# CS246 Project 2

## Overview

Databases that deal purely in structured data (such as dates, numbers, and string enums) just check whether a document (or a row, in a relational database) matches the query. While Boolean yes/no matches are an essential part of full-text search, they are not enough by themselves. Instead, we also need to know how relevant each document is to the query. Full-text search engines have to not only find the matching documents, but also sort them by relevance. Full-text relevance formulae, or similarity algorithms, combine several factors to produce a single relevance score for each document. Of course, relevance is not just about full-text queries; it may need to take structured data into account as well.

In this project, we will understand the use of ranking functions in Elasticsearch (also called Similarity Functions) to sort documents that match a certain query according to their relevance. While working on improving the relevance of query results for our data set, we not only depend on having a good understanding of the dataset that we query, but we also need to have a good benchmark list of queries based on which we can test how well our relevance function is working.

In Task 1, we will understand how the ranking function defines the scores of the matched documents and test a few such functions.
In Task 2, we will change some of the parameters of the elasticsearch in-built ranking functions, to see how these parameters affect the similarity scores of the documents.
In Task 3 we will learn how we can define our own custom ranking function when the in-built similarity functions are not good enough for our task.

## Task 1: Comparing the in-built ranking functions

The primary goal of our first task is to understand how elasticsearch uses Similarity functions to sort documents based on relevance. As mentioned in class, **relevance** is a subjective term. As a result, different applications make use of different Similarity functions to rank results that match the query. While matching documents to a query, elasticsearch first applies the boolean model to find documents that satisfy the query.

So far, in all of our previous tasks, we used the Search Lite queries, which are executed against the `_all` field. We first learn a more complex search syntax that allows specifying search conditions on *multiple fields* of a document with complex boolean conditions. To learn this advanced syntax, **read the pages from The Search API through Executing Filters**. The pages explain how you can express complex boolean queries on multiple fields of a document using `bool`, `must`, `must_not` and `should` keywords. Roughly speaking, `must`, `should`and `must_not` correspond to AND, OR, and NOT in boolean algebra, respectively. Using this syntax, you can express any complex boolean expressions as search conditions.

Once Elasticsearch has identified a list of documents matching the boolean condition specified in the query, it must sort these documents based on their relevance to the query. To achieve this, it applies one of the similarity functions. In order to understand the different Similarity functions, you can start by reading the documentation on the Similarity Module.

### Task 1A: Using the BM25 Similarity function

BM25 is the default similarity ranking function used by Elasticsearch, which is known to work quite well for an article-length sized document corpus. BM25 is similar to traditional TF/IDF, however it allows searching documents without removing stopwords by setting a saturation limit on the term-frequency. For a more detailed explanation, read this page.

To begin, download project2.zip. This zip contains a Wikipedia dataset in the "data" folder. This dataset is similar to the one you indexed in project 1. The only difference is each document has an additional field called **clicks**. In this task, you must build an index named **task1a** with type **wikipage** with the default analyzers and the BM25 Similarity function. As mentioned above, BM25 is the default similarity function, so you can create this index and index the documents as you did in Task 2a of project 1. You can copy **ParseJSON.java** from project1. Once you have added the documents to this index, we can test how good the default ranking function works by running a set of queries on this dataset and seeing whether they are ranked in an order of relevance that we expect.

In Project 1, you submitted 10 benchmark queries as a part of Task 3. Run those 10 queries on this dataset and see if the top results match what you marked as the expected results. For example, the following query that checks for **bear** in either the "title" or "abstract" fields. We expect results with titles "Bear", "Eurasian Brown Bear", "Sloth Bear", "Sun Bear", "Tibetan Blue Bear", "Himalayan Brown Bear", "Syrian Brown Bear", "Knut (polar bear)" and "Asian black bear", that talk about various species of the animal to show up in the top 10 results.

```
curl -s -XGET 'localhost:9200/task1a/_search?pretty' -H 'Content-Type: application/json' -d'
{
  "query": {
    "bool": {
      "should": [
        {
            "match": {
                "title": "bear"
            }
        },
        {
            "match": {
                "abstract": "bear"
            }
        }
      ]
    }
  }
}
'
```

However, when we run this query on the index we just built, we get 76 hits with the top 10 hits getting the following scores:

| title | score |
|---|---|
| Teddy bear | 19.160645 |
| Bear Mountain | 18.565228 |
| Sloth bear | 18.260994 |
| Sun Bear | 18.195362 |
| Smokey Bear | 17.79319 |
| Gummy bear | 17.461761 |
| Fozzie Bear | 17.427917 |
| Tibetan Blue Bear | 16.803623 |
| Syrian Brown Bear | 16.775585 |
| Asian black bear | 16.530083 |

**Note**: In building your indices for Project 2, please make sure that you ***do not change the default setting for the number of shards per index***. As this page explains, IDFs are computed *per-shard* for efficiency reasons in Elasticsearch, so if you change the number of shards, you are likely to get different results. Also please make sure that you use the line number of each page as its _id when you index the pages. If not, you may get different results as well because the way documents are assigned to a shard. Finally, all our results on this page are based on the assumption that ***you created each index using the name specified in each task***. Because Elasticsearch uses the name of the index in assigning a document to a shard, if you use a different index name, you are likely to a result different from what is given by us. (credit to Jia Teoh)

One way of figuring out how relevant our query results are would be by looking at the Precision and Recall scores for the results.

- **Precision:** Precision is the fraction of the documents retrieved that are relevant to the user's information need.

$$precision = \frac{|relevant\ documents\ \cap\ retrieved\ documents|}{|retrieved\ documents|}$$

- **Recall:** Recall is the fraction of the documents that are relevant to the query that are successfully retrieved.

$$recall = \frac{|relevant\ documents\ \cap\ retrieved\ documents|}{|relevant\ documents|}$$

For modern (Web-scale) information retrieval, recall is not as a meaningful metric as precision, as many queries have millions of relevant documents, and few users will be interested in reading all of them. In such cases, we use a metric called Precision@k documents. For example, **Precision@10** corresponds to the number of relevant results in the first 10 hits. This is a useful metric, as the top 10 hits are usually the results that get rendered on the first page of a search. In task 1 and 2 of this project, we will check the relevance of our results using the Precision@10 metric.

$$precision@10 = \frac{|relevant\ documents\ \cap\ first\ 10\ retrieved\ documents|}{10}$$

In the results for the "bear" query, we have 5 of our expected results in the first 10 results. Thus we have a Precision@10 of 0.5 .

## Task 1B: Using the traditional TF/IDF Similarity function

You will now build another index named **task1b**, and this time set the similarity function to `classic`. The `classic` similarity Elasticsearch's implementation of a ranking algorithm that is very similar to the traditional TF-IDF. To understand how you set the similarity function while creating a particular index, read the details given in the Similarity Module documentation. Running the same query as above for "bear" with this index, we get the same 76 hits. But the top 10 hits now get the following scores:

| title | score |
|---|---|
| Bear Mountain | 6.9686794 |
| Smokey Bear | 6.701144 |
| Teddy bear | 6.210139 |
| Sun Bear | 6.121539 |
| Sloth bear | 6.0821376 |
| | |

| | |
|---|---|
| Himalayan Brown Bear | 5.9258537 |
| Fozzie Bear | 5.8738117 |
| Gummy bear | 5.818371 |
| Yogi Bear | 5.6418853 |
| Golden Bear | 5.5861278 |

**Note**: The absolute score of each document will change depending on how you set the similarity function for your index. The above values were obtained by setting the *default similarity* to be `classic` for the index, as opposed to setting the similarity of each field individually. If you set individually, your scores are likely to be higher roughly by a factor of 11.85 compared to the values listed above. This is because Elasticsearch uses different `queryNorm` and coord factors depending on how similarity is set as the Similarity module page explains. To be consistent with the results shown above, please *set the default similarity for your index for this project*.

From the above result, we see 3 of our expected results in the top 10. That gives us a precision@10 value of 0.3. Now run your 10 "benchmark queries" from Project 1 - Task 3, and see how this index fares as compared to the default one.

Do you think the BM25 similarity function ranked the results better than `classical` model? Does BM25 work better only for this particular query or would all queries work better with the default index? It is difficult to tell if the change in similarity function improves the ranking of relevant results for most user queries by looking at just a few queries. We therefore need a large enough benchmark set of queries that can be tested. It is important that this set of benchmark queries are indicative of the kind of queries an average user of the search engine would run, and the results the user would expect.

In Task 3 of Project 1, you submitted a list of queries with their expected results. We have compiled around 500 such queries with their results in benchmark.txt in project2.zip. We have also provided you with a script that runs these queries and calculates the precision@10 for each query. Go ahead and run the script to check which of the two indexes you just built has a higher average precision value. You can run the script using the following command:

```
./benchmark.sh task1a
```

Here the parameter passed is the index on which you would want this script to run. Run this script on `task1b` as well and see which index seems to be performing better. The average precision@10 value for our benchmark query set may be close to 0.2 as most of the queries have only 1 or 2 relevant results listed. You can thus see, that it is important to have a good benchmark query set, so that you can really check the performance of your similarity function.

**Notes on CR/LF issue:** If your host OS is Windows, you need to pay particular attention to how each line of a text file (including your script file) ends. By convention, Windows uses a pair of CR (carriage return) and LF (line feed) characters to terminate lines. On the other hand, Unix (including Linux and Mac OS X) uses only a LF character. Therefore, problems arise when you are feeding a text file generated from a Windows program to a Unix tool (such as bash script). Since the end of the line of the input file is different from what the tools expect, you may encounter unexpected behavior from these tools. If you encounter any wired error when you run your script, you may want to run the `dos2unix` command in VM on your Windows-generated text file. This command converts CR and LF at the end of each line in the input file to just LF.

# Task 2: Understanding how the parameters of the Similarity function affect the ranking

Now you have seen how BM25 works, but sometimes, the results from BM25 aren't optimal. In order to improve the results, we can tune the BM25 function by varying its parameters. The BM25 has three parameters that allow it to be tuned:

- **k1**: This parameter controls how quickly an increase in term frequency results in term-frequency saturation. The default value is 1.2. Lower values result in quicker saturation, and higher values in slower saturation.
- **b**: This parameter controls how much effect field-length normalization should have. A value of 0.0 disables normalization completely, and a value of 1.0 normalizes fully. The default is 0.75.

The optimal values for each of these parameters really depend on the collection of documents that you work with. Finding good values for your collection is a matter of adjusting, checking, and adjusting again. In this task, build an index called **task2** and vary the values of k1 and b and see how varying these values affect the ranking of the results. To understand how to set the values of b and k1, refer to Configuring BM25 section of the elasticsearch documentation.

Running the following command should give you results identical to those you got in task 1a, as this command sets the parameters to the default values for BM25.

```
PUT task2
{
    "settings": {
        "index":{
            "similarity" : {
                "default" : {
                    "type" : "BM25",
                    "k1" : 1.2,
                    "b" : 0.75
                }
            }
        }
    }
}
```

Given below are some of the values of b and k1 and the top 5 results for same "bear" query as above for each of them.

| b | k1 | Top 5 results | | | | |
|---|---|---|---|---|---|---|
| | | **Rank 1** | **Rank 2** | **Rank 3** | **Rank 4** | **Rank 5** |
| 0.00 | 0.00 | Eurasian Brown Bear | Knut (polar bear) | Base Ball Bear | Golden Bear | Big Bear Lake, California |
| 0.00 | 0.80 | Asian black bear | Teddy bear | Tibetan Blue Bear | Short-faced bear | Eurasian Brown Bear |
| 0.25 | 1.20 | Teddy bear | Asian black bear | Tibetan Blue Bear | Sloth bear | Sun Bear |
| 0.50 | 1.60 | Teddy bear | Sloth bear | Sun Bear | Asian black bear | Tibetan Blue Bear |
| 0.75 | 1.60 | Teddy bear | Bear Mountain | Sloth bear | Sun Bear | Smokey Bear |
| 1.00 | 2.00 | Bear Mountain | Smokey Bear | Himalayan Brown Bear | Teddy bear | Sloth bear |

For efficiency reasons, Elasticsearch precomputes and stores "document lengths" in the index when each document is indexed. Therefore, if your new similarity function uses a different "document length" definition from your old simialrity function, you will need to *rebuild your index* in order to get the correct document

length. Fortunately for BM25, different k1 and b parameter settings does not change the "document length", so you do not need to rebuild your index each time you change the parameters. However, please note that if you change the similarity function or its arameter values after creating the index, ***you must close and reopen your index first before your change takes effect.*** Otherwise, Elastic will still run your queries with the old similarity setting.

You will notice that as you vary the values of b and k1, the results change. Some values of these parameters give better results, while others give us worse results. Once again we will use the precision@10 values for the benchmark query set that we have built to see which parameter values are optimal. In this task, keep varying the values of b and k1 as given below and find the parameter values that result in query results that are closest to the expected values for the benchmark query dataset. Everytime you change the index setting with the new values of k1 and b, run the `benchmark.sh` script and see the precision@10 values. The optimal parameter are the ones that give you the highest value for precision@10. The table below has the average precision@10 values for some combinations. You can use these to check that your script works properly. Find the average precision@10 values for the remaining parameter values and find the value that has the best results. Once you find the best parameters, add approriate commands to build.sh to create an index named **task2** with type **wikipage** that uses the best BM25 parameter setting.

| | | **k1 values** | | | | |
|---|---|---|---|---|---|---|
| | | **0.00** | **0.80** | **1.20** | **1.60** | **2.00** |
| **b values** | **0.00** | 0.1547 | 0.2376 | 0.2378 | | |
| | **0.25** | 0.1547 | 0.2521 | | | |
| | **0.50** | | | | | |
| | **0.75** | | | | | |
| | **1.00** | | | | | |

**Note:**

1. You could write a script in python or bash to change the settings of the index with varied BM25 parameters and run the given script to find the best parameter values.
2. You only need to include the command for setting the optimal value in the `build.sh` file that you submit.

# Task 3: Customizing Ranking Function

In most cases, we can get relevant documents simply by using one of the in-built similarity functions of Elasticsearch, but sometimes we may not. In those cases, it may be necessary to leverage our special knowledge on our corpus to create our own similarity function that improves search relevance to a satisfactory level. For example, if we know that most users are primarily interested in looking at "popular pages" from their query, we may want to give higher similarity score to the pages with high monthly pageviews. As the final task of Project 2, we will learn multiple mechanisms in Elasticsearch that enable customizing the similarity scoring function.

## Task 3A: Per-Field Similarity Weights and Boost

Perhaps, the most common scenario that requires an adjustment in the similarity score is when we want to assign different weights to each field of a document. For example, if we know that the title of each page includes the most relevant keywords among all fields, we may want to give a significantly more weight to the matches on the title field than others. In Elasticsearch, this type of "per-field weighting" can be implemented using a "structured multifield search" and a "boost" factor.

In Elasticsearch, it is possible to assign a "boost factor" to each condition of a multifield search, so that a document that has a match on one field is given much higher relevance score than another that has a match on

another field. This is done by specifying the `boost` value on each condition of the query. **Read the page Multiple Query Strings** to learn how.

To understand how Elasticsearch computes the final similarity score of a document for a multifield `bool` query, let us consider the following example query:

```
GET /_search
{
  "query": {
    "bool": {
      "should": [
        {
            "match": {
                "title":  {
                    "query": "War and Peace",
                    "boost": 4
                }
            }
        },
        {
            "match": {
                "author": "Leo Tolstoy"
            }
        }
      ]
    }
  }
}
```

Roughly, the above query looks for documents whose title contains the keywords "war and peace" or author contains the keywords "leo tolstoy." (remember that in a `bool` query, `must`, `should`and `must_not` correspond to AND, OR, and NOT in boolean algebra, respectively.) In addition, it assigns the boost factor 4 to the title field, so that a match on title is considered 4 times as "important" as a match on author. More precisely, Elasticsearch computes the final similarity score of a document to a `bool` query as follows:

1. It computes the simiarlity score on each condition of the query using the similarity function assigned to the field in the condition. For example, let us assume all fields use the default similarity function BM25Similarity. Let us also assume the BM25 score from the "title" condition is `0.3` and the score from the "author" condition is `0.5`.

2. Once the scores from all conditions are computed for a document, Elasticsearch computes the final similarity score of the document to the query by *summing up the scores* from each condition weighted by their "boost" value. In our example, the "title" condition has the boost value 4 and the "author" condition has no boost value, so the final similarity score of document to the query will be `4*0.3 + 1*0.5 = 1.7`.

**Note**: Sometimes, when we combine scores from multiple conditions we may want to take the maximum of those score, not their sum. In those cases, we can use `dis_max` query. We will not consider those cases in this project, but if you are interested in learning about how to use `dis_max` queries, read the page on Dis Max Query.

In project2.zip, we have provided a simple shell script `task3a.sh` that takes a query as the input parameter and sends it as a query to the `task1a` index within Elasticsearch. Currently, the query retrieves the documents that contain (a subset of) query keyword(s) either in the "title" or "abstract" fields. Your job for this task is to change the script such that:

1. The query retrieves documents that contain (a subset of) query keyword(s) **both** in the "title" and "abstract" fields, but **no** query keyword(s) in the "sections" field. That is, *both the title and abstract must contain at least one keyword in the query, but sections should contain no query keyword*.

2. For similarity score computation, assign the boost factor 10 to the "title" field and no boost factor to others.

Update the `task3a.sh` script according to the above requirements and make sure that it works as you expect.

## Task 3B: Per-Document Similarity Weight and DocValues

In certain cases, field-level weight adjustment may not be adequate and we may want to give higher weights to a certain set of documents. For example, if we know that most users are mainly interested in looking at "popular pages", we may want to increase scores for the pages with high monthly pageviews. The implementation of this "document-level" score boosting requires two things:

1. We need a mechanism to store document-specfic feature value (such as monthly page views), which can be *efficiently and quickly looked up* during similarity score calculation
2. We need a mechanism to *customize the similarity score function*, so that we can include a document-specific feature value during score calculation.

In Elasticsearch, we can efficiently access the value of any field of a document during similarity score calculation through a mechanism called *DocValue*. A field that is declared to be "doc_values" is stored in a special data structure called `DocValues` that allows efficient retrieval during similarity score calculation. There are two ways to declare a field to be a `DocValue`: (1) when a field is decleared to be `not_analyzed` for `index`, they are considered to be `DocValue`, (2) a field is explicitly declared to be `doc_values`. **Read this page to learn how to do this.**

In the Wikipedia dataset in project2.zip, we have added a field named "`clicks`", which represents a (fake) monthly pageview of each page. We will use this "signal" inside our similarity score function to improve its effectiveness. Now your job for Task 3B is the following:

1. Create a new index named `task3b` with `wikipage` type
2. Make sure that make sure that the `clicks` field is stored as a `DocValue` with `long` datatype
3. Use "text" datatype for all other fields
4. Use the default standard analyzer and
5. (For now) use the default BM25Similarity with default settings for all fields

Update your `build.sh` to build `task3b` index as above.

Once a document-specific feature value is stored as a `DocValue`, we need to create our own similarity scoring function that can use this value during score calculation. This can be done through `Similarity Plugin`.

**Note**: In addition to `Similarity Plugin` Elasticsearch allows to customize the scoring function using a simple script through Function Score Query. While this mechanism is easier than creating a `Similarity Plugin`, its functionality is limited and slow. For this reason, it is mainly used for experimental settings to try out different scoring functions, but not for a production environment where performace and efficiency are critical.

## Task 3C: Similarity Plug-In for Elasticsearch

The functionality of Elasticsearch can be easily extended with plugins. There are two kinds of plugins: site and jvm. JVM plugins are java code and allow developers to extend the actual engine, while the site plugins are html and javascript. We now learn how we implement a specific function as a ElasticSearch JVM plugin for Similarity function.

### Building the Plugin

To build the similarity plugin, we will create three classes extending the following three abstract classes in ElasticSearch:

1. `Plugin`: This "plugin" class is a glue between Elasticsearch and our own Java class. It allows us to "register" our Java class to a particular name, so that our class can be referenced and used in Elasticsearch configurations.
2. `SimilarityProvider`: This is the "factory" of our similarity class. Whenever Elasticsearch needs our similarity class, it create one using this class.
3. `Similarity`: This is the key class where we implement our similarity score function.

If you are interestd in learning the role of these three classes in more detail, you may find this blog on Custom Similarity For Elasticsearch helpful (a local copy is available here). In `src/main/java/edu/ucla` directory of the project2.zip file, we have included three classes, `CS246Plugin`, `CS246SimilarityProvider`, and `CS246Similarity` that extend the above three classes, respectively.

**Note**: When you unzip the project2.zip file, it is important that you unzip it within a directory named `project2`, because your developed plugin is named after the directory where it is developed. Otherwise, when you try to deploy your plugin later, you may get an unexpected error due to naming mismatch.

As you can see from the source files, the codes for `CS246Plugin` and `CS246SimilarityProvider` are minimal, including minimum scaffolding codes. The only important part is the following function definition in `CS246Plugin.java`:

```
public void onIndexModule(IndexModule indexModule) {
    indexModule.addSimilarity("cs246-similarity", CS246SimilarityProvider::new);
}
```

which registers our similarity class to the name "cs246-similarity". This allows us to use our similarity plugin almost like an in-built similarity function, simply by using the name "cs246-similarity".

The code for the `CS246Similarity` class is much longer and complex, since it has to deal with complex and highly-optimized data structures of Lucene and Elasticsearch. To help you complete the rest of this project, however, we have isolated the key function(s) that you need to understand and modify to the begining of the file. In particular, the following `score()` method is the key function that you will need to modify (together with its three support functions, `idf()`, `docValueBoost()`) to implement our similarity function:

```
 /**
   * Score the document for one term. This function is called for every term in the query.
   * The results from calls to this function are all added to compute the final similarity score.
   * @param stats collection-wise statistics, such as document frequency and total number of docume
   * @param tf  "term frequency" of the current term in the document
   * @param docLen the length of document, |d|
   * @param docValue document-specific signal that can be used for score calculation
   * @return computed similarity score between the current term and the document.
   */
  protected float score(BasicStats stats, float tf, float docLen, long docValue)
  {
    // The first parameter stats has the following collection-level statistics:
    //    stats.numberOfDocuments: the total # of documents in the collection
    //    stats.numberofFieldTokens: the total # of tokens extracted from the field
    //    stats.avgFieldLength: the average length of the field
    //    stats.docFreq: the document frequency
    //    stats.totalTermFrequecy: the collection frequency
    //             (= total # of occurence of the term across all documents)
    //    You may use these statistics to compute the idf value, for example.
    // The second parameter tf has "term frequency"
    // The third parameter docLen is the "document length", |d|
    // The last parameter docValue has the value in the field "clicks" of the document.
    //    Note: If we want to use a value from a field other than "clicks", we need to change
```

```
    //    the CS246Similarity constructor by setting "signalField" to the name of the desired field

    return tf*idf(stats)*docValueBoost(docValue)/docLen;
  }
```

As its input parameter, we pass all key statistics that are needed for (practically all) standard similarity functions (together with the docValue obtained from the "clicks" field). Your job now is to modify `score()`, `idf()`, and `docValueBoost()` functions to implement the following similarity function:

$$f(q,d) = \frac{1}{|d|}\log_2(\text{clicks}_d + 1)\sum_{t \in q} TF_{t,d}\log_2\left(\frac{N+1}{DF_t+1}\right)$$

where $|d|$ is the document length, $\text{clicks}_d$ is the (fake) pageviews of $d$, $TF_{t,d}$ is the term frequency of term $t$ in document $d$, $DF_t$ is the document frequency of $t$, and $N$ is the total number of documents in the corpus. Please note that the `score()` function is called once per every term in the query, and the results from these calls will be added to compute the final similarity score of a document.

To help you compile your Java Similarity classes and package them for deployment at Elasticsearch, we have included a Gradle build script, `build.gradle`, in the project2.zip file. With this script, you can simply execute

```
gradle assemble
```

in your `project2` directory to (re)compile your code and create a plugin zip file suitable for Elasticsearch deployment. If successful, the build script will produce `project2-5.6.2.zip` in `build/distributions`, which can be deployed to Elasticsearch as a plugin. It is okay to know nothing about Gradle build script, as long as you can use our script to compile your code and produce the plugin zip file. But if you want to learn more, read one of many online tutorials on Gradle.

Once your Java Plugin Package zip file is successfully built, you can run our plugin installation script, `install-plugin.sh`, to deploy it to your Elasticsearch:

```
sudo ./install-plugin.sh
```

After deploying your plugin, you will need to wait for 10-30 seconds for Elasticsearch to restart.

Now that you have successfully built a Similarity Plugin and deployed it to Elasticsearch, you need to **rebuild** the `task3b` index, so that it will use our `"cs246-similarity"` as the similarity function of all fields, not the default "BM25". Update your `build.sh` file accordingly to reflect this change and build `task3b` index again. (Again, remember that Elasticsearch may store different document lengths depending on the similarity function used for each field, so it is always safe to rebuild your index if you change your similarity function.)

Test your implementation of similarity plugin and make sure that it works correctly. In debugging your similarity implementation, you may find the Explain API of Elasticsearch useful.

If your similarity function works properly, running the same "bear" query from task 1 on this index, gives us 76 hits with the top 10 hits having the following scores:

| title | score |
|---|---|
| Bear | 143.67433 |
| Bear Mountain | 82.02579 |
| Smokey Bear | 76.40336 |
| Teddy Bear | 76.2774 |
| Sun Bear | 75.67254 |

| | |
|---|---|
| Sloth Bear | 70.36775 |
| Golden Bear | 68.81387 |
| Bear Grylls | 66.08473 |
| Spectacled bear | 63.92398 |
| Gummy bear | 63.896713 |

# Your Final Submission

Your project must be submitted electronically before the deadline through our CCLE Course Website. Navigate to **Projects** on left of the page, and click on the **Project 2** submission section. If you submit multiple times, we will grade only the latest submission.

## What to Submit

The zip file that you submit must be named `project2.zip`, created using a zip compression utility (like using "`zip -r project2.zip *`" command in the VM). You should submit this single file **project2.zip** that has the following packaging structure.

```
project2.zip
    +- resources
    |
    +- src
    |  +- main
    |     +- java
    |        +- edu
    |           +- ucla
    |              +- CS246Plugin.java
    |              |
    |              +- CS246Similarity.java
    |              |
    |              +- CS246SimilarityProvider.java
    |
    +- build.sh
    |
    +- build.gradle
    |
    +- install-plugin.sh
    |
    +- ParseJSON.java (or whatever other script you used for formatting the dataset)
    |
    +- task3a.sh
    |
    +- README.txt (Optional)
```

# Testing Your Submission

Grading is a difficult and time-consuming process, and file naming and packaging convention is very important to test your submission without any error. In order to help you ensure the correct packaging of your submission, we have made a "grading script" p2_test available. In essence, the grading script unzips your submission to a temporary directory and executes your files to test whether they are likely to run OK on the grader's machine. Download the grading script and execute it in the VM like:

```
cs246@cs246:~$ ./p2_test.sh project2.zip
```

(if your project1.zip file is not located in the current directory, you need to add the path to the zip file before project1.zip. You may need to use "chmod +x p1_test" if there is a permission error.)

You **MUST** test your submission using the script before your final submission to minimize the chance of an unexpected error during grading. Again, significant points may be deducted if the grader encounters an error during grading. Please ensure that the Elasticsearch service is running before you execute the script. When evertything runs properly, you will see an output similar to the following from the grading script:

```
Getting files...
Deleting any old indexes...
Running build.sh...
Testing task1a...
SUCCESS!!

Testing task1b...
SUCCESS!!

Testing task2...
SUCCESS!!

Testing task3a...
:
:
:
SUCCESS!!

Testing task3b...
SUCCESS!!
```