OneMinuteUSLawSE Report

Update on 06/26/2020

These past two weeks, I have completed the whole backend of this search website. In addition to the search function, this website also provides the automatic update function which enables non-technical staff to update the data much easier. I will explain some important details of the implementation of this website in the following.

Gevent

In order to improve the performance of our web application, I decided to implement an asynchronous and non-blocking server for our web application. For the asynchronization, I chose the WSGIServer of Gevent library to be the server of our application because Gevent is a library for scalable asynchronous I/O with a fully synchronous programming model. And in order to implement non-blocking feature, I used the monkey patch to replace functions and classes in the standard socket module with their cooperative counterparts in Gevent library.

apscheduler

In order to enable the data in the application to be updated automatically, I imported an apscheuler in another thread to scan the dataset every 24 hours. Once it finds out the data in the datset has been updated, the search model will be retrained.

Update on 06/02/2020

Because the performance of Baidu Speech Recognition API is unsatisfactory, I decided to try another speech recognition API launched by Xunfei.
The comparison between these two API is as follows:
*

	Baidu API	Xunfei API
Performance	Bad. The text extracted from videos by Baidu API is not accurate, and Baidu API is not able to handle English videos well.	Good. Compared with Baidu API, Xunfei API can extract the text from videos much more accurate, moreover, it is able to handle English videos very well.
Complexity	Easy. Baidu API is encapsulated very well, so it is really simple to use it.	Hard. Xunfei web API is implemented by Websocket. The operation is much more complex.
Robustness	Strong. The server of Baidu is really robust. The connection between us to its server is reliable.	Weak. The performance of Xunfei's server is really bad. Maybe we are located in United States while its servers are in China, we have to connect to their servers by VPN.

Although there are some shortcomings of Xunfei API, I have found methods to overcome these shortcomings. Hence, I prefer to use Xunfei API in our project in the future because of its excellent accuracy.

Update on 06/01/2020

In order to increase the quality and efficiency of extracting data from videos, I implemented an automatic speech recognition system. Basically, this system can extract what lawyers say in videos and then output it to a text. The main part of this system is implemented by calling the Baidu Speech Recognition API. The working process of this system is as follow:

a. Apply youtube_dl and ffmpeg library of Python to download videos from Youtube and then extract the audio from videos.

b. Apply ffmpeg to adjust the format of the audio. The specific required format includes: Waveform audio, 16 kHz rate, Mono audio channel.

c. Since the standard version of Baidu Speech Recognition, which is free, only accepts audio less than one mintes for a single submission. I have to use pydub library to split each audio into multiple segments.

d. Package the audio and relative parameters into a JSON format and POST it to Baidu AI Platform.

e. After getting the result texts from Baidu AI Platform, automatically merge the texts according to their corresponding videos.

Although this system can extract what lawyers say in videos and output it to texts, the accuracy is not 100% correct. And there is no system is able to produce 100% accurate recognition. Thus, after getting these texts, we still need to check them manually. However, I think this system has increased the efficiency of extracting data from videos.

Update on 05/28/2020

Abstraction

These two weeks, I have applied Flask to implement the search web based on baseline model(inverted index model) and deployed it online. When users search on this search web, the server will return the title and link of the relative videos to them. And, this web also includes the automatic update function. Once we have a new video and relative information of it, we just need to add the information into the dataset and the server will update the index files automatically.

Current Problems

However, there are still some problems with this search web system. These problems are listed as follows:

1. Data Missing. Currently, we have about 160 videos in our dataset, but almost half of them are lack of the content of description. For those videos without the content of description, the search engine can only use the title of these videos to construct the index files. This will decrease the accuracy when users use our web to do search.

2. Mixed Language. A lot of description documents in our dataset are mixed with Chinese and English. What's more, there are a lot of synonyms in Chinese. When the users try to search by some words that do not exit in our data set, our model is not able to return the relatives videos to them. This is also a problem that will influence the performance of our search engine negatively.

Potential Solutions

According to these two problems, I come up with the following solutions:

1. For the missing data problem, we can only supplement the descriptions part of our dataset. Without the description documents, these videos are not able to be indexed properly by our model.

2. For the mixed language problem, we can try to use the vector space search model to replace our baseline model. If we want to use the vector space model to replace the baseline model, I still need to find some corpus which are mixed with Chinese and English to train the vector space model further.
3. I think there is another solution that may be helpful for the mixed language problems. When we preprocess the text of the data set and the query entered by users, we can try to add one more step in it -- translation. I can try to find a method to translate all the English words to Chinese words before indexing and searching. However, this method is not able to handle the synonyms in our dataset.

Schedule

1. Contact with the relative colleagues, and ask them to help supplement the missing part of the dataset.
2. Try to find more corpus which are mixed by English and Chinese to retrain the vector space model.
3. After the dataset being supplemented, I believe that we need to arrange a series comparative test. The first test is the comparison between the baseline model before and after the supplement. And the second test should be held beween the baseline model and the vector space model.
4. Ask some friends of mine to use our search web. And then gather their feedback to improve both of the functions and contents of our web.

Update on 05/10/2020

According to the assumption of the last experiment result, I generate another groups of query statements to test the performance of the different models. In this new experiment, the result of the weighted Word2Vec model outperformed the baseline model by increasing 8% of the NDCG. This result proved my assumption – when the search query is not the totally same to the title of our videos, the vector space model is able to capture the semantic meaning of the query better.

Update on 05/02/2020

Abstraction

This week, I have designed and implemented four different search engine models for this project. According to their different ways to complete the function of search, I classify them into two types - traditional model and vector space model. Among the four models I implemented, inverted-index model and TFIDF model belong to traditional model, while the non-weighted Word2Vec model and weighted Word2Vec model belong to vector space model.

Data

The training data I used to train traditional models and vector space models is the same. It includes 45 short paragraphs related to immigration law. The link for training data: description_doc.csv.
The test data includes nine query statements and I also the relevance score between each query statement and each searchable texts. However, these nine query statements are all extracted from the titles of searchable texts directly. The link for testing data: relevance.csv.

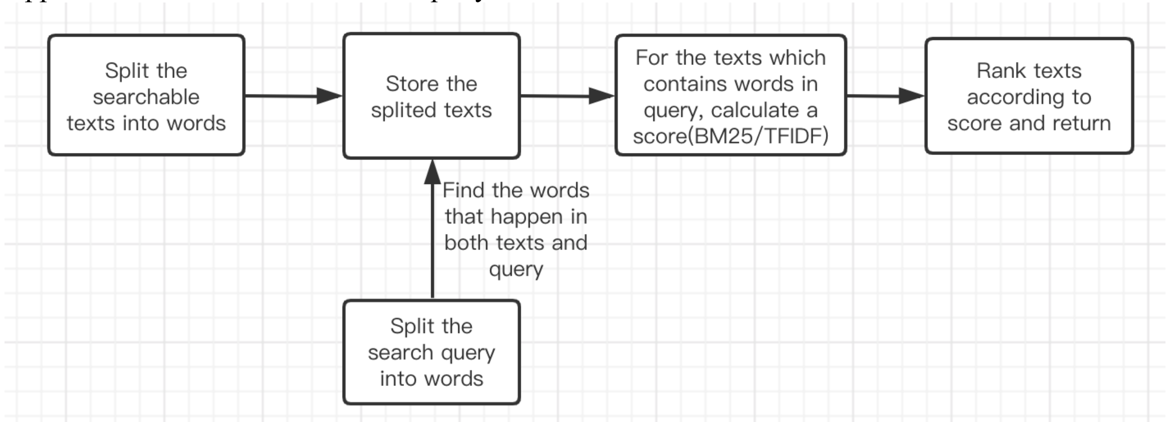
Measuring Metric

I think the searchable texts with higher relevance to the query statement entered by users should be returned earlier to users. Thus, when I test these search engines, I need to consider the rank of search results.
According to my research of ranking-aware metrics, I picked three candidates for this project -

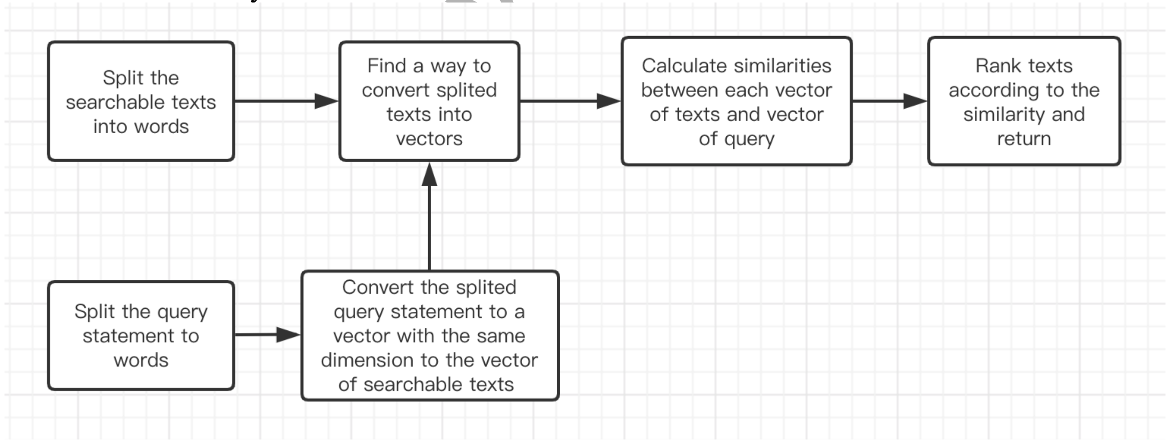
MRR(Mean Reciprocal Rank), MAP(Mean Average Precision) and NDCG(Normalized Discount Cumulative Gain).
Among these three metrics, MRR is the most simple one because it just considers the rank of the first relevant item in the result list. And MAP applies sliding window to calculate the average precision for each query statement but it ignores the magnitude of relevance. NDCG is most complicated metric in these three, it not only considers every item in the result list but also the magnitude of relevance to query statements of them.
Hence, I think NDCG is the most suitable metric for this project because I want the more relevant item to be shown earlier to users when they search on our web.

Principle of Traditional Models and Vector Space Models

The principle of the traditional model is very straightforward. The essence of it is to find the words that happen in both searchable texts and query statement. The workflow of it is as follow:

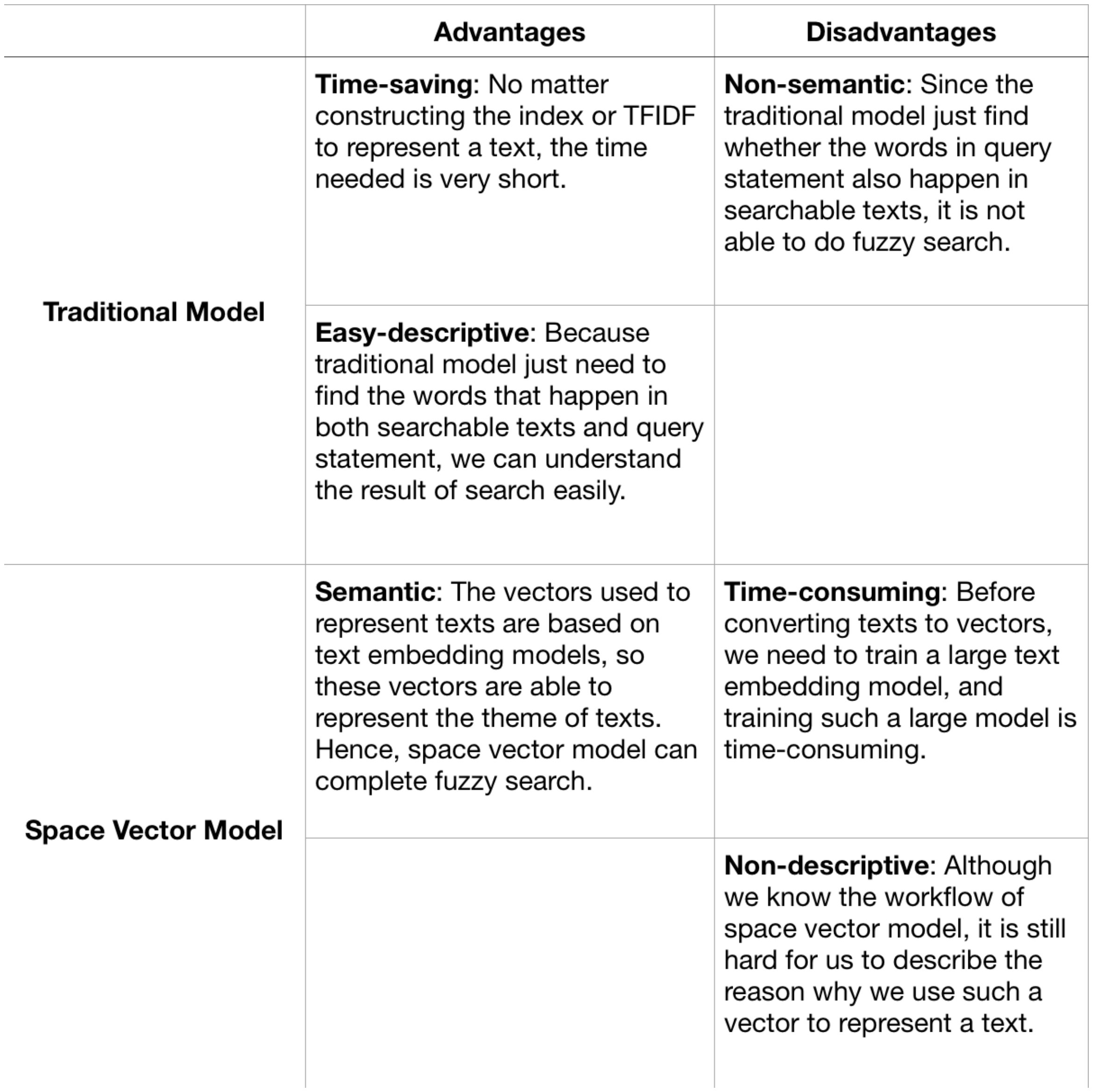

Split the searchable texts into words
Store the splited texts
For the texts which contains words in query, calculate a score(BM25/TFIDF)
Rank texts according to score and return
Find the words that happen in both texts and query
Split the search query into words

Compared with the traditional model, the principle of the vector space model is more complicated. The essence of it is to find a way to represent the theme of searchable texts and query statement, and then calculate the similarity between them. The workflow of it is as follow:


Split the searchable texts into words
Find a way to convert splited texts into vectors
Calculate similarities between each vector of texts and vector of query
Rank texts according to the similarity and return
Split the query statement to words
Convert the splited query statement to a vector with the same dimension to the vector of searchable texts

Advantages and Disadvantages of Traditional Models and Vector Space Models

Based on their different ways to complete the function of search, the advantages and disadvantages of them are obvious. I conclude the advantages and disadvantages of these two types of models as follow:



	Advantages	Disadvantages
Traditional Model	**Time-saving**: No matter constructing the index or TFIDF to represent a text, the time needed is very short.	**Non-semantic**: Since the traditional model just find whether the words in query statement also happen in searchable texts, it is not able to do fuzzy search.
	Easy-descriptive: Because traditional model just need to find the words that happen in both searchable texts and query statement, we can understand the result of search easily.	
Space Vector Model	**Semantic**: The vectors used to represent texts are based on text embedding models, so these vectors are able to represent the theme of texts. Hence, space vector model can complete fuzzy search.	**Time-consuming**: Before converting texts to vectors, we need to train a large text embedding model, and training such a large model is time-consuming.
		Non-descriptive: Although we know the workflow of space vector model, it is still hard for us to describe the reason why we use such a vector to represent a text.

Here, I want to talk about the point of 'semantic'. The reason why traditional models are not able to do fuzzy search is that they just find out whether the words in query statement also happen in searchable texts and they don't 'understand' what these texts talk about. However, when the space vector models convert the searchable texts to vectors by text embedding models, such as Word2Vec or Doc2vec, they have 'memorized' what these texts talk about in the vectors.
For example, if a text contains the word 'salary', and we use the word 'wage' to search. Traditional models are not able to return this text to us because the word 'wage' does not happen in this text, though the word 'salary' holds the same meaning. However, Space Vector model may return this text to us because they understand these two words hold a very closed meaning.

Test of Traditional Models and Vector Space Models

The metric that I selected to test these models is NDCG. In order to apply NDCG test the models that I have implemented, I generated nine query statements, and scored the relevance between each of these query statements and each searchable texts. What I need to notice here is that all of these nine query statements are extracted from the titles of nine of searchable texts directly. This means all the words in these query statements are also contained by the searchable texts. The results of the test are as follow:

		NDCG for Query 1	NDCG for Query 2	NDCG for Query 3	NDCG for Query 4	NDCG for Query 5	NDCG for Query 6	NDCG for Query 7	NDCG for Query 8	NDCG for Query 9	Average NDCG
Traditional Models	Inverted-Index Model	0.872	0.866	0.903	0.804	0.888	1.00	0.865	0.927	0.900	0.892
	TF-IDF Model	0.852	0.843	0.935	0.779	0.891	1.00	0.850	0.928	0.905	0.887
Vector Space Models	Non-weighted Word2Vec	0.811	0.611	0.853	0.566	0.922	1.00	0.800	0.858	0.865	0.809
	Weighted Word2Vec	0.819	0.567	0.874	0.702	0.885	1.00	0.780	0.857	0.849	0.815

Obviously, the performance of traditional models is better than that of the vector space models in this test. In my own opinion, there are two reasons causing this result, and I will talk about these two reasons in the next section.

Assumption based on the Test Results

Firstly, I applied transfer learning on the training of Word2Vec model which is used to convert the searchable texts and query statements to vectors. However, the corpus of this pre-trained Word2Vec model is the 2016 Chinese Wikipedia, which does not contain many words about immigration law. What's more, the corpus used to retrain the model only contains 45 short paragraphs about immigration law. Although I have used all of them to retrain the pre-trained model, I think the information about immigration law learned by this model is still not enough.
Secondly, all the query statements I used in the test were extracted from the titles of searchable texts directly. This means all the words in query statements also contained by searchable text. And as I talked above, traditional models complete search by finding out the words happening in both searchable texts and query statements. Hence, this test has a problem similar to data leak (the training data are also test data), and this caused the traditional models outperformed.

Future Work

In order to have a more comprehensive test and a more fair result, I need to do several things in the future work:

a. Collect more immigration law related paragraphs to continuously train the Word2Vec model. I believe that if I can retrain the pre-trained Word2Vec model by more immigration law related paragraphs, the performance of it would be better.
b. Regenerate a list of query statements for test. This time, I will use some words which are not contained by the searchable texts to generate the query statements. By this way, the test can avoid the data leak problem.

Conclusion

According to my experience, it is impossible for users to enter the words that are all also contained by searchable texts every time. Hence, to continuously train the Word2Vec model and run another test is very necessary for the future application of this search engine in real life. In the new test, I assume the performance of the vector space models will be improved while the performance of the traditional models will be dragged down.

Update on 04/22/2020

• Vectorization

Although the baseline model is able to complete the task of search, I want to import some vectorization techniques to improve the performance of the search engine. Currently, most the traditional search engines are still implemented by inverted index, but they require that the word appears in query also appears in the documents. This means that these models do not understand what the users want to search for, so they just find every word that appears in the query and documents at the same time. However, vector models can understand what the documents describe and what the users want to search for in a way.

• TF-IDF

TF-IDF is the most basic vector model. TF-IDFalgorithm is by far the dominant way of weighting co-occurrence matrices innatural language processing, especially in information retrieval. TF-IDF weight is computed as the product of the term frequency and the inverse document frequency. The steps to compute TF-IDF are as follows:

a. TF (Term Frequency)
The number of times a word appears in a document divdedby the total number of words in the document. Every document has its own termfrequency.

b. IDF (Inverse Document Frequency)
The log of the number of documents divided by the number of documents that contain the word. Inverse document frequency determines the weight of rare words across all documents in the corpus.

c. TF-IDF
Lastly, we can calculate the TF-IDF by multiplying TF and IDF.

d. Cosine Similarity
After converting the documents and query to vectors represented by TF-IDF, we can calculate the cosine similarity between the vector of query and each vector generated by documents to rank the relevance.

Advantages of TF-IDF:

a. Easy to understand.

b. Time-efficient for converting the documents to vectors.

Disadvantages of TF-IDF:

a. The length of vectors generated by TF-IDF is fixed. Take the documents we currently have as an example, there are around 3750 words total in these documents. Hence, we need to convert all the documents and the query entered by users to vectors with length of 3750. When we get a larger dataset, the length of vectors will increase much larger too. Hence, the vector model based on TF-IDF is really space consuming.

b. The essence of TF-IDF vector model is similar to the traditional inverted index search engine. In fact, TF-IDF weight is also very import for the inverted index search engine. For example, the inverted index search engine implemented by me applies the BM25 score to rank the result of search, and the TF-IDF weight is the essential point of the BM25 score.

• Word2Vec

Word2Vec is a word embedding technique which can convert a word into a vector based on the context around the word. The Word2Vec technique is based on a feed-forward, fully connected neural network. There are two types of implementations of Word2Vec -- CBOW and Skip-gram. These two implementations are very similar. The only difference between them is that CBOW predict the central word by context while Skip-gram predict the context by the central word. I will take Skip-gram as an example to explain the working flow of Word2Vec.

a. All the words in corpus will be converted into a vector by one-hot encoding first. If there are v words total in the corpus, the dimension of the vector is [1, |v|].

b. The word w is passed to the hidden layer from |v| neurons.

c. Hidden layer performsthe dot product between weight matrix W[|v|, N] and the input vector w. And we will get an output vector, H[1, N] from this operation.

d. There is no activation function used in hidden layer, so the H[1,N]will be passed directly to the output layer. Output layer will apply dot product between H[1, N] and another matrix W'[N, |v|] , and this operation will give us the vector U[1, |v|]. The dimension of output vector is totally the same as the input vector.

e. Right now, we need to get the probability of each word in the output vector, so we apply softmax activation function compute it. Finally, we need to do back propagation to update the weight matrix W and W'. The well-updated weight matrix W is the final product of this model becuase it is a representation of all the words in this corpus.

Right now, we have known how to convert a word to vector by Word2Vec, but it is not enough because what a search engine focuses on is the similarity between the documents we have and the query entered by users. We need to define two functions to convert these documents and query to a same vector space. Obviously, each document and query are composed by a sequence of words, so this problem can be transferred to how to vectorize the sequence of words with different length. The solution to this problem is very easy. We can convert each document and query to a sequence of word vector by Word2Vec first, and then use TF-IDF of each word as the weight to get weighted average vector of each document and query. After doing this, we can calculate the similarity between them.

Advantages:

a. Space-efficicient. Word2vec in fact is a very practical dimensionality reduction technique. We can vectorize each word to the same dimension first, and then get their weighted average to represent each document.

b. Semantic vector. Compared with pure TF-IDF, the vectors generated by Word2Vec are more semantic. Because it not only considers if a word happens in a text, but also considers the context of this word. Hence, I think the vector generated by Word2Vec is able to represent a text better.

Disadvantages：

a. When the text is very long, the simple weighted average of word vectors is not able to represent this text very well. This is because the ability of it to identify which part in a text more import is not strong enough.

b. Word2Vec model loses all information about word order.

Fortunately, all the text documents in our project right now are very short. Thus, I believe using the weighted average of word vectors generated by Word2Vec to represent the text documents in our project is a wise choice.

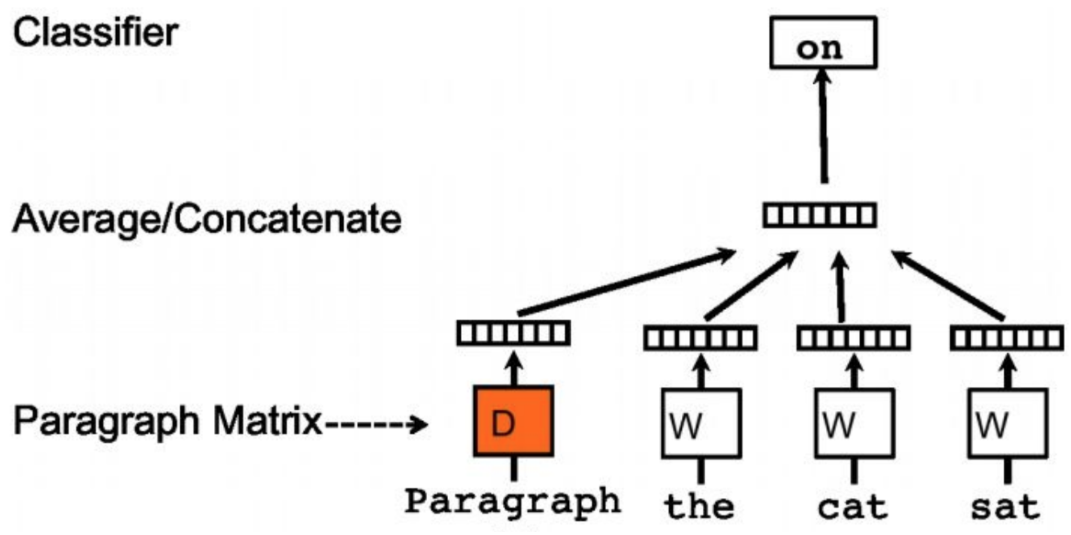
• Doc2Vec(Paragraph Vector)

Doc2Vec model is constructed based on Word2Vec model. Just as what I said before, Word2Vec model ignore the order of words in a sentence, so Doc2Vec fix this weakness by importing one more vector (paragraph vector) during the training. There are two types of Doc2Vec mpdel -- PV-DM(Distributed Memory Model of paragraph vectors) and PV-DBOW(Distributed Bag of Words of paragraph vector). In fact, PV-DM is really similar to the CBOW model in Word2Vec while PV-DBOW is similar to the Skip-gram model in Word2Vec. Here, I take PV-DM as an example to illustrate the work process of Doc2Vec.

a. Just like CBOW model of Word2Vec, PV-DM also uses the context words to predict a central word. However, PV-DM imports one more vector(paragraph vector) in the input layer. This paragraph vector is used to represent a sentence.

b. And then, we need to do the dot product for all of these vectors (word vectors in a window and corresponding paragraph vector) with the weighted matrix. After the dot production, we average all the output vectors to get a vector which is the output for the hidden layer.

c. What we need to notice here is that the paragraph vector of a sentence is shared for every iteration of train within this sentence. This means that this paragraph vector can represent this sentence more and more accurately as the more iterations of training happen.



Advantages:

a. Compared with Word2Vec model, Doc2Vec learns the information of order of words.

Disadvantages:

a. For short text, Doc2Vec performs may not be better than Word2Vec.

b. To train a well-performed Doc2Vec model, we need a large amount of data(documents/text). However, the amount of documents we have right now is too little, so if we want to apply this technique for our project, we need to collect much more data.

Reference:
https://janav.wordpress.com/2013/10/27/tf-idf-and-cosine-similarity/
https://iksinc.online/tag/skip-gram-model/
https://arxiv.org/pdf/1607.00570.pdf
https://radimrehurek.com/gensim/models/doc2vec.html
https://cs.stanford.edu/~quocle/paragraph_vector.pdf

Update on 04/20/2020

• Baseline model based on Inverted Index

I created an inverted index search engine as the baseline model of this project. This search engine is implemented by the Python search engine framework -- Whoosh. The specific steps are as follows:

a. Defined some methods to preprocess the description documents, such as removing all the dash and slash in the documents, and converting all upper case letters to lower case letters.

b. Applied the Jieba Chinese Analyzer to tokenize all the well-preprocessed description documents, and then built the inverted index by Whoosh. After the construction of the inverted index, I stored the index in local, so we can reuse the index when we do search later.

c. Applied Whoosh to load the index that had been constructed, and created a searcher object. Then, I defined every aspect of search in this step, such as the field can be searched, and the score used to rank the search results.

d. Preprocessed the query input by users in the same way as I preprocessed the documents, such as removing all dash and slash in the query, and converting all upper case letters to lower case letters. Then, I tokenize the query by Jieba Chinese Analyzer. Finally, we can use the well-processed query to complete search in the indexed files.

The ranking function that I used to rank the results of search in this baseline model is **BM25**. It is calculated by this formula:

$$Score(Q,d) = \sum_{i}^{n} IDF(q_i) \cdot \frac{f_i \cdot (k_1 + 1)}{f_i + k_1 \cdot (1 - b + b \cdot \frac{dl}{avgdl})}$$

Where the Q means the set of words in a search query and the d refers to a related document. IDF refers to inverse document frequency while the fi refers the frequency of the ith word of query in this document. Compared with the traditional TF-IDF score, BM25 is more flexible and robust because it imports several adjustable parameters -- k1 and b.

Reference:
https://opensourceconnections.com/blog/2015/10/16/bm25-the-next-generation-of-lucene-relevation/

Update on 04/15/2020

• Comparison for different search engine frameworks

1. Whoosh: Whoosh is a full-featured text search engine library written entirely in python, more like Apache Lucene Core , uses Okapi_BM25. It is just a python library, so we have to write API exposing whoosh search. What's more, Whoosh requires that we specify the fields of the index before you begin indexing. However, Whoosh is quite flexible and offers a lot of features when it comes to search.
2. Scout: Scout is a RESTful search server written in Python. The search is powered by SQLite's full-text search extension, and the web application utilizes the Flask framework. We can't specify schema of our index, scout takes only one string to index on which it will execute search. We can index meta data, which can be further used for complex filter queries. In scout we get limited set of options, but is very easy to set up and use, also API is quite simplified for indexing and searching.
3. Slor: Apache Solr is an open source search platform built on a Java library called Lucene. It offers Apache Lucene's search capabilities in a user-friendly way. It offers distributed indexing, replication, load-balanced querying, and automated failover and recovery.
4. Elasticsearch is an open source (Apache 2 license), distributed, a RESTful search engine built on top of the Apache Lucene library. It offers a distributed, multitenant-capable, full-text search engine with an HTTP web interface (REST) and schema-free JSON documents. The official client libraries for Elasticsearch are available in Java, Groovy, PHP, Ruby, Perl, Python, .NET, and Javascript. The distributed search engine includes indices that can be divided into shards, and each shard can have multiple replicas. Each Elasticsearch node can have one or more shards, and its engine also acts as a coordinator to delegate operations to the correct shard(s).
5. Conclusion:
    Firstly, Whoosh and Scout are written by pure Python while Slor and Elasticsearch are written by Java. Although Elasticsearch can also be used in Python project but the configuration will be much more complicated than Whoosh and Scout. Secondly, the most advantage of Elasticsearch is that it is a distributed full-text search engine, but our project doesn't need this function at this time. Thus, I prefer to select the search engine framework between Whoosh and Scout.
    For Whoosh and Scout, I think Whoosh fits our project better. On one hand, Whoosh allows us to define the schema for building the index while Scout dosen't. On the other hand, Scout is not able to handle fuzzy search but Whoosh dose. What's more, Whoosh allows us to import Chinese Tokenizer

to replace its default tokenizer right now, which can make our development easier. And, according to the paper in reference, we can see that to search in around 60,000 indexes, whoosh only takes 0.8 sec, so it is efficient enough to complete the search task for our project. In summary, I prefer to choose Whoosh to be the search engine framework for our project.

Reference:
https://medium.com/@rajatrs5054/scout-vs-whoosh-for-full-text-search-in-python-5f1015591a62
https://logz.io/blog/solr-vs-elasticsearch/
https://zhuanlan.zhihu.com/p/37503363
https://whoosh.readthedocs.io/en/latest/quickstart.html
https://www.jianshu.com/p/fa1d29456d80

• Comparison for different web frameworks

Django vs. Flask: Django provides its own Django ORM (object-relational mapping) and uses data models, while Flask doesn't have any data models at all. Data models allow developers to link database tables with classes in a programming language, so they can work with models in the same way as database references. Django bundles everything together, while Flask is more modular. Main difference between Django and Flask, that Django provides a full-featured Model–View–Controller framework. Its aim is to simplify the process of website development. It relies on less code, reusable components, and rapid development. Flask, on the other hand, is a microframework based on the concept of doing one thing well. It doesn't provide an ORM and comes with only a basic set of tools for web development.

Conclusion:
After the consideration, I think Flask works better than Django for our project. The first reason is that our project is a small project rather than a large project. The second reason is that I am a beginner in web development so it is a little bit complex for me to command and config Django. But the final decision should be made after talking with Zhen Tang

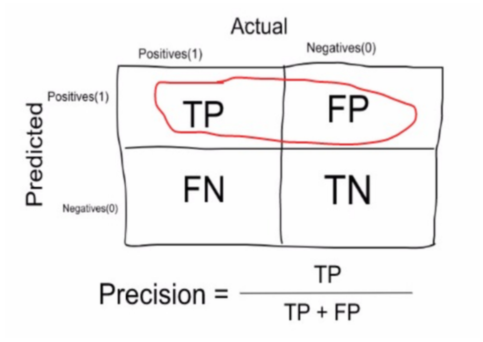
Reference:
https://towardsdatascience.com/django-or-flask-recommendation-from-my-experience-60257b6b7ca6
https://testdriven.io/blog/django-vs-flask/
https://www.zhihu.com/question/41564604

• Comparison of different evaluation metrics

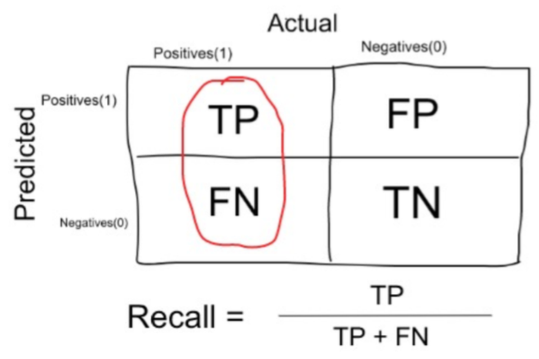
• Based on the research of metrics of evaluating the search engine and recommendation system, I find out that there are two types of metrics -- rank-less metrics and rank-aware metrics. All the metrics I talked before are rank-less metrics, including precision, recall and f1-score. Because they only care about whether we find the relevant items, but they don't care about the rank of the relevant items. If we want the items with higher relevance ranked higher than that with lower relevance, we need to apply the rank-aware metrics to evaluate our system.

Rank-less metrics:

Precision



Recall



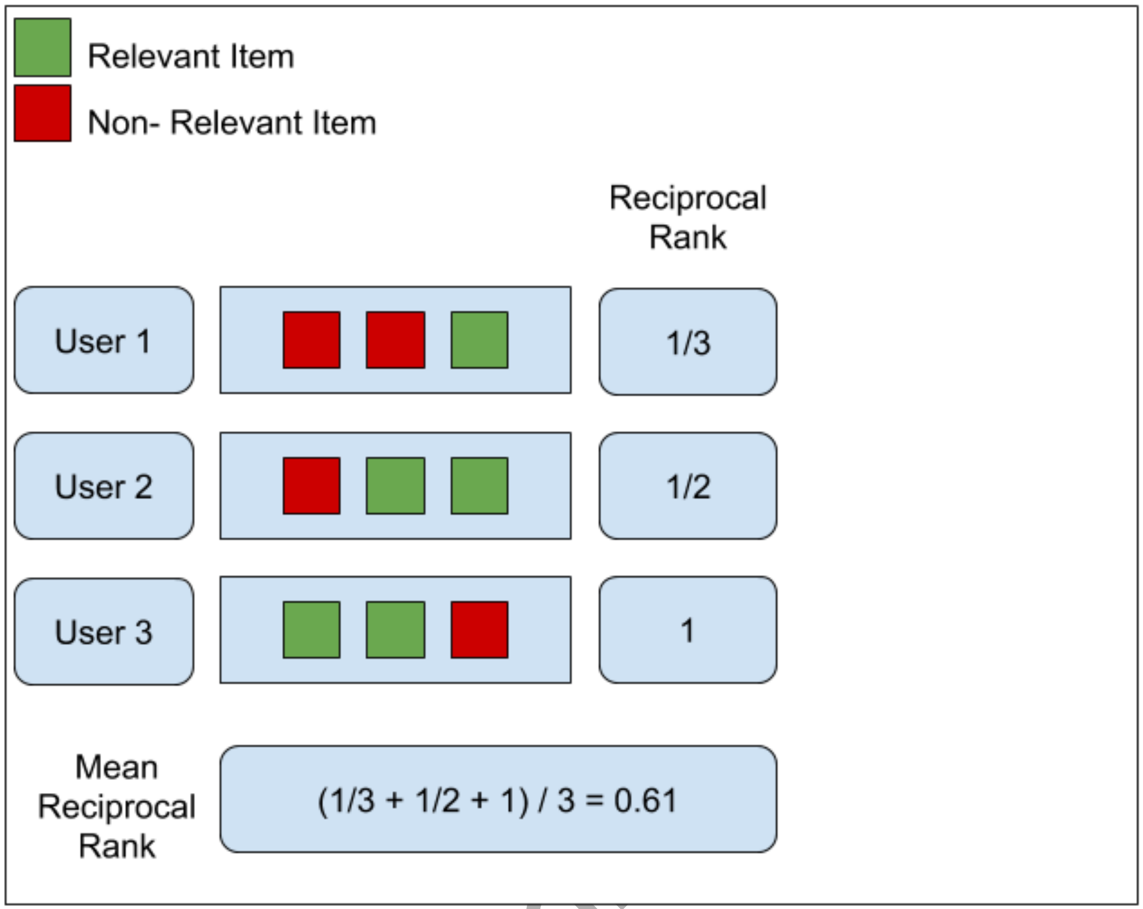
F1 Score is the Harmonic mean of the precision and recall. When x and y are different, Harmonic mean is closer to the smaller number as compared to the larger number.

$$F1 = 2 \times \frac{Precision * Recall}{Precision + Recall}$$

Rank-aware Metrics

- MRR(Mean Reciprocal Rank)

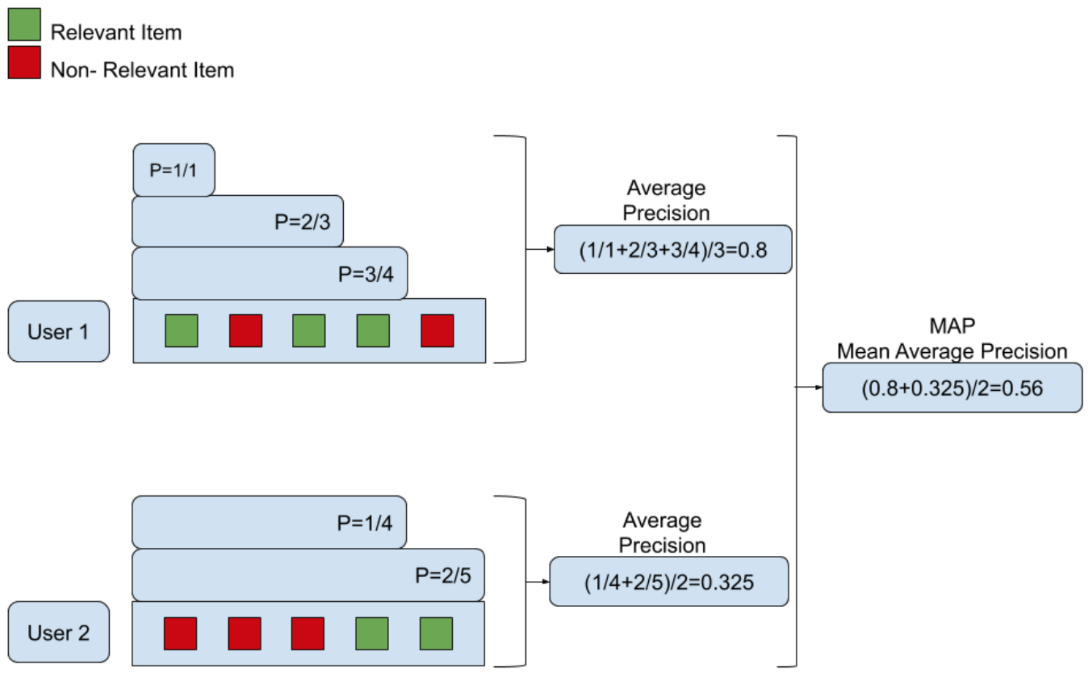
MRR only focuses on finding out the position of the first relevant item. It is a metric of the binary relevance family. The calculation of MRR is very easy. For each search query, we just need to keep track of the position of the first relevant item in the result list. The calculation process as the graph below.



Obviously, the advantage of MRR is that it imports the rank into evaluation and it is very easy to understand. However, it only cares about the position of the first relevant item and ignores the rest of the result list. Because what we want to return to our users is a list of search result, I don't think MRR is not a suitable metric for our project.

- MAP(Mean Average Precision)

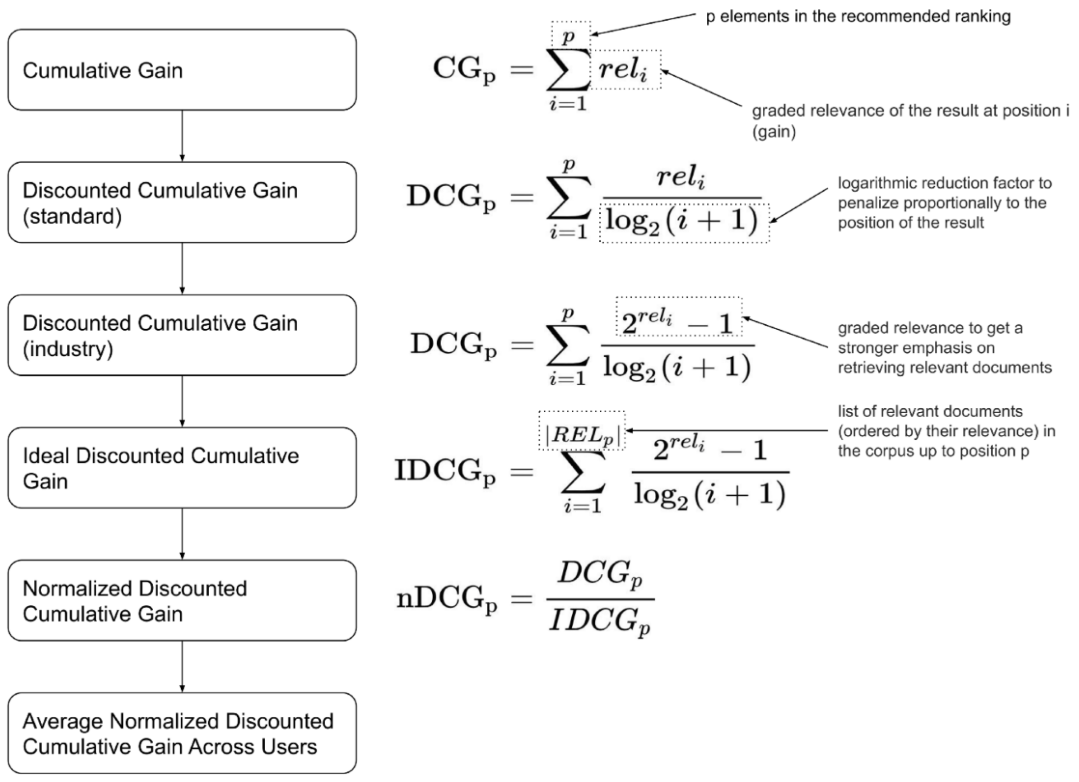
 The calculation of MAP is very similar to precision but it imports the concept of rank. MAP calculate the average percision for each search query like a sliding window. And then sum all the average precision to get the mean of them. It is also a metric of binary-relevant family.
In fact, the AP(average precision) represents the area under the recall-precision plot. I think it is very similar to f1-score because it not only consider precision but also recall.



Obviously, MAP is a version with a rank of precision. If more relevant items are ranked higher in the result list, we will get a better MAP score. This matches the need to show as many relevant items as possible high up the recommended list.
However, MAP can only handle the binary-revelant problem, which means there is no difference among all the relevant items in the result list. It cannot figure out the difference between high-relevant items and low-relevant items.
In our project, I think it is hard to set a magnitude of relevance to each item. Thus, we can treat our project as a binary-revelant problem, and MAP is a good choice for our project.

- NDCG(Normalized Discounted Cumulative Gain)

Compared with MAP, NDCG imports the concept of magnitude of relevance. This means that NDCG can figure out which items are more relevant to the search query. Hence, highly relevant items should come before medium relevant items, which should come before non-relevant items.


Cumulative Gain
Discounted Cumulative Gain (standard)
Discounted Cumulative Gain (industry)
Ideal Discounted Cumulative Gain
Normalized Discounted Cumulative Gain
Average Normalized Discounted Cumulative Gain Across Users
$\mathrm{CG_p} = \sum_{i=1}^{p} rel_i$
p elements in the recommended ranking
graded relevance of the result at position i (gain)
$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)}$
logarithmic reduction factor to penalize proportionally to the position of the result
$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)}$
graded relevance to get a stronger emphasis on retrieving relevant documents
$\mathrm{IDCG_p} = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{\log_2(i+1)}$
list of relevant documents (ordered by their relevance) in the corpus up to position p
$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p}$

As we can see from the graph above, the industrial DCG is very similar to a binary-relevant metric. There are only two circumstances in the industrial DCG —— '0' for non-relevant and '1' for relevant. Thus, I think NDCG is also a good metric for our project.

Reference:
https://medium.com/swlh/rank-aware-recsys-evaluation-metrics-5191bba16832
https://zhuanlan.zhihu.com/p/84206752
https://www.youtube.com/watch?v=pM6DJ0ZZee0
https://www.youtube.com/watch?v=qm1In7NH8WE
https://towardsdatascience.com/evaluate-your-recommendation-engine-using-ndcg-759a851452d1

Created on 04/14/2020

• Purpsoe

• 这个项目主要是要解决 OneMinuteUSLaw 中逐渐增加的视频如何管理的问题。因为视频数量找到逐渐增加，导致用户无法轻易找到他们所想要看的视频。这个项目的想法是为 OneMinuteLaw 添加一个搜索引擎，以此来帮助用户更方便得搜索他们所关注内容相关的视频。

• Algorithm

• 根据调研结果，我发现搜索引擎主要分为两类 —— 全网搜索引擎以及站内搜索引擎。两者的主要区别在于全网搜索引擎的搜索内容是要实时地从全网抓取，而站内搜索引擎的搜索内容全都储

存在自己的网站中。这个项目非常明显的属于后者，因为我们要解决的是对于 OneMinuteLaw 项目已经拍摄好的视频进行搜索，因此所有的搜索内容都是存在于我们站内的。对于站内搜索引擎而言，最重要的部分包括两个：ndex（建立索引系统）和 query（建立查询系统）。

1. 对于搜索引擎的索引，一般而言分为两种：正排索引和倒排索引

正排索引：正排索引的方法我倾向于由我自己手动实现，具体流程为：文本处理（对于很多同一词汇不同写法进行合并）——> 分词（总体上使用中文分词库 jieba，但是对于一些的特定的单词，需要自建词典）——> 去除 stopword ——> 文本向量化（使用一些机器学习的库，例如 sklearn，计算 tfidf 和 ngram，或者做词向量嵌入）——> 把训练得到的模型和特征矩阵进行存储（key 为每个视频的 url）

倒排索引：根据我在网上所阅读的资料，我认为倒排索引的实现可以依赖于 Python 中已有的库 whoosh。whoosh 的优势在于：1. Whoosh 纯由 Python 编写而成，但很快，只需要 Python 环境即可，不需要编译器；2. 默认使用 Okapi BM25F 排序算法，也支持其他排序算法；3. 相比于其他搜索引擎，Whoosh 会创建更小的 index 文件；4. Whoosh 可以储存任意的 Python 对象。我们可以使用中文分词器来替换 whoosh 中默认的分词器，这样我们对就可以对描述文档进行分词，然后将已这些分好的词作为 key 来储存各个视频的 url，以此来建立倒排索引。

2. query：如果我手动实现了正排索引，我将用用户输入的搜索语句生成特征向量（tfidf），然后计算出该特征向量与特征矩阵中代表的各个文本的特征向量计算 cosin 相似度来进行排序并返回。但如果我使用 whoosh 建立起了倒排索引，我将采用 whoosh 中的 api 实现排序搜索，具体实现细节我还需要进一步学习。

3. 因为倒排索引的效率比起正排索引而言更高，我倾向于使用倒排索引来实现我们这个搜索引擎。但是由于我们目前的数据量非常少并且我预计在将来一段时间之内也不会发生巨大的提升，因此手动实现正排索引也足以处理这个搜索引擎所面对的数据量。

• Web Framework

• 在研究完这个搜索引擎的逻辑层实现方法后，我开始研究用户交互的问题。我在网上查阅了 Python 主流的几种 web 框架后，通过对比出他们的优缺点，对 web 框架进行了选择。

1. Django：Django 是 Python 最著名的框架，最大的特点就是大而全。Django 采用了 MVC 的软件设计模式，Django 本身是一个高级别的 PythonWeb 框架，它鼓励快速开发和干净、实用的设计。它是由经验丰富的开发人员构建的，它处理了 Web 开发中的许多麻烦，因此我们可以专注于编写应用程序，而无需重新发明方向盘。

2. Flask：对比 Django，Flask 更轻便小巧，更容易扩展功能。但是 Flask 中送提供的组件比 Django 就要少得多。

3. 经过对比和学习，我发现 Django 提供了一个叫做 haystack 的开源搜索框架，该框架支持 whoosh，因此实现起来会简单。所以我更倾向于使用 Django 作为我们这个项目的框架。

• Database

• 当我使用 whoosh 建立索引的时候，需要为索引建立 schema，我认为 schema 可以为{title: …, path:..., content:...}这样的一个格式，因此我们应该把每个描述文档经过处理得到这三个部分，然后储存进一个数据库或者 csv 表格中，这样会有助于我们批量建立索引和方便我们管理文档。因为我们目前的数据量非常少并且在的将来的一段时间内，我认为数据量的增长也是可以预测的，所以暂时使用一个 csv 表格就足以储存。如果在几年后视频数量达到几千的时候，我们可以使用

一个 mysql 数据库来替代这个表格。

• Metric

• 对于搜索引擎，有两个非常重要的指标需要考虑，一个是 precision 另一个则是 recall。precision 测量的是我们搜索得出的结果中与搜索词相关项的比例，而 recall 则算的是与搜索词相关的项我们找到了几个。这两个指标都非常重要但他们经常是 trade off 的，因此为了可以同时兼顾这两个指标并且不发生矛盾，我认为用 f1-score 来作为我们评价指标是一个好的选择。

• Summary

• 综合上述分析，这个项目目前的解决方案为先对描述文档进行处理并存入一个 csv 表格，然后使用 whoosh 为视频描述文档建立倒排索引，再利用 Django 框架解决用户交互问题，最后使用 f1-score 进行检验。这个方案足以应对我们现在的搜索需求，并且可以做到在模型更新后自动将线上的模型更新。我统计了 OneMinuteLaw 到目前为止上线了 11 个月，一共有 85 个视频，平均一个月是会更新七到八个视频，即使在以后开放了律师投稿环节，假设月视频更新量增长到三倍，即月增加了达到 20 多个，这个系统也完全足以处理，我们可以把系统的自动更新设置为一天一次。