Roll Your Own Mini Search Engine

March 25, 2018

content

Chapter: 1 Introduction....4

1.1 Background....4
1.2 Problem Description....4

Chapter: 2 Algorithm Specification....6

2.1 Overall Structure....6
2.2 Data Collection and web crawler....6
2.2.1 overview....6
2.2.2 Pseudo code....7
2.3 Pretreatment and model training....7
2.3.1 Information extraction....7
2.3.2 Ranking System and Machine learning....8
2.4 Search Process....13
2.4.1 overview....13
2.4.2 Pseudo Code....13
2.5 Web Server....13
2.5.1 Frontend....14
2.5.2 Backend....14

Chapter: 3 Testing Results....15

3.1 Test environment....15
3.2 Running....15
3.2.1 Test Cases....16
3.2.2 Results....16

Chapter: 4 Analysis and comments....17

4.1 Time complexity Analysis....17
4.2 Space complexity Analysis....17
4.3 Comments....17

Chapter: 5 Appendix A: Source Code....17

Chapter: 6 Declaration....29

Chapter: 1 Introduction

1.1 Background

With the growth of the network information, searching valuable information from numerous documents has become an important topic for data mining. Search engine is an interface between the users and the information. The user just input the words they are interested in and get the results easily. A good search engine can accomplish the process quickly and make the results match to the input string.
Currently, there are tremendous number of diverse sorts of search engines in the world. Google, Bing are those general search engines which can search any information open on internet. Everything is a search engine installed on PC used for searching for files on our personal computers. SCI (Science Citation Index) is another type of search engine as it provides a limit range of information for special users.
Search engines come to diverse platforms. Nowadays, we can see the search engines not only work on computers or smart phones but also on wearable device and device in smart home system, like Amazon Echo. Search engines are making information available for us anywhere and anytime.
A search engine is general subject as it is related to database, inverted files and natural language processing. Designing an effective in both performance and user interaction is a difficult but significant job. New technology is always applied for designing a better search engine. For instance, Map/Reduce and Hadoop are invented for building a scalable and distributed search engine file system. Natural Language Processing (NLP) and Recurrent Neuron Networks are introduced in machine's understanding the input of human.
Different sorts of search engines and new technologies are enable our world searched.

1.2 Problem Description

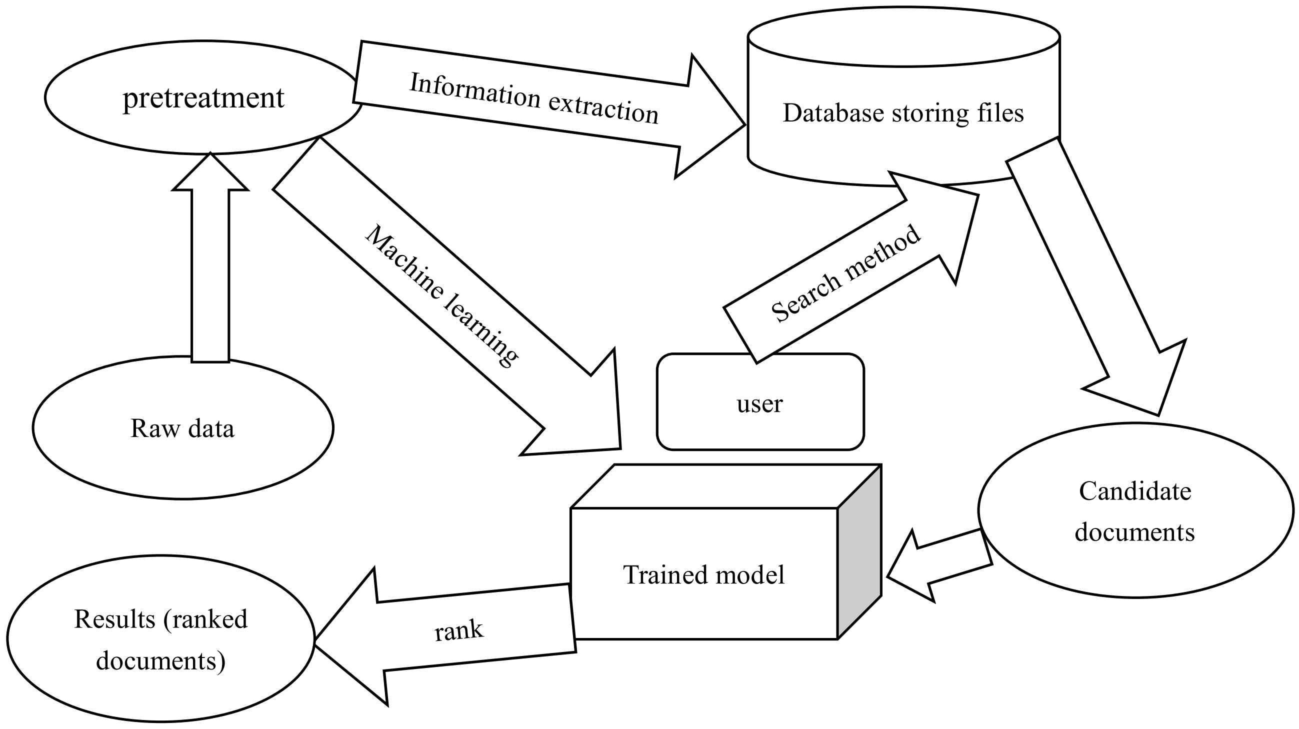
In this project, our job is to create our own mini search engine which can handle inquiries over "The Complete Works of William Shakespeare". The user inputs the content they are interested in and gets the corresponding results. Our tasks include running a word count over the Shakespeare set and try to identify the stop words, creating inverted index over the Shakespeare set with word stemming and writing a query program on top of your inverted file index.
The basic job for us is building a mini search engine. To complete it, we need an inverted file index with the operations related to it as well as a ranking method. We need to complete a application that every time we input a valid sentence, it will return something it found in the "The Complete Works of William Shakespeare".
Additionally, in order to implement a better system, we build another two components. Firstly, we apply machine learning methods to develop our ranking methods which can increase the scores of our ranking system. Secondly, we use web server and frontend web page to implement a web demo to interact with our users.
Finally, after finishing the project, we will obtain a search engine which can handle inquiries over

“The Complete Works of William Shakespeare” and display the results in users’ browser.

Chapter: 2 Algorithm Specification

2.1 Overall Structure

Roughly speaking, our solution can be divided into three parts: pretreatment, file ranking and UI, as the following figure illustrates.



2.2 Data Collection and web crawler

2.2.1 overview

The “The Complete Works of William Shakespeare” is stored in the website *shakespeare.mit.edu.* We need to collect all the data and parse them for the next steps.
The process of downloading all the data in a web page and then parsing them is frequently called web crawler. It consists several parts.

1. Find all the pages we need.
2. Connect to web server and download data
3. Parse HTML data

2.2.2 Pseudo code

Script: fetch

```
Function fetch(url)
  urlList = GetAllUrl(url)
  For (url in urlList)
     html = Download(url)
     information.append(ParseHtml(html))
     Handle the network errors.
  EndFor
```

2.3 Pretreatment and model training

2.3.1 Information extraction

To achieve our goal, we should pretreat the documents before building the search engine. In the pretreatment model we convert the documents into several dictionaries. The keys in the dictionary are the vocabulary occurring in the documents and the values are the corresponding times. We accomplish it by the library nltk (natural language processing toolkit). After that we can generate the list for term frequency–inverse document frequency and inverted file. In information retrieval, tf–idf or TFIDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.[1]

$$term\ frequency = \frac{f_{t,d}}{\sum_{t'\in d} f_{t',d}}$$

$t\ is\ the\ specific\ term\ in\ the\ document\ and\ t'\ is\ the\ set.$

$$inverse\ document\ frequency = \log\frac{N}{n_t}$$

$N\ is\ the\ total\ number\ of\ the\ document\ and\ n_t is\ the\ specific\ documents$
$which\ content\ the\ term.$

Hence, we can use value = tf * idf to represent the importance of a word in a document.

To acquire pretreatment result more quickly, we store the result into a database named "PretreatmentInfo.db".
Here are some tables in the database:
Table name: InvertedFile

Column name	Data type	Usage
WORD	TEXT	Terms
FileNumber	TEXT	Corresponding files

[1] From Wikipedia https://en.wikipedia.org/wiki/Tf%E2%80%93idf

Table name: tf_idf

Column name	Data type	Usage
FileID	INT	File index
WORD	TEXT	Terms
VALUE	REAL	Tf-idf values

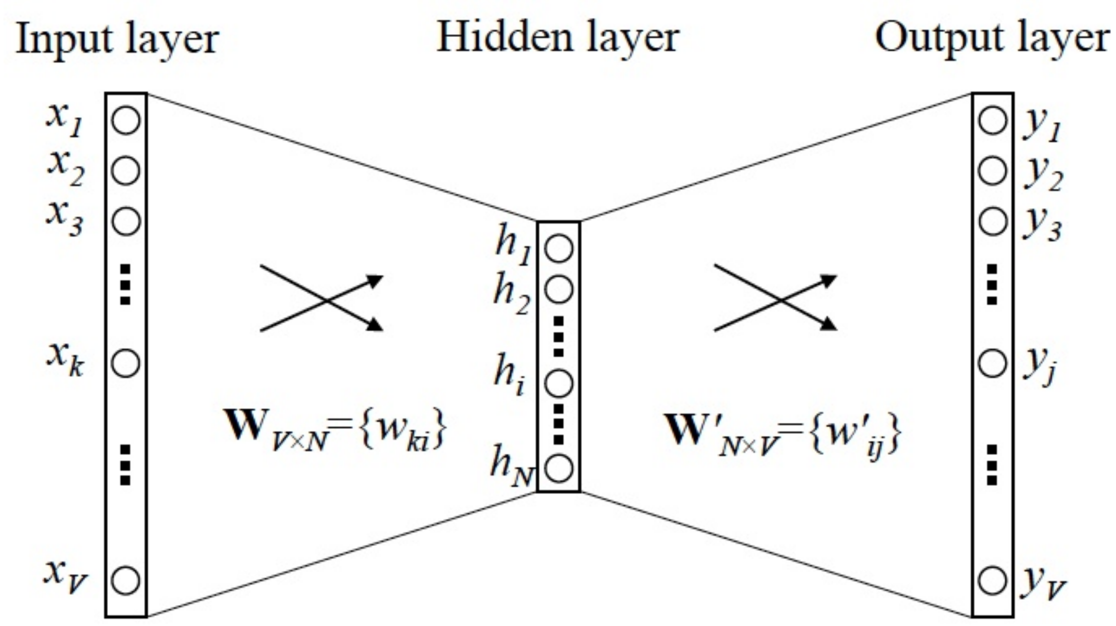
Table name: urlTitleIndex

Column name	Data type	Usage
Title	TEXT	Document's title
URL	TEXT	Corresponding URL
FileID	INT	File index

2.3.2 Ranking System and Machine learning

Word2Vec

Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. In our search engine, we implemented a deep neural network to train the data from the collection of Shakespeare by genism module. In this neural network, two hidden layers are designed to represent the model. One layer consists N hidden neurons (N is the number of words in our collection) which can hold one-hot vectors which is reshaped from our words in the collection. Another hidden layer consists 150 neurons which represents 150 features in our vector space. Then the model will do regression on output from hidden layers by softmax function.
In this model, we set min_count 1 which means the lowest count a word which will be considered effective.
Training is a supervised learning process. In the process, we set windows equal to 5 which means the maximum distance between the current and predicted word within a sentence will be lower than 10 and Number of iterations (epochs) over the corpus equal to 10.



Principle Component Analysis

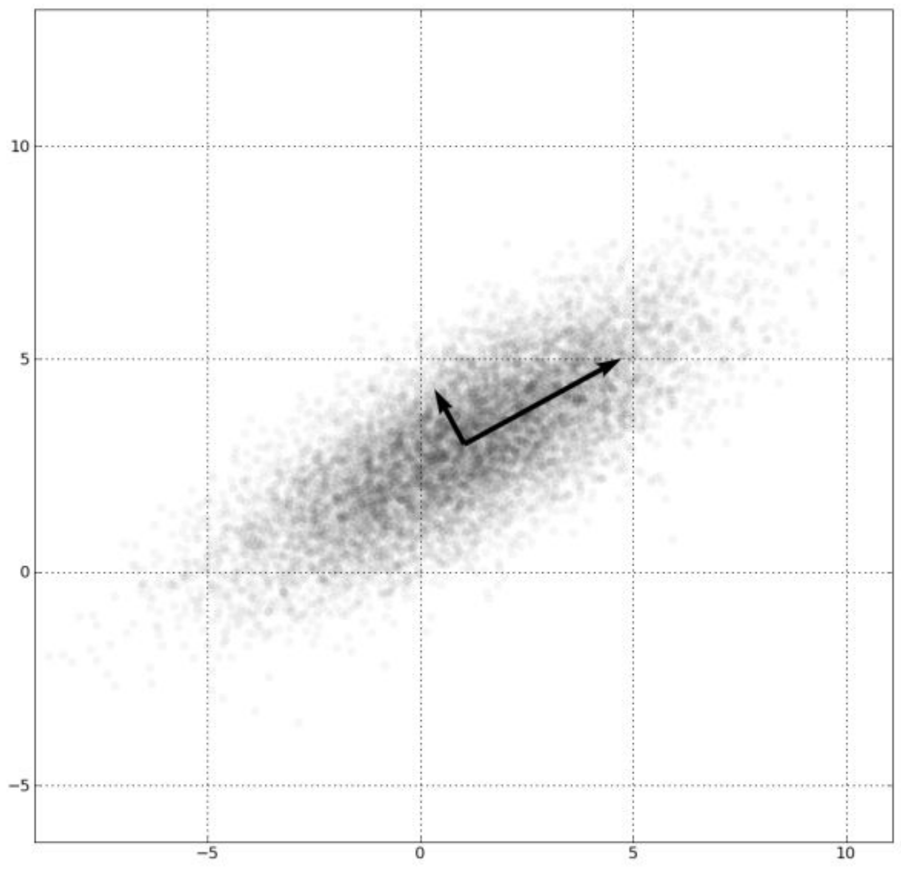
In the process of training our Word2Vec mode. We set our hidden neurons be 150, in order not to be overfitting or underfitting. However, after we have trained our model, it is difficult for us to simply use the 150-dimension vectors to fit our sentence. As a result, we need to reduce the dimension of our vectors and luckily, we have Principle Component Analysis (PCA) to reduce the dimension of vectors.

Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. After that, we can calculate the weight of each dimension, and then those unimportant components while we can maintain most features of our vector space.

The transformation process is defined in such a way that the first principal component has the largest possible variance (that is, accounts for as much of the variability in the data as possible), and each succeeding component in turn has the highest variance possible under the constraint that it is orthogonal to the preceding components.

In this model, we determined that we need to keep 80% of variance of our vactors, after we do sum on first N components, we learned that when we keep first 90 components we will have 80% variance.

To illustrate, we designed an example of two-dimension vector space and do PCA on it.



we can find that when we rotated our axis we will find most of variance of our points are in one dimension which is the most important component. Then we just need to keep the important component and throw the other.

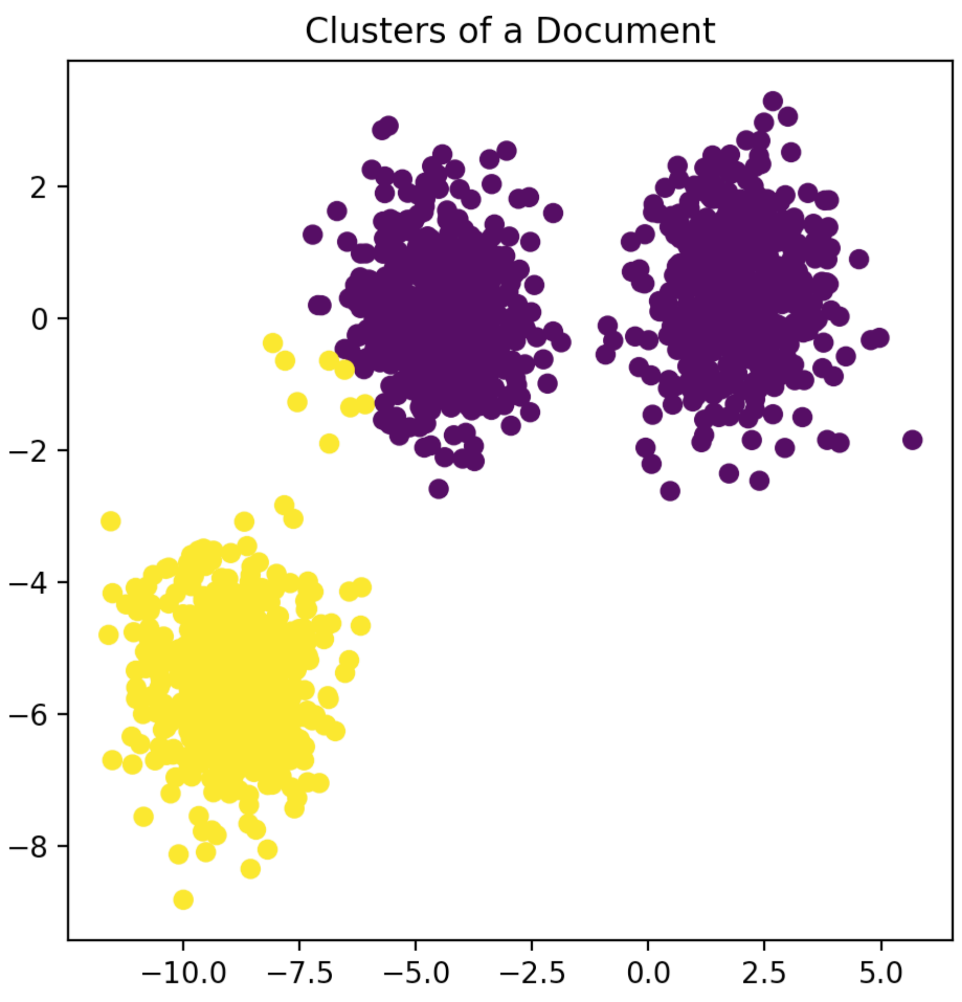
In our project, we implemented the PCA model by sklearn moduel.

Algorithm 1 PCA

```
Function PCA (vectors)
   principleComponents = OrthogonalTransform(vectors)
   variance = Variance(vectors,principleComponents)
   k_sum = [sum(variance[0:i]) for i = 1 to len(variance)]
   k = find(k_sum>0.8)
   // k = 90 we get in this project
   keep first k dimensions of principleComponents
   => reducedPrincipleComponents
   return reducedPrincipleComponents
```

Clustering

After we finished the process of Word2Vec and PCA, we will get a list of 90-dimension vectors which represents our words in the collection. Then, we need to analysis the projection of a document in our word vector space. It is easy to just use the average vector of all vectors of words in a document as the vector of a document. But it is a rough model and can bring many problems. An obvious example which can illustrate the limitation of the model is that a document is mainly about the comparison of two objects. It is potential that our vectors will gather around two points each of which represent one object. And then the vector representing our document is possibly on the mid point of the two points, in fact it is neither two points!
To solve the problem, we design a clustering process implemented by K-Means algorithm to determine the feature vectors of a document. All words of a document will be in the word vector space, and there are possibly several clustering points in the vector space, which represents the meaning of different parts of a document. To illustrate, we have a model of two dimension model.



The document has two clustering points, and the purple points and yellow points represent two sort of points, which belongs to the two clustering points.
When we extend our model to a higher dimension model like our word vector space, we need to

set more clustering points, and in this case, we set 8.
After clustering, we will have 8 clustering points of each document which represents 8 key points of a document,

Algorithm 1 K-Means

```
function KMeans($pointsList, iterTimes$)                ▷ Where pointsList has N points
    Initialize K points: $[c_1 \dots c_k]$
    Let $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ be new arrays
    for $time = 1$ to $iterTimes$ do
        for $i = 1$ to $N$ do
            $p_i = pointsList[i]$
            $s = NearestClusterPoint(p_i)$
            $set[s].append(p)$
        end for
        for $j = 1$ to $K$ do
            $c_j = AveragePoint(set[j])$
        end for
    end for
end function
```

If a word is related to a document, in fact meaning that the word is related to a part of the document. As a result, we just selected the most similar clustering point to determine the relation between a word and a document.

$$\text{rank}(\text{document}, \text{word}) = \max\,(\text{similarity}(\text{clusters}, \text{word}))$$

similarity
When we have finished selecting the vectors which represent the features of a document, we need to determine the similarity or relationship between the two vectors. In this project, we use Pearson product-moment correlation coefficient to represent the relation between two vectors, which is a significant method to evaluate the similarity of two vectors in Machine Learning, although it also has many applications. It avoids the limitations of the method of Euclid distance

$$\text{Pearson} = \frac{Cov(X, Y)}{\sigma_X * \sigma_Y}$$

Sentence Model

In the previous sections, we successfully finished a series of models which depicts the relation of a word and a document. Then we need to make sure the relation of a sentence and a document.
A sentence consists many words, after excluding those stopping words like ‘go’, ‘to’, we will have a list of words and their relations with a document.
It is easy to calculate the average rank of each words. However, different word count in differently in a sentence. Here, we use idf-tf to determine the importance of each word and calculate the rank of the document by the following equation.

$$\text{rank}(\text{document}) = W * wordsRank^T$$

$$\text{W} = (\text{tf}_{\text{idf}} \text{ for each word})$$

$$\text{wordsRank} = (\text{Rank}(\text{document}, \text{word})\ for\ word\ in\ wordsList)$$

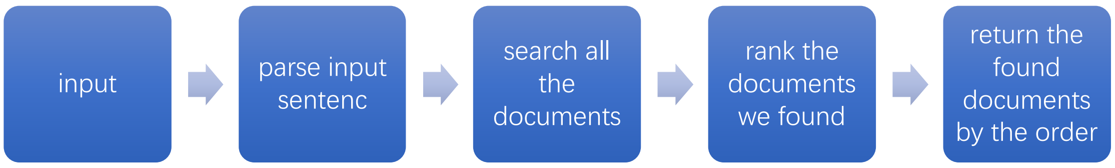
Then we have a list of rank for documents we have got and a simple sort will solve the left problems.

$$\text{sort}(\text{rank}(\text{documents}))$$

2.4 Search Process

2.4.1 overview

In this model, the user input a string and acquire a tuple. Every element in the tuple contain file ID, title and URL of the documents. We first convert string into a list to get the words. Then we connect to the database to get the documents.
Our search file is defined in './main.py' and './Prehandle/SearchDemo
The life circle of we search a sentence can be described followed.


input
parse input sentenc
search all the documents
rank the documents we found
return the found documents by the order

2.4.2 Pseudo Code

Algorithm 3: Search

```
Function Search(input)
    sentenceVectors, words = ParseSentence(input)
    documents = FindDocumentsInInvertedIndex(words)
    rank(documents,sentenceVectors)
    return documents
```

2.5 Web Server

We design a web application to interact with our users.

2.5.1 Frontend

Our frontend app or just html page has a search page allowing users to search for a sentence and a displaying page to show all the documents we found.

2.5.2 Backend

We use flask to implement our web app. When doing a search, web server will pass the sentence from frontend to the search process.

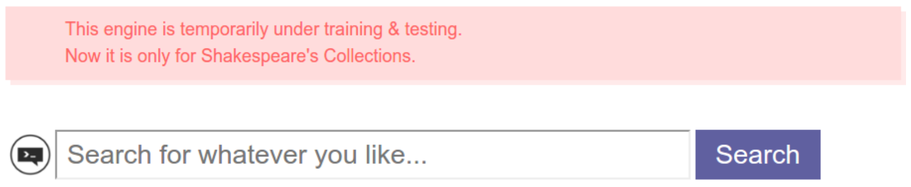
Chapter: 3 Testing Results

3.1 Test environment

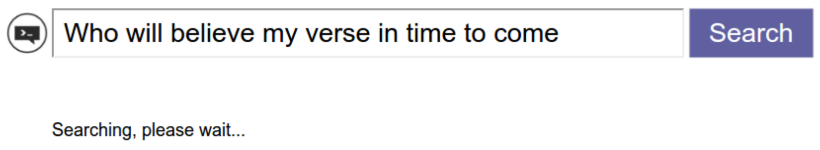
CPU: Intel® Core™ i7-4510U CPU @ 2.00GHz ×4
RAM: 3.80G (with about 2.20G available)
OS: Ubuntu Linux 16.04 (64 bit)
Python: 3.5.2

3.2 Running

The website:

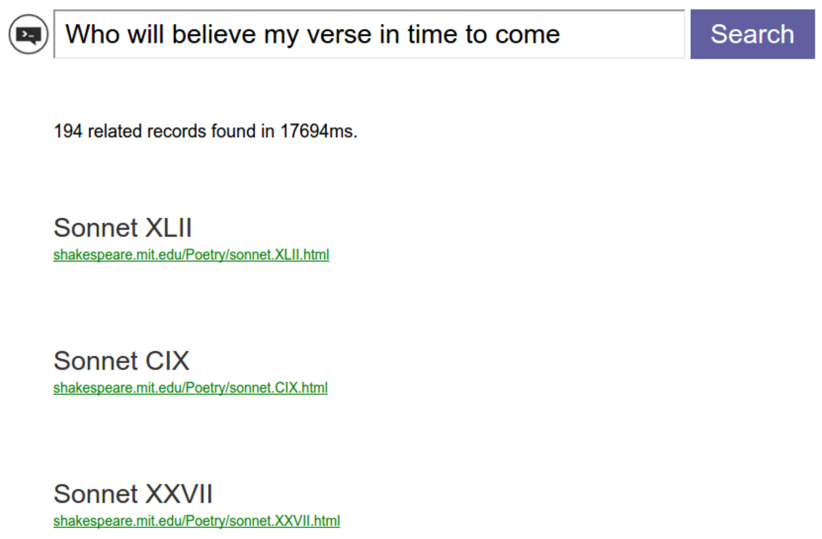

This engine is temporarily under training & testing.
Now it is only for Shakespeare's Collections.

During Searching:


Who will believe my verse in time to come

Searching, please wait...

Result Display:


Who will believe my verse in time to come

194 related records found in 17694ms.

Sonnet XLII
shakespeare.mit.edu/Poetry/sonnet.XLII.html

Sonnet CIX
shakespeare.mit.edu/Poetry/sonnet.CIX.html

Sonnet XXVII
shakespeare.mit.edu/Poetry/sonnet.XXVII.html

The number of results and time cost are displayed here.

3.2.1 Test Cases

A) Name of characters (for example: “Macbeth” from Macbeth)
B) Pronoun of characters (for example: “Messenger”)
C) Well known sentence (for example: “To be or not to be” from Hamlet)
D) Lines from text (for example: “Who will believe my verse in time to come” from Sonnet XVII)
E) Frequently used phrases (for example: “My thanks”)
F) Random string (for example: “dquglijfqeofq”)

3.2.2 Results

Type	# of cases	Avg Time(ms)	Avg Position	Avg # of Records
A	10	1395.0	Only One	1
B	10	1528.0	N/A	18.8
C	10	16500.7	Medium	190.6
D	10	17171.6	Front (Depends)	159.6
E	10	4097.2	N/A	86.4
F	3	N/A	N/A	N/A

(We shall record the Avg Position only when the answer is supposed to be unique)

For Cases of type A), since names are often unique, the engine works perfectly well.
For Cases of type B), the engine works also better than expected. This might be caused by the difference between the texts: the pronouns depends on the scene described in the play.
For Cases of type C), the result is not so satisfying. We have nothing to do on these “well known” lines, so they are obviously not so “well known” to the engine.
For Cases of type D), it depends. For plays, it behaves normally, because plays are often filled with casual conversations which are always similar. But for poems, on the other hand, as poems are always more typical, the engine can always put the correct record in the very front.
For Cases of type E), the number of records fetched is much less than expected. This might also because of the style difference between the works.
For Cases of type F), the server always returns "word not in vocabulary" Error.

Chapter: 4Analysis and comments

4.1 Time complexity Analysis

Suppose the data scale of the words in the input string is M and the candidate documents is N. Since we need to analyze the time complexity of pretreatment (we do it just once). All we need to do is to analyze the time complexity of the search and rank process. In the first stage, we select the candidate documents from the database and since it is a linear algorithm(obviously), the time complexity is kN(k is a constant number associated with the database). In the second stage, we use the machine learning model and rank the documents. In the machine learning process, we compare every word with eight cluster and conclude our results. Therefore, the time complexity is $\mathrm{O}(pNM)$(p is a constant number associated to the learning model). Since we call the inner function in Python, the time complexity of the sorting is $\mathrm{O}(NlogN)$. Hence, the total time complexity in the searching process is $O(NlogN + pNM + kN)$.

4.2Space complexity Analysis

It is hard to analyze the space complexity of the trained model and the database since we don't know the ground-level details of them. And during the search, the space complexity is $O(k_1N + k_2M)k_1\ and\ k_2\ are\ two\ constant\ number$. We need space to store the words and the candidate documents and for each term the space is linear.

4.3Comments

Compare with the traditional methods of text search merely, in this project we use a model based combined with traditional methods and machine learning. With the help of tf-idf method, we needn't handle stop words since their tf-idf values are really small and it makes our model more robust. However, there are still some defects in our model. First, with the growth of the input string, it may take some time to conclude answer. Second, although we use machine learning model, our model is still based on statistics, not on logic.

Chapter: 5 Appendix A: Source Code

```
from PySrc.RankMethodBase import RankMethodBase
from gensim import models
from sklearn.externals import joblib
```

```
import numpy as np
import pandas as pd
import sqlite3
import scipy.stats as stats


'''
class Rank1
   inherit: RankMethodBase
   description: our first rank method, maybe the last one before
peer review
   method: rank: rank the documents inputed by the input string.
'''


class Rank1(RankMethodBase):
   def rank(self,documents,sentence):

      # load the machine learning model we have pretrained.
      # pca is the principle components analysis model
      # model is the Word2Vec model
      pca = joblib.load('./PySrc/model/train_model90.m')
      model = models.Word2Vec.load('./PySrc/model/model')


      # parse our sentence, it will return sentence vectors and
words list.
      sentence_vecs,words = parse_sentence(sentence,pca,model)
      rank_list = []

      # traversal all documents
      for document in documents:

         # get tf-idf list
         idf_vec = get_tf_idf(document,words)
         rank =
calculate_document_similarity(document.clusters,sentence_vecs,model,i
df_vec)
         rank_list.append((document,rank))

      # sort the list
      rank_list = sorted(rank_list,key=lambda x:x[1])
      return rank_list
```

```
 # parse our sentence, it will return sentence vectors and words
list.
def parse_sentence(sentence,pca,model):
    stoplist = set('for a of the and to in , .'.split())
    texts = [model[word] for word in sentence.lower().split() if word
not in stoplist]
    words = [word for word in sentence.lower().split() if word not in
stoplist]
    vec_list = []
    # texts = pd.DataFrame(texts)
    # vec_list = pca.transform(texts)
    for text in texts:
        text = np.reshape(text,(1,-1))
        vec = pca.transform(text)
        vec_list.append(vec)
    return np.array(vec_list),words



# calculate vector similarity by pearson.
def calculate_vector_similarity_by_cos(a,b):
    corr = stats.pearsonr(a,b)
    return corr


# calculate the similarity between a document and a sentence by
calculating all clusters.
def
calculate_document_similarity(clusters,sentence_vecs,model,idf_vec):
    rank_list = []
    for cluster in clusters.iterrows():
        cluster = cluster[1].values
        vec_similarity = []
        for vec in sentence_vecs:
            vec = vec[0]
            similarity =
calculate_vector_similarity_by_cos(vec,cluster)
            vec_similarity.append(similarity)
        vec_similarity = np.array(vec_similarity)

        # vector multiply.
        vec_rank = np.dot(idf_vec,vec_similarity)
        rank_list.append(vec_rank)
    return np.max(rank_list)

# get tf-idf list from database.
def get_tf_idf(document,words):
    conn = sqlite3.connect('./Prehandle/PretreatmentInfo.db')
    cursor = conn.cursor()
    tf_idf_list = []
    for word in words:
        cursor.execute('''SELECT *
                          from tf
                          WHERE FileID = :document and WORD=:word
                         ''',{'document':int(document.name),'word':word})
        tempResult = cursor.fetchall()
        if not tempResult :
            tf_idf_list.append(0)
        else:
            tf_idf_list.append(tempResult[0][2])
    return np.array(tf_idf_list)



'''
file: pca.py
description: script for determining the number of components we keep
in PCA.

'''


from gensim import models
from sklearn.cluster import KMeans
import numpy as np
import os
from sklearn.externals import joblib
import pandas as pd

def is_literature_file(file):
    if file[:4] == 'test':
        return file
    else:
        return None

if __name__ == '__main__':
    # func: test whether the file is a test file.
    txt_dir = '../../Preparement/'
    files = os.listdir(txt_dir)
    files = filter(is_literature_file,files)
    # print(list(files))
    words = ''
    documents = []
    model = models.Word2Vec.load('./model')
    print(len(model['you']))
    stoplist = set('for a of the and to in , .'.split())

    # load the machine learning model we have pretrained.
    # PCA is the PCA model
    pca = joblib.load('train_model.m')

    # Calculate the coveriance matrix for the next step.
    cov = pd.DataFrame(pca.get_covariance())

     # Calculate the coveriance ratio.
    ratio = pca.explained_variance_ratio_
    sumk = []
    # Calculate k_sum(sum of first k elments)
    for i in range(len(ratio)):
        sumk.append(sum(ratio[0:i+1]))

        # make sure when the k_sum is more than 0.8 which means that
it is enough
        if sum(ratio[0:i+1])>0.8:
            print(i)
            break

'''
file: demo.py
description: script for generating the model of words vectors.

'''

import os
from gensim import corpora
from gensim import models

# func: test whether the file is a test file.
def is_literature_file(file):
    if file[:4] == 'test':
        return file
    else:
        return None


if __name__ == '__main__':
    # set the path
    txt_dir = '../../Preparement/'
    files = os.listdir(txt_dir)
    files = filter(is_literature_file,files)

    # load all the files we scrapy from web
    words = ''
    documents = []
    for file in files:
        with open(txt_dir+file,'r') as f:
            document = f.read()
            words = words + document
            documents.append(document)

    # set stoplist set
    stoplist = set('for a of the and to in , .'.split())

    # split words
    texts2 = [[word for word in document.lower().split() if word not
in stoplist] for document in documents]
    texts = [word for word in words.lower().split() if word not in
stoplist]

    # train Word2Vec model
    model =
models.Word2Vec(texts2,size=150,min_count=1,window=5,iter=100)

    # save model
    model.save('./model')



#  Function for the pretreatment,input include the address prefix.
address postfix,
#  the number of the files and the address of the list

def pretreatment(addpre, addpost, numOfFile, addlist):
    # The libraries that we need to import, nltk is a natural
    # language processing library. We use it to handle the texts.
    # math is the library for Calculation
    # sqlite3 is a library which provides the interfaces to the
Database
    import nltk
    import math
    import sqlite3

    # Punctuation is a list created for handling punctuation, since
it is useless
    Punctuation = list('''!@#$%^&*()_+-=[]\;',./{}|:"<>?''')

    # fileList is used to store the files
    fileList = []

    #the number of the files
    numberOfFiles = numOfFile

    # Range is a list used to execute loop
    Range = range(numberOfFiles)

    # Address prefix and postfix for files read
    addressPrefix = addpre
    addressPostfix = addpost

    for i in Range:
        # append files in the system to the list
        fileList.append(open(addressPrefix + str(i) + addressPostfix))

    # titleDic is used for recording titles for every document
    titleDic = {}

    # handle files based on their attributes
    for i in Range:
        temp = str(fileList[i].readline())

        while temp.replace(' ','') == '\n':
            temp = fileList[i].readline()
        if i < 38:
            length = len(temp) - 14
            titleDic[i] = temp[2:length]
        else :
            titleDic[i] = temp
    #    print(temp[:length])

    rawFileList = [fileList[i].read() for i in Range]
    # print(type(rawFileList[1]))
    tokensList = [nltk.word_tokenize(rawFileList[i]) for i in Range]
    # text which a list
    textList = [nltk.Text(tokensList[i]) for i in Range]
    # fdistList is a dictionary
    fdistList = [nltk.FreqDist(textList[i]) for i in Range]
    for i in Range:
        for punc in Punctuation:
            if punc in fdistList[i].keys():
                fdistList[i].pop(punc)

    # invertedFile is a dictionary, keys are words, values are list
for corresponding documents
    invertedFile = {}

    # tf and idf are the values to describe importance for every word
in a document
    tfFile = []
    idfFile = []
    testset = set()

    # generate the invertedFile
    for i in Range:
        for j in fdistList[i].keys():
            testset.add(j)
            if invertedFile.get(j) is None:
                invertedFile[j] = [str(i)]
            else:
                invertedFile[j].append(str(i))

    # generate the tfFile and the idfFile
    for i in Range:
        fileLength = len(textList[i])
        tfDic = {}
        idfDic = {}
    # tf = the times of a word occuring / the total length of the
file
    # idf = log(the total number of the files / (1 + the number of
the special files))

        for j in fdistList[i].keys():
            tfDic[j] = fdistList[i][j] / fileLength
            idfDic[j] = math.log(numberOfFiles /
(1+len(invertedFile[j])))
```

```
        tfFile.append(tfDic)
        idfFile.append(idfDic)

    indexDic = {}

    # the address for the index list
    ListAddress = addlist
    file = open(ListAddress)
    for i in file.readlines():
        pair = i.split(':')
        sel = len(pair[1]) - 1
        url =  'shakespeare.mit.edu/' + ('Poetry/' if int(pair[0]) >=
42 else '') + pair[1][1:]
       # print(url)
        indexDic[int(pair[0])] = url

    # create the database
    # connect to the database and create the corresponding table

    conn = sqlite3.connect('PretreatmentInfo.db')

    print("Database created successfully")
    cursor = conn.cursor()

    # create the table for invertedFile
    cursor.execute('''create table InvertedFile
                   (WORD TEXT PRIMARY KEY  NOT NULL,
                     FileNumber   TEXT     NOT NULL
                   );''')

    print("Table created successfully")


    # create the table for file index
    for i in invertedFile.keys():
        stringi = " ".join(invertedFile[i])
        cursor.execute('''INSERT INTO InvertedFile (WORD, FileNumber)
                       VALUES (?,?)
                       ''',(i,stringi))

    cursor.execute('''create table tf_idf
                   (FileID        INT     NOT NULL,
                     WORD         TEXT    NOT NULL,
                     VALUE         REAL     NOT NULL
```

```
                );''')

    # create the tf-idf table, FileID is represented for the ID of
the file, WORD is
    # represented for the corresponding word. VALUE is represented
for the tf-idf value
    for i in Range:
        for j in fdistList[i].keys():
            value = tfFile[i][j] * idfFile[i][j]
            cursor.execute('''INSERT INTO tf_idf (FileID, WORD, VALUE)
                        VALUES (?,?,?)
                        ''', (i,j,value))


    # create the tf table, FileID is represented for the ID of the
file, WORD is
    # represented for the corresponding word. VALUE is represented
for the tf value
    cursor.execute('''create table tf
                (FileID        INT     NOT NULL,
                 WORD         TEXT     NOT NULL,
                 VALUE         REAL     NOT NULL
                );''')

    for i in Range:
        for j in fdistList[i].keys():
            value = tfFile[i][j]
            cursor.execute('''INSERT INTO tf (FileID, WORD, VALUE)
                        VALUES (?,?,?)
                        ''', (i,j,value))

    # create the idf table, FileID is represented for the ID of the
file, WORD is
    # represented for the corresponding word. VALUE is represented
for the idf value
    cursor.execute('''create table idf
                (FileID        INT     NOT NULL,
                 WORD         TEXT     NOT NULL,
                 VALUE         REAL     NOT NULL
                );''')

    for i in Range:
        for j in fdistList[i].keys():
            value = idfFile[i][j]
            cursor.execute('''INSERT INTO idf (FileID, WORD, VALUE)
                               VALUES (?,?,?)
                               ''', (i,j,value))

    # create index table for the file index
    # FileID: ID of the file
    # Title: Title of the file
    # URL: URL of the file
    cursor.execute('''create table urlTitleIndex
                  ( FileID INT    NOT NULL,
                    Title  TEXT   NOT NULL,
                    URL    TEXT   NOT NULL
                  );  ''')


    for i in Range:
        if i in titleDic.keys() and i in indexDic.keys():
            cursor.execute('''INSERT INTO urlTitleIndex
(FileID,Title,URL)
                               VALUES (?,?,?)
                               ''', (int(i), titleDic[i],indexDic[i]))

    # commit the changes
    # close the database
    conn.commit()
    conn.close()

num = 195
import os
print(os.getcwd())
addr1 = '../Preparement/list.txt'
addr2 = '../Preparement/test'
addr3 = '.txt'

pretreatment(addr2, addr3, num, addr1)

import sqlite3
from PySrc.Document import Document

# the function for searching. Input is a string and output is a list
# and every element in the list is a tuple with three attributes:
# FileID,URL and Title
```

```
def Search(inputStr):
    # convert the string into a list for analyzing later
    inputList = inputStr.split()

    # matchFile is a set for the files matching the input string
    matchFile = set()

    # connect to the database
    conn = sqlite3.connect('./Prehandle/PretreatmentInfo.db')
    cursor = conn.cursor()
    a = list()

    # we should select the information about the documents
    for i in inputList:
        cursor.execute('''select FileNumber
                       from InvertedFile where WORD = ?''',(i,))

    # fetchall is a function to fetch the contents we select before
        result = cursor.fetchall()
        resultList = []
    # we should judge if it is a empty list first
        if  len(result) > 0 and len(result[0]) > 0:
            resultList = result[0][0].split()
        for j in resultList:
            matchFile.add(int(j))
    resultList = []

    # generate the result list
    for i in matchFile:
        cursor.execute('''SELECT *
                       from urlTitleIndex
                       WHERE FileID = ?
                      ''',(i,))
        tempResult = cursor.fetchall()
        if tempResult:
            document = Document(tempResult[0])
            resultList.append(document)
        else:
            pass

    # close the database
    conn.close()

    return resultList
```

Chapter: 6 Declaration

We hereby declare that all the work done in this project titled "Roll Your Own Mini Search Engine" is of our independent effort as a group.