Stop words

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.stem import PorterStemmer

s = input()

stopwords = set(stopwords.words("english"))

word\_tokens = word\_tokenize(s)

ps = PorterStemmer()

final\_output = []

for i in word\_tokens:

if i not in stopwords:

i = ps.stem(i)

final\_output.append(i)

print(" ".join(final\_output))

Indexing

from glob import glob

from pprint import pprint

files = "m\*"

dic = {}

for file in glob(files):

with open(file,'r') as f:

txt = f.read().lower().split()

for word in txt:

indices = [i for i, x in enumerate(txt) if x == word]

if word not in dic.keys():

dic[word] = []

if(len(indices) > 0):

dic[word].append((file, len(indices), indices))

f.close()

pprint(dic)

#Inverted index ends here

print("Enter search term or phrase")

query = input().split()

if len(query) == 1:

print([dic[i] for i in dic.keys() if i == query[0]])

#Inverted index for one term ends here

else:

result = set([x[0] for x in dic[query[0]]])

for i in query:

result = result.intersection(set([x[0] for x in dic[i]]))

if(len(result) > 1):

print(result)

#keyerror here means no matches found

#Inverted index just for documents ends here

for i in result:

adjacent = True

word\_count = 0

all\_positions = []

for j in query:

word\_count = word\_count + 1

for inverted\_index in dic[j]:

if inverted\_index[0] == i :

all\_positions.extend(inverted\_index[2])

all\_positions.sort()

if word\_count > 1 :

counter = 0

for index in range(len(all\_positions) - 1):

if all\_positions[index+1] - all\_positions[index] == 1:

counter = 1

break

if counter == 0:

adjacent = False

print(all\_positions)

if adjacent == True:

print("Adjacent : True")

else:

print("Adjacent : False")

#Inverted index for adjacency check ends here

TF-IDF

from pprint import pprint

from math import log

doc1 = " The car is driven on the road but it crashed down hard "

doc2 = " The truck is driven on the highway but there was no car "

doc3 = " I have a car but I do not own a truck "

doc4 = " Car and truck are important automobiles "

doc5 = " I want to buy a car "

#can do with glob and read content from file too, just saying

docs = [doc1.lower().split(),

doc2.lower().split(),

doc3.lower().split(),

doc4.lower().split(),

doc5.lower().split()]

#TF starts here

dic\_tf = {}

for doc in docs:

temp\_dic = {}

for term in set(doc):

term\_frequency = doc.count(term)/len(doc)

temp\_dic[term] = term\_frequency

dic\_tf[docs.index(doc)] = temp\_dic

pprint(dic\_tf)

print()

#IDF starts here

total\_terms = []

for doc in docs:

total\_terms.extend(doc)

total\_terms = set(total\_terms)

dic\_idf = {}

for term in total\_terms:

count = 0

for doc in docs:

if term in doc:

count += 1

dic\_idf[term] = log(len(docs)/count)

pprint(dic\_idf)

print()

#TF-IDF starts here

dic\_tf\_idf = dic\_tf

for i in dic\_tf\_idf:

for j in dic\_tf\_idf[i]:

dic\_tf\_idf[i][j] = dic\_tf\_idf[i][j] \* dic\_idf[j]

pprint(dic\_tf\_idf)

Apriori

import numpy as np

import pandas as pd

from apyori import apriori

from pprint import pprint

df = pd.read\_csv("store\_data.csv", header=None)

records\_temp = []

for i in range(len(df.values)):

records\_temp.append([str(df.values[i,j]) for j in range(len(df.values[0]))])

records = []

for record in records\_temp:

records.append(list(filter(lambda x: x!= 'nan', record)))

association\_rules = apriori(records, min\_support = 0.0045, min\_confidence = 0.2, min\_lift = 3, min\_length = 2)

association\_rules = list(association\_rules)

pprint(association\_rules)

for rule in association\_rules:

print("Rule : " + ",".join(rule[2][0][0]) + " -> " + ",".join(rule[2][0][1]))

#Support is rule[1]

#Confidence is rule[2][0][2]

#Lift is rule[2][0][3]

Encoding

from math import \*

n = [i for i in range(1,31,2)]

def unary(n):

if n==0:

return '1'

return '0'\*n + '1'

def gamma(k):

binary = bin(k)[2:]

return (len(binary) - 1)\*'0' + binary

def delta(k):

binary = bin(k)[bin(k).index('1')+1:]

return gamma(floor(log(k,2))+1) + binary

b=10

def golomb(k):

q = n//b

r = n-q\*b

x1 = unary(q)

ubits = ceil(log(b,2))

bitlength= 2\*\*ubits - b

if(r>=bitlength):

return x1+bin(r + bitlength)[2:].zfill(ubits)

else:

return x1+bin(r )[2:].zfill(ubits-1)

for i in n:

print(delta(i))

Classification

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB, MultinomialNB, BernoulliNB

from sklearn.metrics import accuracy\_score, confusion\_matrix

df = pd.read\_csv("../input/train.csv")

print(df.head())

df.isnull().sum()

df["Age"] = df["Age"].fillna(df["Age"].mean())

df["Embarked"] = df["Embarked"].fillna('S')

df = df.drop(['Name', 'Cabin','Ticket'], axis = 1)

mapper = { 'S' : 1, 'C' : 2, 'Q' : 3}

df["Embarked"] = df["Embarked"].map(lambda x: mapper[x])

mapper = { 'male' : 1, 'female' : 2}

df["Sex"] = df["Sex"].map(lambda x: mapper[x])

gaussian = GaussianNB()

x = df.drop('Survived', axis=1)

y = df['Survived']

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y, test\_size = 0.3, random\_state = 42)

df.columns

gaussian.fit(x\_train, y\_train)

y\_pred = gaussian.predict(x\_test)

confusion\_matrix(y\_pred, y\_test)

accuracy\_score(y\_test, y\_pred) \* 100

PageRank

import numpy as np

def pagerank(M, eps=1.0e-8, d=0.85):

N = M.shape[1]

v = np.random.rand(N, 1)

v = v / np.linalg.norm(v, 1)

last\_v = np.ones((N, 1), dtype=np.float32) \* np.inf

M\_hat = (d \* M) + (((1 - d) / N) \* np.ones((N, N), dtype=np.float32))

while np.linalg.norm(v - last\_v, 2) > eps:

last\_v = v

v = np.matmul(M\_hat, v)

return v

M = np.array([[0, 0, 0, 0, 1],

[0.5, 0, 0, 0, 0],

[0.5, 0, 0, 0, 0],

[0, 1, 0.5, 0, 0],

[0, 0, 0.5, 1, 0]])

v = pagerank(M, 0.001, 0.85)

print (v)

1. Write a program to implement apriori algorithm from web log data.

**CODE:**

import pandas as pd

from itertools import combinations

data = [

{'1': 'a' , '2':'b' , '3':'c'},

{'1': 'b' , '2':'c' , '3':'d'},

{'1': 'a' , '2':'b' , '3':'c' , '4':'d'},

{'1': 'b' , '2':'c'},

{'1': 'a' , '2':'b' , '3':'d'},

{'1': 'd' , '2':'e'},

{'1': 'a' , '2':'b' , '3':'c'},

{'1': 'c' , '2':'d' , '3':'e'},

{'1': 'a' , '2':'b' , '3':'c'}]

df = pd.DataFrame(data)

def apriori(trans, support=0.2, minlen=1):

data=pd.get\_dummies(trans.unstack().dropna()).groupby(level=1).sum()

collen, rowlen =data.shape

pattern = []

for cnum in range(minlen, rowlen+1):

for cols in combinations(data, cnum):

patsup = data[list(cols)].all(axis=1).sum()

patsup=float(patsup)/collen

pattern.append([",".join(cols), patsup])

sdf = pd.DataFrame(pattern, columns=["Pattern", "Support"])

results=sdf[sdf.Support >= support]

return results

Ap = apriori(df)

print(Ap)

1. Write a program to demonstrate hierarchical and K-means clustering.

**CODE FOR HIERARCHIAL CLUSTERING:**

import numpy as np

import matplotlib.pyplot as plt

from scipy.cluster.hierarchy import dendrogram, linkage

#from matplotlib import pyplot as plt

s1 = 'name python computer'.split()

s2 = 'alpha python'.split()

s3 = 'alpha work computer'.split()

s4 = 'name office linkage'.split()

s5 = 'linkage study'.split()

terms = sorted(list(set(s1+s2+s3+s4+s5)))

term\_doc = [[0 for i in range(len(terms))] for j in range(len([s1,s2,s3,s4,s5]))]

c = 0

for i in [s1,s2,s3,s4,s5]:

term\_doc[c] = [i.count(terms[j]) for j in range(len(terms))]

c+=1

A = term\_doc

X = np.array(A)

linked = linkage(X, 'single')

lo = 1

hi = np.shape(X)[0]+1

labelList = range(lo, hi)

plt.figure(figsize = (hi-lo, 5))

dendrogram(linked,

orientation = 'top',

labels=labelList,

distance\_sort = 'descending',

show\_leaf\_counts=True)

plt.show()

**CODE FOR k–Means CLUSTERING:**

from copy import deepcopy

import numpy as np

import pandas as pd

from matplotlib import pyplot as plt

data = pd.read\_csv('xclara.csv')

f1 = data['V1'].values

f2 = data['V2'].values

X = np.array(list(zip(f1, f2)))

def dist(a, b, ax=1):

return np.linalg.norm(a - b, axis=ax)

k = 3

C\_x = np.random.randint(0, np.max(X)-20, size=k)

C\_y = np.random.randint(0, np.max(X)-20, size=k)

C = np.array(list(zip(C\_x, C\_y)), dtype=np.float32)

print("Initial Centroids")

print(C)

C\_old = np.zeros(C.shape)

clusters = np.zeros(len(X))

error = dist(C, C\_old, None)

while error!=0:

for i in range(len(X)):

distances = dist(X[i], C)

cluster = np.argmin(distances)

clusters[i] = cluster

C\_old = deepcopy(C)

for i in range(k):

points = [X[j] for j in range(len(X)) if clusters[j] == i]

C[i] = np.mean(points, axis=0)

error = dist(C, C\_old, None)

colors = ['r', 'g', 'b', 'y', 'c', 'm']

fig, ax = plt.subplots()

for i in range(k):

points = np.array([X[j] for j in range(len(X)) if clusters[j] == i])

ax.scatter(points[:, 0], points[:, 1], s=7, c=colors[i])

1. Write a program to reformulate the original query using Rocchio method.

**CODE:**

q0 = [float(i) for i in input('Enter query matrix: ').split()]

n = int(input('Enter number of relevant docs: '))

relevant = []

for i in range(n):

ai = [float(i) for i in input('Enter relevant matrix ' + str(i+1) + ' out of ' + str(n) + ': ').split()]

relevant.append(ai)

m = int(input('Enter number of irrelevant docs: '))

irrelevant = []

for i in range(n):

ai = [float(i) for i in input('Enter relevant matrix ' + str(i+1) + ' out of ' + str(n) + ': ').split()]

irrelevant.append(ai)

relevant\_centroid = [0 for i in range(len(q0))]

for i in range(len(q0)):

x = 0

for j in range(n):

x+=relevant[j][i]

relevant\_centroid[i] = x/n

irrelevant\_centroid = [0 for i in range(len(q0))]

for i in range(len(q0)):

x = 0

for j in range(n):

x+=irrelevant[j][i]

irrelevant\_centroid[i] = x/n

alpha = float(input('Enter alpha: '))

beta = float(input('Enter beta: '))

gamma = float(input('Enter gamma: '))

qf = [0 for i in range(len(q0))]

for i in range(len(q0)):

qf[i] = alpha\*q0[i] + beta\*relevant\_centroid[i] - gamma\*irrelevant\_centroid[i]

print('Reformulated query matrix: ', end='')

print(\*qf)