Introduction to Web Science: Assignment #9

Dr. Nelson

Alexander Nwala

Thursday, December 4, 2014

Contents

Problem 1

(10 points; 2 points for each question and 2 points for aesthetics)

Support your answer: include all relevant discussion, assumptions, examples, etc.

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).

SOLUTION 1

The solution for this problem is outlined by the following steps:

1. **Collect 98 blogs:** This was achieved by continuously retrieving the url due to invoking the head of http://www.blogger.com/next-blog?navBar=true&blogID=3471633091411211117, as outlined by Listing 1.

Listing 1: Grab 100 Unique Blogs

```
#grab 100 unique blogs
def getNUniqueBlogs(count):
     listOfBlogs = []
     if( count>0 ):

          try:
               outputFile = open('listOfUniqueBlogs.txt', 'w')

               outputFile.write('http://f-measure.blogspot.com/\n')
               outputFile.write('http://ws-dl.blogspot.com/\n')
          except:
               exc_type, exc_obj, exc_tb = sys.exc_info()
               fname = os.path.split(exc_tb.tb_frame.f_code.co_filename)[1]
               print(fname, exc_tb.tb_lineno, sys.exc_info() )

          while len(listOfBlogs) != count:

               blogUrl = 'http://www.blogger.com/next-blog?navBar=true&blogID
                   =3471633091411211117'
```

```
                finalBlog = followTheRedirectCurl(blogUrl)

                if len(finalBlog) > 0:
                    if finalBlog not in listOfBlogs:

                        finalBlog = finalBlog.lower()

                        parsedUrl = urlparse.urlparse(finalBlog)
                        parsedUrl = parsedUrl.scheme + '://' + parsedUrl.netloc +
                            '/'

                        listOfBlogs.append(parsedUrl)
                        outputFile.write(parsedUrl + '\n')

        outputFile.close()
getNUniqueBlogs(100)
```

2. **Generate blog matrix:** This was achieved by utilizing the PCI book's [1] generatefeedvector.py with the additional modification of exploring multiple pages (pagination) of blogs as outlined in Listing 2.

```
The file blogVector.txt contains the blog matrix
```

Listing 2: Generate Blog Matrix

```
#modified to look at all pages of the blog
def generateFeedVector(countOfBlogsToExplore=10):
...
#get wc for other pages - start
    listOfFeedPages = listOfPageFeeds(feedurl)

    #print '...bef', title, len(wc)
    for feedPage in listOfFeedPages:
        feedPage = feedPage.strip()
        (sameTitle, nextPageWordCount) = getwordcounts(feedPage)

        consolidateDictionaries(wc, nextPageWordCount)

    #print '...aft', title, len(wc)

    #get wc for other pages - encoded
...
```

3. **Limit the number of terms to the 500 most "popular":** This was achieved my adding the following block (Listing 3.) into generatefeedvector.py

Listing 3: 500 Most Popular Blogs

```
...
#Limit the number of terms to the most "popular" (i.e., frequent) 500 terms
#sort by frequency
arrayOfTermTermFrequencyTuples = sorted(arrayOfTermTermFrequencyTuples, key=lambda
    tup: tup[1], reverse=True)
```

```
if( len(arrayOfTermTermFrequencyTuples) > 0 ):

     for termFrequencyTuple in  arrayOfTermTermFrequencyTuples:
          term = termFrequencyTuple[0]
          termFrequency = termFrequencyTuple[1]

          #print term, termFrequency

          #get 500 most popular terms
          if( len(wordlist) <= 500 ):
               wordlist.append(term)
          else:
               break
...
```

4. **Create blog page count histogram:** This was achieved due to Listing 4. (Generated the data) and Listing 5. (Plotted the data - Figure 1). Pagination was achieved due to the recursive method recursivelyGetFeedPagesForBlog() (Listing 4.) which continued to get the links to the next page until no link was available.

Listing 4: Generate Blog Histogram

```
#Generate blog page count histogram data
def getBlogPageCount(maxCountOfBlogsToExplore=5):

     if( maxCountOfBlogsToExplore > 0 ):

          try:
               inputFile = open('100listOfUniqueBlogs.txt', 'r')
               lines = inputFile.readlines()
               inputFile.close()
               outputFile = open('BLOG-PAGECOUNT.txt', 'w')
          except:
               exc_type, exc_obj, exc_tb = sys.exc_info()
               fname = os.path.split(exc_tb.tb_frame.f_code.co_filename)[1]
               print(fname, exc_tb.tb_lineno, sys.exc_info() )


          dictionaryOfPageCounts = {}
          for line in lines:
               line = line.strip()


               listOfFeedPages = listOfPageFeedsPageCount(line)
               pageCount = len(listOfFeedPages) + 1

               dictionaryOfPageCounts.setdefault(pageCount, 0)
               dictionaryOfPageCounts[pageCount] += 1


               if( len(dictionaryOfPageCounts) == maxCountOfBlogsToExplore ):
                    break
```

```
            outputFile.write('PageCount PageCountFrequency \n')
            if( len(dictionaryOfPageCounts) > 0 ):
                for pageCount, pageCountFrequency in dictionaryOfPageCounts.items()
                    :
                     outputFile.write(str(pageCount) + ' ' + str(pageCountFrequency
                         ) + '\n')

            outputFile.close()

def recursivelyGetFeedPagesForBlog(blogUrl, listOfPages=[]):

     if( len(blogUrl) > 0 ):
            try:
                html = requests.get(blogUrl)
            except:
                countOfPages = 0

            soup = BeautifulSoup(html.text)
            nextLink = soup.find('link', { 'rel' : 'next' })

            if nextLink is not None:
                nextLink = nextLink['href']
                listOfPages.append(nextLink)

                recursivelyGetFeedPagesForBlog(nextLink, listOfPages)
     return listOfPages
```

Listing 5: Plot Blog Histogram

```
#!/usr/bin/env Rscript
rawInput <- read.table('BLOG-PAGECOUNT.txt', header=T)
hist(rawInput$PageCountFrequency, main="", xlab="BlogPageCount", col="lightblue")
```

Problem 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).

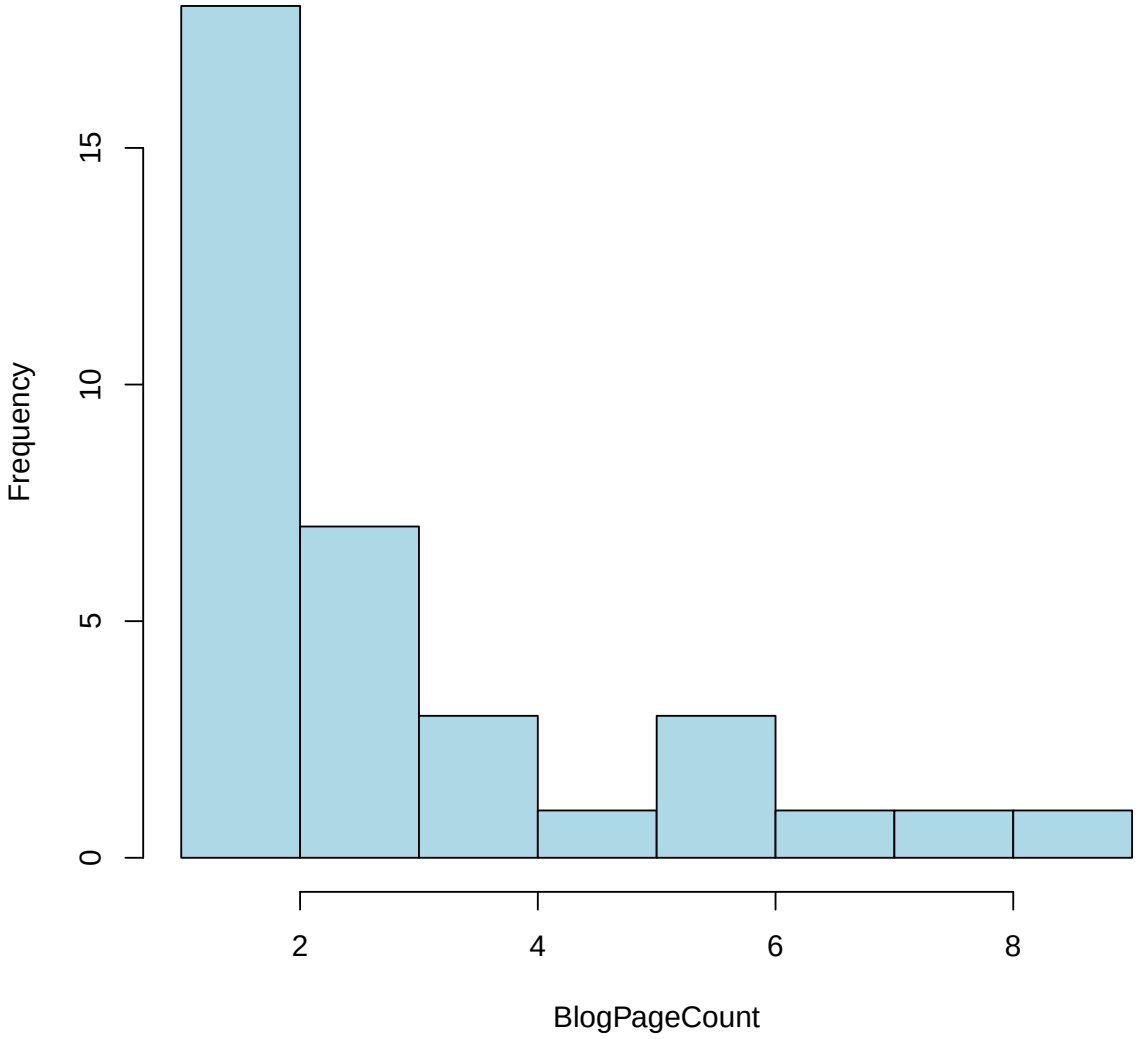
SOLUTION 2

The solution for this problem is outlined by the following steps:

1. **Create dendograms:** This was achieved due to Listing 6. Figure 2. shows the JPEG dendogram and Listing 7. contains an excerpt of the ASCII dendogram.

```
The file blogAsciiDendogram.txt contains the ASCII blog dendogram
The file blogclust.jpg contains the Jpeg blog dendogram
```

Figure 1: Distribution of Pages



Problem 2 continued on next page. . .

Listing 6: Create Dendograms

```
#create dendograms
def createAsciiDendogram():
     blognames,words,data=clusters.readfile('blogVector.txt')
     clust=clusters.hcluster(data)

     clusters.printclust(clust,labels=blognames)

def creatJPegDendogram():
     blognames,words,data=clusters.readfile('blogVector.txt')
     clust=clusters.hcluster(data)
     clusters.drawdendrogram(clust,blognames,jpeg='blogclust.jpg')
```

Listing 7: Blogs ASCII Dendogram Excerpt

```
-
  -
    JasonTankerley.com
    -
      Final Approach
      -
        -
          -
            -
              Jazz with Jerry Brabenec
              -
                Best Mp3 Blog
                Web Science and Digital Libraries Research Group
            -
              Fu-Jen Catholic University Digital Music Club
              NO Kiddin
                        .
                        .
                        .
                        Evin On Earth
                        -
                          HK
                          Confessions of a very troubled mind.
                      -
                        NallsBlog
                        -
                          In Search of an Elusive Band
                          -
                            -
                              Bruunski Beats
                              F-Measure
                        .
                        .
                        .
                  -
                    Nana Goes Lala
                    -
                      T i F Fs
                      -
```

```
                    hmm?
                    And youll see me waiting For you On our corner of the
                        street
```

Problem 3

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

SOLUTION 3

The solution for this problem is outlined by the following steps:

1. **Create clusters:** This was achieved due to Listing 8. Table 1. outlines the k-value/iteration tuples.

Listing 8: Create Clusters

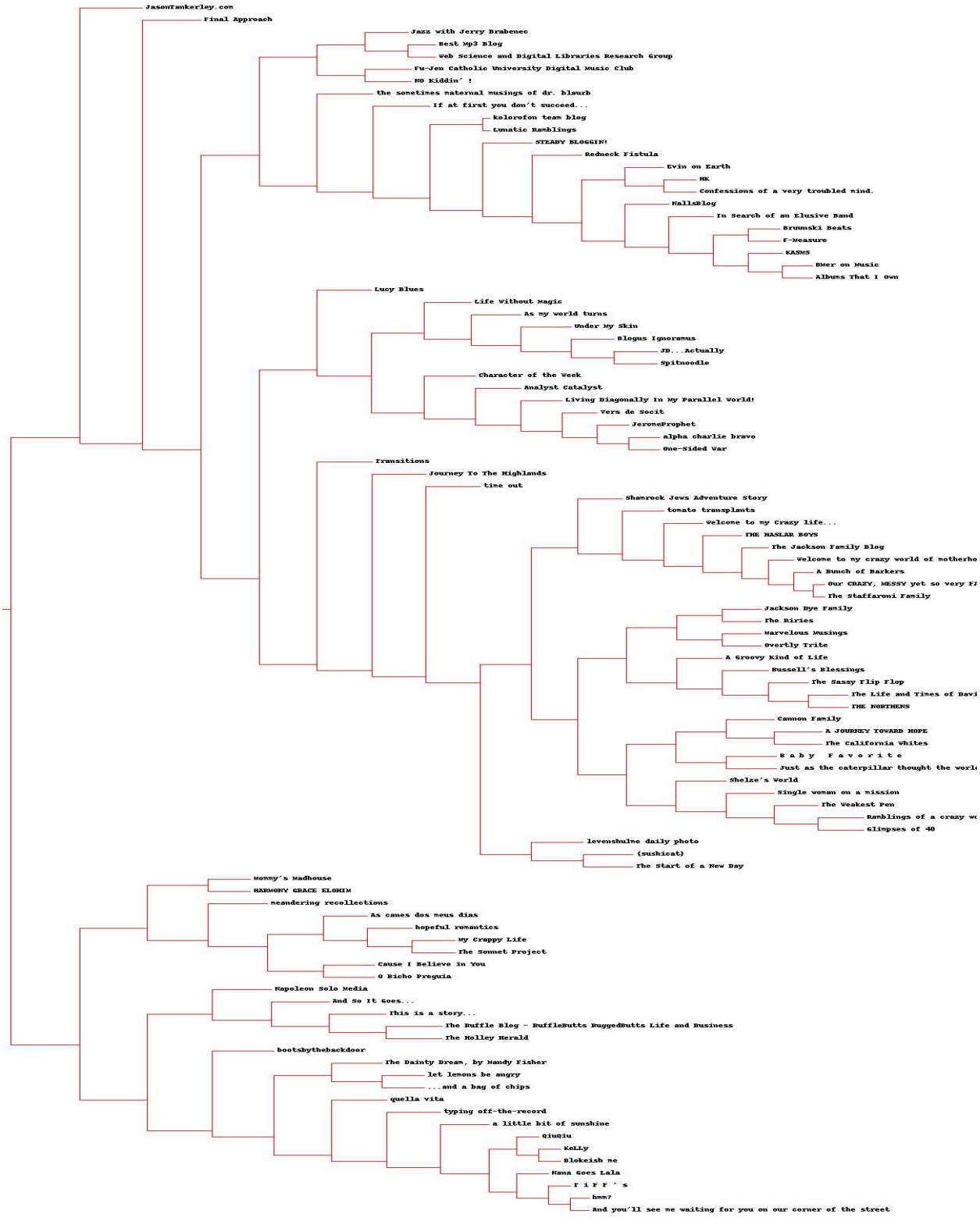
```
#create clusters
def createKMeansClusters(kValue):

    if( kValue>0 ):
        blognames,words,data=clusters.readfile('blogVector.txt')
        kclust=clusters.kcluster(data,k=kValue)

        count = 0
        for cluster in kclust:

            if( len(cluster) > 0 ):
                print 'cluster', count
                for instance in cluster:
                    print '...',blognames[instance]

                count += 1
```

The file kMeansClusters.txt contains the clusterings/iteration count for k = 5, 10, 20

Table 1: k-value vs Iteration Count

ITEM	k-value	Iteration
1	5	5
2	10	7
3	20	5

Figure 2: Blogs Dendogram



Problem 4

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

SOLUTION 4

The solution for this problem is outlined by the following steps:

1. **Reduce dimension:** This was achieved due to Listing 9. The iteration count was achieved by imposing a counter as long as the error continued to drop across iterations. Figure 3. Shows the resultant blog matrix in 2d.

Listing 9: Create MDS

```
#MDS
def createMDS():
     blognames,words,data=clusters.readfile('blogVector.txt')
     coords,iterationCount=clusters.scaledown(data)
     clusters.draw2d(coords,blognames,jpeg='blogs2d.jpg')

     print 'iterationCount', iterationCount
```

Problem 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.

SOLUTION 5

The solution for this problem is outlined by the following steps:

1. **Calculate TFIDF values:** This was achieved by replacing the word count in generateFeedVector() with the TFIDF values as outlined by Listing 10. Given a word (w) in a file (f), Listing 10. computes the term frequency of w by dividing the word count of w by the length of f (total number of words). And the inverse document frequency is computed by dividing the corpus length by the total number of files which contain w. Thereafter, the term frequency-inverse document frequency is computed by multiplying the term frequency by the log of the inverse document frequency.

Listing 10: Blog Matrix For TFIDF

```
generateFeedVector()
...
```

Problem 5 continued on next page...

Figure 3: MDS Diagram, iterationCount = 226

Problem 5 continued on next page...

```
        word = word.encode('ascii', 'ignore')
        if word in wc:

            #tf for this word:
            termFrequency = wc[word]/float(len(wc))
            #idf
            inverseDocumentFrequency = logBase2(iteration/float(apcount[word]))
            #tfidf
            tfIdf = termFrequency*inverseDocumentFrequency

            #print '...tfIdf', tfIdf

            out.write('\t%f' % tfIdf)
        else:
            out.write('\t0')
...
```

2. **Create dendogram:** Similar to the first dendogram, Figure 4. outlines the dendogram resulting from using TFIDF values instead of word counts.

Comparison/Contrast - (Figure 2 and Figure 4)

The blogs dendogram due to TFIDF is more balanced compared to the blogs dendogram (word count) which is more skewed (unbalanced partitions). Also, the blogs dendogram (TFIDF) contains more clusters meaning TFIDF does a more fine-grained similarity comparison.

Although structurally the dendograms look dissimilar, semantically they are similar. Take the random cluster neigborhood of the F-measure blog as outlined in Figure 5, it can be seen that a couple of blogs are common within the neighborhood of F-measure.

Figure 4: Blogs Dendogram (TFIDF Variant)

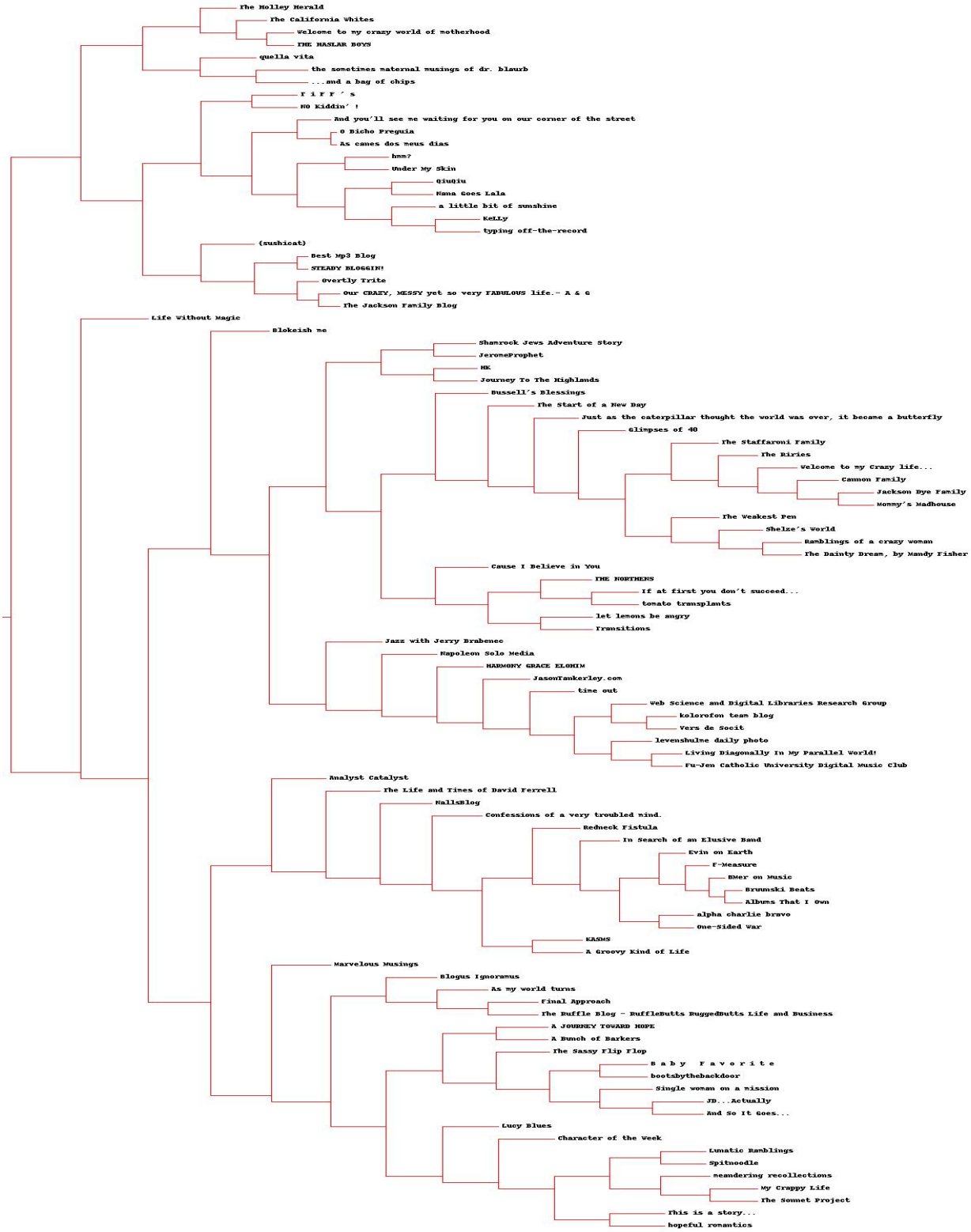
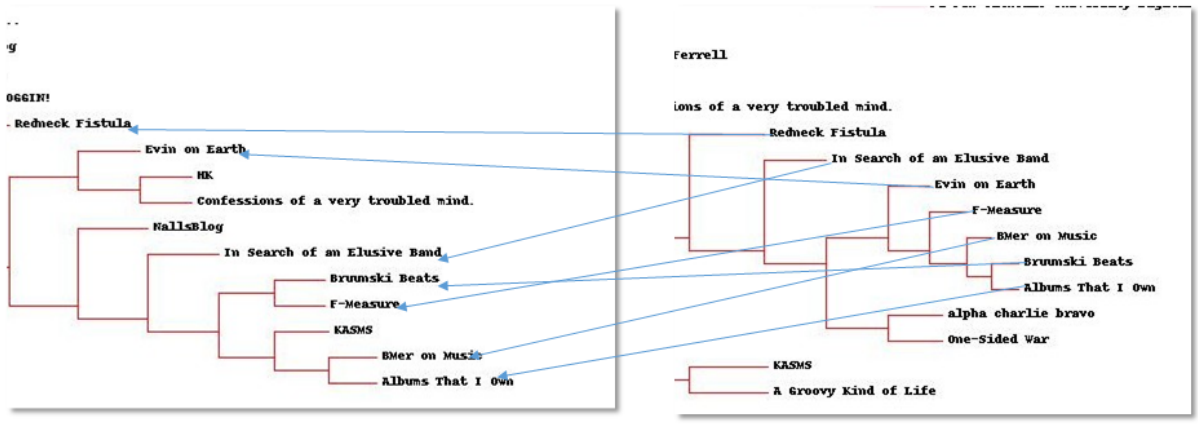


Figure 5: Similarity Of Dendograms (TFIDF (left) vs. Word Count(Right)) Around F-measure Neighborhood



References

[1] Toby Segaran. Programming Collective Intelligence, 2007.