Soumya (U00864145)

Ram somesh (U00873934)

Feature extraction:

```
from __future__ import print_function
import util
import sys #File IO
import re #Regex
import nltk
import collections
from nltk.tokenize import RegexpTokenizer
import codecs
import string
import os
import math

class Posting:
    def __init__(self, docID):
        self.docID = docID
        self.positions = 1

    def append(self, pos):
        self.positions = pos

    def sort(self):
        ''' sort positions'''
        self.positions.sort()

    def merge(self, positions):
        self.positions.extend(positions)

    def term_freq(self):
        ''' return the term frequency in the document'''
        #ToDo
            tf=len(self.positions)
            return tf


class IndexItem:
    def __init__(self, term):
        self.term = term
        self.posting = {} #postings are stored in a python dict for easier index building
        self.sorted_postings= [] # may sort them by docID for easier query processing

    def add(self, docid, pos):
        ''' add a posting'''
        if not self.posting.has_key(docid):
            self.posting[docid] = Posting(docid)
        self.posting[docid].append(pos)
```

```
    def sort(self):
        ''' sort by document ID for more efficient merging. For each document also sort the positions'''
        # ToDo
            for docid in self.posting.keys():
                self.posting[docid].sort()
                self.sorted_postings.extend(self.posting[docid].positions)

class InvertedIndex:

    def __init__(self):
        self.items = {} # list of IndexItems
        self.nDocs = 0  # the number of indexed documents
        self.endmost={}

    def tokenization(body):
        body = re.sub("[^a-z0-9]+", " ", body)
        tokens = nltk.tokenize.word_tokenize(body)
        return tokens

    def indexDoc(self, subdir,doc): # indexing a Document object
        # ToDo: indexing only title and body; use some functions defined in util.py
        # (1) convert to lower cases,
        # (2) remove stopwords,
        # (3) stemming
            #f = open(os.path.join(subdir, doc))
            b=[] #created a list for storing a line of the document body everytime
            numOfLines=0    # number of lines in document body
            #print(subdir)
        #print(doc)
            f = open(os.path.join(subdir, doc))  #for the given path, the code will iterate within the whole
directory
            #print(file)

            #parsing the files
            for line in f:
           if 'Subject:' in line:    # if there exists 'Subject'in the line
              l=line[12:].strip()   # the content after subject: is stored after stripping and splitting
              subLn=l.split(' ')
            root=[]  # empty list to store root words of subject content
              for p in subLn:
                 root.append(re.sub(r"[^a-zA-Z]+"," ",p))  # after reducing to root words, appended to list
              for e in root:
                        b.append(e.lower().strip()) # converted to lower, appened to b[]
           if 'Lines:'==str(line[0:6]): # if lines: is in starting of line
            if line[8].isspace():
                numOfLines=int(line[7:8])  # content after that is stored (line:x)
              else:
                if line[9].isspace():
                   numOfLines=int(line[7:9])  # (line:xx)

                else:
```

```
            numOfLines=int(line[7:10]) # (line:xxx)

    #print(numOfLines)
    fileReadingBackwards=reversed(open(os.path.join(subdir, doc)).readlines())
    #reversed() is used to read the file backwards from last line, to get accurate lines:xx

finish=0
    for l in fileReadingBackwards:
   if numOfLines>=finish: # if it is not out of doc body lines
    root=l.replace('\\n', "").rstrip('\n\r') # lines of doc body is stripped
    reducedLn=re.sub(r"[^a-zA-Z]+"," ",root) # reduced to lowercase
    for lmn in reducedLn.lower().strip().split():
          b.append(lmn)  # appended to b[]
      finish=finish+1    # increments till it reaches last line from backwards

reduced=b
    reducedList=[] #created an empty dictionary
    #docBody=doc.body #body of the document is assigned to docBody
    #print(doc.docID)

    #reduced=nltk.word_tokenize(docBody) drops the non-alphanumeric terms


    #docBody = re.sub("[^a-z0-9]+", " ", docBody)
    #reduced=nltk.tokenize.word_tokenize(docBody)
    #i=[i.lower() for i in reduced]
    #all terms are coverted to lower cases
    #print (i)
    for words in reduced:
    #reduced terms are passed through stopwords and stemming in util
       if util.isStopWord(words):
          reducedList.append(util.stemming(words))
    #normalized terms are stored in reducedList Dictionary
    #print (reducedList)

posDict={} #Dictionary is created to store (terms, positions) in documents
    for position, word in enumerate(reducedList):
    #{term1:[pos1, pos3..], term2:[..], ..} dictionary is created
       if word in posDict.keys():
          posDict[word].append(position)
       else:
             posDict[word]=[position]
    #print(posDict)

wordKeys=[] #keys in posDict are stored
wordPos=[] #values in posDict are stored
wordKeys=posDict.keys()
#print(wordKeys)
for i in range(0,len(posDict),1):
    #appends the values of the corresponding key into wordPos
   wordPos.append(posDict[wordKeys[i]])
```

```
    #print(wordPos)
    #i=0

        for word in posDict:
        #to assign the positions of a word to its respective docID
       docdic={}
       #pos=wordPos[i]
       if word in self.items.keys():
            #if word is already in the item.keys(), then just add the docID
          pos=posDict[word]
          self.items[word].add(doc,len(pos))
                  #for every doc, add the docId and positions to the endmost dictionary
          for i in self.items[word].posting.keys():
                      docdic[i]=self.items[word].posting[i].positions
          self.endmost[word]=docdic
       else:
            #if word isnt present in items.keys(), create an index item, then add docId to positions
         iiObj=IndexItem(word)
         pos=posDict[word]
         iiObj.add(doc,len(pos))
         self.items[word] =iiObj
         for i in self.items[word].posting.keys():
              docdic[i]=self.items[word].posting[i].positions
         self.endmost[word]=docdic

           #i=i+1
        #print(self.endmost
    #for key in sorted(self.endmost):
        #srt=self.endmost.keys()
        #srt.sort()
        #for s in srt:
           #print((s, self.endmost[s]))
        return self.endmost # returns the inverted index with term and doc ID, listings


def sort(self):
    '' sort all posting lists by docID''
    #dict=collection.OrderedDict(sorted(dict()))
        #return dict

def find(self, term):
    return self.items[term]

def numofterms(term, toks):
        return toks.count(term.lower())

def tf(term, toks):
        return numofterms(term, toks) / float(len(toks))

def df(term, tokslist):
```

```
            x = 0
            for toks in tokslist:
                if numofterms(term, toks) > 0:
                        x += 1
            return x

    def idf(term, tokslist):
            return len(tokslist) / float(df(term, tokslist))

    def tfidf(term, toks, tokslist):
        return tf(term, toks) * idf(term, tokslist)

    def idf(self, term):
        ''' compute the inverted document frequency for a given term'''
        #ToDo: return the IDF of the term

    # more methods if needed


def test():
    ''' test your code thoroughly. put the testing cases here'''
    print ('Pass')

def indexingCranfield(path, featureDef, classDef, trainData):
    #ToDo: indexing the Cranfield dataset and save the index to a file
    # command line usage: "python index.py cran.all index_file"
    # the index is saved to index_file

    iiObj = InvertedIndex()

    for subdir, dirs, files in os.walk(path):   #arg 1
            for file in files:  # all the files in that path are taken as corpus
                    d=iiObj.indexDoc(subdir,file)

    #for the documents in corpus, all the terms are sorted

    iiObj.sort() #feature ids are sorted

    print ('Indexing is done')
    endmost=d
    id=range(0,len(endmost),1)  # range will be number of terms in inverted index
    f=open(featureDef,'w')          # arg 2, feature_definition_file.txt should be given
    index=0
    for i in endmost.keys():  # for all the terms in the inverted index
        f.write(str(id[index]))  # feature id
        f.write(',')
        f.write(str(i))         # feature
        f.write('\n')           # each line - each term
        index=index+1                   # feature id is incremented

    mapping = {}            # new dictionary for mapping the feature and its id
```

```
    mappingFile=open(featureDef)            # arg 2, feature_definition_file.txt should be given
    itr=0
    for line in endmost.keys():
        mapping[line] = itr # split line into parts
        itr=itr+1
    #print(len(endmost))
    #print(len(mapping))
    i=0
    for l in endmost.keys(): # mapping is done as term as key, id as value
        mapping[l]=i
        i=i+1

    #print(len(mapping))

    #empty lists are created for storing their corresponding related files in them
    c1 = []
    c2 = []
    c3 = []
    c4 = []
    c5 = []
    c6 = []

    with open(classDef) as mappingFile:         #arg 3, class_definition_file should be given
        for line in mappingFile:
            parts = line.split() # split line into parts

                # strip the class_definition_file, second part is appended to its corresponding list
                if parts[0].strip() == 'class-1':
                    c1.append(parts[1].strip())
                if parts[0].strip() == 'class-2':
                    c2.append(parts[1].strip())
                if parts[0].strip() == 'class-3':
                    c3.append(parts[1].strip())
                if parts[0].strip() == 'class-4':
                    c4.append(parts[1].strip())
                if parts[0].strip() == 'class-5':
                    c5.append(parts[1].strip())
                if parts[0].strip() == 'class-6':
                    c6.append(parts[1].strip())

    file_wt=open(trainData, 'w')        #arg 4, training_dataset_file should be given
    for subdir, dirs, files in os.walk(path):   #arg 1, path to mini_newsgroups is given
        for file in files:
                exp = subdir[37:].strip()         # the name of subdirectory in the path starts with 37th position
which may get altered with different paths in different systems

            #print(exp)
                #checked in c1, c2....lists, and prints its corresponding class number in training set
            if exp in c1:
                file_wt.write(str(1)+' ')
            if exp in c2:
```

```
            file_wt.write(str(2)+' ')
        if exp in c3:
            file_wt.write(str(3)+' ')
        if exp in c4:
            file_wt.write(str(4)+' ')
        if exp in c5:
            file_wt.write(str(5)+' ')
        if exp in c6:
            file_wt.write(str(6)+' ')

        #file_wt.write('\n')
        for e in endmost.keys():
            fl=endmost[e]# fl has endmost values
                df=len(fl)          # therefore, length of values for each feature will be its df

            if file in fl.keys():
                identity=mapping[e]             # feature ids are in identity
                file_wt.write(str(identity)+':')            # written to dataset
                tf=fl[str(file)]          # no of postings will be the tf
                    #tfNorm = 1 + math.log10(tf)
                idf=2000/df             # idf = num of docs in corpus / df
                    #idfNorm = math.log10(idf)
                tfidf=tf*idf #tfidf calculation
                file_wt.write(str(tfidf)+' ')    # written to dataset next to feature id as feature value

        file_wt.write('\n')             # each line is for each document in corpus

if __name__ == '__main__':
    #arguments are given for the command line
    path = sys.argv[1]
    featureDef = sys.argv[2]
    classDef = sys.argv[3]
    trainData = sys.argv[4]


    indexingCranfield(path, featureDef, classDef, trainData)
```

Classification.py:

```
from sklearn.datasets import load_svmlight_file
from sklearn.model_selection import cross_val_score

feature_vectors, targets = load_svmlight_file("/home/soumya/Desktop/training_dataset_file.txt")
# loads the training data with load_svmlight_file from sklearn

#print(feature_vectors)
#print(targets)

# accuracy calculation for ultinomialNB
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB()
scores = cross_val_score(clf, feature_vectors, targets, cv=5, scoring='f1_macro')
# cross validation with 5 folds
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

# accuracy calculation for BernoulliNB
from sklearn.naive_bayes import BernoulliNB
clf = BernoulliNB()
scores = cross_val_score(clf, feature_vectors, targets, cv=5, scoring='f1_macro')
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
#scores1 = cross_val_score(clf, feature_vectors, targets, cv=5, scoring='precision_macro')
#print(sorted(scores1.keys()))

# accuracy calculation for knn
from sklearn.neighbors  import KNeighborsClassifier
clf = KNeighborsClassifier()
scores = cross_val_score(clf, feature_vectors, targets, cv=5, scoring='f1_macro')
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))

## accuracy calculation for svm
from sklearn.svm  import SVC
clf = SVC()
scores = cross_val_score(clf, feature_vectors, targets, cv=5, scoring='f1_macro')
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

feature_selection:

```
# Part 3: Feature Selection
# Import required libraries

import matplotlib.pyplot as p
import sklearn.metrics as metrics
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import BernoulliNB
from sklearn.svm import SVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors  import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, mutual_info_classif
from sklearn.datasets import load_svmlight_file

feature_vectors, targets = load_svmlight_file("/home/soumya/Desktop/training_dataset_file.txt")



# Select Kbest - Chi-squared
nbMetric = BernoulliNB()
mnbMetric = MultinomialNB()
svmMetric = SVC()
knnMetric  = KNeighborsClassifier()

from sklearn.metrics import f1_score


K = [1000, 2000, 3000, 4000, 8000, 9000]

# storing best scores for all metrics
nbMetric_f1_score = []
mnbMetric_f1_score = []
svmMetric_f1_score = []
knnMetric_f1_score = []

for k_val in K:

    X_new1 = SelectKBest(chi2, k=k_val).fit_transform(feature_vectors, targets)
    f1 = cross_val_score(nbMetric, X_new1, targets, cv=5, scoring='f1_macro')
    nbMetric_f1_score.append(f1.mean())



for k_val in K:
```

```
    X_new1 = SelectKBest(chi2, k=k_val).fit_transform(feature_vectors, targets)
    f1 = cross_val_score(mnbMetric, X_new1, targets, cv=5, scoring='f1_macro')
    mnbMetric_f1_score.append(f1.mean())

for k_val in K:

    X_new1 = SelectKBest(chi2, k=k_val).fit_transform(feature_vectors, targets)
    f1 = cross_val_score(svmMetric, X_new1, targets, cv=5, scoring='f1_macro')
    svmMetric_f1_score.append(f1.mean())


for k_val in K:

    X_new1 = SelectKBest(chi2, k=k_val).fit_transform(feature_vectors, targets)
    f1 = cross_val_score(knnMetric, X_new1, targets, cv=5, scoring='f1_macro')
    knnMetric_f1_score.append(f1.mean())

#plotting graph to k values and f1 scores for all the metrics
p.figure(figsize=(7,7))
p.plot(K, nbMetric_f1_score,label = "Naive Bayes")
p.plot(K, mnbMetric_f1_score, label = "Multinomial Naive Bayes")
p.plot(K, svmMetric_f1_score, label = "SVM")
p.plot(K, knnMetric_f1_score, label = "KNN")
p.xlabel("K value in Select K-best Model- Chi square")
p.ylabel("F1 Macro Score for classifiers")
p.legend(loc = 'upper')
p.show()



# storing best scores for all metrics for mutual information
K = [1000, 2000, 3000, 4000, 8000, 9000]
nbMetric_f1_score = []
mnbMetric_f1_score = []
svmMetric_f1_score = []
knnMetric_f1_score = []

for k_val in K:

    X_new2 = SelectKBest(mutual_info_classif, k=k_val).fit_transform(feature_vectors, targets)
    f1 = cross_val_score(nbMetric, X_new2, targets, cv=5, scoring='f1_macro')
    nbMetric_f1_score.append(f1.mean())



for k_val in K:

    X_new2 = SelectKBest(mutual_info_classif, k=k_val).fit_transform(feature_vectors, targets)
    f1 = cross_val_score(mnbMetric, X_new2, targets, cv=5, scoring='f1_macro')
    mnbMetric_f1_score.append(f1.mean())
```

```
for k_val in K:

    X_new2 = SelectKBest(mutual_info_classif, k=k_val).fit_transform(feature_vectors, targets)
    f1 = cross_val_score(svmMetric, X_new2, targets, cv=5, scoring='f1_macro')
    svmMetric_f1_score.append(f1.mean())


for k_val in K:

    X_new2 = SelectKBest(mutual_info_classif, k=k_val).fit_transform(feature_vectors, targets)
    f1 = cross_val_score(knnMetric, X_new2, targets, cv=5, scoring='f1_macro')
    knnMetric_f1_score.append(f1.mean())


#plotting graph to k values and f1 scores for all the metrics
p.figure(figsize=(7,7))
p.plot(K, nbMetric_f1_score,label = "Naive Bayes")
p.plot(K, mnbMetric_f1_score, label = "Multinomial Naive Bayes")
p.plot(K, svmMetric_f1_score, label = "SVM")
p.plot(K, knnMetric_f1_score, label = "KNN")
p.xlabel("K value in Select K-best Model- Mutual Information")
p.ylabel("F1 Macro Score for classifiers")
p.legend(loc = 'upper')
p.show()
```

Clustering:

```
# Part 4: Document Clustering
# Import required libraries

import matplotlib.pyplot as p
import sklearn.metrics as metrics
from sklearn.metrics import accuracy_score
from sklearn.feature_selection import mutual_info_classif
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import mutual_info_classif
```

```
from sklearn.datasets import load_svmlight_file
from sklearn.cluster import KMeans, AgglomerativeClustering


feature_vectors, targets = load_svmlight_file("/home/soumya/Desktop/training_dataset_file.txt")




clusters = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
# Reduce dimension of data to save computation time
X = SelectKBest(mutual_info_classif, k=500).fit_transform(feature_vectors, targets)

X = X.toarray()

KSM = []
KMI = []
ASM = []
AMI = []


for n in clusters:
    kmeans_model = KMeans(n_clusters = n).fit(X)
    labels = kmeans_model.labels_
    sscore = metrics.silhouette_score(X, labels, metric='euclidean')
    smi = metrics.adjusted_mutual_info_score(targets, labels)
    KSM.append(sscore)
    KMI.append(smi)

    single_linkage_model = AgglomerativeClustering(n_clusters=n, linkage='average').fit(X)
    labels = single_linkage_model.labels_
    sscore = metrics.silhouette_score(X, labels, metric='euclidean')
    smi = metrics.adjusted_mutual_info_score(targets, labels)
    ASM.append(sscore)
    AMI.append(smi)

# plotting graphs for kmeans sihouette, kmeans mutual info, linkage sihouette, linkage mutual info to
observe performance
f, a = p.subplots(4, sharex=True)
f.suptitle('kmeans sihouette, kmeans mutual info, linkage sihouette, linkage mutual info')
p.xlabel('num of clusters')
p.ylabel('f1-scores')
a[0].plot(clusters, KSM)
a[1].plot(clusters, KMI)
a[2].plot(clusters, ASM)
a[3].plot(clusters, AMI)
p.show()
```

Util:

```
from nltk.stem.porter import *
import re
import nltk

stemmer = nltk.PorterStemmer()

def isStopWord(word):
   ''' using the NLTK functions, return true/false'''

   # ToDo
   f=open("stopwords", 'r')
   #stopwords file is uploaded
   stopword=[]
   for words in f:
   #words are split in document, stored in a list
```

```
        wrd=words.split()
        for e in wrd:
            #each word is appended
             stopword.append(e)
    #print stopword
    if word in stopword:
            #if word is a stopword, it drops that word in reduced list
            return False
    else:
            return True
    f.close()

def stemming(word):
    ''' return the stem, using a NLTK stemmer. check the project description for installing and using it'''

    # ToDo

    st=stemmer.stem(word)
    #stemmer stem all words to a root word
    return st
```