Report for building a mini search engine and some investigations

191840273 Xing Zeming
201300013 Liang Sicheng

2023.06.16

Abstract

As we have learned a lot about information retrieval this term, for practice, we build a mini search engine. We will present and analyze our work in this report, including explaining the structure and meaning of our code, and we will also report on some investigations on advanced IR technologies.

1 Introduction

To practice what we have learned about information retrieval, we build a mini search engine. Our work can be summarized as three parts: indexing, k-way merging, searching and ranking. In the next sections, we will talk about our work and analyze our code for every single part.

Our data are all from Wikipedia. We didn't design the crawler, and we downloaded the dump directly. We have tried to use the whole data from Wikipedia, but the XML of it is about 90GB and too huge to analyze and index. So we chose the abstract version of Wikipedia which is around 7GB and much smaller than the former one. At the same time, we gain faster search performance but lose search accuracy.

After talking about the mini search engine, we will talk about FAISS, a vector retrieval framework open-sourced by Facebook. We will briefly explain how it works.

2 Mini search engine

2.1 Indexing

In this part, we chose to call the API to parse the XML and stem words, so we just focused on the indexing.

We use this function to load stopwords.

```
stopwordsList = set()
with open("stopwords.txt", 'r') as f:
    for line in f:
```

```
        line = line.strip()
        stopwordsList.add(line)
```

To use the parse function from xml.sax, we need to define a handler class.

```
class WikiContentHandler(ContentHandler):
    def __init__(self):
        self.docID = -1
        self.isTitle = False
        self.title = ""
        self.buffer = ""
        self.url =""

    def characters(self, content):
        self.buffer = self.buffer + content

    def startElement(self, element, attributes):
        print(element)
        if element == "title":
            self.buffer = ""
            self.isTitle = True
        if element == "doc":
            self.docID += 1
        if element == "abstract":
            self.buffer = ""

    def endElement(self, element):
        if element == "title":
            processBuffer(self.buffer, self.docID, self.isTitle)
            self.isTitle = False
            self.title = self.buffer
            self.buffer = ""
        elif element == "url":
            self.url = self.buffer[1:]
            self.buffer = ""
        elif element == "abstract":
            processBuffer(self.buffer, self.docID, self.isTitle)
            try:
                documentTitleMapping.write(str(self.docID)+"#"+self.title+'   url:'+self.
                                                                    url +"\n")

            except:
                documentTitleMapping.write(
                    str(self.docID)+"#"+self.title.encode('utf-8')+'   url:'+self.url +"\n
                                                                    ")
            self.buffer = ""
```

When searching, we want to give a higher rank to urls(pages) if the words in the query are in the title of them than the urls which just contain the words in the query in their bodies. So we use a variable IsTitle to detect

whether the text of this element is in the title of the page. In the function endElement we progress the buffer of the element. We will introduce the function progressbuffer later and now we focus on the function endElement. When the element is title we progress the buffer, set IsTitle false, and record the title using self.title. When the element is url, we record the url using self.url. When the element is abstract, we progress the buffer and do indexing in dictionary doucumentTitleMapping which can be used to print the result after the searching and ranking.

Now we talk about the function progressbuffer.

```
def processBuffer(text, docID, isTitle):
    global path_to_index
    text = text.lower()
    text = cleanText(text)
    regExp.sub(' ', text)
    words = text.split()
    tokens = list()
    for word in words:
        if word not in stopwordsList:
            tokens.append(word.strip())
    if isTitle == True:
        addToIndex(tokens, docID, "t")
    else:
        addToIndex(tokens,docID,'b')
        if docID%fileLim == 0:
                f = open(path_to_index + "/" + str(docID) + ".txt", "w")
                for key, val in sorted(invertedIndex.items()):
                    s = str(key)+"="
                    for k, v in sorted(val.items()):
                        s += str(k) + ":"
                        for k1, v1 in v.items():
                            s = s + str(k1) + str(v1) + "#"
                        s = s[:-1]+","
                    f.write(s[:-1]+"\n")
                f.close()
                invertedIndex.clear()
                stemmingMap.clear()
```

Actually, the code is easy to understand. The function cleanText is designed to clean the text using regular expression. The first loop is to delete words in the stopwords set. And then according to the variable IsTitle, we add the tokens to the index. And after we add the body of a page to the index, we need to identify whether an index file is full, as we set the size limit of a file is 25000. If the index size reaches the limit, we will write the index to a file.

And then we show the function addToIndex. It is easy to understand so we will skip it.

```
def addToIndex(wordList, docID, t):
    for word in wordList:
        word = word.strip()
        word = re.sub(r'[\ \.\-\:\&\$\!\*\+\%\,\@]+',"",word)
```

```
        if len(word) >= 3 and len(word) <= 500 and word not in stopwordsList:
            if word not in stemmingMap.keys():
                stemmingMap[word] = ps.stem(word)
            word = stemmingMap[word]
            if word not in stopwordsList:
                if word in invertedIndex:
                    if docID in invertedIndex[word]:
                        if t in invertedIndex[word][docID]:
                            invertedIndex[word][docID][t] += 1
                        else:
                            invertedIndex[word][docID][t] = 1
                    else:
                        invertedIndex[word][docID] = {t: 1}
                else:
                    invertedIndex[word] = dict({docID: {t: 1}})
```

2.2 K-way merging

In this part, we use the k-way mergesort algorithm to merge the index that we get in the first index part. We will only explain the function kWayMerge.

```
def kWayMerge():
    global total
    for i in range(numOfSplittedFiles):
        processedFiles[i] = 1
        try:
            filePointers[i] = open(splittedFilePathList[i], 'r')
        except:
            pass
        currentRowofFile[i] = filePointers[i].readline()
        termDict[i] = currentRowofFile[i].strip().split('=')
        if termDict[i][0] not in kWayHeap:
            heappush(kWayHeap, termDict[i][0])

    while True:
        if processedFiles.count(0) == numOfSplittedFiles:
            break
        else:
            total += 1
            word = heappop(kWayHeap)
            for i in range(numOfSplittedFiles):
                if processedFiles[i] and termDict[i][0] == word:
                    if word not in invertedIndex:
                        invertedIndex[word] = termDict[i][1]
                    else:
                        invertedIndex[word] += ',' + termDict[i][1]
```

```
                    currentRowofFile[i] = \
                        filePointers[i].readline().strip()

                    if currentRowofFile[i]:
                        termDict[i] = currentRowofFile[i].split('=')
                        if termDict[i][0] not in kWayHeap:
                            heappush(kWayHeap, termDict[i][0])
                    else:
                        processedFiles[i] = 0
                        filePointers[i].close()
                        os.remove(splittedFilePathList[i])
            if total >= chunkSize:
                total = 0
                writePrimaryIndex()
                invertedIndex.clear()
```

We note that the variable numOfSplittedFiles represents the number of files we get after indexing. In the first loop, we load files and build a heap using the first line of every file. And then we use the heap to pop the "least" word and add it to the second index. When the size of the index reaches the limit size of chunks, we write it to a file. At the end, we get the merged index, and we use 'secondindex.txt' to record the first word of every file we get in the second indexing.

2.3 Searching and Ranking

In this part, we search in the second index and get the posting list, then we rank the docs we get and print the results. The following two functions both are using binary search to get the target.

```
def getFileNumber(word):
    position = bisect(secondaryIndex, word)
    if position - 1 >= 0 and secondaryIndex[position - 1] == word:
        if position - 1 != 0:
            position -= 1
        if position + 1 == len(secondaryIndex) \
            and secondaryIndex[position] == word:
            position += 1
    return position


def getPostingList(word):
    position = getFileNumber(word)
    primaryFile = 'finalIndex/index' + str(position) + '.txt'
    file = open(primaryFile, 'r')
    data = file.readlines()
    low = 0
    high = len(data)
    mid = int()
    while low <= high:
```

```
        mid = int(low + (high - low) / 2)
        cur = data[mid].split('=')[0]
        if cur == word:
            break
        elif cur < word:
            low = mid + 1
        else:
            high = mid - 1
    return data[mid].split('=')[1].split(',')
```

The function getFileNumber is used to find the word's index file position. And the function getPostingList is used to find the line in the file containing the word we search and get the posting list.

The function parseQuery is too long so that we only show a part of it.

```
        for word in finalTokens:
            postingList = getPostingList(word)
            numDoc = len(postingList)
            idf = log10(noDocs / numDoc)
            for pl in postingList:
                docId, freqList = pl.split(":")
                categoryFreq = freqList.split("#")
                tf = 0
                for cf in categoryFreq:
                    cat = cf[0]
                    freq = int(cf[1:])
                    tf += (freq * weight[cat])
                finalDict[docId] += float(log10(1 + tf)) * float(idf)
```

As we can see in the code, we use tf-idf to rank the docs. Actually, we want to implement BM25 in the beginning. But because we use the abstract version of Wikipedia, we think tf-idf is enough.

2.4 Summary

Now, we have finished showing the framework of our mini search engine. Due to time constraints, we did not design the interface for it. If you want to see its final performance, please follow the instructions in the README file to run the code one by one.

3 Introduction of FAISS

With the development of deep learning techniques, multiplex recall in recommendation and search systems often includes vector recall. The simplest way to do it is violent enumeration. Assuming that the number of candidate sets is N and the vector dimension is D. The complexity of violent enumeration is $N \times D$. In order to improve the retrieval speed, approximate search (ANN) has been proposed, since the size of online candidate pools in real business is generally in the order of millions or even hundreds of millions.

FAISS is a vector retrieval framework open-sourced by FaceBook. given a vector it will find a set of vectors with higher similarity to it than the others in all known vector libraries. The core algorithm in FAISS is product quantization(PQ), where the product is the Cartesian product. This means that the original vector is decomposed into a Cartesian product of several low-dimensional vectors and the resulting low-dimensional vector space is quantized so that the original vector can be represented by the quantized encoding of the low-dimensional vectors.

3.1 Product Quantization

PQ splits the high-dimensional vector into multiple subvectors, each of which is compressed into a single number so that the high-dimensional vector can be represented by several numbers. The progress is as follows: first, N D-dimensional vectors are decomposed into M groups that every group has N D/M-dimensional vectors.Then each set of subvectors is clustered using k-means so that each set of subvectors has K mapping result named **codebook**. Each mapping result can be either a cluster center vector or a category ID, and the original vector can be expressed as a Cartesian product of codebooks.

As the paper ***Product quantization for nearest neighbor search*** showed $K = 256$ and $M = 8$ is often a reasonable choice.

3.2 Vector Retrieval

There are two ways to compute the Euclidean distance between the query vector and the candidate vector: Symmetric distance computation (SDC) and Asymmetric distance computation (ADC), i.e., symmetric and asymmetric computation. The difference between them is whether the query vector is quantized or not.

SDC represents the query vector and the candidate vector separately using the cluster centroids, and the distance between the vectors can be approximated by the distance between the cluster centroid vectors. Here is the format of it where $q_j(x)$ denotes the quantized version of the jth subvector of vector x.

$$\hat{d}(x, y) = d(q(x), q(y)) = \sqrt{\sum_j d\left(q_j(x), q_j(y)\right)^2}$$

ADC represents the candidate vector by the cluster center vector and represents the query vector by the original vector, and the distance between the vectors is calculated as follows, where $u_j(x)$ denotes the jth subvector of vector x.

$$\hat{d}(x, y) = d(x, q(y)) = \sqrt{\sum_j d\left(u_j(x), q_j(u_j(y))\right)^2}$$

4 Inverted Index

In order to be able to handle queries in the tens of millions or even billions, it is necessary to use inverted indexes to speed up queries.

The algorithm first clusters N vectors using k-means into $k^{'}$ classes. Each class is represented by a cluster center vector, and then the residual vector of the original vector and its cluster center vector is calculated, and encoded using the PQ quantizer described above. The encoded result is pegged to the inverted chain of class IDs where the original vector is located.

In the first step of clustering, two neighboring vectors may not necessarily fall in the same cluster, so to improve the recall rate, we will retrieve from ω clusters. The retrieval process of query vector x is as follows:

1. Find the ω closest clustering centers to the vector x.
2. For each clustering center, calculate the residual vector of the lookup vector and the clustering center.
3. Iterate through the inverted chain of each clustering center, calculate the distance between the query vector and the candidate vector on the inverted chain according to the ADC algorithm in the previous section and take the top k vectors with the smallest distance.

Assuming that the inverted chain is balanced, we will have $\frac{N}{k^{'}} \times \omega$ candidates to retrieve while we have N candidates when not using the inverted index. Retrieval accuracy is related to $k^{'}$ and ω. When k is the same, the greater the w, the higher the accuracy, and when w is the same, the smaller the k, the higher the accuracy.

5 Contributions

Xing Zeming: the indexing part of the code, investigation of FAISS, writing the report.

Liang Sicheng: the kwaymerge part and the searching part of the code, investigation of FAISS, uniformity of code style and debugging.