Simple Search Engine

Friday, 18.12.2020

Prahitha Movva
S20180010108

Contents

Abstract

The objective of this project is to implement a simple search engine using our learnings from the course. This report discusses how I approached the data and the different methods used during the ranking and querying. I have also added a spell checker, which checks if the query entered by the user has any spelling errors and then corrects them before returning the result.

Keywords: search engine, TF-IDF, matching score, spell correction, preprocessing, tkinter

Why TF-IDF?

In the typical case, we have many more documents in our corpus than labeled documents. That means the IDF can be calculated much more accurately and completely when using the whole corpus.

Next, considering the case (our case) where the corpus we can get our hands on is "big enough". In this case, the number of iterations needed for training could possibly be smaller when using TF-IDF because the learning algorithm wouldn't need to learn as much.

Finally, in this same case, we could also provide TF only, or TF and IDF separately (or even include TF-IDF as well). I decided that this could potentially generate better results, for example, when using a sophisticated kernel function. So I went ahead and implemented the TF-IDF for the search engine.

Data collection and preprocessing

The dataset (100k+ reviews) which I've taken is from Kaggle and is called the "Amazon fine food reviews". After downloading it and reading the CSV using pandas, I've removed the columns that weren't adding much value. Hence I've dropped all the columns except 'summary', 'text', and 'score'.

During the preprocessing phase, I've split the words into tokens and converted them all to lowercase. Furthermore, I have also removed the stop words and performed stemming using the Porter Stemmer.

TF-IDF values calculation

Now that I have clean data, I moved on with calculating the DF and TF-IDF values.

For calculating the DF values, I used a dictionary where the word is the key and the set of document values is the value. Since we don't necessarily need all the list of docs, I replaced it with its count.

To calculate the TF-IDF, I've used another dictionary with (document number, token) pair as key and any TF-IDF score as the value. I then iterate through all the documents using Counter (which gives the frequency of tokens), calculate TF and IDF, and finally store as a (doc number, token) pair.

```
tokens_body = preprocessing(review)

tokens_title = preprocessing(titles[0])

counter = Counter(tokens_body + tokens_title)

for token in np.unique(tokens_body):

        unique_body = len(np.unique(tokens_body))

        unique_title = len(np.unique(tokens_title))

        tf = counter[token]/(unique_body + unique_title)

        df = DF[token]
```

```
        idf = np.log(docsize/(df+1))
        tf_idf[fileno, token] = tf*idf
```

Finally, I've pickled this result of TF-IDF values in a file called "tf_idf_scores.pkl". This helps in directly querying without having to create the index and calculating the scores every time the code is run.

Document retrieval using TF-IDF matching score and spell correction

For example, if the query is "dog food", I've preprocessed the query, following the previously mentioned method. I have also checked for spelling errors in the query using Levenshtein edit distance and basic probability. For example, if the query is "dogg food", it still returns the same results as "dog food".

Then, checking in every document if these words exist. If the word exists, then the TF-IDF value is added to the matching score of that particular docID. Finally, I've sorted them based on the TF-IDF score and return the result to the user.

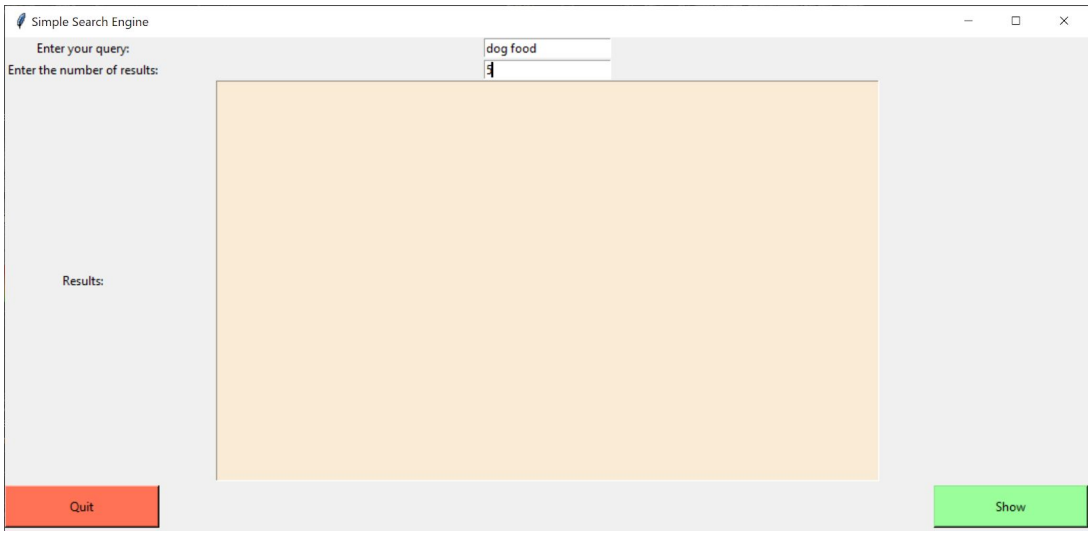
```
def matching_score(query):
    tokens = preprocessing(query)
    query_weights = {}
    for key in tf_idf:
        if key[1] in tokens:
            try:
                query_weights[key[0]] += tf_idf[key]
            except:
                query_weights[key[0]] = tf_idf[key]

    return query_weights

def initiate_search(k, query):
    query_words = query.split(' ')
```

```
    final_string = " "


    for word in query_words:
        final_string = final_string + ' ' + correction(word)


    results = matching_score(final_string)
    results.update((x, y*score[x]) for x, y in results.items())
    results = (sorted(results.items(), key=lambda item: item[1],
reverse=True))
    result = results[:k]
    return result
```

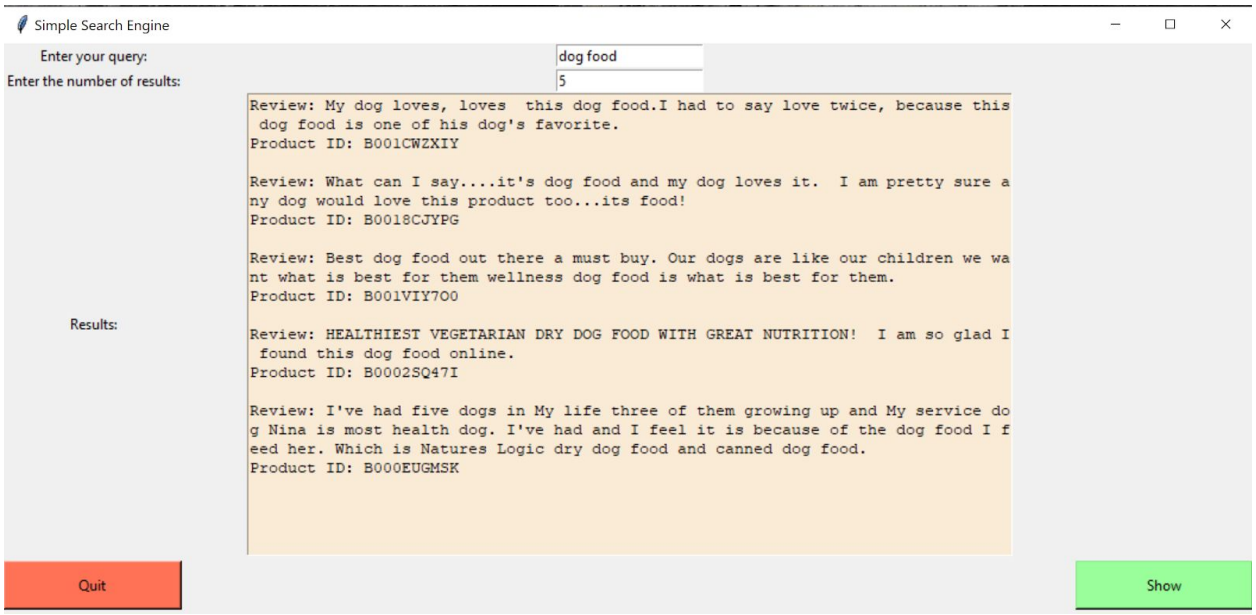
Adding UI

I've used Tkinter by Python for adding UI to the project. Below are the images attached.

Entering a query:



Result for the query:



Console result and time for querying:

```
Time taken to return query results: 3.140625
My dog loves, loves  this dog food.I had to say love twice, because this dog food is o
ne of his dog's favorite.
B001CWZXIY


What can I say....it's dog food and my dog loves it.  I am pretty sure any dog would l
ove this product too...its food!
B0018CJYPG


Best dog food out there a must buy. Our dogs are like our children we want what is bes
t for them wellness dog food is what is best for them.
B001VIY7O0


HEALTHIEST VEGETARIAN DRY DOG FOOD WITH GREAT NUTRITION!  I am so glad I found this dog food online.
B0002SQ47I


I've had five dogs in My life three of them growing up and My service dog Nina is most health dog. I've had and I feel it is because of the dog food I feed her. Which is Natures Logic dry
dog food and canned dog food.
B000EUGMSK
```