1. Build inverted index

```
import numpy as np
import pandas as pd
import os
import re
import json
from google.colab import drive
drive.mount('/content/drive')
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
from pandas.core.internals import blocks
class InfoRetrival:
  def __init__(self,corpus):
    self.corpus = corpus
    self.documents = []
    self.titles = []
    self.tokens = []
    self.termDocs = {}
    self.invertIndex = {}
    self.boolIndex = {}
    self.positionIndex = {}
    self.biwords = {}
    self.blocks = {}
    self.BSBI = {}
    self.SPIMI = {}
    for i in os.listdir(corpus):
      if i.split('.')[-1] == 'txt':
        self.documents.append(open(corpus+i).read())
```

```
        self.titles.append(i[:-4])
    def makeTerms(self,document):
      terms = document.split(" ")
      n = len(terms)
      for i in range(n):
        if not terms[i].islower():
          terms[i] = terms[i].lower()
        terms[i] = re.sub(r'[^a-zA-Z0-9]', '', terms[i])
      return terms
    def removeStopWords(self,terms):
      # "i","am","it","is","the","we","was","a","an","and","or","to","so",
      stopWords =
[".",";","!","@","#","$","%","^","&","*","(",")","{","}","[","]",":","'","/","
<",">","","+"]
      i = 0
      while i < len(terms):
        if terms[i] in stopWords  or len(terms[i])==0:
          terms.remove(terms[i])
        else:
          i+=1
      return terms
    def stemming(self):
      ps = PorterStemmer()
      for title in self.termDocs:
        n = len(self.termDocs[title])
        for i in range(n):
          self.termDocs[title][i] = ps.stem(self.termDocs[title][i])
    def termsDocuments(self):
      for title,docs in zip(self.titles,self.documents):
```

```
      self.termDocs[title] = self.removeStopWords(self.makeTerms(docs))
  def tokenize(self):
    self.termsDocuments()
    self.stemming()
    for title in self.termDocs:
      n = len(self.termDocs[title])
      for i in range(n):
        if self.termDocs[title][i] in self.tokens:
          continue
        self.tokens.append(self.termDocs[title][i])
    self.tokens.sort()
def invertedIndex(self):
    self.tokenize()
    for token in self.tokens:
      for doc in self.termDocs:
        if token in self.termDocs[doc]:
          if token not in self.invertIndex:
            self.invertIndex[token] = []
          self.invertIndex[token].append(doc)
```

Output:

{'a': ['doc1', 'doc2', 'doc3', 'doc4', 'doc7', 'doc8', 'doc9', 'doc10', 'doc5', 'doc6'], 'among': ['doc6'],…… 'your': ['doc8', 'doc10'], 'youso': ['doc8']}

2. Process Boolean queries

```
def booleanIndex(self):
 self.tokenize()
 for i in self.tokens:
  for j in self.titles:
   if i not in self.boolIndex:
    self.boolIndex[i] = []
   if j in self.invertIndex[i]:
    self.boolIndex[i].append(1)
   else:
    self.boolIndex[i].append(0)


def boolean_and(self,row1,row2):
 n = len(row1)
 for i in range(n):
  if row1[i] == row2[i] == 1:
   row1[i] = 1
  else:
   row1[i] = 0
 return row1
def boolean_or(self,row1,row2):
 n = len(row1)
 for i in range(n):
  if row1[i] == 1 or row2[i] == 1:
   row1[i] = 1
  else:
   row1[i] = 0
```

```
  return row1
def boolean_not(self,row):
  n = len(row)
  for i in range(n):
    if row[i] :
      row[i] = 0
    else:
      row[i] = 1
  return row
def booleanQuery(self,query):
  query = query.split(" ")
  opearator = ""
  flag = False
  n = 10
  result = []
  for i in query:
    if i in ['and','or']:
      opearator = i
    elif i == 'not':
      flag = True
    else:
      if flag :
        flag = False
        result = self.boolean_not(self.boolIndex[i])
      if opearator == "and":
        result = self.boolean_and(result,self.boolIndex[i])
        operator = ""
      elif opearator == "or":
        result = self.boolean_or(result,self.boolIndex[i])
```

```
            operator = ""
        else:
            result = self.boolIndex[i]
    ans = []
    for i in range(n):
      if result[i]:
        ans.append(self.titles[i])
    return ans
```

Output:

obj.booleanIndex()

obj.boolIndex

obj.printBooleanIndex()

	a	abl	about	accept	achehumili	act	action	after	afterjack	afternoon	...	wring	yard	yesterday	yet	you	youd	youll	youngster	your	youso
doc1	1	0	0	0	0	1	0	1	0	0	...	0	0	0	0	1	0	0	0	0	0
doc2	1	0	1	0	0	0	1	0	0	0	...	0	0	0	0	1	0	0	0	0	0
doc3	1	1	0	1	1	1	0	1	0	0	...	0	0	0	0	1	0	0	0	0	0
doc4	1	0	0	0	0	0	0	1	0	0	...	0	0	0	0	0	0	0	0	0	0
doc7	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
doc8	1	0	1	0	0	0	0	1	1	0	...	1	1	1	1	1	1	1	1	1	1
doc9	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
doc10	1	0	1	0	0	0	0	0	0	1	...	0	0	0	0	1	0	0	0	1	0
doc5	1	0	0	0	0	0	0	0	0	1	...	0	0	0	0	0	0	0	0	0	0
doc6	1	0	1	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

obj.booleanQuery("tiger or lion")

['doc2', 'doc7', 'doc10', 'doc6']

3. Build positional index and process phrase queries

```
def positionalIndex(self):
  ps = PorterStemmer()
  for i in self.tokens:
    self.positionIndex[i] = {}
    for j in self.titles:
      document = open(self.corpus+(j+".txt")).read().lower().split(" ")
      n = len(document)
      row = []
      for k in range(n):
        if i == ps.stem(document[k]):
          row.append(k)
      if len(row)>0:
        self.positionIndex[i][j] = row
def commonDocument(self,terms):
  commonDocs = set(self.positionIndex[terms[0]])
  for i in range(1,len(terms)):
    commonDocs                                                    =
commonDocs.intersection(self.positionIndex[terms[i]])
  return list(commonDocs)
def phraseQuery(self,query):
  print("HELLo")
  positions = {}
  ps = PorterStemmer()
  query = query.lower().split(" ")
  n = len(query)
```

```
for i in range(n):
  query[i] = ps.stem(query[i])
result = []
commonDocs = self.commonDocument(query)
for doc in commonDocs:
  for position in self.positionIndex[query[0]][doc]:
    i = 1
    while i<n:
      if (position+i) in self.positionIndex[query[i]][doc]:
        i+=1
      else:
        break
    if i == n:
      result.append(doc)
return result
```

Output:

obj.positionalIndex()

obj.positionIndex

{'a': {'doc1': [0, 21, 38, 52, 107, 110, 120, 137], 'doc2': [110, 114, 122, 139], 'doc3': [2, 6, 33, 42, 53, 89, 115, 245, 249, 307], 'doc4': [3, 21, 48, 60, 65], ……. 'youngster': {'doc8': [1243]}, 'your': {'doc8': [203, 255, 313, 401, 924,

1670], 'doc10': [136, 150, 513]}}

---

obj.phraseQuery("once upon")

['doc10', 'doc7', 'doc3', 'doc8']

4. Build bi-gram index and process wildcard queries

```
def biwordIndexing(self):
 for token in self.tokens:
  n = len(token)
  if n>0:
   l = ['$'+token[0]]
   if ('$'+token[0]) not in self.biwords :
    self.biwords['$'+token[0]] = [token]
   else:
    self.biwords['$'+token[0]].append(token)
   for i in range(n-1):
    if (token[i:i+2]) not in self.biwords :
     self.biwords[token[i:i+2]] = [token]
    else:
     self.biwords[token[i:i+2]].append(token)
   if (token[n-1]+'$') not in self.biwords :
    self.biwords[token[n-1]+'$'] = [token]
   else:
    self.biwords[token[n-1]+'$'].append(token)


def wildcardQuery(self,query):
 s = ['$'+query[0]]
 k = 0
 if s[-1] in ["*$","$*"]:
  s = ['$'+query[1]]
  k = 1
 for i in range(k,len(query)-1):
  if '*' not in query[i:i+2]:
```

```
            s.append(query[i:i+2])
        s.append(query[-1]+'$')
        result = set(self.biwords[s[0]])
        for i in range(1,len(s)):
          result = result.intersection(set(self.biwords[s[i]]))
        return list(result)
      def printBooleanIndex(self):
        return pd.DataFrame(self.boolIndex,index=self.titles)
```

Output:

obj.biwordIndexing()

obj.biwords

{'$a': ['a', 'abl', 'about', 'accept', 'achehumili', 'act', 'action', 'after', 'afterjack', 'afternoon', 'again', 'againjack', 'ago', 'agre', 'aliv', 'all', 'allth', 'alon', 'along', 'alreadi', 'also', 'alway', 'am', 'among', 'amus', 'an', 'and', 'angri', 'ani', 'anim', 'animaly', 'anoth', 'anymor', 'anywher', 'appear', 'are', 'arent', 'arm', 'arrang', 'arriv', 'as', 'asham', 'ask', 'asleep', 'astand',…… 'where', 'whi', 'which', 'while', 'whileand', 'white', 'who', 'whole', 'widow', 'wife', 'wifefeefifofum', 'will', 'window', 'winter', 'with', 'without', 'woke', 'woman', 'womangood', 'wonder', 'work', 'worth', 'would', 'wring'], 'wr': ['wring'], '$y': ['yard', 'yesterday', 'yet', 'you', 'youd', 'youll', 'youngster', 'your', 'youso'], 'ya': ['yard'], 'yo': ['you', 'youd', 'youll', 'youngster', 'your', 'youso'], 'gs': ['youngster']}

---

obj.wildcardQuery("a*e")

['agre', 'are', 'ate', 'axe']

5. Correct spellings in the query using edit distance

```
def correctWords(self,word):
 result ,dp,row = {},[],list(word)
 for token in self.tokens:
  column = list(token)
  for i in range(len(row)+1):
   dp.append([0 for j in range(len(column)+1)])
  for i in range(len(row)+1):
   for j in range(len(column)+1):
    if j == 0:
     dp[i][j] = i
    elif i == 0:
     dp[i][j] = j
  for i in range(1,len(row)+1):
   for j in range(1,len(column)+1):
    if row[i-1] == column[j-1]:
     dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
    else:
     dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1
  if dp[-1][-1] not in result:
   result[dp[-1][-1]] = []
  result[dp[-1][-1]].append(token)
 return result[min(result.keys())]
```

Output:

```
obj.correctWords('lon')
['alon', 'lion', 'lone', 'long', 'look', 'on', 'son', 'soon']
```

6. Implement BSBI algorithm

```
 def leastFactor(self,n):
  for i in range(2,n):
   if n%i == 0:
    return i
  return 1
 def makeBlock(self):
  ps = PorterStemmer()
  n = len(self.titles)
  b = self.leastFactor(n)
  c = 0
  self.blocks = {}
  c = 0
  # print(b,n)
  for i in range(0,n,b):
   self.blocks["block"+str(c+1)] = {}
   for j in range(i,i+b):
    terms =
self.removeStopWords(self.makeTerms(open(self.corpus+self.titles[j]+".t
xt").read()))
     for term in terms:
      term = ps.stem(term)
      if term not in self.blocks["block"+str(c+1)]:
       self.blocks["block"+str(c+1)][term] = []
      if self.titles[j] not in self.blocks["block"+str(c+1)][term]:
       self.blocks["block"+str(c+1)][term].append(self.titles[j])
   c+=1
 def mergeDocs(self,doc1,doc2):
```

```
  doc1.sort()
  doc2.sort()
  m = len(doc1)
  n = len(doc2)
  result = []
  i,j = 0,0
  while i<m and j<n:
    if doc1[i] == doc2[j]:
      result.append(doc1[i])
      i+=1
      j+=1
    elif doc1[i] < doc2[j]:
      result.append(doc1[i])
      i+=1
    else:
      result.append(doc2[j])
      j+=1
  while i < m:
    result.append(doc1[i])
    i+=1
  while j < n:
    result.append(doc2[j])
    j+=1
  return result
def mergeBlocks(self,blockA,blockB):
  termsA = list(blockA.keys())
  termsB = list(blockB.keys())
  termsA.sort()
  termsB.sort()
```

```
    m = len(termsA)
    n = len(termsB)
    result = {}
    i,j = 0,0
    while i < m  and j <n :
      if termsA[i] == termsB[j]:
        result[termsA[i]]                                                    =
self.mergeDocs(blockA[termsA[i]],blockB[termsB[j]])
        i+=1
        j+=1
      elif termsA[i] < termsB[j]:
        result[termsA[i]] =blockA[termsA[i]]
        i+=1
      else:
        result[termsB[j]] = blockB[termsB[j]]
        j+=1
    while  i < m:
      result[termsA[i]] = blockA[termsA[i]]
      i+=1
    while  j < n:
      result[termsB[j]] = blockB[termsB[j]]
      j+=1
    return result
  def blockSortBasedIndexing(self):
    n = len(self.blocks)
    self.BSBI = self.mergeBlocks(self.blocks['block1'],{})
    for i in range(2,n+1):
      self.BSBI = self.mergeBlocks(self.BSBI,self.blocks[f'block{i}'])
```

Output:

obj.makeBlock()

obj.blocks

{'block1': {'a': ['doc1', 'doc2'], 'hare': ['doc1'], 'wa': ['doc1'], 'make': ['doc1'], 'fun': ['doc1'], 'of': ['doc1', 'doc2'], 'the': ['doc1', 'doc2'], 'tortois': ['doc1'], 'one': ['doc1', 'doc2'], 'day': ['doc1'], 'for': ['doc1', 'doc2'], 'be': ['doc1'], 'so': ['doc1'], 'slowdo': ['doc1'], 'you': ['doc1', 'doc2'], 'ever': ['doc1'], 'get': ['doc1'],…….. 'inde': ['doc6'], 'an': ['doc6'], 'eye': ['doc6'], 'upon': ['doc6'], 'these': ['doc6'], 'but': ['doc6'], 'he': ['doc6'], 'never': ['doc6'], 'find': ['doc6'], 'right': ['doc6'], 'opportun': ['doc6'], 'make': ['doc6'], 'hi': ['doc6'], 'prey': ['doc6'], 'day': ['doc6'], 'fight': ['doc6'], 'among': ['doc6'], 'themselv': ['doc6'], 'separ': ['doc6'], 'own': ['doc6'], 'way': ['doc6'], 'got': ['doc6'], 'know': ['doc6'], 'about': ['doc6'], 'so': ['doc6'], 'plan': ['doc6'], 'trick': ['doc6'], 'all': ['doc6'], 'found': ['doc6'], 'attack': ['doc6'], 'hid': ['doc6'], 'behind': ['doc6'], 'bush': ['doc6'], 'wait': ['doc6'], 'for': ['doc6'], 'lone': ['doc6'], 'came': ['doc6'], 'closer': ['doc6'], 'pounc': ['doc6'], 'meal': ['doc6']}}

---

obj.blockSortBasedIndexing()

obj.BSBI

{'a': ['doc1', 'doc10', 'doc2', 'doc3', 'doc4', 'doc5', 'doc6', 'doc7', 'doc8', 'doc9'], 'abl': ['doc3'], 'about': ['doc10', 'doc2', 'doc6', 'doc8'], 'accept': ['doc3'], …….'your': ['doc10', 'doc8'], 'youso': ['doc8']}

7. Implement SPIMI algorithm

```
def singlePassInMemoryIndexing(self,corpus):
  path = self.corpus+'/dictionary.json'
  for title in self.titles:
    if len(self.SPIMI)==0:
      for word in sorted(list(set(self.termDocs[title]))):
        self.SPIMI[word] = [title]
    else:
      for word in sorted(list(set(self.termDocs[title]))):
        if word in self.SPIMI:
          self.SPIMI[word].append(title)
        else:
          self.SPIMI[word] = [title]
```

Output:

```
obj.singlePassInMemoryIndexing("/content/drive/MyDrive/stories")
```

{'a': ['doc1', 'doc2', 'doc3', 'doc4', 'doc7', 'doc8', 'doc9', 'doc10', 'doc5', 'doc6'], 'act': ['doc1', 'doc3'], 'after': ['doc1', 'doc3', 'doc4', 'doc8'], 'agre': ['doc1', 'doc10'], 'amus': ['doc1'], 'and': ['doc1', 'doc2', 'doc3', 'doc4', 'doc7', 'doc8', 'doc9', 'doc10', 'doc5', 'doc6'], ,…… 'ago': ['doc6'], 'among': ['doc6'], 'attack': ['doc6'], 'closer': ['doc6'], 'danger': ['doc6'], 'eye': ['doc6'], 'fight': ['doc6'], 'four': ['doc6'], 'graze': ['doc6'], 'inde': ['doc6'], 'lone': ['doc6'], 'opportun': ['doc6'], 'pounc': ['doc6'], 'prey': ['doc6'], 'save': ['doc6'], 'separ': ['doc6'], 'tiger': ['doc6'], 'uniti': ['doc6']}

8. Implement Dynamic indexing

```
class MyEngine:
    def __init__(self,corpus):
        ps = PorterStemmer()
        self.corpus = corpus
        self.document = []
        self.data = {}
        self.dictionary = {}
        self.id = 0
        self.doc_id = 0
        self.model = {}
        self.model['documents'] ={}
        self.model['data'] = {}
        self.model['postings'] = {}
        for file in os.listdir(corpus):
            self.model['data'][file[:-4]] = []
            self.model['documents'][file[:-4]] = {}
            self.model['documents'][file[:-4]]['id'] = self.doc_id
            self.model['documents'][file[:-4]]['status'] = True
            self.doc_id += 1
            document =
self.removeAllTheSpecialCharacter(open(self.corpus+file,encoding='utf-
8').read()).lower().split()
            for word in document:
                word = ps.stem(re.sub(r'[^a-zA-Z0-9]', '',word ))
                if word not in self.dictionary:
                    self.dictionary[word] =self.id
```

```
                self.model['postings'][word] =
[self.model['documents'][file[:-4]]['id']]
                self.id += 1
            elif self.model['documents'][file[:-4]]['id'] not in
self.model['postings'][word]:

self.model['postings'][word].append(self.model['documents'][file[:-
4]]['id'])
            self.model['data'][file[:-4]].append(self.dictionary[word])
        words_length = len(self.dictionary)
        for word in self.model['postings']:
            self.model['postings'][word].sort()
            prev = self.model['postings'][word][0]
            n = len(self.model['postings'][word])
            for i in range(1,n):
                self.model['postings'][word][i] -= prev
                prev = self.model['postings'][word][i]

        with open('stories.json','w') as fp:

            json.dump(self.model,fp)
    def removeAllTheSpecialCharacter(self,word):
        special_characters =
[".",";","!","@","#","$","%","^","&","*","(",")","{","}","[","]",":","'","/","
<",">","","+"]
        return "".join(filter(lambda char: char not in special_characters ,
word))
obj = MyEngine('stories/')
print(obj.model)
```

Ouput:

{"documents":{"america":{"id":0,"status":true},"animals":{"id":1,"status":true},"carnivorous":{"id":2,"status":true},"fishes":{"id":3,"status":true},"herbivorous":{"id":4,"status":true},"humans":{"id":5,"status":true},"india":{"id":6,"status":true},"mammals":{"id":7,"status":true},"omnivourous":{"id":8,"status":true},"trees":{"id":9,"status":true}},"data":{"america":[0,1,2,3,4,5,6,5,7,8,9,0,1,2,10,6,10,6,4,11,12,13,14,15,16,17,4,18,19,3,20,2,12,21,22,23,24,25,26,27,28,29,30,31,32,33,34,0,1,2,11,35,16,36,37,38,39,40,41,42,2,0,21,2,3,43,0,44,41,31,0,45,3,46,18,11,0,47,48,13,49,50,51,31,52,53,18,54,51,55,38,56,57,18,17,31,38,58,57,18,59,18,60,61,55,38,0,62,63,64,31,65,66,38,12,67,3,68,69,70,18,11,0,71,67,13,16,0,4,31,0,72,71,67,16,0,47,0,73,74,11,75,76,31,0,71,67,77,31,78,79,11,80,81,77],"animals":[82,83,84,85,86,16,0,87,88,89,38,90,91,82,92,86,93,94,95,83,96,57,97,98,99,100,31,101,102,103,104,105,16,106,107,108,19,3,12,],"group":[2,1,4,5],"formal":[2],"refer":[2],"carnivoran":[2],"evolv":[2,7],"special":[2],"eat":[2,2,6],"flesh":[2],"fifth":[2],"largest":[2],"compris":[2],"at":[2],"least":[2],"fish":[3],"aquat":[3],"craniat":[3],"gillbear":[3],"lack":[3],"limb":[3],"digit":[3],"includ":[3,6],"definit":[3,6],"hagfish":[3],"lamprey":[3],"cartilagin":[3],"boni":[3],"well":[3],"variou":[3],"extinct":[3],"relat":[3],"approxim":[3],"rayfin":[3],"belong":[3],"class":[3,4],"actinopterygii":[3],"those":[3],"be":[3,6],"teleost":[3],"herbivor":[4],"anatom":[4],"physiolog":[4],"adapt":[4],"plant":[4,4,5],"for":[4,3,6],"exampl":[4],"foliag":[4]……..way":[9],"tower":[9],"sunlight":[9],"angiosperm":[9],"hardwood":[9],"rest":[9],"gymnosperm":[9],"softwood":[9],"longliv":[9],"reach":[9],"sever":[9],"thousand":[9],"old":[9],"exist":[9],"trillion":[9],"matur":[9]}}

9. Implement vector space model with various functions.

```
import numpy as np
import pandas as pd
import os
import re
import json
from math import sqrt
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
class Vector:
    def __init__(self,corpus):
        self.corpus = corpus
        self.numberOfDocuments = len(os.listdir(self.corpus))
        self.dictionary = []
        self.documents = {}
        self.vectors = {}
        for document in os.listdir(self.corpus):
                                            self.documents[document[:-4]]        =
self.tokenize(open(self.corpus+"/"+document,encoding="utf-8").read())
        self.dictionary.sort()
        # for document in os.listdir(self.corpus):
        for document in self.documents:
            self.setVector(document)

    def tokenize(self,document):
        ps = PorterStemmer()
        document = re.sub(r'[^a-z A-Z]',"",document)
        document = document.lower().split()
```

```
        n = len(document)
        for i in range(n):
            # print(document)
            document[i] = ps.stem(document[i])
            if document[i] not in self.dictionary:
                self.dictionary.append(document[i])
        return document


    def setVector(self,document): # document name    document is list type no reutn type (0,2,4,1)
        ps = PorterStemmer()
        document_title = document
        document = sorted(self.documents[document])
        self.vectors[document_title] = []
        for token in self.dictionary:
            if token in document:
                self.vectors[document_title].append(document.count(token))
            else:
                self.vectors[document_title].append(0)


    def getVector(self,document)
        return self.vectors[document]
    def cosineSimilarity(self,query,document):

print(self.dotProduct(query,document),self.modVector(query),self.modVector(document),self.dotProduct(query,document)/(self.modVector(query)*self.modVector(document)))
```

```
        try:
                                                                    return
self.dotProduct(query,document)/(self.modVector(query)*self.modVector(d
ocument))
        except ZeroDivisionError:
            pass

    def dotProduct(self,vector1,vector2):
        l = 0
        for i in range(len(vector1)):
            l += (vector1[i]*vector2[i])
        return l

    def modVector(self,vector): #vector
        d = 0
        for i in vector:
            d += i**2
        return sqrt(d)
    def convert_vector(self,query):
        query = query.lower()
        query = sorted(query.split())
        vector = []
        ps = PorterStemmer()
        for i in range(len(query)):
            query[i] = ps.stem(query[i])
        for token in self.dictionary:
            if token in query:
                vector.append(query.count(token))
            else:
```

```
            vector.append(0)
        return vector
    def getDocument(self,query):
        doc = None
        query = self.convert_vector(query)
        d = -1
        try:
            for document in self.vectors:
                if d < self.cosineSimilarity(query,self.getVector(document)):
                    # print(d)
                    d = self.cosineSimilarity(query,self.getVector(document))
                    doc = document
            return doc
        except TypeError:
            return "Sorry! No documents retrived"
obj = Vector("documents")
print(obj.getDocument("tree"))
```

Output:

Tree

10. Implement Naïve Bayes classification algorithm

```
import pandas as pd
from collections import Counter
class Bayes:
    def __init__(self,csv_file):
        self.data = pd.read_excel(csv_file)
        self.tables = {}
        self.table_names = self.data.columns[1:-1]
        self.predict = self.data.columns[-1]
        self.predicted_values = self.data[self.predict].unique()
        for name in self.table_names:
            self.createTable(name)
    def createTable(self,table_name):
        unique = self.data[table_name].unique()
        c = 0
        self.tables[table_name] =
pd.DataFrame(data=None,index=unique,columns=self.predicted_values)
        for row in self.predicted_values: #no yes
            for col in unique: #rows
                self.tables[table_name][row][col] = len(
            self.data[(self.data[table_name]==col)&
(self.data[self.predict]==row)])/len(self.data[self.data[self.predict]==row])
    def classifyQuery(self,query):
        query = query.split()
        count = dict(Counter(query))
        query = list(count.keys())
        result = 0
        m = 0
```

```
        try:
            for c in self.predicted_values:
                a                                                    =
len(self.data[self.data[self.predict]==c])/len(self.data[self.predict])
                for k,i in zip(self.tables,query):
                    a *= (self.tables[k][c][i]**count[i])
                if m<a:
                    m = a
                    result = c
            print("The data predict : ",result)
        except KeyError:
            print("Unable to predict result")
            print("Please Provide relavant data / check splelling mistakes..... ")
        finally:
            print("Successfully executed")
obj = Bayes('data.xlsx')
obj.classifyQuery("Rain Mild High Strong")
```

Output:

```
The data predict : No
Successfully executed
```

11. Implement K-Means algorithm

```
from random import random
class Kmeans:
    def __init__(self,docIds,n):
        self.numberOfClusters = n
        self.documents = docIds
        self.clusters = {}
        for i in range(self.numberOfClusters):
            try:
                self.clusters[f'cluster {i+1}'] = [[]

,docIds[i]]
            except IndexError:
                print("Enter valid number of clusters.... ")
    def mean(self,s):
        return sum(s)/len(s)
    def distance(self,docId,mean):
        return abs(docId-mean)
    def clusturing(self):
        while True:
            present = {}
            for cluster in self.clusters:
                present[cluster] = []
            for docId in self.documents:
                m = 1000
                c = ""
                for cluster in self.clusters:
```

```
                if m > self.distance(docId,self.clusters[cluster][-1]):
                    m = self.distance(docId,self.clusters[cluster][-1])
                    c = cluster
            present[c].append(docId)
        if present[c] == self.clusters[c][0]:
            break
        else:
            for cluster in self.clusters:
                self.clusters[cluster][0] = present[cluster]
                self.clusters[cluster][1] = self.mean(present[cluster])
l = [1,2,3,4,50,60,70,80,90,100,110]
n = int(input("Enter number of clusters : "))
obj = Kmeans(l,n)
obj.clusturing()
for cluster in obj.clusters:
    print(cluster ," : ",obj.clusters[cluster][0])
```

Output:

```
Enter number of clusters : 2
Cluster 1 : [1,2,3,4]
Cluster 2 : [50,60,70,80,90,100,110]
```