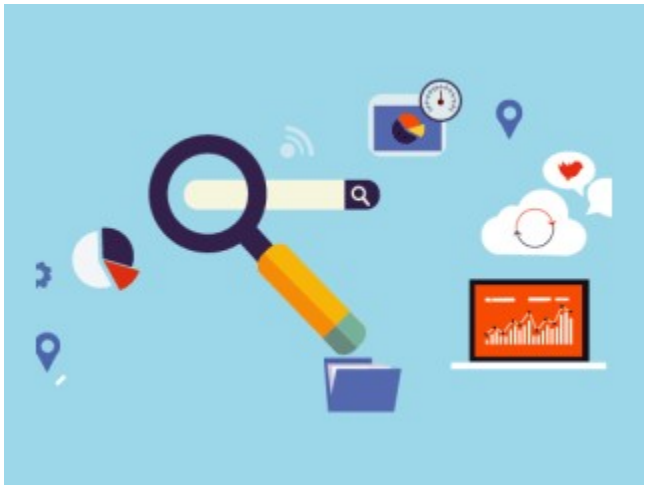
Published in Analytics Vidhya

Dhaval Taunk
Oct 10, 2021 · 6 min read · Listen

Search Engine in Python from scratch



Source:- https://www.pinterest.com/pin/847450854860596454/?amp_client_id=CLIENT_ID(_)&mweb_unauth_id=&simplified=true

In this post, I will be going through all the details of building a search engine from scratch using the Wikipedia dump (approximately 84GB in size). I will be going through the step-by-step process of creating a primary index of

Dhaval Taunk
90 Followers
MS by Research @IIITH, Ex Data Scientist @ Yes Bank | Former Intern @ Haptik, IIT Guwahati | Machine Learning | Deep Learning | NLP
Follow

More from Medium

Ramiz Sami
WebGIS Development in 2023: A Guide to the Tools and Technologies I Use for...

Somnath ... in JavaScript in Plai...
Coding Won't Exist In 5 Years. This Is Why

Josep Ferrer in Geek Culture
6 ChatGPT mind-blowing extensions to use it anywhere

Anmol Tomar in CodeX

Say Goodbye to Loops in Python, and Welcome Vectorization!

data and a secondary index—also, how to implement search

functionality to output the results in minimum time and all. I will be dividing the post into two parts viz Requirements, Wiki dump details, Indexing and Searching. So tighten your seat belts, and let's get started.

1. Requirements

> *Python3*
>
> *NLTK stopwords*
>
> *PyStemmer*

2. Wiki dump details

For creating the search engine, in this post, I will be referring to the **Wikipedia** dump for the English language, which is approximately **84GB** in size. One can download the data from the below-given link:-

> *https://dumps.wikimedia.org/enwiki/20210720/enwiki-*

> *20210720-pages-articles-multistream.xml.bz2*

You will need to download and extract the downloaded dump. Alternatively, you can work on the zipped data as well. I will be explaining all the things on the extracted dump.

3. Creating the Index for Wiki dump

(i) Parsing the XML dump

First of all, you will be required to parse the XML and get the necessary data. For this, there are a couple of parsers available in python. Some of them are:-

> *SAX*
>
> *Etree*
>
> *DOM*

Here I will be using SAX parser to parse the XML. You can try out different other parsers as well. In the below gist, I have

shown how to use the SAX parser to parse the data.

```
class XMLParser(xml.sax.ContentHandler):

        def __init__(self, page_processor, create_index):

                self.tag = ''
                self.title = ''
                self.text = ''
                self.page_processor = page_processor
                self.create_index = create_index

        def startElement(self,name,attrs):

                self.tag=name

        def endElement(self,name):

                if name=='page':

                        print(num_pages)

                        id_title_map[num_pages]=self.title.lower()
                        title, body, category, infobox, link, reference = self

                        self.create_index.index(title, body, category, infobox

                        self.tag = ""
                        self.title = ""
                        self.text = ""

        def characters(self, content):

```

I am using only two fields, i.e. title and text, from XML in the above code. I am giving my own id's from the variable num_pages. How I am using the title, and the text is explained below in different sections.

(ii) How to preprocess text

It is an essential task, as this step will ensure we are not adding unnecessary terms to the Index. Otherwise, it will blow the index size. Majorly, I will remove stopwords, tokenise the text, remove HTML tags, remove non-ASCII characters, etc. It is shown in the below code.

```
class TextPreProcessor():

        def __init__(self, html_tags, stemmer, stop_words):

                self.html_tags=html_tags
                self.stemmer=stemmer
                self.stop_words=stop_words

        def remove_stopwords(self, text_data):

                cleaned_text = [word for word in text_data if word not in self

                return cleaned_text

        def stem_text(self, text_data):

                cleaned_text = self.stemmer.stemWords(text_data)

                return cleaned_text

        def remove_non_ascii(self, text_data):

                cleaned_text = ''.join([i if ord(i) < 128 else ' ' for i in te

                return cleaned_text

        def remove_html_tags(self, text_data):

                cleaned_text = re.sub(self.html_tags, ' ', text_data)

                return cleaned_text

        def remove_special_chars(self, text_data):
```

```
34
35                cleaned_text = ''.join(ch if ch.isalnum() else ' ' for ch in t
36
37                return cleaned_text
38
39        def remove_select_keywords(self, text_data):
40
41                text_data = text_data.replace('\n', ' ').replace('File:', ' ')
42                text_data = re.sub('(http://[^ ]+)', ' ', text_data)
43                text_data = re.sub('(https://[^ ]+)', ' ', text_data)
44
45                return text_data
46
47        def tokenize_sentence(self, text_data, flag=False):
48
49                if flag:
50                        text_data = self.remove_select_keywords(text_data)
51                        text_data = re.sub('\{.*?\}|\[.*?\]|\=\=.*?\=\=', ' ',
52                cleaned_text = self.remove_non_ascii(text_data)
53                cleaned_text = self.remove_html_tags(cleaned_text)
54                cleaned_text = self.remove_special_chars(cleaned_text)
```

(iii) Extract different fields

There can be different possible fields on which we can do query searching. I will be using six fields: title, body, category, infobox, links, and references. One can search generic queries or field-specific queries using these fields.

```
class PageProcessor():

        def __init__(self, text_pre_processor):

                self.text_pre_processor=text_pre_processor

        def process_title(self, title):

                cleaned_title = self.text_pre_processor.preprocess_text(title

                return cleaned_title

        def process_infobox(self, text):

                cleaned_infobox=[]
                try:
                        text = text.split('\n')
                        i=0
                        while '{{Infobox' not in text[i]:
                                i+=1

                        data = []
                        data.append(text[i].replace('{{Infobox', ' '))
                        i+=1

                        while text[i]!='}}':
                                if '{{Infobox' in text[i]:
                                        dt = text[i].replace('{{Infobox', ' '
                                        data.append(dt)
                                else:
                                        data.append(text[i])
                                i+=1

```

```
34                      infobox_data =  ' '.join(data)
35
36                      cleaned_infobox = self.text_pre_processor.preprocess_
37              except:
38                      pass
39
40              return cleaned_infobox
41
42      def process_text_body(self, text):
43
44              cleaned_text_body = []
45              cleaned_text_body = text_pre_processor.preprocess_text(text,
46
47              return cleaned_text_body
48
49      def process_category(self, text):
50
51              cleaned_category = []
52              try:
53                      text = text.split('\n')
54                      i=0
55                      while not text[i].startswith('[[Category:'):
56                              i+=1
57
58                      data = []
59                      data.append(text[i].replace('[[Category:', ' ').repla
60                      i+=1
61
62                      while text[i].endswith(']]'):
63                              dt = text[i].replace('[[Category:', ' ').repl
64                              data.append(dt)
65                              i+=1
```

```
66
67                      category_data =  ' '.join(data)
68                      cleaned_category = self.text_pre_processor.preprocess
69              except:
70                      pass
71
72              return cleaned_category
73
74      def process_links(self, text):
75
76              cleaned_links = []
77              try:
78                      links = ''
79                      text = text.split("==External links==")
80
81                      if len(text)>1:
82                              text=text[1].split("\n")[1:]
83                              for txt in text:
84                                      if txt=='':
85                                              break
86                                      if txt[0]=='*':
87                                              text_split=txt.split(' ')
88                                              link=[wd for wd in text_split
89                                              link=' '.join(link)
90                                              links+=' '+link
91
92                      cleaned_links = self.text_pre_processor.preprocess_te
93              except:
94                      pass
95
96              return cleaned_links
97
98      def process_references(self, text):
```

```
 99
100                 cleaned_references = []
101                 try:
102                         references = ''
103                         text = text.split('==References==')
104
105                         if len(text)>1:
106                                 text=text[1].split("\n")[1:]
107                                 for txt in text:
108                                         if txt=='':
109                                                 break
```

(iv) Creating an intermediate index

Creating the final Index directly will be a heavy task and can blow up memory as well. Therefore, we will first create an intermediate index on data sections and then perform the final merging to make a final index. We will be using the SPIMI approach. You can read more about it using the below link.

Single-pass in-memory indexing

Next: Distributed indexing Up: Index construction Previous: Blocked sort-based indexing Contents...

nlp.stanford.edu

The below code shows how to create an intermediate index.

```
class CreateIndex():

        def __init__(self, write_data):

                self.write_data = write_data

        def index(self, title, body, category, infobox, link, reference):

                global num_pages
                global index_map
                global id_title_map

                words_set, title_dict, body_dict, category_dict, infobox_dict,

                words_set.update(title)
                for word in title:
                        title_dict[word]+=1

                words_set.update(body)
                for word in body:
                        body_dict[word]+=1

                words_set.update(category)
                for word in category:
                        category_dict[word]+=1

                words_set.update(infobox)
                for word in infobox:
                        infobox_dict[word]+=1

                words_set.update(link)
                for word in link:
                        link_dict[word]+=1
```

```
34
35                  words_set.update(reference)
36                  for word in reference:
37                          reference_dict[word]+=1
38
39                  for word in words_set:
40                          temp = re.sub(r'^((.)(?!\2\2\2))+$',r'\1', word)
41                          is_rep = len(temp)==len(word)
42
43                          if not is_rep:
44                                  posting = str(num_pages)+':'
45
46                                  if title_dict[word]:
47                                          posting += 't'+str(title_dict[word])
48
49                                  if body_dict[word]:
50                                          posting += 'b'+str(body_dict[word])
51
52                                  if category_dict[word]:
53                                          posting += 'c'+str(category_dict[word]
54
55                                  if infobox_dict[word]:
56                                          posting += 'i'+str(infobox_dict[word])
57
58                                  if link_dict[word]:
59                                          posting += 'l'+str(link_dict[word])
60
61                                  if reference_dict[word]:
62                                          posting += 'r'+str(reference_dict[word
63
64                                  posting+=';'
```

Here I am processing the text, token by token, adding it to a

dictionary, and then writing it in intermediate files. The format of index files now looks this:-

```
apple-2314:t3b6i2r1;6432:t5c8b3i1;
```

Here in the above example, "*apple*" is a token. Then after the hyphen, every pair is separated by ";". The first value before ":" represents the docID for the document, and then the corresponding string and value represents frequency.

> *Ex.:- t3b6i2r1 represents token appears 3 times in title, 6 times in the body, 2 times in infobox and 1time in reference.*

(iv) Merging the intermediate index

Now when we are done with writing the intermediate Index. We need to merge the indexes because there will be instances where the token will have its info split into multiple files. We need to merge it for creating the final Index. The below code do this:-

```
class MergeFiles():

        def __init__(self, num_itermed_files, write_data):

                self.num_itermed_files = num_itermed_files
                self.write_data = write_data

        def merge_files(self):

                files_data = {}
                line = {}
                postings = {}
                is_file_empty = {i:1 for i in range(self.num_itermed_files)}
                tokens = []

                i = 0
                while i < self.num_itermed_files:

                        files_data[i] = open(f'../output_data/english_wiki_ind
                        line[i] = files_data[i].readline().strip('\n')
                        postings[i] = line[i].split('-')
                        is_file_empty[i]=0
                        new_token = postings[i][0]
                        if new_token not in tokens:
                                tokens.append(new_token)
                        i+=1

                tokens.sort(reverse=True)
                num_processed_postings=0
                data_to_merge = defaultdict(str)
                num_files_final = 0

                while sum(is_file_empty.values()) != self.num_itermed_files:
```

```
34
35                      token = tokens.pop()
36                      num_processed_postings+=1
37
38                      if num_processed_postings%30000==0:
39
40                              num_files_final = self.write_data.write_final_
41
42                              data_to_merge = defaultdict(str)
43
44                              # # Uncomment this if there is memory issue
45                              # if num_processed_postings%150000==0:
46                              #       password = 'your_password'
47                              #       command = 'sh -c "sync; echo 3 > /proc
48                              #       shell_script = f"echo {password} | sud
49                              #       os.system(shell_script)
50                              #       print('Cache cleared')
51
52                      i=0
53                      while i < self.num_itermed_files:
54
55                              if is_file_empty[i]==0:
56
57                                      if token==postings[i][0]:
58
59                                              line[i] = files_data[i].readli
60                                              data_to_merge[token]+=postings
61
62                                              if len(line[i]):
63                                                      postings[i]=line[i].sp
64                                                      new_token = postings[i
65
```

```
66                                                  if new_token not in to
67                                                      tokens.append(
```

The format of the final Index looks like the below example:-

```
apple-2141:5;1232:1;5432:78;
```

Here in the above example, "*apple*" is a token. Then after the hyphen, every pair is separated by ";". The first value before ":" represents the docID for the document, and the second value represents the frequency of the token for that particular token from the field file. Here in the final files are separated fields wise files, unlike in intermediate indexes.

In this way, the final Index will look similar to the above example shown. If you want to see the entire code, you can visit the Github code link given at the last of this blog and also try out different approaches you want to try out.

(v) Secondary Index

You can create a secondary index as and when required. This will help in searching fast by keeping the information of tokens in the secondary Index. I will be using the below format of the

secondary Index. One can try out other possible forms as well.

```
apple-563-1-3-4--2-4-6-
```

According to the above format, *'apple'* will be the token. Then the value 563 will denote the frequency of apple in the entire Wikipedia dump. After that, the number 1 will indicate in which file number the token is present. The token can appear in any field. The number 1 will suggest that if the token is present in that field, it will only be there in that particular file number for that field. After that, all other values will be optional. I will be going about the details of the field values below:-

As mentioned above, I will be using fields title, body, category, infobox, link, reference. So the values between '-' will denote the line number in the final index files. Below example shows it:-

```
'apple-563-1-3-4--2-4-6-'.split('-') ---> ['apple',
```

```
'563', '1', '3', '4', '', '2', '4', '6', '']
```

Here the fourth element ('3') indicates that 'apple' appears at line number 3 of file number 1 for the title file. Similarly, '4' will show that it will be present at line number 4 of the body file; it does not appear in the category file as it is an empty string (''), it appears at line number '2' for the infobox file, it appears at line number '4' of link file, and at line number '6' in the reference file. One thing that will be common for all the files is that if the token is present for any field, it will be present in that particular file number only, nowhere else.

4. Implementing Search functionality

One of the crucial things required for searching functionality is implementing the ranking functionality to rank the document according to its relevance. But before that, few other things are also needed, which I will be explaining below:-

(i) Preprocessing the query

The preprocessing will be the same which we did during the

indexing phase. All the steps will be the same as the initial steps. Then we will get the final preprocessed query.

(ii) Identify query type

This is one of the mode useful steps as it will guide us whether we need to the simple query or field query or both. So basically a query can be of 3 types:-

```
Type 1:- world war II
Type 2:- t:world cup i:2012
Type 3:- Sachin Tendulkar t:world cup i:2012
```

We can identify the type of query using the below code-

```
def identify_query_type(self, query):

        field_replace_map = {
                        ' t:':';t:',
                        ' b:':';b:',
                        ' c:':';c:',
                        ' i:':';i:',
                        ' l:':';l:',
                        ' r:':';r:',
                }

        if ('t:' in query or 'b:' in query or 'c:' in query or 'i:' in query o

                for k, v in field_replace_map.items():
                        if k in query:
                                query = query.replace(k, v)

                query = query.lstrip(';')

                return query.split(';')[0], query.split(';')[1:]

        elif 't:' in query or 'b:' in query or 'c:' in query or 'i:' in query

                for k, v in field_replace_map.items():
                        if k in query:
                                query = query.replace(k, v)

                query = query.lstrip(';')

```

(iii) Ranking functionality

Ranking functionality is used to get the ranked results. So that the relevant results are on top. Usually, tf-idf is a metric to rank documents.

So what is tf-idf? It is composed of two terms **TF** and **IDF.**

TF (Term Frequency):- It tells us the frequency of a term in a document.

IDF (Inverse Document Frequency):- In documents, a lot of words that occur have a large frequency. But this large frequency reduces the relevance of the word for that document. So to reduce the effect of that large frequency words, IDF is used. It is basically a log of the total number of documents divided by the frequency of the word for that document.

So the final tf-idf value is the product of tf and idf values.

```
tf-idf = tf * idf
```

(iv) Variants of tf-idf

Search Engines Information Retrieval Information Extraction NLP

Term frequency		Document frequency		Normalization	
n (natural)	$tf_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(tf_{t,d})$	t (idf)	$\log \frac{N}{df_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented)	$0.5 + \frac{0.5 \times tf_{t,d}}{\max_t(tf_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-df_t}{df_t}\}$	u (pivoted unique)	$1/u$ (Section 6.4.4)
b (boolean)	$\begin{cases} 1 & \text{if } tf_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(tf_{t,d})}{1+\log(ave_{t \in d}(tf_{t,d}))}$				

Source:- https://nlp.stanford.edu/IR-book/pdf/06vect.pdf

One can use any one of the above-given tf-idf variants. It's up to their implementation and choice.

So, it is all for now from my side. If you want to see the full code, you can visit the below link:-

GitHub - DhavalTaunk08/Wiki-Search-Engine

Contribute to DhavalTaunk08/Wiki-Search-Engine development by creating an account on GitHub.

github.com

If you want to read more about machine learning, deep learning, do visit the below link:-

Dhaval Taunk - Medium

In one of my last blog post, How to fine-tune bert on text classification task , I had explained fine-tuning BERT for...

medium.com

Happy reading.....