```
      ================================================================================================
                                        Ch2 - Recommendations
          ----------------------------------------------------------------------------------
# A dictionary of movie critics and their ratings of a small
# set of movies
critics={
  'Lisa Rose': {
    'Lady in the Water': 2.5,
    'Snakes on a Plane': 3.5,
    'Just My Luck': 3.0,
    'Superman Returns': 3.5,
    'You, Me and Dupree': 2.5,
    'The Night Listener': 3.0},
  'Gene Seymour': {
    'Lady in the Water': 3.0,
    'Snakes on a Plane': 3.5,
    'Just My Luck': 1.5,
    'Superman Returns': 5.0,
    'The Night Listener': 3.0,
    'You, Me and Dupree': 3.5},
  'Michael Phillips': {
    'Lady in the Water': 2.5,
    'Snakes on a Plane': 3.0,
    'Superman Returns': 3.5,
    'The Night Listener': 4.0},
  'Claudia Puig': {
    'Snakes on a Plane': 3.5,
    'Just My Luck': 3.0,
    'The Night Listener': 4.5,
    'Superman Returns': 4.0,
    'You, Me and Dupree': 2.5},
  'Mick LaSalle': {
    'Lady in the Water': 3.0,
    'Snakes on a Plane': 4.0,
    'Just My Luck': 2.0,
    'Superman Returns': 3.0,
    'The Night Listener': 3.0,
    'You, Me and Dupree': 2.0},
  'Jack Matthews': {
    'Lady in the Water': 3.0,
    'Snakes on a Plane': 4.0,
    'The Night Listener': 3.0,
    'Superman Returns': 5.0,
    'You, Me and Dupree': 3.5},
  'Toby': {
    'Snakes on a Plane':4.5,
    'You, Me and Dupree':1.0,
    'Superman Returns':4.0},}

from math import sqrt

# Returns a distance-based similarity score for person1 and person2
def sim_distance(prefs,person1,person2):
  # Get the list of shared_items:
  # initiate the similarity array
  si={}
  # for each item in the preference array of person 1...
  for item in prefs[person1]:
    # ...if that item is in the preference array of person 2
    if item in prefs[person2]:
      # then let the similarity array, at index 'item,' equal 1.
      si[item]=1

  # if they have no ratings in common, return 0
```

```
  if len(si)==0: return 0

  # Add up the squares of all the differences
  sum_of_squares=sum([pow(prefs[person1][item]-prefs[person2][item],2)
                      for item in prefs[person1] if item in prefs[person2]])

  # return a value between 0 and 1, where 1 implies perfect correlation
  return 1/(1+sum_of_squares)

# Returns the Pearson correlation coefficient for p1 and p2
def sim_pearson(prefs,p1,p2):
  # Get the list of mutually rated items. Same as in Euclidian Dist.
  si={}
  for item in prefs[p1]:
    if item in prefs[p2]: si[item]=1

  # if they are no ratings in common, return 0
  if len(si)==0: return 0

  # Sum calculations
  n=len(si)

  # Sums of all the preferences
  sum1=sum([prefs[p1][it] for it in si])
  sum2=sum([prefs[p2][it] for it in si])

  # Sums of the squares
  sum1Sq=sum([pow(prefs[p1][it],2) for it in si])
  sum2Sq=sum([pow(prefs[p2][it],2) for it in si])

  # Sum of the products
  pSum=sum([prefs[p1][it]*prefs[p2][it] for it in si])

  # Calculate numerator and denominator for Pearson score
  num=pSum-(sum1*sum2/n)
  den=sqrt((sum1Sq-pow(sum1,2)/n)*(sum2Sq-pow(sum2,2)/n))
  # Correct for divide by zero
  if den==0: return 0

  # Get Pearson Score
  r=num/den

  # Return Pearsosn Score for function.
  return r

# Returns the best matches for person from the prefs dictionary.
# Number of results and similarity function are optional params.
def topMatches(prefs,person,n=5,similarity=sim_pearson):
  # create an array of all critics in comaprison to me
  scores=[(similarity(prefs,person,other),other)
                  # Making sure that there are no repeats.
                  for other in prefs if other!=person]
  # sort this array (indexed by similiarity) from 0-1
  scores.sort()
  # reverse the array so that more similiar critics are first
  scores.reverse()
  # return the first 0-n critics
  return scores[0:n]

# Gets recommendations for a person by using a weighted average
# of every other user's rankings
def getRecommendations(prefs,person,similarity=sim_pearson):
  # initiate an array for summed similarity (with value: movie)
  totals={}
```

```
  # initiate an array for the pow(sim,2) of each movie
  simSums={}
  # for every other critic in the preference array...
  for other in prefs:
    # ...don't compare me to myself, but...
    if other==person: continue
    # ...if not then set sim to our similarity.
    sim=similarity(prefs,person,other)

    # ignore scores of zero or lower
    if sim<=0: continue
    # and for each item in the prefs array (excluding me)
    for item in prefs[other]:

      # only score movies I haven't seen yet
      if item not in prefs[person] or prefs[person][item]==0:
        # Similarity * Score:
        # add the item to my array with starting value 0
        totals.setdefault(item,0)
        # add other people's recommendations
        totals[item]+=prefs[other][item]*sim
        # Sum of similarities
        simSums.setdefault(item,0)
        # likewise for simliarities
        simSums[item]+=sim

  # Create the normalized list
  rankings=[(total/simSums[item],item) for item,total in totals.items()]

  # Return the sorted list
  rankings.sort()
  rankings.reverse()
  return rankings

# flip the index so that movies are the index with critics and rating
# being the result.
def transformPrefs(prefs):
  # initiate a new result array
  result={}
  # for each person in pref...
  for person in prefs:
    # ...take each movie in their inventory and....
    for item in prefs[person]:
      # ...create an empty entry in result with the movie as the index
      result.setdefault(item,{})

      # Flip item and person and add it to the result array
      result[item][person]=prefs[person][item]
  # return your result and you're done!
  return result

# Create a dictionary of items showing which other items they are most
# similar to.
def calculateSimilarItems(prefs,n=10):
  # initiate the result array
  result={}
  # Invert the preference matrix to be item-centric from above function
  itemPrefs=transformPrefs(prefs)
  # start a counter
  c=0
  # for each item in our inverted matrix:
  for item in itemPrefs:
    # add one to the counter
    c+=1
```

```
    # if the remainder of c/100 == 0 then print out counter and length of matrix
    if c%100==0: print "%d / %d" % (c,len(itemPrefs))
    # Find the most similar items to this one (euclid Dist)
    scores=topMatches(itemPrefs,item,n=n,similarity=sim_distance)
    # take item entry in result and assign it it's similarity score
    result[item]=scores
  # return result matrix
  return result

# Create an array of items which are most similar to the movie of choice
# in order of similarity by ratings.
def getRecommendedItems(prefs,itemMatch,user):
  userRatings=prefs[user]
  scores={}
  totalSim={}
  # Loop over items rated by this user
  for (item,rating) in userRatings.items( ):

    # Loop over items similar to this one
    for (similarity,item2) in itemMatch[item]:

      # Ignore if this user has already rated this item
      if item2 in userRatings: continue
      # Weighted sum of rating times similarity
      scores.setdefault(item2,0)
      scores[item2]+=similarity*rating
      # Sum of all the similarities
      totalSim.setdefault(item2,0)
      totalSim[item2]+=similarity

  # Divide each total score by total weighting to get an average
  rankings=[(score/totalSim[item],item) for item,score in scores.items( )]

  # Return the rankings from highest to lowest
  rankings.sort( )
  rankings.reverse( )
  return rankings

def loadMovieLens(path='/home/dbt/Coding/Python/ProgrammingCollectiveIntelligence/chapter2/data'):
  # Get movie titles
  movies={}
  # for each line in u.times:
  for line in open(path+'/u.item'):
    # split the line by the pipes as an id, rating, and title
    (id,title)=line.split('|')[0:2]
    # using the rating id as an index, plug in the title
    movies[id]=title

  # Load data
  prefs={}
  # for each line in u.data
  for line in open(path+'/u.data'):
    (user,movieid,rating,ts)=line.split('\t')
    prefs.setdefault(user,{})
    prefs[user][movies[movieid]]=float(rating)
  return prefs

      ================================================================================================
                                        Ch2 - DeliciousRec
         ------------------------------------------------------------------------------------
from pydelicious import get_popular,get_userposts,get_urlposts
import time

def initializeUserDict(tag,count=5):
```

```
  user_dict={}
  # get the top count' popular posts
  for p1 in get_popular(tag=tag)[0:count]:
    # find all users who posted this
    for p2 in get_urlposts(p1['href']):
      user=p2['user']
      user_dict[user]={}
  return user_dict

def fillItems(user_dict):
  all_items={}
  # Find links posted by all users
  for user in user_dict:
    for i in range(3):
      try:
        posts=get_userposts(user)
        break
      except:
        print "Failed user "+user+", retrying"
        time.sleep(4)
    for post in posts:
      url=post['href']
      user_dict[user][url]=1.0
      all_items[url]=1

  # Fill in missing items with 0
  for ratings in user_dict.values():
    for item in all_items:
      if item not in ratings:
        ratings[item]=0.0
      ================================================================================================
                                          Ch3 - Clustering
         ------------------------------------------------------------------------------------
from PIL import Image,ImageDraw

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

from math import sqrt

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)

    # 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]

def printclust(clust,labels=None,n=0):
  # indent to make a hierarchy layout
  for i in range(n): print ' ',
  if clust.id<0:
    # negative id means that this is branch
    print '-'
  else:
```

```
    # positive id means that this is an endpoint
    if labels==None: print clust.id
    else: print labels[clust.id]

  # 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)

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)

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


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

  # 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')

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))

def rotatematrix(data):
  newdata=[]
  for i in range(len(data[0])):
    newrow=[data[j][i] for j in range(len(data))]
    newdata.append(newrow)
  return newdata

import random

def kcluster(rows,distance=pearson,k=4):
  # 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]
  for i in range(len(rows[0]))] for j in range(k)]

  lastmatches=None
  for t in range(100):
    print 'Iteration %d' % t
    bestmatches=[[] for i in range(k)]

    # Find which centroid is the closest for each row
    for j in range(len(rows)):
      row=rows[j]
      bestmatch=0
      for i in range(k):
        d=distance(clusters[i],row)
        if d<distance(clusters[bestmatch],row): bestmatch=i
      bestmatches[bestmatch].append(j)

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

    # 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

  return bestmatches

# a metric for finding the intersections of two data sets, given that
# the dataset is 0 and 1s.
def tanamoto(v1,v2):
  c1,c2,shr=0,0,0

  for i in range(len(v1)):
    if v1[i]!=0: c1+=1 # in v1
    if v2[i]!=0: c2+=1 # in v2
    if v1[i]!=0 and v2[i]!=0: shr+=1 # in both

  return 1.0-(float(shr)/(c1+c2-shr))
```

```
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]))]))

    # 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')
```

```
import feedparser
import re

# Returns title and dictionary of word counts for an RSS feed
def getwordcounts(url):
  # Parse the feed
  d=feedparser.parse(url)
  wc={}

  # Loop over all the entries
  for e in d.entries:
    if 'summary' in e: summary=e.summary
    else: summary=e.description

    # Extract a list of words
    words=getwords(e.title+' '+summary)
    for word in words:
      wc.setdefault(word,0)
      wc[word]+=1
  return d.feed.title,wc

def getwords(html):
  # Remove all the HTML tags
  txt=re.compile(r'<[^>]+>').sub('',html)

  # Split words by all non-alpha characters
  words=re.compile(r'[^A-Z^a-z]+').split(txt)

  # Convert to lowercase
  return [word.lower() for word in words if word!='']


apcount={}
wordcounts={}
feedlist=[line for line in file('feedlist.txt')]
for feedurl in feedlist:
  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 'Failed to parse feed %s' % feedurl

wordlist=[]
for w,bc in apcount.items():
  frac=float(bc)/len(feedlist)
  if frac>0.1 and frac<0.5:
    wordlist.append(w)

out=file('blogdata1.txt','w')
out.write('Blog')
for word in wordlist: out.write('\t%s' % word)
out.write('\n')
for blog,wc in wordcounts.items():
  print blog
  out.write(blog)
  for word in wordlist:
```

```
    if word in wc: out.write('\t%d' % wc[word])
    else: out.write('\t0')
  out.write('\n')
```

==========

Ch4 - Search Indexing and Optimization

```
import urllib2
from BeautifulSoup import *
from urlparse import urljoin
import sqlite3
import nn
mynet=nn.searchnet('nn.db')

# Create a list of words to ignore
ignorewords={'the':1,'of':1,'to':1,'and':1,'a':1,'in':1,'is':1,'it':1}

class crawler:
  # Initialize the crawler with the name of database
  def __init__(self,dbname):
    self.con=sqlite3.connect(dbname)

  def __del__(self):
    self.con.close()

  def dbcommit(self):
    self.con.commit()

  # Auxilliary function for getting an entry id and adding
  # it if it's not present
  def getentryid(self,table,field,value,createnew=True):
    cur=self.con.execute(
    "select rowid from %s where %s='%s'" % (table,field,value))
    res=cur.fetchone()
    if res==None:
      cur=self.con.execute(
      "insert into %s (%s) values ('%s')" % (table,field,value))
      return cur.lastrowid
    else:
      return res[0]

  # Index an individual page
  def addtoindex(self,url,soup):
    if self.isindexed(url): return
    print 'Indexing '+url

    # Get the individual words
    text=self.gettextonly(soup)
    words=self.separatewords(text)

    # Get the URL id
    urlid=self.getentryid('urllist','url',url)

    # Link each word to this url
    for i in range(len(words)):
      word=words[i]
      if word in ignorewords: continue
      wordid=self.getentryid('wordlist','word',word)
      self.con.execute("insert into wordlocation(urlid,wordid,location) values (%d,%d,%d)" %
(urlid,wordid,i))

  # Extract the text from an HTML page (no tags)
  def gettextonly(self,soup):
    v=soup.string
```

```
  if v==Null:
    c=soup.contents
    resulttext=''
    for t in c:
      subtext=self.gettextonly(t)
      resulttext+=subtext+'\n'
    return resulttext
  else:
    return v.strip()

# Seperate the words by any non-whitespace character
def separatewords(self,text):
  splitter=re.compile('\\W*')
  return [s.lower() for s in splitter.split(text) if s!='']


# Return true if this url is already indexed
def isindexed(self,url):
  u=self.con.execute \
    ("select rowid from urllist where url='%s'"%url).fetchone()
  if u!=None:
    # Check if it has actually been crawled
    v=self.con.execute(
    'select * from wordlocation where urlid=%d'%u[0]).fetchone()
    if v!=None: return True
  return False

# Add a link between two pages
def addlinkref(self,urlFrom,urlTo,linkText):
  words=self.separateWords(linkText)
  fromid=self.getentryid('urllist','url',urlFrom)
  toid=self.getentryid('urllist','url',urlTo)
  if fromid==toid: return
  cur=self.con.execute("insert into link(fromid,toid) values (%d,%d)" % (fromid,toid))
  linkid=cur.lastrowid
  for word in words:
    if word in ignorewords: continue
    wordid=self.getentryid('wordlist','word',word)
    self.con.execute("insert into linkwords(linkid,wordid) values (%d,%d)" % (linkid,wordid))

# Starting with a list of pages, do a breadth
# first search to the given depth, indexing pages
# as we go
def crawl(self,pages,depth=2):
  # for page i in the pages we have
  for i in range(depth):
    # initiate a newpages array
    newpages={}
    # for each page in pages
    for page in pages:
      # First
      try:
        # open the page
        c=urllib2.urlopen(page)
      except:
        # if not, tell us which page doesnt work
        print "Could not open %s" % page
        # and continue with the loop
        continue
      # Second
      try:
        # read the output with Beautiful Soup
        soup=BeautifulSoup(c.read())
        # add the souped page to our self array,
```

```
        # indexed by original page
        self.addtoindex(page,soup)

        # search for all links - from <a href>
        links=soup('a')
        # for each link in links
        for link in links:
          # if there is an href in the dictionary of links
          if ('href' in dict(link.attrs)):
            # join that page to the link
            url=urljoin(page,link['href'])
            # and look for the "'"  ----->?
            if url.find("'")!=-1: continue
            url=url.split('#')[0]  # remove location portion
            # if the url has an http and isn't in self
            if url[0:4]=='http' and not self.isindexed(url):
              #add it to newpages, indexed by url
              newpages[url]=1
            # get just the text of the link
            linkText=self.gettextonly(link)
            # add the link reference to the self array
            self.addlinkref(page,url,linkText)
        # add this to the database
        self.dbcommit()
      except:
        # print which page couldn't be indexed
        print "Could not parse page %s" % page
    # after running, newpages becomes pages and the process repeats
    pages=newpages

# Create the database tables
def createindextables(self):
  self.con.execute('create table urllist(url)')
  self.con.execute('create table wordlist(word)')
  self.con.execute('create table wordlocation(urlid,wordid,location)')
  self.con.execute('create table link(fromid integer,toid integer)')
  self.con.execute('create table linkwords(wordid,linkid)')
  self.con.execute('create index wordidx on wordlist(word)')
  self.con.execute('create index urlidx on urllist(url)')
  self.con.execute('create index wordurlidx on wordlocation(wordid)')
  self.con.execute('create index urltoidx on link(toid)')
  self.con.execute('create index urlfromidx on link(fromid)')
  self.dbcommit()

def calculatepagerank(self,iterations=20):
  # clear out the current page rank tables
  self.con.execute('drop table if exists pagerank')
  self.con.execute('create table pagerank(urlid primary key,score)')

  # initialize every url with a page rank of 1
  for (urlid,) in self.con.execute('select rowid from urllist'):
    self.con.execute('insert into pagerank(urlid,score) values (%d,1.0)' % urlid)
  self.dbcommit()

  for i in range(iterations):
    print "Iteration %d" % (i)
    for (urlid,) in self.con.execute('select rowid from urllist'):
      pr=0.15

      # Loop through all the pages that link to this one
      for (linker,) in self.con.execute(
      'select distinct fromid from link where toid=%d' % urlid):
        # Get the page rank of the linker
        linkingpr=self.con.execute(
```

```
          'select score from pagerank where urlid=%d' % linker).fetchone()[0]

          # Get the total number of links from the linker
          linkingcount=self.con.execute(
          'select count(*) from link where fromid=%d' % linker).fetchone()[0]
          pr+=0.85*(linkingpr/linkingcount)
        self.con.execute(
        'update pagerank set score=%f where urlid=%d' % (pr,urlid))
      self.dbcommit()
  # Initiatalize the crawler with the name of the database
  def __init__(self,dbname):
    pass

class searcher:
  def __init__(self,dbname):
    self.con=sqlite3.connect(dbname)

  def __del__(self):
    self.con.close()

  def getmatchrows(self,q):
    # Strings to build the query
    fieldlist='w0.urlid'
    tablelist=''
    clauselist=''
    wordids=[]

    # Split the words by spaces
    words=q.split(' ')
    tablenumber=0

    for word in words:
      # Get the word ID
      wordrow=self.con.execute(
      "select rowid from wordlist where word='%s'" % word).fetchone()
      if wordrow!=None:
        wordid=wordrow[0]
        wordids.append(wordid)
        if tablenumber>0:
          tablelist+=','
          clauselist+=' and '
          clauselist+='w%d.urlid=w%d.urlid and ' % (tablenumber-1,tablenumber)
        fieldlist+=',w%d.location' % tablenumber
        tablelist+='wordlocation w%d' % tablenumber
        clauselist+='w%d.wordid=%d' % (tablenumber,wordid)
        tablenumber+=1

    # Create the query from the separate parts
    fullquery='select %s from %s where %s' % (fieldlist,tablelist,clauselist)
    print fullquery
    cur=self.con.execute(fullquery)
    rows=[row for row in cur]

    return rows,wordids

  def getscoredlist(self,rows,wordids):
    totalscores=dict([(row[0],0) for row in rows])

    # This is where we'll put our scoring functions
    weights=[(1.0,self.locationscore(rows)),
             (1.0,self.frequencyscore(rows)),
             (1.0,self.pagerankscore(rows)),
             (1.0,self.linktextscore(rows,wordids)),
             (5.0,self.nnscore(rows,wordids))]
```

```
    for (weight,scores) in weights:
      for url in totalscores:
        totalscores[url]+=weight*scores[url]

    return totalscores

  def geturlname(self,id):
    return self.con.execute(
    "select url from urllist where rowid=%d" % id).fetchone()[0]

  def query(self,q):
    rows,wordids=self.getmatchrows(q)
    scores=self.getscoredlist(rows,wordids)
    rankedscores=[(score,url) for (url,score) in scores.items()]
    rankedscores.sort()
    rankedscores.reverse()
    for (score,urlid) in rankedscores[0:10]:
      print '%f\t%s' % (score,self.geturlname(urlid))
    return wordids,[r[1] for r in rankedscores[0:10]]

  def normalizescores(self,scores,smallIsBetter=0):
    vsmall=0.00001 # Avoid division by zero errors
    if smallIsBetter:
      minscore=min(scores.values())
      return dict([(u,float(minscore)/max(vsmall,l)) for (u,l) in scores.items()])
    else:
      maxscore=max(scores.values())
      if maxscore==0: maxscore=vsmall
      return dict([(u,float(c)/maxscore) for (u,c) in scores.items()])

  def frequencyscore(self,rows):
    counts=dict([(row[0],0) for row in rows])
    for row in rows: counts[row[0]]+=1
    return self.normalizescores(counts)

  def locationscore(self,rows):
    locations=dict([(row[0],1000000) for row in rows])
    for row in rows:
      loc=sum(row[1:])
      if loc<locations[row[0]]: locations[row[0]]=loc

    return self.normalizescores(locations,smallIsBetter=1)

  def distancescore(self,rows):
    # If there's only one word, everyone wins!
    if len(rows[0])<=2: return dict([(row[0],1.0) for row in rows])

    # Initialize the dictionary with large values
    mindistance=dict([(row[0],1000000) for row in rows])

    for row in rows:
      dist=sum([abs(row[i]-row[i-1]) for i in range(2,len(row))])
      if dist<mindistance[row[0]]: mindistance[row[0]]=dist
    return self.normalizescores(mindistance,smallIsBetter=1)

  def inboundlinkscore(self,rows):
    uniqueurls=dict([(row[0],1) for row in rows])
    inboundcount=dict([(u,self.con.execute('select count(*) from link where toid=%d' % u).fetchone()
[0]) for u in uniqueurls])
    return self.normalizescores(inboundcount)

  def linktextscore(self,rows,wordids):
    linkscores=dict([(row[0],0) for row in rows])
    for wordid in wordids:
```

```
      cur=self.con.execute('select link.fromid,link.toid from linkwords,link where wordid=%d and
linkwords.linkid=link.rowid' % wordid)
      for (fromid,toid) in cur:
        if toid in linkscores:
          pr=self.con.execute('select score from pagerank where urlid=%d' % fromid).fetchone()[0]
          linkscores[toid]+=pr
    maxscore=max(linkscores.values())
    normalizedscores=dict([(u,float(l)/maxscore) for (u,l) in linkscores.items()])
    return normalizedscores

  def pagerankscore(self,rows):
    pageranks=dict([(row[0],self.con.execute('select score from pagerank where urlid=%d' %
row[0]).fetchone()[0]) for row in rows])
    maxrank=max(pageranks.values())
    normalizedscores=dict([(u,float(l)/maxrank) for (u,l) in pageranks.items()])
    return normalizedscores

  def nnscore(self,rows,wordids):
    # Get unique URL IDs as an ordered list
    urlids=[urlid for urlid in dict([(row[0],1) for row in rows])]
    nnres=mynet.getresult(wordids,urlids)
    scores=dict([(urlids[i],nnres[i]) for i in range(len(urlids))])
    return self.normalizescores(scores)
```

Ch4 - Neural Networks

```
from math import tanh
import sqlite3

def dtanh(y):
    return 1.0-y*y

class searchnet:
    def __init__(self,dbname):
      self.con=sqlite3.connect(dbname)

    def __del__(self):
      self.con.close()

    def maketables(self):
      self.con.execute('create table hiddennode(create_key)')
      self.con.execute('create table wordhidden(fromid,toid,strength)')
      self.con.execute('create table hiddenurl(fromid,toid,strength)')
      self.con.commit()

    def getstrength(self,fromid,toid,layer):
      if layer==0: table='wordhidden'
      else: table='hiddenurl'
      res=self.con.execute('select strength from %s where fromid=%d and toid=%d' %
(table,fromid,toid)).fetchone()
      if res==None:
          # default strength is -.2: so that extra words have a slightly negative effect on the
activation level of a hidden node
          if layer==0: return -0.2
          # for links from the hidden layer to URLs, the mehod will return a default of 0
          if layer==1: return 0
      return res[0]

    # this is the code that trains the network.
    def setstrength(self,fromid,toid,layer,strength):
      if layer==0: table='wordhidden'
      else: table='hiddenurl'
```

```
      res=self.con.execute('select rowid from %s where fromid=%d and toid=%d' %
(table,fromid,toid)).fetchone()
      if res==None:
        self.con.execute('insert into %s (fromid,toid,strength) values (%d,%d,%f)' %
(table,fromid,toid,strength))
      else:
        rowid=res[0]
        self.con.execute('update %s set strength=%f where rowid=%d' % (table,strength,rowid))

    # This function createsdefault-weighted links between the words and the hidden node, and between
the query node and the URL results which are returned by this query:
    def generatehiddennode(self,wordids,urls):
      if len(wordids)>3: return None
      # Check if we already created a node for this set of words
      sorted_words=[str(id) for id in wordids]
      sorted_words.sort()
      createkey='_'.join(sorted_words)
      res=self.con.execute(
      "select rowid from hiddennode where create_key='%s'" % createkey).fetchone()

      # If not, create it
      if res==None:
        cur=self.con.execute(
        "insert into hiddennode (create_key) values ('%s')" % createkey)
        hiddenid=cur.lastrowid
        # Put in some default weights
        for wordid in wordids:
          self.setstrength(wordid,hiddenid,0,1.0/len(wordids))
        for urlid in urls:
          self.setstrength(hiddenid,urlid,1,0.1)
        self.con.commit()

    def getallhiddenids(self,wordids,urlids):
      l1={}
      for wordid in wordids:
        cur=self.con.execute(
        'select toid from wordhidden where fromid=%d' % wordid)
        for row in cur: l1[row[0]]=1
      for urlid in urlids:
        cur=self.con.execute(
        'select fromid from hiddenurl where toid=%d' % urlid)
        for row in cur: l1[row[0]]=1
      return l1.keys()

    def setupnetwork(self,wordids,urlids):
        # value lists
        self.wordids=wordids
        self.hiddenids=self.getallhiddenids(wordids,urlids)
        self.urlids=urlids

        # node outputs
        self.ai = [1.0]*len(self.wordids)
        self.ah = [1.0]*len(self.hiddenids)
        self.ao = [1.0]*len(self.urlids)

        # create weights matrix
        self.wi = [[self.getstrength(wordid,hiddenid,0)
                    for hiddenid in self.hiddenids]
                   for wordid in self.wordids]
        self.wo = [[self.getstrength(hiddenid,urlid,1)
                    for urlid in self.urlids]
                   for hiddenid in self.hiddenids]

    def feedforward(self):
```

```
        # the only inputs are the query words
        for i in range(len(self.wordids)):
            self.ai[i] = 1.0

        # hidden activations
        for j in range(len(self.hiddenids)):
            sum = 0.0
            for i in range(len(self.wordids)):
                sum = sum + self.ai[i] * self.wi[i][j]
            self.ah[j] = tanh(sum)

        # output activations
        for k in range(len(self.urlids)):
            sum = 0.0
            for j in range(len(self.hiddenids)):
                sum = sum + self.ah[j] * self.wo[j][k]
            self.ao[k] = tanh(sum)

        return self.ao[:]

    def getresult(self,wordids,urlids):
      self.setupnetwork(wordids,urlids)
      return self.feedforward()

    # goes backward though the output->hiddenlayer to see what the weight should be, go to the hidden
node and change the input node, then change the output node.
    def backPropagate(self, targets, N=0.5):
        # calculate errors for output
        output_deltas = [0.0] * len(self.urlids)
        for k in range(len(self.urlids)):
            error = targets[k]-self.ao[k]
            output_deltas[k] = dtanh(self.ao[k]) * error

        # calculate errors for hidden layer
        hidden_deltas = [0.0] * len(self.hiddenids)
        for j in range(len(self.hiddenids)):
            error = 0.0
            for k in range(len(self.urlids)):
                error = error + output_deltas[k]*self.wo[j][k]
            hidden_deltas[j] = dtanh(self.ah[j]) * error

        # update output weights
        for j in range(len(self.hiddenids)):
            for k in range(len(self.urlids)):
                change = output_deltas[k]*self.ah[j]
                self.wo[j][k] = self.wo[j][k] + N*change

        # update input weights
        for i in range(len(self.wordids)):
            for j in range(len(self.hiddenids)):
                change = hidden_deltas[j]*self.ai[i]
                self.wi[i][j] = self.wi[i][j] + N*change

    # Sets up the network, runs feedforward, then the back progogation.
    def trainquery(self,wordids,urlids,selectedurl):
      # generate a hidden node if necessary
      self.generatehiddennode(wordids,urlids)

      self.setupnetwork(wordids,urlids)
      self.feedforward()
      targets=[0.0]*len(urlids)
      targets[urlids.index(selectedurl)]=1.0
      error = self.backPropagate(targets)
      self.updatedatabase()
```

```
    # Saves the results that you find to the database.
    def updatedatabase(self):
      # set them to database values
      for i in range(len(self.wordids)):
          for j in range(len(self.hiddenids)):
              self.setstrength(self.wordids[i],self. hiddenids[j],0,self.wi[i][j])
      for j in range(len(self.hiddenids)):
          for k in range(len(self.urlids)):
              self.setstrength(self.hiddenids[j],self.urlids[k],1,self.wo[j][k])
      self.con.commit()
```

==========

Ch5 - Classifiers One

```
import random
import math

# The dorms, each of which has two available spaces
dorms=['Zeus','Athena','Hercules','Bacchus','Pluto']

# People, along with their first and second choices
prefs=[('Toby', ('Bacchus', 'Hercules')),
       ('Steve', ('Zeus', 'Pluto')),
       ('Karen', ('Athena', 'Zeus')),
       ('Sarah', ('Zeus', 'Pluto')),
       ('Dave', ('Athena', 'Bacchus')),
       ('Jeff', ('Hercules', 'Pluto')),
       ('Fred', ('Pluto', 'Athena')),
       ('Suzie', ('Bacchus', 'Hercules')),
       ('Laura', ('Bacchus', 'Hercules')),
       ('James', ('Hercules', 'Athena'))]

# [(0,9),(0,8),(0,7),(0,6),...,(0,0)]
domain=[(0,(len(dorms)*2)-i-1) for i in range(0,len(dorms)*2)]

def printsolution(vec):
  slots=[]
  # Create two slots for each dorm
  for i in range(len(dorms)): slots+=[i,i]

  # Loop over each students assignment
  for i in range(len(vec)):
    x=int(vec[i])

    # Choose the slot from the remaining ones
    dorm=dorms[slots[x]]
    # Show the student and assigned dorm
    print prefs[i][0],dorm
    # Remove this slot
    del slots[x]

def dormcost(vec):
  cost=0
  # Create list a of slots
  slots=[0,0,1,1,2,2,3,3,4,4]

  # Loop over each student
  for i in range(len(vec)):
    x=int(vec[i])
    dorm=dorms[slots[x]]
    pref=prefs[i][1]
    # First choice costs 0, second choice costs 1
    if pref[0]==dorm: cost+=0
```

```
    elif pref[1]==dorm: cost+=1
    else: cost+=3
    # Not on the list costs 3

    # Remove selected slot
    del slots[x]

  return cost
```

Ch5 – Optimization

```
import time
import random
import math

people = [('Seymour','BOS'),
          ('Franny','DAL'),
          ('Zooey','CAK'),
          ('Walt','MIA'),
          ('Buddy','ORD'),
          ('Les','OMA')]
# Laguardia
destination='LGA'

flights={}
#
for line in file('schedule.txt'):
  origin,dest,depart,arrive,price=line.strip().split(',')
  flights.setdefault((origin,dest),[])

  # Add details to the list of possible flights
  flights[(origin,dest)].append((depart,arrive,int(price)))

def getminutes(t):
  x=time.strptime(t,'%H:%M')
  return x[3]*60+x[4]

# Prints a line containing each person's name, origin, departure time,
# arrival time, and price for outgoing and return flights.
def printschedule(r):
  for d in range(len(r)/2):
    name=people[d][0]
    origin=people[d][1]
    out=flights[(origin,destination)][int(r[d])]
    ret=flights[(destination,origin)][int(r[d+1])]
    print '%10s%10s %5s-%5s $%3s %5s-%5s $%3s' % (name,origin,
                                                  out[0],out[1],out[2],
                                                  ret[0],ret[1],ret[2])

def schedulecost(sol):
  totalprice=0
  latestarrival=0
  earliestdep=24*60

  for d in range(len(sol)/2):
    # Get the inbound and outbound flights
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[d])]
    returnf=flights[(destination,origin)][int(sol[d+1])]

    # Total price is the price of all outbound and return flights
    totalprice+=outbound[2]
    totalprice+=returnf[2]
```

```
    # Track the latest arrival and earliest departure
    if latestarrival<getminutes(outbound[1]): latestarrival=getminutes(outbound[1])
    if earliestdep>getminutes(returnf[0]): earliestdep=getminutes(returnf[0])

  # Every person must wait at the airport until the latest person arrives.
  # They also must arrive at the same time and wait for their flights.
  totalwait=0
  for d in range(len(sol)/2):
    origin=people[d][1]
    outbound=flights[(origin,destination)][int(sol[d])]
    returnf=flights[(destination,origin)][int(sol[d+1])]
    totalwait+=latestarrival-getminutes(outbound[1])
    totalwait+=getminutes(returnf[0])-earliestdep

  # Does this solution require an extra day of car rental? That'll be $50!
  if latestarrival>earliestdep: totalprice+=50

  return totalprice+totalwait

def randomoptimize(domain,costf):
  best=999999999
  bestr=None
  for i in range(0,1000):
    # Create a random solution
    r=[float(random.randint(domain[i][0],domain[i][1]))
       for i in range(len(domain))]

    # Get the cost
    cost=costf(r)

    # Compare it to the best one so far
    if cost<best:
      best=cost
      bestr=r
  return r

def hillclimb(domain,costf):
  # Create a random solution
  sol=[random.randint(domain[i][0],domain[i][1])
      for i in range(len(domain))]
  # Main loop
  while 1:
    # Create list of neighboring solutions
    neighbors=[]

    for j in range(len(domain)):
      # One away in each direction
      if sol[j]>domain[j][0]:
        neighbors.append(sol[0:j]+[sol[j]+1]+sol[j+1:])
      if sol[j]<domain[j][1]:
        neighbors.append(sol[0:j]+[sol[j]-1]+sol[j+1:])

    # See what the best solution amongst the neighbors is
    current=costf(sol)
    best=current
    for j in range(len(neighbors)):
      cost=costf(neighbors[j])
      if cost<best:
        best=cost
        sol=neighbors[j]

    # If there's no improvement, then we've reached the top
    if best==current:
      break
```

```
  return sol

def annealingoptimize(domain,costf,T=10000.0,cool=0.95,step=1):
  # Initialize the values randomly
  vec=[float(random.randint(domain[i][0],domain[i][1]))
       for i in range(len(domain))]

  while T>0.1:
    # Choose one of the indices
    i=random.randint(0,len(domain)-1)

    # Choose a direction to change it
    dir=random.randint(-step,step)

    # Create a new list with one of the values changed
    vecb=vec[:]
    vecb[i]+=dir
    if vecb[i]<domain[i][0]: vecb[i]=domain[i][0]
    elif vecb[i]>domain[i][1]: vecb[i]=domain[i][1]

    # Calculate the current cost and the new cost
    ea=costf(vec)
    eb=costf(vecb)
    p=pow(math.e,(-eb-ea)/T)

    # Is it better, or does it make the probability
    # cutoff?
    if (eb<ea or random.random()<p):
      vec=vecb

    # Decrease the temperature
    T=T*cool
  return vec

def geneticoptimize(domain,costf,popsize=50,step=1,
                    mutprob=0.2,elite=0.2,maxiter=100):
  # Mutation Operation
  def mutate(vec):
    i=random.randint(0,len(domain)-1)
    if random.random()<0.5 and vec[i]>domain[i][0]:
      return vec[0:i]+[vec[i]-step]+vec[i+1:]
    elif vec[i]<domain[i][1]:
      return vec[0:i]+[vec[i]+step]+vec[i+1:]

  # Crossover Operation
  def crossover(r1,r2):
    i=random.randint(1,len(domain)-2)
    return r1[0:i]+r2[i:]

  # Build the initial population
  pop=[]
  for i in range(popsize):
    vec=[random.randint(domain[i][0],domain[i][1])
         for i in range(len(domain))]
    pop.append(vec)

  # How many winners from each generation?
  topelite=int(elite*popsize)

  # Main loop
  for i in range(maxiter):
    scores=[(costf(v),v) for v in pop]
    scores.sort()
    ranked=[v for (s,v) in scores]
```

```
    # Start with the pure winners
    pop=ranked[0:topelite]

    # Add mutated and bred forms of the winners
    while len(pop)<popsize:
      if random.random()<mutprob:

        # Mutation
        c=random.randint(0,topelite)
        pop.append(mutate(ranked[c]))
      else:

        # Crossover
        c1=random.randint(0,topelite)
        c2=random.randint(0,topelite)
        pop.append(crossover(ranked[c1],ranked[c2]))

    # Print current best score
    print scores[0][0]

  return scores[0][1]
```

==========

Ch5 - Document Classifies

```
from sqlite3 as sqlite
import re
import math

# breaks up the text into words dividing the text on any character that
# isn't a letter. This leaves only actual words, converted to lowercase
def getwords(doc):
  splitter=re.compile('\\W*')
#   print doc
  # Split the words by non-alpha characters
  words=[s.lower() for s in splitter.split(doc)
         if len(s)>2 and len(s)<20]

  # Return the unique set of words only
  return dict([(w,1) for w in words])

class classifier:
  def __init__(self,getfeatures,filename=None):
    # Counts of feature/category combinations (feature count)
    self.fc={}
    # Counts of documents in each category (category count)
    self.cc={}
    self.getfeatures=getfeatures

  def setdb(self,dbfile):
    self.con=sqlite.connect(dbfile)
    self.con.execute('create table if not exists fc(feature,category,count)')
    self.con.execute('create table if not exists cc(category,count)')

  #Increase the count of a feature/category pair
  def incf(self,f,cat):
    self.fc.setdefault(f,{})
    self.fc[f].setdefault(cat,0)
    self.fc[f][cat]+=1
    count=self.fcount(f,cat)
    if count==0:
      self.con.execute("insert into fc values ('%s','%s',1)"
                       % (f,cat))
```

```
  else:
    self.con.execute(
      "update fc set count=%d where feature='%s' and category='%s'"
      % (count+1,f,cat))

# Increase the count of a category
def incc(self,cat):
  self.cc.setdefault(cat,0)
  self.cc[cat]+=1
  count=self.catcount(cat)
  if count==0:
    self.con.execute("insert into cc values ('%s',1)" % (cat))
  else:
    self.con.execute("update cc set count=%d where category='%s'"
                     % (count+1,cat))

# The number of times a feature has appeared in a category
def fcount(self,f,cat):
  if f in self.fc and cat in self.fc[f]:
    return float(self.fc[f][cat])
  return 0
  res=self.con.execute(
    'select count from fc where feature="%s" and category="%s"'
    %(f,cat)).fetchone()
  if res==None: return 0
  else: return float(res[0])

# The number of items in a category
def catcount(self,cat):
  if cat in self.cc:
    return float(self.cc[cat])
  return 0
  res=self.con.execute('select count from cc where category="%s"'
                       %(cat)).fetchone()
  if res==None: return 0
  else: return float(res[0])

# The total number of items
def totalcount(self):
  return sum(self.cc.values())
  res=self.con.execute('select sum(count) from cc').fetchone();
  if res==None: return 0
  return res[0]

# The list of all categories
def categories(self):
  return self.cc.keys()
  cur=self.con.execute('select category from cc');
  return [d[0] for d in cur]

def train(self,item,cat):
  features=self.getfeatures(item)
  # Increment the count for every feature with this category
  for f in features:
    self.incf(f,cat)

  # Increment the count for this category
  self.incc(cat)
  self.con.commit()

def fprob(self,f,cat):
  if self.catcount(cat)==0: return 0

  # The total number of times this feature appeared in this
```

```
    # category divided by the total number of items in this category
    return self.fcount(f,cat)/self.catcount(cat)

  def weightedprob(self,f,cat,prf,weight=1.0,ap=0.5):
    # Calculate current probability
    basicprob=prf(f,cat)

    # Count the number of times this feature has appeared in
    # all categories
    totals=sum([self.fcount(f,c) for c in self.categories()])

    # Calculate the weighted average
    bp=((weight*ap)+(totals*basicprob))/(weight+totals)
    return bp

class naivebayes(classifier):

  def __init__(self,getfeatures):
    classifier.__init__(self,getfeatures)
    self.thresholds={}

  def docprob(self,item,cat):
    features=self.getfeatures(item)

    # Multiply the probabilities of all the features together
    p=1
    for f in features: p*=self.weightedprob(f,cat,self.fprob)
    return p

  def prob(self,item,cat):
    catprob=self.catcount(cat)/self.totalcount()
    docprob=self.docprob(item,cat)
    return docprob*catprob

  def setthreshold(self,cat,t):
    self.thresholds[cat]=t

  def getthreshold(self,cat):
    if cat not in self.thresholds: return 1.0
    return self.thresholds[cat]

  # calculates the probability for each category, determines which
  # one is the largest, and whether it exceeds the next largest by
  # more than its threshold. If no categories accomplish this, return
  # default values.
  def classify(self,item,default=None):
    probs={}
    # Find the category with the highest probability
    max=0.0
    for cat in self.categories():
      probs[cat]=self.prob(item,cat)
      if probs[cat]>max:
        max=probs[cat]
        best=cat

    # Make sure the probability exceeds threshold*next best
    for cat in probs:
      if cat==best: continue
      if probs[cat]*self.getthreshold(best)>probs[best]: return default
    return best

class fisherclassifier(classifier):
  # Essentially cprob = clf/(clf+nclf)
  def cprob(self,f,cat):
```

```
    # The frequency of this feature in this category:
    # clf=Pr(feature|category) for this category
    clf=self.fprob(f,cat)
    if clf==0: return 0

    # The frequency of this feature in all the categories:
    # freqsum = Sum of Pr(feature|category) for all the categories
    freqsum=sum([self.fprob(f,c) for c in self.categories()])

    # The probability is the frequency in this category divided by
    # the overall frequency
    p=clf/(freqsum)

    return p

  def fisherprob(self,item,cat):
    # Multiply all the probabilities together
    p=1
    features=self.getfeatures(item)
    for f in features:
      p*=(self.weightedprob(f,cat,self.cprob))

    # Take the natural log and multiply by -2
    fscore=-2*math.log(p)

    # Use the inverse chi2 function to get a probability
    return self.invchi2(fscore,len(features)*2)
  def invchi2(self,chi, df):
    m = chi / 2.0
    sum = term = math.exp(-m)
    for i in range(1, df//2):
        term *= m / i
        sum += term
    return min(sum, 1.0)
  def __init__(self,getfeatures):
    classifier.__init__(self,getfeatures)
    self.minimums={}

  def setminimum(self,cat,min):
    self.minimums[cat]=min

  def getminimum(self,cat):
    if cat not in self.minimums: return 0
    return self.minimums[cat]
  def classify(self,item,default=None):
    # Loop through looking for the best result
    best=default
    max=0.0
    for c in self.categories():
      p=self.fisherprob(item,c)
      # Make sure it exceeds its minimum
      if p>self.getminimum(c) and p>max:
        best=c
        max=p
    return best


def sampletrain(cl):
  cl.train('Nobody owns the water.','good')
  cl.train('the quick rabbit jumps fences','good')
  cl.train('buy pharmaceuticals now','bad')
  cl.train('make quick money at the online casino','bad')
  cl.train('the quick brown fox jumps','good')
```

```
import feedparser
import re

# Takes a filename of URL of a blog feed and classifies the entries
def read(feed,classifier):
  # Get feed entries and loop over them
  f=feedparser.parse(feed)
  for entry in f['entries']:
    print
    print '-----'
    # Print the contents of the entry
    print 'Title:     '+entry['title'].encode('utf-8')
    print 'Publisher: '+entry['publisher'].encode('utf-8')
    print
    print entry['summary'].encode('utf-8')


    # Combine all the text to create one item for the classifier
    fulltext='%s\n%s\n%s' % (entry['title'],entry['publisher'],entry['summary'])

    # Print the best guess at the current category
    print 'Guess: '+str(classifier.classify(entry))

    # Ask the user to specify the correct category and train on that
    cl=raw_input('Enter category: ')
    classifier.train(entry,cl)


def entryfeatures(entry):
  splitter=re.compile('\\W*')
  f={}

  # Extract the title words and annotate
  titlewords=[s.lower() for s in splitter.split(entry['title'])
          if len(s)>2 and len(s)<20]
  for w in titlewords: f['Title:'+w]=1

  # Extract the summary words
  summarywords=[s.lower() for s in splitter.split(entry['summary'])
          if len(s)>2 and len(s)<20]

  # Count uppercase words
  uc=0
  for i in range(len(summarywords)):
    w=summarywords[i]
    f[w]=1
    if w.isupper(): uc+=1

    # Get word pairs in summary as features
    if i<len(summarywords)-1:
      twowords=' '.join(summarywords[i:i+1])
      f[twowords]=1

  # Keep creator and publisher whole
  f['Publisher:'+entry['publisher']]=1

  # UPPERCASE is a virtual word flagging too much shouting
  if float(uc)/len(summarywords)>0.3: f['UPPERCASE']=1

  return f
```

```
      ================================================================================================
                                        Ch7 - Feed Filters
          ----------------------------------------------------------------------------------------
my_data=[['slashdot','USA','yes',18,'None'],
         ['google','France','yes',23,'Premium'],
         ['digg','USA','yes',24,'Basic'],
         ['kiwitobes','France','yes',23,'Basic'],
         ['google','UK','no',21,'Premium'],
         ['(direct)','New Zealand','no',12,'None'],
         ['(direct)','UK','no',21,'Basic'],
         ['google','USA','no',24,'Premium'],
         ['slashdot','France','yes',19,'None'],
         ['digg','USA','no',18,'None'],
         ['google','UK','no',18,'None'],
         ['kiwitobes','UK','no',19,'None'],
         ['digg','New Zealand','yes',12,'Basic'],
         ['slashdot','UK','no',21,'None'],
         ['google','UK','yes',18,'Basic'],
         ['kiwitobes','France','yes',19,'Basic']]

class decisionnode:
  def __init__(self,col=-1,value=None,results=None,tb=None,fb=None):
    self.col=col
    self.value=value
    self.results=results
    self.tb=tb
    self.fb=fb

# Divides a set on a specific column. Can handle numeric
# or nominal values
def divideset(rows,column,value):
   # Make a function that tells us if a row is in
   # the first group (true) or the second group (false)
   split_function=None
   if isinstance(value,int) or isinstance(value,float):
      split_function=lambda row:row[column]>=value
   else:
      split_function=lambda row:row[column]==value

   # Divide the rows into two sets and return them
   set1=[row for row in rows if split_function(row)]
   set2=[row for row in rows if not split_function(row)]
   return (set1,set2)


# Create counts of possible results (the last column of
# each row is the result)
def uniquecounts(rows):
   results={}
   for row in rows:
      # The result is the last column
      r=row[len(row)-1]
      if r not in results: results[r]=0
      results[r]+=1
   return results

# Probability that a randomly placed item will
# be in the wrong category
def giniimpurity(rows):
  total=len(rows)
  counts=uniquecounts(rows)
  imp=0
  for k1 in counts:
```

```
    p1=float(counts[k1])/total
    for k2 in counts:
      if k1==k2: continue
      p2=float(counts[k2])/total
      imp+=p1*p2
  return imp

# Entropy is the sum of p(x)log(p(x)) across all
# the different possible results
def entropy(rows):
   from math import log
   log2=lambda x:log(x)/log(2)
   results=uniquecounts(rows)
   # Now calculate the entropy
   ent=0.0
   for r in results.keys():
      p=float(results[r])/len(rows)
      ent=ent-p*log2(p)
   return ent




def printtree(tree,indent=''):
   # Is this a leaf node?
   if tree.results!=None:
      print str(tree.results)
   else:
      # Print the criteria
      print str(tree.col)+':'+str(tree.value)+'? '

      # Print the branches
      print indent+'T->',
      printtree(tree.tb,indent+'  ')
      print indent+'F->',
      printtree(tree.fb,indent+'  ')


def getwidth(tree):
  if tree.tb==None and tree.fb==None: return 1
  return getwidth(tree.tb)+getwidth(tree.fb)

def getdepth(tree):
  if tree.tb==None and tree.fb==None: return 0
  return max(getdepth(tree.tb),getdepth(tree.fb))+1


from PIL import Image,ImageDraw

def drawtree(tree,jpeg='tree.jpg'):
  w=getwidth(tree)*100
  h=getdepth(tree)*100+120

  img=Image.new('RGB',(w,h),(255,255,255))
  draw=ImageDraw.Draw(img)

  drawnode(draw,tree,w/2,20)
  img.save(jpeg,'JPEG')

def drawnode(draw,tree,x,y):
  if tree.results==None:
    # Get the width of each branch
    w1=getwidth(tree.fb)*100
    w2=getwidth(tree.tb)*100
```

```
    # Determine the total space required by this node
    left=x-(w1+w2)/2
    right=x+(w1+w2)/2

    # Draw the condition string
    draw.text((x-20,y-10),str(tree.col)+':'+str(tree.value),(0,0,0))

    # Draw links to the branches
    draw.line((x,y,left+w1/2,y+100),fill=(255,0,0))
    draw.line((x,y,right-w2/2,y+100),fill=(255,0,0))

    # Draw the branch nodes
    drawnode(draw,tree.fb,left+w1/2,y+100)
    drawnode(draw,tree.tb,right-w2/2,y+100)
  else:
    txt=' \n'.join(['%s:%d'%v for v in tree.results.items()])
    draw.text((x-20,y),txt,(0,0,0))


def classify(observation,tree):
  if tree.results!=None:
    return tree.results
  else:
    v=observation[tree.col]
    branch=None
    if isinstance(v,int) or isinstance(v,float):
      if v>=tree.value: branch=tree.tb
      else: branch=tree.fb
    else:
      if v==tree.value: branch=tree.tb
      else: branch=tree.fb
    return classify(observation,branch)

def prune(tree,mingain):
  # If the branches aren't leaves, then prune them
  if tree.tb.results==None:
    prune(tree.tb,mingain)
  if tree.fb.results==None:
    prune(tree.fb,mingain)

  # If both the subbranches are now leaves, see if they
  # should merged
  if tree.tb.results!=None and tree.fb.results!=None:
    # Build a combined dataset
    tb,fb=[],[]
    for v,c in tree.tb.results.items():
      tb+=[[v]]*c
    for v,c in tree.fb.results.items():
      fb+=[[v]]*c

    # Test the reduction in entropy
    delta=entropy(tb+fb)-(entropy(tb)+entropy(fb)/2)

    if delta<mingain:
      # Merge the branches
      tree.tb,tree.fb=None,None
      tree.results=uniquecounts(tb+fb)

def mdclassify(observation,tree):
  if tree.results!=None:
    return tree.results
  else:
    v=observation[tree.col]
```

```
    if v==None:
      tr,fr=mdclassify(observation,tree.tb),mdclassify(observation,tree.fb)
      tcount=sum(tr.values())
      fcount=sum(fr.values())
      tw=float(tcount)/(tcount+fcount)
      fw=float(fcount)/(tcount+fcount)
      result={}
      for k,v in tr.items(): result[k]=v*tw
      for k,v in fr.items(): result[k]=v*fw
      return result
    else:
      if isinstance(v,int) or isinstance(v,float):
        if v>=tree.value: branch=tree.tb
        else: branch=tree.fb
      else:
        if v==tree.value: branch=tree.tb
        else: branch=tree.fb
      return mdclassify(observation,branch)

def variance(rows):
  if len(rows)==0: return 0
  data=[float(row[len(row)-1]) for row in rows]
  mean=sum(data)/len(data)
  variance=sum([(d-mean)**2 for d in data])/len(data)
  return variance

def buildtree(rows,scoref=entropy):
  if len(rows)==0: return decisionnode()
  current_score=scoref(rows)

  # Set up some variables to track the best criteria
  best_gain=0.0
  best_criteria=None
  best_sets=None

  column_count=len(rows[0])-1
  for col in range(0,column_count):
    # Generate the list of different values in
    # this column
    column_values={}
    for row in rows:
       column_values[row[col]]=1
    # Now try dividing the rows up for each value
    # in this column
    for value in column_values.keys():
      (set1,set2)=divideset(rows,col,value)

      # Information gain
      p=float(len(set1))/len(rows)
      gain=current_score-p*scoref(set1)-(1-p)*scoref(set2)
      if gain>best_gain and len(set1)>0 and len(set2)>0:
        best_gain=gain
        best_criteria=(col,value)
        best_sets=(set1,set2)
  # Create the sub branches
  if best_gain>0:
    trueBranch=buildtree(best_sets[0])
    falseBranch=buildtree(best_sets[1])
    return decisionnode(col=best_criteria[0],value=best_criteria[1],
                        tb=trueBranch,fb=falseBranch)
  else:
    return decisionnode(results=uniquecounts(rows))
```