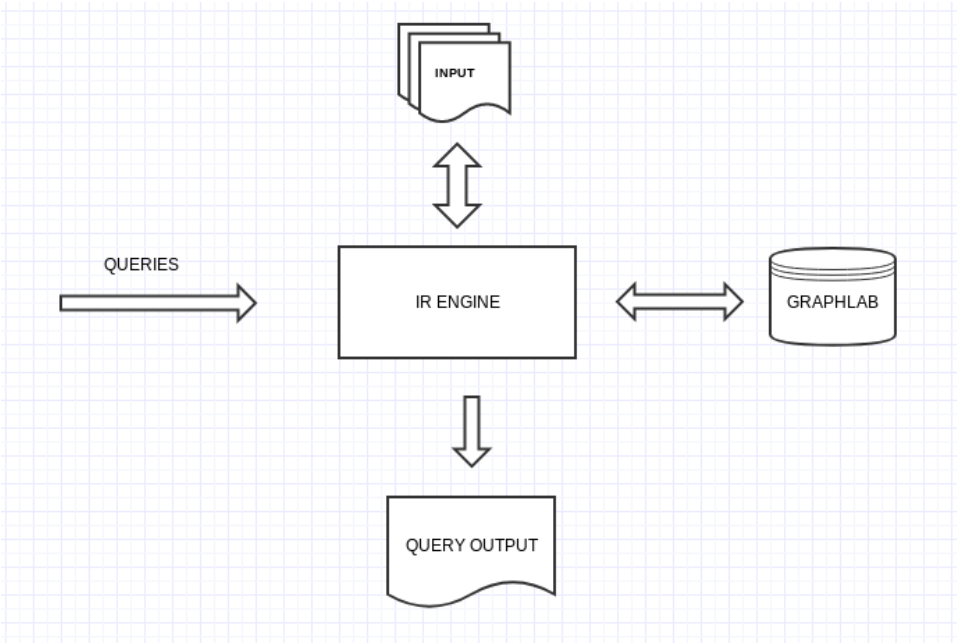
Introduction

This assignment is concerned with building Simple IR Tool, which includes Text Preprocessing and building Positional Inverted Index and Search Execution. It builds on the material covered in the first three labs.

GraphLab

GraphLab is a python package that allows programmers to handle large scale data.This assignment makes use of graphlab extensively.

ARCHITECTURE



At First, Each document is parsed by the document class and fed into the engine in a sequential manner. Engine transforms the parsed document in to a dictionary of key and value pairs. Once all the document is parsed and converted into dictionary pairs,the engine collects the dictionary pairs and feds into graphlab.

Each row in graphlab is subjected to preprocessing which is a pipeline process. At first, each row is fed into tokenization layer and then a layer of stopword removal and finally a layer of stemming. To illustrate each layer, tokenization makes use of regular expressions. Below snippet is the cross section layer of tokenization in a pipe line process.

```
def _tokenize(self,row):
    return filter(lambda word:re.match("^[\w]+$",word),[word for word in re.split("[\W+]",row)])
```

And then stopwords layer of a pipeline makes use of a file with a list of stopwords in it,which actually traverses through each word of a row, examines whether it is a stopword or not.Once a stopword is identified, it gets removed from that row.And stemming is implemented by making use of python library called nltk. Choice of Stemming library goes as per user. I have chosen, Porter Stemmer, reason behind it was, ease to use.

```
def _stemming(self,row):
    return map(lambda word: self.pStemmer.stem(word),row)

def _tokenize(self,row):
    return filter(lambda word:re.match("^[\w]+$",word),[word for word in re.split("[\W+]",row)])

def _rmove_stopWords(self,row):
    return [word for word in row if word not in self._stopWords ]
```

Another instance of graphlab is created. Each row that comes out of pipeline process is transformed into a dictionary of key and value pairs. For better understanding, below is a snippet that takes each row and then converts into a dictionary. Here key is nothing but the word and the value is a dictionary which as its own set of key and value pairs. i.e each key and value is the document and list of positions the term appeared in that document.Figuring out the index of each word in a document simulatenously happens while creating a dictionary of keys as douments and values are positions.

```
fn = lambda row: row['HEADLINE']+' '+row['TEXT']

_combined_cols = self._apply_preprocessing(['HEADLINE','TEXT'],fn)

self._service._add_column('RAW_TEXT',_combined_cols)

fn = lambda row:self._pre_processing._action(row)

docs_tokenized = self._apply_preprocessing('RAW_TEXT',fn)

doc_ids = self._service._fetch_column('DOCID')

_cache = self._combine_SArray(doc_ids,docs_tokenized)

fn = lambda term: self._update_term(term[0],term[1],docId)

for docId,eWord in _cache:
    map(fn,list(enumerate(eWord)))

return self._cache_term
```

Below is a snippet of the above process and as well the snapshot. The reason behind using this sort of transformation, is to achieve flexibilty on further accessing the document for a particular term.

One major part to achieve is implementing positional inverted index. But having such an above methodology of organzing the terms and documents would make the process of implementing the positional inverted index much easier. To fetch positional inverted index for a term, all i need to do is , fetch the key which is the term, and the result (i.e) the value is the dictionary of documents whose key is the document id and the value is the positions. Below is the snippet which is responsible for achieving positional inverted index

Term : {'Doc Id': [1,2,34] (List of positions)}

```
def _update_term(self,index,term,docId):

    if self._cache_term.get(term,False):
            if self._cache_term[term].get(docId,False):
                self._cache_term[term][docId].update([index])
            else:
                self._cache_term[term][docId] = set([index])
    else:
            self._cache_term[term][docId] = set([index])
```

SEARCH

Analyzing the queries and executng them was challenging . Once the query is submitted to engine, each query is read sequentially by making use of regular expression. Then segregating them into type of query i.e whether it is a phrase search or proximity search or boolean search by means of flag.

```
def _make_query(self,query):

    _ps = ps()
    sub_queries = shlex.split(query)

    _and_query =  re.compile(r'(.+)(\bAND\b|\bOR\b)(.+)')
    _hash_query = re.compile(r'\s?#\s?(\d+)\s?\(\s?([a-z]+)\s?,\s?([a-z]+)\s?\)\s?$')

    if _and_query.match(query):

        _result = _and_query.search(query).groups()

        _lterm,lflag = self._find_not_in_term(_result[0])

        _rterm,rflag = self._find_not_in_term(_result[2])

        self.enqueue({'lterm' :_lterm.lower(),'operator':_result[1],'rterm' :_rterm.lower(),'lflag':lflag,'rflag':rflag})

    elif _hash_query.match(query):

        _result =  _hash_query.search(query).groups()
        term = '#'+_result[0]+'('+_result[1].lower()+','+_result[2].lower()+')'
        self.enqueue({'lterm' :term})
    elif len(sub_queries) == 1:
        fn = lambda word:word.lower()
        sub_queries = map(fn,sub_queries)
        _quotes = '"'
        if len(sub_queries[0].split(' ')) > 1:
            self.enqueue({'lterm' :_quotes+sub_queries[0]+_quotes})
        else:
            self.enqueue({'lterm':sub_queries[0]})
```

Then, once identifying the type of query, it is then fed into a queue by means of QueueExecuter.
As we know, Queue datastructure follows first in first out. So the first query that gets in , gets executed first. To illustrate each type of search bit further, i prefer to go with an example.

Boolean search:

Term_1 AND Term_2

To execute the above query, i have implemented few supplementary functions, which would ease the process

For Instance

self._fetch_column_by_word(key) -> returns the dictionary of documents and its positions of a term

I would fetch keys alone, which is the document id's and stored in a list. Similary the same function is used to get the document ids of another term, which is stored in a list Now a set operation i.e **Intersection** is applied on both the lists and returns the common documents

```
def _parse(self,lcollection,rcollection,operator):

    _gl_type = gl.data_structures.sframe.SFrame


    if type(lcollection) is list and type(rcollection) is _gl_type:

        self._lcollection = lcollection
        self._rcollection = rcollection[0]['INFO'].keys()

        print 'self._rcollection',self._rcollection

    elif type(lcollection) is _gl_type and type(rcollection) is list:

       self._lcollection = lcollection[0]['INFO'].keys()
       self._rcollection = rcollection

    elif type(lcollection) is list and type(rcollection) is list:

        self._lcollection = lcollection
        self._rcollection = rcollection

    elif type(lcollection) is _gl_type and type(lcollection) is _gl_type:

        self._lcollection = lcollection[0]['INFO'].keys()
        self._rcollection = rcollection[0]['INFO'].keys()



    if operator == 'AND':
        return list(set(self._lcollection).intersection(set(self._rcollection)))
    elif operator == 'OR':
        return list(set(self._lcollection) | set(self._rcollection))
    elif operator == 'NOT':
        return  list(set(self._lcollection) - set(self._rcollection))
    else:
        return None
```

Term1 OR Term2

The above process is continued until fetching the documnet ids in a list and then instead of intersection, union operation is applied to achieve OR Operation.

Phrase Search And Proximity Search:

Phrase search and proximity search are differeniated by distance measure. To identify it as a phrase and belong to one document, the distance between the words should be unit distance. While the proximity search, the distance measure goes by user choice.

For instance

12(Term1,Term2) is interpreted as find all documents which has terms Term1 and Term2 with distance 12

"Term 1 and Term2" is interpreted as find all the documments which has terms Term1 and Term2 with unit distance

```
def _phrase_search(self,phrase,distance):

    fn = map(str,map(lambda x:x.strip('\"\''),phrase.split(' ')))

    _split = filter(re.compile(r'\w+').search,fn)

    _split = self._preprocess_search(_split)
    _records = self._service._fetch_records(_split,'WORD')

    if len(_records) == 2:
        return self._is_doc_with_phrase(_records[0],_records[1],distance)
    return None

def _is_doc_with_phrase(self,lrecord,rrecord,distance):

   _ldocs = lrecord['INFO']
   _rdocs = rrecord['INFO']

   _rsult_docs =  self._common_documents(_ldocs,_rdocs)
   _rsult_phrase = []

   for _docIdx in _rsult_docs:
       _rsult_phrase.append(self._neighbor_terms(_ldocs[_docIdx],_rdocs[_docIdx],_docIdx,distance))
   _phrase_rslt = map(lambda x: x[1],filter(lambda _match: _match[0] == True,_rsult_phrase))

   # returns documents
   return _phrase_rslt

def _neighbor_terms(self,llist,rlist,docIdx,distance):

    _combinations = product(llist,rlist)

    _result = filter(lambda term: abs(term[0] - term[1]) <= distance, _combinations)

    return (True if len(_result) > 0 else False,docIdx)
```

RANK ING

Ranking is implemented by a functionality called Filter in graphlab

To illustrate Further, I have implemented a function which takes a list of words,and the function returns the documents corresponding to each term. For instance, look at the snapshot below

```
def _execute_query(self,query):

    query = map(str,query)

    records = self._domain._fetch_records(query,'WORD')
    docs  = set()

    records_size = len(records)
    [docs.update(records[i]['INFO'].keys()) for i in range(records_size)]

    N = len(docs)
    _query_document = []
    for edoc in docs:
        score = 0.0
        for i in range(records_size):
            if records[i]['INFO'].get(edoc,False):
                tf = len(records[i]['INFO'].get(edoc))
                df = len(records[i]['INFO'].keys())
                score = score + ((1 + math.log10(tf))*(math.log10(float(N)/float(1+df))))
        _query_document.append((edoc,score))

    return sorted(_query_document,key=lambda x:x[1],reverse=True)[:1000]
```

And then apply the Term Frequency and Document Frequency methodology on the records fetched from the above.

For instance,

Term1 = {'d1':[1,2,3,4],'d2':[4,5,6],'d3':[2,7] }

To calculate Term Frequency for a document `d1`

tf(term1, d1) <--------len([1,2,3,4]) <-------- [1,2,3,4] <-----term1['d1']

To calculate the Document Frequncey for a term

['d1','d2','d3'] <------- term1.keys()

After calculating the tf,df, Plug those values into TDIDF Formula and get the rank for each document and those values are saved into file.