CS 595: Assignment 9

Mallika Kogatam

Fall 2014

Contents

1 Problem 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).
```

1.1 Answer

1. The script shown in Listing 1 outputs a list of 100 random blogs in to a file .

2. The script show in Listing 2 outputs a list of atom feed for the respective blog. The atom feed grabbed will give only 25 entries by default.

3. The scripts were written based on the technique that is taught in the class.

4. To get the words from each feed I used *Segaran's generatefeedvector.py* from Program Collective Intelligence Text book.

5. I modified the *generatefeedvector.py* code to get the popular 500 words from all the atom feed.

6. I included a piece of code at line 31 in Listing 3to eliminate the words which falls under the stop words, the stop word list used is the "MySql full-text stopwords".

7. The code does not handle the entries if the entries for the blog is beyond 500, as the maximum limit of the entries for an atom feed is limited to only 500 entries at a time.

8. So I decided to visit each page and grab all the entries for a blog. This is implemented in lines 50 to 56 in Listing 3

9. To get the most popular words among all the blogs, firstly I am keeping track of the occurrence of each word in all the blogs.

10. So that I can pick only the top 500 words by sorting the words based on the occurrence value.

11. This is implemented at lines 106 -108 and 121-124 in Listing 3. And then I am limiting down the word to top 500 words only.

12. The script shown in Listing 3 outputs the matrix in expected format which is stored in *blogdata-500.txt* which is used for each subsequent question in this assignment, this file is uploaded in the github.

13. To get the number of pages in each blog, I wrote a small python code as shown in Listing 4 which outputs the file *noOfPages.txt*

14. The R script shown in the Listing 5 generates a Histogram shown in Figure 1 for Blogs Vs no of Pages.

15. From the Histogram in Figure 1, it's pretty clear that 60 blogs out of 100 have pages between 1 and 10, and only couple of blogs have pages between 280 and 290.

```
1  #! /bin/bash
2
3  echo ""
4
5  if [ $# -ne 1 ]
6  then
7      echo "Usage: $0 #"
8      exit
9  fi
10
11 num=$1
12
13 echo "http://f-measure.blogspot.com/" > blogList-$num.txt
14 echo "http://ws-dl.blogspot.com/" >> blogList-$num.txt
15
16 for ((i=0;i< $num;i++))
17 do
18     curl -I -L 'http://www.blogger.com/next-blog?navBar=true&blogID=3471633091411211117' |
           grep Location | tail -n 1 | cut -d" " -f2 | cut -d"?" -f 1 >> blogList-$num.txt
19 done
```

Listing 1: Shell Program for getting 100 blogs

```
1  #! /bin/bash
2
3  if [ $# -ne 1 ]
4  then
5      echo "Usage: $0 #"
6      exit
7  fi
8
9  blogFile=$1
10
11 echo -n "" > `basename $blogFile .txt`-atom.txt
12
13 for line in `cat $blogFile`
14 do
15     #curl $line | grep 'rel="alternate"' | grep atom | sed 's/.*href=//' | sed 's/\/>//' |
           sed 's/"//g' >> `basename $blogFile .txt`-atom.txt
16
17
18     test=`curl $line | grep 'rel="alternate"' | grep atom | sed 's/.*href=//' | sed 's
           /\/>//' | sed 's/"//g' | sed 's/ //g'`
19     echo $test >> `basename $blogFile .txt`-atom.txt
20 done
```

Listing 2: Shell Program for getting atom feed for the blogs

```
1   import feedparser
2   import collections
3   import re
4   import operator
5
6
7   def getwords(html):
8       text  = re.compile(r'<[^>]+>').sub('', html)
9       words = re.compile(r'[^A-z^a-z]+').split(text)
10      return [word.lower() for word in words if word]
11
12  def checkNextPage(url):
13      f = urllib.urlopen(url)
14      soup = BeautifulSoup(f.read(), from_encoding=f.info().getparam('charset'))
15
16      try:
17          link = soup.find('link', rel='next',href = True)['href']
18      except TypeError:
19          link = None
20      return link
21
22  def getwordcounts(url):
23
24      fd = feedparser.parse(url)
25      wc        = collections.defaultdict(int)
26      stopwords = []
27
28      stopWordList = open('stopWordList.txt').readlines()
29      pages = len(fd['entries'])
30
31      for stopWord in stopWordList:
32          stopWord = stopWord.strip()
33          stopwords.append(stopWord)
34
35
36
37      for e in fd.entries:
38          if 'summary' in e:
39              summary = e.summary
40          else:
41              summary = e.description
42
43          words = getwords('%s %s' % (e.title, summary))
44
45          for word in words:
46              if word not in stopwords:
47                  wc[word] += 1
48
49
50      nextLink = checkNextPage( url )
51          while nextLink:
52              nextLink = checkNextPage( nextLink )
53              d        = feedparser.parse(nextlink)
54              pages    = len(d['entries'])
55              for e in d.entries:
56                  if 'summary' in e:
57                      summary = e.summary
58                  else:
59                      summary = e.description
60
61                  words = getwords('%s %s' % (e.title, summary))
62
63                  for word in words:
64                      if word not in stopwords:
65                          #print word
66                          wc[word] += 1
67
68      if 'title' not in fd.feed:
69          print 'Invalid url', url
70          return 'bogus data', wc
71
```

```
72        return fd.feed.title, wc
73
74   def main():
75
76        # XXX: break this up into smaller functions, write tests for them
77        apcount      = collections.defaultdict(int)
78        wordcounts = {}
79        feedlist    = open('blogList-120-atom.txt').readlines()
80        totalWordCount = {}
81
82        for url in feedlist:
83            title, wc = getwordcounts(url)
84            wordcounts[title] = wc
85
86            for word, count in wc.iteritems():
87                if count > 1:
88                    apcount[word] += 1
89
90                    try:
91                        totalWordCount[word] += count
92                    except KeyError:
93                        totalWordCount[word] = count
94
95
96        wordlist = []
97        for w, bc in apcount.iteritems():
98            frac = float(bc)/len(feedlist)
99            #print frac
100           if frac > 0.1 and frac < 0.5:
101               wordlist.append(w)
102
103       countOfWords = []
104
105       for word in wordlist:
106           countOfWords.append((word,totalWordCount[word]))
107
108       countOfWords.sort(key=lambda rating: rating[1], reverse = True )
109
110       for words in countOfWords[0:500]:
111           print words[0] , words[1]
112       out = file('blogdata-120-1500pages.txt', 'w')
113       out.write('Blog')
114
115       for w in countOfWords[0:500]:
116           #print w
117           out.write('\t' + w[0])
118       out.write('\n')
119
120       for blogname, counts in wordcounts.iteritems():
121           blogname = blogname.encode('UTF-8')
122           out.write(blogname)
123
124           for w in countOfWords[0:500]:
125               word = w[0]
126               if w in counts:
127                   out.write('\t%d' % counts[word])
128               else:
129                   out.write('\t0')
130           out.write('\n')
131
132       out.close()
133
134   if __name__ == '__main__':
135       main()
```

Listing 3: Python code for grabbing popular 500 words from 100 atom feeds

5

```
 1  #! /usr/bin/env python
 2  import os
 3  import sys
 4  import urllib
 5  import time
 6  import feedparser
 7
 8  from bs4 import BeautifulSoup
 9
10  def checkNextPage(url):
11
12      f = urllib.urlopen(url)
13      soup = BeautifulSoup(f.read(), from_encoding=f.info().getparam('charset'))
14
15      try:
16          link = soup.find('link', rel='next',href = True)['href']
17      except TypeError:
18          link = None
19      return link
20
21
22  def main():
23      feedlist   = open('blogList-120-atom.txt').readlines()
24
25      for url in feedlist:
26          try:
27              d = feedparser.parse(url)
28
29              title = d['feed']['title']
30
31              count    = 1
32              nextLink = checkNextPage( url )
33
34              while nextLink:
35                  nextLink = checkNextPage( nextLink )
36                  count += 1
37
38              print u'|'.join((str(count),title)).encode('utf-8').strip()
39
40
41
42          except KeyError:
43              pass
44
45  if __name__ == '__main__':
46      main()
```

Listing 4: Python code for grabbing number of pages for each blog

```
1  data <- read.csv("noOfPages.txt", stringsAsFactors = F, header = FALSE, sep = "|")
2
3  incdata = data[, 1]
4
5  incdata <- as.numeric(incdata)
6
7  brk <- seq(0, 300, 10)
8
9  png("q1-histogram1.png")
10 hist(incdata, main = "Blogs vs. Number of Pages", breaks=brk, freq = T, xlab="Pages", ylab="
       Blogs")
```

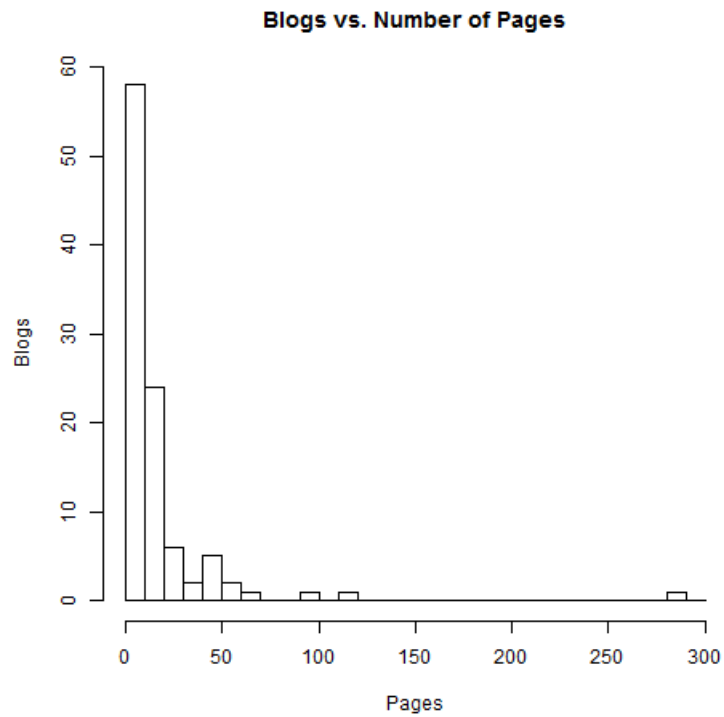Listing 5: R Script for generating a Histogram

Figure 1: Histogram showing Blogs Vs No of Pages

2 Problem 2

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).
```

2.1 Answer

1. The script show in Listing 6 uses the *Toby Segaran's clusters.py* code in Listing 11 and produces the Dendrogram shown in Figure 2

2. The printclust function on line 8 in Listing 6 prints the dendrogram. The drawdendrogram function on line 11 saves a JPEG of the dendrogram.

3. The ascii file of the dendrogram is uploaded to github, and the file name is *ascii-dendrogram.txt*.

4. Unfortunately, it is difficult to see, but this dendogram shows that the blogs calculated to be most like *F-Measure* are split into two clusters, the blogs in the first cluster are *YOUNGEST INDIE* and *Music Liberation*, the blogs in second cluster are *The Devils Music* and *McCrak's Juke*.

5. The blog calculated to be most like *Web Science and Digital Libraries Research Group* are *Koranteng's Toli* and *words of advance for young people.*

```
1   import clusters
2
3
4   blognames,words,data=clusters.readfile('blogdata120-atom-500.txt')
5   clust = clusters.hcluster(data)
6
7   # print ASCII dendrogram
8   clusters.printclust(clust, labels=blognames)
9
10  # save JPEG dendrogram
11  clusters.drawdendrogram(clust, blognames, jpeg='blogclust.jpg')
```

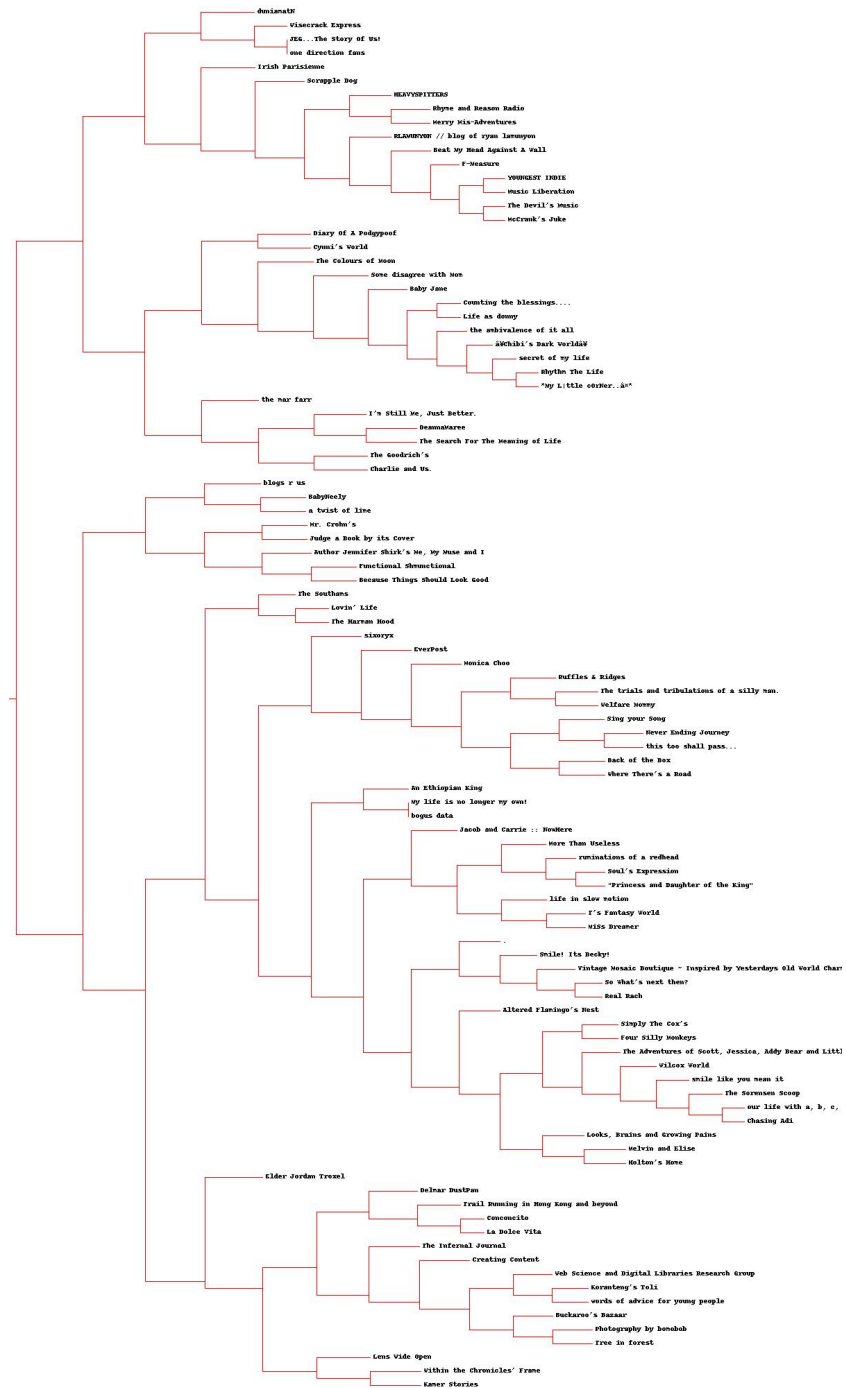Listing 6: Python code for grabbing number of pages for each blog

duniana3N
Wisecrack Express
JEG...The Story Of Us!
one direction fans
Irish Parisienne
Scrapple Dog
HEAVYSPITTERS
Rhyme and Reason Radio
Merry Mis-Adventures
BLAVAHVON // blog of ryan lavonyon
Beat My Head Against A Wall
F-Measure
YOUNGEST INDIE
Music Liberation
The Devil's Music
McCrunk's Juke
Diary Of A Podgypoof
Cynni's World
The Colours of Moon
Some disagree with Mom
Baby Jane
Counting the blessings....
Life as dommy
the ambivalence of it all
â¥Chibi's Dark Worldâ¥
secret of my life
Rhythm The Life
^My Little cøriNer..å°^
the mar farr
I'm Still Me, Just Better.
DeannaWaree
The Search For The Meaning of Life
The Goodrich's
Charlie and Us.
blogs r us
BabyNeely
a twist of line
Mr. Crohn's
Judge a Book by its Cover
Author Jennifer Shirk's Me, My Muse and I
functional Shrunctional
Because Things Should Look Good
The Southans
Lovin' Life
The Harman Hood
sixoryx
EverPost
Monica Choo
Ruffles & Ridges
The trials and tribulations of a silly man.
Welfare Mommy
Sing your Song
Never Ending Journey
this too shall pass...
Back of the Box
Where There's a Road
An Ethiopian King
My life is no longer my own!
bogus data
Jacob and Carrie :: NowHere
More Than Useless
ruminations of a redhead
Soul's Expression
"Princess and Daughter of the King"
life in slow motion
I's Fantasy World
Miss Dreamer
.
Smile! Its Becky!
Vintage Mosaic Boutique - Inspired by Yesterdays Old World Char
So what's next then?
Real Rach
Altered Flamingo's Nest
Simply The Cox's
Four Silly Monkeys
The Adventures of Scott, Jessica, Addy Bear and Littl
Wilcox World
smile like you mean it
The Sorensen Scoop
our life with a, b, c,
Chasing Adi
Looks, Brains and Growing Pains
Melvin and Elise
Holton's Home
Elder Jordan Troxel
Delmar DustPan
Trail Running in Hong Kong and beyond
Concoucito
La Dolce Vita
The Infernal Journal
Creating Content
Web Science and Digital Libraries Research Group
Kormteng's Toli
words of advice for young people
Buckaroo's Bazaar
Photography by bonobob
free in forest
Lens Wide Open
Within the Chronicles' Frame
Kamer Stories

Figure 2: Dendrogram Produced by Program in Listing 6

3   Problem 3

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?

3.1   Answer

The blog clustering is performed by the script shown in Listing 7, which makes use of Toby Segaran's *clusters.py* using the function *kcluster* on lines 8, 12, and 16.

```python
#!/usr/local/bin/python

import clusters

blognames,words,data=clusters.readfile('blogdata120-atom-500.txt')

print "For k=5"
kclust=clusters.kcluster(data, k=5)
print

print "For k=10"
kclust=clusters.kcluster(data, k=10)
print

print "For k=20"
kclust=clusters.kcluster(data, k=20)
print
```

Listing 7: Python script for clustering the blogs using K-means, using k=5, 10, and 20

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
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
Iteration 10
Iteration 11

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

For k=20
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
```

Thus, for $k = 5$ we get 12 iterations, for $k = 10$ we get 7 iterations, and for $k = 20$ we get 6 iterations.

4 Problem 4

Question

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

4.1 Answer

1. The blog space is generated using multidimensional scaling from the script `makeMDS.py` shown in Listing 8, which makes use of Toby Segaran's *clusters.py* using the functions *scaledown* on line 7 and *draw2d* on line 9.

```python
#!/usr/local/bin/python

import clusters

blognames,words,data=clusters.readfile('blogdata120-atom-500.txt')

coords = clusters.scaledown(data)

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

Listing 8: Python script for generating a MDS from the blog data

2. unfortunately the blog space produced does not fit well on a letter-sized page shown in Figure 3.

3. Listing 9 shows the output from running this script, which took 253 iterations on this run.

```
 1   4532.41130905
 2   3366.98206519
 3   3326.3306574
 4   3308.09464169
 5   3297.94258112
 6   3291.51602489
 7   3287.28793801
 8   3283.77516396
 9   3280.99776391
10   3278.44993199
11   3275.81243841
12   3273.31397104
13   3270.8684595
14   3268.42055835
15   3266.16460163
16   3264.23476083
17   3262.42227912
18   3260.51325928
19   3258.82854152
20   3257.18175939
21   3255.37923243
22   3253.30711541
23   3251.00612794
24   3248.77308262
25   3246.70118667
26   3244.47115104
27   3241.94493708
28   3239.40739168
29   3236.99558481
30   3234.81656727
31   3232.88203014
32   3231.04567171
33   3229.23063872
34   3227.47967223
35   3225.91794645
36   3224.58649843
37   3223.20000436
38   3222.0867435
39   3221.01271989
40   3219.95473094
41   3218.75956428
42   3217.59979619
43   3216.4522233
44   3215.34629148
45   3214.36905778
46   3213.26833436
47   3212.24470346
48   3211.31994368
49   3210.29982355
50   3209.20246315
51   3207.95132503
52   3206.82241072
53   3205.91909053
54   3204.92220274
55   3204.31947561
56   3203.74640843
57   3203.27606811
58   3202.67555472
59   3201.89979232
60   3200.88578203
61   3200.01531056
62   3199.2966446
63   3198.45478403
64   3197.6201472
65   3196.73438036
66   3196.02119032
67   3195.41196943
68   3194.72376496
69   3194.0329727
70   3193.41355901
71   3192.78060731
```

72	3191.90770762
73	3190.80437498
74	3189.55398117
75	3188.23142174
76	3186.89288539
77	3185.71338432
78	3184.5789406
79	3183.5206291
80	3182.51360921
81	3181.42690675
82	3180.41702115
83	3179.54898356
84	3178.76281891
85	3177.99771823
86	3177.24097774
87	3176.58456935
88	3176.0328065
89	3175.38333701
90	3174.71747303
91	3174.01886829
92	3173.32737481
93	3172.70330603
94	3172.1138461
95	3171.52419342
96	3170.90758333
97	3170.4179592
98	3170.02282149
99	3169.5821675
100	3169.07816383
101	3168.448593
102	3167.75862833
103	3167.05280662
104	3166.42332043
105	3165.9904937
106	3165.73529802
107	3165.54265976
108	3165.33739243
109	3165.07807778
110	3164.91141097
111	3164.62059016
112	3164.17905363
113	3163.65023788
114	3163.14226209
115	3162.70328486
116	3162.36024226
117	3161.99817832
118	3161.70317464
119	3161.37377857
120	3160.9434041
121	3160.44914717
122	3159.86024781
123	3159.29974867
124	3158.58691893
125	3157.78851118
126	3156.84328215
127	3155.8879416
128	3155.00788822
129	3154.23167922
130	3153.44215741
131	3152.76590863
132	3152.13154349
133	3151.49922048
134	3150.91731722
135	3150.33609096
136	3149.74067727
137	3149.15263975
138	3148.52983426
139	3147.83979099
140	3147.25183635
141	3146.72688685
142	3146.10697938
143	3145.55140026

144	3145.038229
145	3144.50839712
146	3143.94311783
147	3143.40783945
148	3142.95445563
149	3142.52189469
150	3141.99941939
151	3141.35987167
152	3140.6610068
153	3140.01716194
154	3139.47786778
155	3138.91063005
156	3138.29379482
157	3137.61144599
158	3136.89591297
159	3136.14533364
160	3135.38865056
161	3134.64653422
162	3133.91168296
163	3133.23112402
164	3132.52038824
165	3131.88007921
166	3131.30351041
167	3130.66122515
168	3129.90421203
169	3129.1405021
170	3128.32782048
171	3127.47536739
172	3126.54560283
173	3125.62757639
174	3124.67176012
175	3123.72838126
176	3122.76127432
177	3121.76341232
178	3120.77006527
179	3119.75759981
180	3118.76498364
181	3117.77794841
182	3116.78570912
183	3115.80880423
184	3114.85906684
185	3113.95665663
186	3113.0751775
187	3112.2783084
188	3111.51872253
189	3110.81819246
190	3110.12121549
191	3109.49131097
192	3108.87630713
193	3108.24307894
194	3107.61218582
195	3106.9346598
196	3106.17407566
197	3105.30253203
198	3104.46828824
199	3103.70960692
200	3103.04853569
201	3102.374827
202	3101.76925634
203	3101.10225207
204	3100.39181475
205	3099.80313972
206	3099.32543062
207	3098.83756084
208	3098.39524227
209	3097.89054287
210	3097.45857954
211	3097.16391173
212	3096.85125512
213	3096.55032349
214	3096.25660576
215	3096.00089054

```
216 │ 3095.71194597
217 │ 3095.44859148
218 │ 3095.20946085
219 │ 3095.07182068
220 │ 3095.06452066
221 │ 3095.04654607
222 │ 3094.9754195
223 │ 3094.85516015
224 │ 3094.79234814
225 │ 3094.67410977
226 │ 3094.52095848
227 │ 3094.33078357
228 │ 3094.08903464
229 │ 3093.79059454
230 │ 3093.51468176
231 │ 3093.20791956
232 │ 3092.91243999
233 │ 3092.65113648
234 │ 3092.34912442
235 │ 3092.05447449
236 │ 3091.80452138
237 │ 3091.55424259
238 │ 3091.27376913
239 │ 3090.97576404
240 │ 3090.65628056
241 │ 3090.30346956
242 │ 3089.97015589
243 │ 3089.57549888
244 │ 3089.19007073
245 │ 3088.82871046
246 │ 3088.44864717
247 │ 3088.11602282
248 │ 3087.91481431
249 │ 3087.80681744
250 │ 3087.74255397
251 │ 3087.66780485
252 │ 3087.65822646
253 │ 3087.66845007
```

Listing 9: Output from script *makeMDS.py*

Figure 3: Blog space produced by the *makeMDS.py* script

5   Problem 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.

5.1   Answer

1. I used the python which is used to get the matrix for the questions 1 but added few lines of code to calculate the TFIDF value for each word

2. IDF value is calculated at line 149 in Listing 10

3. The TF value for each word with respective to each blog is calculated at line 166 which is used to calculate TFIDF value at line 167 in Listing 10

4. The matrix with blog name and the words along with their TFIDF value is calculated by The program Listed in Listing 10. The output is uploaded in the github as *bloglist-500-matrix.txt*.

5. I used the same program which I used in Problem 2 to generate the dendrogram shown in Figure 4

6. Unfortunately, it is difficult to see, but this dendogram shows that the blogs calculated to be most like *F-Measure* are split into two clusters, the blogs in the first cluster is *Music Liberation* contrast to the out from the dendrogram from the second question the blog */YOUNGEST INDIE*

7. The blogs in second cluster are *YOUNGEST INDIE* and *McCrak's Juke* where as the the blog *The Devil's Music* is in the second cluster instead of *YOUNGEST INDIE* in the dendrogram produced for second question .

8. Unlike in the dendrogram produced for the second question The blog calculated to be most like *Web Science and Digital Libraries Research Group* is *Buckaroo's Bazar*

9. From my observation of both the dendrograms I found that the hierarchy of the clusters are changing but somehow that clusters are around the same blogs with different hierarchy.

10. The accuracy of matching the most alike blogs is changing.

5.2   Output for 5th problem

```python
import feedparser
import collections
import re
import operator
import math

INPUT_FILE  = "blogList-120-atom.txt"
OUTPUT_FILE = "blogList-120-500-matrix.txt"

def getwords(html):
    text  = re.compile(r'<[^>]+>').sub('', html)
    words = re.compile(r'[^A-z^a-z]+').split(text)

    return [word.lower() for word in words if word]

def getwordcounts(url):

    fd = feedparser.parse(url)
    wc         = collections.defaultdict(int)
    stopwords = []

    stopWordList = open('stopWordList.txt').readlines()
    pages = len(fd['entries'])

    for stopWord in stopWordList:
        stopWord = stopWord.strip()
        stopwords.append(stopWord)

    for e in fd.entries:
        if 'summary' in e:
            summary = e.summary
        else:
            summary = e.description
        words = getwords('%s %s' % (e.title, summary))

        for word in words:
            if word not in stopwords:
                wc[word] += 1

    if pages == 500:
        next_link = url + "?start-index=501"
        d         = feedparser.parse(next_link)
        pages     = len(d['entries'])
        for e in d.entries:
            if 'summary' in e:
                summary = e.summary
            else:
                summary = e.description

            words = getwords('%s %s' % (e.title, summary))

            for word in words:
                if word not in stopwords:
                    #print word

                    wc[word] += 1

        if pages == 500:
            next_link = url + "?start-index=1001"
            for e in d.entries:
                if 'summary' in e:
                    summary = e.summary
                else:
                    summary = e.description

                words = getwords('%s %s' % (e.title, summary))

                for word in words:
                    if word not in stopwords:
                        #print word
```

```python
72                              wc[word] += 1
73                    if pages == 500:
74                        next_link = url + "?start-index=1501"
75                        for e in d.entries:
76                            if 'summary' in e:
77                                summary = e.summary
78                            else:
79                                summary = e.description
80
81                            words = getwords('%s %s' % (e.title, summary))
82
83                            for word in words:
84                                if word not in stopwords:
85                                    wc[word] += 1
86
87        if 'title' not in fd.feed:
88            print 'Invalid url', url
89            return 'bogus data', wc
90
91        return fd.feed.title, wc
92
93  def main():
94
95        # XXX: break this up into smaller functions, write tests for them
96
97        apcount    = collections.defaultdict(int)
98        wordcounts = {}
99        feedlist   = open( INPUT_FILE ).readlines()
100       totalWordCount = {}
101
102       for url in feedlist:
103           title, wc = getwordcounts(url)
104           wordcounts[title] = wc
105
106           for word, count in wc.iteritems():
107               if count > 1:
108                   apcount[word] += 1
109
110                   try:
111                       totalWordCount[word] += count
112                   except KeyError:
113                       totalWordCount[word] = count
114
115       wordlist = []
116
117
118       for w, bc in apcount.iteritems():
119           frac = float(bc)/len(feedlist)
120           #print frac
121           if frac > 0.1 and frac < 0.5:
122               wordlist.append(w)
123
124       countOfWords = []
125
126       for word in wordlist:
127           countOfWords.append((word, totalWordCount[word]))
128
129       countOfWords.sort(key=lambda rating: rating[1], reverse = True )
130
131       countOfWords = countOfWords[0:500]
132
133       out = file(OUTPUT_FILE, 'w')
134       out.write('Blog')
135
136       idfWordCount = {}
137
138       for w in countOfWords:
139           word = w[0]
140           noOfBlogs = 0
141           for blogname, counts in wordcounts.iteritems():
142
143               if word in counts:
```

```
144                    noOfBlogs += 1
145                #print noOfBlogs
146            idf = math.log( 100.0 / noOfBlogs   , 2 )
147
148
149            idfWordCount[word]   = idf
150
151
152        for w in countOfWords:
153            #print w
154            out.write('\t' + w[0])
155
156        out.write('\n')
157
158        for blogname, counts in wordcounts.iteritems():
159            blogname = blogname.encode('UTF-8')
160            out.write(blogname)
161
162            for w in countOfWords:
163                word = w[0]
164                occurance = w[1]
165
166                tf = float(counts[word]) / occurance
167                tfidf = tf * idfWordCount[word]
168
169                # d is integer, f is float
170                out.write('\t%f' % tfidf)
171
172            out.write('\n')
173
174        out.close()
175
176  if __name__ == '__main__':
177      main()
```

Listing 10: Python code for grabbing popular 500 words from 100 atom feeds and their TFIDF values

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-
from PIL import Image, ImageDraw
from math import sqrt
import random


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)


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)
```

```python
72
73          # loop through every pair looking for the smallest distance
74              for i in range(len(clust)):
75                  for j in range(i + 1, len(clust)):
76              # distances is the cache of distance calculations
77                      if (clust[i].id, clust[j].id) not in distances:
78                          distances[(clust[i].id, clust[j].id)] = \
79                              distance(clust[i].vec, clust[j].vec)
80
81                      d = distances[(clust[i].id, clust[j].id)]
82
83                      if d < closest:
84                          closest = d
85                          lowestpair = (i, j)
86
87          # calculate the average of the two clusters
88              mergevec = [(clust[lowestpair[0]].vec[i] + clust[lowestpair[1]].vec[i])
89                          / 2.0 for i in range(len(clust[0].vec))]
90
91          # create the new cluster
92              newcluster = bicluster(mergevec, left=clust[lowestpair[0]],
93                                     right=clust[lowestpair[1]], distance=closest,
94                                     id=currentclustid)
95
96          # cluster ids that weren't in the original set are negative
97              currentclustid -= 1
98              del clust[lowestpair[1]]
99              del clust[lowestpair[0]]
100             clust.append(newcluster)
101
102         return clust[0]
103
104
105 def printclust(clust, labels=None, n=0):
106     # indent to make a hierarchy layout
107         for i in range(n):
108             print ' ',
109         if clust.id < 0:
110         # negative id means that this is branch
111             print '-'
112         else:
113         # positive id means that this is an endpoint
114             if labels == None:
115                 print clust.id
116             else:
117                 print labels[clust.id]
118
119     # now print the right and left branches
120         if clust.left != None:
121             printclust(clust.left, labels=labels, n=n + 1)
122         if clust.right != None:
123             printclust(clust.right, labels=labels, n=n + 1)
124
125
126 def getheight(clust):
127     # Is this an endpoint? Then the height is just 1
128         if clust.left == None and clust.right == None:
129             return 1
130
131     # Otherwise the height is the same of the heights of
132     # each branch
133         return getheight(clust.left) + getheight(clust.right)
134
135
136 def getdepth(clust):
137     # The distance of an endpoint is 0.0
138         if clust.left == None and clust.right == None:
139             return 0
140
141     # The distance of a branch is the greater of its two sides
142     # plus its own distance
143         return max(getdepth(clust.left), getdepth(clust.right)) + clust.distance
```

```
144
145
146   def drawdendrogram(clust, labels, jpeg='clusters.jpg'):
147     # height and width
148       h = getheight(clust) * 20
149       w = 1200
150       depth = getdepth(clust)
151
152     # width is fixed, so scale distances accordingly
153       scaling = float(w - 150) / depth
154
155     # Create a new image with a white background
156       img = Image.new('RGB', (w, h), (255, 255, 255))
157       draw = ImageDraw.Draw(img)
158
159       draw.line((0, h / 2, 10, h / 2), fill=(255, 0, 0))
160
161     # Draw the first node
162       drawnode(
163           draw,
164           clust,
165           10,
166           h / 2,
167           scaling,
168           labels,
169           )
170       img.save(jpeg, 'JPEG')
171
172
173   def drawnode(
174       draw,
175       clust,
176       x,
177       y,
178       scaling,
179       labels,
180       ):
181       if clust.id < 0:
182           h1 = getheight(clust.left) * 20
183           h2 = getheight(clust.right) * 20
184           top = y - (h1 + h2) / 2
185           bottom = y + (h1 + h2) / 2
186       # Line length
187           ll = clust.distance * scaling
188       # Vertical line from this cluster to children
189           draw.line((x, top + h1 / 2, x, bottom - h2 / 2), fill=(255, 0, 0))
190
191       # Horizontal line to left item
192           draw.line((x, top + h1 / 2, x + ll, top + h1 / 2), fill=(255, 0, 0))
193
194       # Horizontal line to right item
195           draw.line((x, bottom - h2 / 2, x + ll, bottom - h2 / 2), fill=(255, 0,
196                   0))
197
198       # Call the function to draw the left and right nodes
199           drawnode(
200               draw,
201               clust.left,
202               x + ll,
203               top + h1 / 2,
204               scaling,
205               labels,
206               )
207           drawnode(
208               draw,
209               clust.right,
210               x + ll,
211               bottom - h2 / 2,
212               scaling,
213               labels,
214               )
215       else:
```

```
216        # If this is an endpoint, draw the item label
217            draw.text((x + 5, y - 7), labels[clust.id], (0, 0, 0))
218
219
220  def rotatematrix(data):
221        newdata = []
222        for i in range(len(data[0])):
223            newrow = [data[j][i] for j in range(len(data))]
224            newdata.append(newrow)
225        return newdata
226
227
228  def kcluster(rows, distance=pearson, k=4):
229      # Determine the minimum and maximum values for each point
230        ranges = [(min([row[i] for row in rows]), max([row[i] for row in rows]))
231                  for i in range(len(rows[0]))]
232
233    # Create k randomly placed centroids
234        clusters = [[random.random() * (ranges[i][1] - ranges[i][0]) + ranges[i][0]
235                    for i in range(len(rows[0]))] for j in range(k)]
236
237        lastmatches = None
238        for t in range(100):
239            print 'Iteration %d' % t
240            bestmatches = [[] for i in range(k)]
241
242      # Find which centroid is the closest for each row
243            for j in range(len(rows)):
244                row = rows[j]
245                bestmatch = 0
246                for i in range(k):
247                    d = distance(clusters[i], row)
248                    if d < distance(clusters[bestmatch], row):
249                        bestmatch = i
250                bestmatches[bestmatch].append(j)
251
252      # If the results are the same as last time, this is complete
253            if bestmatches == lastmatches:
254                break
255            lastmatches = bestmatches
256
257      # Move the centroids to the average of their members
258            for i in range(k):
259                avgs = [0.0] * len(rows[0])
260                if len(bestmatches[i]) > 0:
261                    for rowid in bestmatches[i]:
262                        for m in range(len(rows[rowid])):
263                            avgs[m] += rows[rowid][m]
264                    for j in range(len(avgs)):
265                        avgs[j] /= len(bestmatches[i])
266                    clusters[i] = avgs
267
268        return bestmatches
269
270
271  def tanamoto(v1, v2):
272        (c1, c2, shr) = (0, 0, 0)
273
274        for i in range(len(v1)):
275            if v1[i] != 0:  # in v1
276                c1 += 1
277            if v2[i] != 0:  # in v2
278                c2 += 1
279            if v1[i] != 0 and v2[i] != 0:  # in both
280                shr += 1
281
282        return 1.0 - float(shr) / (c1 + c2 - shr)
283
284
285  def scaledown(data, distance=pearson, rate=0.01):
286        n = len(data)
287
```

```
288    # The real distances between every pair of items
289        realdist = [[distance(data[i], data[j]) for j in range(n)] for i in
290                    range(0, n)]
291
292    # Randomly initialize the starting points of the locations in 2D
293        loc = [[random.random(), random.random()] for i in range(n)]
294        fakedist = [[0.0 for j in range(n)] for i in range(n)]
295
296        lasterror = None
297        for m in range(0, 1000):
298        # Find projected distances
299            for i in range(n):
300                for j in range(n):
301                    fakedist[i][j] = sqrt(sum([pow(loc[i][x] - loc[j][x], 2)
302                                          for x in range(len(loc[i]))]))
303
304        # Move points
305            grad = [[0.0, 0.0] for i in range(n)]
306
307            totalerror = 0
308            for k in range(n):
309                for j in range(n):
310                    if j == k:
311                        continue
312        # The error is percent difference between the distances
313                    errorterm = (fakedist[j][k] - realdist[j][k]) / realdist[j][k]
314
315            # Each point needs to be moved away from or towards the other
316            # point in proportion to how much error it has
317                    grad[k][0] += (loc[k][0] - loc[j][0]) / fakedist[j][k] \
318                        * errorterm
319                    grad[k][1] += (loc[k][1] - loc[j][1]) / fakedist[j][k] \
320                        * errorterm
321
322            # Keep track of the total error
323                    totalerror += abs(errorterm)
324            print totalerror
325
326        # If the answer got worse by moving the points, we are done
327            if lasterror and lasterror < totalerror:
328                break
329            lasterror = totalerror
330
331        # Move each of the points by the learning rate times the gradient
332            for k in range(n):
333                loc[k][0] -= rate * grad[k][0]
334                loc[k][1] -= rate * grad[k][1]
335
336        return loc
337
338
339 def draw2d(data, labels, jpeg='mds2d.jpg'):
340     img = Image.new('RGB', (2000, 2000), (255, 255, 255))
341     draw = ImageDraw.Draw(img)
342     for i in range(len(data)):
343         x = (data[i][0] + 0.5) * 1000
344         y = (data[i][1] + 0.5) * 1000
345         draw.text((x, y), labels[i], (0, 0, 0))
346     img.save(jpeg, 'JPEG')
```
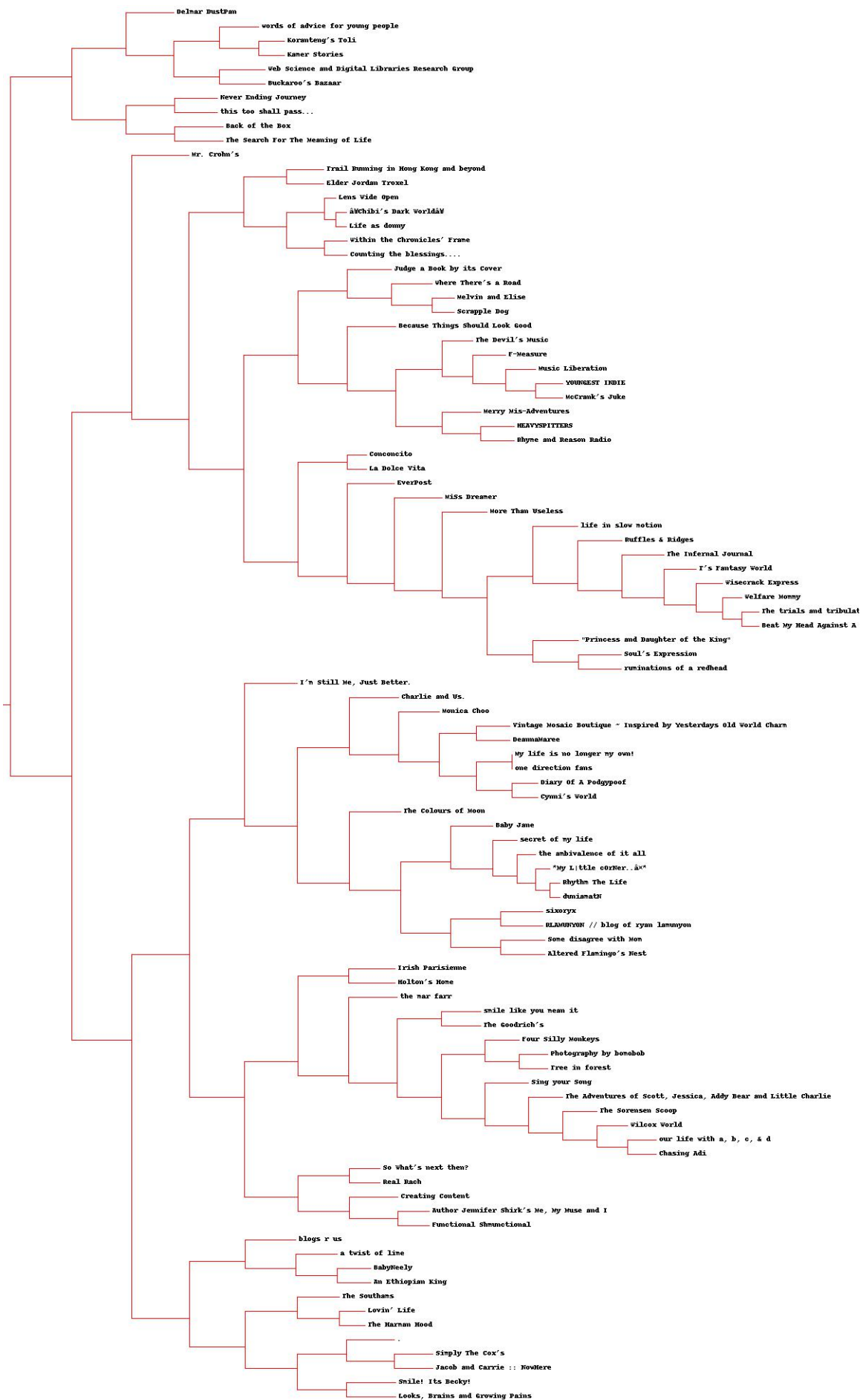
Listing 11: Segaran's *clusters.py*

Figure 4: Dendrogram produced based on word's TFIDF values script

Bibliography

[1] Feedparser documentation. http://pythonhosted.org//feedparser/.

[2] Online xml viewer. http://xmlgrid.net/.

[3] Tfidf. http://en.wikipedia.org/wiki/Tf

[4] Unicode documentation. https://docs.python.org/2/howto/unicode.html.

[5] Using feedparser in python. http://www.pythonforbeginners.com/feedparser/using-feedparser-in-python.

[6] Toby Segaran. *Programming Collective Intelligence.* O'Reilly, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition, August 2007.