Team Members: Anthony Edward Drow, UID U00759458

Syed Waqar Uddin, UID U00765829

Due Date April 22nd, 2019

IR Project 2 Source Code

Index.py:

```
import nltk
from collections import defaultdict
from nltk.stem.snowball import EnglishStemmer  # Assuming we're working with English
import numpy


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

        # Dictionary with terms as keys and their associated feature ID's
        self.termKeyDictionary = {}

        # Dictionary with featureID's as keys and their associated term
        self.featureIDKeyDictionary = {}

        #self.vector = numpy.zeros(len(self.items))
        self.documents = {}
        self.uniqueFeatureID = 1
        if not stopwords:
            self.stopwords = set()
        else:
            self.stopwords = set(stopwords)
```

```
    # Returns feature ID of the word in the dictionary, or None if it does not exist.
    def lookup(self, word):
        """
        Lookup a word in the index
        """
        word = word.lower()
        if self.stemmer:
            word = self.stemmer.stem(word)

        # Return the id of the word in dictionary.
        if word in self.termKeyDictionary:
            return self.termKeyDictionary[word]

    def add(self, word):
        # Skip stop words
        word = word.lower()
        if word not in self.stopwords:
            stemWord = self.stemmer.stem(word)
            if stemWord in self.termKeyDictionary:
                existingFeatureID = self.termKeyDictionary[stemWord]
                # increment count
            else:
                self.featureIDKeyDictionary[self.uniqueFeatureID] = stemWord
                self.termKeyDictionary[stemWord] = self.uniqueFeatureID
                self.uniqueFeatureID += 1

    def isStopWord(self, word):
        word = word.lower()
        if word in self.stopwords:
            return True
        else:
            return False


def tests():
    # Tests to verify that the index is working correctly.
    index = Index(nltk.word_tokenize, EnglishStemmer(), nltk.corpus.stopwords.words('english'))

    index.add('Industrial Disease')
    index.add('With')
    index.add('BAlls')
    index.add('Ball')
    index.add('Private Investigations')
```

```
    index.add('So Far Away')
    index.add('Twisting by the Pool')
    index.add('Skateaway')
    index.add('Walk of Life')
    index.add('Romeo and Juliet')
    index.add('Tunnel of Love')
    index.add('Money for Nothing')
    index.add('Sultans of Swing')

    # Index lookup tests.
    print("Index Lookup tests:")
    print(index.lookup('Industrial Disease'))
    print(index.lookup('Balling'))

    # Expect None here since this term does not exist.
    print(index.lookup('Missing'))

    print("\n")
    # Test for that stop words work.
    print("Stop Word Tests:")
    print(index.isStopWord("And") == True)
    print(index.isStopWord("or") == True)
    print(index.isStopWord("Facts") == False)

if __name__ == '__main__':
    tests()
```

Feature-extract.py:

```
import argparse
import Index
import collections
from Index import Index
import math
import os
import re
import nltk
from nltk.stem.snowball import EnglishStemmer  # Assuming we're working with English
```

```
def createFeatureDefinitionFile(fileName, index):
    f = open(fileName, "w")

    # Write out the feature definition file. Make sure we cast the key to a string so it can be
appended.
    for key, value in index.featureIDKeyDictionary.items():
        f.write("(" + str(key) + ", " + value + ")\n")

def parseVocabFile(fileName, index):
    f = open(fileName, "r")

    print("Parsing vocab file...")
    lines = f.readlines()
    for line in lines:
        for word in nltk.word_tokenize(line):
            index.add(word)

    print("Finished parsing vocab file.")
    # print(len(index.termKeyDictionary))

def createTrainingDataFile(fileName, newsDirectory, index, classDictionary, termWeightVal):

    currentDir = os.path.dirname(os.path.realpath(__file__))

    if os.path.isabs(newsDirectory):
        currentDir = newsDirectory
    else:
        # Get correct directory first.
        currentDir += "\\" + newsDirectory

    print(currentDir)

    IDFDictionary = {}

    # This list will store all of dictionaries for each documents terms and their associated TF
values.
    listOfDocDictionaries = []
    listOfDocClasses = []

    totalFiles = 0
    for subdir, dirs, files in os.walk(currentDir):
        totalFiles += len(files)
```

```
    # Iterate through the news group directory and parse files.
    for subdir, dirs, files in os.walk(currentDir):
        for file in files:
            fullPath = os.path.join(subdir, file)
            currentLineNum = 0
            # Output the current file we are processing
            print(fullPath)
            f = open(fullPath, "r")

            docWordCount = 0
            docDictionary = {}

            # Parse all the lines of each file and check for the key strings
            lines = f.readlines()
            for line in lines:
                if line.startswith("Subject:"):
                    # process this line
                    for word in nltk.word_tokenize(line):
                        featureID = index.lookup(word)
                        # We don't want to add the word if its not in our index. This ignores characters
treated as
                        # tokens such as : . , [ etc.
                        if featureID != None:
                            if featureID not in docDictionary:
                                docDictionary[featureID] = 1
                            else:
                                docDictionary[featureID] += 1

                            # Add the docID to IDFDictionary for the current term if it's not in there.
                            currentFileName = file
                            if featureID not in IDFDictionary:
                                IDFDictionary[featureID] = [currentFileName]
                            else:
                                docList = IDFDictionary[featureID]
                                if currentFileName not in docList:
                                    docList.append(currentFileName)
                            docWordCount += 1

                        # Stop words should still be taken into account when counting the total words in a
document
                        else:
                            if index.isStopWord(word):
                                docWordCount += 1
```

```
    if line.startswith("Lines:"):
        currentLineNum = re.findall('\d+', line)

# Process lines starting from bottom of file up based on lines input.
linesProcessed = 0
if not currentLineNum:
    # Only one file had a bad input, so here is its line count
    if currentFileName == '39668':
        currentLineNum = [13]
    elif currentFileName == '104595':
        currentLineNum = [7]
    elif currentFileName == '15387':
        currentLineNum = [13]
    elif currentFileName == '59559':
        currentLineNum = [30]
    elif currentFileName == '60237':
        currentLineNum = [11]
    elif currentFileName == '75916':
        currentLineNum = [11]
    elif currentFileName == '75918':
        currentLineNum = [57]
    elif currentFileName == '76277':
        currentLineNum = [27]

# Loop the file lines starting at the bottom, break if we hit our line limit.
for line in reversed(list(open(fullPath))):
    # if linesProcessed == int(currentLineNum[0]):
    if linesProcessed == int(currentLineNum[0]) - 1:
        break
    else:
        # Process this lines terms
        for word in nltk.word_tokenize(line):
            featureID = index.lookup(word)
            if featureID != None:
                if featureID not in docDictionary:
                    docDictionary[featureID] = 1
                else:
                    docDictionary[featureID] += 1
                docWordCount += 1

                # Add the docID to IDFDictionary for the current term if it's not in there.
                currentFileName = file
                if featureID not in IDFDictionary:
```

```
                    IDFDictionary[featureID] = [currentFileName]
                else:
                    docList = IDFDictionary[featureID]
                    if currentFileName not in docList:
                        docList.append(currentFileName)
            # Stop words should still be taken into account when counting the total words in a
document
            else:
                if index.isStopWord(word):
                    docWordCount += 1
        linesProcessed += 1

    # Calculate TF. Here key is the current term in the document and val is the frequency it
shows
    # up in the document.
    for key, val in docDictionary.items():
        # docDictionary[key] = val / docWordCount
        docDictionary[key] = math.log2(1 + val)

    od = collections.OrderedDict(sorted(docDictionary.items()))
    # print(od)
    # We are done processing this document's terms, add it to the list and move on to the next
file.
    listOfDocDictionaries.append(od)


    className = os.path.basename(subdir)
    if className in classDictionary:
        listOfDocClasses.append(classDictionary[className])
        # writeFile.write(str(classDictionary[className]) + " ")



# Modify the filename so we know which file is which.
if termWeightVal == 1:
    fileName += "TFIDF.txt"
elif termWeightVal == 2:
    fileName += "IDF.txt"
elif termWeightVal == 3:
    fileName += "TF.txt"

writeFile = open(fileName, "w")
```

```
    # Loops through the list of each document's term dictionary and writes out its class and
features.
    for index, dictionary in enumerate(listOfDocDictionaries):
        writeFile.write(str(listOfDocClasses[index]) + " ")

        # Calculate final TF IDF scores for each feature and then output it to the file based on which
we care about.
        for featureID in dictionary:
            TF = dictionary[featureID]
            docCountOfCurrentTerm = len(IDFDictionary[featureID])
            IDF = math.log10(totalFiles / docCountOfCurrentTerm)

            TFIDF = TF * IDF
            if termWeightVal == 1:
                writeFile.write(str(featureID) + ":" + str(TFIDF) + " ")
            elif termWeightVal == 2:
                writeFile.write(str(featureID) + ":" + str(IDF) + " ")
            elif termWeightVal == 3:
                writeFile.write(str(featureID) + ":" + str(TF) + " ")
            # writeFile.write(str(featureID) + ":" + str(TFIDF) + " ")

        writeFile.write("\n")


def createClassFile(fileName):
    f = open(fileName, "w")

    # We can hard code these class values and write them out to the file.
    classDictionary = {}
    classDictionary["comp.graphics"] = 1
    classDictionary["comp.os.ms-windows.misc"] = 1
    classDictionary["comp.sys.ibm.pc.hardware"] = 1
    classDictionary["comp.sys.mac.hardware"] = 1
    classDictionary["comp.windows.x"] = 1

    classDictionary["rec.autos"] = 2
    classDictionary["rec.motorcycles"] = 2
    classDictionary["rec.sport.baseball"] = 2
    classDictionary["rec.sport.hockey"] = 2

    classDictionary["sci.crypt"] = 3
    classDictionary["sci.electronics"] = 3
    classDictionary["sci.med"] = 3
```

```
    classDictionary["sci.space"] = 3

    classDictionary["misc.forsale"] = 4

    classDictionary["talk.politics.misc"] = 5
    classDictionary["talk.politics.guns"] = 5
    classDictionary["talk.politics.mideast"] = 5

    classDictionary["talk.religion.misc"] = 6
    classDictionary["alt.atheism"] = 6
    classDictionary["soc.religion.christian"] = 6

    for key, val in classDictionary.items():
        f.write("(" + key + ", " + str(val) + ")\n")

    return classDictionary

def tests():
    print("Running tests for feature extraction.")
    # Create index
    index = Index(nltk.word_tokenize, EnglishStemmer(), nltk.corpus.stopwords.words('english'))

    # Verify that the Vocab file exists.
    exists = os.path.isfile("VocabList.txt")
    if exists == False:
        print('Cannot find vocab file!!')
        return 0;
    else:
        print("Found Vocab file.")

    parseVocabFile("VocabList.txt", index)

    termCount = len(index.termKeyDictionary)
    print("Term count is: " + str(termCount))
    if termCount == 44985:
        print("Correct number of terms found!!")
    else:
        print("Incorrect nubmer of terms found.")


    print("\n")
    print("Index and TFIDF tests:")
    testIndex = Index(nltk.word_tokenize, EnglishStemmer(),
```

```
nltk.corpus.stopwords.words('english'))

    # Add 2 "Documents"
    terms = nltk.word_tokenize("Jack and Jill went up the hill cause Jack.")
    for term in terms:
        testIndex.add(term)

    terms = nltk.word_tokenize("Jill is very hungry.")
    for term in terms:
        testIndex.add(term)

    print("Jack featureID: " + str(testIndex.lookup("Jack")))
    # TF is log2 of 1 plus the term count.
    TF = math.log2(1 + 1)
    print("Jack TF: " + str(TF))

    # IDF is log10 of total Document count over number of docs the term appears in.
    IDF = math.log10(2 / 1)
    print("Jack IDF: " + str(IDF))

    TFIDF = TF * IDF
    print("Jack TFIDF: " + str(TFIDF))



if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument("NewsGroupDir", help="The input file name containing the documents
to parse.")
    parser.add_argument("featureDefFile", help="The name of the file to output the results to.")
    parser.add_argument("classDefFile", help="The name of the file to output the results to.")
    parser.add_argument("trainingDataFile", help="The name of the file to output the results to.")
    args = parser.parse_args()

    # First we parse the list of words in the dataset.
    index = Index(nltk.word_tokenize, EnglishStemmer(), nltk.corpus.stopwords.words('english'))
    parseVocabFile("VocabList.txt", index)

    # Create class definition file.
    print("Generating Class file...")
    classDictionary = createClassFile(args.classDefFile)
    print("Finished generating class file.")
```

```
    # Create feature definition file
    print("Creating feature defintion file...")
    createFeatureDefinitionFile(args.featureDefFile, index)
    print("Finished writing feature defintion file")


    # Create the training data file
    # 1 = TFIDF, 2 = IDF, 3 = TF
    termWeightVal = 1
    print("Generating training data file...")
    createTrainingDataFile(args.trainingDataFile, args.NewsGroupDir, index, classDictionary,
termWeightVal)
    print("Finished generating training data.")

    # Uncomment to run tests
    tests()
```

Classification.py:

```
from sklearn.datasets import load_svmlight_file
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
import warnings
warnings.filterwarnings('ignore')
clf = MultinomialNB()
feature_vectors, targets = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf, feature_vectors, targets, cv=5, scoring='f1_macro')
print("MultinomialNB(f1 macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(),scores.std() *
2))
clf = MultinomialNB()
feature_vectors, targets = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf, feature_vectors, targets, cv=5,scoring='precision_macro')
print("MultinomialNB(precision macro) Accuracy: %0.2f (+/- %0.2f)" %
(scores.mean(), scores.std() * 2))
clf = MultinomialNB()
feature_vectors, targets = load_svmlight_file("trainingdatafileTFIDF.txt")
```

```
scores = cross_val_score(clf, feature_vectors, targets, cv=5,scoring='recall_macro')
print("MultinomialNB(recall macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(),
scores.std() * 2))
clf2 = BernoulliNB()
feature_vectors2, targets2 = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf2, feature_vectors2, targets2, cv=5,scoring='f1_macro')
print("BernoulliNB(f1 macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(),
scores.std() * 2))
clf2 = BernoulliNB()
feature_vectors2, targets2 = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf2, feature_vectors2, targets2, cv=5,scoring='precision_macro')
print("BernoulliNB(precision_macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(),
scores.std() * 2))
clf2 = BernoulliNB()
feature_vectors2, targets2 = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf2, feature_vectors2, targets2, cv=5,scoring='recall_macro')
print("BernoulliNB(recall macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(),
scores.std() * 2))
clf3 = KNeighborsClassifier()
feature_vectors3, targets3 = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf3, feature_vectors3, targets3, cv=5,scoring='f1_macro')
print("KNeighborsClassifier(f1 macro) Accuracy: %0.2f (+/- %0.2f)" %
(scores.mean(), scores.std() * 2))
clf3 = KNeighborsClassifier()
feature_vectors3, targets3 = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf3, feature_vectors3, targets3, cv=5,scoring='precision_macro')
print("KNeighborsClassifier(precision macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(),
scores.std() * 2))
clf3 = KNeighborsClassifier()
feature_vectors3, targets3 = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf3, feature_vectors3, targets3, cv=5,scoring='recall_macro')
print("KNeighborsClassifier(recall macro) Accuracy: %0.2f (+/- %0.2f)" %(scores.mean(),
scores.std() * 2))
clf4 = SVC(gamma='auto')
feature_vectors4, targets4 = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf4, feature_vectors4, targets4, cv=5,scoring='f1_macro')
print("SVC (f1 macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() *2))
clf4 = SVC(gamma='auto')
feature_vectors4, targets4 = load_svmlight_file("trainingdatafileTFIDF.txt")
scores = cross_val_score(clf4, feature_vectors4, targets4, cv=5,scoring='precision_macro')
print("SVC (precision macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(),scores.std() * 2))
clf4 = SVC(gamma='auto')
feature_vectors4, targets4 = load_svmlight_file("trainingdatafileTFIDF.txt")
```

```
scores = cross_val_score(clf4, feature_vectors4, targets4, cv=5,scoring='recall_macro')
print("SVC (recall macro) Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(),scores.std() * 2))
```

Feature_selection.py:

```
from sklearn.datasets import load_svmlight_file
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC
from sklearn.feature_selection import chi2, mutual_info_classif
from sklearn.feature_selection import SelectKBest
import matplotlib.pyplot as pyplot
feature_vectors1, targets1 = load_svmlight_file("trainingdatafileTFIDF.txt")
feature_vectors2, targets2 = load_svmlight_file("trainingdatafileTFIDF.txt")
feature_vectors3, targets3 = load_svmlight_file("trainingdatafileTFIDF.txt")
import warnings
warnings.filterwarnings('ignore')
kvalue = [500, 1000, 2000, 3000, 4000, 5000, 6000, 8000,10000]
print ("Calculating and generating the plot:")
multinomialnbf1scores = []
bernoullinbf1scores = []
svcf1scores = []
kneighbourf1scores = []
for i in kvalue:
 cls1 = MultinomialNB()
 X_new1 = SelectKBest(chi2, k=i).fit_transform(feature_vectors1, targets1)
 f1 = cross_val_score(cls1, X_new1, targets1, cv=5, scoring='f1_macro')
 multinomialnbf1scores.append(f1.mean())
for i in kvalue:
 cls2 = BernoulliNB()
 X_new1 = SelectKBest(chi2, k=i).fit_transform(feature_vectors2, targets2)
 f1 = cross_val_score(cls2, X_new1, targets2, cv=5, scoring='f1_macro')
 bernoullinbf1scores.append(f1.mean())
for i in kvalue:
```

```
 cls3 = SVC(gamma='auto')
 X_new1 = SelectKBest(chi2, k=i).fit_transform(feature_vectors3, targets3)
 f1 = cross_val_score(cls3, X_new1, targets3, cv=5, scoring='f1_macro')
 svcf1scores.append(f1.mean())
for i in kvalue:
 cls4 = KNeighborsClassifier()
 X_new1 = SelectKBest(chi2, k=i).fit_transform(feature_vectors3, targets3)
 f1 = cross_val_score(cls4, X_new1, targets3, cv=5, scoring='f1_macro')
 kneighbourf1scores.append(f1.mean())
pyplot.figure(figsize=(9,9))
pyplot.plot(kvalue, multinomialnbf1scores,label = "Multinomial Naive Bayes")
pyplot.plot(kvalue, bernoullinbf1scores, label = "Bernoulli Naive Bayes")
pyplot.plot(kvalue, svcf1scores, label = "SVM")
pyplot.plot(kvalue, kneighbourf1scores, label = "KNN")
pyplot.xlabel("K")
pyplot.ylabel("f1_macro (CHI Square)")
pyplot.legend(loc = 'best')
pyplot.show()
print ("Calculating and generating the second plot:")
kvalue = [100, 400, 600, 800, 1000, 1200]
multinomialnbmif1scores = []
bernoullinbmif1scores = []
svcmif1scores = []
kneighbourmif1scores = []
for i in kvalue:
 cls1 = MultinomialNB()
 X_new1 = SelectKBest(mutual_info_classif, k=i).fit_transform(feature_vectors1,targets1)
 f1 = cross_val_score(cls1, X_new1, targets1, cv=5, scoring='f1_macro')
 multinomialnbmif1scores.append(f1.mean())
for i in kvalue:
 cls2 = BernoulliNB()
 X_new1 = SelectKBest(mutual_info_classif, k=i).fit_transform(feature_vectors2,targets2)
 f1 = cross_val_score(cls2, X_new1, targets2, cv=5, scoring='f1_macro')
 bernoullinbmif1scores.append(f1.mean())
for i in kvalue:
 cls3 = SVC(gamma='auto')
 X_new1 = SelectKBest(mutual_info_classif, k=i).fit_transform(feature_vectors3,targets3)
 f1 = cross_val_score(cls3, X_new1, targets3, cv=5, scoring='f1_macro')
 svcmif1scores.append(f1.mean())
for i in kvalue:
 cls4 = KNeighborsClassifier()
 X_new1 = SelectKBest(mutual_info_classif, k=i).fit_transform(feature_vectors3,targets3)
 f1 = cross_val_score(cls4, X_new1, targets3, cv=5, scoring='f1_macro')
```

```
 kneighbourmif1scores.append(f1.mean())
pyplot.figure(figsize=(9,9))
pyplot.plot(kvalue, multinomialnbmif1scores,label = "Multinomial Naive Bayes")
pyplot.plot(kvalue, bernoullinbmif1scores, label = "Bernoulli Naive Bayes")
pyplot.plot(kvalue, svcmif1scores, label = "SVM")
pyplot.plot(kvalue, kneighbourmif1scores, label = "KNN")
pyplot.xlabel("K")
pyplot.ylabel("f1_macro (Mutual Information)")
pyplot.legend(loc = 'best')
pyplot.show()
```

Clustering.py:

```
from sklearn.datasets import load_svmlight_file
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.cluster import KMeans, AgglomerativeClustering
import matplotlib.pyplot as pyplot
from sklearn.feature_selection import chi2, mutual_info_classif
from sklearn.feature_selection import SelectKBest
import warnings
warnings.filterwarnings('ignore')
print ("Please wait while the values are computed:")
feature_vectors3, targets3 = load_svmlight_file("trainingdatafileTFIDF.txt")
numberOfClusters = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,20, 21, 22, 23, 24,
25]
kMeansSilhouette = []
kMeansMutualInformation = []
agglomerativeClusteringSilhoutte = []
agglomerativeClusteringMutualInformation = []
topHundredFeatures = SelectKBest(mutual_info_classif,
k=100).fit_transform(feature_vectors3, targets3)
topHundredFeatures = topHundredFeatures.toarray()
for i in numberOfClusters:
 kmeans_model = KMeans(n_clusters=i).fit(topHundredFeatures)
 clustering_labels = kmeans_model.labels_
```

```
 silhouettescore = metrics.silhouette_score(topHundredFeatures,
clustering_labels, metric='euclidean')
 mutualInformationscore = metrics.normalized_mutual_info_score(targets3,
clustering_labels)
 kMeansSilhouette.append(silhouettescore)
 kMeansMutualInformation.append(mutualInformationscore)
#for i in numberOfClusters:
 single_linkage_model = AgglomerativeClustering(n_clusters=i,
linkage='ward').fit(topHundredFeatures)
 clustering_labels2 = single_linkage_model.labels_
 silhouettescore2 = metrics.silhouette_score(topHundredFeatures,
clustering_labels2, metric='euclidean')
 mutualInformationscore2 = metrics.normalized_mutual_info_score(targets3,
clustering_labels2)
 agglomerativeClusteringSilhoutte.append(silhouettescore2)
 agglomerativeClusteringMutualInformation.append(mutualInformationscore2)
pyplot.figure(figsize=(9,9))
pyplot.plot(numberOfClusters, kMeansSilhouette, label = "k-Means Silhouette Score")
pyplot.plot(numberOfClusters, agglomerativeClusteringSilhoutte, label =
"Agglomerative Clustering Silhouette Score")
pyplot.xlabel("Number Of Clusters")
pyplot.ylabel("K-Means & Agglomerative Clustering Silhouette scores")
pyplot.legend(loc = 'best')
pyplot.show()
pyplot.figure(figsize=(9,9))
pyplot.plot(numberOfClusters, kMeansMutualInformation, label = "k-Means MutualInformation
Score")
pyplot.plot(numberOfClusters, agglomerativeClusteringMutualInformation, label =
"Agglomerative Clustering MutualInformation Score")
pyplot.xlabel("Number Of Clusters")
pyplot.ylabel("K-Means & Agglomerative Clustering MutualInformation Scores")
pyplot.legend(loc = 'best')
pyplot.show()
print("---Plot graph finish---")
```