TED-Recommender
Web Application

<u>Search feature:</u>

The first feature of the web application is to allow user to search for their favorite TED-Talks using query relate to any context of the video, such as video title, author, or the script. To complete this task, I implemented rank search with term frequency and inverse document frequency weight assign to each word appear in the dataset.

1. Back-end algorithms:

Back-end logic of the whole application is run on ted_engine.py file, which built using python 3.6. Algorithm is implemented base on the instruction on https://nbviewer.jupyter.org/url/crystal.uta.edu/~cli/cse5334/ipythonnotebook/P1.ipynb.

Preprocessing Data:

I begin the app by retrieving the data from the dataset using pandas library.

```
import pandas as pd
import numpy
from nltk.tokenize import RegexpTokenizer
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer

class ted_engine:
        # Needed attributes
        tedData = pd.read_csv('ted_data.csv')
        tokenizer = RegexpTokenizer(r'[a-zA-Z]+')
        stops = stopwords.words('english')
        stemmer = PorterStemmer()
        total_words = []
        final_document = []
        weight_vectors = []
        posting_lists = {}

        def __init__(self):
                tedData = self.tedData
                tokenizer = self.tokenizer
                stops = self.stops
                stemmer = self.stemmer
                total_words = self.total_words
                final_document = self.final_document
                weight_vectors = self.weight_vectors
                posting_lists = self.posting_lists

                for i in range(len(tedData)):
```

After reading the data, I try to retrieved the heading I need such as title, author name, description, and script.

For each read data, I process **Tokenize** to get all the words in separate. Next step is to eliminate all the common words by using nltk library and remove all the words appear in **Stop words** list. Another important step for preprocessing the data is to **Stemming** the words. So what I do is shorten all the words, so the same words with different time clause or plural clause, it would be treated the same. All the words are then store in a **Bag of Words** for each document, which is an array.

Process:

The purpose of this step is to construct posting-lists of all the words appear in the dataset. Each word will have their weight with the corresponding document.

For **TF-IDF** weight of each word, for each document, count the time each word appears using a hash table and give the count to their value. Next thing is to count how many documents contain the word. The weight of the word would be given by:

$$weight = tf \times \log(\frac{n}{df})$$

where tf is the frequency of each word.

n is the number of documents in the dataset

df is the number of documents contain the words

```
for document in final_document:
        weight_vector = {}
        for term in document:
                if term not in weight_vector:
                        tf = document.count(term)/len(document)
                        df = sum(1 for document in final_document if term in document)
                        n = len(final_document)
                        '''
                        idf = math.log(len(final_document) / (1 + containing))
                        '''
                        weight = tf * math.log(n/df)
                        weight_vector[term] = weight

        weight_vectors.append(weight_vector)

# construct posting lists
for i in range(len(weight_vectors)):
        document = weight_vectors[i]
        for token in document:
                if token not in posting_lists:
                        posting_lists[token] = []
                posting_lists[token].append([i, document[token]])
                posting_lists[token] = sorted(posting_lists[token], key=lambda x: x[1], reverse=True)
```

After we have **TF-IDF** weight for each word, we construct a posting-lists for all the word to store their weight correspond with the documents.

Search:
For the given query, we can go in the posting list and look for each term in the query.

```
def search(self, query):
        q = self.tokenizer.tokenize(query)
        tokens = []
        query_weight = {}
        for t in q:
                t = t.lower()
                if t not in self.stops:
                        t = self.stemmer.stem(t)
                        tokens.append(t)

        for term in tokens:
                if term not in query_weight:
                        tf = tokens.count(term) / len(tokens)
                        query_weight[term] = tf

        sim = {}
        for term in query_weight:
                if term in self.posting_lists:
                        for post in self.posting_lists[term]:
                                document = post[0]
                                if document not in sim:
                                        sim[document] = 0
                                sim[document] += post[1] * query_weight[term]
        sim = sorted(sim, key=sim.get, reverse=True)
```

The similarity point for each term in the query to a document would be the weight of the term in the document multiply the weight of the term in the query. The next important thing is to **Sort** the similarity array to retrieve the most related documents.

Front-end:
For the front end I use a free template on https://bootstrapmade.com/ and modify it to fit my purpose.

Hosting:

Hosting was not really a problem using Digital Ocean with a free $100 credit given. I can easily pull my source code from Github to the Linux server, run the code and use its DNS to connect to them.