Amara Naas (anaas)
CS595 — Fall 2013

Assignment 9

12/5/2013

In my approach to attempt this assignment I used some functions which available at chapter three of our text book[1] and used the *100 blogs* from [2], including:

- `http://f-measure.blogspot.com/feeds/posts/default`
- `http://ws-dl.blogspot.com/feeds/posts/default`

as specified in assignment 9. I found that 17 out of the 100 blogs do not have the correct RSS blog so I had to replace them with the correct ones. *17blogsListError.txt* shows these 17 blogs. In order to solve the four problems in this assignment I created two python programs *Assignment9A.py*[1] and *Assignment9B.py*[1] which will solve problem 1 and problems 2-4 respectively.

Problem 1

In first program I used two functions and the *list.txt*[1] file. First *getwordcounts* function passes the summary or the description, that*RSS' list* of entries have, to the second *getwords* function and returns title and dictionary of word counts for an RSS feed.

For solving problem 1 the feeding list of 100 blogs and Assignment9A.py are used which will do the following:

- Looping over the feeds to read blogs from list.txt and generates the word counts for each blog and the number of blogs each word appeared in apcount dictionary.

```
apcount={}
wordcounts={}
f = open("list1.txt","r")
blogs_list = f.readlines()
f.close()
for feedurl in blogs_list:
        try:
                title,wc=getwordcounts(feedurl)
                wordcounts[title]=wc
                for word,count in wc.items():
                        apcount.setdefault(word,0)
                        if count>1:
                                apcount[word]+=1
        except:
                print '\nProblem parsing:\n- %s' % feedurl
                print sys.exc_info()
```

- Sorting words in descending order, reducing their total number by maximum (50 %) and minimum (10 %) percentages, and filter out single letter words.

[1]File uploaded to github

```
# Sort the word, blogcount in descending order.
# When we apply the filtering criteria, the words
# will already be in order by frequency
wordlist=[]
for w,bc in sorted(apcount.items(), key=itemgetter(1), reverse=True):
        frac=float(bc)/len(blogs_list)
## you can reduce the total number of words
## included by selecting only those words
## that are within maximum and minimum
## percentages. In this case, you can start with 10 percent
## as the lower bound and 50 percent as the upper bound.
```

- Using the list of first 500 words and the list of blogs to create a text file called *BlogMatrix.txt* which contains a big matrix of all the word counts for each of the blogs.

```
                wordlist.append(w)
## The final step is to use the list of words and the list of blogs
## to create a text file containing a big matrix of all the word
## counts for each of the blogs.
out=file('BlogMatrix1.txt','w')
out.write('Blog')

# First 500 words
for word in wordlist[0:500]: out.write('\t%s' % word)
out.write('\n')
for blog,wc in wordcounts.items():
        print blog
        out.write(blog.encode('utf-8'))
        for word in wordlist[0:500]:
                if word in wc: out.write('\t%d' % wc[word])
                else: out.write('\t0')
```

Problem 2

For problem two I used four function from [1] as described in *Assignment9B.py*.

1. First *readfile* function will read list of words for each column-data, list of blogs names, and a big list of data where every item in the list is the data for that blog in particular row.

```
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
```

2. For defining closeness a *pearson* correlation function is used because some blogs contain more entries or much longer entries than others, and will thus contain more words overall. It tries to determine how well two sets of data fit onto a straight line.

```
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
```

3. The *bicluster* class is used because the cluster in a hierarchical clustering algorithm has some properties such as being a point in the tree with two branches, or an endpoint associated with a blog, or contains data about its location.

```
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
```

4. hierarchical function *hcluster* creates group of clusters as following:

(a) Creates initial group clusters which are just the original items.

```
        distances={}
        currentclustid=-1
# Clusters are initially just the rows
        clust=[bicluster(rows[i],id=i) for i in range(len(rows))]
```

(b) Finding the two best matches of blogs to form the new single cluster until only one cluster is remains.

```
while len(clust)>1:
        lowestpair=(0,1)
        closest=distance(clust[0].vec,clust[1].vec)
# loop through every pair looking for the smallest distance
        for i in range(len(clust)):
                for j in range(i+1,len(clust)):
# distances is the cache of distance calculations
                        if (clust[i].id,clust[j].id) not in
                            distances:
                                distances[(clust[i].id,clust[j].id)]=
                                distance(clust[i].vec,clust[j].vec)
                        d=distances[(clust[i].id,clust[j].id)]
                        if d<closest:
                                closest=d
                                lowestpair=(i,j)
# calculate the average of the two clusters
        mergevec=[(clust[lowestpair[0]].vec[i]+
            clust[lowestpair[1]].vec[i]) /2.0 for i in
            range(len(clust[0].vec))]
# create the new cluster
        newcluster=bicluster(mergevec,left=clust[lowestpair[0]],
        right=clust[lowestpair[1]],
        distance=closest,id=currentclustid)
# cluster ids that weren't in the original set are negative
        currentclustid-=1
        del clust[lowestpair[1]]
        del clust[lowestpair[0]]
        clust.append(newcluster)
return clust[0]
```

(c) This function uses pearson correlation and averaging the data for the two old clusters.

(d) The final cluster returned by this function can be searched recursively to recreate all the clusters and their end nodes.

5. *printcluster* function writes all clusters to file called *Q2-ASCII.txt* using the re-

cursive search.

```
def printclust(clust,labels=None,n=0):
        # indent to make a hierarchy layout

        for i in range(n):    f.write (' '),
        if clust.id<0:
        # negative id means that this is branch
                f.write ('-\n')
        else:
        # positive id means that this is an endpoint
                if labels==None:
                        f.write (clust.id+'\n')
                else:
                        f.write (labels[clust.id]+'\n')
        # now print the right and left branches
        if clust.left!=None:  printclust(clust.left,labels=labels,n=n+1)
        if clust.right!=None:
            printclust(clust.right,labels=labels,n=n+1)
```

6. The *drawdendrogram* function uses the Python Imaging Library (PIL) which generates images with text and lines and save it in blogclust.jpg. This function dose the following:

   (a) Calls another recursive function called *getheight* which gives the total height of a given cluster. If the cluster is an endpoint, then its height is 1; otherwise, its height is the sum of the heights of its branches.

   ```
   def getheight(clust):
           # Is this an endpoint? Then the height is just 1
           if clust.left==None and clust.right==None:
                   return 1
           # Otherwise the height is the same of the heights of
           # each branch
           return getheight(clust.left)+getheight(clust.right)
   ```

   (b) It also calls another recursive function called *getdepth* which returns the distance (The total error of the root node) of an endpoint or the max distance of a branch.

   ```
   def getdepth(clust):
           # The distance of an endpoint is 0.0
           if clust.left==None and clust.right==None:
                   return 0
           # The distance of a branch is the greater of its two sides
           # plus its own distance
   ```

```
        return
            max(getdepth(clust.left),getdepth(clust.right))+clust.distance
```

(c) It has a scaling factor which determined by dividing the fixed width (1200) by the total depth returned from getdepth function.

```
def drawdendrogram(clust,labels,jpeg='clusters.jpg'):
        # height and width
        h=getheight(clust)*20
        w=2000
        depth=getdepth(clust)
        # width is fixed, so scale distances accordingly
        scaling=float(w-150)/depth
```

(d) After it creates a draw object for this image it calls the *drawnode* function to be halfway down the left side of this image.

```
        # Create a new image with a white background
        img=Image.new('RGB',(w,h),(255,255,255))
        draw=ImageDraw.Draw(img)
        draw.line((0,h/2,10,h/2),fill=(255,0,0))
        # Draw the first node
        drawnode(draw,clust,10,(h/2),scaling,labels)
        img.save(jpeg,'JPEG')
```

7. *drawnode* function takes a cluster and its location and draws lines one long vertical line and two horizontal lines for each child where the lengths of the horizontal lines show the similarity of clusters. Shorter line indicate that clusters were almost identical.

```
def drawnode(draw,clust,x,y,scaling,labels):
    if clust.id<0:
            h1=getheight(clust.left)*20
            h2=getheight(clust.right)*20
            top=y-(h1+h2)/2
            bottom=y+(h1+h2)/2

    # Line length
            ll=clust.distance*scaling
    # Vertical line from this cluster to children
            draw.line((x,top+h1/2,x,bottom-h2/2),fill=(255,0,0))
    # Horizontal line to left item
            draw.line((x,top+h1/2,x+ll,top+h1/2),fill=(255,0,0))
    # Horizontal line to right item
            draw.line((x,bottom-h2/2,x+ll,bottom-h2/2),fill=(255,0,0))
    # Call the function to draw the left and right nodes
```

```
        drawnode(draw,clust.left,x+ll,top+h1/2,scaling,labels)
        drawnode(draw,clust.right,x+ll,bottom-h2/2,scaling,labels)
else:
# If this is an endpoint, draw the item label
        draw.text((x+5,y-7),labels[clust.id],(0,0,0))
```

Problem 3

I used kcluster function from [1] to solve problem 3 which operates based on an algorithm called K-Means Clustering. This algorithm breaks the data into distinct groups because it is told in advance how many distinct clusters to generate. this function dose the following:

- Takes the same data rows obtained from *readfile* function and the number of clusters *(k)* as an input.
- Place a K points randomly in space that represent the centre of the cluster and assigns each blog to the nearest one.

```
# Determine the minimum and maximum values for each point
ranges=[(min([row[i] for row in rows]),max([row[i] for row in
    rows]))
for i in range(len(rows[0]))]
# Create k randomly placed centroids
clusters=[[random.random()*(ranges[i][1]-ranges[i][0])+ranges[i][0]
```

- Moves centroids to the average location of all the nodes that assigned to them.

```
# Move the centroids to the average of their members
        for i in range(k):
                avgs=[0.0]*len(rows[0])
                if len(bestmatches[i])>0:
                        for rowid in bestmatches[i]:
                                for m in range(len(rows[rowid])):
                                        avgs[m]+=rows[rowid][m]
                        for j in range(len(avgs)):
                                avgs[j]/=len(bestmatches[i])
                        clusters[i]=avgs
```

- Repeats this process until the assignments stop changing.

```
# If the results are the same as last time, this is complete
        if bestmatches==lastmatches: break
```

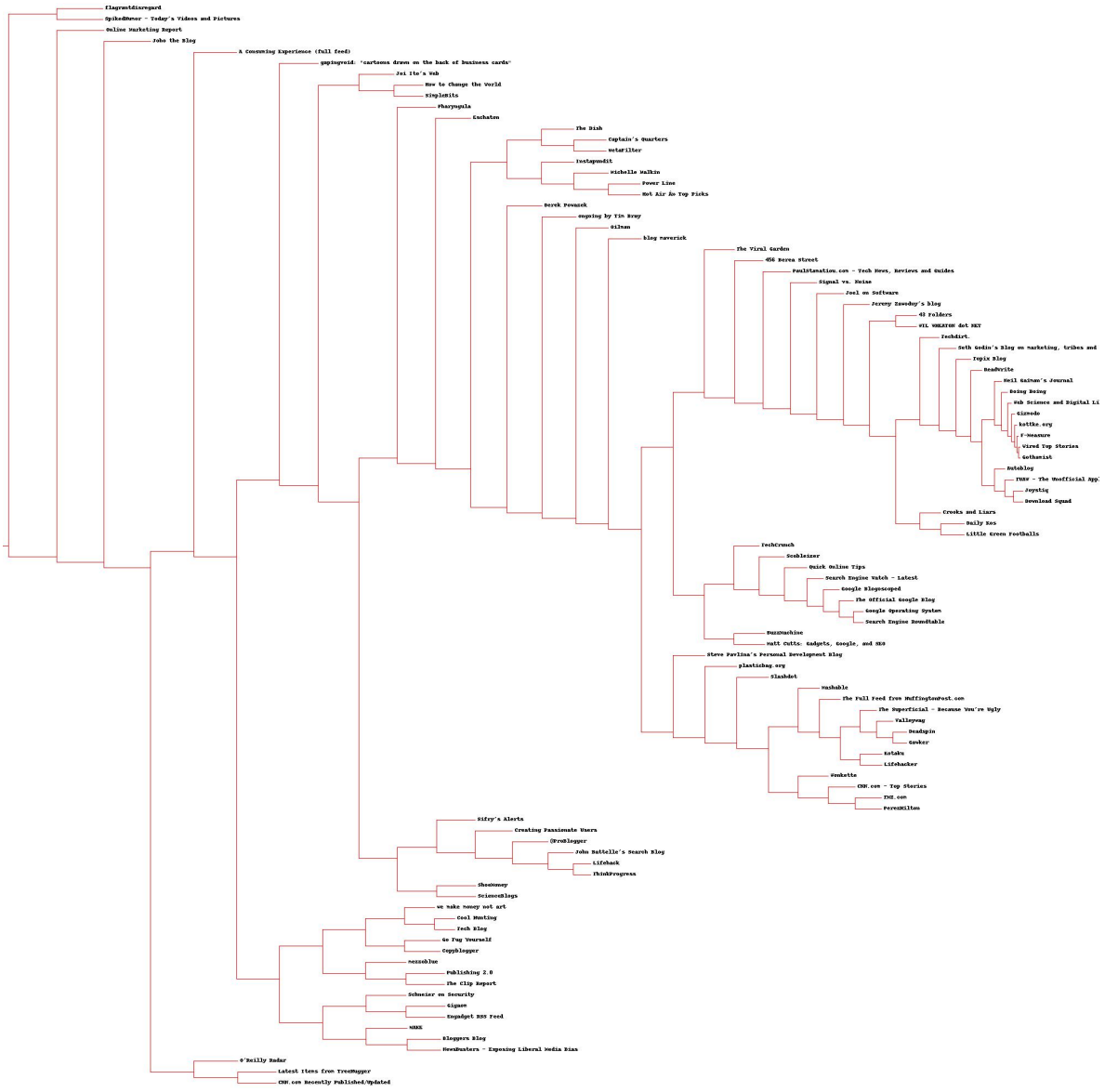

Figure 1: Dendrogram showing blog clusters

- Returns the k lists and the number of iterations it takes to produce the final result.

Table 1 shows the results when k is 5, 10, and 20.

k=5	k=10	k=20
5	7	5

Table 1: The interations required for k=5,10,20

Problem 4

For viewing data in two dimensions I used two functions *scaledown* and *draw2d* that obtained from [1]. First function uses multidimensional scaling technique and does the following:

- Takes the data vector obtained from *readfile* function and finds the difference between each pair of blogs and matches them to a distances using Pearson correlation. These distances will represent the distances between the blogs in the chart.

```
def scaledown(data,distance=pearson,rate=0.01):
        n=len(data)
        # 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]))]))
```

- Every node representing a blog is moved according to the combination of all the other nodes pushing and pulling on it based on how close they are.

```
        # 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
```

- This procedure is repeated many times until the distance cannot be reduced by moving the nodes any more.

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

- Returns the X and Y coordinates of the blogs on the two-dimensional chart.

The second function uses use PIL to generate an image with all the labels of all the different blogs plotted at the coordinates of that blog which obtained from first function.

```
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')
```

Problem 5

For Creating an ASCII and JPEG dendrogram that clusters the most similar blogs using TFIDF calculations I made one change to the *Assignment9A.py* which is finding the total number of words in each count and run the code in *Assignment9X.py* to create the dendrogram. It seems that there are some changes due to the use of the same 500 words. Each time we parse the blogs we get different data and using TFIDF with the same words from question 2 increase the chances of a pair of blogs being different. Figure 4 and 5 show the ASCII and JPEG dendrogram.

Figure 2: 2D representation of blog space

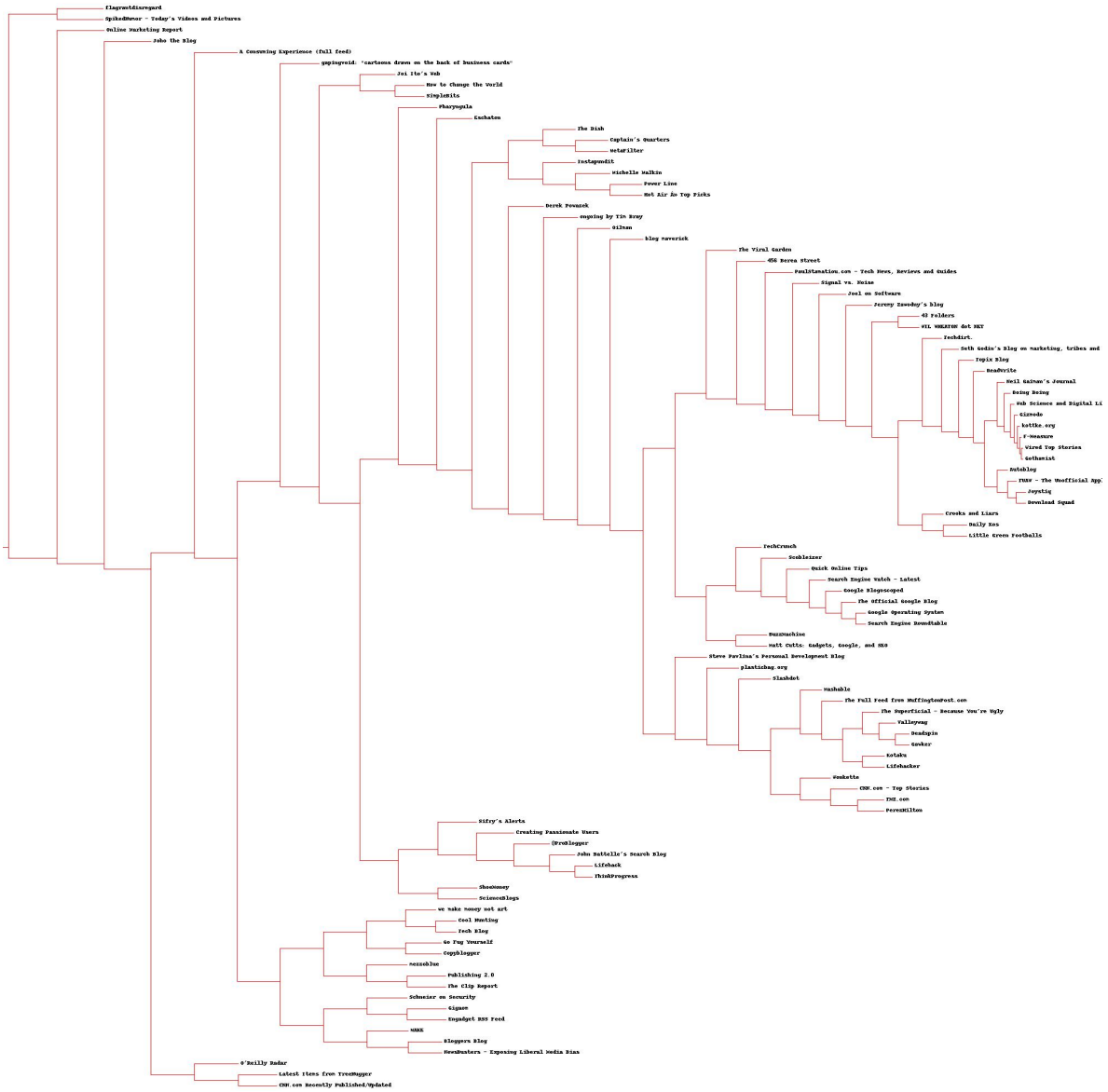

Figure 3: Dendrogram showing blog clusters using TFIDF

Figure 4: 2D representation of blog space using TFIDF

References

[1] T. Segaran, Discovering Groups, pp. 29–53 in: "Programming Collective Intelligence", O'REILLY, Aug. 2007.

[2] `https://github.com/nico/collectiveintelligence-book/blob/master/feedlist.txt`.