Assignment 1 Report

Of

Information Retrieval

Group Members:

Milind Jain	**2020A7PS0153H**
Ashwin Naveen Pugalia	**2020A7PS1080H**
M. Bhargav	**2020A7PS0025H**

Under the supervision of

FACULTY: Dr. Aruna Malapati

SUBMITTED AS AN EVALUATION COMPONENT OF

Course: Information Retrieval - CS F469

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI

HYDERABAD CAMPUS

ABSTRACT

The Report provides an implementation of a search engine that supports different types of queries. The following are the same main features implemented:

For single-word queries, the search engine uses an inverted index with a dictionary data structure to provide fast access to the posting list for any given word in constant time. The search engine also employs Porter stemming algorithm for normalization of tokenized words to reduce them to their root form.

For phrase queries, the search engine uses a skip-gram model for generating word embeddings of documents, allowing the engine to calculate similarity scores between a given query and a set of documents.

The search engine also uses TF-IDF (term frequency-inverse document frequency) scores to rank documents by relevance to a query.

Wildcard queries are also supported by the search engine, with a Prefix and Suffix Trie used for handling trailing and leading wildcard queries respectively.

Finally, the search engine supports boolean retrieval queries, with the query treated as an AND or OR query based on user input. The posting list for each normalized term is retrieved from the inverted index and set union or intersection operations are performed.

Overall, the implementation provides an efficient and effective search engine that supports a range of queries, making it a valuable tool for information retrieval.

IMPLEMENTATION

Inverted Index Construction and Preprocessing

Before constructing the inverting index we need to install essential datasets for stopword removal and normalization done during the preprocessing step. We have used the available nltk datasets for the same purpose.

```
# nltk.download('punkt')
# nltk.download('words')
# nltk.download('stopwords')
```

Normalization

Porter stemming algorithm is used for the purpose of normalization of tokenized words. It reduces a given term to its root form, thus the query is robust to more typical morphological and inflexional endings. We have the used the PorterStemmer by nltk library for the same purpose.

```
# stemming porter object
stemmer = PorterStemmer()
root = Path(".")
```

Before construction of inverted index, we have converted all the pdf files given to word documents as directly reading from pdfs lead to a higher loss as compared to word documents and the pdfs might contain watermarks or unselectable text which can hamper our text extraction purpose. We have used **textract** and **sent_tokenize** by nltk library to extract the raw text from the documents and tokenize every sentence to preserve the context.

```
# list of all doc names
files = list()
for dir in [r"\Auto", r"\Property"]:
    cur_dir = r".\Docs" + dir
    for file in os.listdir(cur_dir):
        cur_path = r".\Docs" + dir + "\\" + file
        files.append(cur_path)
files.sort()

docs = list()
for x in range(len(files)):
    for i in sent_tokenize(textract.process(files[x]).decode("utf8")):
        docs.append((i, x))
```

Inverted Index Construction

The structure of our inverted index is a list of tuples of form *(doc index, file index, frequency)* where doc index is the index of the newly transformed documents, file index is the index of file the transformed document belongs to, and the frequency is the number of times the normalized term or the **key** appears in the transformed document. The size of

our transformed documents is dynamic in nature, the minimum threshold we have kept is 500 characters. We keep on adding sentences to a document until we get atleast 500 characters. Benefit of adding sentences instead of words is that context of each transformed document is preserved, it starts with a sentence and there is no abrupt ending of the context when the user queries for any document.

For a given sentence, we tokenize it to words using **nltk.word_tokenize** and convert those words to lower case. if the word is a stopword, we skip it, otherwise we normalize the word using porter stemming and construct the inverted index with **key** as the normalized term. Our inverted_index is a dictionary whose **key** is the normalized term and **value** is the list of tuples as mentioned above.

After the construction of the inverted index table, we sort each list by the third element of the tuple, that is frequency of the **key** so that we get relevant results whenever user queries for anything.

```
# set of Stop words
stop_words = set(stopwords.words('english'))

"""
{
   key : string (normalized)
   value: list of ("doc index", file_index, frequency)
}
"""
inverted_idx = dict()

# list of string modified document
documents = list()

count_id = 0

def process(doc_index):
    """
    Reads file, tokenize it, normalizes it and builds the inverted index
    """

    result = doc_index + 1
    global count_id
    text = docs[doc_index][0]
    file_index = docs[doc_index][1]
    while doc_index + 1 < len(docs):
        doc_index += 1
        if docs[doc_index][1] == docs[doc_index - 1][1] and len(text +
docs[doc_index][0]) <= 500:
            text += docs[doc_index][0]
        else:
            result = doc_index
            break
```

```
    tokens = nltk.tokenize.word_tokenize(str(text))

    new_token = list()
    for i in tokens:
        new_token.append(i.lower())
    tokens = new_token

    curr_str = ""
    normalised_word_freq = dict()
    for j in range(len(tokens)):
        curr_str += tokens[j] + " "

        normal = stemmer.stem(tokens[j].lower())
        if normalised_word_freq.get(normal) != None:
            normalised_word_freq[normal] += 1
        else:
            normalised_word_freq[normal] = 1

    documents.append((curr_str, files[file_index]))

    visited = set()
    for j in range(len(tokens)):
        normalised_word = stemmer.stem(tokens[j].lower())
        if tokens[j].lower() not in stop_words and normalised_word not in
visited:
            visited.add(normalised_word)

            if inverted_idx.get(normalised_word) != None:
                inverted_idx[normalised_word].append((count_id, file_index,
normalised_word_freq[normalised_word]))
            else:
                inverted_idx[normalised_word] = [(count_id, file_index,
normalised_word_freq[normalised_word])]
    count_id += 1

    return result

i = 0
while i < len(docs):
    i = process(i)

for x in inverted_idx:
    inverted_idx[x] = sorted(inverted_idx[x], key=lambda y: -y[2])
```

After the entire preprocessing stage inverted index table, transformed document list and the list of original files is dumped to respective binary files so that whenever user wants to query, our search engine can just open those files and use those data structures for querying directly instead of preprocessing again and again.

```
my_path = root / "Pickled_files" / "Inverted_index"
dbfile = open(my_path, 'wb')
pickle.dump(inverted_idx, dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Documents"
dbfile = open(my_path, 'wb')
pickle.dump(documents, dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Files"
dbfile = open(my_path, 'wb')
pickle.dump(files, dbfile)
dbfile.close()
```

Boolean Retrieval System

Initializing PorterStemmer Object for normalization of query terms

```
# stemming porter object
stemmer = PorterStemmer()
stop_words = set(stopwords.words('english'))
root = Path("../")
```

Loading the inverted_index, transformed documents and file name binary files

```
my_path = root / "Pickled_files" / "Inverted_index"
dbfile = open(my_path, 'rb')
inverted_idx = pickle.load(dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Documents"
dbfile = open(my_path, 'rb')
documents = pickle.load(dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Files"
dbfile = open(my_path, 'rb')
files_list = pickle.load(dbfile)
dbfile.close()
```

Query

For boolean retrieval we have a created a function **boolean_retrieval** which takes in two arguments, the **query** string and the **type** of the retrieval, if **type** is True then the query is treated as an **AND** query and if false, its treated as an **OR** query. **query** can be a multiword string. First we tokenize the **query** and convert those tokens to lower case. The token is checked for stopword and numbers and further normalized using porter stemming. The posting list for each of the normalized term is retrived from the inverted index and set union or intersection operation is done as the **type** of the query. While processing the posting list we also count the cumulative frequencies of all the terms document-wise and after getting the final list of documents, we sort them on basis of the frequencies in descending order and return the top 3 results along with original file names these documents are present in.

```
def boolean_retrieval(query , type) :
    ans = list()
    tokens = nltk.tokenize.word_tokenize(str(query))
    new_token = list()
    for i in tokens:
        cur_str = i.lower()
        if cur_str in stop_words or cur_str.isnumeric():
            continue
        new_token.append(cur_str)
    tokens = new_token
    rank , file_id = dict() , dict()
    set_id = list()
    for words in tokens :
        freq = dict()
        normalized_word = stemmer.stem(words)
        if inverted_idx.get(normalized_word) == None :
            return ans
        ids = set()
        for files in inverted_idx[normalized_word] :
```

```
            ids.add(files[0])
            file_id[files[0]] = files[1]
            if rank.get(files[0]) == None :
                rank[files[0]] = files[2]
            else :
                rank[files[0]] += files[2]
        set_id.append(ids)

    if type :
        merged_id = list(set.intersection(*set_id))
    else :
        merged_id = list(set.union(*set_id))
    merged_id = sorted(merged_id , key = lambda y : -rank[y])
    for i in range(0 , min(3 , len(merged_id))) :
        ans.append((documents[merged_id[i]] ,
files_list[file_id[merged_id[i]]]))
    return ans
```

Extracting the original file name from its path

```
def extract_file(path) :
    ans = ""
    j = len(path) - 1
    while j >= 0 and path[j] != '\\' :
        ans += path[j]
        j -= 1
    return ans[::-1]
```

AND-Query

```
and_query = boolean_retrieval("Illegal Drugs ComPensation" , True)
for i in range(len(and_query)) :
    print(str(i + 1) + ") " , extract_file(and_query[i][1]))
    print(and_query[i][0][0])
    print("-----------------------------------")
```

```
1)  BRIT-PO-Policy-Wording-May-2016-1.docx
any punitive or exemplary damages , compensations , fines or any penalties of
whatsoever nature which the insured is ordered to pay by a forum , authority
or body of competent jurisdiction ; in respect of coverage clause 1 eviction
of squatters only : - a legal expenses ..............................
-----------------------------------
```

OR-Query

```
or_query = boolean_retrieval("Policy insured" , False)
for i in range(len(or_query)) :
    print(str(i + 1) + ") " , extract_file(or_query[i][1]))
    print(or_query[i][0][0])
    print("-----------------------------------")
```

```
1)  Residential-Property-Owners-Policy-Wording-1910.docx
these documents are the policy statement of fact and/or schedule endorsements
```

```
notice to policyholders condition precedent any term expressed condition
precedent is extremely important if you are in breach of any of these
obligations at the time of a loss we will have no obligation to indemnify you
in relation to any claim for that loss however if a condition precedent is
intended to reduce the risk of a loss ..............................
2)  Property-Owner-Policy-Wording.docx
complete property owners insurance | 1 2 | complete property owners insurance
complete property owners insurance | 3 4 | complete property owners insurance
complete property owners insurance | 23 6 | complete property owners
insurance complete property owners insurance ..........................
---------------------------------
3)  Residential-Property-Owners-Policy-Wording-1910.docx
once you have gained possession of your residential property the most we will
pay shall be 75 % of the monthly rent that was previously payable for a
maximum further period of two months provided that you agree to re-let your
residential property where an offer equal to or greater than 85 % of the
preceding rent is offered you are responsible for the first unpaid month ’ s
rent ( which you have collected as a deposit ) ..........................

---------------------------------
```

Ranking documents using skip grams word embedding model for phrase queries

Setup

The corpus/documents are extracted from the pickle files.

Stemmer has been initialised to convert term to its root form (Example : received -> receive)

```
stemmer = PorterStemmer()
stop_words = set(stopwords.words('english'))
root = Path("../")

my_path = root / "Pickled_files" / "Documents"
dbfile = open(my_path, 'rb')
documents = pickle.load(dbfile)
dbfile.close()
```

Model

Skip-gram model is a type of neural network architecture that is commonly used for word embedding. It is a form of unsupervised learning that aims to learn the distributional representation of words.The basic idea behind skip-gram model is to predict the context words surrounding a given target word, rather than predicting the target word from the context words. The context words are defined as the words that occur within a certain window of the target word.

The skip-gram model consists of an input layer, a hidden layer, and an output layer. The input layer represents the target word, and the output layer represents the context words. The hidden layer is used to transform the input word into a distributed representation, or embedding, which is used to predict the context words.During training, the skip-gram model is fed with a large corpus of text. For each target word in the corpus, a training example is created by randomly selecting one of its context words as the output, and setting the remaining context words as inputs to the model. The model is then trained to predict the output context word given the input context words.

The training process involves adjusting the weights of the model to minimize the difference between the predicted context word and the actual context word. The weights are adjusted using backpropagation algorithm, which updates the weights based on the difference between the predicted and actual output.After training, the skip-gram model generates a dense vector representation for each word in the vocabulary. The vector representation captures the semantic and syntactic information of the word, and can be used as input to other machine learning models for various natural language processing tasks such as text classification, information retrieval, and machine translation.

Building the model

The following code initializes an empty list called data, which will be used to store the tokenized and preprocessed documents. Then, it loops over each document in the documents list and tokenizes each sentence into a list of words. Each word is then converted to lowercase and added to a temporary list called temp. The temp list is then appended to the data list, which now contains a list of tokenized and preprocessed documents.

Next, the Word2Vec model from the gensim library is used to train a word embedding model on the data list. The min_count parameter specifies the minimum frequency count of a word in the corpus for it to be included in the model. The vector_size parameter specifies the dimensionality of the word vectors. The window parameter specifies the maximum distance between the target word and the context word within a sentence. The sg parameter specifies the training algorithm to be used - 1 for skip-gram and 0 for CBOW. In this case, sg is set to 1 which corresponds to the skip-gram algorithm for training.

After training, the word embedding model skipgram_model contains dense vector representations for each word in the corpus. These vector representations can be used for various natural language processing tasks such as text classification, information retrieval, and machine translation.

```
data = list()
for i in documents:
    temp = list()
    # tokenize the sentence into words
    for j in word_tokenize(i[0]):
        temp.append(j.lower())
    data.append(temp)
```

```
skipgram_model = gensim.models.Word2Vec(data, min_count = 1, vector_size =
100, window = 5,sg = 1)
```

Generating word embeddings for documents

The following code defines two functions.

1.**preprocess(text)** takes a string of text as input, tokenizes it, removes stop words and filters out words with length less than two. It returns a list of preprocessed tokens.

2.**text_embedding(text, model)** takes a string of text and a pre-trained word embedding model as inputs. It preprocesses the text using the preprocess() function, obtains the word embeddings for each word in the preprocessed text from the pre-trained word embedding model, takes the average of the word embeddings to generate a document embedding, and returns the document embedding as a numpy array.

Finally, the **document_vecs** variable is assigned a list of document embeddings generated using the text_embedding() function and the pre-trained word embedding model (skipgram_model) for each document in the documents list.

```
def preprocess(text):
    tokens = []
    for word in word_tokenize(text, language='english'):
        if len(word) >= 2 and word not in stop_words:
            tokens.append(word)
    return tokens

# Define a function to generate text embeddings
def text_embedding(text, model):
    # Preprocess the text (tokenize, remove stop words, stem)
    preprocessed_text = preprocess(text)
    # Get the word embeddings for each word in the document
    word_embeddings = [model.wv[word] for word in preprocessed_text if word
in model.wv.key_to_index]
    # Take the average of the word embeddings to generate a document
embedding
    if len(word_embeddings) > 0:
        document_embedding = np.mean(word_embeddings, axis=0)
    else:
        document_embedding = np.zeros(model.vector_size)
    return document_embedding

document_vecs = [text_embedding(document[0], skipgram_model) for document in
documents]

pd.DataFrame(document_vecs)
```

```
          0         1         2         3         4         5         6   \
0  0.024118  0.054352  0.160210 -0.014218  0.082009 -0.344629  0.258807
1  0.119907  0.201983  0.211899  0.003732  0.064681 -0.353865  0.322289
```

```
2     0.107741  0.121338  0.233221 -0.062143  0.128717 -0.314511  0.198515
3     0.102654  0.103504  0.217527 -0.102527  0.098577 -0.286220  0.182237
4     0.139915  0.166985  0.191470  0.063486  0.078979 -0.259502  0.209747
...        ...       ...       ...       ...       ...       ...       ...
3470  0.017150  0.070993 -0.009402 -0.076820  0.096241 -0.286062  0.302957
3471  0.036497  0.094441  0.112825  0.013753  0.073670 -0.256091  0.275500
3472 -0.030065  0.098614  0.016564  0.152271  0.040411 -0.251735  0.280965
3473 -0.033734  0.049069 -0.025258  0.205588  0.010855 -0.381962  0.347696
3474 -0.047206  0.048118  0.087636  0.061849  0.004788 -0.079298  0.117113

            7         8         9   ...        90        91        92  \
0     0.318142  0.164285 -0.074738   ...  0.309035  0.063572  0.216361
1     0.236876  0.101685 -0.054317   ...  0.284209  0.123695  0.225444
2     0.248890  0.078341  0.021138   ...  0.342786  0.130239  0.213134
3     0.312340  0.114414  0.029829   ...  0.398635  0.064415  0.182253
4     0.178108  0.101391 -0.017329   ...  0.303954  0.129494  0.266528
...        ...       ...       ...   ...       ...       ...       ...
3470  0.333233  0.254625 -0.026865   ...  0.512615  0.011278  0.205708
3471  0.281152  0.163027 -0.041075   ...  0.313975  0.034412  0.218213
3472  0.301714  0.021698 -0.108164   ...  0.231512 -0.007439  0.065900
3473  0.320560 -0.067996 -0.131631   ...  0.295148 -0.023226  0.016403
3474  0.159525 -0.012236 -0.053205   ...  0.048973 -0.020183  0.044832

            93        94        95        96        97        98        99
0    -0.125844  0.121549  0.045422  0.353171 -0.105587  0.175716  0.116122
1    -0.242821  0.067655  0.023104  0.535277  0.168928  0.202390 -0.000568
2    -0.227343  0.135269  0.011459  0.648255  0.190503  0.134647  0.188918
3    -0.214377  0.013815 -0.035877  0.563727  0.154394  0.101942  0.284522
4    -0.182365  0.157443  0.129105  0.537782  0.098033  0.201487  0.122230
...        ...       ...       ...       ...       ...       ...       ...
3470 -0.164048  0.111797  0.189495  0.457768 -0.117088  0.092861  0.195990
3471 -0.150709  0.151913  0.127191  0.378745 -0.085460  0.078333  0.080446
3472 -0.048478  0.091253  0.212383  0.110363 -0.340176  0.071022  0.059467
3473 -0.024847  0.175199  0.241717  0.068951 -0.538251 -0.000923  0.057817
3474 -0.013773  0.057653  0.013054  0.049953 -0.088546  0.017593  0.052812

[3475 rows x 100 columns]
```

Similarity measure

Here cosine_similarity is used to compare the query vector and and the document vector.

```
# Define a function to calculate cosine similarity between two vectors
def cosine_similarity(u, v):
    return np.dot(u, v) / (np.linalg.norm(u) * np.linalg.norm(v))
```

Saving document vectors for future use

```
my_path = root / "Pickled_files" / "Document_vectors"
dbfile = open(my_path, 'wb')
pickle.dump(document_vecs, dbfile)
```

```
dbfile.close()

my_path = root / "Pickled_files" / "Document_vectors"
dbfile = open(my_path, 'rb')
document_vecs = pickle.load(dbfile)
dbfile.close()

def extract_file(path) :
    ans = ""
    j = len(path) - 1
    while j >= 0 and path[j] != '\\' :
        ans += path[j]
        j -= 1
    return ans[::-1]
```

Processing the query

This code calculates the similarity scores between a given query and a set of documents. It first generates a document vector for each document in the set using the text_embedding function. Then, it generates a query vector using the same function. The cosine similarity between the query vector and each document vector is calculated and stored in the similarity_scores list. Finally, the documents are ranked in descending order based on their similarity scores with the query, and the top five documents are printed.

```
def printResults(query,ranked_docs):
    print(query)
    print()
    for i in range(0,min(3,len(ranked_docs))):
        print(str(i + 1) + ". " , extract_file(ranked_docs[i][1]))
        print(ranked_docs[i][0])
        print("------------------------------------")

# Example usage: compare a query with a set of documents
input_file = open("phrase_input.txt","r")
query = input_file.readline()
query_vec = text_embedding(query,skipgram_model)
# Calculate similarity scores between the query and all documents
similarity_scores = [cosine_similarity(query_vec, document_vec) for
document_vec in document_vecs]
# Rank documents based on similarity scores
ranked_documents = [document for _, document in sorted(zip(similarity_scores,
documents), reverse=True)]
# Print the ranked documents
printResults(query,ranked_documents)
```

```
Medical reports

1.  AU127-1.docx
proof of claim ; medical reports as soon as possible , any person making
claim must give us written proof of claim.the injured person may be required
```

```
to take medical examinations by physicians we choose , as often as we
reasonably require.we must be given authorization to obtain medical reports
and copies of records pertinent to the claim .
-----------------------------------
2.  PP_00_01_06_98.docx
promptly send us copies of any notices or legal papers received in connection
with the accident or loss.submit , as often as we reasonably require : to
physical exams by physicians we select.we will pay for these exams.to
examination under oath and subscribe the same.authorize us to obtain :
medical reports ; and other pertinent records.submit a proof of loss when
required by us .
-----------------------------------
3.  rsa_property_owners_policy_wording.docx
if you believe that we have not delivered the service you expected , we want
to hear from you so that we can try to put things right.we take all
complaints seriously and following the steps below will help us understand
your concerns and give you a fair response.step 1 if your complaint relates
to your policy then please raise this with your insurance adviser .
-----------------------------------
```

Single Word Query

Loading the inverted_index, transformed documents and file name binary files

```
my_path = root / "Pickled_files" / "Inverted_index"
dbfile = open(my_path, 'rb')
inverted_idx = pickle.load(dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Documents"
dbfile = open(my_path, 'rb')
documents = pickle.load(dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Files"
dbfile = open(my_path, 'rb')
files = pickle.load(dbfile)
dbfile.close()
```

Query

Input Query is first normalized using PorterStemming algorithm and the normalized term is searched in the inverted index. If the match is found, the posting list is returned which has documents already ranked in decreasing order of term frequency. The top 3 results are returned to the user.

```
def single_word_query(query_str):
    """
    Normalize query string and search in inverted index and retervive doc
    """
```

```
    query_str = stemmer.stem(query_str.lower())
    if inverted_idx.get(query_str) == None:
        return ["No match found :("]
    else:
        ans = []
        for i in range(min(len(inverted_idx[query_str]), 3)):
            ans.append((documents[inverted_idx[query_str][i][0]][0],
files[inverted_idx[query_str][i][1]]))
        return ans
```

Extracting the original file name from its path

```
def extract_file(path) :
    ans = ""
    j = len(path) - 1
    while j >= 0 and path[j] != '\\' :
        ans += path[j]
        j -= 1
    return ans[::-1]
```

Some example Queries

```
file_input = open("single_query.txt" , "r")
f = file_input.read().splitlines()
for i in f:
    print(i)
    results = single_word_query(i)
    if(results[0] == "No match found :(") :
        print(results[0])
        print("----------------------------------")
        continue
    for i in range(len(results)) :
        print(str(i + 1) + ". " , extract_file(results[i][1]))
        print(results[i][0])
    print("----------------------------------")
```

```
ontario
1.  1215E.2.docx
all arbitrations will be governed by the arbitration act , 1991 ( ontario )
.in court the matter may be decided in a lawsuit brought against us by you or
other insured persons in an ontario court.if so , we have the right to ask
the court to decide who is legally responsible and the amount of compensation
owing , unless another ontario court has already done so in an action that
was defended .
2.  1215E.2.docx
the policy covering the other automobile must be issued by an insurance
company licensed in ontario , or one that has filed with the financial
services commission of ontario to provide this coverage.it is called direct
compensation because you will collect from us , your insurance company , even
though you , or anyone else using or operating the automobile with your
consent , were not entirely at fault for the accident .
3.  1215E.2.docx
```

ontario automobile policy (oap 1) owner ’ s policy approved by the superintendent of financial services for use as the standard owner 's policy on or after june 1 , 2016 about this policy this is your automobile insurance policy.it is written in easy to understand language.please read it carefully so you know your rights and obligations and the rights and obligations of your insurance company .

MilinD
No match found :(

IlLegal
1. 1215E.2.docx
illegal use we wo n't pay for loss or damage caused in an incident : if you are unable to maintain proper control of the automobile because you are driving or operating the automobile while under the influence of intoxicating substances ; if you are convicted of one of the following offences under the criminal code of canada relating to the operation , ...
2. BRIT-PO-Policy-Wording-May-2016-1.docx
any punitive or exemplary damages , compensations , fines or any penalties of whatsoever nature which the insured is ordered to pay by a forum , authority or body of competent jurisdiction ; in respect of coverage clause 1 eviction of squatters only : - a legal expenses .....
3. Residential-Property-Owners-Policy-Wording-1910.docx
once you have gained possession of your residential property the most we will pay shall be 75 % of the monthly rent that was previously payable for a maximum further period of two months provided that you agree to re-let your residential property where an offer equal to or greater than 85 % of the preceding rent is offered you are responsible for the first unpaid month ’ s rent (which you have collected as a deposit) .....

Canadian
1. 1215E.2.docx
all of the dollar limits described in this policy are in canadian funds.definitions automobile in this policy , motorized snow vehicle is included in the definition of automobile.regulations may include , or exclude , certain other types or classes of vehicles as automobiles.in this policy , there is a difference between a described automobile and the automobile.when we refer to an automobile as described , we mean any automobile specifically shown on the certificate of automobile insurance .
2. 1215E.2.docx
if you are insured for loss or damage caused by fire or lightning , there is no deductible for these losses.additional benefits whatever loss or damage coverage you choose under this section , your coverage will include the following additional benefits.payment of charges we will pay general average , salvage and fire department charges and any canadian or u.s. customs duties for which you are legally responsible as a result of an insured peril.example your car is damaged in a fire .
3. 7thEditionPolicy.docx
14optional insurance (continued) if the accident occurs in any other state or in a canadian province and you have purchased any coverage at all under

```
this part , your policy will automatically apply to that accident as follows
: if the state or province has : a financial responsibility law or similar
law specifying limits of liability for bodily injury or property damage
higher than the limits you have purchased , your policy will provide the
higher specified limits .
--------------------------------
```

Ranking documents based on tfidf scores

TF-IDF is a technique used to evaluate the importance of words in a document or corpus. It measures the frequency of a word in a document (TF) and the importance of a word in a corpus (IDF). The score for a word is obtained by multiplying its TF and IDF values. This method is widely used in information retrieval systems to rank documents by relevance to a query.

Setup

The corpus/documents are extracted from the pickle files.

The inverted_index built already is extracted , which would be used for extracting term and document frequencies.

```
stemmer = PorterStemmer()
stop_words = set(stopwords.words('english'))
root = Path("../")

my_path = root / "Pickled_files" / "Documents"
dbfile = open(my_path, 'rb')
documents = pickle.load(dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Inverted_index"
dbfile = open(my_path, 'rb')
inverted_index = pickle.load(dbfile)
dbfile.close()
```

Algorithm

The TF-IDF scoring algorithm calculates a score for each term in the query and each document in the collection, using the following formula:

TF-IDF(term, document) = TF(term, document) * IDF(term)

where TF(term, document) is the term frequency of the term in the document, and IDF(term) is the inverse document frequency of the term. The term frequency represents the number of times a term appears in a document, while the inverse document frequency represents the rarity of the term across the collection of documents.

The TF-IDF score for each term in the query is calculated in the same way. Then, for each document, the algorithm calculates the dot product between the query's TF-IDF scores and the document's TF-IDF scores. The resulting score represents the relevance of the document to the query string.

By ranking the documents based on their scores, the algorithm can identify the most relevant documents for a given query. This technique is widely used in search engines and other information retrieval applications, as it provides an effective way to match documents to user queries.

Implementation

The rank_docs_by_tfidf function is a Python implementation of the TF-IDF (Term Frequency-Inverse Document Frequency) algorithm used for ranking documents based on their relevance to a given query string.

The function takes a query string as input and returns a list of document IDs that are ranked in descending order of their relevance to the query. The algorithm first processes the query string by removing stop words and stem each term using a stemming algorithm to reduce them to their root forms.

Then, the algorithm calculates the TF-IDF score for each term in the query. It multiplies the term frequency in the query with the inverse document frequency (IDF) of the term. For each document that contains the term, it multiplies the document's TF-IDF score with the query's TF-IDF score for that term and adds the result to the document's score.

Finally, the algorithm returns a list of document IDs sorted in descending order of their scores. This allows the user to quickly identify the documents that are most relevant to the query string.

```
def rank_docs_by_tfidf(query):
    query_terms = (query.split(" "))
    query_term_freq = {}
    query_terms = [term.lower() for term in query_terms if term.lower() not
in stop_words]
    for term in query_terms:
        if query_term_freq.get(term) == None:
            query_term_freq[term] = 0
        query_term_freq[term] += 1
    query_terms = list(query_term_freq.keys())

    document_scores = {}
    for i in range(len(documents)):
        document_scores[i] = 0

    for term in query_terms:
        qtf = query_term_freq[term]
        document_freq = 1
        normalised_term = stemmer.stem(term)
```

```
        if inverted_index.get(normalised_term) != None:
            document_freq += len(inverted_index[normalised_term])
            query_score = qtf * (1/document_freq)
            for doc in inverted_index[normalised_term]:
                doc_score = doc[2]*(1/document_freq)
                document_scores[doc[0]] += query_score*doc_score

    ranked_docs = [doc[0] for doc in
sorted(document_scores.items(),key=lambda x:x[1])[::-1]]
    return ranked_docs

def extract_file(path) :
    ans = ""
    j = len(path) - 1
    while j >= 0 and path[j] != '\\' :
        ans += path[j]
        j -= 1
    return ans[::-1]
```

Sample Queries for testing the algorithm

The query string is initialized and then passed to the rank_docs_by_tfidf function, which returns a list of document IDs ranked in descending order of their relevance to the query.The for loop is then used to print the top 3 documents that are most relevant to the query string. The loop iterates for a maximum of 3 times or the number of documents available, whichever is minimum. For each document, the loop prints the document's content, followed by an empty line for readability.

```
def printResults(query,ranked_docs):
    print(query)
    print()
    for i in range(0,min(3,len(ranked_docs))):
        print(str(i + 1) + ". " , extract_file(documents[ranked_docs[i]][1]))
        print(documents[ranked_docs[i]][0])
        print("-----------------------------------")

input_file = open("phrase_input.txt","r")
query = input_file.readline()
ranked_docs = rank_docs_by_tfidf(query)
printResults(query,ranked_docs)
```

```
Medical reports

1.  AU127-1.docx
allstate austo insurance poliacy policy : issued to : m effective : p l e d o
c u m e anu127-1 t allstate insurance company stable of contents general 2
when and where the policy applies 2 changes 2 duty to report autos 2
combining limits of two or more autosa payment of benefits ; autopsy 9
consent of beneficiary 9 part 4 automobile disability income protection
coverage cw 9proof of claim ; medical reports 9 prohibited 2 transfer 2
```

cancellation 2 insuring agreement 9 insured persons 9 definitions 9 part 1 automobilemliability insurance exclusions what is not covered 9 coverages aa and bb 3 insuring agreement 3 to whom and when payment is made 10 proof of claim ; **medical reports** 10 padditional payments allstate will make 3 insured persons 4 part 5 uninsured motorists insurance coverage ss 10 insured autos 4 definitions 4 insuring agreement 10 insured persons 10 exclusions what is not covered 4 definitions 11 financial responsibility ....

---

2. AU127-1.docx

proof of claim ; **medical reports** as soon as possible , any person making claim must give us written proof of claim.the injured person may be required to take **medical** examinations by physicians we choose , as often as we reasonably require.we must be given authorization to obtain **medical reports** and copies of records pertinent to the claim .

Wildcard Queries

Initializing PorterStemmer Object for normalization of query terms and getting the stop_words corpus

```
# stemming porter object
stemmer = PorterStemmer()
root = Path("../")

stop_words = set(stopwords.words('english'))
```

Loading the inverted_index, transformed documents and file name binary files

```
my_path = root / "Pickled_files" / "Inverted_index"
dbfile = open(my_path, 'rb')
inverted_idx = pickle.load(dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Documents"
dbfile = open(my_path, 'rb')
docs = pickle.load(dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Files"
dbfile = open(my_path, 'rb')
files = pickle.load(dbfile)
dbfile.close()

def isnumber(token):
    for char in token:
        if not (char >= '0' and char <= '9'):
            return False
    return True
```

Trie

We are handling two types of wildcard queries : trailing and leading. We are using a Prefix and Suffix Trie for the same purpose. Every word is checked for stopword and **isnumber** condition, if its neither, the word is inserted in the Prefix Trie and the reverse of the word is inserted into the Suffix Trie. After the processing both Trie objects are pickled and dumped to a binary file for future use.

```
def insert(word , root) :
    cur = 0
    for char in word :
        if root[cur].get(char) == None :
            root[cur][char] = len(root)
            root.append(dict())
        cur = root[cur][char]
    root[cur]["end"] = True
    if root[cur].get("cnt") == None :
        root[cur]["cnt"] = 0
    root[cur]["cnt"] += 1
```

Retrieving Words from Trie

Any Wildcard Query has an attribute called **is_suf** which denotes whether the query was a Trailing Wildcard Query or not. The corresponding Query and the Trie Objects are passed to the **wildcard** function. Using the query we traverse over the Trie and upon its end we call a dfs traversal which collects the entire subtree of the point giving us the set of words for the particular wildcard query. Our Trie has a **cnt** attribute in it which signifies how many times the particular prefix/suffix was inserted and hence the retrieved set of words can easily be ranked in decreasing order of **cnt** to get the best 3 results.

```
def wildcard(word, root , is_suf) :
    ans = list()
    cur = 0
    for char in word :
        if root[cur].get(char) == None :
            return ans
        cur = root[cur][char]
    vis = dict()
    def dfs(token , id) :
        if vis.get(id) != None :
            return
        vis[id] = 1
        if root[id].get("cnt") != None:
            if is_suf :
                token = token[::-1]
            ans.append((token , root[id]["cnt"]))
        for labels in root[id].keys() :
            if len(labels) > 1 :
                continue
            new_str = token + labels
```

```
            dfs(new_str , root[id][labels])
    dfs(word , cur)
    ans = sorted(ans , key = lambda y : -y[1])
    ret_list = list()
    for i in range(min(3 , len(ans))) :
        ret_list.append(ans[i])
    return ret_list

Prefix = list()
Prefix.append(dict())
Suffix = list()
Suffix.append(dict())
```

Insertion into the Trie

```
for sentence in docs:
    sentence = sentence[0]
    tokens = nltk.tokenize.word_tokenize(str(sentence))
    new_token = list()
    for i in tokens:
        new_token.append(i.lower())
    tokens = new_token
    for words in new_token:
        if words in stop_words or len(words) <= 2 or isnumber(words):
            continue
        insert(words , Prefix)
        insert(words[::-1] , Suffix)

my_path = root / "Pickled_files" / "Prefix_wildcard"
dbfile = open(my_path, 'wb')
pickle.dump(Prefix, dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Suffix_wildcard"
dbfile = open(my_path, 'wb')
pickle.dump(Suffix, dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Prefix_wildcard"
dbfile = open(my_path, 'rb')
Prefix = pickle.load(dbfile)
dbfile.close()

my_path = root / "Pickled_files" / "Suffix_wildcard"
dbfile = open(my_path, 'rb')
Suffix = pickle.load(dbfile)
dbfile.close()
```

For the retrieved set of words from Trie, we normalize each term using PorterStemming and retrieve the corresponding posting list from the inverted index table.

```
def query(word , type) :
    ans = set()
    if type :
        word_list = wildcard(word , Suffix , True)
    else :
        word_list = wildcard(word , Prefix , False)
    for words in word_list :
        normalized_word = stemmer.stem(words[0])
        if inverted_idx.get(normalized_word) != None :
            for i in range(min(len(inverted_idx[normalized_word]), 2)):
                ans.add((docs[inverted_idx[normalized_word][i][0]][0],
files[inverted_idx[normalized_word][i][1]]))
    return ans

def extract_file(path) :
    ans = ""
    j = len(path) - 1
    while j >= 0 and path[j] != '\\' :
        ans += path[j]
        j -= 1
    return ans[::-1]
```

Wildcard Examples

```
results = query("ill" , False)
cnt = 0
for i in results :
    print(str(cnt + 1) + ") " , extract_file(i[1]))
    cnt += 1
    print(i[0])
    print("-----------------------------------")
```

```
1)  Property-Owner-Policy-Wording.docx
specified illnesses contingencies any occurrence of a specified illness at
the premises , except where the premises is a private dwelling any discovery
of an organism at the premises likely to result in the occurrence of a
specified illness , except where the premises is a private dwelling any
occurrence of legionellosis at the premises d the discovery of vermin or
pests at the premises e any accident causing defects in the drains or other
sanitary arrangements at the premises which causes restrictions on the use of
the premises on the order or advice of the competent local authority .
-----------------------------------
2)  1215E.2.docx
illegal use we wo n't pay for loss or damage caused in an incident : if you
are unable to maintain proper control of the automobile because you are
driving or operating the automobile while under the influence of intoxicating
substances ; if you are convicted of one of the following offences under the
criminal code of canada relating to the operation , care or control of the
automobile , or committed by means of an automobile , or any similar offence
under any law in canada or the united states : causing bodily harm by
criminal negligence dangerous operation of motor vehicles failure to stop at
```

the scene of an accident operation of motor vehicle when impaired or with more than 80 mg of alcohol in the blood refusal to comply with demand for breath sample causing bodily harm during operation of vehicle while impaired or over 80 mg of alcohol in the blood , or operating a motor vehicle while disqualified from doing so ; if you use or permit the automobile to be used in a race or speed test , or for illegal activity ; if you drive the automobile while not authorized by law ; and if another person , with your permission , drives or operates the automobile under any of these conditions .

---

3) complete-property-owner-policy-wording-policies-incepting-or-renewing-from-010418-acom686-11.docx

specified illnesses contingencies any occurrence of a specified illness at the premises , except where the premises is a private dwelling any discovery of an organism at the premises likely to result in the occurrence of a specified illness , except where the premises is a private dwelling any occurrence of legionellosis at the premises the discovery of vermin or pests at the premises any accident causing defects in the drains or other sanitary arrangements at the premises which causes restrictions on the use of the premises on the order or advice of the competent local authority .

---

4) BRIT-PO-Policy-Wording-May-2016-1.docx

any punitive or exemplary damages , compensations , fines or any penalties of whatsoever nature which the insured is ordered to pay by a forum , authority or body of competent jurisdiction ; in respect of coverage clause 1 eviction of squatters only : - a legal expenses incurred in relation to any dispute where the cause of action involves the insured ’ s legitimate tenant ; b any claim resulting from the occupation of the insured premises or part thereof by squatters prior to the inception of this policy ; c any action consciously taken by the insured that hinders the insurer or appointed representative or adversely affects the course of the legal proceedings initiated for the eviction of squatters ; in respect of coverage clause 5 criminal proceedings only , arising out of any criminal proceedings or allegations in respect of : the ownership , possession of or use of any vehicle ; or any investigation by hmrc or the department for work and pensions ; or assault , violence , fraud , conspiracy to defraud , dishonesty or malicious falsehood ; or the manufacture , dealing in or use of alcohol , illegal drugs or indecent or obscene materials ; or any illegal immigration ; or any money laundering offence under part 7 of the proceeds of crime act 2002 ; or bribery and corruption ; contravention of sanctions .

Runtime Analysis:

Dictionary is the data structure used for constructing the inverted_index which allows a fast access to the posting list for any given word in constant time. After the processing step, inverted_index is dumped onto a binary file which can be loaded in any file for querying purpose without repeatitive preprocessing. This allows the time for any one word query to be equivalent to time taken by PorterStemming to normalize it.

For wildcard Queries, we are using Prefix and Suffix Tries. Nodes of Tries are stored in a list where each node is a dictionary which allows constant time access as mentioned above. The time complexity to traverse a particular string in Trie will be O(n) where n is the length of the query string. The worse case time complexity of returning the results for a particular subtree in Trie is $O(\max(N_1 \ldots\ldots N_m))$ where N_i is the length of ith string inserted.

For boolean retrieval system, inverted_index is used to get the various posting lists corresponding to each term in the query, further on which set intersection / union operations are applied which takes about $O(\max(S_1 \ldots S_k))$ where S_i is the size of posting list of ith term.