CS 589: Homework Assignment 2

Due: October 21 11:59pm

In homework 1, you implemented BM25 and the vector space model to retrieve StackOverflow's title, body and answer *separately*. In this homework, you will implement three retrieval models using ElasticSearch: TF-IDF, BM25, and Dirichlet language model. More specifically, you will learn how to:

- Install ElasticSearch on your machine;
- Create an index, insert a large corpus to the index using the ElasticSearch search engine;
- Retrieve documents using a customized similarity function (e.g., TF-IDF, BM25 and Dirichlet language model) using ElasticSearch;
- Evaluate your model's performance using ElasticSearch;

1 Data sets description

The dataset of Assignment 2 is the same as the dataset you used in Assignment 1, i.e., it contains three subdatasets: Python, Java and Javascript. Each dataset contains two documents: `$lang_qid2all.txt` (all the SO questions) and `$lang_cosidf.txt` (the relevance judgment between `qid1` and `qid2`).

2 Step 1: Installing ElasticSearch

This section briefly introduce how to install ElasticSearch and Kibana (a web UI tool for ES). For more information on ElasticSearch and Kibana, please refer to Sep 28's lecture on ElasticSearch and the slides: https://bit.ly/2GJj8l6.

2.1 Install ElasticSearch

- Download the latest version from the official website: https://www.elastic.co/downloads/elasticsearch.
- Unzip it and you will see a folder called 'elasticsearch-7.9.2'. In a terminal window, go to the directory and run *bin/elasticsearch* (or *bin\elasticsearch.bat* on Windows) to start the service.
- Open this link http://localhost:9200/ in a browser to check whether you have installed ElasticSearch successfully.

2.2 Install Kibana

- Download the latest version from the official website: https://www.elastic.co/downloads/kibana.

- Unzip it and you will see a folder called 'kibana-7.9.2-darwin-x86_64'. In a terminal window, go to the directory and run *bin/kibana* (or *bin\kibana.bat* on Windows) to start the service.
- Open this link `http://localhost:5601` in a browser.

3 Step 2: Index you data

Your task is to create a new index in the ElasticSearch and add documents from the `$lang_qid2all.txt` to it using the bulk indexing API.

- First, create an index using Kibana's `PUT` command. For example, if your index is named `index_name`, you should use the following command:

```
PUT /index_name
{
  "settings":{
    "analysis": {
      "analyzer": {
        "my_analyzer": {
          "tokenizer": "whitespace",
          "filter": [
            "lowercase",
            "porter_stem"
          ]
        }
      }
    }
  },
  "mappings": {
      "properties": {
        "title": {
          "type": "text",
          "analyzer":"my_analyzer",
          "similarity": "BM25"
        },
        "body": {
          "type": "text",
          "analyzer":"my_analyzer",
          "similarity": "BM25"
        },
        "answer":  {
          "type":  "text",
          "analyzer":"my_analyzer",
          "similarity": "BM25"
        }
      }
    }
}
```

Figure 1: The command used for creating an index.

In the above example, we have specified three parts of configuration parameters: (1) the `type` of the indexed data. There exists a list of `type`s to choose from: `https://bit.ly/3d3Db9I`. Because StackOverflow questions are strings, we should set `type` as `text`, i.e., analyzed, unstructured text; (2) analyzer, i.e., the special algorithms that determine how a string field in a document is transformed into terms in an inverted index. In this assignment, we will normalize words to their base form or lowercase by using analyzers. As shown in the above example, we have specified these choices in `my_analyzer`; (3) which similarity function to choose from. In this assignment, we need to implement three similarity functions: BM25, TF-IDF and Dirichlet LM. The default similarity function for ElasticSearch is BM25, therefore for TF-IDF and Dirichlet LM, we need to explicitly specify the similarity function using `similarity` as the above example shows. [1]

[1] In ElasticSearch, each index has a specified similarity function, and whenever the similarity function is changed, we need to re-index the data.

You can run the above command in the **Dev Tools** in Kibana to create the index. More information about creating index can be found here: https://bit.ly/2SxqVFb.

- Next, insert the StackOverflow questions to the index you just created. This step is called bulk indexing. To bulk index the data in `$lang_qid2all.txt`, first, you need to convert the questions in `$lang_qid2all.txt` to a JSON file, including all components of the questions. The following is an example of a JSON file, which only contains two documents.

```
{"index": {"_id": "30342145"}}
{"title": "redemption setting display name ", "body": "using add pst file attached pst
outlook problem use one adds correctly display name attached personal folders provide
display name set doesnt use still add pst personal folders way changes display name
according rd optional argument setting display name something different expecting
inform info needed ", "answer": "pst provider sometimes ignores specified displayed
name pst store added try set name property store added "}
{"index": {"_id": "28311568"}}
{"title": "randomly sample rows first dimension array unknown dimension ", "body":
"dimension known task trivial take d array np random randint np random choice shape
however function dimension array arbitrary handle situation ", "answer": "use size st
dimension determine random range prefer include ellipsis indexing array selects rows
default "}
```

  After storing the data in a JSON file, bulk index this JSON file with the RESTful API in a terminal window. For example, if the index name you have created is `index_name` and the JSON file is `python.json`, then you can run the following RESTful command in a terminal window to index all documents:

```
curl -s -H "Content-Type:  application/json" -XPOST localhost:9200/index_name/_doc/_bulk
--data-binary "@python.json"
```

  Bulk indexing in multiple batches. In ElasticSearch, the size of the file `python.json` in above command cannot exceed 100MB. However, the size of `$lang_qid2all.txt` exceeds 100MB, as a result, you need to split the data in `$lang_qid2all.txt` into multiple batches of JSON files and then bulk index each file separately. More information about bulk indexing can be found here: https://bit.ly/34qyMK5.

- Wait for a few seconds while the documents get indexed and then check that your index works correctly. You can do it by sending a simple request to the ElasticSearch and see whether it returns some results, for example, you can use the following command to check all the documents in the index through the Dev Tools in Kibana.

```
GET /index_name/_search
{
    "query":{
        "match_all":{}
    }
}
```

Note that you need to create 3 (dataset=Python, Java, JS) $\times$ 3 (similarity function=TF-IDF, BM25, Dirichlet LM)=9 indices, and bulk index the corresponding documents.

4 Step 3: Develop a Ranking Function

In this step we will retrieve relevant documents (`qid2s`) for each query (`qid1`) in the file `$lang_cosidf.txt` and compute the NDCG@10 for the search result at the same time. You will implement this section as

Algorithm 1: Ranking Function

Input: 1000 `qid1s` in `$lang_cosidf.txt`
Output: NDCG@10 score

1. `from elasticsearch import Elasticsearch`
2. `es = Elasticsearch()`
3. `ndcg_list=[]`
4. For each `qid1`:
5. `qid1_title=es.get(index=index_name, doc_type='_doc', id=qid1)['_source']['title']`
6. Load `ratings` for `qid1` # Computed by Algorithm 2
7. `_search = ranking(qid1, qid1_title, ratings)` # Figure 2
8. `result = es.rank_eval(index="python_tfidf",body=_search)`
9. `ndcg = result['metric_score']`
10. Append `ndcg` to `ndcg_list`
11. Compute and return the NDCG@10 score from `ndcg_list`

Algorithm 2: Generate `ratings` for each query

Input: 1000 `qid1s` in `$lang_cosidf.txt`
Output: Save `ratings` for all `qid1s`

1. For each `qid1`:
2. `ratings=[]`
3. Read 30 `qids2s` and their corresponding `labels` from `$lang_cosidf.txt`
4. For each `qid2, label` in `qid2s, labels:`
5. `ratings.append("_index":index_name,"_id":qid2,"rating":int(label))`
6. Save `ratings` as a JSON file

a Python for loop as shown in Algorithm 1, which is based on the Python Elasticsearch API [2]. In the loop of Algorithm 1, you need to compute the NDCG@10 score for each `qid1` by using the Ranking evaluation API [3] which actually looks like the function `ranking` defined in Figure 2. For each `qid1`, you need to feed the following three variables into `ranking`:

- `qid1`.
- `qid1_title` as the query. You can obtain `qid1_title` by a simple search in the index which corresponds to line 5 in Algorithm 1.
- The ground truth list from `$lang_cosidf.txt` as the `ratings`. You can generate the `ratings` for each `qid1` in the file `$lang_cosidf.txt` according to the **label** between `qid1` and `qid2s`. Algorithm 2 shows how to generate `ratings` for each `qid1`. We give an example of `ratings` for the `qid1:` 21024066 as below. Note that there should be 30 elements in the `ratings` when you actually perform this query. Here I just list 5 of them to illustrate it.

```
[
 {"_index": "index_name", "_id": "11917547", "rating": 1},
 {"_index": "index_name", "_id": "43330205", "rating": 0},
 {"_index": "index_name", "_id": "28853457", "rating": 0},
 {"_index": "index_name", "_id": "35788626", "rating": 0},
 {"_index": "index_name", "_id": "35027279", "rating": 0},

 '''
]
```

Besides the above three variables (the framed parts in Figure 2), the other fields such as `boost`,

[2] https://elasticsearch-py.readthedocs.io/en/master/
[3] https://www.elastic.co/guide/en/elasticsearch/reference/7.9/search-rank-eval.html

```
def ranking( qid1 , qid1_title , ratings ):
    _search = {
        "requests": [
            {
                "id": str(qid1),
                "request":{
                    "query":{
                        "bool": {
                            "must_not":{
                                "match": {
                                    "_id": qid1
                                }
                            },
                            "should": [
                            {
                                "match": {
                                    "title": {
                                        "query": qid1_title,
                                        "boost": 3.0,
                                        "analyzer": "my_analyzer"
                                    }
                                    }
                            },
                            {
                                "match": {
                                    "body":{
                                        "query": qid1_title,
                                        "boost": 0.5,
                                        "analyzer":"my_analyzer"
                            }}},
                            {
                                "match": {
                                    "answer":{
                                        "query": qid1_title,
                                        "boost": 0.5,
                                        "analyzer":"my_analyzer"
                            }}}
                            ]}}
                    },
                    "ratings":ratings
                    }
        ],
        "metric":{
            "dcg":{
                "k": 10,
                "normalize": True
            }
        }

    }
    return _search
```

Figure 2: The Command used for one query.

`analyzer` and `metric` are fixed and you can simply use the code in Figure 2. It is worth noting that `boost` is used to evaluate the performance of the weighted sum of the 3 components, instead of each component individually. As shown in Figure 3, the `result` returned by `es.rank_eval` in line 8 in Algorithm 1 is just like a dictionary which contains a lot of information, but we just extract the NDCG@10 score (the framed part in Figure 3) by performing line 9 in Algorithm 1.


```
{'metric_score': 0.6309297535714574, 'details': {'33949786': {'metric_score': 0.6309297535714574, 'unrate
d_docs': [{'_index': 'python_bm25', '_id': '46256747'}, {'_index': 'python_bm25', '_id': '45807493'}, {'_
index': 'python_bm25', '_id': '46565850'}, {'_index': 'python_bm25', '_id': '45555412'}], 'hits': [{'hit'
: {'_index': 'python_bm25', '_type': '_doc', '_id': '39691902', '_score': 75.21713}, 'rating': 0}, {'hit'
: {'_index': 'python_bm25', '_type': '_doc', '_id': '33992029', '_score': 68.947754}, 'rating': 1}, {'hit
': {'_index': 'python_bm25', '_type': '_doc', '_id': '46256747', '_score': 59.253178}, 'rating': None}, {
'hit': {'_index': 'python_bm25', '_type': '_doc', '_id': '45807493', '_score': 58.482243}, 'rating': None
}, {'hit': {'_index': 'python_bm25', '_type': '_doc', '_id': '45219736', '_score': 57.11952}, 'rating': 0
}, {'hit': {'_index': 'python_bm25', '_type': '_doc', '_id': '42330839', '_score': 56.58361}, 'rating': 0
}, {'hit': {'_index': 'python_bm25', '_type': '_doc', '_id': '46565850', '_score': 56.161354}, 'rating':
None}, {'hit': {'_index': 'python_bm25', '_type': '_doc', '_id': '45555412', '_score': 55.94519}, 'rating
': None}, {'hit': {'_index': 'python_bm25', '_type': '_doc', '_id': '44461197', '_score': 55.004856}, 'ra
ting': 0}, {'hit': {'_index': 'python_bm25', '_type': '_doc', '_id': '40685452', '_score': 54.86174}, 'ra
ting': 0}], 'metric_details': {'dcg': {'dcg': 0.6309297535714574, 'ideal_dcg': 1.0, 'normalized_dcg': 0.6
309297535714574, 'unrated_docs': 4}}}}, 'failures': {}}
```


Figure 3: An example of the result returned from .

5 Submissions

Finally, you need to submit the **code** and a **report**. **code** should include the Python code and all the command you used in this assignment. **Report** should include the following contents:

- Report 9 (3 (similarity function: tf-idf, bm25, Dirichlet LM) $\times$ 3 (dataset: Java, Python, JS)) NDCG@10 scores.