Section 1 [dataset] : Corpus details and source

Source:https://www.kaggle.com/datasets/amritvirsinghx/environmental-news-nlp-dataset

Due to the global pandemic situation, various lockdowns have been imposed in different countries, this data contains short snippets of news from 2017 to Jan 2020.

Different news sources are present in this dataset. Consisting of a total of 418 different files that included, CNN, BBCNEWS, FOXNEWS and MSNBC.

Section 2 [Requirements] : Set of free text test queries, wild card queries, Phrase queries

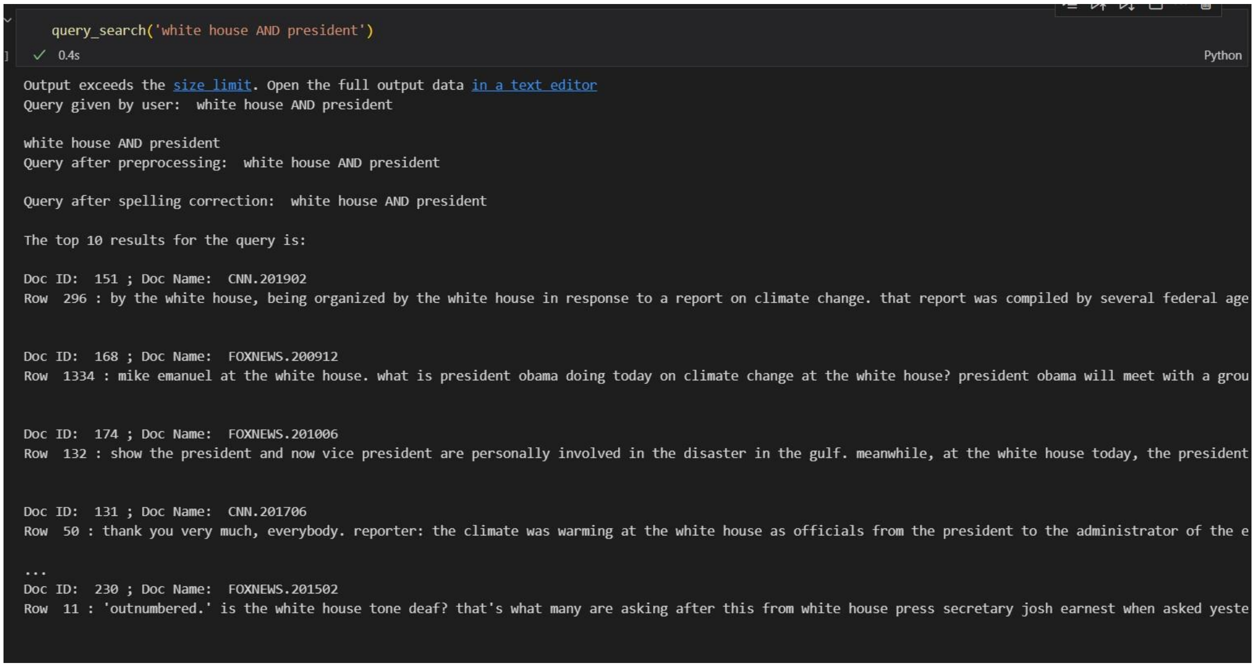
- **Boolean Retrieval**
  Query given by user:
  white house AND president
  Pollution and Emission
- **Phrase Query**
  Query given by user:
  pollution from cars
  Carbon dioxide in atmosphere
- **Wild Card Query** Query given by user:
  *poll*n*

Section 3 [Design]: Data structures used with brief reasons and similarity scheme

The data structures we have used include:

- Lists
- Dictionaries
- Arrays
- Similarity scheme - Cosine similarity

Section 4.1 [Result of Boolean Retrieval] :on free text queries

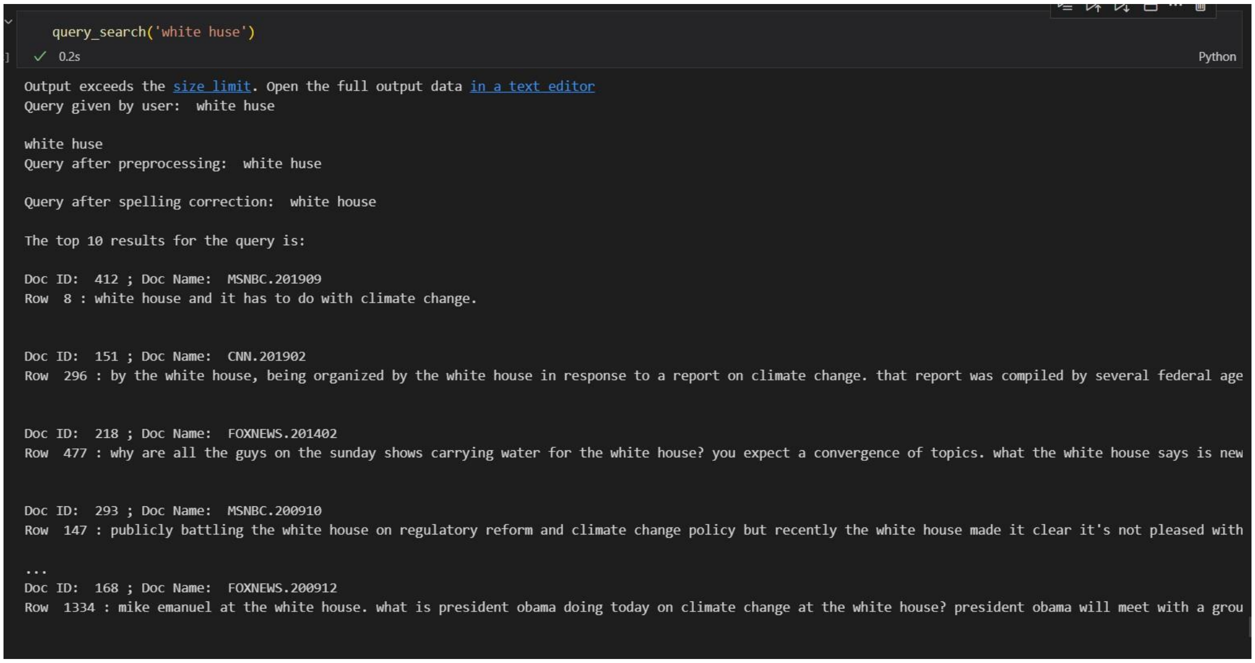


```
query_search('white house AND president')
```

```
Output exceeds the size limit. Open the full output data in a text editor
Query given by user:  white house AND president

white house AND president
Query after preprocessing:  white house AND president

Query after spelling correction:  white house AND president

The top 10 results for the query is:

Doc ID:  151 ; Doc Name:  CNN.201902
Row  296 : by the white house, being organized by the white house in response to a report on climate change. that report was compiled by several federal age


Doc ID:  168 ; Doc Name:  FOXNEWS.200912
Row  1334 : mike emanuel at the white house. what is president obama doing today on climate change at the white house? president obama will meet with a grou


Doc ID:  174 ; Doc Name:  FOXNEWS.201006
Row  132 : show the president and now vice president are personally involved in the disaster in the gulf. meanwhile, at the white house today, the president


Doc ID:  131 ; Doc Name:  CNN.201706
Row  50 : thank you very much, everybody. reporter: the climate was warming at the white house as officials from the president to the administrator of the e

...
Doc ID:  230 ; Doc Name:  FOXNEWS.201502
Row  11 : 'outnumbered.' is the white house tone deaf? that's what many are asking after this from white house press secretary josh earnest when asked yeste
```

The above example tests boolean queries using inverted index.

Section 4.2 [Result with inverted index] : on free text queries with rank



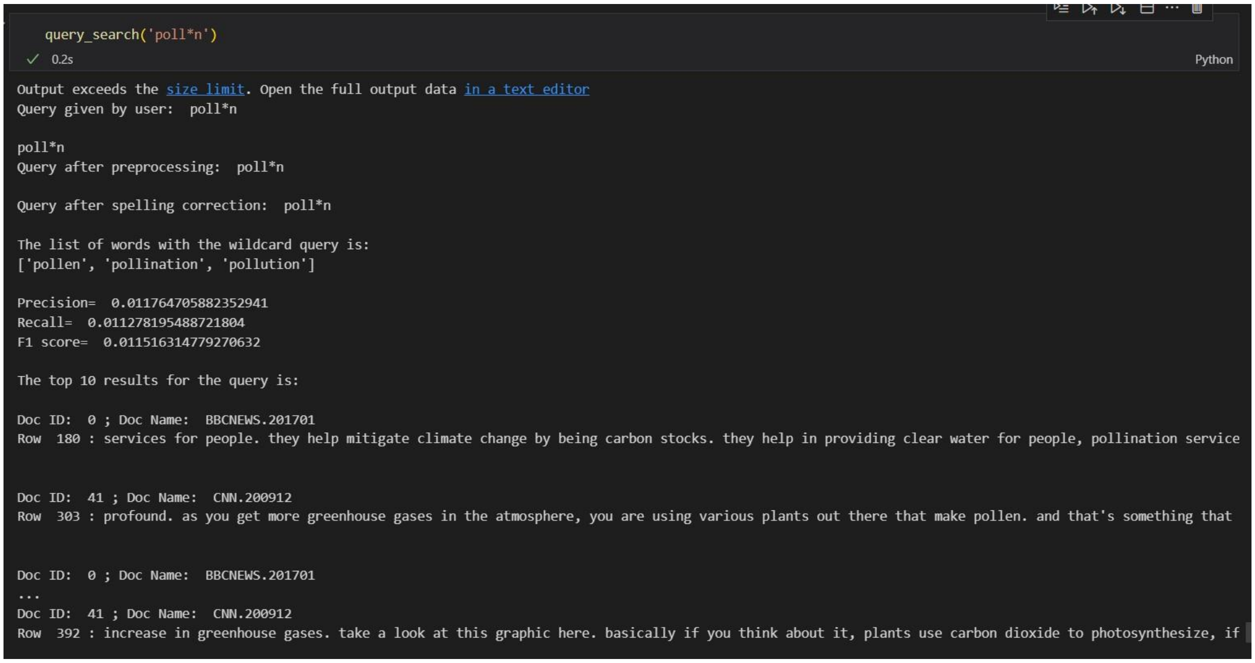
```
query_search('white huse')
```

```
Output exceeds the size limit. Open the full output data in a text editor
Query given by user:  white huse

white huse
Query after preprocessing:  white huse

Query after spelling correction:  white house

The top 10 results for the query is:

Doc ID:  412 ; Doc Name:  MSNBC.201909
Row  8 : white house and it has to do with climate change.


Doc ID:  151 ; Doc Name:  CNN.201902
Row  296 : by the white house, being organized by the white house in response to a report on climate change. that report was compiled by several federal age


Doc ID:  218 ; Doc Name:  FOXNEWS.201402
Row  477 : why are all the guys on the sunday shows carrying water for the white house? you expect a convergence of topics. what the white house says is new


Doc ID:  293 ; Doc Name:  MSNBC.200910
Row  147 : publicly battling the white house on regulatory reform and climate change policy but recently the white house made it clear it's not pleased with

...
Doc ID:  168 ; Doc Name:  FOXNEWS.200912
Row  1334 : mike emanuel at the white house. what is president obama doing today on climate change at the white house? president obama will meet with a grou
```

Here we use inverted index to retrieve all documents which have news related to “white huse”

Note: We have intentionally given “white huse” to show that our code performs spelling correction

Section 4.3 [Result of Wild Card queries] :

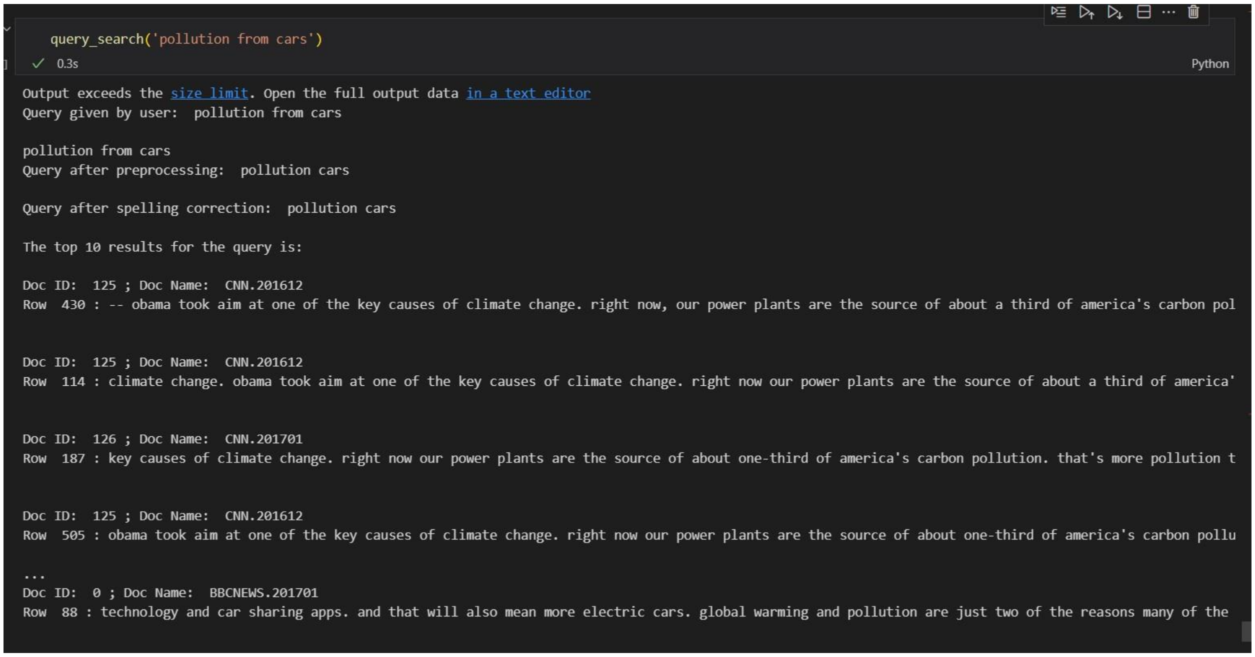


```
query_search('poll*n')
```

```
Output exceeds the size limit. Open the full output data in a text editor
Query given by user:  poll*n

poll*n
Query after preprocessing:  poll*n

Query after spelling correction:  poll*n

The list of words with the wildcard query is:
['pollen', 'pollination', 'pollution']

Precision=  0.011764705882352941
Recall=  0.011278195488721804
F1 score=  0.011516314779270632

The top 10 results for the query is:

Doc ID:  0 ; Doc Name:  BBCNEWS.201701
Row  180 : services for people. they help mitigate climate change by being carbon stocks. they help in providing clear water for people, pollination service


Doc ID:  41 ; Doc Name:  CNN.200912
Row  303 : profound. as you get more greenhouse gases in the atmosphere, you are using various plants out there that make pollen. and that's something that


Doc ID:  0 ; Doc Name:  BBCNEWS.201701
...
Doc ID:  41 ; Doc Name:  CNN.200912
Row  392 : increase in greenhouse gases. take a look at this graphic here. basically if you think about it, plants use carbon dioxide to photosynthesize, if
```

This is an example for wildcard queries, when entered poll*n, the matching words are displayed.Then it prints all documents in with the words appear.

Section 4.4 [Result of Phrase queries]:



```
query_search('pollution from cars')
✓ 0.3s
```

```
Output exceeds the size limit. Open the full output data in a text editor
Query given by user:  pollution from cars

pollution from cars
Query after preprocessing:  pollution cars

Query after spelling correction:  pollution cars

The top 10 results for the query is:

Doc ID:  125 ; Doc Name:  CNN.201612
Row  430 : -- obama took aim at one of the key causes of climate change. right now, our power plants are the source of about a third of america's carbon pol


Doc ID:  125 ; Doc Name:  CNN.201612
Row  114 : climate change. obama took aim at one of the key causes of climate change. right now our power plants are the source of about a third of america'


Doc ID:  126 ; Doc Name:  CNN.201701
Row  187 : key causes of climate change. right now our power plants are the source of about one-third of america's carbon pollution. that's more pollution t


Doc ID:  125 ; Doc Name:  CNN.201612
Row  505 : obama took aim at one of the key causes of climate change. right now our power plants are the source of about one-third of america's carbon pollu

...
Doc ID:  0 ; Doc Name:  BBCNEWS.201701
Row  88 : technology and car sharing apps. and that will also mean more electric cars. global warming and pollution are just two of the reasons many of the
```

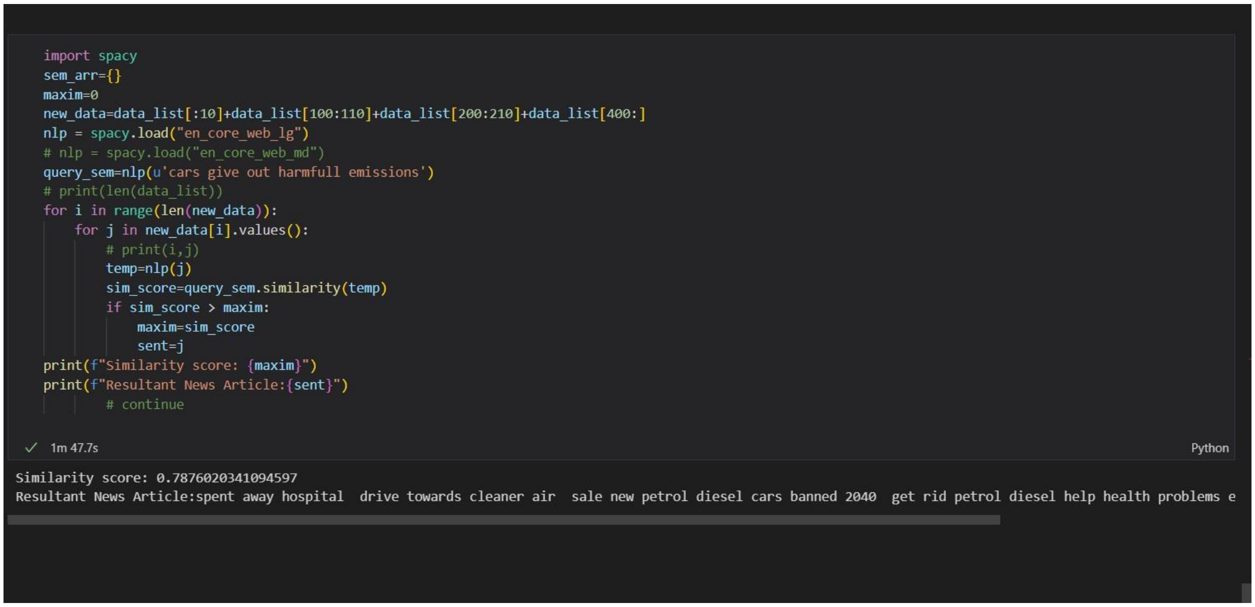
This is an example for phrase queries this retrieval uses Biword and positional index to retrieve all the document ids and text from those documents which have similar content as the queries

Section 4.5 [how the evaluator can test]: an arbitrary text query relevant to your corpus ?

The evaluator cannot enter any arbitrary query to search our corpus however under the semantic matching section he can enter any arbitrary query and we will return the most similar text document i.e the text document having the highest score when compared with out query

Section 4.6 [Any one additional functionality} : relevance feedback, semantic matching, re-ranking of results, and finding out query intention.

Semantic matching has been implemented in our code where if a user enters a query and the code returns the semantically highest scoring text document from our corpus.



```
import spacy
sem_arr={}
maxim=0
new_data=data_list[:10]+data_list[100:110]+data_list[200:210]+data_list[400:]
nlp = spacy.load("en_core_web_lg")
# nlp = spacy.load("en_core_web_md")
query_sem=nlp(u'cars give out harmfull emissions')
# print(len(data_list))
for i in range(len(new_data)):
    for j in new_data[i].values():
        # print(i,j)
        temp=nlp(j)
        sim_score=query_sem.similarity(temp)
        if sim_score > maxim:
            maxim=sim_score
            sent=j
print(f"Similarity score: {maxim}")
print(f"Resultant News Article:{sent}")
        # continue
```

```
Similarity score: 0.7876020341094597
Resultant News Article:spent away hospital  drive towards cleaner air  sale new petrol diesel cars banned 2040  get rid petrol diesel help health problems e
```

Code:

```
#!/usr/bin/env python
# coding: utf-8


# ## **AIR Assignment - Team 24**
# ## Dataset : Environmental News NLP Dataset
#
#
#
# #
---



# #### Importing all the required packages


# In[1]:



from textblob import TextBlob
from nltk.corpus import stopwords

from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from collections import Counter
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

```
import pandas as pd

import os

import math

import copy
import nltk

import string

import glob

import zipfile


get_ipython().system('pip install nltk')


nltk.download('stopwords')
nltk.download('wordnet')

nltk.download('punkt')



# #### Extract the data from the zip file


# In[3]:



path_to_zip_file = "archive.zip"

with zipfile.ZipFile(path_to_zip_file, 'r') as zip_ref:

    zip_ref.extractall("content")



# #### Extract and store the data row-wise in a list
# In[2]:



docid_doc = {}
doc_no=0



alldata=['' for i in range(418)]

data_list = []



for k in glob.glob("content/TelevisionNews/*.csv"):

  nam = k.split("/")
  print(nam)
```

```
  docid_doc[doc_no] = nam[-1][:-4].split("\\")[-1]
#dictionary with mapping from document ID to document name
  print(docid_doc)
  doc_no+=1
  data = pd.read_csv(k)
  # print(data.head(5))
  column_names = data.columns[:]
  row_data={}
  row_no = 0
  for index, row in data.iterrows():
    content=''
    content=str(row[column_names[-1]])
    row_data[row_no] = content                #storing data row wise
    row_no+=1
  data_list.append(row_data)                  #documents with
dictionary of row_no and content in row
  data_list
print("Corpus is\n")
data_list
```

```
print(data_list[0].values())


# #### Pre-Process the corpus - Converting to lower case


# In[3]:



for i in range(len(data_list)):                  #iterating through
documents
  for j in data_list[i].keys():                  #iterating through
rows
    data_list[i][j]=data_list[i][j].lower()
data_list

data_list[0][0]


# #### Stopwords removal


# In[4]:
```

```


stopwords = list(stopwords.words('english'))

stopwords.append("i\'ve")

stopwords.append("i\'m")
stopwords.remove("no")

stopwords.remove("not")

stopwords.remove("than")

stopwords.remove("which")



def remove_stopwords(text):
    """custom function to remove the stopwords"""
    return " ".join([word for word in str(text).split() if word not
in stopwords])


for i in range(len(data_list)):
  for j in data_list[i].keys():
    data_list[i][j] = remove_stopwords(data_list[i][j])
data_list[0][0]



# #### Punctuation Removal


# In[5]:



punctuation='[")?,\!(}:{;$%^&]/<>=#_.-–'
for ele in punctuation:
  for i in range(len(data_list)):
    for j in data_list[i].keys():
      data_list[i][j] = data_list[i][j].replace(ele, ' ') #Replace
the punctuation symbols with a space

for i in range(len(data_list)):
    for j in data_list[i].keys():
      data_list[i][j]=data_list[i][j].replace("'s",' ')
      data_list[i][j] =data_list[i][j].replace("'re",' ')
      data_list[i][j] =data_list[i][j].replace("'",' ')      #Replace
the apostrophe characters with NULL
```

```
data_list[0][0]
```

```
# #### Lemmatization


# In[6]:


import nltk

nltk.download('omw-1.4')

lemmatizer = WordNetLemmatizer()

lemma=[]

corpus = {} for i in
range(len(data_list)):

  corpus[i]= []

  for j in data_list[i].keys():

    lem=''

    datalist=data_list[i][j].split(' ')

    for k     datalist:
    in
      lem+=' '+lemmatizer.lemmatize(k,pos='a')       #Lemmatize the

words to     root word by bringing context to it.
the
    if(i      in range(len(lemma))):
    not
      lemma.append({})

    lemma[i][j] = lem
    corpus[i].append(lem)                                # Adds the
lemmatized words into corpus



#lemma

corpus[0][0]



# #### Tokenization


# In[7]:


doc_id = 0

token_in_doc ={}

list_of_tokens = []

doc_rows = {}
```

```
for i in range(len(lemma)):
  doc_rows[doc_id] = []                            #doc_rows is
dictionary with document id and list of rows in the corpus.
  for j in lemma[i].keys():
    doc_rows[i].append(j)

    if(doc_id not in token_in_doc):
      token_in_doc[doc_id]={}
    a = word_tokenize(lemma[i][j])                 # Tokkenizes the
sentences into individual words
    token_in_doc[doc_id][j] = a                    #token_words is a dict
which doc_id and tokens mapping exists
    list_of_tokens += a

  doc_id+=1
```

```
list_of_tokens[0:20]


# #### Finding top 100 biwords

# In[8]:


bigrams = zip(list_of_tokens, list_of_tokens[1:])
counts = Counter(bigrams)                         #finds the common
bigrams in the corpus a =
counts.most_common(100)
# print(a)
common_biwords = []
for i in a:
    s = i[0][0]+" "+i[0][1]
    common_biwords.append(s)                     # common biwords is
tuple of biwords and its frequency
print(common_biwords)


# #### Building Inverted Index Dictionary


# In[9]:
```

```
inverted_index_dictionary={}
term_id = 1
term_termids = {}

for i in token_in_doc.keys():            #iterate through the
documents
  for k in token_in_doc[i]:              #iterate through the rows
```

```
    for j in range(len(token_in_doc[i][k])): #iterate through the
tokens
      term = token_in_doc[i][k][j]
      if(term not in term_termids):     #To add new term in the
inverted index
        term_termids[term] = term_id
        term_id+=1
        inverted_index_dictionary[term_id-1] = {'docs':{},
'doc_freq':1}
        inverted_index_dictionary[term_id-1]['docs'][i]=
{'pos':{k:[j]}, 'term_freq' :1 }
      else:                            #To append the rows and
postiion for the existing term in the index
        if(i not in
inverted_index_dictionary[term_termids[term]]['docs']):
          inverted_index_dictionary[term_termids[term]]['docs'][i] =
{'pos':{}, 'term_freq' :0 }

inverted_index_dictionary[term_termids[term]]['doc_freq']+=1
        if(k not in
inverted_index_dictionary[term_termids[term]]['docs'][i]['pos']):

inverted_index_dictionary[term_termids[term]]['docs'][i]['pos'][k] =
[]

inverted_index_dictionary[term_termids[term]]['docs'][i]['pos'][k].a
ppend(j)

inverted_index_dictionary[term_termids[term]]['docs'][i]['term_freq'
]+=1
    #biword indexing - to index the dictionary instead of the corpus
if biword for m in
    common_biwords:
```

```
      b = m.split(" ")
      for j in range(0,len(token_in_doc[i][k])-1):
        term1 = token_in_doc[i][k][j]
        term2 = token_in_doc[i][k][j+1]
        if(term1 == b[0] and term2 == b[1]):
          term = term1+" "+term2
          if(term not in term_termids):
            term_termids[term] = term_id
            term_id+=1
            inverted_index_dictionary[term_id-1] = {'docs':{},
'doc_freq':1}
            inverted_index_dictionary[term_id-1]['docs'][i]=
{'pos':{k:[j]}, 'term_freq' :1 }
          else:
            if(i not in
inverted_index_dictionary[term_termids[term]]['docs']):

inverted_index_dictionary[term_termids[term]]['docs'][i] =
{'pos':{}, 'term_freq' :0 }

inverted_index_dictionary[term_termids[term]]['doc_freq']+=1
            if(k not in
inverted_index_dictionary[term_termids[term]]['docs'][i]['pos']):

inverted_index_dictionary[term_termids[term]]['docs'][i]['pos'][k] =
[]

inverted_index_dictionary[term_termids[term]]['docs'][i]['pos'][k].a
ppend(j)
```

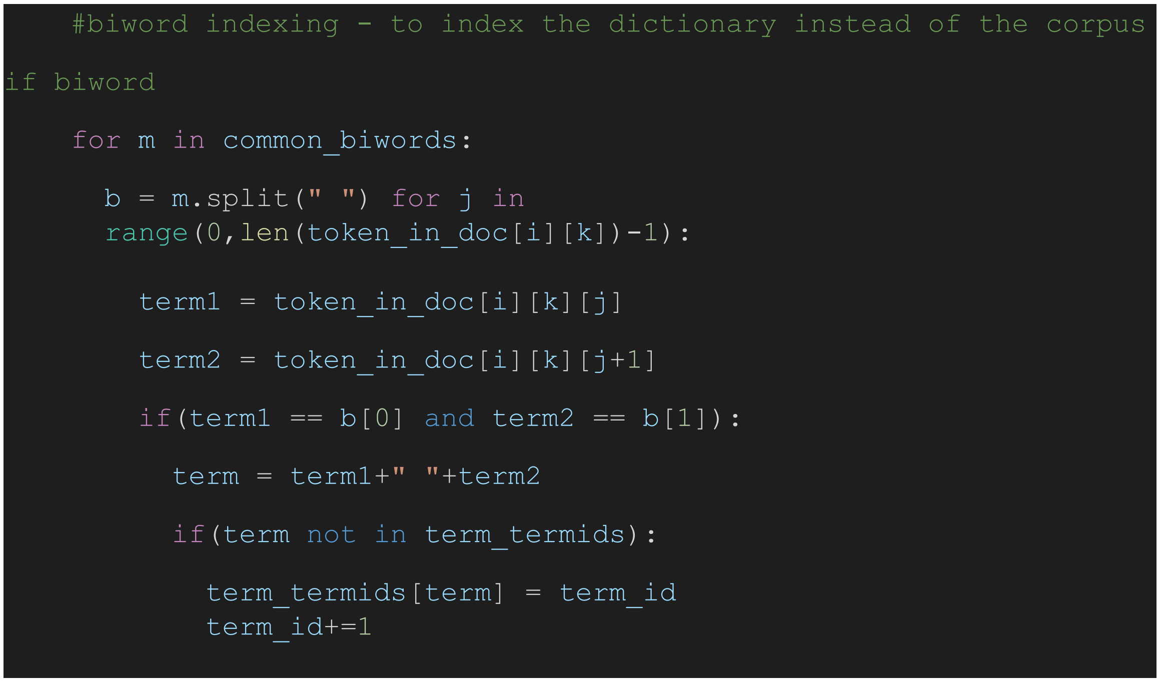
```
inverted_index_dictionary[term_termids[term]]['docs'][i]['term_freq'
]+=1



inverted_index_dictionary[term_termids['prime minister']]


# #### Building Inverted Index B-Tree
```

```
# In[10]:



# Create node
class BTreeNode:

    def __init__(self, leaf=False):

        self.leaf = leaf

        self.keys = []

        self.child = []




class BTree:
    def __init__(self, t):

        self.root = BTreeNode(True)

        self.t = t


    #search for the key in the B-tree

    def search_key(self, k, x=None):
        if x is not None:
            i = 0
            while i < len(x.keys) and k > x.keys[i][0]:
                i += 1
            if i < len(x.keys) and k == x.keys[i][0]:
                return (x.keys,i)
            elif x.leaf:
                return None
            else:
                return self.search_key(k, x.child[i])
        else:
            return self.search_key(k, self.root)


    # Insert the key : The node contains the tuple with term id and posting list
    def insert_key(self, k):
        root = self.root
        if len(root.keys) == (2 * self.t) - 1:
            temp = BTreeNode()
            self.root = temp
            temp.child.insert(0, root)
            self.split(temp, 0)
```

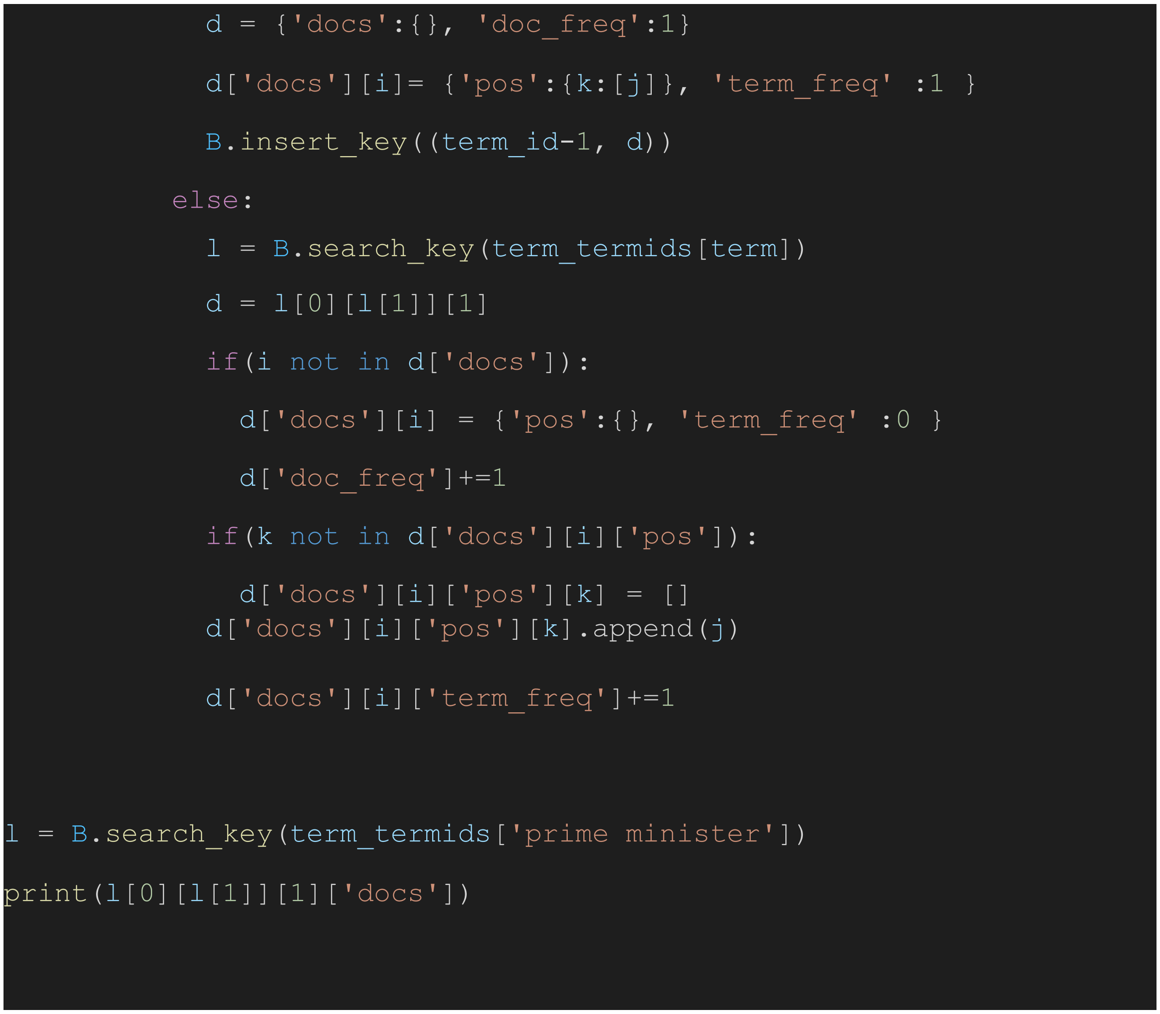
```
            self.insert_non_full(temp, k)
        else:
            self.insert_non_full(root, k)


    # Insert non full condition
    def insert_non_full(self, x, k):
        i = len(x.keys) - 1
        if x.leaf:
            x.keys.append((None, None))
            while i >= 0 and k[0] < x.keys[i][0]:

                x.keys[i + 1] = x.keys[i]
                i -= 1

            x.keys[i + 1] = k

        else:

            while i >= 0 and k[0] < x.keys[i][0]:

                i -= 1

            i += 1

            if len(x.child[i].keys) == (2 * self.t) - 1:

                self.split(x, i)

                if k[0] > x.keys[i][0]:

                    i += 1

            self.insert_non_full(x.child[i], k)



    #Split function- splits the node and self balances it if the 
keys in the node reaches the limit
    def split(self, x, i):

        t = self.t

        y = x.child[i]

        z = BTreeNode(y.leaf)

        x.child.insert(i + 1, z)

        x.keys.insert(i, y.keys[t - 1])

        z.keys = y.keys[t: (2 * t) - 1]

        y.keys = y.keys[0: t - 1]

        if not y.leaf:

            z.child = y.child[t: 2 * t]

            y.child = y.child[0: t ]
```

```
B = BTree(3) #3 indicate the degree of the each node
```

```
#Building inverted index
term_id = 1
term_termids = {}

for i in token_in_doc.keys():           #iterate through the documents
  for k in token_in_doc[i]:             #iterate through the rows
    for j in range(len(token_in_doc[i][k])):#j is the index of tokens
      term = token_in_doc[i][k][j]
      if(term not in term_termids):     #To add new term in the inverted index
        term_termids[term] = term_id
        term_id+=1
        d = {'docs':{}, 'doc_freq':1}
        d['docs'][i]= {'pos':{k:[j]}, 'term_freq' :1 }
        B.insert_key((term_id-1, d))
      else:                             #To append the rows and postiion for the existing term in the index
        l = B.search_key(term_termids[term])
        d = l[0][l[1]][1]
        if(i not in d['docs']):
          d['docs'][i] = {'pos':{}, 'term_freq' :0 }
          d['doc_freq']+=1
        if(k not in d['docs'][i]['pos']):
          d['docs'][i]['pos'][k] = []
        d['docs'][i]['pos'][k].append(j)
        d['docs'][i]['term_freq']+=1
```



```
    #biword indexing - to index the dictionary instead of the corpus
if biword
    for m in common_biwords:
      b = m.split(" ") for j in
      range(0,len(token_in_doc[i][k])-1):
        term1 = token_in_doc[i][k][j]
        term2 = token_in_doc[i][k][j+1]
        if(term1 == b[0] and term2 == b[1]):
          term = term1+" "+term2
          if(term not in term_termids):
            term_termids[term] = term_id
            term_id+=1
```



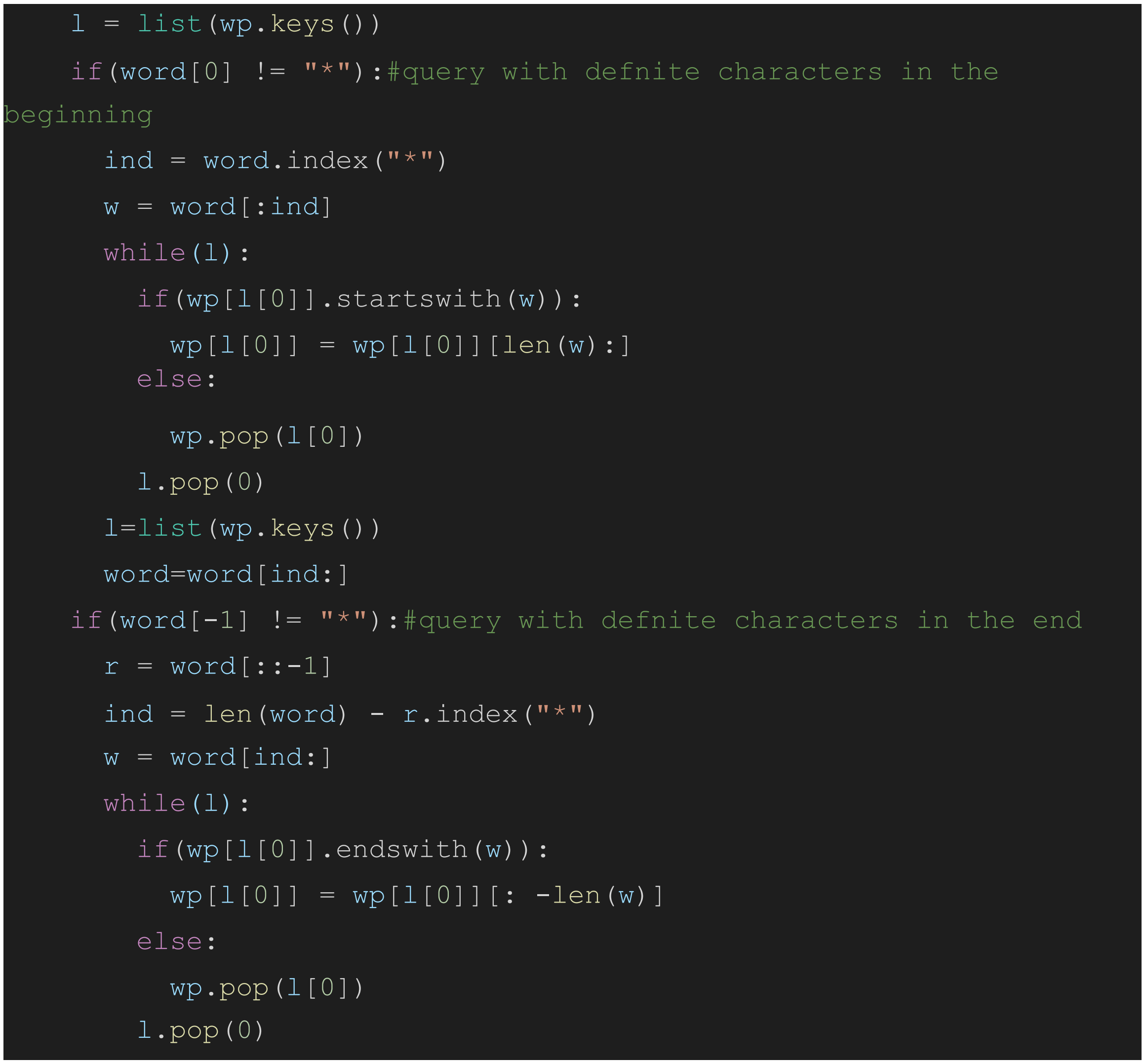
```
            d = {'docs':{}, 'doc_freq':1}
            d['docs'][i]= {'pos':{k:[j]}, 'term_freq' :1 }
            B.insert_key((term_id-1, d))
          else:
            l = B.search_key(term_termids[term])
            d = l[0][l[1]][1]
            if(i not in d['docs']):
              d['docs'][i] = {'pos':{}, 'term_freq' :0 }
              d['doc_freq']+=1
            if(k not in d['docs'][i]['pos']):
              d['docs'][i]['pos'][k] = []
            d['docs'][i]['pos'][k].append(j)
            d['docs'][i]['term_freq']+=1


l = B.search_key(term_termids['prime minister'])
print(l[0][l[1]][1]['docs'])
```

```
# #### Search - Single word query


# In[11]:



# One word query search using B-tree
def word_search_Btree(word):
  doc = B.search_key(term_termids[word])
  ans = {}
  for j in doc[0][doc[1]][1]['docs']:
    ans[j] = []
    for k in doc[0][doc[1]][1]['docs'][j]['pos']:
      ans[j].append(k)
  return ans

#One word query search using Dictionary
def word_search_dictionary(word):
  doc = inverted_index_dictionary[term_termids[word]]
  ans = {}
  for j in doc['docs']:
    ans[j] = [] for k in
    doc['docs'][j]['pos']:
```

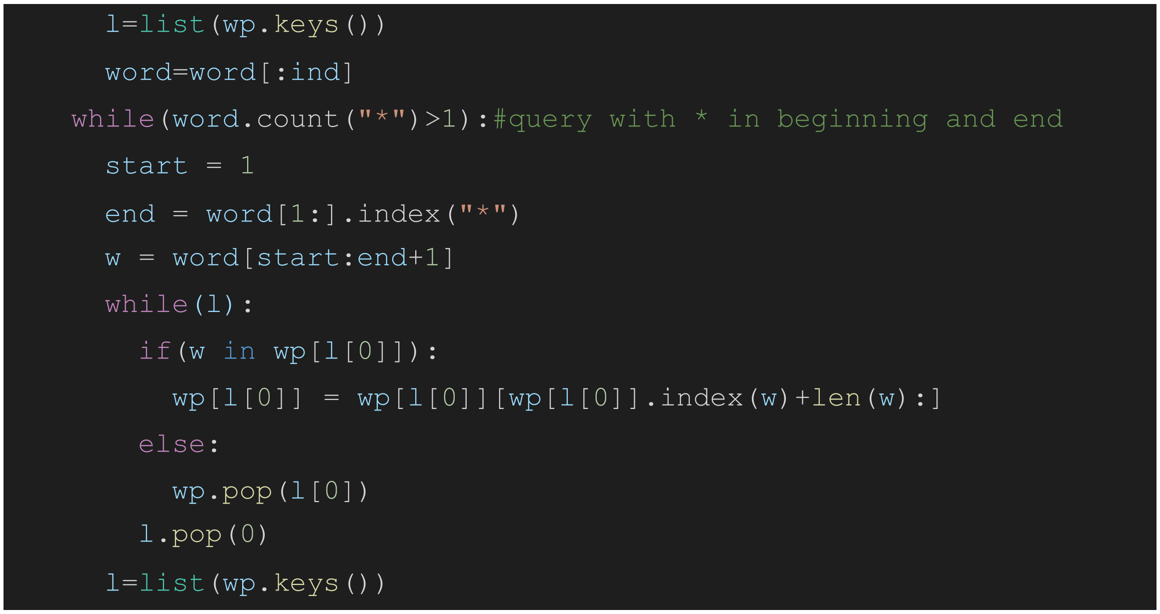
```
      ans[j].append(k)
  return ans



ans1 = word_search_Btree('money') ans2
= word_search_dictionary('money')

print("Posting list retrieved from B-Tree")
print(ans1)

print("POsting list retrieved from dictionary")

print(ans2)




# #### K-Gram Generation


# In[12]:



#K-gram generation K=2 for wildcard query

k_gram = {} #k-grams with list of termids where that k gram is occuring

def generateNGrams(word1, n):

  word ='$'+word1+'$'

  for i in range(0,len(word)-(n-1)):

    gram = word[i:i+n]

    if(gram in k_gram):

      j=0
      c=0

      while(j<len(k_gram[gram])):

        if(word1 < k_gram[gram][j]):

          k_gram[gram].insert(j,word1)

          c=1

          break

        else:
          j+=1

      if(c==0):

        k_gram[gram].append(word1)
    else:
      k_gram[gram] = [word1]

for i in term_termids:
```

```
  generateNGrams(i,2)   #Generate kgrams for k=2 for the corpus
k_gram



# #### Wildcard Query Search

# In[30]:


global all_wildcard_words
all_wildcard_words = {}

def intersection(d):#finding the intersection of words with all the k-grams present
  l=list(d.keys())#dictionary with k-gram and list of words with k-gram
  ans=d[l[0]]
  for k in range(1,len(l)):
    i=0
    j=0
    temp=[]
    while(i<len(ans) and j<len(d[l[k]])):
      if(ans[i] == d[l[k]][j]):
        temp.append(ans[i])
        i+=1
        j+=1
      elif(ans[i]>d[l[k]][j]):
        j+=1
      else:
        i+=1
    ans = temp
  return ans

def post_filtering(word,l , a): #postfiltering to remove any false positives
  temp=[]
  if(a==1):#with only one *
    if(word[0] == "*"):#for search with suffix query
      word=word[1:]
      for i in l:
```

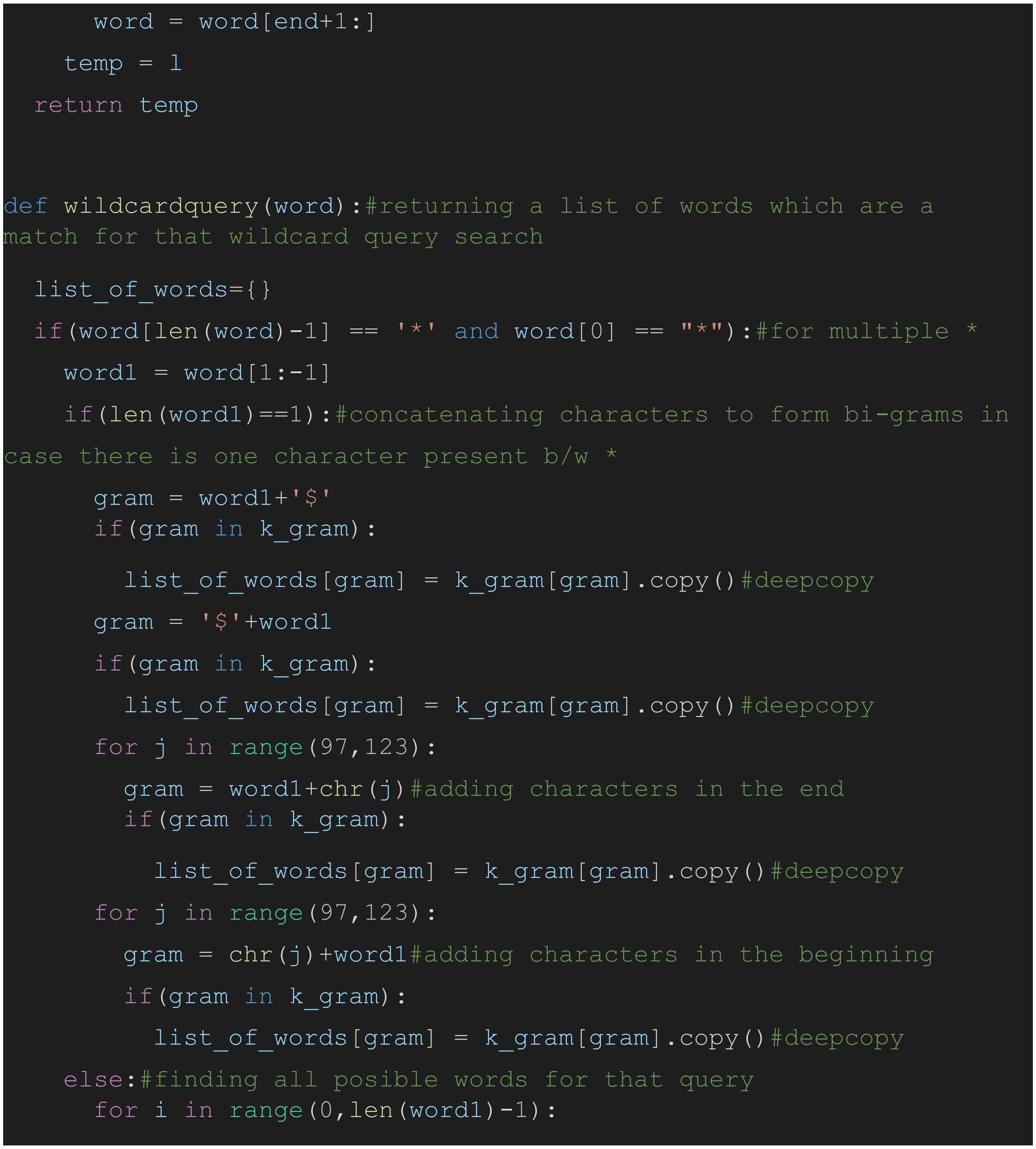
```
        if(i.endswith(word)):
          temp.append(i)
    elif(word[len(word)-1]=="*"):#for search with prefix query
      word=word[:-1]
      for i in l:
        if(i.startswith(word)):
          temp.append(i)
    else:
      word = word.split("*")#for search with * in the middle
      for i in l:
        if(i.startswith(word[0]) and i.endswith(word[1])):
          temp.append(i)
  else:
    wp = {}#dictionary needed for post_filtering for query with multiple *
    for i in l:
      wp[i] = i
```



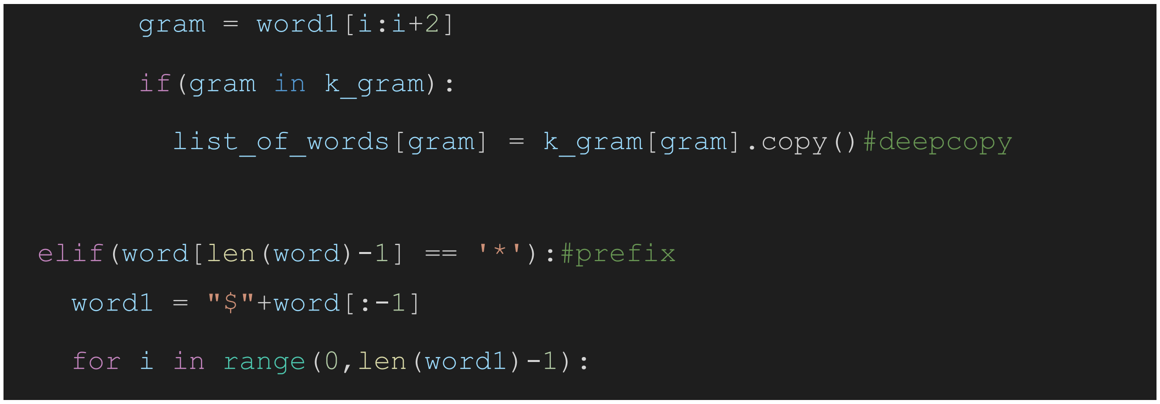
```
    l = list(wp.keys())
    if(word[0] != "*"):#query with defnite characters in the beginning
      ind = word.index("*")
      w = word[:ind]
      while(l):
        if(wp[l[0]].startswith(w)):
          wp[l[0]] = wp[l[0]][len(w):]
        else:
          wp.pop(l[0])
        l.pop(0)
      l=list(wp.keys())
      word=word[ind:]
    if(word[-1] != "*"):#query with defnite characters in the end
      r = word[::-1]
      ind = len(word) - r.index("*")
      w = word[ind:]
      while(l):
        if(wp[l[0]].endswith(w)):
          wp[l[0]] = wp[l[0]][: -len(w)]
        else:
          wp.pop(l[0])
        l.pop(0)
```



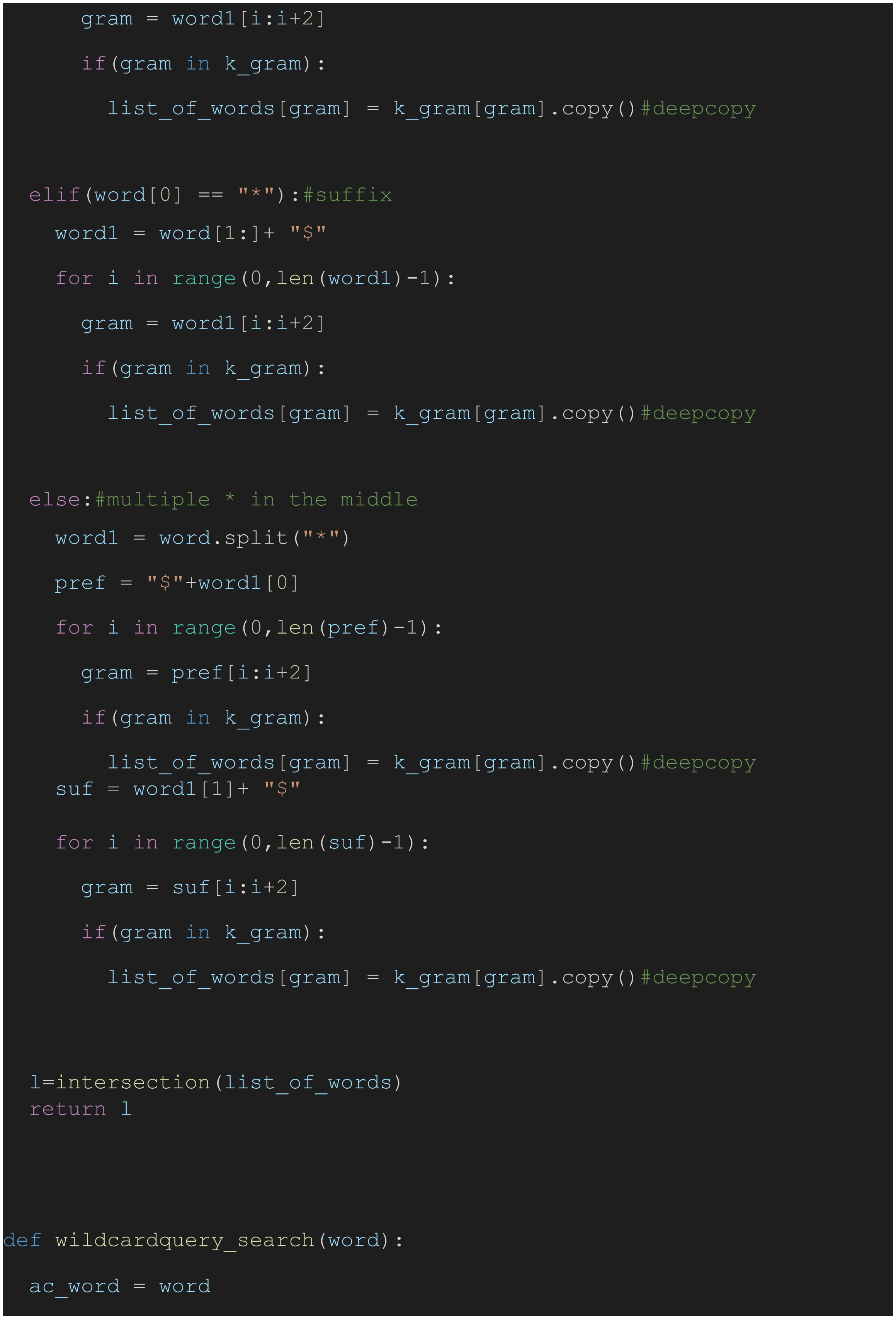
```
      l=list(wp.keys())
      word=word[:ind]
    while(word.count("*")>1):#query with * in beginning and end
      start = 1
      end = word[1:].index("*")
      w = word[start:end+1]
      while(l):
        if(w in wp[l[0]]):
          wp[l[0]] = wp[l[0]][wp[l[0]].index(w)+len(w):]
        else:
          wp.pop(l[0])
        l.pop(0)
      l=list(wp.keys())
```



```
      word = word[end+1:]
    temp = 1
  return temp


def wildcardquery(word):#returning a list of words which are a
match for that wildcard query search
  list_of_words={}
  if(word[len(word)-1] == '*' and word[0] == "*"):#for multiple *
    word1 = word[1:-1]
    if(len(word1)==1):#concatenating characters to form bi-grams in
case there is one character present b/w *
      gram = word1+'$'
      if(gram in k_gram):
        list_of_words[gram] = k_gram[gram].copy()#deepcopy
      gram = '$'+word1
      if(gram in k_gram):
        list_of_words[gram] = k_gram[gram].copy()#deepcopy
      for j in range(97,123):
        gram = word1+chr(j)#adding characters in the end
        if(gram in k_gram):
          list_of_words[gram] = k_gram[gram].copy()#deepcopy
      for j in range(97,123):
        gram = chr(j)+word1#adding characters in the beginning
        if(gram in k_gram):
          list_of_words[gram] = k_gram[gram].copy()#deepcopy
    else:#finding all posible words for that query
      for i in range(0,len(word1)-1):
```



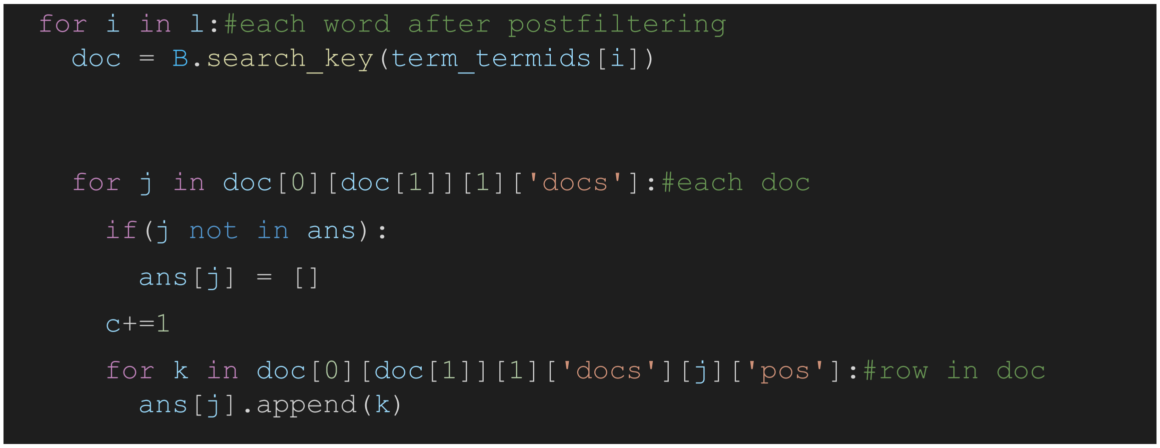
```
        gram = word1[i:i+2]
        if(gram in k_gram):
          list_of_words[gram] = k_gram[gram].copy()#deepcopy

  elif(word[len(word)-1] == '*'):#prefix
    word1 = "$"+word[:-1]
    for i in range(0,len(word1)-1):
```



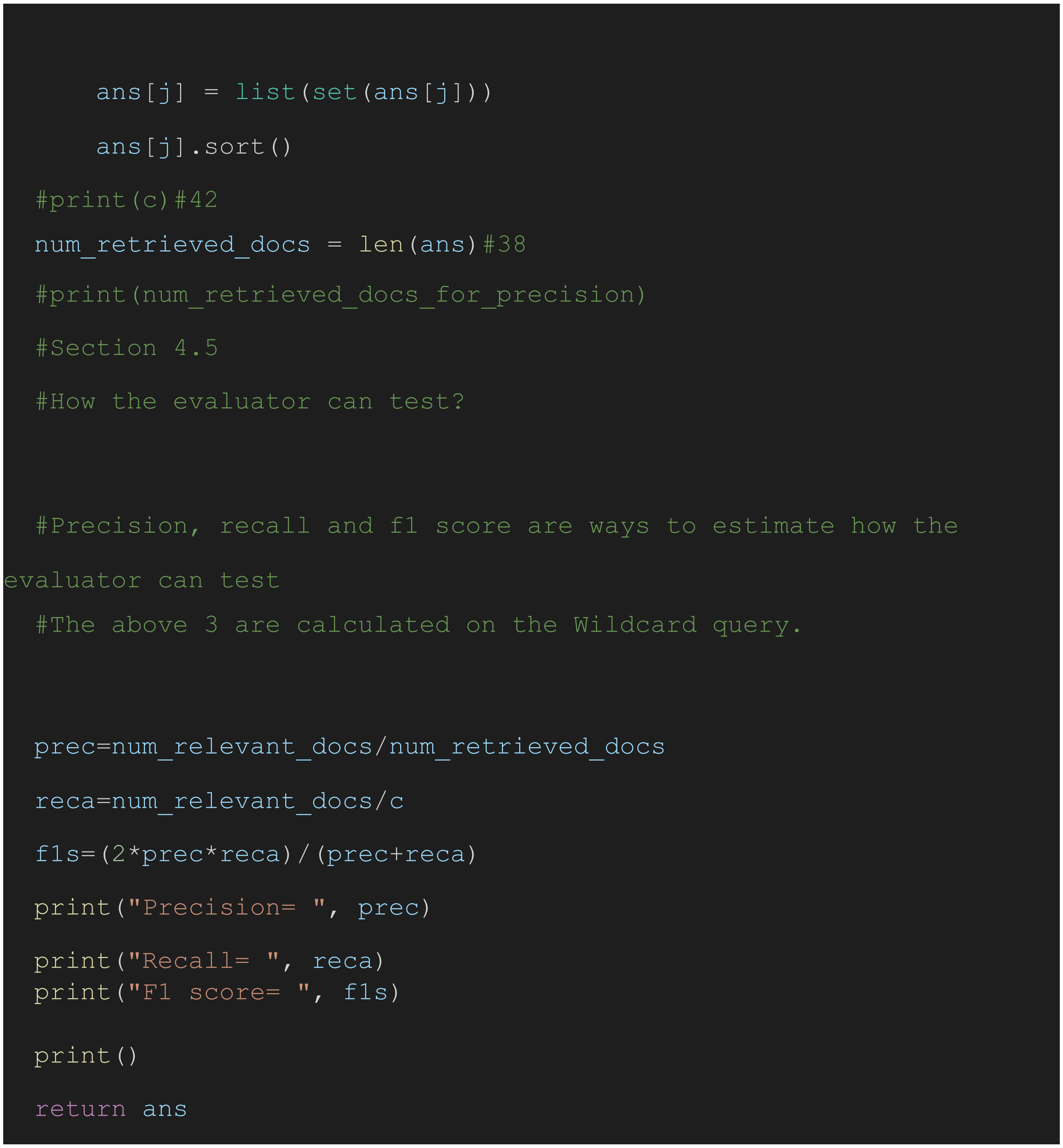
```
      gram = word1[i:i+2]
      if(gram in k_gram):
        list_of_words[gram] = k_gram[gram].copy()#deepcopy

  elif(word[0] == "*"):#suffix
    word1 = word[1:]+ "$"
    for i in range(0,len(word1)-1):
      gram = word1[i:i+2]
      if(gram in k_gram):
        list_of_words[gram] = k_gram[gram].copy()#deepcopy

  else:#multiple * in the middle
    word1 = word.split("*")
    pref = "$"+word1[0]
    for i in range(0,len(pref)-1):
      gram = pref[i:i+2]
      if(gram in k_gram):
        list_of_words[gram] = k_gram[gram].copy()#deepcopy
    suf = word1[1]+ "$"

    for i in range(0,len(suf)-1):
      gram = suf[i:i+2]
      if(gram in k_gram):
        list_of_words[gram] = k_gram[gram].copy()#deepcopy


  l=intersection(list_of_words)
  return l



def wildcardquery_search(word):

  ac_word = word
```

```
l=[]

if(word.count('*')==1): l
  = wildcardquery(word)
```

```
  l = post_filtering(word, l , 1)
else:
  #with multiple *
  word1 = word
  if(word[0]!='*'):#suffix type
    ind = word.index("*")
    temp = word[:ind+1]
    l+= wildcardquery(temp)
    word = word[ind:]
  if(word[-1] !="*"):#prefix type
    r = word[::-1]
    ind = len(word) - r.index("*")
    temp = word[ind-1:]
    l+= wildcardquery(temp)
    word = word[:ind]
  while(word.count('*') > 1):
    start = 0
    end = word[1:].index("*")
    temp = word[start:end+2]

    l+= wildcardquery(temp)
    word = word[end+1:]
  l=post_filtering(word1, l , 0)
print("The list of words with the wildcard query is:")
print(l)#words after post-filtering
print()
num_relevant_docs = len(l) #35
#print(num_relevant_docs_for_precision)
global all_wildcard_words
all_wildcard_words[ac_word]= l
ans = {}
c=0
```



```
  for i in l:#each word after postfiltering
    doc = B.search_key(term_termids[i])


    for j in doc[0][doc[1]][1]['docs']:#each doc
      if(j not in ans):
        ans[j] = []
      c+=1
      for k in doc[0][doc[1]][1]['docs'][j]['pos']:#row in doc
        ans[j].append(k)
```



```


      ans[j] = list(set(ans[j]))

      ans[j].sort()

  #print(c)#42

  num_retrieved_docs = len(ans)#38

  #print(num_retrieved_docs_for_precision)

  #Section 4.5

  #How the evaluator can test?



  #Precision, recall and f1 score are ways to estimate how the

evaluator can test
  #The above 3 are calculated on the Wildcard query.



  prec=num_relevant_docs/num_retrieved_docs

  reca=num_relevant_docs/c

  f1s=(2*prec*reca)/(prec+reca)

  print("Precision= ", prec)

  print("Recall= ", reca)
  print("F1 score= ", f1s)

  print()

  return ans
```

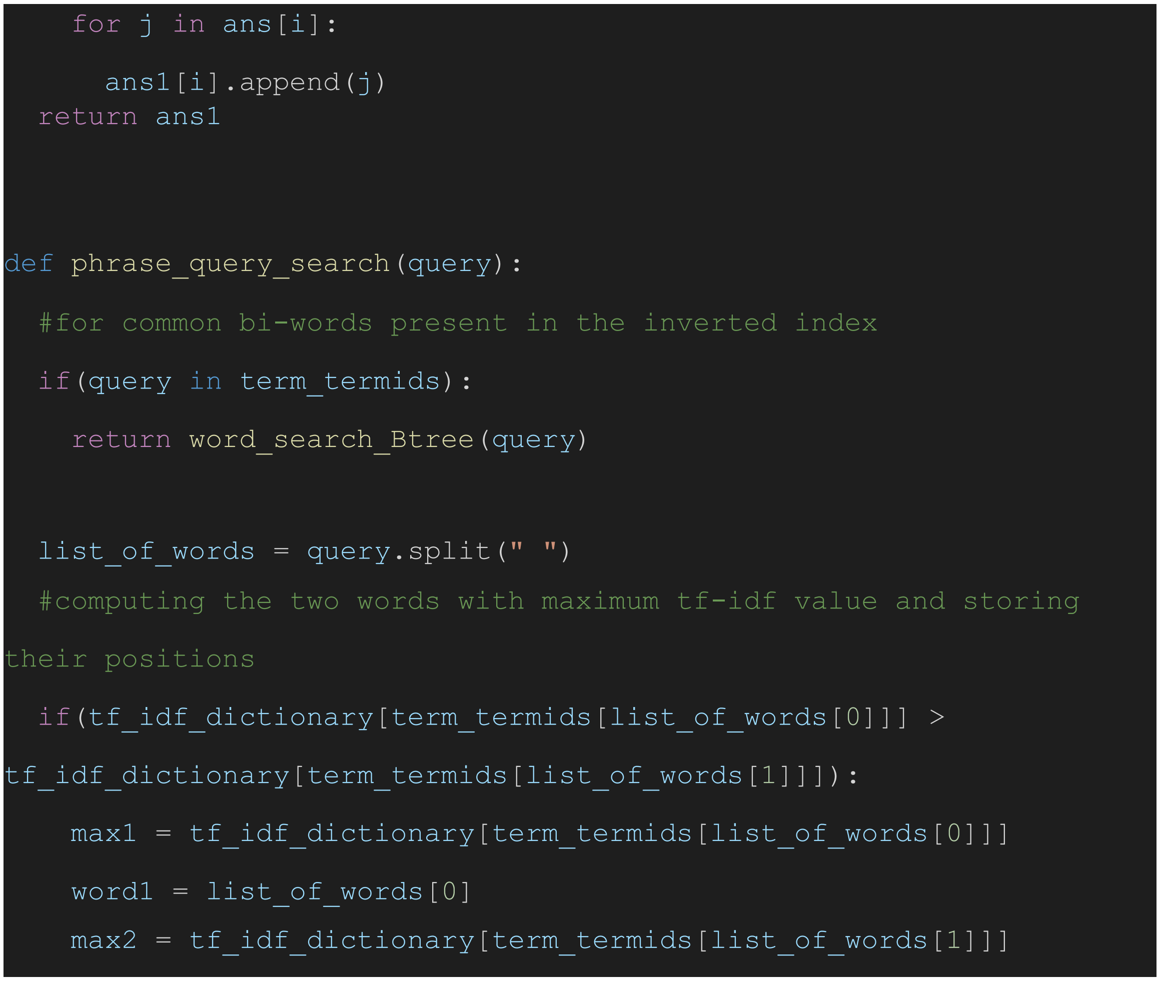
```
ans = wildcardquery_search('t*z*')
print('The results for c*li* is the following:')
print(ans)


global all_wildcard_words
all_wildcard_words={}



# #### TF-IDF Computation



# In[14]:




tf_idf_dictionary ={}



tf_idf_list=[]



for i in term_termids.keys():

```

```
    termcollection_freq=0
    for doc in inverted_index_dictionary[term_termids[i]]['docs']:


termcollection_freq+=inverted_index_dictionary[term_termids[i]]['docs'][doc]['term_freq']



    dfvalue=inverted_index_dictionary[term_termids[i]]['doc_freq']

    idfvalue=math.log(418/dfvalue)



    tfidf=termcollection_freq*idfvalue
    tf_idf_list.append(tfidf)




#adding tfidf to dictionary

for i in range(len(inverted_index_dictionary)):

    tf_idf_dictionary[i+1]=tf_idf_list[i]


print('The tf-idf of the term 
climate:',tf_idf_dictionary[term_termids['climate']])
print('The tf-idf of the term prime 
minister:',tf_idf_dictionary[term_termids['prime minister']])
#tf-idf of a common biword




# #### Search - Phrase query


# In[15]:




def phrase_search(phrase, word1,word2,k):#Modified Intersect For 
Proximity Constraint K
  list_of_words = [word1, word2]

  searched_words = {}

  ans = {}

  for i in list_of_words:

    l = B.search_key(term_termids[i])#finding the posting list of 
the two words
```

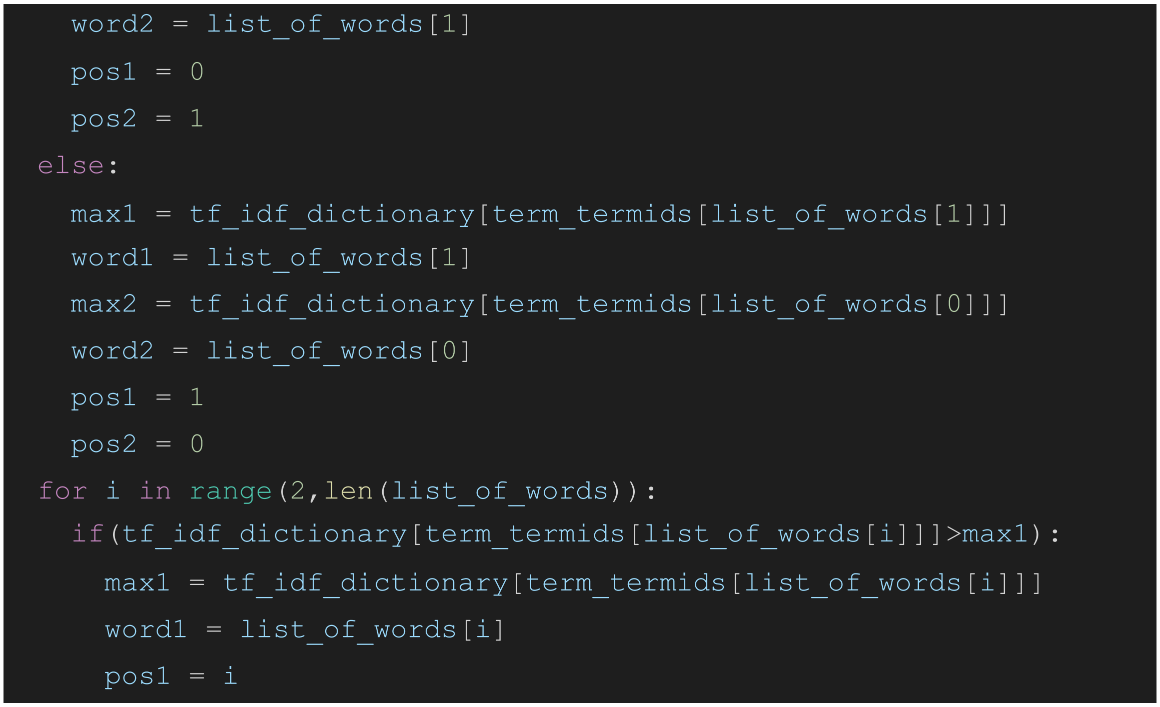
```
    searched_words[i] = copy.deepcopy(l[0][l[1]][1]['docs'])
  a = list(searched_words.keys())
```

```
  first = searched_words[a[0]]
  second = searched_words[a[1]]
  first_keys = list(first.keys())#document ids
  second_keys = list(second.keys())
  while(first_keys):#for each document
    if(first_keys[0] in second_keys):#first document id
      rows_first = list(first[first_keys[0]]['pos'].keys())#row ids
      rows_second = list(second[first_keys[0]]['pos'].keys())
      while(rows_first):#for each row
        if(rows_first[0] not in rows_second):#if rowid of first term
is not present in second term posting list
          first[first_keys[0]]['pos'].pop(rows_first[0])#remove
rowid from first
          if(first[first_keys[0]]['pos'] == {}):#remove documents
with no common rows
            first.pop(first_keys[0])
          rows_first.pop(0)
        else:
          x1=0
          flag=0
          while(x1<
len(searched_words[a[0]][first_keys[0]]['pos'][rows_first[0]]) and
flag==0):
            x2=0
            while(x2<
len(searched_words[a[1]][first_keys[0]]['pos'][rows_first[0]]) and
flag==0):
              #checking if both words occur in the window of size k

if(abs(searched_words[a[0]][first_keys[0]]['pos'][rows_first[0]][x1]
-searched_words[a[1]][first_keys[0]]['pos'][rows_first[0]][x2])<=k):
                if(first_keys[0] in ans):
                  ans[first_keys[0]].append(rows_first[0])
                else:
                  ans[first_keys[0]]=[rows_first[0]]
                flag =1#stops parsing through row when the first
matxh is found
              x2+=1
            x1+=1 val =
          rows_first.pop(0)
```

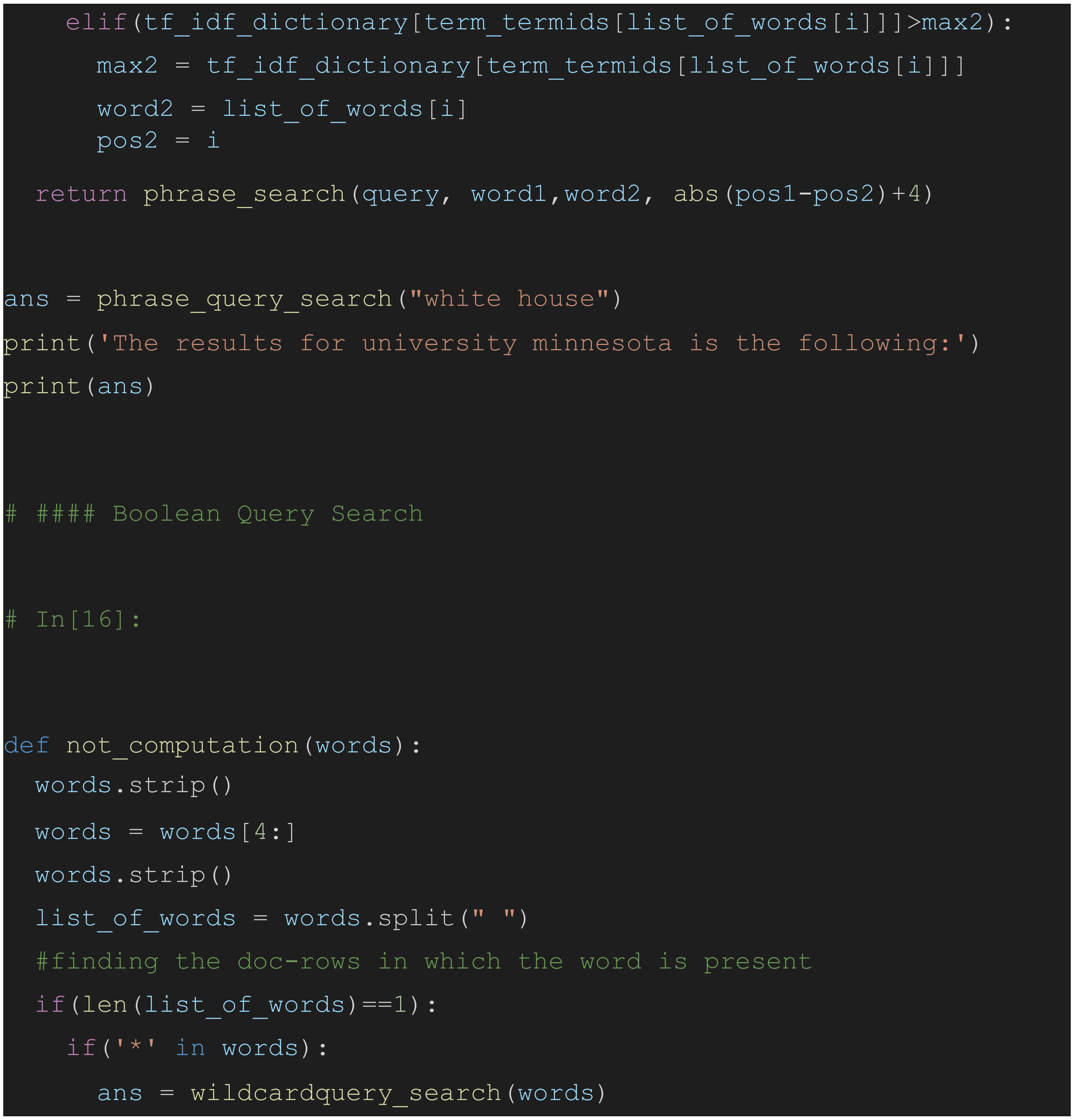
```
          rows_second.remove(val)


      while(rows_second):#deleting extra rows from second word's
posting list
        second[first_keys[0]]['pos'].pop(rows_second[0])
        if(second[first_keys[0]]['pos'] == {}):
            second.pop(first_keys[0])
        rows_second.pop(0)
      val = first_keys.pop(0)
      second_keys.remove(val)

    else:
      first.pop(first_keys[0])#deleting extra docs from first word's
posting list
      first_keys.pop(0)
  while(second_keys):#deleting extra docs from second word's posting
list
    second.pop(second_keys[0])
    second_keys.pop(0)

    first = second
  ans1 = {}
  for i in ans:
    ans1[i] = []
```



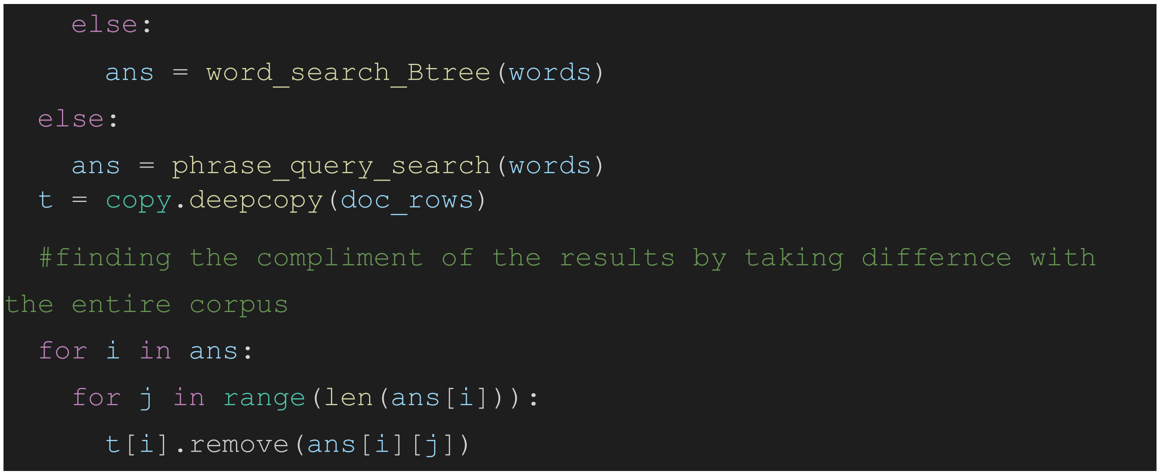
```
    for j in ans[i]:

      ans1[i].append(j)
  return ans1



def phrase_query_search(query):

  #for common bi-words present in the inverted index

  if(query in term_termids):

    return word_search_Btree(query)


  list_of_words = query.split(" ")

  #computing the two words with maximum tf-idf value and storing

their positions

  if(tf_idf_dictionary[term_termids[list_of_words[0]]] >

tf_idf_dictionary[term_termids[list_of_words[1]]]):

    max1 = tf_idf_dictionary[term_termids[list_of_words[0]]]

    word1 = list_of_words[0]

    max2 = tf_idf_dictionary[term_termids[list_of_words[1]]]
```



```
    word2 = list_of_words[1]

    pos1 = 0

    pos2 = 1

  else:

    max1 = tf_idf_dictionary[term_termids[list_of_words[1]]]

    word1 = list_of_words[1]

    max2 = tf_idf_dictionary[term_termids[list_of_words[0]]]

    word2 = list_of_words[0]

    pos1 = 1

    pos2 = 0

  for i in range(2,len(list_of_words)):

    if(tf_idf_dictionary[term_termids[list_of_words[i]]]>max1):

      max1 = tf_idf_dictionary[term_termids[list_of_words[i]]]

      word1 = list_of_words[i]

      pos1 = i
```



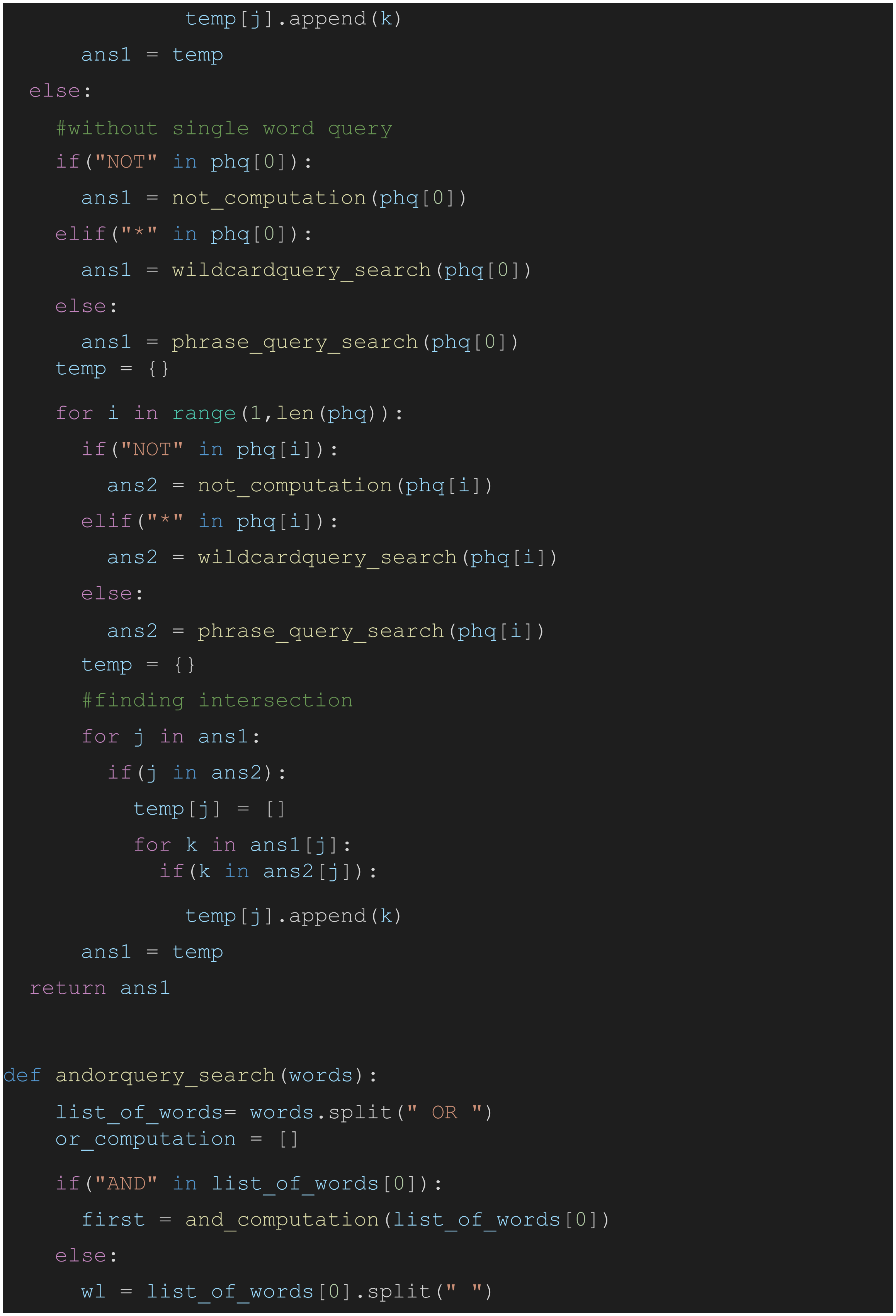
```
    elif(tf_idf_dictionary[term_termids[list_of_words[i]]]>max2):
      max2 = tf_idf_dictionary[term_termids[list_of_words[i]]]
      word2 = list_of_words[i]
      pos2 = i
  return phrase_search(query, word1,word2, abs(pos1-pos2)+4)


ans = phrase_query_search("white house")
print('The results for university minnesota is the following:')
print(ans)



# #### Boolean Query Search


# In[16]:



def not_computation(words):
  words.strip()
  words = words[4:]
  words.strip()
  list_of_words = words.split(" ")
  #finding the doc-rows in which the word is present
  if(len(list_of_words)==1):
    if('*' in words):
      ans = wildcardquery_search(words)
```



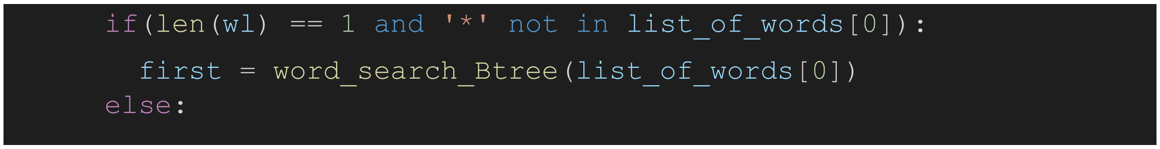
```
    else:
      ans = word_search_Btree(words)
  else:
    ans = phrase_query_search(words)
  t = copy.deepcopy(doc_rows)
  #finding the compliment of the results by taking differnce with
the entire corpus
  for i in ans:
    for j in range(len(ans[i])):
      t[i].remove(ans[i][j])
    if(t[i] == []):
      t.pop(i)
  return t


#intersection with optimization strategy
def and_computation(words):
  list_of_words= words.split(" AND ")
  docs_freq = {}#stores doc-freq mapping to list of words
  searched_words = {}
  phq = []
  for i in list_of_words:
    wl = i.split(" ")
    if(len(wl)==1 and '*' not in i):
      l = B.search_key(term_termids[i])#finding posting list of
one-word queries if(l[0][l[1]][1]['doc_freq'] in
      docs_freq):
        docs_freq[l[0][l[1]][1]['doc_freq']].append(i)
      else:
        docs_freq[l[0][l[1]][1]['doc_freq']] = [i]
      searched_words[i] = l[0][l[1]][1]
    else:
      phq.append(i)#adding wildcard query and phrase query to this
list


  k = list(docs_freq.keys())#finding the terms with least term freq
to find intersection
  k.sort()
  a = []
  for i in k:
    a+=docs_freq[i]#for intersection in ascending order of doc freq
```

```
  ans={}
  first = {}
  if(searched_words):
    for i in searched_words[a[0]]['docs']:
      first[i] =
list(searched_words[a[0]]['docs'][i]['pos'].keys())#storing posting
list of word with least doc freq
    for i in range(1,len(a)):
      second = {}
      for j in searched_words[a[i]]['docs']:
        second[j] =
list(searched_words[a[i]]['docs'][j]['pos'].keys())#storing posting
list of word from
      temp = {}
      #intersection
      for k in first:
        if(k in second):
          temp[k] = [] for l
          in first[k]:
            if(l in second[k]):
              temp[k].append(l)
      first = temp
    ans1 = {}
    for i in first:#changing the format to docid-list of rows from
posting list
      ans1[i] = []
      for j in first[i]:
        ans1[i].append(j)
    for i in phq:
      if("NOT" in i):
        ans2 = not_computation(i)
      elif("*" in i):
        ans2 = wildcardquery_search(i)
      else:
        ans2 = phrase_query_search(i)
      temp = {}#finding the intersection
      for j in ans1:
        if(j in ans2):
          temp[j] = [] for
          k in ans1[j]:
```

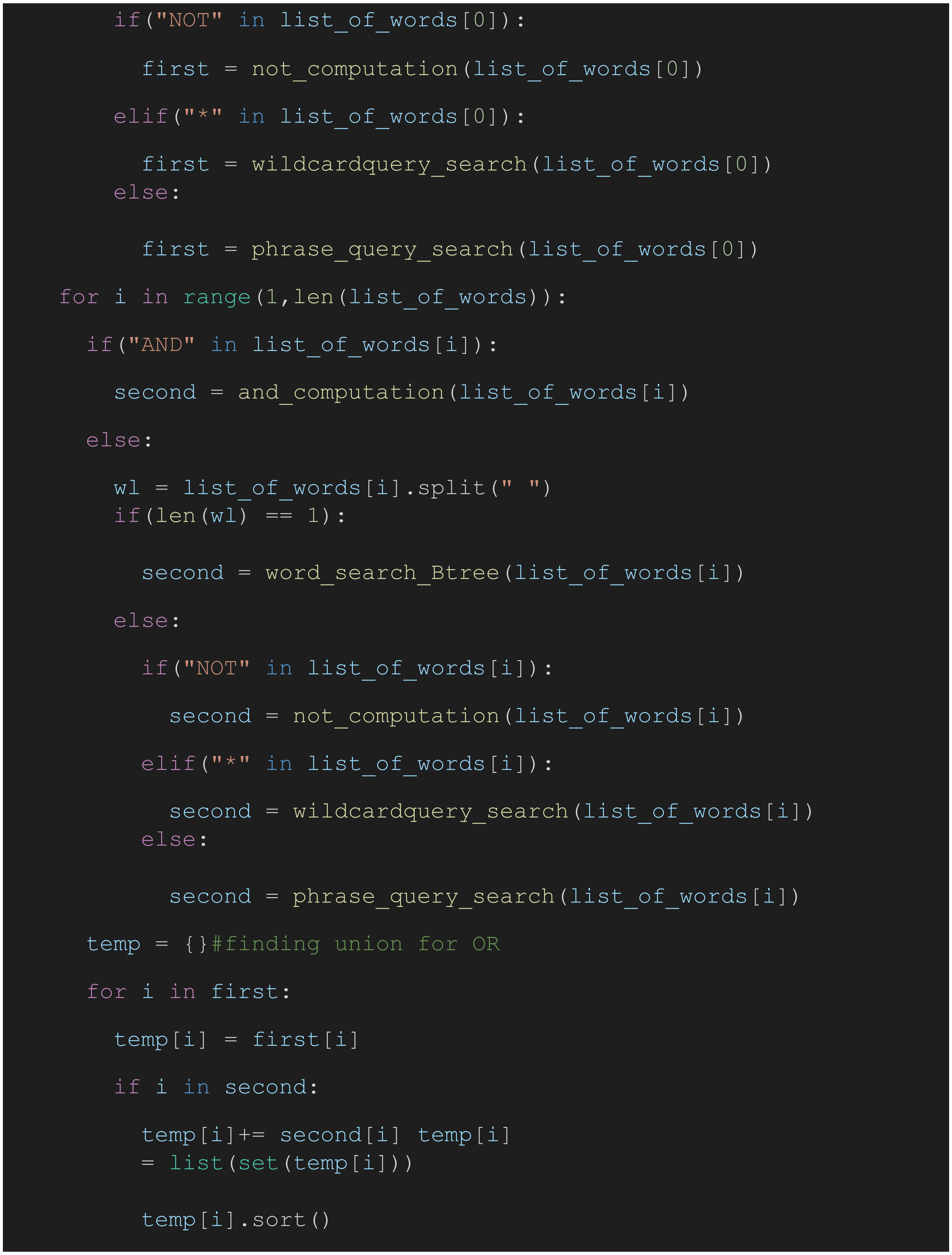
```
            if(k in ans2[j]):
```



```
              temp[j].append(k)
      ans1 = temp
  else:
    #without single word query
    if("NOT" in phq[0]):
      ans1 = not_computation(phq[0])
    elif("*" in phq[0]):
      ans1 = wildcardquery_search(phq[0])
    else:
      ans1 = phrase_query_search(phq[0])
    temp = {}

    for i in range(1,len(phq)):
      if("NOT" in phq[i]):
        ans2 = not_computation(phq[i])
      elif("*" in phq[i]):
        ans2 = wildcardquery_search(phq[i])
      else:
        ans2 = phrase_query_search(phq[i])
      temp = {}
      #finding intersection
      for j in ans1:
        if(j in ans2):
          temp[j] = []
          for k in ans1[j]:
            if(k in ans2[j]):

              temp[j].append(k)
      ans1 = temp
  return ans1


def andorquery_search(words):
    list_of_words= words.split(" OR ")
    or_computation = []

    if("AND" in list_of_words[0]):
      first = and_computation(list_of_words[0])
    else:
      wl = list_of_words[0].split(" ")
```



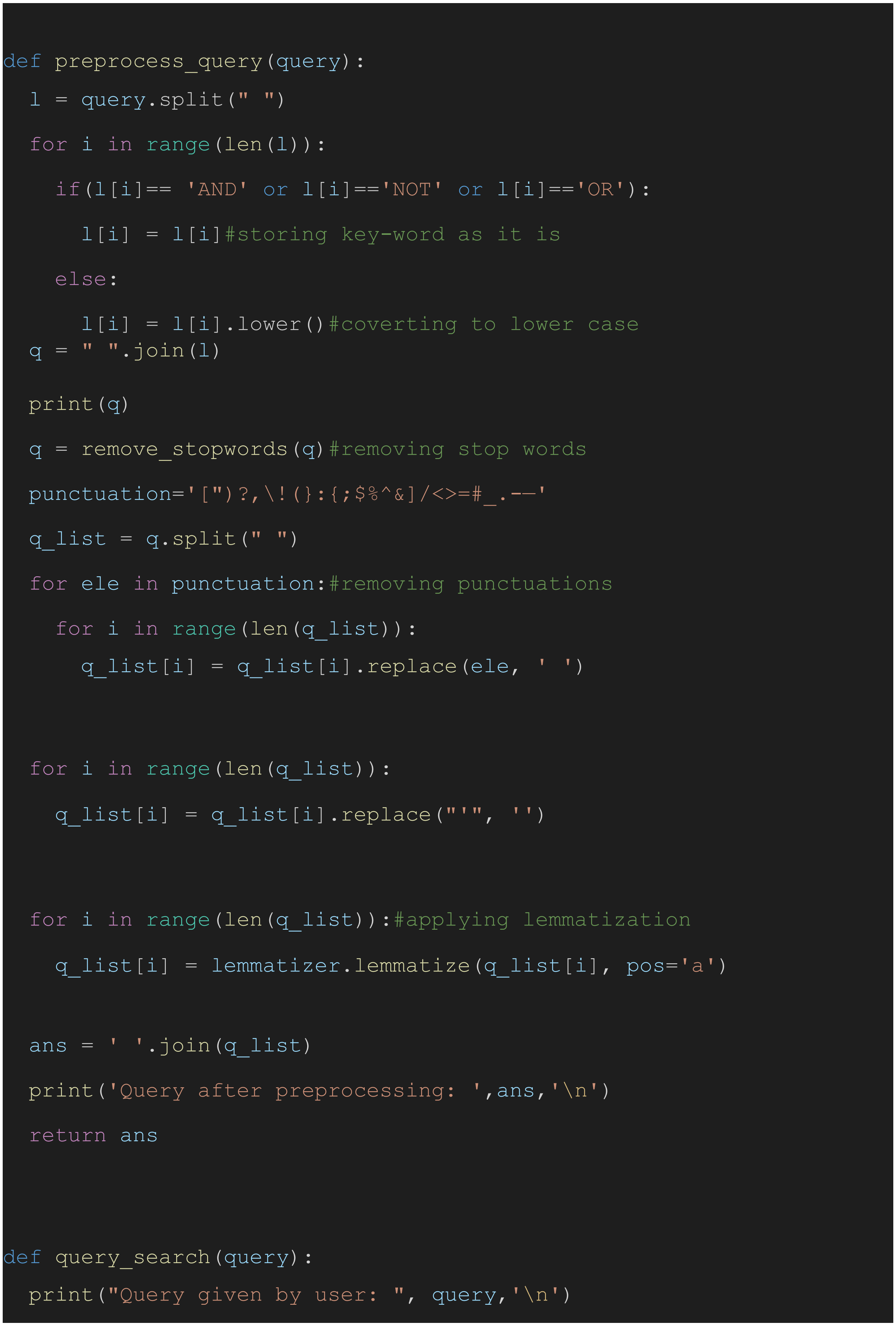
```
      if(len(wl) == 1 and '*' not in list_of_words[0]):
        first = word_search_Btree(list_of_words[0])
      else:
```



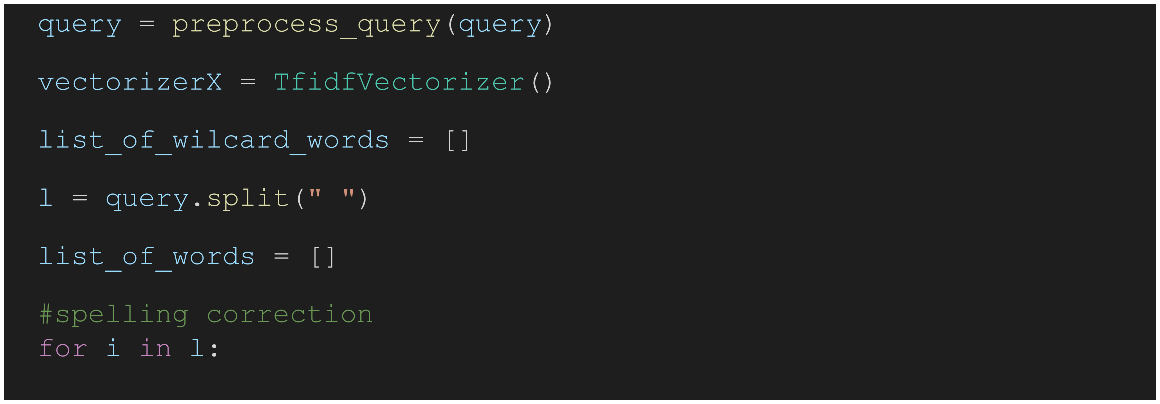
```
        if("NOT" in list_of_words[0]):
          first = not_computation(list_of_words[0])
        elif("*" in list_of_words[0]):
          first = wildcardquery_search(list_of_words[0])
        else:
          first = phrase_query_search(list_of_words[0])
    for i in range(1,len(list_of_words)):
      if("AND" in list_of_words[i]):
        second = and_computation(list_of_words[i])
      else:
        wl = list_of_words[i].split(" ")
        if(len(wl) == 1):
          second = word_search_Btree(list_of_words[i])
        else:
          if("NOT" in list_of_words[i]):
            second = not_computation(list_of_words[i])
          elif("*" in list_of_words[i]):
            second = wildcardquery_search(list_of_words[i])
          else:
            second = phrase_query_search(list_of_words[i])
      temp = {}#finding union for OR
      for i in first:
        temp[i] = first[i]
        if i in second:
          temp[i]+= second[i] temp[i]
          = list(set(temp[i]))
          temp[i].sort()
          second.pop(i)

      for j in second:

        temp[j] = second[j]

      first = temp

    ans = first
    ans1 = {}

    for i in ans:

      if(ans[i]!=[]):

        ans1[i] = ans[i]

    ans = ans1

    return ans


ans = andorquery_search('NOT hello AND good morning')
```

```
print('The results for NOT hello AND good morning is the

following:')

print(ans)



# #### Vector Space Model



# In[17]:




def document_vector(document_data, vectorizerX):
  vectorizerX.fit(document_data)

  doc_vector = vectorizerX.transform(document_data)

  return doc_vector



def ranking(query, ans, vectorizerX, only_not):
  doc_data = []

  mapping = {}#mapping for rows of each doc to id
  ctr= 0

  for i in ans:# i is the doc_id

    for j in ans[i]:

      doc_data.append(corpus[i][j])#data of the retrieved rows

      mapping[ctr] = (i,j)

      ctr+=1


  doc_vector = document_vector(doc_data, vectorizerX)

  query_vector = vectorizerX.transform([query])

  cosineSimilarities =

cosine_similarity(doc_vector,query_vector).flatten()

  if(only_not):

    related_docs = cosineSimilarities.argsort()[:-10:]#display in

ascending order if only NOT is present
  else:

    related_docs = cosineSimilarities.argsort()[:-10:-1]#else

display in descending order



  return related_docs,mapping




# #### Query Pre-Processing and Searching:
```

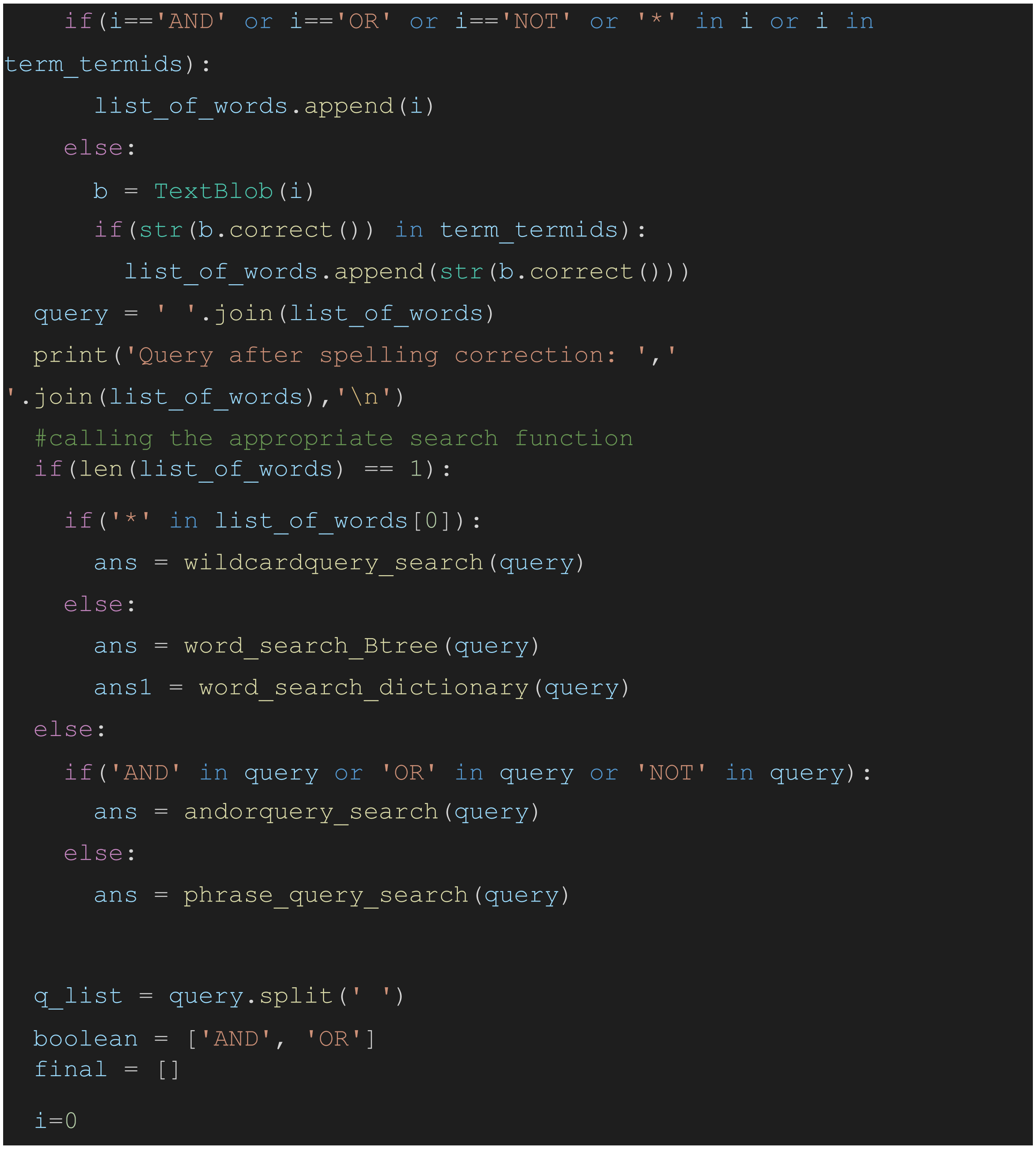
```


# In[18]:


```



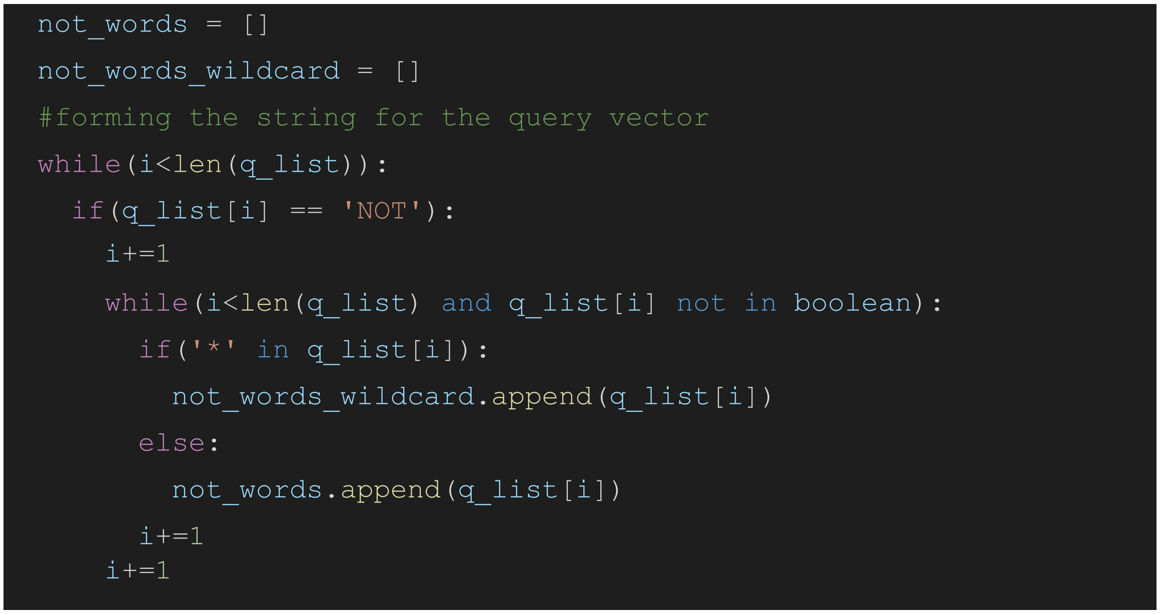
```
def preprocess_query(query):
  l = query.split(" ")
  for i in range(len(l)):
    if(l[i]== 'AND' or l[i]=='NOT' or l[i]=='OR'):
      l[i] = l[i]#storing key-word as it is
    else:
      l[i] = l[i].lower()#coverting to lower case
  q = " ".join(l)
  print(q)
  q = remove_stopwords(q)#removing stop words
  punctuation='[")?,\!(}:{;$%^&]/<>=#_.-—'
  q_list = q.split(" ")
  for ele in punctuation:#removing punctuations
    for i in range(len(q_list)):
      q_list[i] = q_list[i].replace(ele, ' ')


  for i in range(len(q_list)):
    q_list[i] = q_list[i].replace("'", '')


  for i in range(len(q_list)):#applying lemmatization
    q_list[i] = lemmatizer.lemmatize(q_list[i], pos='a')

  ans = ' '.join(q_list)
  print('Query after preprocessing: ',ans,'\n')
  return ans



def query_search(query):
  print("Query given by user: ", query,'\n')
```



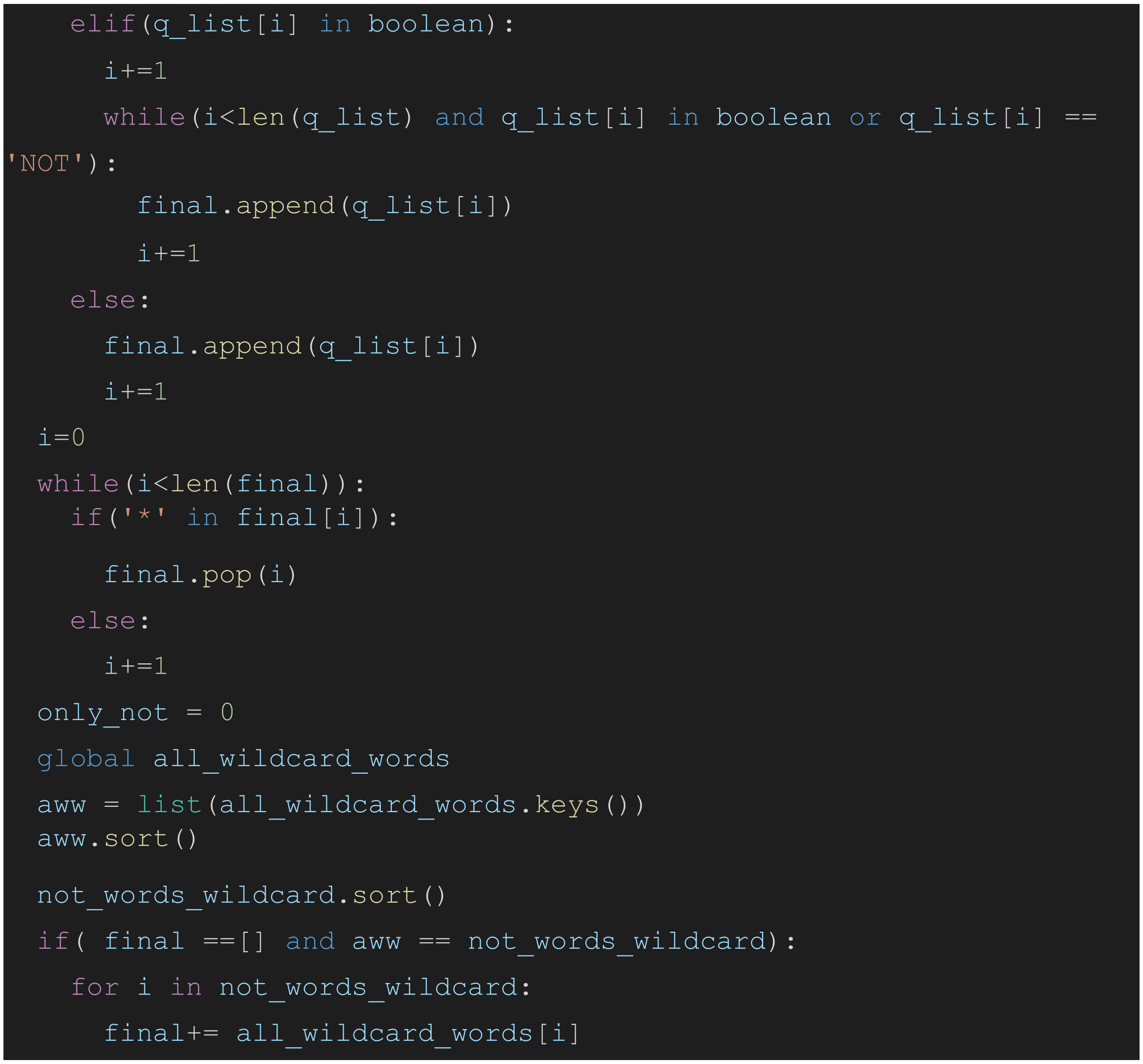
```
  query = preprocess_query(query)
  vectorizerX = TfidfVectorizer()
  list_of_wilcard_words = []
  l = query.split(" ")
  list_of_words = []
  #spelling correction
  for i in l:
```



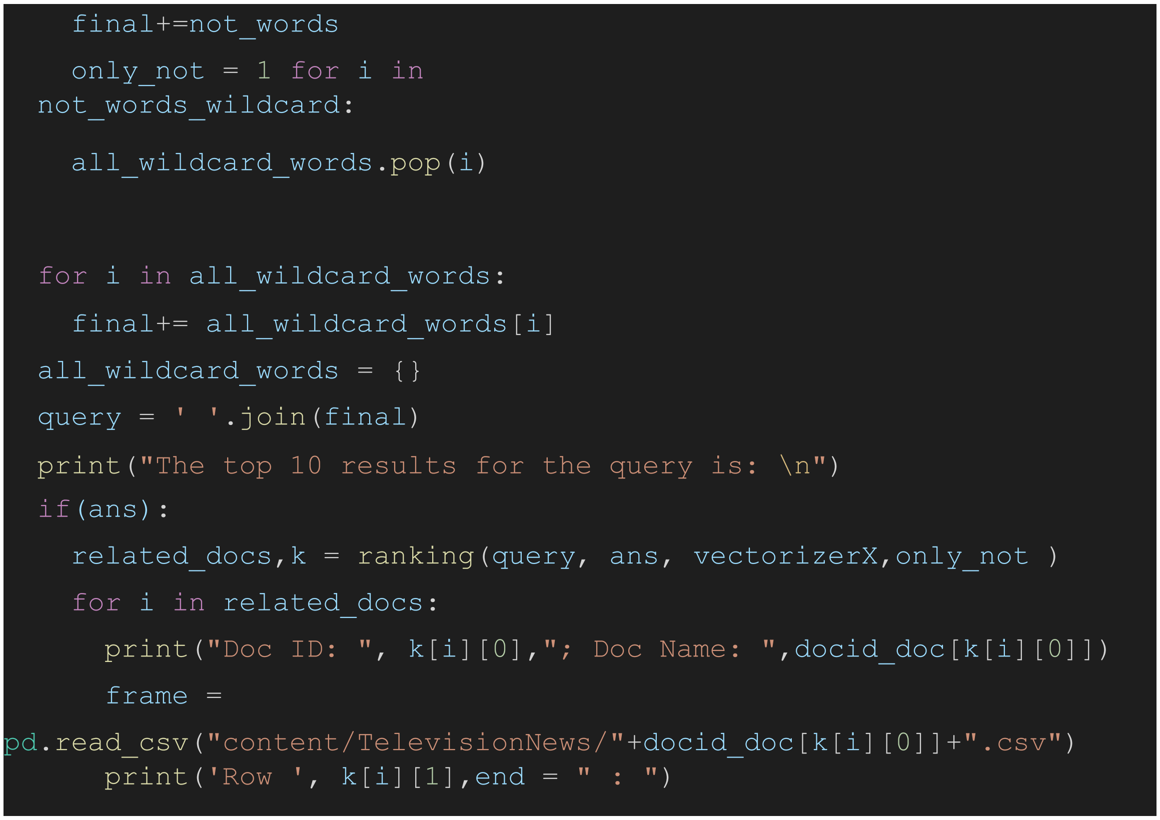
```
    if(i=='AND' or i=='OR' or i=='NOT' or '*' in i or i in
term_termids):
      list_of_words.append(i)
    else:
      b = TextBlob(i)
      if(str(b.correct()) in term_termids):
        list_of_words.append(str(b.correct()))
  query = ' '.join(list_of_words)
  print('Query after spelling correction: ','
'.join(list_of_words),'\n')
  #calling the appropriate search function
  if(len(list_of_words) == 1):
    if('*' in list_of_words[0]):
      ans = wildcardquery_search(query)
    else:
      ans = word_search_Btree(query)
      ans1 = word_search_dictionary(query)
  else:
    if('AND' in query or 'OR' in query or 'NOT' in query):
      ans = andorquery_search(query)
    else:
      ans = phrase_query_search(query)


  q_list = query.split(' ')
  boolean = ['AND', 'OR']
  final = []
  i=0
```



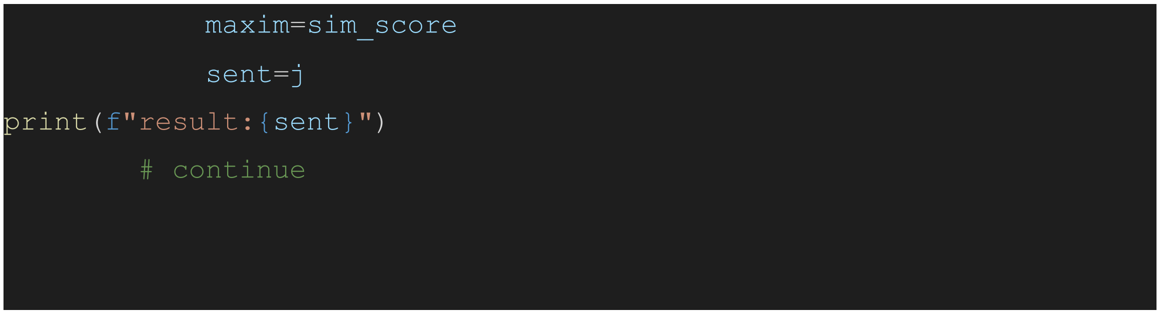
```
  not_words = []
  not_words_wildcard = []
  #forming the string for the query vector
  while(i<len(q_list)):
    if(q_list[i] == 'NOT'):
      i+=1
      while(i<len(q_list) and q_list[i] not in boolean):
        if('*' in q_list[i]):
          not_words_wildcard.append(q_list[i])
        else:
          not_words.append(q_list[i])
        i+=1
      i+=1
```



```
    elif(q_list[i] in boolean):
      i+=1
      while(i<len(q_list) and q_list[i] in boolean or q_list[i] ==
'NOT'):
        final.append(q_list[i])
        i+=1
    else:
      final.append(q_list[i])
      i+=1
  i=0
  while(i<len(final)):
    if('*' in final[i]):
      final.pop(i)
    else:
      i+=1
  only_not = 0
  global all_wildcard_words
  aww = list(all_wildcard_words.keys())
  aww.sort()
  not_words_wildcard.sort()
  if( final ==[] and aww == not_words_wildcard):
    for i in not_words_wildcard:
      final+= all_wildcard_words[i]
```



```
    final+=not_words
    only_not = 1 for i in
  not_words_wildcard:
    all_wildcard_words.pop(i)


  for i in all_wildcard_words:
    final+= all_wildcard_words[i]
  all_wildcard_words = {}
  query = ' '.join(final)
  print("The top 10 results for the query is: \n")
  if(ans):
    related_docs,k = ranking(query, ans, vectorizerX,only_not )
    for i in related_docs:
      print("Doc ID: ", k[i][0],"; Doc Name: ",docid_doc[k[i][0]])
      frame =
pd.read_csv("content/TelevisionNews/"+docid_doc[k[i][0]]+".csv")
      print('Row ', k[i][1],end = " : ")
      print(frame.loc[k[i][1]]['Snippet'])
      print("\n")
  else:#when no documents are retrieved then ranking is done based on relevance
    ans = doc_rows
    related_docs,k = ranking(query, ans, vectorizerX,only_not )
    for i in related_docs:
      print("Doc ID: ", k[i][0],"; Doc Name: ",docid_doc[k[i][0]])
      frame = pd.read_csv("content/TelevisionNews/"+docid_doc[k[i][0]]+".csv")
      print('Row ', k[i][1],end = " : ")
      print(frame.loc[k[i][1]]['Snippet'])
      print("\n")



# In[23]:


query_search('white house AND m*ger')

# #### Semantic Matching:
# In[26]:



import spacy
sem_arr={}
maxim=0
new_data=data_list[:10]+data_list[100:110]+data_list[200:210]+data_list[400:]
nlp = spacy.load("en_core_web_lg")
# nlp = spacy.load("en_core_web_md")
query_sem=nlp(u'cars give out harmfull emissions')
# print(len(data_list))
for i in range(len(new_data)):
    for j in new_data[i].values():
        # print(i,j)
        temp=nlp(j)
        sim_score=query_sem.similarity(temp)
        if sim_score > maxim:
```



```
            maxim=sim_score
            sent=j
print(f"result:{sent}")
        # continue
```