CS-313 : Text Mining (LAB)

Shubham Dey (M.Sc. CS) Roll No. : 23419CMP026 GitHub Repo

Program 01

Read from a '.txt' file and count the number of sentences, words, and characters in the file.

```python
# count no. of sentences
def countLines(file):
    f = open(file, 'r')
    dat = f.read().split('.')
    f.close()
    dat.pop()
    count = len(dat)
    return count

# count no. of words
def countWords(file):
    f = open(file, 'r')
    count = 0
    line = f.readline()
    while line:
        count += len(line.split())
        line = f.readline()
    f.close()
    return count

# count no. of characters
def countChars(file):
    f = open(file, 'r')
    count = 0
    line = f.readline()
    while line:
        for word in line.split():
            for i in word:
                if i.isalnum():
                    count += 1
        line = f.readline()
    f.close()
    return count

print('No. of Sentences = ', countLines(r'Files/data.txt'))
print('No. of Words = ', countWords(r'Files/data.txt'))
print('No. of Characters = ', countChars(r'Files/data.txt'))
```

Program 02

Read from a '.pdf' file and count the number of sentences, words, and characters in the file.

```python
from pypdf import PdfReader

# count no. of sentences
def countLines(file):
    f = PdfReader(file)
    count = 0
    for page in f.pages:
        dat = page.extract_text().split('.')
        dat.pop()
        count += len(dat)
    return count

# count no. of words
def countWords(file):
    f = PdfReader(file)
    count = 0
    for page in f.pages:
        dat = page.extract_text().split(' ')
        for i in dat:
            if i != '':
                count += 1
    return count

# count no. of characters
def countChars(file):
    f = PdfReader(file)
    count = 0
    for page in f.pages:
        dat = page.extract_text()
        for i in dat:
            if i.isalnum():
                count += 1
    return count

print('No. of Sentences = ', countLines(r'Files/data.pdf'))
print('No. of Words = ', countWords(r'Files/data.pdf'))
print('No. of Characters = ', countChars(r'Files/data.pdf'))
```

Program 03

Read from a '.docx' file and count the number of sentences, words, and characters in the file.

```python
from docx import Document

# count no. of sentences
def countLines(file):
    f = Document(file)
    count = 0
    for para in f.paragraphs:
        dat = para.text.split('.')
        dat.pop()
        count += len(dat)
    return count

# count no. of words
def countWords(file):
    f = Document(file)
    count = 0
    for para in f.paragraphs:
        dat = para.text.split(' ')
        for i in dat:
            if i != '':
                count += 1
    return count

# count no. of characters
def countChars(file):
    f = Document(file)
    count = 0
    for para in f.paragraphs:
        dat = para.text
        for i in dat:
            if i.isalnum():
                count += 1
    return count

print('No. of Sentences = ', countLines(r'Files/data.docx'))
print('No. of Words = ', countWords(r'Files/data.docx'))
print('No. of Characters = ', countChars(r'Files/data.docx'))
```

Program 04

Implement Term-Document incidence matrix for boolean retrieval.

```python
import os
import pandas as pd

def getFiles():
    lst = []
    for f in os.listdir():      # gives list of files in current directory
        if f.endswith('.txt'):  # taking only .txt files from current directory
            lst.append(f)
    return lst

# create term-document matrix
def createMatrix(files):
    fp = []     # index of this list work as documentID
    for file in files:
        f = open(file, 'r')
        fp.append(f)
    words = []  # to store unique words which acts as rows
    matrix = []
    for i, f in enumerate(fp):
        dat = f.read().split()
        for word in dat:
            if word not in words:
                words.append(word)
                matrix.append([0 for _ in range(len(fp))])
            matrix[words.index(word)][i] = 1
    return matrix, words

# display term-document matrix
def printMatrix(matrix, words, files):
    df = pd.DataFrame(matrix, columns=files, index=words)
    print(df)

# perform AND Query
def andQuery(w1, w2, matrix, words):
    try:    # gets row of word-1 if available, else error
        l1 = matrix[words.index(w1)]
        l1 = int(''.join(map(str, l1)), 2)
    except ValueError:
        return -1
    try:    # gets row of word-2 if available, else error
        l2 = matrix[words.index(w2)]
```

```
        l2 = int(''.join(map(str, l2)), 2)
    except ValueError:
        return -2
    return bin(l1 & l2)[2:]     # bitwise and

# perform OR Query
def orQuery(w1, w2, matrix, words):
    try:    # gets row of word-1 if available, else works with word-2
    ↪  only
        l1 = matrix[words.index(w1)]
        l1 = int(''.join(map(str, l1)), 2)
    except ValueError:
        print(f'{w1} Not Found')
        l1 = 0
    try:    # gets row of word-2 if available, else works with word-1
    ↪  only
        l2 = matrix[words.index(w2)]
        l2 = int(''.join(map(str, l2)), 2)
    except ValueError:
        print(f'{w2} Not Found')
        l2 = 0
    return bin(l1 | l2)[2:]     # bitwise or

# perform NOT Query
def notQuery(w, matrix, words):
    try:    # gets row of word if available, else gives all 0's
        l = matrix[words.index(w)]
        l = ''.join(map(str, l))
    except:
        l = '0' * len(matrix[0])
    return ''.join(['0' if i=='1' else '1' for i in l])     #
    ↪  complementing


if __name__ == '__main__':
    files = getFiles()
    matrix, words = createMatrix(files)

    print('\nTerm-Document Incidence Matrix: ')
    printMatrix(matrix, words, files)

    print('\nVocabulary:')
    print(words)

    print('\nTerms in lowercase & Operator in uppercase')
    query = input('Enter query: ').split()
```

```
85
86      if len(query) == 3:
87          if query[1] == 'AND':
88              ans = andQuery(query[0], query[2], matrix, words)
89              if type(ans) == 'str':
90                  ans = ans.zfill(len(matrix[0]))
91              if ans == -1:
92                  print(f'\n{query[0]} Not Found.')
93              elif ans == -2:
94                  print(f'\n{query[2]} Not Found.')
95              elif ans == '0'*len(matrix[0]):
96                  print(f'\n{query[0]} AND {query[2]} Not Available.')
97              else:
98                  print(f'\n{query[0]} AND {query[2]} are Available in
                     ↪  :')
99                  for i in range(len(ans)):
100                     if ans[i] == '1':
101                         print(files[i])
102
103         elif query[1] == 'OR':
104                 ans = orQuery(query[0], query[2], matrix, words)
105                 ans = ans.zfill(len(matrix[0]))
106                 if ans != '0'*len(matrix[0]):
107                     print(f'\n{query[0]} OR {query[2]} are Available in
                         ↪  :')
108                     for i in range(len(ans)):
109                         if ans[i] == '1':
110                             print(files[i])
111         else:
112             print("Invalid Query")
113
114     elif len(query) == 2 and query[0] == 'NOT':
115         ans = notQuery(query[1], matrix, words)
116         if ans == '0'*len(matrix[0]):
117             print(f'\n{query[1]} Available in ALL Docs.')
118         else:
119             print(f'\n{query[1]} is NOT Available in :')
120             for i in range(len(ans)):
121                 if ans[i] == '1':
122                     print(files[i])
123     else:
124         print("Invalid Query")
```

Program 05

Implement Inverted-Index for boolean retrieval.

```
import os

def getFiles():
    lst = []
    for f in os.listdir():      # gives list of files in current
    ↪  directory
        if f.endswith('.txt'):  # taking only .txt files from current
        ↪  directory
            lst.append(f)
    return lst

# create inverted index
def index(files):
    fp = []     # index of this list work as documentID
    for file in files:
        f = open(file, 'r')
        fp.append(f)
    dix = {}    # dictionary: key= terms & value= postings
    for i, f in enumerate(fp):
        dat = f.read().split()
        for word in dat:
            try:
                flag = True # doc not listed for the term
                for j in range(len(dix[word])):
                    if dix[word][j] == i:
                        flag = False # doc already listed for the term
                        break
                    elif dix[word][j] > i:
                        dix[word].insert(j, i)
                        flag = False
                        break
                if flag:
                    dix[word].append(i)
            except KeyError:
                dix[word] = [i] # inserting new term
    return dix

# perform AND Query
def andQuery(dix, w1, w2):
    try:    # get posting for word-1, else error
        l1 = dix[w1]
    except KeyError:
        return -1
```

```
    try:    # get posting for word-2, else error
        l2 = dix[w2]
    except KeyError:
        return -2

    # intersection of the two postings
    i, j = 0, 0
    lst = []
    while i < len(l1) and j < len(l2):
        if l1[i] == l2[j]:
            lst.append(l1[i])
            i += 1
            j += 1
        elif l1[i] < l2[j]:
            i += 1
        else:
            j += 1
    return lst

# perform OR Query
def orQuery(dix, w1, w2):
    try:    # get posting for word-1, else works with word-2 only
        l1 = dix[w1]
    except KeyError:
        print(f'{w1} Not Found')
        l1 = []
    try:    # get posting for word-2, else works with word-1 only
        l2 = dix[w2]
    except KeyError:
        print(f'{w2} Not Found')
        l2 = []

    # union of the two postings
    i, j = 0, 0
    lst = []
    while i < len(l1) and j < len(l2):
        if l1[i] == l2[j]:
            lst.append(l1[i])
            i += 1
            j += 1
        elif l1[i] < l2[j]:
            lst.append(l1[i])
            i += 1
        else:
            lst.append(l2[j])
            j += 1
```

```
    while i < len(l1):
        lst.append(l1[i])
        i += 1
    while j < len(l2):
        lst.append(l2[j])
        j += 1
    return lst

# perform NOT Query
def notQuery(dix, w, N):
    try:    # get posting for word
        l = dix[w]
    except KeyError:
        l = []
    lst = [i for i in range(N) if i not in l]
    return lst

if __name__ == '__main__':
    files = getFiles()
    idx = index(files)
    print('\nInverted Index:')
    for i in idx:
        print(i, ':', idx[i])
    print('\nVocabulary')
    for i in idx:
        print(i, end=', ')
    print()

    print('\nTerms in lowercase & Operator in uppercase')
    query = input('Enter query: ').split()

    if len(query) == 3:
        if query[1] == 'AND':
            ans = andQuery(idx, query[0], query[2])
            if ans == -1:
                print(f'{query[0]} Not Found.')
            elif ans == -2:
                print(f'{query[2]} Not Found.')
            elif not ans:
                print(f'{query[0]} AND {query[2]} Not Available.')
            else:
                print(f'{query[0]} AND {query[2]} are Available in :')
                for i in ans:
                    print(files[i])

        elif query[1] == 'OR':
```

```
            ans = orQuery(idx, query[0], query[2])
            if ans:
                print(f'{query[0]} OR {query[2]} are Available in :')
                for i in ans:
                    print(files[i])
        else:
            print("Invalid Query")

    elif len(query) == 2 and query[0] == 'NOT':
        ans = notQuery(idx, query[1], len(files))
        if ans:
            print(f'{query[1]} is NOT Available in :')
            for i in ans:
                print(files[i])
        else:
            print(f'\n{query[1]}Available in ALL Docs.')
    else:
        print("Invalid Query")
```

Program 06

Using NLTK perform Tokenization, Normalization, Stemming & Lemmetization.

```python
import nltk
nltk.download('punkt')      # download resources for tokenization & puntuations
nltk.download('stopwords')  # download resources for stopwords
nltk.download('wordnet')    # download resources for lemmetization

from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.stem.wordnet import WordNetLemmatizer
from truecase import get_true_case
import string

# document
text = "Mr. Smith is feeling Relaxed today, as The weather in USA is awesome. Did something troubled Him in U.S.A.? The birds are flying."
print('\nDocument:\n', text)

# Sentence Tokenization
tokenized_sent = sent_tokenize(text)
print('\nSentence Tokenization:\n', tokenized_sent)

# Word Tokenization
tokenized_word = word_tokenize(text)
print('\nWord Tokenization\n', tokenized_word)

# Frequency Distribution
fdist = FreqDist(tokenized_word)
print('\nMost frequent 5 words:\n', fdist.most_common(5))

# Lowercasing
lower_token = []
for token in tokenized_word:
    lower_token.append(token.lower())
print('\nLowercasing:\n', lower_token)

# Truecasing Sentences
true_text = []
for text in tokenized_sent:
    true_text.append(get_true_case(text))
```

```
print('\nTruecasing Sentences:\n', true_text)

# Tokenize Truecase Words
true_token = []
for token in true_text:
    true_token.extend(word_tokenize(token))
print('\nTruecase Words\n', true_token)

# Removing Punctuations
punctuations = list(string.punctuation)
tokens = []
for i in lower_token:
    if i not in punctuations:
        tokens.append(i)
print('\nAfter removing Punctuations:\n', tokens)

# Removing Stopwords
stopwords_english = stopwords.words("english")
filtered_tokens=[]
for w in tokens:
    if w not in stopwords_english:
         filtered_tokens.append(w)
print('\nAfter removing Stopwords:\n', filtered_tokens)

# Stemming
ps = PorterStemmer()
stem_words=[]
for w in filtered_tokens:
     stem_words.append(ps.stem(w))
print('\nAfter Stemming:\n', stem_words)

# Lemmetization
lem = WordNetLemmatizer()
lem_words=[]
for w in filtered_tokens:
     lem_words.append(lem.lemmatize(w))
print('\nAfter Lemmetization:\n', lem_words)
```

Program 07

Naive Bayes classification using scikit-learn.

```
from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

# documents
doc1 = 'Chinese Beijing Chinese'
doc2 = 'Chinese Chinese Shanghai'
doc3 = 'Chinese Macao'
doc4 = 'Tokyo Japan Chinese'

docs = [doc1, doc2, doc3, doc4]    # doc list
y = [1, 1, 1, 0]                # labels: 1=> chinese, 0=> not chinese

# displaying the training dataset
d = {'Documents': docs, 'Labels': y}
df = pd.DataFrame(d)
print('\n1=> chinese, 0=> not chinese')
print('\nTraining Dataset:\n', df)

# converting text to numeric using tf-idf
tfidf = TfidfVectorizer()
X = tfidf.fit_transform(docs).toarray()
print('\nAfter tf-idf :\n', X)
print('\nFeatures:\n', tfidf.get_feature_names_out())

# training Gaussian Naive Bayes
gnb = GaussianNB()
gnb.fit(X, y)
# training Multinomial Naive Bayes
mnb = MultinomialNB()
mnb.fit(X, y)
# training Bernoulli Naive Bayes
bnb = BernoulliNB()
bnb.fit(X, y)

# classifying new document
doc5 = ['Chinese Chinese Chinese Tokyo Japan']
a = tfidf.transform(doc5).toarray()

print('\nNew document: ', doc5[0])
print('\nGaussian NB: ', gnb.predict(a)[0])
print('Multinomial NB: ', mnb.predict(a)[0])
print('Bernoulli NB: ', bnb.predict(a)[0])
```

Program 08

Rocchio classification using scikit-learn.

```
import pandas as pd
from sklearn.neighbors import NearestCentroid
from sklearn.feature_extraction.text import TfidfVectorizer

# documents
doc1 = 'Chinese Beijing Chinese'
doc2 = 'Chinese Chinese Shanghai'
doc3 = 'Chinese Macao'
doc4 = 'Tokyo Japan Chinese'

docs = [doc1, doc2, doc3, doc4]    # doc list
y = [1, 1, 1, 0]                # labels: 1=> chinese, 0=> not chinese

# displaying the training dataset
d = {'Documents': docs, 'Labels': y}
df = pd.DataFrame(d)
print('\n1=> chinese, 0=> not chinese')
print('\nTraining Dataset:\n', df)

# converting text to numeric using tf-idf
tfidf = TfidfVectorizer()
X = tfidf.fit_transform(docs).toarray()
print('\nAfter tf-idf :\n', X)
print('\nFeatures:\n', tfidf.get_feature_names_out())

# training rocchio classifier
rocchio = NearestCentroid()
rocchio.fit(X, y)

# classifying new document
doc5 = ['Chinese Chinese Chinese Tokyo Japan']
a = tfidf.transform(doc5).toarray()

print('\nNew document: ', doc5[0])
print('Prediction: ', rocchio.predict(a)[0])
```

Program 09

k-means Clustering using scikit-learn.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans

docs = ["This is the most beautiful place in the world.",
        "This man has more skills to show in cricket than any other game.",
        "Hi there! how was your ladakh trip last month?",
        "There was a player who had scored 200+ runs in single cricket innings in his career.",
        "I have got the opportunity to travel to Paris next year for my internship.",
        "May be he is better than you in batting but you are much better than him in bowling.",
        "That was really a great day for me when I was there at Lavasa for the whole night.",
        "That's exactly I wanted to become, a highest ratting batsmen ever with top scores.",
        "Does it really matter wether you go to Thailand or Goa, its just you have spend your holidays.",
        "Why don't you go to Switzerland next year for your 25th Wedding anniversary?",
        "Travel is fatal to prejudice, bigotry, and narrow mindedness., and many of our people need it sorely on these accounts.",
        "Stop worrying about the potholes in the road and enjoy the journey.",
        "No cricket team in the world depends on one or two players. The team always plays to win.",
        "Cricket is a team game. If you want fame for yourself, go play an individual game.",
        "Because in the end, you won't remember the time you spent working in the office or mowing your lawn. Climb that goddamn mountain.",
        "Isn't cricket supposed to be a team sport? I feel people should decide first whether cricket is a team game or an individual sport."]

label = ['travel', 'cricket', 'travel', 'cricket', 'travel', 'cricket', 'travel', 'cricket', 'travel', 'travel', 'travel', 'travel', 'cricket', 'cricket', 'travel', 'cricket']
true_k = 2      # no. of clusters

# converting text to numeric using tf-idf
tfidf = TfidfVectorizer(stop_words='english')
```

```
X = tfidf.fit_transform(docs)

# training k-means
model = KMeans(n_clusters=true_k, max_iter=100)
model.fit(X)

# display top 10 terms in the clusters
order_centroids = model.cluster_centers_.argsort()[:, ::-1]
terms = tfidf.get_feature_names_out()
for i in range(true_k):
    print("\nCluster-%d (top 10 terms):" % i)
    for ind in order_centroids[i, :10]:
        print(terms[ind], end=", ")
    print()

new = ["Nothing is easy in cricket. Maybe when you watch it on TV, it looks easy. But it is not. You have to use your brain and time the ball."]
X = tfidf.transform(new)
predicted = model.predict(X)[0]
print('\nNew Document:\n', new[0])
print("\nPrediction = Cluster-", predicted)
```

Program 10

Perform Parts-Of-Speech (POS) Tagging using NLTK

```
from nltk import word_tokenize, pos_tag

import nltk
nltk.download('averaged_perceptron_tagger')     # download resources
 ↪  for POS tagger

# document
sent = "Albert Einstein was born in Ulm Germany in 1879"

# tokenizing the document
tokens = word_tokenize(sent)
print('Tokens:\n', tokens)

# POS tagging
pos = pos_tag(tokens)

print('\nPOS tagging of the tokens:')
for i in pos:
    print(i)
```