CineMatch Search Engine

Katheeravan Balasubramaniam
147744
School of Computer Sciences
Universiti Sains Malaysia
Georgetown, Malaysia
katheeravan305@student.usm.my

Omsyaran Chandran
148869
School of Computer Sciences
Universiti Sains Malaysia
Georgetown, Malaysia
omsyaranc@student.usm.my

***Abstract*— A search engine is used to facilitate the process of finding specific information from a large collection of information that is stored. Thus, information retrieval techniques had to be performed information need. A search engine consists of 2 primary functions which are the indexing process and the querying process. Indexing focuses on developing the structure that supports searching of information while querying process utilizes that structure to produce a result which would be ranked based on relevancy of the search query. In this paper, a domain specific search engine is built with the use of Apache Solr. A domain is specified for this search engine which would be the Entertainment domain. This paper elaborates on the methods and techniques that have been used to develop the search engine. Followed with the evaluation of the search engine based on the results that it produces.**

Keywords—search engine, information retrieval, Apache Solr, indexing process, querying process.

I. INTRODUCTION

This paper documents the entire process of developing a domain specific search engine. A specific domain would be selected, and a search engine will be built for that domain. A domain is selected to narrow down the scope of words and phrases that can be used as a query to perform a search and obtain results from it. Specifying domain does help to limit down on the dataset that had to be collected.

Search engine relies heavily on the information retrieval concept. This is because the primary task of a search engine is to locate and display all the data that matches or is relevant to the query that is made by the user. The search engine facilitates the information need of a user from a huge collection of data that is stored. Additionally, the search engine is also responsible to carry out the indexing and querying process. The search engine would create a ranked list based on the score that is computed by the retrieval model.

Since the domain would be specified, the data that is indexed for the search engine usage must be related to that specified domain only. Each of the subprocess within the indexing and querying process will be given equal importance in development to ensure that search engine would perform up to its ability. The common issues in designing a search engine such as coverage, information retrieval related, and performance will be tackled.

II. OVERVIEW

A. Problem Domain

The domain that is picked would be the Entertainment domain. Thus, the data that is collected for this search engine will be related to the Entertainment domain only. There are tons of subdomains such as movies, song, celebrities, events, art, dance, animation, radio, and many more under the Entertainment domain. Hence, for this case only 1 subdomain out of the Entertainment domain will be picked which is the movie subdomain. So, the dataset used for the search engine will be made up of movies dataset. The keyword that is used for query will be related to this subdomain as well. The idea is for the user to query for any specific movie information and the relevant movie which matches with the query will be displayed as the result.

B. Problem Statements

There are a total of 3 problem statements for the development of this search engine. The first problem statement is the need to develop a movie search engine that provides a comprehensive and precise information about movies. The second problem statement is the need to create a movies search engine that effectively addresses searching issue related to misspelled and inaccurate queries. The third and last problem statement is the need to design a movies search engine that supports filtering options such as searching for movies with specific actors, genres, movie rating or movie titles.

C. Search Problem Scope (3 Queries)

The search problem scope must be defined to set the scope of the search that can be done on the search engine. Plus, it helps to signify the limitations of the search engine. The search problem scope for the search engine can be established with 3 queries which is defined in Table 1.

Query	Result Description
dark waters, dark knight	Movies named “dark knight” should be displayed first at the top of the result search. Then, the movie that has the word “dark” or “knight” shall be displayed. After this is displayed, if any of the movies has the word “dark” and “knight” in its overview, it would be displayed next. The same applies for the phrase “dark waters”.
robot movies, movies about robot	Movies that are related to robots would be displayed at top first. Then, the movies which shares the synonym of the word robot such as bots and machines will be displayed next at the search result. This is applied for both queries.
tum, dikaprio	Actor names are often spelled incorrectly since each name are uniquely spelled. Thus, when the search query is made for the word ‘tum’, movie star names that has the word ‘Tom’ in it would be displayed as the search result. As for the word ‘dikaprio’, the search result shall show movies that was acted by “Leonardo DiCaprio”.

Table 1: The 3 sample queries which establishes the search problem scope for this movie search engine.

III. Dataset Description

The datasets that are used for this search engine is obtained from the Kaggle website. Only 1 dataset is used for this search engine development. The dataset contains data about the top thousand IMDB ranked movies. The name of the movie dataset is 'IMDB Movie' dataset. This dataset contains information about movies such as poster links, movie title, release year, certificate, runtime, genre, IMDB rating, movie overview, meta score, director name, movie cast, number of votes and movie gross. This will be the information that can be used by the user to query at the search engine.

IV. Related Works

A. Sentence-BERT Semantic Search System

In this paper, the Sentence-BERT (SBERT) is used to develop a search system, a semantic search system to be exact. [1] This system is developed in Japan, to locate similar medical accident that has occurred in the Japanese closed medical accident claims database.

The implementation of the SBERT is done by using a Sentence-Transformers which is a Python library. This Python library is based on PyTorch which is basically a deep learning framework and the Transformers repository that is provided by Hugging Face Inc. SBERT can be broken down into 2 types. The first type would be the bi-encoder model and the second type would be the cross-encoder model. [1] The bi-encoder model which is the first type is used for in this paper. This is because, the bi-encoder model can run at higher speed. In other words, this bi-encoder model operates at higher processing speed.

Higher processing speed is prioritized since in this paper, the aim was to find and identify the texts that are similar with each other from the Japanese database that has more than 30.000 texts stored in it. Hence, the SBERT model that uses the bi-encoder model is developed and trained with the texts that is stored in the electronic health record system of the University of Tokyo Hospital, the UTH dataset. [1] It has roughly around 120 million clinic texts data stored in it. The text similarity search engine is built based on the SBERT model that is trained with the UTH dataset. The top-X number of text that has high similarity score with the query that is made would be extracted and displayed. The similarity score between the query and the texts is measured by using the Euclidean distance formula. The distance between 2 texts can be computed since the text embeddings was performed by the bi-encoder model for all the texts. [1] The smaller the distance, the higher the similarity.

To perform comparison and evaluation, another model is created which is the Okapi BM25 model. The Okapi BM25 model has its own search engine implemented. However, the Okapi BM25 did not use Euclidean distance formula to measure the similarity between the query and the texts. It uses the inverse document frequency weight instead. Thus, higher BM25 value signifies higher similarity between the query and the text. In this paper, the accuracy is computed by determining the number of top-X number of texts labels that matches with the query that is made. [1] The focus was not set on the ranking result of the extracted result after the query is made. Hence, the Hamming score and Hamming loss is used as the evaluation metrics.

Hamming score is used to measure the multilabel classification and the Hamming loss is used as a mismatch measure. Higher Hamming score and lower Hamming loss is desired. In the end, the trained SBERT model performed much better than the Okapi BM25 model. This is because SBERT model has been trained with the UTH dataset which helped to produce better text embeddings. [1] Text embedding affect the Euclidean distance calculation which affects the overall accuracy of the result.

V. Design Methods

A search engine has 2 primary processes that had to be built to ensure that the search engine works correctly. The first one would be the indexing process while the second one would be the query process.

A. Indexing Process

Indexing process in the development of a search engine consists of 3 sub-process that had to be performed to cover the entire indexing process. This would be the text acquisition, text transformation and lastly the index creation. For the text acquisition, the focus will be on the documents collection where the data that will be indexed has to be identified. Hence, for this movie search engine, the 'IMDB Movies' dataset is used. It comes in the CSV format, and it is obtained from the Kaggle website. The next step is the text transformation that will be done on the dataset that had been picked. Text transformation is important since it facilitates the query search that will be made by the user. It also helps to create a better index for the search engine.

The first text transformation that is made would be the square bracket removal "[]" for the "Series Title" and "Overview" field. The square bracket had to be removed since the presence of the square brackets affected the query result which yields incorrect results. Thus, this is achieved by setting the "multiValue" parameter to "false" as shown in Figure 1. This prevents the square brackets from appearing.

The second text transformation is based on the tokenization for the hyphenated words. Words such as 'film-noir' and 'sci-fi' from the 'Genre' field were being tokenized into 'film' and 'noir' as for 'sci-fi', it was tokenized into 'sci' and 'fi'. Thus, the default tokenization method used by the 'Genre' field had to be changed. Hence, a new tokenization class from the Solr class had to be used instead which is the "solr.PatternTokenizerFactory". [2] The parameter "pattern" had to be specified with a regular expression since this tokenizer class is regular expression based as shown in Figure 2.

The last text transformation is related to phonetics. Phonetics matching is implemented for the 'Movie_Cast' field from the dataset. Phonetics are basically words that have different spellings but share the same sound when it is pronounced. Phonetics matching is used for the movie cast name since names can be spelled differently but, it can have the same sound when it is pronounced. For instance, the name "Brad Pitt" sounds the same as the word "Brat Pit". Thus, when the user types in the query "Brat Pit", movies acted by "Brad Pitt" will be shown as the result of the query. This phonetic matching is done with the use of the Solr class which is the "solr.BeiderMorseFilterFactory". [2] Parameters such as

"nameType", "ruleType"," concat", and "languageSet" within this class are left with the default values as shown in Figure 3.

The text transformations are consistently applied where all these text transformations are done at the indexing and at the querying part. All this text transformation is applied by using the Schema API functionality that is provided by Apache Solr. Thus, the text transformation will be written in JSON format. This JSON script will then be pasted at the 'Add Field Type' text editor box. This can be found at the Solr Admin page under the Schema after clicking 'Manage Field Type'. [2] The core had to be clicked earlier, where for this assignment it would be the 'movies' core. After adding the new field type, another new JSON script which replaces the default field type with the newly added field type must be executed as shown in Figure 4. The JSON script which performs the replacement of field type is then submitted at the 'Replace Field Type' text editor box.

All the newly added field type which contains all the text processing steps specified will be added to the 'managed schema.xml' file. The added field type can be searched inside this file. The 'managed schema.xml' is responsible for performing the text processing on the dataset during indexing and on the queries that will be made by the user.

B. *Querying Process*

For query processing, two features were used which are spell checking and synonym filter. Spell Checking is a component in Solr that helps to provide users with new suggestion words based on similar or other terms depending on the query. [2] These suggestions provided can be based on a particular field in the indexed data or created in a separate file. In the proposed design, spell checking was implemented on a particular field in the dataset called "Series_Title". When searching for movies, users will usually enter the movie's name but sometimes they will make spelling mistakes as they might not know how the movie title is exactly spelled. Thus, a spell-checking feature was implemented in the "Series_Title" field which represents the movie name. When a user enters a query, the query will be used to cross check all the retrieved documents' "Series_Title" field to identify any similar or slightly different words. These words will then be shown to users as suggestions. For example, if a user enters the query "Dark", the spell-checking feature will suggest words like "Days, Dare, Darko and Dawn". To perform spell checking in Solr, the source of terms must be specified in 'solrconfig.xml' of the core being used. There are three approaches to perform spell checking which are "IndexBasedSpellChecker", "DirectSolrSpellChecker" and "FileBasedSpellChecker". [2] Since the suggestions are retrieved directly from a field in the dataset, "DirectSolrSpellChecker" is used. This needs to be specified in the requesthandler of 'solrconfig.xml' together with the field name. [2] To access these suggestions, "spellcheck" and "spellcheck.build" attributes need to be set to true in Solr API and "spellcheck.accuracy" also need to be specified. [2] In case the query is made up of multiple words, the spell-checking component will give suggestions for both individual words. The Solr API used for spell checking is shown in Figure 5.

```
http://localhost:8983/solr/movies/spell?
spellcheck.q=${query}&spellcheck=true&spellcheck.build=true&spellcheck.accuracy=0.2
```

Figure 5: Solr API for Spell Checking Feature.

Next is Synonym Filter. By default, Solr implements a synonym filter on data during the indexing process. The "solr.SynonymGraphFilterFactory" [2] class will be used for this. Users will be required to specify synonym words in the synonym.txt file of their core. Synonym filter is where when a user enters a query, that query can have different words with similar meanings. For example, if a user has entered the words "robot", "droid" and "machine" in the synonym.txt file. When a user search using the query "robot", all documents containing the words "robot", "machine" and "droid" will be retrieved.

C. *Search Engine Interface*

The interface for the search engine was developed using Next.Js, Typescript and MUI library. Next.JS is a web framework that allows users to build a web application and enables them to integrate other functionalities. Solr was integrated in this application with the help of Axios which is a promised based HTTP library. Axios was used to make API calls to Solr for document retrieval. The API used for document retrieval is shown in Figure 5 and Figure 7. The user interface of the application is shown in Figure 6.

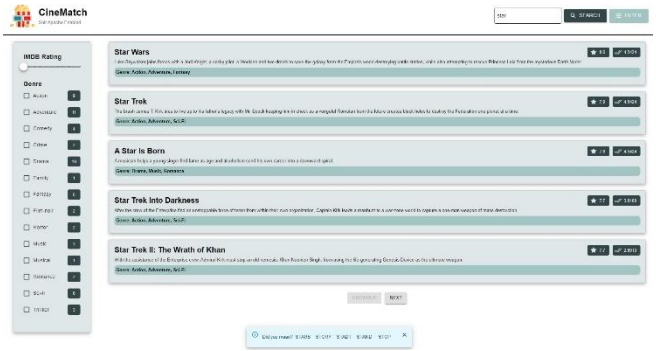

Figure 6: The UI for the CineMatch search engine.

VI. Optimization Methods

There are a total of 3 optimization methods that were implemented to optimize the search result. The optimization method implemented are pagination, faceting and reranking results.

A. *Optimization Method 1: Pagination*

Pagination is used to display all the documents retrieved from Solr in a more organized way. When dealing with a query that has only a few available documents, retrieving the complete documents is acceptable. However, if many documents are retrieved simultaneously, it becomes highly inefficient. This is called the "Deep Paging" issue. The efficient way to do document retrieval is to retrieve a small number of documents at a time. To apply pagination provided by Solr, users need to specify "start" and "row" attributes in Solr API. [2] Figure 3 shows the Solr API used for retrieving movie documents in Solr. The "start" and "row" attributes have been specified in the API. For the "start" attribute the

current page number is multiplied by 5 and the “rows” attribute is set to 5 which means 5 movie documents will be retrieved at one time.

```
http://localhost:8983/solr/movies/select?
facet.field=Genre&facet.mincount=1&facet.sort=count&facet=true&indent=true&q.op=OR&q=${solrQuery}&rq=
{!ltr%20model=EfiModelv1%20efi.whole="${query}"%20efi.separate=${query}}&fq=-Movie_Cast:(${query})AND
-Overview:movies AND -Director:(${query})&fl=*,score,[features]&start=${(currentPage - 1) * 5
}&rows=5&useParams=`
```

Figure 7: Solr API for Document Retrieval.

B. *Optimization Method 2 : Faceting*

The faceting is related to user interaction where retrieved documents are arranged into categories depending on one of the fields of indexed data. In the developed search engine, faceting was implemented on the “Genre” field. So, users will be presented with genres and the number of documents matching the “Genres”. In order to implement faceting, “facet.field”, “facet.mincount”, “facet.sort” and “facet” attributes need to be specified in the Solr API which is shown in Figure 3. [2] “Facet.field” needs to be set with the field and “facet” needs to be set to true. [2] Faceting is implemented to give users a more advanced filtering option when searching for movie documents. “Genre” field was chosen to implement faceting as it is the most common categorical attribute used for filtering movie results.

C. *Optimization Method 3[Reranking]*

The reranking results is where all the matching documents related to a query are reranked using scores from another different query. To implement reranking in Solr, the Learning To Rank (LTR) module needs to be used. [2] There are two main steps in reranking which are uploading features and uploading models.

Figure 8 shows the JSON used for Uploading Feature step in reranking. The “Series_Title” feature was given the most importance for reranking. Two features were uploaded to the reranking model which are “SeriesTitleWhole” and “SeriesTitleSeparate”. If a user enters the query “dark knight”, the “SeriesTitleWhole” feature will search for documents with “dark knight” as a whole string in the “Series_Title” field while the “SeriesTitleSeparate” feature will search for documents containing either one of the words in the “Series_Title” field. After uploading the features, the model for reranking needs to be uploaded. [2]

Figure 9 shows the JSON used for Uploading Model step in reranking. [2] In this model, the weightage for the features uploaded previously will be specified. Features with higher weightage will be given more importance. Basically, documents with the entire query present in the “Series_Title” field will be ranked first, then will be documents with either one of the words separately and followed by documents with the query present in the “Overview” field. This is done because users will always use movie titles to search for any movies. Thus, this field was given more importance for reranking. When reranking, each document will receive a new score depending on the features uploaded and the documents can be rearranged using them. The model used for reranking is Linear Model. Some of the scores for the documents will be 0 as only the “Series_Title " field is specified in the reranking model and if the query is not present as a whole string or separately in that field of the document, it will have a score of 0. Although the score is 0, these documents will still be shown in results as the query will be present in the “Overview” field of the document.

```
[
  {
    "store": "myEfiFeatureStore",
    "name": "SeriesTitleWhole",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": {
      "q": "{!field f=Series_Title}${whole}"
    }
  },
  {
    "store": "myEfiFeatureStore",
    "name": "SeriesTitleSeparate",
    "class": "org.apache.solr.ltr.feature.SolrFeature",
    "params": {
      "q": "{!field f=Series_Title}${separate}"
    }
  }
]
```

Figure 8: JSON for Uploading Feature Step in Reranking.

```
{
  "store": "myEfiFeatureStore",
  "name": "EfiModelv1",
  "class": "org.apache.solr.ltr.model.LinearModel",
  "features": [
    {
      "name": "SeriesTitleWhole"
    },
    {
      "name": "SeriesTitleSeparate"
    }
  ],
  "params": {
    "weights": {
      "SeriesTitleWhole": 0.8,
      "SeriesTitleSeparate": 0.8

    }
  }
}
```

Figure 9: JSON for Uploading Model Step in Reranking.

VII. Results and Evaluation

The movie search engine is evaluated based on the following measures which are Precision, Recall and F1-score. All these metrics are based on the confusion matrix which has True Positives, False Positives, True Negatives and False Negatives.

Precision [3] is computed by dividing the total number of movies that is relevant to the query with the total number of movies that is identified by the search engine. Recall [3] is calculated by dividing the total number of movies that is relevant to the query with the total number of movies that is relevant to the query in the dataset. F1-Score [3] is a measure that uses the Precision and Recall value that had to be calculated first. The F1-Score is calculated by taking the

product of Precision and Recall and dividing it with the sum of Precision and Recall. Then, it is multiplied with 2.

A. *Result Obtained*

The Precision, Recall and F1-Score would be calculated for each query to evaluate the search engine's in retrieving the movie information.

Query	Precision	Recall	F1-Score
dark knight	6/15	1	0.5714
dark waters	6/15	1	0.5714
robot movies	6/6	1	1
movies about robot	6/6	1	1
tum hardy	5/43	1	0.208
dikaprio	11/11	1	1

Table 2: The Precision, Recall, F1-Score for each query is computed.

B. *Result Evaluation*

The search engine's performance can be determined and evaluated after computing the Precision, Recall and F1-Score for each query.

Higher Precision and Recall value is desired since it implies that the search engine manages to obtain and display the correct and relevant movie information back to the user. The same can be said for the F1-Score as well since F1-Score is the harmonic mean of Precision and Recall. F1-Score value that is nearing to 1 is preferred since it signifies that the search engine can output the correct and relevant movie information back to the user.

For the CineMatch search engine, the Precision value for each queries varies. For example, for the query "robot movies" and "movies about robots" the precision is 1 which indicates that all the retrieved movie information is related to the query. As for the query "dark knight" and "dark waters", the precision score is slightly lower. This is because the top 6 movie has one of the words from the query in its title which makes it relevant to the query. The remainder 9 of it did not have the query words in it its title, but it did have the words in the 'Overview' section. For the query "tum hardy", the total number of retrieved movie information is 43. And the number of movies that was acted by Tom Hardy was 5. The reason why so many movies are retrieved is because there are a lot of actors who shares the same first name as Tom Hardy such as Tom Cruise, Tom Hanks, and Tom Hiddleston. Since the first name is the same, movies acted by these actors are retrieved as well. For 'dikaprio', the precision score is 1 which signifies that all the retrieved movies are acted by Leonardo DiCaprio. This is because Leonardo DiCaprio last name is unique which means that no actors have 'DiCaprio' as their last name.

The Recall score for all the queries is 1 since the False Negative (FN) value for each of the query is 0. The False Negative value for all the queries is 0 because all the movies related to the query made by the user is retrieved from the dataset.

Lastly, the F1-Score for the queries can be computed when the Precision and Recall values are determined. The query "robot movies" and "movies about robots" has the highest F1-Score value while the query "tum hardy" has the lowest F1-Score value. This indicates that the CineMatch search engine does perform well when queries related to movie genres are made. The same can be said for the query "dikaprio". Overall, the CineMatch search engine functions properly.

VIII. Conclusion

In conclusion, a search engine was developed using Solr for retrieving documents related to movies. Solr is a very efficient tool for indexing and querying data. Solr provides users with a lot of additional features and components to be added to their search engine. There were three main parts in the development of this search engine which are indexing process, querying process and optimization. For the indexing process, the "multivalue" parameter of data before indexing was set to false, pattern based tokenizer was used for tokenizing words with hyphens and phonetics matching was implemented on both index and query. Next for the querying process, spell checking and synonym filter were implemented. Furthermore, for optimization faceting, pagination and reranking were implemented. Overall, the search engine worked efficiently. Documents were retrieved accurately from Solr and other features that were implemented worked without any issues. The retrieved documents were re-ranked correctly as intended, the spell checking component gave suggestions based on the "Series_Title" field and phonetics matching helped to search for movies based on actors' name even with misspelled query. A filtering option based on IMDB Rating was also implemented which was able to filter movies without issues. As for a future plan, other additional features from Solr such as query expansion, MoreLikeThese and better reranking of documents can be added to the search engine to make it more effective. We could also use a larger dataset which includes more movies data with additional fields.

References

[1] Fujishiro, N., Otaki, Y., & Kawachi, S. (2023). Accuracy of the Sentence-BERT Semantic Search System for a Japanese Database of Closed Medical Malpractice Claims. Applied Sciences, 13(6), 4051.

[2] Apache Solr Reference Guide :: Apache Solr Reference Guide. (n.d.). https://solr.apache.org/guide/solr/9_2/

[3] Kanstrén, T. (2022, March 30). A Look at Precision, Recall, and F1-Score - Towards Data Science. *Medium.* https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec

Appendix

Code Link: https://github.com/Katheeravan305/cinematch_solr_se
Demo Presentation Slide Link:
https://www.canva.com/design/DAFj6U0HBLI/galEQY3BcO7MYQDynBp3Vg/view?utm_content=DAFj6U0HBLI&utm_campaign=designshare&utm_medium=link&utm_source=publishsharelink

```
{
  "add-field-type": {
    "name": "text_general_brac_remove",
    "class": "solr.TextField",
    "positionIncrementGap": "100",
    "multiValued": false,
    "indexAnalyzer": {
      "tokenizer": {
        "class": "solr.StandardTokenizerFactory"
      },
      "filters": [
        {
          "class": "solr.StopFilterFactory",
          "words": "stopwords.txt",
          "ignoreCase": true
        },
        {
          "class": "solr.LowerCaseFilterFactory"
        }
      ],
      "charFilters": [
        {
          "class": "solr.PatternReplaceCharFilterFactory",
          "pattern": "[\\[\\]]",
          "replacement": ""
        }
      ]
    },
    "queryAnalyzer": {
      "tokenizer": {
        "class": "solr.StandardTokenizerFactory"
      },
      "filters": [
        {
          "class": "solr.StopFilterFactory",
          "words": "stopwords.txt",
          "ignoreCase": true
        },
        {
          "class": "solr.SynonymGraphFilterFactory",
          "ignoreCase": true,
          "expand": true,
          "synonyms": "synonyms.txt"
        },
        {
          "class": "solr.LowerCaseFilterFactory"
        }
      ],
      "charFilters": [
        {
          "class": "solr.PatternReplaceCharFilterFactory",
          "pattern": "[\\[\\]]",
          "replacement": ""
        }
      ]
    }
  }
}
```

Figure 1: The JSON script for the square bracket removal.

Figure 2: The JSON script for setting the new tokenizer that is used for the hyphenated word.

```
{
  "add-field-type": {
    "name": "moviecastver1",
    "class": "solr.TextField",
    "positionIncrementGap": "100",
    "multiValued": true,
    "indexAnalyzer": {
      "tokenizer": {
        "class": "solr.StandardTokenizerFactory"
      },
      "filters": [
        {
          "class": "solr.StopFilterFactory",
          "words": "stopwords.txt",
          "ignoreCase": true
        },
        {
          "class": "solr.LowerCaseFilterFactory"
        },
        {
          "class": "solr.BeiderMorseFilterFactory",
          "nameType": "GENERIC",
          "ruleType": "APPROX",
          "concat": true,
          "languageSet": "auto"
        }
      ]
    },
    "queryAnalyzer": {
      "tokenizer": {
        "class": "solr.StandardTokenizerFactory"
      },
      "filters": [
        {
          "class": "solr.StopFilterFactory",
          "words": "stopwords.txt",
          "ignoreCase": true
        },
        {
          "class": "solr.SynonymGraphFilterFactory",
          "ignoreCase": true,
          "expand": true,
          "synonyms": "synonyms.txt"
        },
        {
          "class": "solr.LowerCaseFilterFactory"
        },
        {
          "class": "solr.BeiderMorseFilterFactory",
          "nameType": "GENERIC",
          "ruleType": "APPROX",
          "concat": true,
          "languageSet": "auto"
        }
      ]
    }
  }
}
```

Figure 3: The JSON script that adds the phonetics matching.

Figure 4: The JSON script used to replace the field type of a field.

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 10,
    "params": {
      "q": "500",
      "facet.field": "Genre",
      "indent": "true",
      "fl": "*,score,[features]",
      "q.op": "OR",
      "facet.mincount": "1",
      "fq": "-Movie_Cast:(500)AND -Overview:movies AND -Director:(500)",
      "facet": "true",
      "facet.sort": "count",
      "rq": "{!ltr model=EfiModelv1 efi.whole=\"500\" efi.separate=500}"
    }
  },
  "response": {
    "numFound": 1,
    "start": 0,
    "maxScore": 2.9376981,
    "numFoundExact": true,
    "docs": [
      {
        "Poster_Link": [
          "https://m.media-
amazon.com/images/M/MV5BMTk5MjM4OTU1OV5BMl5BanBnXkFtZTcwODkzNDIzMw@@._V1_UX67_CR0,0,67,98_AL_.jpg"
        ],
        "Series_Title": "(500) Days of Summer",
        "Released_Year": [
          2009
        ],
        "Certificate": [
          "UA"
        ],
        "Runtime": [
          "95 min"
        ],
        "Genre": [
          "Comedy, Drama, Romance"
        ],
        "IMDB_Rating": [
          7.7
        ],
        "Overview": "An offbeat romantic comedy about a woman who doesn't believe true love exists, and
the young man who falls for her.",
        "Meta_score": [
          76
        ],
        "Director": [
          "Marc Webb"
        ],
        "No_of_Votes": [
          472242
        ],
        "Gross": [
          32391374
        ],
        "Movie_Cast": [
          "Zooey Deschanel, Joseph Gordon-Levitt, Geoffrey Arend, Chloë Grace Moretz"
        ],
        "id": "e59d8839-ee8a-4982-9cb2-67a98f67ffae",
        "_version_": 1766766160675602440,
        "score": 4.163464,
        "[features]": "SeriesTitleWhole=2.602165,SeriesTitleSeparate=2.602165"
      }
    ]
  },
  "facet_counts": {
    "facet_queries": {},
    "facet_fields": {
      "Genre": [
        "comedy",
        1,
        "drama",
        1,
        "romance",
        1
      ]
    },
    "facet_ranges": {},
    "facet_intervals": {},
    "facet_heatmaps": {}
  }
}
```


Figure 10: Response of Facet API for the Query "500".

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 100
  },
  "command": "build",
  "response": {
    "numFound": 0,
    "start": 0,
    "numFoundExact": true,
    "docs": []
  },
  "spellcheck": {
    "suggestions": [
      "dark",
      {
        "numFound": 5,
        "startOffset": 0,
        "endOffset": 4,
        "origFreq": 6,
        "suggestion": [
          {
            "word": "dare",
            "freq": 1
          },
          {
            "word": "darko",
            "freq": 1
          },
          {
            "word": "days",
            "freq": 4
          },
          {
            "word": "dawn",
            "freq": 2
          },
          {
            "word": "d'arc",
            "freq": 1
          }
        ]
      }
    ],
    "correctlySpelled": false,
    "collations": [
      "collation",
      {
        "collationQuery": "dare",
        "hits": 1,
        "misspellingsAndCorrections": [
          "dark",
          "dare"
        ]
      },
      "collation",
      {
        "collationQuery": "darko",
        "hits": 1,
        "misspellingsAndCorrections": [
          "dark",
          "darko"
        ]
      },
      "collation",
      {
        "collationQuery": "days",
        "hits": 18,
        "misspellingsAndCorrections": [
          "dark",
          "days"
        ]
      },
      "collation",
      {
        "collationQuery": "dawn",
        "hits": 2,
        "misspellingsAndCorrections": [
          "dark",
          "dawn"
        ]
      },
      "collation",
      {
        "collationQuery": "d'arc",
        "hits": 1,
        "misspellingsAndCorrections": [
          "dark",
          "d'arc"
        ]
      }
    ]
  }
}
```


Figure 11: Response of Spell Check API for the Query "dark".