved away from or towards the
254             # point in proportion to how much error it has
255             grad[k][0]+=((loc[k][0]-loc[j][0])/fakedist[j][k])*
256                        errorterm
257             grad[k][1]+=((loc[k][1]-loc[j][1])/fakedist[j][k])*
258                        errorterm
259
260             # Keep track of the total error
261             totalerror+=abs(errorterm)
262         print totalerror
263
264     # If the answer got worse by moving the points, we are done
265     if lasterror and lasterror<totalerror: break
266     lasterror=totalerror
267
268     # Move each of the points by the learning rate times the
269     # gradient
270     for k in range(n):
271         loc[k][0]-=rate*grad[k][0]
272         loc[k][1]-=rate*grad[k][1]
273
274     return loc
275
276 def draw2d(data, labels, jpeg='mds2d.jpg'):
277     img=Image.new('RGB',(2000,2000),(255,255,255))
278     draw=ImageDraw.Draw(img)
279     for i in range(len(data)):
280         x=(data[i][0]+0.5)*1000

```

```
278     y=(data[i][1]+0.5)*1000
279     draw.text((x,y),labels[i],(0,0,0))
280     img.save(jpeg,'JPEG')
```

Listing 8: Segaran's *clusters.py*

References

- [1] SEGARAN, T. *Programming Collective Intelligence*, first ed. O'Reilly, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, August 2007.