Assignment 1
DOCUMENTATION

~Aditya Mohan Mishra
2018ME10582

1.General Information regarding functions.

invidx_cons.py

```
preprocesser(input_data_list, min_len=1):
    '''
    list<soup_obj> ->   dict<str -> int>

    input:
        input_data_list: list of all text corpus in one document(beautifulsoup object).
        min_len: min length of string to be considered a token. default is 1.

    output:
        token_dictionary: token -> term frequency dictionary.
    '''
extracter(file):
    '''
    str  ->  dict<string -> list<bs4_obj> >

    input:
        file: file path of one corpus file

    output:
        d: dict mapping from doc_no to list of bs4 obj of text contained.
    '''
generate_inverted_index(path_dir):
    '''
    str-> dict< str->  dict<key->value> > , str

    input:
        path_dir: directory path

    output:
```

vocab: the Inverted index of whole corpus. Maps string to its df and posting's list in string form.

doc_embed: the embeddings used for the document nos in memory, and document vector's normalisation constant.

dict: str->(dict: int docno->int score)

'''
write_dict_to_file(vocab, doc_embed, dict_file, idx_file):
'''
dict< str-> dict<key->value> >, str , str, str --> None
** Creates two files.
input:
vocab: maps term to it's postings list string and document frequency
doc_embed: a string mapping doc_embeddings to it's document no and normalization constant.
dict_file: path where we need to create the dict file
idx_file: path where we need to create the index file

files outputted:
outputs index file by zlib compression and binary encoding.
updates the vocab dict to include offset and length of the postings list.
converts the vocab dict into a char seperated string.
utf8 encoded and then compressed via zlib and outputted.

'''
train(corpus_path,dict_file,idx_file):
'''
str, str, str-> 2 files outputted

input:
corpus_path: path of directory containing the corpus files
dict_file: path of dictionary file to be outputted
idx_file: path of index file to be outputted
output:
outputs the two required files.

'''

vecsearch.py

```
preprocess(filepath, vocab, doc_embed):
    '''Queryfile -> list of queries -> list of query tokens vs scores
    str, dict< str->dict< key->value > > , dict< str->[str,float] >   -->  dict< int->dict< str->float > >
    input:
        filepath: path of file contain query, having proper format
        vocab: dictionary mapping terms to the offset of posting lists, length of the its list and the
document frequency
        doc_embed: maps embedded id to document name and normalisaion factor.

    output:
        queries: dictionary mapping query no to a dictionary containing terms and thier respective
tf.idf value


    '''
tokenize(query, vocab, doc_embed):
    '''
    str, dict< str->dict< key->value > > , dict< str->[str,float] > -->  dict< str->float >

    input:
        query: string containing the relavent query
        vocab: dictionary mapping terms to the offset of posting lists, length of the its list and the
document frequency
        doc_embed: maps embedded id to document name and normalisaion factor.

    output:
        score: dictionary mapping the term to their tf.idf value.
    '''
retrive_qrels(query_filepath,vocab_file, idx_file,output_filepath ,k) :
    '''
    str, str, str, str, int -> output file

    input:
        query_filepath: path to a file containing keyword queries, with each line corresponding to a
query.
        vocab_file:path to the vocabulary file
        idx_file: path to the indexfile
        output_filepath: path to the file where final result is to be outputed.
        k: default = 10:  number spcifying how many top-scoring results have to be returned for
each query.
```

```
    output:
        Qrels file.
    '''
top_k(query_tokens,vocab,f,doc_embed,k):
    '''queries -> list_of_documents in descending order of score. (if this is less than k then retrive
the rest from the doc list
    dict< str->float >, dict< str->dict< key->value > > , file object, dict< str->[str,float] >, int -> dict<
str,float >

    input:
        query_tokens:dictionary mapping the term to their tf.idf value
        vocab: dictionary mapping terms to the offset of posting lists, length of the its list and the
document frequency
        doc_embed: maps embedded id to document name and normalisaion factor.
        f: file object on posting list file
        k: cut off

    output:
        topDoc: Documents (with thier scored mapped to them) in Desending order of thier score.

    '''
get_score(term,term_score,vocab,f,N,doc_embed,pst_lst = defaultdict(lambda:0)):
    '''
    str, float, dict< str->dict< key->value > > ,file obj, int, dict< str->[str,float] >, dict<str, float> -->
dict<str, float>

    input:
        term: the query term.
        term_score: float value of tfidf of the term
        vocab: dictionary mapping terms to the offset of posting lists, length of the its list and the
document frequency
        doc_embed: maps embedded id to document name and normalisaion factor.
        f: file obj of posting's lists
        N: total no of documents(to compute idf)
    output:
        pstlst: Returns documents mapped with the relevent part of VSR cosine score(the score
due to current term is added.)
    '''
extract(vocab_file):
    '''
    str -> dict< str->dict< key->value > > , dict< str->[str,float] >

    input:
```

```
    vocab_file: The path to the dict file

  output:
    vocab: dictionary mapping terms to the offset of posting lists, length of the its list and the
document frequency
    doc_embed: maps embedded id to document name and normalization factor.
  '''
```

2.File storage and access:

All the objects' data are converted into string using various characters to separate different entries and values. Further those strings are converted into binary encoding, utf-8 here but can be converted to ascii too. Then the binary streams are further compressed by zlib's compress and decompress functions, zlib is a standard python library. Then they're written to a file.

In case of posting lists we store the starting byte and length of the compressed representation of string against the term in the dictionary. When we need he posting list of a term, we generate a file object, use seek() to move the pointer to the start, use read() to get the relevant part out. Then we decompress, decode and then parse the string to get the postings list.

3.Parsing of Query files:

Regex is used to parse the query files and then preprocess them. <title> and <num> tags are used to identify the position of relevant text. Any format of the two suggested on teams is compatible with this piece of code.

4.Parsing of corpus:

Beautiful Soup is used to extract text from the tags easily. Then punctuation is removed. Then stopwords are removed using the stopword list in nltk. Further Porter stemmer and tokenizer are used to generate tokens for the corpus. Also the NER tagging is handled separately to create extra tokens.

5.Prefix Search:

Query having prefix operator is compiled into a regex pattern and then searched through he vocabulary list. The favorable terms are added to the query tokens.

6.Name entity restriction:

A prefix of p:,l:,o:, or n: is appended to restrict the options and similar changes are done in query processing. This handles the Name entity restriction on its own.