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. After little research I found out that we can set the max limit by adding *max-result=1000*. So I appended the maximum limit to each atom feed extracted from Listing 2.
4. The scripts were written based on the technique that is taught in the class.
5. To get the words from each feed I used *Segaran's generatefeedvector.py* from Program Collective Intelligence Text book.
6. I modified the *generatefeedvector.py* code to get the popular 500 words from all the atom feed.
7. I included a piece of code at line 34 in Listing 3to eliminate the words which falls under the stop words, the stop word list used is the "MySql full-text stopwords".
8. 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.
9. So to get the entries which are beyond 500, I wrote a piece of code from line 37 to 48 which handles that scenario. I am considering the 2000 entries for an atom feed and extracting the words from all those entries.
10. 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.
11. So that I can pick only the top 500 words by sorting the words based on the occurrence value.
12. 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.
13. 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.
14. To get the number of pages in each blog, I wrote a small python code as shown in Listing 4 which outputs the file *PagesInBlogs.txt*

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

16. From the Histogram in Figure 1, it's pretty clear that 30 blogs out of 100 have pages between 100 and 200, and only couple of blogs have pages between 2800 and 3000.

```
#! /bin/bash

echo ""

if [ $# -ne 1 ]
then
    echo "Usage: $0 #"
    exit
fi

num=$1

echo "http://f-measure.blogspot.com/" > blogList-$num.txt
echo "http://ws-dl.blogspot.com/" >> blogList-$num.txt

for ((i=0;i< $num;i++))
do
    curl -I -L 'http://www.blogger.com/next-blog?navBar=true&blogID=3471633091411211117' |
        grep Location | tail -n 1 | cut -d" " -f2 | cut -d"?" -f 1 >> blogList-$num.txt
done
```

Listing 1: Shell Program for getting 100 blogs

```
#! /bin/bash

if [ $# -ne 1 ]
then
    echo "Usage: $0 #"
    exit
fi

blogFile=$1

echo -n "" > `basename $blogFile .txt`-atom.txt

for line in `cat $blogFile`
do
    #curl $line | grep 'rel="alternate"' | grep atom | sed 's/.*href=//' | sed 's/\/>//' |
        sed 's/"//g' >> `basename $blogFile .txt`-atom.txt


    test=`curl $line | grep 'rel="alternate"' | grep atom | sed 's/.*href=//' | sed 's
        /\/>//' | sed 's/"//g' | sed 's/ //g'`
    echo "$test?max-results=1000" >> `basename $blogFile .txt`-atom.txt
done
```

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

```
import feedparser
import collections
import re
import operator


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

                        wc[word] += 1
            if pages == 500:
                next_link = url + "?start-index=1501"
```

```
72                  for e in d.entries:
73                      if 'summary' in e:
74                          summary = e.summary
75                      else:
76                          summary = e.description
77
78                      words = getwords('%s %s' % (e.title, summary))
79
80                      for word in words:
81                          if word not in stopwords:
82                              wc[word] += 1
83
84      if 'title' not in fd.feed:
85          print 'Invalid url', url
86          return 'bogus data', wc
87
88      return fd.feed.title, wc
89
90  def main():
91
92      # XXX: break this up into smaller functions, write tests for them
93      apcount    = collections.defaultdict(int)
94      wordcounts = {}
95      feedlist   = open('blogList-120-atom.txt').readlines()
96      totalWordCount = {}
97
98      for url in feedlist:
99          title, wc = getwordcounts(url)
100         wordcounts[title] = wc
101
102         for word, count in wc.iteritems():
103             if count > 1:
104                 apcount[word] += 1
105
106                 try:
107                     totalWordCount[word] += count
108                 except KeyError:
109                     totalWordCount[word] = count
110
111
112     wordlist = []
113     for w, bc in apcount.iteritems():
114         frac = float(bc)/len(feedlist)
115         #print frac
116         if frac > 0.1 and frac < 0.5:
117             wordlist.append(w)
118
119     countOfWords = []
120
121     for word in wordlist:
122         countOfWords.append((word,totalWordCount[word]))
123
124     countOfWords.sort(key=lambda rating: rating[1], reverse = True )
125
126     for words in countOfWords[0:500]:
127         print words[0] , words[1]
128     out = file('blogdata-120-1500pages.txt', 'w')
129     out.write('Blog')
130
131     for w in countOfWords[0:500]:
132         #print w
133         out.write('\t' + w[0])
134     out.write('\n')
135
136     for blogname, counts in wordcounts.iteritems():
137         blogname = blogname.encode('UTF-8')
138         out.write(blogname)
139
140         for w in countOfWords[0:500]:
141             word = w[0]
142             if w in counts:
143                 out.write('\t%d' % counts[word])
```

```
            else:
                out.write('\t0')
        out.write('\n')

    out.close()

if __name__ == '__main__':
    main()
```

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

```
import feedparser
import unicodedata

def gePagesCount(url):
    mainUrl = url.strip()
    d = feedparser.parse(url)
    pages = len(d['entries'])
    title = d['feed']['title']

    counter = 0
    if pages == 500:
        url = mainUrl +"&start-index=501"

        d = feedparser.parse(url)
        pages = int(pages) + int(len(d['entries']))



        if pages == 1000:
            url = mainUrl +"&start-index=1001"

            d = feedparser.parse(url)
            pages = int(pages) + int(len(d['entries']))

            if pages == 1500:
                url = mainUrl +"&start-index=1501"

                d = feedparser.parse(url)
                pages = int(pages) + int(len(d['entries']))

                if pages == 2000:
                    url = mainUrl +"&start-index=2001"

                    d = feedparser.parse(url)
                    pages = int(pages) + int(len(d['entries']))

                    if pages == 2500:
                        url = mainUrl +"&start-index=2501"

                        d = feedparser.parse(url)
                        pages = int(pages) + int(len(d['entries']))

    print u'|'.join((str(pages),title)).encode('utf-8').strip()

def main():
    feedlist    = open('blogList-120-atom.txt').readlines()

    for url in feedlist:
        try:
            gePagesCount(url)
        except KeyError:
            pass

if __name__ == '__main__':
    main()
```

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

```
data <- read.csv("PagesInBlogs2.txt", stringsAsFactors = F, header = FALSE, sep = "|")

incdata = data[, 1]

incdata <- as.numeric(incdata)

brk <- seq(0, 3000, 100)

png("q1-histogram1.png")
hist(incdata, main = "Blogs vs. Number of Pages", breaks=brk, freq = T, xlab="Pages", ylab="
    Blogs")
```

Listing 5: R Script for generating a Histogram

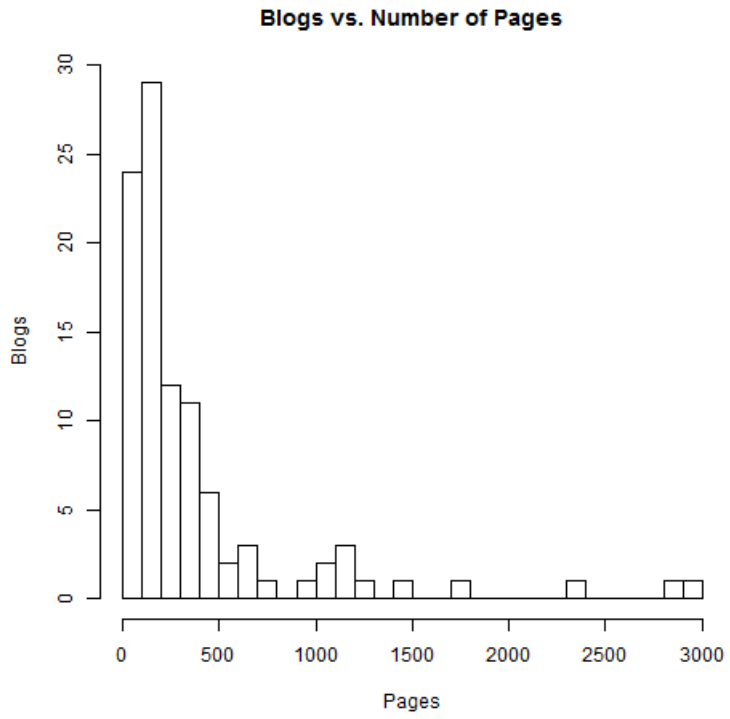

Figure 1: Histogram showing Blogs Vs No of Pages

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

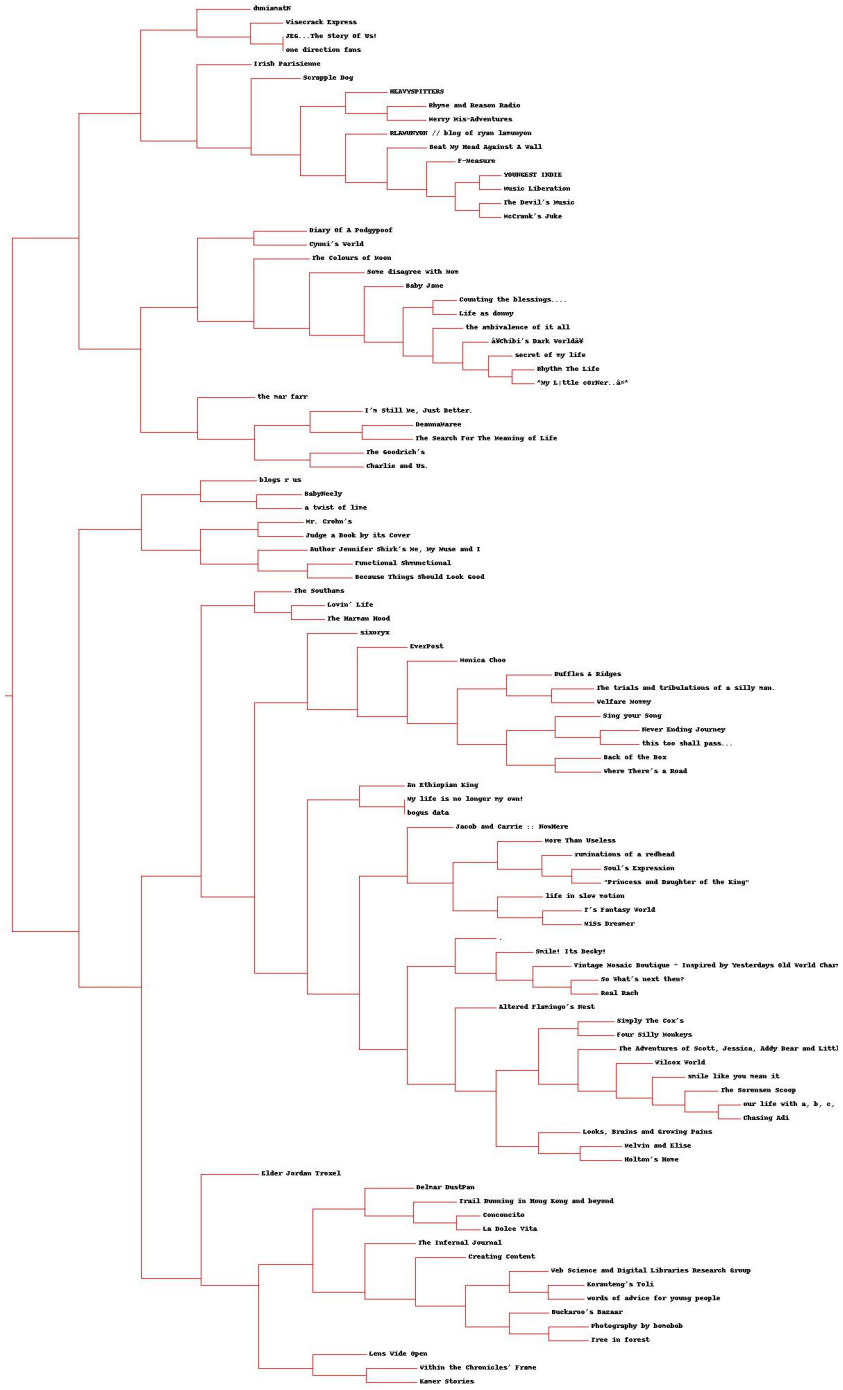

Figure 2: Dendrogram Produced by Program in Listing 6

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.

```
1  #!/usr/local/bin/python
2
3  import clusters
4
5  blognames,words,data=clusters.readfile('blogdata120-atom-500.txt')
6
7  print "For k=5"
8  kclust=clusters.kcluster(data, k=5)
9  print
10
11 print "For k=10"
12 kclust=clusters.kcluster(data, k=10)
13 print
14
15 print "For k=20"
16 kclust=clusters.kcluster(data, k=20)
17 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.

```
1 #!/usr/local/bin/python
2
3 import clusters
4
5 blognames,words,data=clusters.readfile('blogdata120-atom-500.txt')
6
7 coords = clusters.scaledown(data)
8
9 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.

```
4532.41130905
3366.98206519
3326.3306574
3308.09464169
3297.94258112
3291.51602489
3287.28793801
3283.77516396
3280.99776391
3278.44993199
3275.81243841
3273.31397104
3270.8684595
3268.42055835
3266.16460163
3264.23476083
3262.42227912
3260.51325928
3258.82854152
3257.18175939
3255.37923243
3253.30711541
3251.00612794
3248.77308262
3246.70118667
3244.47115104
3241.94493708
3239.40739168
3236.99558481
3234.81656727
3232.88203014
3231.04567171
3229.23063872
3227.47967223
3225.91794645
3224.58649843
3223.20000436
3222.0867435
3221.01271989
3219.95473094
3218.75956428
3217.59979619
3216.4522233
3215.34629148
3214.36905778
3213.26833436
3212.24470346
3211.31994368
3210.29982355
3209.20246315
3207.95132503
3206.82241072
3205.91909053
3204.92220274
3204.31947561
3203.74640843
3203.27606811
3202.67555472
3201.89979232
3200.88578203
3200.01531056
3199.2966446
3198.45478403
3197.6201472
3196.73438036
3196.02119032
3195.41196943
3194.72376496
3194.0329727
3193.41355901
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

```
3095.71194597
3095.44859148
3095.20946085
3095.07182068
3095.06452066
3095.04654607
3094.9754195
3094.85516015
3094.79234814
3094.67410977
3094.52095848
3094.33078357
3094.08903464
3093.79059454
3093.51468176
3093.20791956
3092.91243999
3092.65113648
3092.34912442
3092.05447449
3091.80452138
3091.55424259
3091.27376913
3090.97576404
3090.65628056
3090.30346956
3089.97015589
3089.57549888
3089.19007073
3088.82871046
3088.44864717
3088.11602282
3087.91481431
3087.80681744
3087.74255397
3087.66780485
3087.65822646
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

```
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

```

```
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

```
#!/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)
```

```
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
```

```
  # The real distances between every pair of items
    realdist = [[distance(data[i], data[j]) for j in range(n)] for i in
                range(0, n)]

  # Randomly initialize the starting points of the locations in 2D
    loc = [[random.random(), random.random()] for i in range(n)]
    fakedist = [[0.0 for j in range(n)] for i in range(n)]

    lasterror = None
    for m in range(0, 1000):
    # Find projected distances
        for i in range(n):
            for j in range(n):
                fakedist[i][j] = sqrt(sum([pow(loc[i][x] - loc[j][x], 2)
                                           for x in range(len(loc[i]))]))

    # Move points
        grad = [[0.0, 0.0] for i in range(n)]

        totalerror = 0
        for k in range(n):
            for j in range(n):
                if j == k:
                    continue
            # The error is percent difference between the distances
                errorterm = (fakedist[j][k] - realdist[j][k]) / realdist[j][k]

            # Each point needs to be moved away from or towards the other
            # point in proportion to how much error it has
                grad[k][0] += (loc[k][0] - loc[j][0]) / fakedist[j][k] \
                    * errorterm
                grad[k][1] += (loc[k][1] - loc[j][1]) / fakedist[j][k] \
                    * errorterm

            # Keep track of the total error
                totalerror += abs(errorterm)
            print totalerror

    # If the answer got worse by moving the points, we are done
        if lasterror and lasterror < totalerror:
            break
        lasterror = totalerror

    # Move each of the points by the learning rate times the gradient
        for k in range(n):
            loc[k][0] -= rate * grad[k][0]
            loc[k][1] -= rate * grad[k][1]

    return loc


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

Listing 11: Segaran's *clusters.py*

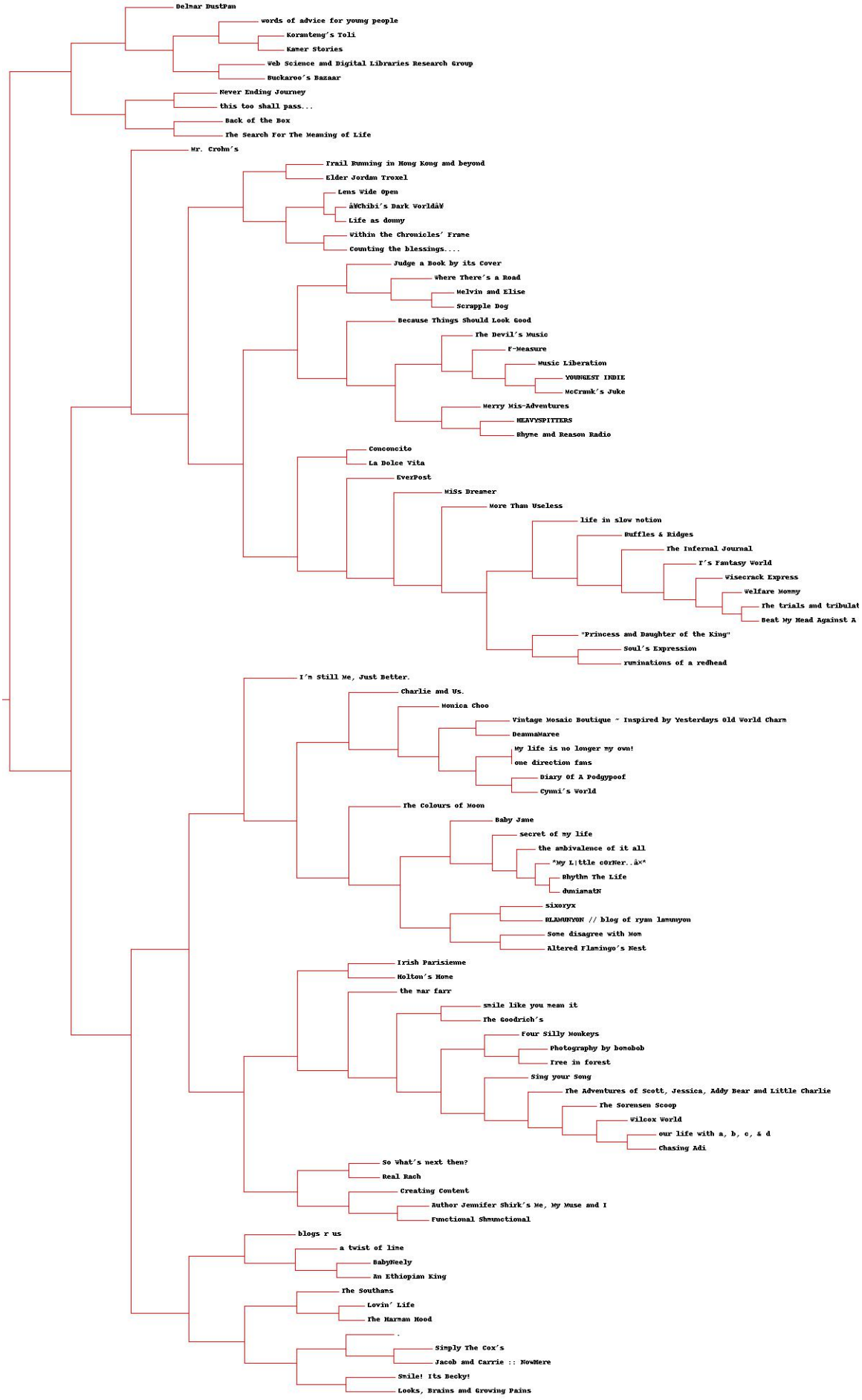

Figure 4: Dendrogram produced based on word's TFIDF values script

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.