**INFORMATION RETRIEVAL**

**PROJECT TITLE: Text Mining**

**Project Members:  
  
Srivalli Kompella (U00965253)  
Sai sri Malempati (U00973516)**

**SOURCE CODE**

**feature-extract.py**

|  |
| --- |
| **import** **math**  **import** **os**  **import** **sys**  **import** **news**  **import** **time**  **import** **datetime**  **import** **nltk**  **from** **collections** **import** defaultdict  **from** **nltk.stem.snowball** **import** EnglishStemmer # Assuming we're working with English  **class** **Index**:  """ Inverted index datastructure """  **def** **\_\_init\_\_**(self, tokenizer, stemmer=None, stopwords=None):  """  tokenizer -- NLTK compatible tokenizer function  stemmer -- NLTK compatible stemmer  stopwords -- list of ignored words  """  self.tokenizer = tokenizer  self.stemmer = stemmer  self.index = defaultdict(list)  self.documents = {}  self.unique\_id = **0**  **if** **not** stopwords:  self.stopwords = set()  **else**:  self.stopwords = set(stopwords)  **def** **lookup**(self, word):  """  Lookup a word in the index  """  word = word.lower()  **if** self.stemmer:  word = self.stemmer.stem(word)  **return** [self.documents.get(id, None) **for** id **in** self.index.get(word)]  **def** **add**(self, document):  """  Add a document string to the index  """  content = document.title+ document.body  **for** token **in** [t.lower() **for** t **in** nltk.word\_tokenize(content)]:  **if** token **in** self.stopwords:  **continue**  **if** self.stemmer:  token = self.stemmer.stem(token)  **if** self.unique\_id **not** **in** self.index[token]:  self.index[token].append(document.docID)  self.documents[self.unique\_id] = document  self.unique\_id += **1**  **class** **feature\_extract**():  **def** **\_\_init\_\_**(self):  self.featureLookup={}  self.class\_map\_dic={}  self.index = Index(nltk.word\_tokenize,  EnglishStemmer(),  nltk.corpus.stopwords.words('english'))  **def** **find\_class**(self,x):  x=x.split("\_",**1**)[**1**]  **for** keys,values **in** self.class\_map\_dic.items():  **if** x **in** values:  **return** keys  **def** **calculate\_idf**(self,termfreq):  idf\_value = self.index.unique\_id / termfreq  idf = abs(math.log(idf\_value, **10**))  **return** idf  **def** **remove\_dupes**(self,orglist):  duplist=[]  **for** x **in** orglist:  **if** x **not** **in** duplist:  duplist.append(x)  **return** duplist  **def** **load\_training\_data\_file\_TF**(self,training\_file):  #f=open(training\_file.type,"w")  tfdoclist={}  idfdoclist={}  tfidfdoclist={}  **for** term **in** self.index.index.keys():  **for** doc **in** self.remove\_dupes(self.index.index[term]):  termfreq=self.index.index[term].count(doc)  idfval=self.calculate\_idf(termfreq)  tfidf=termfreq\*idfval  term\_id=self.featureLookup[term]  **if**(doc **in** tfdoclist.keys()):  tfdoclist.get(doc)[term\_id]=termfreq  **else**:  tfdoclist[doc]={term\_id:termfreq}  **if**(doc **in** idfdoclist.keys()):  idfdoclist.get(doc)[term\_id]=idfval  **else**:  idfdoclist[doc]={term\_id:idfval}  **if**(doc **in** tfidfdoclist.keys()):  tfidfdoclist.get(doc)[term\_id]=tfidf  **else**:  tfidfdoclist[doc]={term\_id:tfidf}  #write termfrequency file  **print**("Loading Term Frequency Training data file..")  f=open(training\_file+".TF","w")  **for** key,value **in** tfdoclist.items():  docstring=self.find\_class(key)  docstring+=' '  docstring+=str(value).replace(',','').replace(': ',':').split('{')[**1**].split('}')[**0**]  docstring+='**\n**'  f.write(docstring)  f.close()  **print**("Succesfully Loaded Term Frequency Training data file")  **print**("Loading IDF Training data file.....")  #write IDF file  f=open(training\_file+".IDF","w")  **for** key,value **in** idfdoclist.items():  docstring=self.find\_class(key)  docstring+=' '  docstring+=str(value).replace(',',' ').replace(': ',':').split('{')[**1**].split('}')[**0**]  docstring+='**\n**'  f.write(docstring)  f.close()  **print**("Succesfully IDF Training data file")  **print**("Loading TF-IDF Training data file.....")  #write TFIDF file  f=open(training\_file+".TFIDF","w")  **for** key,value **in** tfidfdoclist.items():  docstring=self.find\_class(key)  docstring+=' '  docstring+=str(value).replace(',',' ').replace(': ',':').split('{')[**1**].split('}')[**0**]  docstring+='**\n**'  f.write(docstring)  f.close()  **print**("Succesfully TF-IDF Training data file")  **def** **load\_feature\_definition\_file**(self,feature\_file):  f=open(feature\_file,'w')  #define the feature\_id & initiate to 1  ftr\_id=**1**  **for** trm **in** self.index.index.keys():  f.write('('+str(ftr\_id)+','+trm+')**\n**')  self.featureLookup[trm] = ftr\_id  ftr\_id=ftr\_id+**1**  f.close()  **def** **load\_class\_definition\_file**(self,class\_file):  f = open(class\_file, "w")  #declare the class mapping  self.class\_map\_dic = {  '1': ['comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware',  'comp.windows.x'], '2': ['rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey'],  '3': ['sci.crypt', 'sci.electronics', 'sci.med', 'sci.space'], '4': ['misc.forsale'],  '5': ['talk.politics.misc', 'talk.politics.guns', 'talk.politics.mideast'],  '6': ['talk.religion.misc', 'alt.atheism', 'soc.religion.christian']}  **for** key, value **in** self.class\_map\_dic.items():  **for** x **in** value:  f.write('(' + x + ',' + key + ')**\n**')  f.close()  **def** **feature\_extraction**(self,newsdir,feature\_file,class\_file,training\_file):  #Load newdirectory files  inputdocument= news.read\_news(newsdir)  **print**("Generating Index for documents (it might take approximately 57 seconds)....")  #Perform the Indexing  **for** doc **in** inputdocument.docs:  self.index.add(doc)  #Load class definition,feature definition,training files  **print**("Loading Class definition file..")  self.load\_class\_definition\_file(class\_file)  **print**("Loading feature definition file")  self.load\_feature\_definition\_file(feature\_file)  **print**("Loading Training File")  self.load\_training\_data\_file\_TF(training\_file)  **def** **test**():  #check whether read all files from directory given  newsdoc=news.read\_news("mini\_newsgroups")  **assert** len(newsdoc.docs) == **2000**  **print**("Test Case :: Loading newsdirectory-2k Documents PASSED")  #cehck whether Index created after stop words removed and done stemmed for a document  doc=newsdoc.docs[**1**]  **print**("\*\*\*\*\*\*\* Document considered for Index testing::")  **print**(doc.title+ doc.body)  index = Index(nltk.word\_tokenize,  EnglishStemmer(),  nltk.corpus.stopwords.words('english'))  index.add(doc)  indexstr=''  **for** x **in** index.index.keys():  indexstr+=x+' '  **print**('\*\*\*\*\* Document after removal of stopwords and stemming \*\*\*\*\*\*\*')  **print**(indexstr)  **print**("Test Case :: Index created passed")  #check whether feature\_definition\_file,class\_definition\_file,training\_data\_file created.  **if**(os.path.exists('feature\_definition\_file')):  **print**('Test Case :: Loading feature\_definition\_file passed')  **if**(os.path.exists('class\_definition\_file')):  **print**('Test Case :: Loading class\_definition\_file passed')  **from** **sklearn.datasets** **import** load\_svmlight\_file  feature\_vectors, targets = load\_svmlight\_file("training\_data\_file.TF")  **print**("Test Case :: Loading training\_data\_file.TF passed")  **from** **sklearn.datasets** **import** load\_svmlight\_file  feature\_vectors, targets = load\_svmlight\_file("training\_data\_file.IDF")  **print**("Test Case :: Loading training\_data\_file.IDF passed")  **from** **sklearn.datasets** **import** load\_svmlight\_file  feature\_vectors, targets = load\_svmlight\_file("training\_data\_file.TFIDF")  **print**("Test Case :: Loading training\_data\_file.TFIDF passed")  **if** \_\_name\_\_ == '\_\_main\_\_':  feature\_obj=feature\_extract()  #feature\_obj.feature\_extraction("mini\_newsgroups","feature\_definition\_file","class\_definition\_file","training\_data\_file")  feature\_obj.feature\_extraction(str(sys.argv[**1**]), str(sys.argv[**2**]),str(sys.argv[**3**]),str(sys.argv[**4**])) |

**test.py**

|  |
| --- |
| feature\_extarctobj=\_\_import\_\_('feature-extract')  **print**("\*\*\*\*\*\*\* Running Test Cases of feature extraction \*\*\*\*\*\*\*\*\*\*\*")  feature\_extarctobj.test() |

**news.py**

|  |
| --- |
| **import** **os**  **from** **os** **import** walk  **from** **doc** **import** Document  **class** **read\_news**():  **def** **\_\_init\_\_**(self,newsdir):  self.docs=[]  filespaths = []  #get path of newsdirectory  pathfile=os.getcwd() + "**\\**" + newsdir #"mini\_newsgroups"  #loop in the subdirectoy and read the files  **for** (dirpath, dirnames, filenames) **in** walk(pathfile):  **for** x **in** dirnames:  **for** (subdirpath, subdirnames, files) **in** walk(pathfile+"**\\**"+x):  **for** f **in** files:  self.readfiles(subdirpath,f,x)  **def** **readfiles**(self,dirname,filename,subdir):  #read file subject and last xx lines  filepath=dirname+"**\\**"+filename  cf = open(filepath)  docid = filename+"\_"+subdir  number\_of\_lines=**0**  title = ''  body = ''  linemessage = ''  startlines=False  **for** line **in** cf:  **if** 'Subject:' **in** line:  title = line[**9**:].strip() # got title  **elif** 'Lines:' **in** line:  **try**:  number\_of\_lines=int(line[**6**:])  **except** **Exception** **as** e:  **if** 'dog' **in** str(e):  number\_of\_lines=**24**  startlines = True  line = ''  **if** startlines:  #last\_line = cf.readlines()[-number\_of\_lines:]  last\_line=[i.replace('**\n**','') **for** i **in** cf.readlines()[-number\_of\_lines:]]  linemessage= ''.join(last\_line)  body = linemessage;  #convert file to document format  self.docs.append(Document(docid, title,body)) |

**doc.py**

|  |
| --- |
| '''  The document class, containing information from the raw document and possibly other tasks  The collection class holds a set of docuemnts, indexed by docID  '''  **class** **Document**:  **def** **\_\_init\_\_**(self, docid, title, body):  self.docID = docid  self.title = title  self.body = body  # add more methods if needed  **class** **Collection**:  ''' a collection of documents'''  **def** **\_\_init\_\_**(self):  self.docs = {} # documents are indexed by docID  **def** **find**(self, docID):  ''' return a document object'''  **if** self.docs.has\_key(docID):  **return** self.docs[docID]  **else**:  **return** None  # more methods if needed |

**classification.py**

|  |
| --- |
| **import** **warnings**  **import** **datetime**  **from** **sklearn.datasets** **import** load\_svmlight\_file  **from** **sklearn.model\_selection** **import** cross\_val\_score  **from** **sklearn.naive\_bayes** **import** MultinomialNB, BernoulliNB  **from** **sklearn.neighbors** **import** KNeighborsClassifier  **from** **sklearn.svm** **import** SVC  warnings.filterwarnings("ignore")  #run for multinomial navie Bayes classifier  **print**("\*\*\*\*\*\*\*multinomial Naive Bayes classifier\*\*\*\*\*\*\*\*")  clf = MultinomialNB()  #load Term frquency file as features and targets for multinomial  feature\_vectors, targets = load\_svmlight\_file("training\_data\_file.TF")  #run the cross\_validation method with f1\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='f1\_macro')  **print**("f1\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  #run the cross\_validation method with precision\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='precision\_macro')  **print**("precision\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  #run the cross\_validation method with recall\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='recall\_macro')  **print**("recall\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  **print**("**\n**")  #run for Naive Bayes classifier  **print**("\*\*\*\*\*\*\*\*Naive Bayes classifier\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  #load Term frequency file as features and targets for navie bayes  clf = BernoulliNB()  feature\_vectors, targets = load\_svmlight\_file("training\_data\_file.IDF")  #run the cross\_validation method with f1\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='f1\_macro')  **print**("f1\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  #run the cross\_validation method with precision\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='precision\_macro')  **print**("precision\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  #run the cross\_validation method with recall\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='recall\_macro')  **print**("recall\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  **print**("**\n**")  #run for k-nearest neighbors classifier  **print**("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*k-nearest neighbors classifier \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  clf = KNeighborsClassifier()  #load Term frequency file as features and targets for k-nn  feature\_vectors, targets = load\_svmlight\_file("training\_data\_file.TFIDF")  #run the cross\_validation method with f1\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='f1\_macro')  **print**("f1\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  #run the cross\_validation method with precision\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='precision\_macro')  **print**("precision\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  #run the cross\_validation method with recall\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='recall\_macro')  **print**("recall\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  **print**("**\n**")  #run for C-Support Vector Classifier  **print**("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*C-Support Vector Classifier\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  clf = SVC()  #load Term frequency file as features and targets for svm bayes  feature\_vectors, targets = load\_svmlight\_file("training\_data\_file.TFIDF")  #run the cross\_validation method with f1\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='f1\_macro')  **print**("f1\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  #run the cross\_validation method with precision\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='precision\_macro')  **print**("precision\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**))  #run the cross\_validation method with recall\_macro scoring and get the scores  scores = cross\_val\_score(clf, feature\_vectors, targets, cv=**5**, scoring='recall\_macro')  **print**("recall\_macro: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() \* **2**)) |

**feature\_selection.py**

|  |
| --- |
| **import** **datetime**  **from** **matplotlib** **import** pyplot  **from** **sklearn.datasets** **import** load\_svmlight\_file  **from** **sklearn.feature\_selection** **import** SelectKBest  **from** **sklearn.feature\_selection** **import** chi2, mutual\_info\_classif  **from** **sklearn.model\_selection** **import** cross\_val\_score  **from** **sklearn.naive\_bayes** **import** MultinomialNB, BernoulliNB  **from** **sklearn.neighbors** **import** KNeighborsClassifier  **from** **sklearn.svm** **import** SVC  multinominal\_chi=[]  multinominal\_mutal=[]  bernouli\_chi=[]  bernouli\_mutual=[]  knn\_chi=[]  knn\_mutual=[]  c\_support\_chi=[]  c\_support\_mutal=[]  #print(datetime.datetime.now())  #kvals=[500, 1500, 2500, 3500, 4500, 5500, 6500, 8500,10000]  **print**("Note:: This Programme run more than 20 minutes...")  #considered K values ranging from 500 to 2k  kvals=[\*range(**500**,**2000**,**200**)]  **print**(kvals)  #iterate for all the values of k  **for** k\_value **in** kvals:  **print**("Processing feature\_Selection on all algorithms for k-value:",k\_value)  clf = MultinomialNB()  #load training file TF  X, y = load\_svmlight\_file("training\_data\_file.TF")  #select K best feature using ch-square method  X\_new1 = SelectKBest(chi2, k=k\_value).fit\_transform(X, y)  scores = cross\_val\_score(clf, X\_new1, y, cv=**5**, scoring='f1\_macro')  multinominal\_chi.append(scores.mean())  # select K best feature using mutualinfo method  X\_new2 = SelectKBest(mutual\_info\_classif, k=k\_value).fit\_transform(X, y)  scores = cross\_val\_score(clf, X\_new2, y, cv=**5**, scoring='f1\_macro')  multinominal\_mutal.append(scores.mean())  clf = BernoulliNB()  X, y = load\_svmlight\_file("training\_data\_file.IDF")  # select K best feature using ch-square method  X\_new1 = SelectKBest(chi2, k=k\_value).fit\_transform(X, y)  scores = cross\_val\_score(clf, X\_new1, y, cv=**5**, scoring='f1\_macro')  bernouli\_chi.append(scores.mean())  # select K best feature using mutualinfo method  X\_new2 = SelectKBest(mutual\_info\_classif, k=k\_value).fit\_transform(X, y)  scores = cross\_val\_score(clf, X\_new2, y, cv=**5**, scoring='f1\_macro')  bernouli\_mutual.append(scores.mean())  clf = KNeighborsClassifier()  X, y = load\_svmlight\_file("training\_data\_file.TFIDF")  # select K best feature using ch-square method  X\_new1 = SelectKBest(chi2, k=k\_value).fit\_transform(X, y)  scores = cross\_val\_score(clf, X\_new1, y, cv=**5**, scoring='f1\_macro')  knn\_chi.append(scores.mean())  # select K best feature using mutualinfo method  X\_new2 = SelectKBest(mutual\_info\_classif, k=k\_value).fit\_transform(X, y)  scores = cross\_val\_score(clf, X\_new2, y, cv=**5**, scoring='f1\_macro')  knn\_mutual.append(scores.mean())  clf = SVC()  X, y = load\_svmlight\_file("training\_data\_file.TFIDF")  # select K best feature using ch-square method  X\_new1 = SelectKBest(chi2, k=k\_value).fit\_transform(X, y)  #apply cross val score method with scoring of f1\_macro  scores = cross\_val\_score(clf, X\_new1, y, cv=**5**, scoring='f1\_macro')  c\_support\_chi.append(scores.mean())  # select K best feature using mutualinfo method  X\_new2 = SelectKBest(mutual\_info\_classif, k=k\_value).fit\_transform(X, y)  scores = cross\_val\_score(clf, X\_new2, y, cv=**5**, scoring='f1\_macro')  c\_support\_mutal.append(scores.mean())  #plot figure for Chi-square  pyplot.figure(**1**)  #pyplot.subplot(211)  pyplot.plot(kvals, multinominal\_chi,label = "Multinomial Naive Bayes")  pyplot.plot(kvals, bernouli\_chi, label = "Bernoulli Naive Bayes")  pyplot.plot(kvals, knn\_chi, label = "KNN")  pyplot.plot(kvals, c\_support\_chi, label = "SVM")  pyplot.xlabel("K")  pyplot.ylabel("f1\_macro")  pyplot.title("CHI Square")  pyplot.legend(loc = 'best')  #plot figure for mutualinformation  pyplot.figure(**2**)  pyplot.plot(kvals, multinominal\_mutal,label = "Multinomial Naive Bayes")  pyplot.plot(kvals, bernouli\_mutual, label = "Bernoulli Naive Bayes")  pyplot.plot(kvals, knn\_mutual, label = "KNN")  pyplot.plot(kvals, c\_support\_mutal, label = "SVM")  pyplot.xlabel("K")  pyplot.ylabel("f1\_macro")  pyplot.title("Mutual Information")  pyplot.legend(loc = 'best')  #print(datetime.datetime.now())  pyplot.show() |

**clustering.py**

|  |
| --- |
| **from** **matplotlib** **import** pyplot  **from** **sklearn.cluster** **import** KMeans, AgglomerativeClustering  **from** **sklearn** **import** metrics  **from** **sklearn.datasets** **import** load\_svmlight\_file  **from** **sklearn.feature\_selection** **import** SelectKBest  **from** **sklearn.feature\_selection** **import** mutual\_info\_classif  feature\_vectors, targets = load\_svmlight\_file("training\_data\_file.TFIDF")  **print**("Note:: This Programme runs for approximatley 5 minutes...")  #K clsuters range from 2 to 25  clust\_list=[\*range(**2**,**26**)]  **print**(clust\_list)  silhouette\_score\_kmeans=[]  normalized\_score\_kmeans=[]  silhouette\_score\_agglormative=[]  normalized\_score\_agglormative=[]  #consider 100 best features for clustering  **print**("Selecting 100 best features.....")  X=SelectKBest(mutual\_info\_classif, k=**100**).fit\_transform(feature\_vectors, targets).toarray()  #run for each cluster number  **for** n\_clust **in** clust\_list:  **print**("Running for clusters:",n\_clust)  #apply kmeans clustering algorithm  kmeans\_model = KMeans(n\_clusters=n\_clust).fit(X)  clustering\_labels = kmeans\_model.labels\_  #calculate sc score  silhoutescore=metrics.silhouette\_score(X, clustering\_labels, metric='euclidean')  silhouette\_score\_kmeans.append(silhoutescore)  #calculate NMI score  normalized\_scores=metrics.normalized\_mutual\_info\_score(targets, clustering\_labels)  normalized\_score\_kmeans.append(normalized\_scores)  **for** n\_clust **in** clust\_list:  #apply hierarchial clustering algorithm  single\_linkage\_model = AgglomerativeClustering(n\_clusters=n\_clust, linkage='ward').fit(X)  clustering\_labels=single\_linkage\_model.labels\_  #calculate sc score  silhoutescore = metrics.silhouette\_score(X, clustering\_labels, metric='euclidean')  silhouette\_score\_agglormative.append(silhoutescore)  #calculate NMI score  normalized\_scores = metrics.normalized\_mutual\_info\_score(targets, clustering\_labels)  normalized\_score\_agglormative.append(normalized\_scores)  #plot figue for sc  plot1=pyplot.figure(**1**)  pyplot.plot(clust\_list, silhouette\_score\_kmeans,label="KMeans")  pyplot.plot(clust\_list, silhouette\_score\_agglormative,label="Hierarchical clustering")  pyplot.title("Sihouette Coefficient Scores")  pyplot.xlabel("Number of Clusters")  pyplot.ylabel("The Measures")  pyplot.legend(loc='best')  #plot figure for NMI  plot2=pyplot.figure(**2**)  pyplot.plot(clust\_list, normalized\_score\_kmeans,label="KMeans")  pyplot.plot(clust\_list, silhouette\_score\_agglormative,label="Hierarchical clustering")  pyplot.title("Normalized Mutual Information Scores")  pyplot.xlabel("Number of Clusters")  pyplot.ylabel("The Measures")  pyplot.legend(loc='best')  pyplot.show() |