Design Document:

Language: Python 3.8

My code requires the users to run at the root directory of the project. Then, users can build the indexes by calling:

```
$ cd …/searchEngine

$ python ./implementation/build.py input-documents-directory index-type index-directory
```

Example:

```
$ cd c:/Users/TianxiangLi /Desktop/ searchEngine
$ python ./implementation/build.py ./BigSample stem ./indexes
```

The program will write the index file to the index directory with name [index-type].txt. Then, the user can start query processing. If the users are calling the static query processing, here is the format of the command-line call:

```
$ python ./implementation/query.py index-directory query-file-path retrieval-model index-type result-file-path
```

Example:

```
$ python ./implementation/query.py ./indexes ./queryfile.txt cosine single ./results/cosine-single
```

If the users are calling the dynamic query processing, here is the format of the command-line call:

```
$ python ./implementation/query_dynamic.py index-directory query-file-path result-file-path
```

Example:

```
$ python ./implementation/query.py ./indexes ./queryfile.txt ./results/dynamic
```

Both for static query processing and dynamic one, the programs will write the retrieved document list for the queries to the result directory and output the runtime of the program on the console. For static query processing, the comments in the results will indicate which retrieval model the program used. For dynamic query processing, the comments in the results will indicate the retrieval model and the indexes the program used.

Here are the basic designs:

Query parsing:

While parsing the queries, the program will read lines from the query file and extract the id and title for each query. If the query parsing is called by static query processing, the function will return a dictionary of query list with query ids as the keys and the corresponding parsed and tokenized titles as the values. If called by dynamic query processing, the values of the query list will be a complete string of the title for further processing according to different index types. The parsing and tokenizing of the titles apply the same functions as those in project 1 when the program processes the documents according to the type of index.

While processing the index files, the program will create an object of index, which contains the necessary variables and counts for the relevance ranking models. The initiator of the index will read in lines from the index files, extract the lexicon and posting lists, and gather the document frequency, document term frequency, total number of terms, total term frequency, document lengths, and term positions.

Models:

The models function and call different types of modeling according to the retrieval type. It takes in query term frequencies, retrieval type, and an index object as an input for the necessary variables. The models function will output a list of scores for queries, each item will contain tuples of document id with its similarity score. There are three retrieval models:

1. Cosine: The cosine model uses term weights to calculate the similarity scores according to the lecture:

$$SC(Q, D_i) = \frac{\sum_{j=1}^{t} w_{qj}\, d_{ij}}{\sqrt{\sum_{j=1}^{t} \left(d_{ij}\right)^2 \sum_{j=1}^{t} \left(w_{qj}\right)^2}}$$

   For each term weight w, the program will use the normalized tf-idf recommended in the lecture:

$$w_{\{ij\}} = \frac{(\log tf_{ij} + 1.0) idf_j}{\sum_{j=1}^{t} [(\log tf_{ij}) + 1.0) idf_j]^2}$$

   And idf as: $idf_j = \log \frac{total\ number\ of\ documents}{df_j}$

2. BM25: The BM25 model modifies based on the probabilistic model, here is the way to calculate the similarity scores according to the lecture:

$$\sum_{j=1}^{t} \mathrm{w} \left(\frac{(k_1 + 1) tf_{ij}}{tf_{ij} + k_1 \left(1 - b + b \frac{|D|}{avgdl}\right)} \right) \left(\frac{(k_2 + 1) qtf_j}{k_2 + qtf_j}\right)$$

   I will tune the parameters $k_1 = 1.2, k_2 = 500, b = 0.75$. The program use the index object to get document length and the average document length by calculating $\frac{\text{total terms}}{\text{number of documents}}$.

The w, which is the idf of the query term, is separately calculated through the following equation:

$$\mathrm{w} = \log\left(\frac{\text{total number of documents} - \text{df of term} + 0.5}{df\ of\ term\ + 0.5}\right)$$

3. Language model: The language model calculates the probability of the query given the documents. In this project, I will use Dirichlet smoothing for the language model and use the log of the probability as the similarity score:

$$\log P(q_i|D) = \sum_{i=1}^{n} \log \frac{tf_{q_i,D} + \mu \frac{tf_{q_i,C}}{|C|}}{|D| + \mu}$$

I will tune μ as the average document length.

Static query processing steps:

1. Based on the user input, the main function will check the validity of command line arguments and open the query file, output file, and index file.
2. The main function will create an index object using the initiator of Index class from query parsing file. Then the main function will parse the query file and get the parsed query list and query term frequency list, where it passes them and the index object to models function and get the sorted scores.
3. The main function will write the scores to the output file

Dynamic query processing steps:

1. The main function set the retrieval model, scope of positions for the positional index, and number of documents checking if there are enough documents.

2. The main function will check the validity of command line arguments and open the query file and output file.
3. The main function uses the phrase index. It processes the phrase index file and check whether the phrase terms in the queries are common or not: I will use the mean document frequency of all terms as the indicator of whether the phrase term is common or not. For the queries with common phrase terms, the main function will call the models to calculate the scores.
4. The main function then uses the positional index for those queries without common phrase terms. It checks whether the query terms are within the set scope to see whether the document is relevant. For the relevant documents, the main function will call the models to calculate the scores and rank them.
5. If there are not enough documents retrieved for certain query, the main function will use stem index and calculate the scores using models function and append to the scores of that query.
6. Last, the main function will write the scores to the output file.