Twitter Search Engine

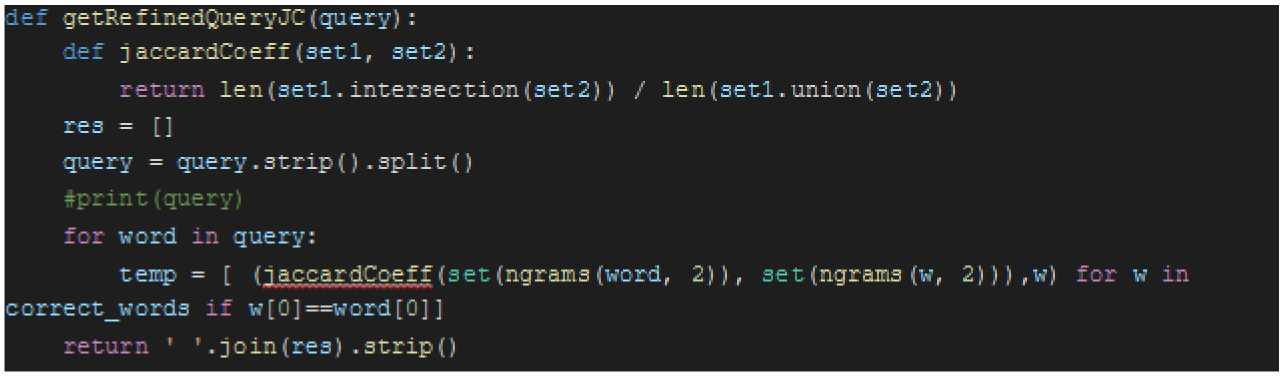
-Group 20

Group Members:

- Logu R(S20190010111)
- Harshith Jupuru (S20190010072)
- Sree Nitish B (S20190010166)
- Koushik Bhukya(S20190010018)

Project Structure:
queries/
admin.py
apps.py
models.py
test.py
urls.py
features.py

-getRefinedQueryJC(query)

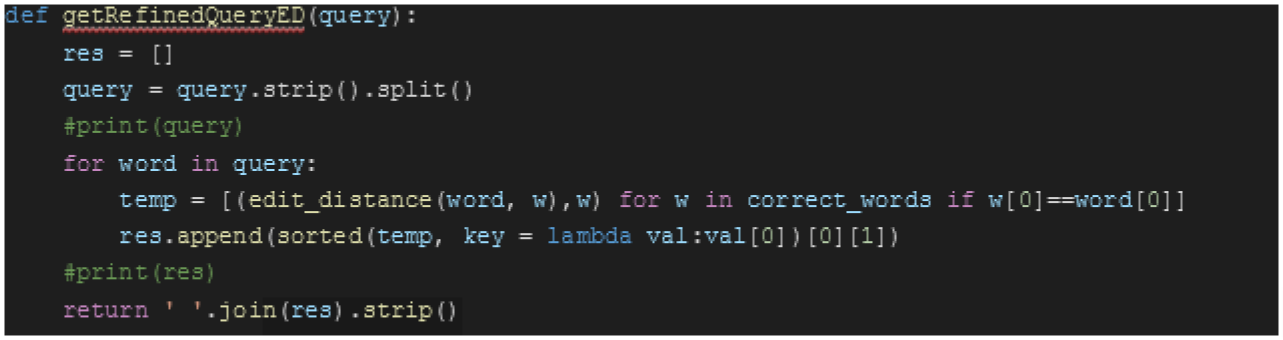


```
def getRefinedQueryJC(query):
    def jaccardCoeff(set1, set2):
        return len(set1.intersection(set2)) / len(set1.union(set2))
    res = []
    query = query.strip().split()
    #print(query)
    for word in query:
        temp = [ (jaccardCoeff(set(ngrams(word, 2)), set(ngrams(w, 2))),w) for w in
correct_words if w[0]==word[0]]
    return ' '.join(res).strip()
```

Input : Query string, Output : Refined query string
Uses nltk.corpus.words where correctly spelled words were present

Finds the jaccard coefficient for each term in query (bi-grams) with each term in nltk.corpus.words to find the nearby words to correct the spelling of query terms. It will replace the terms with the most matching term from nltk.corpus.words.

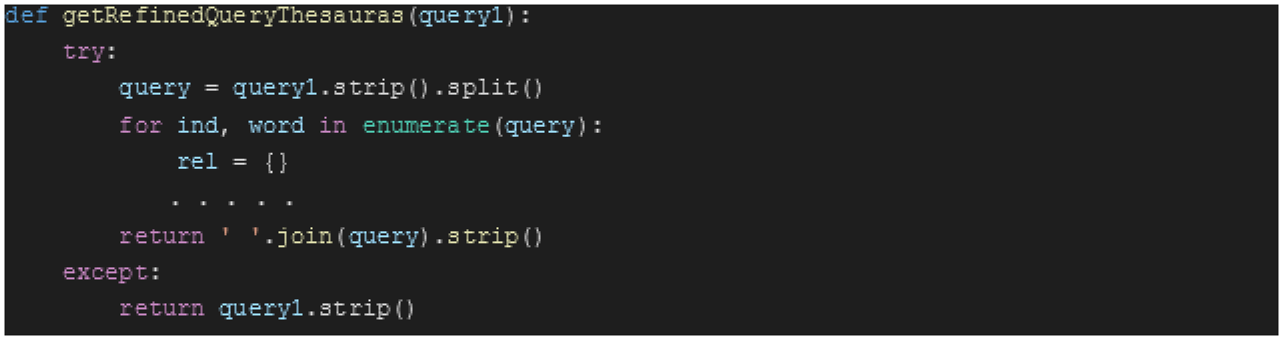
--getRefinedQueryED(query)



```
def getRefinedQueryED(query):
    res = []
    query = query.strip().split()
    #print(query)
    for word in query:
        temp = [(edit_distance(word, w),w) for w in correct_words if w[0]==word[0]]
        res.append(sorted(temp, key = lambda val:val[0])[0][1])
    #print(res)
    return ' '.join(res).strip()
```

Input : Query string, Output : Refined query string
Uses nltk.corpus.words to find the edit distance for each term which includes addition, deletion, substitution, transposition to find the nearby words and replace them in query string.

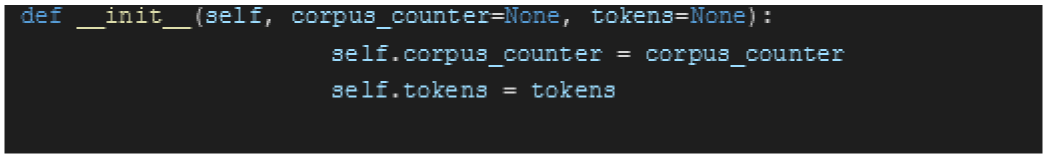
--getRefinedQueryThesauras(query)



```
def getRefinedQueryThesauras(query1):
    try:
        query = query1.strip().split()
        for ind, word in enumerate(query):
            rel = {}
            . . . . .
        return ' '.join(query).strip()
    except:
        return query1.strip()
```

Input : Query string, Output : Refined query string
Uses wordnet to find synonyms for query terms to improve the recall.

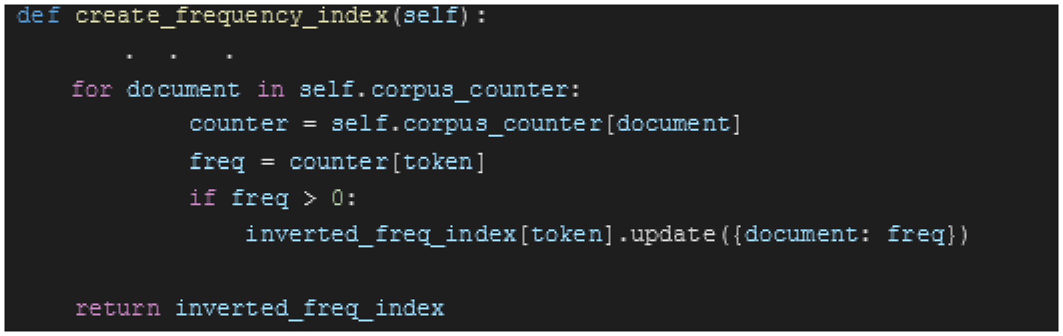
indexing.py -
class Indexer
-init



```
def __init__(self, corpus_counter=None, tokens=None):
                        self.corpus_counter = corpus_counter
                        self.tokens = tokens
```

Initializes the indexer class instance

-create_frequency_index

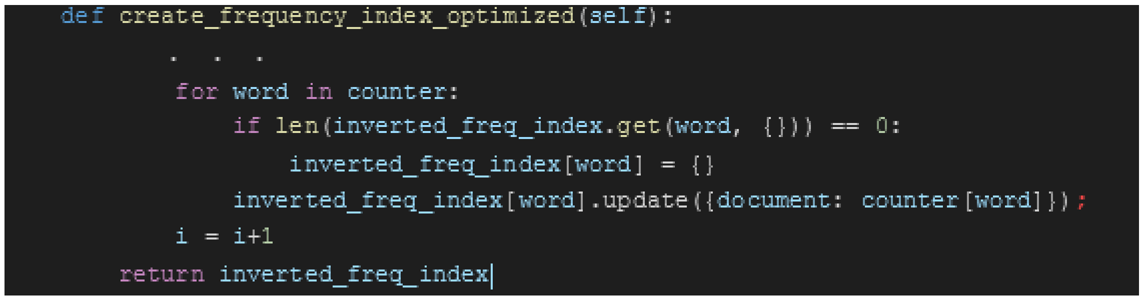


```
def create_frequency_index(self):
        .  .   .
    for document in self.corpus_counter:
            counter = self.corpus_counter[document]
            freq = counter[token]
            if freq > 0:
                inverted_freq_index[token].update({document: freq})

    return inverted_freq_index
```

Create frequency_index which will return inverse document frequency as follows:
{"Index1/token1":{docId1:freqOfToken1InDoc1, docId2:freqOfToken1InDoc2,...},
"Index2/token2":{docId1:freqOfToken2InDoc1, docId2:freqOfToken2InDoc2,...},...}
where index/token is a unique term in the corpus
The above was done for each token for each document if the token present in document then it is added in the inverted_freq_index dictionary which takes a time of numOfTokens*numOfDocs

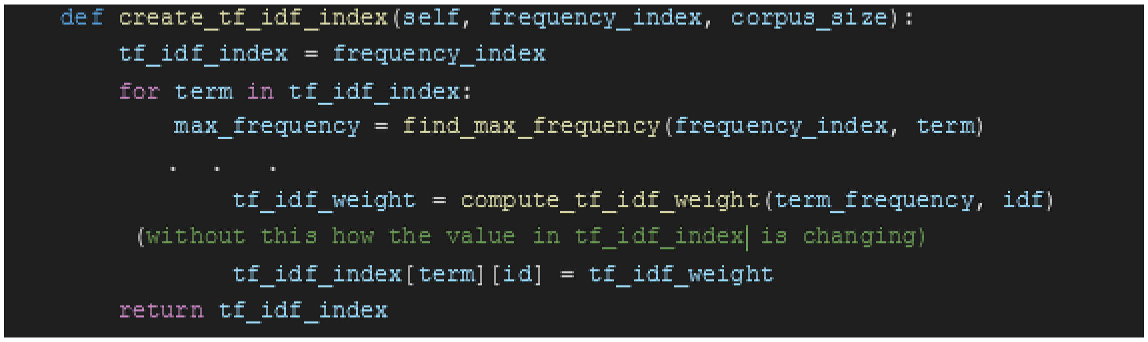
-create_frequency_index_optimized



```
def create_frequency_index_optimized(self):
        .  .  .
        for word in counter:
            if len(inverted_freq_index.get(word, {})) == 0:
                inverted_freq_index[word] = {}
            inverted_freq_index[word].update({document: counter[word]});
        i = i+1
    return inverted_freq_index
```

Creates an inverted index with hashes using message id as key and frequency as value
Similar to above create_frequency_index but here for each document for each token we create the indexing.

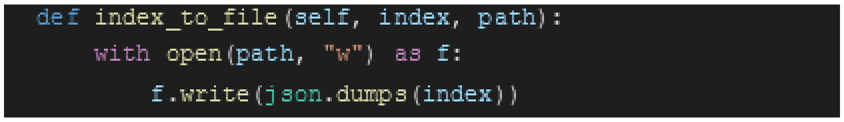
-create_tf_idf-index



```
def create_tf_idf_index(self, frequency_index, corpus_size):
    tf_idf_index = frequency_index
    for term in tf_idf_index:
        max_frequency = find_max_frequency(frequency_index, term)
        .  .   .
            tf_idf_weight = compute_tf_idf_weight(term_frequency, idf)
     (without this how the value in tf_idf_index is changing)
            tf_idf_index[term][id] = tf_idf_weight
    return tf_idf_index
```

Create tf_idf for each document by passing above created frequency_index i.e, inverse document frequency and length of corpus i.e, number of total documents.

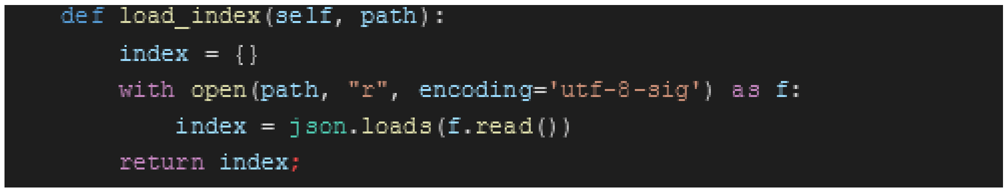
-index_to_file



```
 def index_to_file(self, index, path):
     with open(path, "w") as f:
         f.write(json.dumps(index))
```

Saves the indexed data in given location

-load_index



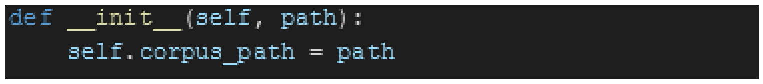
```
    def load_index(self, path):
        index = {}
        with open(path, "r", encoding='utf-8-sig') as f:
            index = json.loads(f.read())
        return index;
```

Loads the created indexes stored at the given path

preprocessor.py -

class Preprocessor

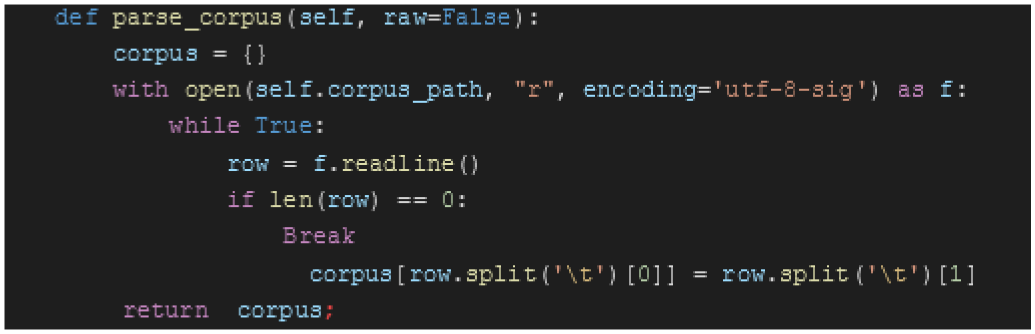
-init



```
def __init__(self, path):
    self.corpus_path = path
```

Initialize the preprocessor instance

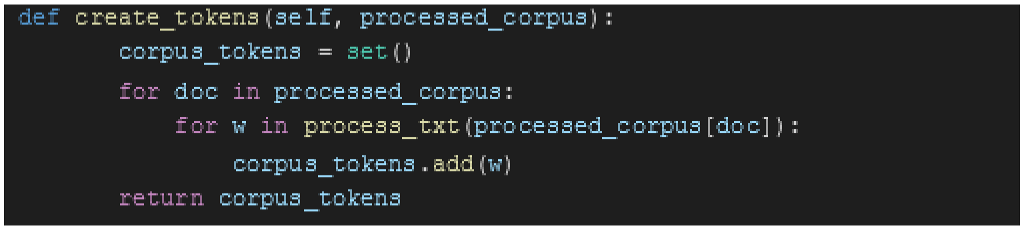
-parse_corpus



```
    def parse_corpus(self, raw=False):
        corpus = {}
        with open(self.corpus_path, "r", encoding='utf-8-sig') as f:
            while True:
                row = f.readline()
                if len(row) == 0:
                    Break
                      corpus[row.split('\t')[0]] = row.split('\t')[1]
        return  corpus;
```

Create corpus which will return the corpus in the format as below:
{‘docId1’:’documentContent1’, ‘docId1’:’documentContent1’,....}

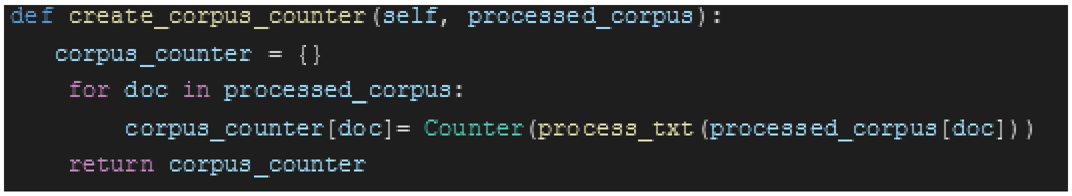
-create_tokens



```
def create_tokens(self, processed_corpus):
        corpus_tokens = set()
        for doc in processed_corpus:
            for w in process_txt(processed_corpus[doc]):
                corpus_tokens.add(w)
        return corpus_tokens
```

Create tokens by passing the above created corpus and the output is as follows:
(‘Set’, ‘of’, all’, ‘unique’, ‘tokens’,.....)

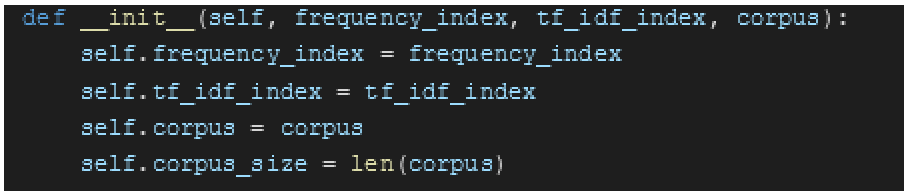
-create_corpus_counter



```
def create_corpus_counter(self, processed_corpus):
    corpus_counter = {}
     for doc in processed_corpus:
         corpus_counter[doc]= Counter(process_txt(processed_corpus[doc]))
     return corpus_counter
```

Create corpus counter by passing the corpus created and the result will be stored in c_counter as follows:
{docId1:{"term1inDoc1":freq, "term2inDoc1":freq,...}, docId2:{"term1inDoc2":freq, "term2inDoc2":freq,...},....}
Ex: corpus_counter[doc_id] = {"fifa": 2, "world": 1, "cup": 1, …}.

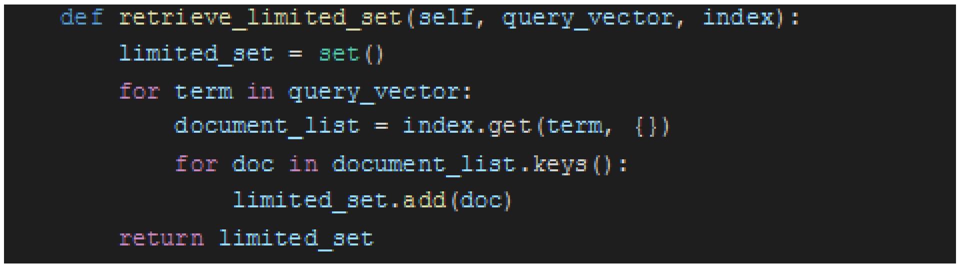
query.py -
class Query
-init



```
def __init__(self, frequency_index, tf_idf_index, corpus):
    self.frequency_index = frequency_index
    self.tf_idf_index = tf_idf_index
    self.corpus = corpus
    self.corpus_size = len(corpus)
```

Initialize the query instance

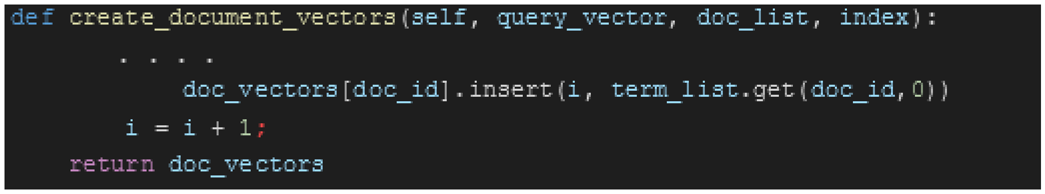
-retrieve_limited_set



```
    def retrieve_limited_set(self, query_vector, index):
        limited_set = set()
        for term in query_vector:
            document_list = index.get(term, {})
            for doc in document_list.keys():
                limited_set.add(doc)
        return limited_set
```

Return a set of documents id’s which contains at least one query term in the document

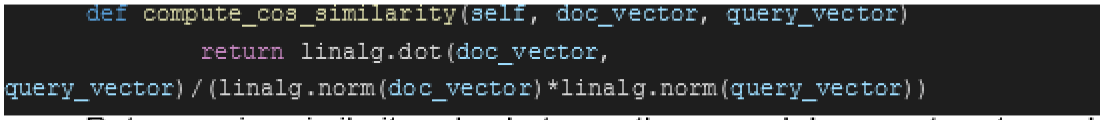
-create_document_vectors



```
def create_document_vectors(self, query_vector, doc_list, index):
        . . . .
            doc_vectors[doc_id].insert(i, term_list.get(doc_id,0))
        i = i + 1;
    return doc_vectors
```

Return the document vectors in the format as follows:
{doc1Id:[queryterm1: tfidfWeight, queryterm2: tfidfWeigth, ..., querytermN: tfidfWeight], doc2Id:[queryterm1: tfidfWeight, queryterm2: tfidfWeigth, ..., querytermN: tfidfWeight],...upto docM:[...]}

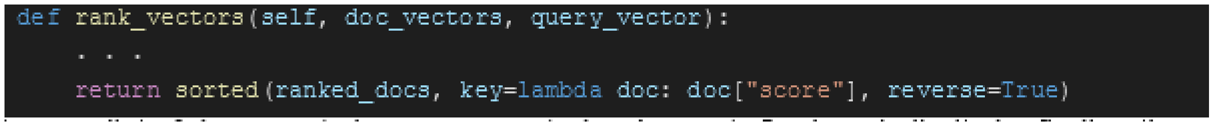
-compute_cos_similarity



```
      def compute_cos_similarity(self, doc_vector, query_vector)
              return linalg.dot(doc_vector,
query_vector)/(linalg.norm(doc_vector)*linalg.norm(query_vector))
```

Return cosine similarity value between the passed document vector and query vector by finding dot product between those two normalized vectors.

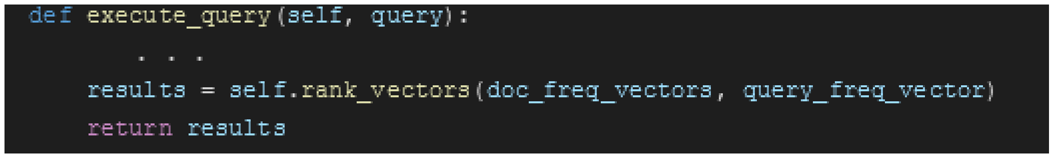
-rank_vectors



```
def rank_vectors(self, doc_vectors, query_vector):
    . . .
    return sorted(ranked_docs, key=lambda doc: doc["score"], reverse=True)
```

Returns a list of documents in reverse sorted order w.r.t. Cosine similarity by finding the cosine similarity with the query vector. The output format is as follows:
[{‘id’ : ’DocumentId1’, ‘score’ : cosine_similarity_score}, {‘id’ : ’DocumentId2’, ‘score’ : cosine_similarity_score}, {‘id’ : ’DocumentId2’, ‘score’ : cosine_similarity_score},....]
Ex: [{‘id’ : ‘349514522081361 92’, ‘score’ : 0.98342567}, …..]

-execute_query



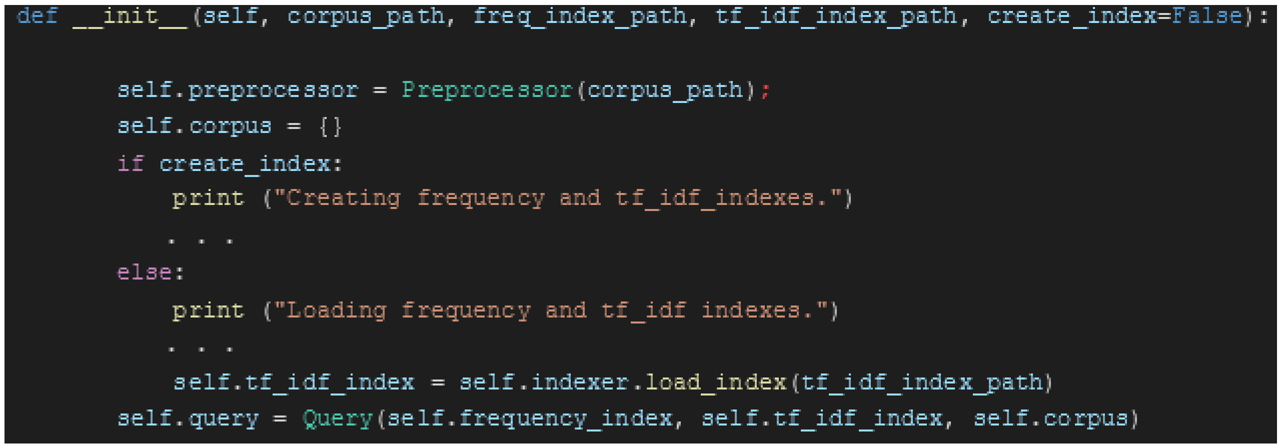
```
def execute_query(self, query):
        . . .
    results = self.rank_vectors(doc_freq_vectors, query_freq_vector)
    return results
```

Process the query, retrieve limited documents i.e, documents contains at least one query term, create document vectors with query terms, rank the documents using cosine similarity.

system.py -

class System

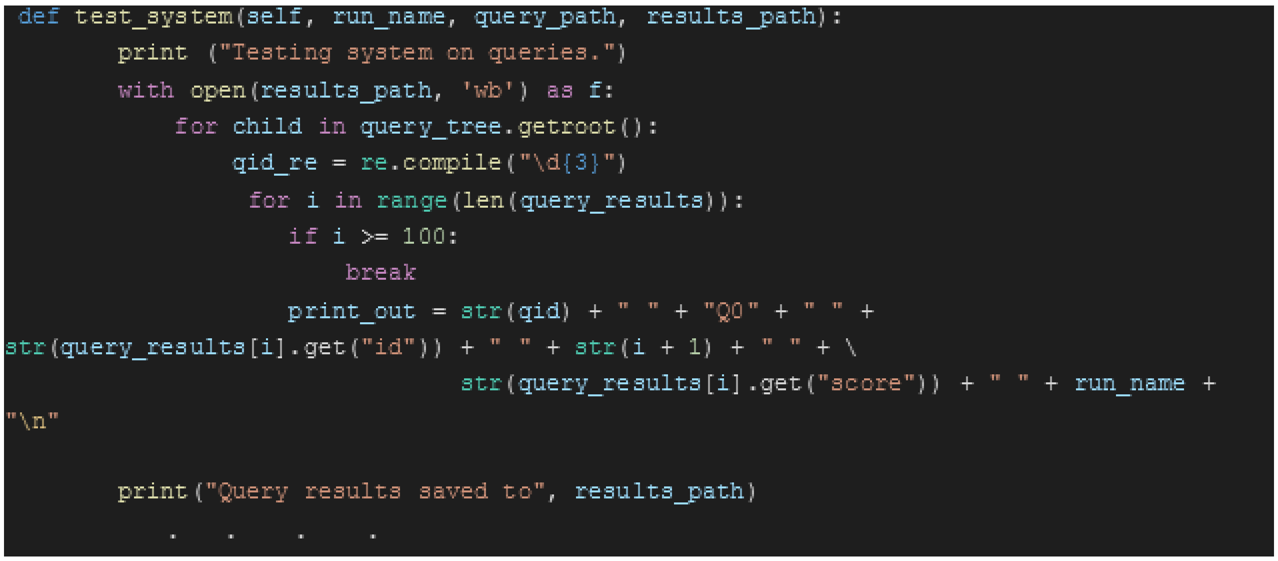
-init



```
def __init__(self, corpus_path, freq_index_path, tf_idf_index_path, create_index=False):

        self.preprocessor = Preprocessor(corpus_path);
        self.corpus = {}
        if create_index:
            print ("Creating frequency and tf_idf_indexes.")
            . . .
        else:
            print ("Loading frequency and tf_idf indexes.")
            . . .
            self.tf_idf_index = self.indexer.load_index(tf_idf_index_path)
        self.query = Query(self.frequency_index, self.tf_idf_index, self.corpus)
```

-test_system

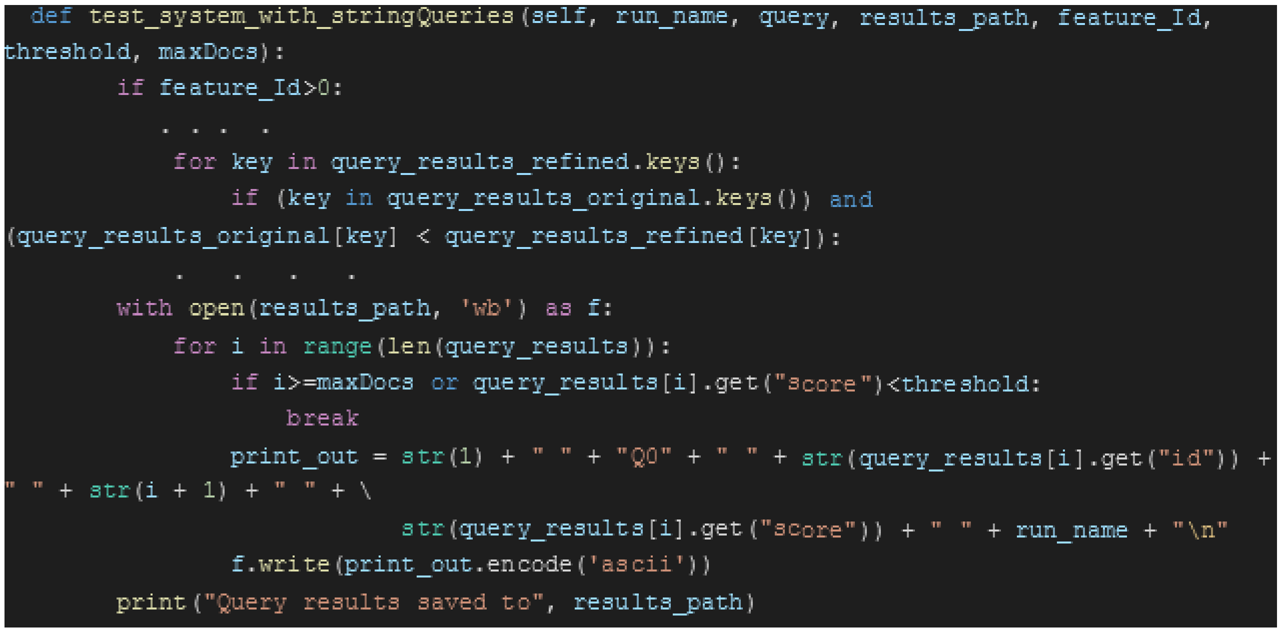


```
def test_system(self, run_name, query_path, results_path):
        print ("Testing system on queries.")
        with open(results_path, 'wb') as f:
            for child in query_tree.getroot():
                qid_re = re.compile("\d{3}")
                 for i in range(len(query_results)):
                    if i >= 100:
                        break
                    print_out = str(qid) + " " + "Q0" + " " + 
str(query_results[i].get("id")) + " " + str(i + 1) + " " + \
                                str(query_results[i].get("score")) + " " + run_name + 
"\n"

        print("Query results saved to", results_path)
            .    .     .     .
```

Input: query path(file location with list of queries), results path(file location to save the results)
For each query it calls Query.execute_query function by passing the query and getting the documents in ranked order w.r.t. Cosine similarity and store the result in the following format - “ queryId Q0 DocumentId RankOfDocument CosineSimilarityScore runname ” in the given result location path

-test_system_with_stringQueries



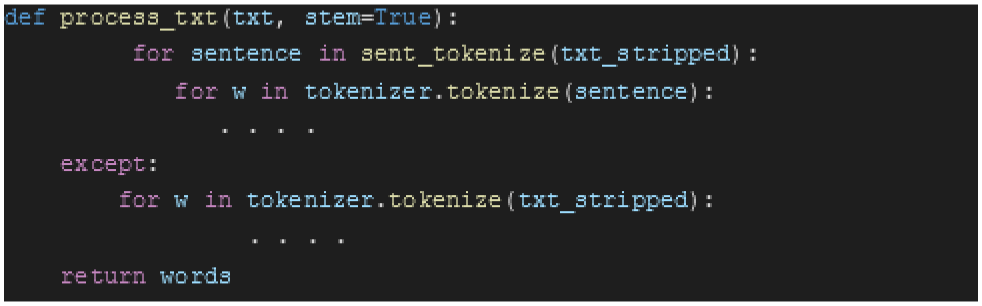
```
 def test_system_with_stringQueries(self, run_name, query, results_path, feature_Id, 
threshold, maxDocs):
        if feature_Id>0:
           .  .  .   .
            for key in query_results_refined.keys():
                if (key in query_results_original.keys()) and 
(query_results_original[key] < query_results_refined[key]):
            .    .    .    .
        with open(results_path, 'wb') as f:
            for i in range(len(query_results)):
                if i>=maxDocs or query_results[i].get("score")<threshold:
                    break
                print_out = str(1) + " " + "Q0" + " " + str(query_results[i].get("id")) + 
" " + str(i + 1) + " " + \
                            str(query_results[i].get("score")) + " " + run_name + "\n"
                f.write(print_out.encode('ascii'))
        print("Query results saved to", results_path)
```

test_system_with_stringQueries
Inputs: run_name, query, results_path, feature_Id, threshold, maxDocs
Query: single string
Results_path: Location to store the result of fetched results

feature_Id:
0: Results for original query
1: Results for both original and Refined query using jaccard coefficient
2: Results for both original and Refined query using edit_distance
3: Results for both original and Refined query using Thesaurus
Threshold: Cosine similarity value above which only documents will be in the result
maxDocs: Maximum documents to store as result

utils.py -

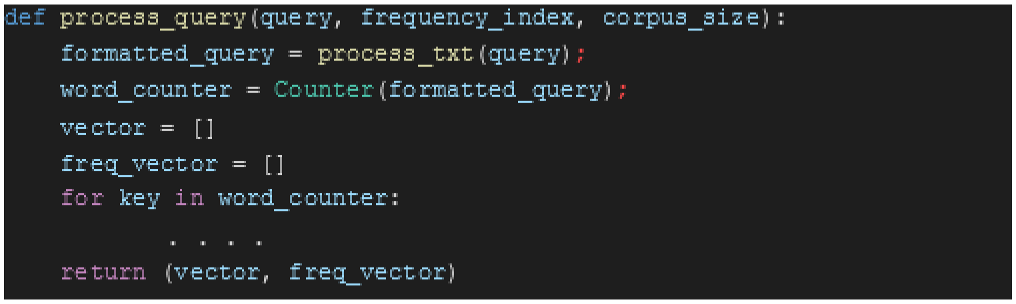
-process_txt



```
def process_txt(txt, stem=True):
         for sentence in sent_tokenize(txt_stripped):
            for w in tokenizer.tokenize(sentence):
               . . . .
    except:
        for w in tokenizer.tokenize(txt_stripped):
                 . . . .
    return words
```

It will process the text by cleaning the data using regular expressions using re, lowering the case of strings, tokenization using TreebankWordTokenizer, stemming using EnglishStemmer, removing the stopwords using nltk ect.

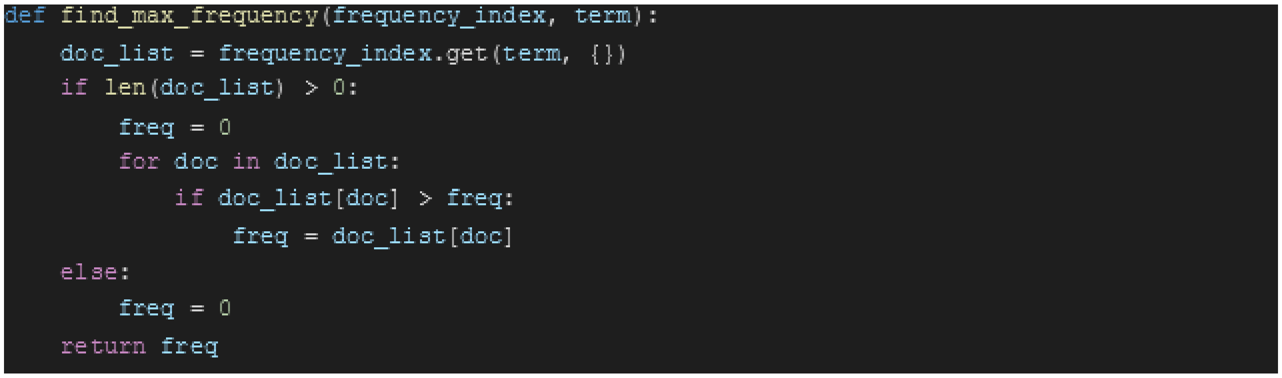
-process_query



```
def process_query(query, frequency_index, corpus_size):
    formatted_query = process_txt(query);
    word_counter = Counter(formatted_query);
    vector = []
    freq_vector = []
    for key in word_counter:
           . . . .
    return (vector, freq_vector)
```

Processes the query by finding the tokens and the frequency to find the weight of each token and the function will return two lists where former list contains the unique tokens and the latter list contains the idf value of the token found by considering its frequency as follows: idf value of a token is (0.5+0.5*term_frequency)*idf, where term_frequency and idf were found by using compute_term_frequency, compute_idf respectively.

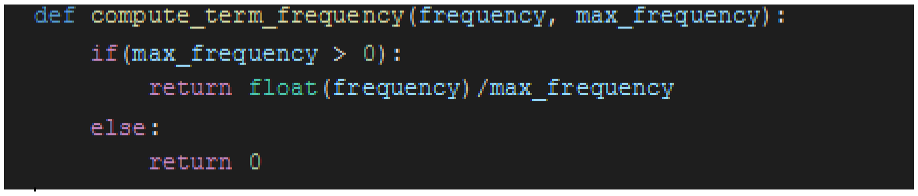
-find_max_frequency



```
def find_max_frequency(frequency_index, term):
    doc_list = frequency_index.get(term, {})
    if len(doc_list) > 0:
        freq = 0
        for doc in doc_list:
            if doc_list[doc] > freq:
                freq = doc_list[doc]
    else:
        freq = 0
    return freq
```

Finds the maximum frequency a term appears in the corpus

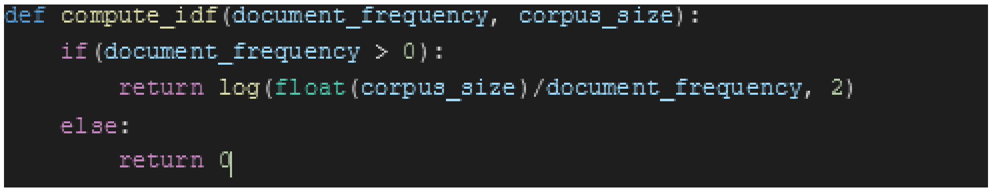
-compute_term_frequency



```
def compute_term_frequency(frequency, max_frequency):
    if(max_frequency > 0):
        return float(frequency)/max_frequency
    else:
        return 0
```

Computes the term_frequency by dividing the frequency of term with maximum frequency a term appears in the corpus which was found above.

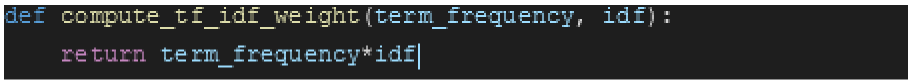
-compute_idf



```
def compute_idf(document_frequency, corpus_size):
    if(document_frequency > 0):
        return log(float(corpus_size)/document_frequency, 2)
    else:
        return 0
```

Finds idf value for a term by calculating the log of size of corpus divided by the document frequency of a term base 2.

-compute_tf_idf_weight



```
def compute_tf_idf_weight(term_frequency, idf):
    return term_frequency*idf
```

Computes the tf_idf by multiplying the idf value of a term with its document frequency in the given document.

view.py -
mapping the format in which to call the function

Prerequisites

1. pip install -r requirements.txt
2. Download nltk words (for query refinement)
    a. nltk.download('words')

To Create Indexing

In the project folder , Twitter_IRS/IR_System ,

1) Run System.py
2) This Creates the Indexing for the respective dataset and est. time is 25-30 minutes (SSD), this test depends on the system.

To Run System to retrieve Twitter Messages

In the project folder, Twitter_IRS/

1) python manage.py runserver
2) Open https://127.0.0.1:8000/

Runs on port 8000