Search Engine - Project Part 1 Source Code

Christopher Ngan - 20423692

Contents

Chapter 1

Introduction

Report is structured to match the directory structure of the code. Base directory represents the home directory that all scripts are ran from.

```
run_pipeline.py
run_offline.py
process_topics.py
evaluate_results.py
    util/...
    online/..
    pipeline/..
    evaluate/..
```

To build the index/run the pipeline, run:

```
python run_pipeline.py --index <path to save index to/load data from> --file <path to LAtimes dataset>
```

To run the search engine online, run:

```
python run_online.py --index <path to save index to/load data from> --file <path to LAtimes dataset>
```

Chapter 2

Base Directory

run_pipeline.py

```
from pipeline.TRECReader import TRECReader
from util.Tokenizer import Tokenizer
from util.Index import Index

import util

from optparse import OptionParser




def build_index(filename, index_file = None):
    index = Index()
    if index_file == '' or \
        (index_file != '' and not index.can_load(index_file)):
        print 'Recomputing the index'
        tr = TRECReader(filename)
        t = Tokenizer()
        doc_count = 0

        for doc in tr.stream_docs():
            doc_id = doc['doc_id']
            doc_content = doc['doc_content']
            print 'Processing document', doc_id
            # dictionary to count term occurrences in the document
            term_counter = {}
            for token in t.tokenize_str(doc_content):
                if not term_counter.has_key(token):
                    term_counter[token] = 0
                term_counter[token] += 1
            for k in term_counter.keys():
                index.put(k, doc_id, term_counter[k])
            doc_count += 1
        print doc_count, 'documents processed'
        if index_file != '' and not index.can_load(index_file):
            print 'saving the index to', index_file
            index.save(index_file)
    else:
        print 'Loading the index from', index_file
        index = index.load(index_file)
    return index
if __name__ == '__main__':

    option, args = util.get_options()

    index_file = option.index
    index = build_index(
            option.filename,
            index_file = index_file,
            )

    import pdb; pdb.set_trace()
```

run_online.py

```
from online.Ranker import Ranker
from util.Tokenizer import Tokenizer
from run_pipeline import build_index

from util import get_options


if __name__ == "__main__":
    options,args = get_options()

    index = build_index(options.filename, index_file = options.index)
    tokenizer = Tokenizer()

    ranker = Ranker(index, tokenizer)
    print 'Search Engine is online!'
    while (True):
        query = raw_input('===Please enter query===\n')
        print ranker.run_query(query, max_items=1000)
```

process_topics.py

```
from online.Ranker import Ranker
from util.Tokenizer import Tokenizer
from run_pipeline import build_index
from util import get_options

import sys

import pandas as pd


import time

def get_numbers_from_str(string):
    """
    Get any numbers from a string
    """
    res = []
    for c in string.strip():
        if c.isdigit():
            res.append(c)
    return ''.join(res)


def get_topics(topic_file):
    """
    Programatically extract topics
    """
    topics = []
    lines =  open(topic_file, 'r').readlines()
    topic_id = ''
    for line in lines:
        # hard code
        if line.find('<num>') == 0:
            topic_id = get_numbers_from_str(line)
            print topic_id
        if line.find('<title>') == 0:
            query = line.replace('<title>', '').strip()
            print query
            topics.append(dict(
                topicID = topic_id,
                query = query,
                ))
    return topics


if __name__ == '__main__':
    options,args = get_options()
    # get the file of the submissions from process_topics.py
    output_file = sys.argv[1]

    index = build_index(options.filename, index_file = options.index)
    tokenizer = Tokenizer()

    ranker = Ranker(index, tokenizer)
    topics = get_topics('data/topics.401-450.txt')
    all_results = []
    # topics to ignore
    ignore_topic_ids = [416, 423, 437, 444, 447]
    for topic in topics:
        if not int(topic['topicID']) in ignore_topic_ids:
            start_time = time.time()
            results = ranker.run_query(topic['query'], max_items=1000)
            run_time = time.time() - start_time
            for res in results:
                all_results.append(dict(
                    topicID = topic['topicID'],
                    q0 = 0,
                    docno = res['doc_id'],
                    rank = res['rank'],
                    score = res['score'],
                    runTag = 'cjngan_run0',
                    run_time = run_time,
                    query = topic['query'],
                    ))
    df = pd.DataFrame(all_results)
    df.to_csv(output_file, index=False)
    submission_file = output_file.replace('.csv', '-submission.csv')
    submission_cols = ['topicID', 'q0', 'docno', 'rank', 'score', 'runTag']
    df[submission_cols].to_csv(
            submission_file, index=False, sep=' ', header=None)
```

evaluate_results.py

```
import pandas as pd
from evaluate.metrics import compute_precision_at_k,\
                    compute_dcg

import numpy as np
import sys

if __name__ == '__main__':
    # use of pandas to load in csv files
    df_truth = pd.read_csv(
            'data/LA-only.trec8-401.450.minus416-423-437-444-447.txt',
            sep=' ',
            header=None,
            )
    dat_file = sys.argv[1]
    # the data files (note that its not the submission file)
    df_exp = pd.read_csv(dat_file)
    # since columns are not named
    df_truth.columns = ['topicID', 'q', 'docno', 'relevance']
    df_truth.drop(['q'], axis=1,inplace=True)
    df_j =pd.merge(df_exp, df_truth, on=['topicID', 'docno'], how='left').fillna(0)

    k = 10

    metrics = []

    for topicID, df_g in df_j.groupby('topicID'):
        if topicID not in [416, 423, 437, 444, 447]:

            relevances = df_g.sort('rank', ascending=True)['relevance']
            ranks = np.array(xrange(1,k+1))
            ### all results matching the query
            true_relevancy = df_truth[
                    df_truth['topicID'] == topicID
                    ].sort('relevance', ascending=False)['relevance']


            precision = compute_precision_at_k(relevances, k)

            ideal_dcg = compute_dcg(true_relevancy, ranks,k)
            m_dcg = compute_dcg(relevances, ranks,k)
            ndcg = m_dcg/ideal_dcg
            metrics.append(dict(
                    topicID = topicID,
                    k = k,
                    precision = precision,
                    ndcg = ndcg,
                    queryTime = np.mean(df_g['run_time']),
                    ))
    df = pd.DataFrame(metrics)
    # can read this for results
    df.to_csv("data/summary.csv")
    print df.describe()
```

Chapter 3

pipeline

pipeline/init.py

```
"""
Offline processing that can be pipelined
"""
```

pipeline/TRECReader.py

```
"""
Reads the TREC latimes dataset
"""
import gzip
import re

from xml.dom import pulldom
import xml

from util.Tokenizer import Tokenizer

class TRECReader:
    def __init__(self,
            trec_file,
            doc_id_tag='DOCNO',
            doc_end_tag='</DOC>',
            relevant_tags=[
                'HEADLINE',
                'TEXT',
                'GRAPHIC',
                'SUBJECT', # as per given src code
                ]
            ):
        if trec_file.endswith('.gz'):
            # Reads gzip files
            self.file_stream = gzip.open(trec_file, 'r')
        else:
            # general text file
            self.file_stream = open(trec_file, 'r')
        self.infile_name = trec_file
        self.doc_id_tag = doc_id_tag
        self.doc_end_tag = doc_end_tag
        self.relevant_tags = relevant_tags + [doc_id_tag]
```

```
def next_doc(self):
    """
    Load one doc at a time, no need to load whole thing in mem
    Reads a single doc into memory so that it
    can be processed by an xml tree
    """
    doc_content = ''
    for line in self.file_stream:
        line = line.strip() + ' '

        doc_content += line
        if line.find(self.doc_end_tag) == 0:
            yield doc_content
            doc_content = ''




def stream_docs(self):
    """
    Generator that streams a dictionary
    with:
      @doc_id      - identifier of the document
      @doc_content - raw text of the document
    """
    ## Pull out the doc
    #for event, node in self.pull_xml_parser():
    for doc_content in  self.next_doc():
        tag_stack = []
        # parse the xml into a streamer
        str_buffer = []
        doc_id = ''
        for event, node in pulldom.parseString(doc_content):
            if node.nodeName in self.relevant_tags:
                if event == pulldom.START_ELEMENT:
                    # next set of text is within
                    # some relevant tag
                    tag_stack.append(node.nodeName)
                if event == pulldom.END_ELEMENT:
                    tag_stack.pop()
            elif event == pulldom.CHARACTERS and len(tag_stack) > 0:
                # is within a relevant stack an
                # interested in reading the document

                peeked_val = tag_stack[len(tag_stack)-1]
                if peeked_val == self.doc_id_tag:
                    # This is the docID
                    doc_id = node.nodeValue.strip()
                else:
                    str_buffer.append(node.nodeValue)

        # so we don't have to keep creating new strings
        doc_content = ''.join(str_buffer)
        yield dict(
                doc_id=doc_id,
                doc_content=doc_content,
                )
```

Chapter 4

util

util/init.py

```
"""
Util classes used in both
"""
from optparse import OptionParser

import heapq

def get_options():
    parser = OptionParser()
    parser.add_option("-f", "--file", dest="filename",
            help="Data file to load data from",
            )
    parser.add_option("-i", "--index", dest="index",
            help="File to load/save index from",
            default='',
            )
    options, args = parser.parse_args()
    if not options.filename:   # if filename is not given
        parser.error('Filename not given')

    return (options, args)




class MinHeap:
    def __init__(self, maxsize=-1):
        self.heap = []
        self.maxsize = maxsize

    def qsize(self):
        return len(self.heap)

    def put(self, score, data):
        heapq.heappush(self.heap, (score,data))
        if self.maxsize != -1 and self.qsize() > self.maxsize:
            heapq.heappop(self.heap)
    def get_min(self):
        return heapq.heappop(self.heap)
```

util/Index.py

```
import cPickle as pickle
import os
import sys

import json

from util.Lexicon import Lexicon

class Index:
    def __init__(self):
        self.index = {}
        self.term_lexicon = Lexicon()
        self.doc_lexicon = Lexicon()
        # tracks the length of each documnet
        self.doc_length_count = {}
        # tracks the number of occurences per term
        self.term_length_count = {}
        # number of words in the collection
        self.coll_term_count = 0.
        # number of docs in the collection
        self.coll_doc_count = 0.
        # aerage document length
        self.avg_doc_length = 0.


    def get_doc_key(self, doc):
        """
        Returns the key that is mapped from the doc id
        """
        dv = self.doc_lexicon.map_k_to_v(doc)
        if not self.doc_length_count.has_key(dv):
            # a new document
            self.doc_length_count[dv] = 0
            self.coll_doc_count += 1.
        return dv

    def get_term_key(self, term):
        """
        Returns the key that is mapped from the term string
        """
        tk = self.term_lexicon.map_k_to_v(term)
        if not self.term_length_count.has_key(tk):
            # a new term
            self.term_length_count[tk] = 0
        if not self.index.has_key(tk):
            # a new term
            self.index[tk] = []
        return tk


    def put(self,term, doc, count):
        """
        Adds a <term str, doc_id, term frequency>
        to the index while updating
        counters and lexicon
        """
        tk = self.get_term_key(term)
        dv = self.get_doc_key(doc)
        # assumes that terms per doc or only
        # processed once and not repeated.
        # update all counts accordingly.
        self.doc_length_count[dv] += count
        self.coll_term_count += count
        self.term_length_count[tk] += count

        # Index is a hash with term as key
        # and vals of list of integers
        # where first val is doc, second is term freq
        self.index[tk].append(dv)
        self.index[tk].append(count)
        # update the average
        self.avg_doc_length = self.coll_term_count / self.coll_doc_count


    def _get_term_doc_list(self, term):
        """
        Private function that returns
        the list of doc, counts
        given a term
        """
        tk = self.term_lexicon.map_k_to_v(term)
        return self.index[tk]

    def save(self, file_name):
        """
        Save the index to a file
        """
        pickle.dump(self, open(file_name, 'wb'))

    def load(self, file_name):
        """
        Load index from a file
        """
        return pickle.load(open(file_name, 'rb'))

    def can_load(self, file_name):
        """
        Checks if the index pickle file
        can be loaded
        """
        return os.path.isfile(file_name)

    def next_doc_from_tokens(self, tokens):
        """
        Applies doc at a time query processing
        """

        # remove tokens that are not part of the vocab
        rel_tokens = set() # set so duplicates aren't added
        for t in tokens:
            if self.term_lexicon.has_key(t):
                rel_tokens.add(t)
            else:
                print t, ': term does not exist in vocab'
        rel_tokens= list(rel_tokens)
        pointers = [0]*len(rel_tokens)
        has_docs = True
        # the list of doc/counts per term
```

```
rel_term_list = map(
        self._get_term_doc_list,
        rel_tokens
        )
n = len(rel_tokens)
while(has_docs):
    # assume that the max int will never be reached
    next_doc_id = sys.maxint
    for i in xrange(n):
        term_list_pointer = pointers[i]
        if term_list_pointer != -1:
            d_id = rel_term_list[i][term_list_pointer]
            # assumes that document ids
            # are stored in incremental order
            next_doc_id = min(d_id, next_doc_id)
    if next_doc_id == sys.maxint:
        print 'Done query processing'
        # processed all docs, stop while loop
        has_docs = False
    else:
        # relevant docs still exist


        # create the hash that will be returned for
        # the ranker. Just dump allll the data
        doc_data = dict(
                doc_id = self.doc_lexicon.map_v_to_k(next_doc_id),
                # number of terms in collection
                coll_term_count = self.coll_term_count,
                # number of document in collection
                coll_doc_count = self.coll_doc_count,
                # assumes all term freq are init zero
                # matches to the query tokens
                term_hash= { term : dict(
                    count_of_docs_with_term=0,
                    term_frequency = 0,
                    ) \
                        for term in tokens
                    },
                # the number of terms in document
                doc_length = self.doc_length_count[next_doc_id],
                # average doc length
                avg_doc_length = self.avg_doc_length,
                )
        for i in xrange(n):
            term = rel_tokens[i]
            t_li_pointer = pointers[i]
            term_li = rel_term_list[i]
            # only iterate if the term list has any more documents
            # to process. Only process the items that have the
            # document id that is being processed
            if t_li_pointer != -1 and term_li[t_li_pointer] == next_doc_id:
                if len(term_li) <= (t_li_pointer + 2):
                    # this documet is the last one
                    # that has this term. Stop
                    # iterating after this
                    pointers[i] = -1
                else:
                    # the list is a pairing of
                    # doc IDs and the term count
```

```
                            # so jump 2
                            pointers[i] += 2

                        count = term_li[t_li_pointer + 1]
                        # update the doc raw data
                        doc_data['term_hash'][term]['term_frequency'] = float(count)
                        doc_data['term_hash'][term]['count_of_docs_with_term'] = \
                                len(term_li)/2.

                yield doc_data
```

util/Lexicon.py

```
class Lexicon:
    def __init__(self):
        self.mapper = {}
        self.inv_mapper = [] # all mapped values are keys

    def has_key(self, k):
        return self.mapper.has_key(k)

    def map_k_to_v(self,k):
        if not self.mapper.has_key(k):
            self.mapper[k] = len(self.inv_mapper)
            self.inv_mapper.append(k)
        return self.mapper[k]

    def map_v_to_k(self, v):
        if v > len(self.inv_mapper) - 1:
            raise Exception("Invalid mapped value")
        else:
            return self.inv_mapper[v]


if __name__ == '__main__':
    le = Lexicon()
    print le.map_k_to_v('dog')
    print le.map_k_to_v('cat')
    print le.map_v_to_k(0)
    print le.map_v_to_k(1)
```

util/Tokenizer.py

```
class Tokenizer:
    def __init__(self):
        pass

    def tokenize_str(self, string):
        str_buff = []
        for c in string:
            if c.isalnum():
                str_buff.append(c.lower())
            elif len(str_buff) > 0:
                yield ''.join(str_buff)
                str_buff = []
        if len(str_buff) >0:
            yield ''.join(str_buff)

if __name__ == '__main__':
    a = 'The cat. The cat is fat'
    t = Tokenizer()
    print list(t.tokenize_str(a))
```

Chapter 5

online

online/init.py

```
"""
Live search engine that can read results
"""
```

online/Ranker.py

```
from util.Tokenizer import Tokenizer
from util.Index import Index
from util import MinHeap
import math

from collections import Counter


class Ranker:
    """
    Applying ranking algorithm from query
    """
    def __init__(self, index, tokenizer):
        assert isinstance(index, Index)
        assert isinstance(tokenizer, Tokenizer)
        self.index = index
        self.tokenizer = tokenizer


    def run_query(self, query_str, max_items=-1):
        """
        -1  - infinite queue size
        Returns:
            [dict(
                doc_id = <abc123>,
                score = <123>,
                rank = <123>
            )]
        """
        tokens = list(self.tokenizer.tokenize_str(
            query_str
            ))
        min_q = MinHeap(maxsize=max_items)
        # NOTE: the queue ranks the CLOSEST item with a high
        # score. If the ranking alg requires that the HIGHLY
        # relevant items have high scores, just output the score.
        # If the ranking alg requires the that the HIGHLY relevant
        # items have LOW scores, multiply score by -1.
        for doc in self.index.next_doc_from_tokens(tokens):
            score = self.compute_score(doc, tokens)
            min_q.put(score, doc['doc_id'])
        rel_docs = []

        # empty out the priority queue
        while(min_q.qsize() > 0):
            score, doc_id = min_q.get_min()
            rank = min_q.qsize() + 1
            rel_docs.append(dict(
                doc_id = doc_id,
                score = score,
                rank = rank,
                ))
        # does it in place
        rel_docs.reverse()
        return rel_docs



    def compute_score(self, doc, tokens, k1=1.2, b=0.75, k2=7):
        """
        Compute the document score with BM25
        different scoring functions
        """
        score = 0
        counter = Counter(tokens)
        terms = set(tokens)
        for token in terms:
            fi = doc['term_hash'][token]['term_frequency']
            qfi = counter[token] # number of times term shows up in query
            ni = doc['term_hash'][token]['count_of_docs_with_term']
            dl = doc['doc_length']
            N = doc['coll_doc_count']
            avg_dl = doc['avg_doc_length']
            K = k1*(1-b + b*dl/avg_dl)
            t1 = (k1 + 1)*fi/(K + fi)
            t2 = (k2 + 1)*qfi/(k2 + qfi)
            t3 = math.log((N-ni+0.5)/(ni+0.5))
            score += (t1*t2*t3)
        return score
```

Chapter 6

evaluate

evaluate/init.py

evaluate/metrics.py

```
import numpy as np

def compute_precision_at_k(relevances, k):
    """
    Compute precision at rank k
    """
    trimmed_rel = relevances[:k]
    tp = sum(trimmed_rel == 1)
    return tp / float(k)

def compute_dcg(relevance, ranks,k):
    """
    Compute dcg@k
    """
    trimmed_rel = relevance[:k]
    trimmed_ranks = ranks[:k]
    return np.sum(np.divide(
        trimmed_rel,
        np.log2(trimmed_ranks + 1),
         ))
```