Vertical search engine and text classifier

Dominik Klepl

This report outlines an implementation of a vertical search engine for news articles and a machine learning model for classification of topic of a text. The whole project was implemented using Python 3.7.5 programming language. The complete source code of both tools is included in the appendix. The dataset used for training the model is provided as well.

First section of the report describes the implementation of the vertical search engine including all of its functions. We begin with the crawler component of the search engine. Next the process of creating and updating the index is described. Finally, the implementation of query processor and user interface of the search engine is described.

The second section of the report is focused on the text classifier. First the dataset used for training and validating the model is introduced. Then the process of training and selecting the best performing model together with the limitations of the procedure is discussed. Lastly the deployment of the model for interaction with users is discussed.

1. Vertical search engine

The search engine described in this report is a vertical engine. This means that it is not all-purpose search engine but rather is specialized for finding news articles only.

1.1 Crawler

The first component of the search engine is the crawler which visits a URL, extracts information about news articles such as title, summary, link and date published and saves it in a database of articles. Our crawler is a RSS feed reader.

Five different RSS feeds were chosen for the crawler to extract articles from. The list of used RSS feeds is showed in the figure 1.

1. http://feeds.bbci.co.uk/news/world/rss.xml
2. http://feeds.bbci.co.uk/news/uk/rss.xml
3. http://www.independent.co.uk/news/uk/rss
4. feed:https://rss.nytimes.com/services/xml/rss/nyt/World.xml
5. feed:https://rss.nytimes.com/services/xml/rss/nyt/US.xml
6. feed://feeds.washingtonpost.com/rss/world?tid=lk_inline_manual_13

Figure 1 List of used RSS feeds

The crawler loads the content of a RSS feed which is then parsed into individual entries, i.e. news articles. Each of the entries is further parsed into title, summary, link and date published. For feed parsing the feedparser package was used. Figure 2 shows how information from each entry is extracted in a structured manner.

```
1. def process_entry(entry, ID):
2.    ID = ID
3.    title = entry.title
4.    summary = entry.summary
5.    link = entry.link
6.    published = str(entry.published_parsed.tm_mday) + '/' + \
7.                str(entry.published_parsed.tm_mon) + '/' + \
8.                str(entry.published_parsed.tm_year)
9.    return [ID, title, summary, link, published]
```

Figure 2 Extracting information about article

The function process_entry accepts single entry form the parsed RSS feed and returns a list of extracted information about the entry. The function also assigns a unique ID to the article. This ID is determined from the number of rows of the database. For example, the first extracted article has ID 1. Therefore, the article ID can be also used as indicator of time that article has been stored in the database for. In lines 6-8 the format of the date is transformed to DD/MM/YYYY so that all dates in the database are stored in the same format. The output of the function is essentially a single row of the database.

This entry processing is then applied to all entries in the feed that are not stored in the database as showed in figure 3.

```
1. old_news = 0  # count how many news in feed were already scraped
2.
3. #database of previously stored news
4. database = pd.read_csv("database.csv", index_col = 'Unnamed: 0')
5.
6. #dataframe for saving the extracted info
7. data = []
8.
9. database_len = len(database) #number of articles in database
10.ID = database_len + 1 #determine the initial ID
11.for entry in entries:
12.        #append only articles that aren't in database
13.   if entry.link not in database['link'].values:
14.      processed = process_entry(entry = entry, ID_n = n)
15.      data.append(processed)
16.      ID += 1 #increase ID value
17. #increase number of already scraped articles
18.   else: old_news += 1
```

Figure 3 Crawling single RSS feed and storing in database

All entries in the feed are processed using a for loop. First, we check whether that article is stored already by looking for its link in the database. If the article is new than it is processed and appended to a list. The value

of ID is also increased so that next processed article receives larger ID value. If the article was stored already the counter of "old_news" is increased. This information will be later used for the hit rate adjustment.

After the for loop is finished the "data" list with processed entries is appended to the database and saved as comma-separated-values file (csv). However, this happens only when at least one new article was found. This process of crawling and storing information in a database is then repeated for all RSS feeds. Now, the crawler has to be able to run automatically, e.g. look for new articles every hour. Furthermore, the time interval within which each RSS feed is crawled should not be fixed. Instead it should be fine-tuned so that the feed is not crawled when there are no new articles. In other words, the time interval should ideally correspond to the time when all articles in the feed are not yet stored in the database. Figure 4 shows implementation of this automated crawling with adjustment of hit rate.

```
1. hit_rate = 900 #start with 15 minutes
2. hit_rates = {}
3. while True:
4.     now = datetime.now()
5.     dt_string = now.strftime("%d/%m/%Y %H:%M:%S")
6.
7.     print("Crawling {}" .format(url))
8.     perc, added = crawler.crawl(URL=url, PATH=OUTPUT_DIR)
9.     # add to index and compute average word2vec
10.    indexer.update_index_vecs(df=added, index_path=INDEX_PATH, vec_path=VECTOR_PATH)
11.
12.    if perc >= 50:
13.        hit_rate += perc*30
14.        print("Increasing hit rate by {} minutes." .format(n/60))
15.    if perc < 50:
16.        hit_rate -= perc*30
17.        print("Decreasing hit rate by {} minutes." .format(n/60))
18.
19.    hit_rates[dt_string] = [perc, hit_rate]
20.    save_rate = pd.DataFrame.from_dict(hit_rates, orient='index')
21.    save_rate.columns = ["old ratio", "hit_rate"]
22.    save_rate.to_csv("data/rate_history.csv")
23.
24.    time.sleep(hit_rate)
```

Figure 4 Automated crawling and hit rate adjustment

First, the hit rate, i.e. the time interval between crawling, is initialized at random, in this case 15 minutes (entered in seconds). We also store the history of the hit rates (line 2) and save them in a file (lines 19-22). In case the crawling is interrupted, the hit rate can be recovered from the file instead of repeating random initialization. In the code snippet, the crawler will run, in theory, forever because the while loop has no option of changing the "True" (line 3) to "False". It can be however, interrupted manually at any point. Each iteration of the crawling consists of measuring the current date and time (line 4-5), crawling the feed and updating

database (8), updating index (decribed in section 1.2) and adjusting the hit rate (line 12-17). The process of crawling was already described above as is wrapped in the crawl function. The output of this function is percentage of articles already stored in database (perc object) and dataframe with new articles (added object). This dataframe is then used to update the index. The perc object is a single number, percentage, which is used to adjust the hit rate. If the percentage is more than or equals 50% the time interval between crawling is increased by 30 seconds per 1%. For example, if 100% of articles have been scrapped before then the time interval is increased by 100 * 30 s (30 minutes). On the other hand, we may need to decrease the hit rate as well. This happens when the percentage is less than 50% and the time is decreased by 30 seconds per 1%. For illustration how the hit rate adjustment looks in practice, the history of crawling one feed is plotted in figure X. This process is then repeated for each RSS feed.

The result of crawling is a database of articles stored as a csv file. A sample of the database is displayed in figure 5.

ID	title	summary	link	published
1	Iran petrol price hike: Protests erupt over surprise rationing	Several cities see protesters take to the streets as petrol prices go up by at least 50%.	https://www.bbc.co.uk/news/world-middle-east-50444429	16/11/2019
2	US election 2020: Obama issues warning to 'revolutionary' Democrats	"The average American doesn't think we have to completely tear down the system," Mr Obama warns.	https://www.bbc.co.uk/news/world-us-canada-50445743	16/11/2019
3	Czech anti-government protesters mark anniversary of revolution	Czech police estimate that about 200,000 people were at the anti-government demonstration.	https://www.bbc.co.uk/news/world-europe-50446661	16/11/2019

Figure 5 Sample of database of articles

1.2 Index and ranked retrieval preparation

An inverted index is used by this search engine. The index is implemented as a Python dictionary. It consists of pairs of keys (words) and posting lists with article IDs that contain that word. A small section of the index is showed in figure 6.

```
1. index = {
2.     "becomes": [812, 1041, 1079],
3.     "bed": [959],
4.     "bednar": [1623],
5.     "bedroom": [1834, 1870],
6.     "bee": [227]
7. }
```

Figure 6 Sample of inverted index

For example, the word “becomes” occurs in 3 articles. Using the database created by the crawler, information about these 3 articles can be recovered.

Before any crawling occurs, the index is empty. After each crawling, the new articles are processed, and their IDs are appended to the index.

First step is text pre-processing as showed in figure 7.

```
1. def process_string(text):
2.   text = text.lower() #to lowercase
3.   text = text.translate(str.maketrans('', '', string.punctuation))
4.   text = stop_lemmatize(text)
5.   return text
```

Figure 7 Processing of text in article

The “text” refers here to the title and summary of an article. First, all words are turned to lowercase (line 1). Next, all punctuation is removed from the text (line 2). Finally, the stop words are removed, and the words are lemmatized (line 3). Removing stop words means to remove words that often do not carry much information about the content of the text. Examples of such words are “I”, “is” or “and”. Lemmatization is process of reducing the words to “normalized” form, e.g. transform verbs to present tense (ran to run), nouns to singular (men to man). An implementation of lemmatizer from the nltk package was used. This implementation requires the words to be tagged, i.e. to which word class they belong (noun, verb etc.). nltk provides a pretrained neural network that performs this tagging.

The pre-processed text is then stored in the index. Insertion of single article in the index is showed in figure 8.

```
1. def index_one(entry, index):
2.   words = entry.text.split()
3.   ID = int(entry.ID)
4.   for word in words:
5.     if word in index.keys():
6.       if ID not in index[word]:
7.         index[word].append(ID)
8.     else:
9.       index[word] = [ID]
10.  return index
```

Figure 8 Adding single article in index

The function takes a single row from the database and the index which is a Python dictionary. First the pre-processed text is split into single words. Next, the ID of that article is extracted and turned to integer. This step is necessary because the index is later written to a JSON file which can store only whole numbers. Then we iterate over the words. First, the code checks whether that word is already in the index. If the word is already present, the code checks whether the ID of the corresponding article is already in the posting list. This step

prevents storing same article multiple times in the same posting list as this might happen if a word is used more than once in article's title and/or summary. Provided that the word is already in the index, but its posting list does not contain the article ID, the ID is appended to the posting list. If the word is not in the index, new word-posting list pair is created, and the ID is inserted in the posting list. The function then returns the updated index.

This process remains the same for all entries and therefore a for loop over the rows of the database can be used. Building/updating the index is wrapped in following function (figure 9).

```
1. def build_index(processed_df, index_path="data/index.json", first_time=False):
2.     if first_time:
3.         index = {}
4.     else:
5.         with open(index_path, 'r') as old_idx:
6.             index = json.load(old_idx)
7.
8.     index = index_all(df=processed_df, index=index)
9.
10.    # write to json
11.    with open(index_path, 'w') as new_idx:
12.        json.dump(index, new_idx, sort_keys=True, indent=4)
```

Figure 9 Building/updating inverted index

The function is written so that it can handle both cases when no index exists yet and where an index was previously created by setting the first_time parameter to true or false. In former case, index is initialized as empty dictionary (lines 1-2). In latter case, the index is loaded from the "index_path" which is path to the index stored as JSON file. The index_all function (line 8) performs a for loop over rows/entries of the database which contains also the pre-processed text. Finally, the index is written to a JSON file (lines 11-12). The additional arguments of the json.dump function assure a more readable format of the index. It orders the words in the index alphabetically and indents the data, so it is more readable for humans.

Every time new articles are added to the database they have to be added to the index as well. This can be done by passing only the newly added rows of the database to the build_index function described above.

In order for the search engine to support ranked retrieval, some additional information needs to be extracted from the articles. Ranked retrieval means to order the results of a search by relevance to the search query. Since this is required for all articles, same as updating the index, the preparation for ranked retrieval is carried out at the same as updating the index. Furthermore, the ranked retrieval functions use the same pre-processed text as the index updaters. Therefore it is efficient to perform these two processes together.

Averaged word2vec is used to perform ranked retrieval. This method uses a word2vec model which is an artificial neural network trained on large corpus of words. From hidden layer of such network, a unique vector for each word can be extracted. These vectors have the semantics of the corresponding word embedded in

them. By obtaining vectors of all words in an article and averaging them, it is possible to compute a representation of the article in form of single vector.

Compared to other methods of ranked retrieval such as vector space model, this method is faster and requires less storage because the length of the vector is fixed. On the other hand, vector space model adds one element to the vectors for every unique word and therefore the length of vectors increases as more articles are collected. The code used for computing the article vector is showed in figure 10.

```
1. word2vec = gensim.models.KeyedVectors.load_word2vec_format('GoogleNews-vectors-
   negative300.bin.gz', binary=True)
2. def average_vectors(word2vec_model, doc):
3.     # remove out-of-vocabulary words
4.     doc = [word for word in doc if word in word2vec_model.vocab]
5.     if len(doc) == 0:
6.       return np.zeros(300)
7.     else:
8.       return np.mean(word2vec_model[doc], axis=0)
```

Figure 10 Computing the averaged word2vec vector for single article

For obtaining the word vectors, a pretrained word2vec model from Google was used which embeds the words in vectors with length 300. Gensim package was used for loading the pretrained model (line 1). A for loop is run over all words in the article and their vectors are appended to the "doc" list (line 4). In case that no vectors were extracted, i.e. none of the words in the article were present in the training dataset for the word2vec, a vector with zeros is returned (lines 5-6). Otherwise, the mean of all vectors is computed which is the vector representing the article (lines 7-8). This approach is then repeated for all documents.

The vectors of all articles are then stored in a csv file together with the corresponding article ID. As mentioned the computation of article vectors is done after the index updating. Therefore, the functions for computing the vectors are wrapped together with the index updating functions in one function, "update_index_vecs" which was previously showed in figure 4 (line 10). Definition of this function is showed in figure 11.

```
1. def update_index_vecs (df, index_path="data/index.json", vec_path="data/doc_vecs.cs
   v", first_time=False):
2.     if df is None:
3.         print("Nothing to update")
4.     else:
5.         to_add = transform_df(df)
6.         build_index(transformed_df=to_add, index_path=index_path, first_time=first_
   time)
7.         get_vectors_all(transformed_df=to_add, vec_path=vec_path, first_time=first_
   time)
```

Figure 11 Wrapper for updating index and computing article vectors

The function takes the output of the crawler, i.e. the newly crawled articles. In case, that no new articles were found, the function simply does not do anything and reports that no changes were made (lines 2-3). If there is at least one new article the code first performs text pre-processing and merges the title and summary to one text (line 5). Then the index is updated as described earlier (line 6) and saved in the "index_path". Finally, vectors of all articles are computed and saved in the "vec_path". There is again an parameter for cases when there is no index nor vector file to update yet, when the first_time parameter is set to true, both of these files are created.

1.3 Query processor