KindleMeSome: Amazon Kindle Reviews

António Ribeiro
FEUP
up201906761@edu.fe.up.pt

Diogo Maia
FEUP
up201904974@edu.fe.up.pt

Luís Viegas
FEUP
up201904979@edu.fe.up.pt

ABSTRACT

This paper addresses the implementation of a books and reviews search engine, with information collected from Amazon's Kindle book store. The data treatment pipeline employed the Pandas data analysis library and the search engine was set up with the Apache Solr platform, which supplies information retrieval resources and tools. We set up the indexing schema and constructed queries that we felt were appropriate for the dataset at hand. We then applied evaluation techniques to our results, analysed the results, detailing and discussing our conclusions. Unnoticed errors in this stage were later corrected and documented. In order to expand our horizons and explore more of what Solr has to offer, a small and simple webpage was created. This front-end used other tools the platform has at its disposition, whose main purpose is improving the UX of common tasks, such as searching a book title.

1 INTRODUCTION

Amazon [1] is one of the most well-known and reputable e-commerce and web services providers, founded by Jeff Bezos in 1994 and currently valued at 1.22T dollars. The online retailer was initially solely focused on selling books (dubbed the "Earth's biggest bookstore" by its founder [2]), fuelled by a lack of platforms for this purpose. The Kindle is an e-book reading device made by Amazon, that allows users to read their saved Amazon Kindle store books on the go.

This paper describes the development of a search engine, capable of handling user-provided queries. To achieve this objective we had to carefully clean, analyze and study the obtained dataset (done through the panda library).

The search engine was assembled in the Apache Solr information retrieval platform [2]. It can be described as a search engine, made in Java, that aims to optimize the search of terms or other forms of information in a given document collection, stored in CSV, JSON, etc. It is responsible for indexing and storing that information, having at our disposal multiple tools for indexing, ordering, text parsing/-analysis and transformation, querying, and full-text search (the Lucene engine). It is similar to ElasticSearch, offering more complex and advanced features.

Apache Solr, along with ElasticSearch are possibly the two most known open-source search engines. They offer very similar features and interfaces, with very close performance metrics (as far as we know). Solr was eventually our choice, as a consequence of this Curricular Unit provided resources. It is the tool referenced in the tutorials that kick-started the setup process in our machines (with a selection of commands and a docker image), often the most boring and troublesome stage. However, further research demonstrates that ElasticSearch has much better documentation, a setback faced several times in the semester while using Solr.

2 DATASET

The following sections address data treatment/processing, that resulted in a more comprehensible dataset and its posterior characterization.

2.1 Description

The dataset is composed of 3 JSON files, supplied by professor Jianmo Ni [3]. We first learned about the datasets from a Kaggle post, that referenced the website. As stated, the data is freely available for research purposes (as long as we cite the paper in question, [3]). Filling out the requested google form, allows anyone to access the complete datasets on the website. In this case, the reviews (from 2014) and metadata from the kindle book store (2014 and 2018). The reviews' dataset is already condensed to a subset of the data in which all users and items have at least 5 reviews (according to the owner).

Our train of thought was connecting these datasets to get each review mapped onto a book, resorting to the official store identifier, the asin code [4].

2.2 Data Volume

File	Size (MB)	no of objects
meta_Kindle_Store_2014.json	497.4	434.702
meta_Kindle_Store_2018.json	589.4	493.859
Kindle_Store_5.json	827.8	982.619

Table 1: Dataset volume.

2.3 Iteration Steps

The following sections describe the pipeline steps executed until the final data collection is reached, illustrated by the diagram (1).

[1] www.amazon.com
[2] https://solr.apache.org/
[3] http://deepyeti.ucsd.edu/jianmo/amazon/index.html
[4] https://www.oreilly.com/library/view/amazon-hacks/0596005423/ch01s03.html

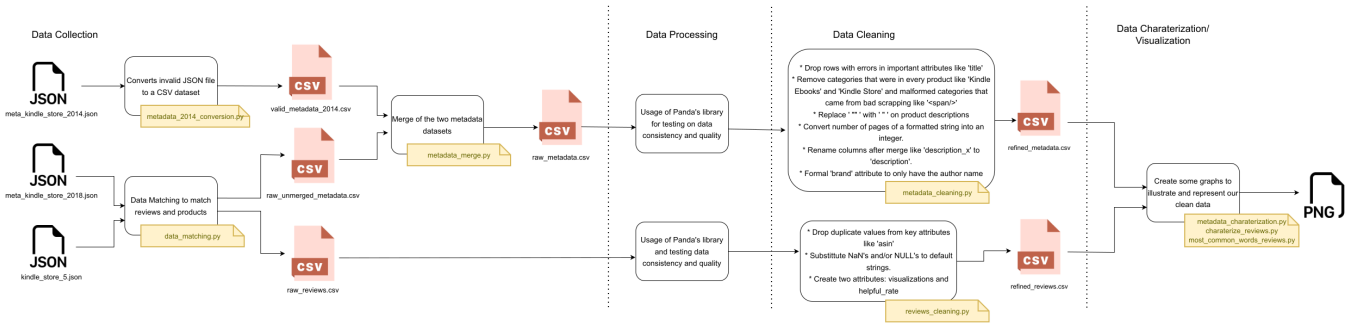

Figure 1: Data Flow Diagram.

2.3.1 Collection. As mentioned, our datasets were provided from an online source, hosted by a UCSD professor. We made this decision since they were relatively big (which, in theory, would translate to a greater diversity), and no web-scrapping steps were required. They came in JSON format, frequently used in API requests, which was probably how these were obtained.

At first glance, the metadata collection (2018) is complete in terms of product information however, several issues arose, which we will later discuss in the cleaning portion of this section.

We found two of the most important attributes (later discussed in 2.3.2) were missing. Searching for other, more complete datasets, we came across the same 2014 metadata, which is very similar to the one we described above but has the fields we need. We couldn't simply exchange them, since the title attribute was not present, so we decided to maintain the 2018 metadata source and extract the description, price and the image from the old dataset (the book cover and the description, did not radically change between the four years). Thanks to the high amount of 'asin' matches between the two (31889), it was not a setback.

On the other hand, the Review object is generally clean and simple, containing the necessary information for our objective.

2.3.2 Processing. This phase corresponded to the data processing, matching and manipulation, using the Pandas library.

Handling the 2014 metadata file was not as straightforward as we would like. The JSON objects had been stored in an incorrect format. The description attribute (one of the three we needed) was, sometimes, encapsulated in double quotes ("), and others in single quotes ('). A simple regex substitution proved to be very hard since the field is text rich ('I'm Thor', would end up tampered as "I"m Thor"). We ended up using Python's native supported *ast.literal_eval* to extract a dictionary, and then a JSON conversion, line by line.

To prune the global database for the rest of the pipeline, we needed to join the products (from each year) and reviews, so that we can reduce the total data to the ones that match all 3 datasets. We then perform 2 actions:

- Extract the description, the price and the cover image url from the 2014 file, and add them to each product (in the 2018 set), pruning all that didn't appear in both.
- Drop the reviews that didn't match any of the products left from the previous action and vice versa.

At this point, 2 CSV files were generated. We then performed the following steps:

- Extracted useful information from the 'details' inner JSON object (file size, language, publication date and publisher, asin), converting them into more useful formats, such as int for the file size, instead of string.
- Renamed some columns that were not very explicit in terms of description (e.g. brand is the authors name, print length is the number of pages).
- Added a numeric rating evaluation to each book, based on the reviews matched in the other CSV file (according to the overall attribute). The same was done to each review, wrangling the 'helpful' array into two different attributes.

2.3.3 Cleaning. The metadata file proved to be the hardest to clean: most of the values of the dataset were missing (tech1, tech2 et al.), some title attributes left unfilled.

At first glance, the metadata collection (2018) is very complete in terms of product information, although the selection included some examples of *dirty data*:

- Some fields were left unfilled in all lines, e.g. tech1/tech2, similar_item and in times, even the title.
- A lot of redundant data, e.g. "Kindle Store", "Kindle eBooks" were always mentioned as two of the book's categories, asin identifier was present twice (as an attribute in the main object, and inside the details object), the publication date was written in its own column and in the publisher name. Although one of them was left blank, it included two date keys.
- Tainted data: the category array was mishandled, and included the html tag. The rank string,

ended with the suffix "Paid in Kindle Store (" for all occurrences.
- Unnecessary repeated information: the main_cat string was the same for all lines of the dataset, the same is equally true for the Simultaneous Device Usage column.

Steps taken to ensure a cleaner dataset:

- Removal of duplicate asins in the metadata files (keeping the first record), resulting in a mandatory 1-to-many multiplicity.
- Remove the missing data from key attributes.
- Removed all elements that we did not consider useful for the project's objective (main_cat, device usage, file size, word-wise, also bought/viewed, ISBN, etc), leaving us with the fields specified in the table **??**.
- Dropped the details column, after extracting some fields inside the column's object.
- Removed all categories, except the 3rd index in the array (which included the true book category, e.g. "Business & Money").
- Attributes more easily understood in a different type were converted (e.g. "272 pages" string, to the integer 272).

On a side note, we opted to leave both date columns (reviewTime and unixreviewTime), for more straightforward queries. In case we just need the formatted date string (for display), we take reviewTime. Else, if the query requires date calculations, we will use unixreviewTime. In our perspective, the speed performance of the queries is more crucial than memory requirements.

The review file cleaning process included steps such as adding an "Unnamed reviewer" string to reviewerName's NA values and dropping review duplicates.

Field	Meaning
asin	product's store identifier
title	book title
author	author's first and last name
description	small book content description
publication date	date of debut in string format
no of pages	length of the book
price	book price at the time the data was collected
category	category according to Kindle's store sections
ratings avg	average product's rating
imgURL	url that references the amazon's book cover image

Table 2: Metadata fields and their contextual meaning.

Field	Meaning
reviewerID	user/reviewer amazon account id
asin	reviewed book identifier
visualizations	number of users who viewed the review
helpful_ratio	ration of users who found the comment helpful
reviewerName	the reviewer's amazon nickname
reviewText	the text review
overall	rating given by the user (0-5)
summary	synopsis of the review
unixReviewTime	review issue date (unix time stamp format, since epoch)
reviewTime	string formatted review date

Table 3: Review fields and their contextual meaning.

2.4 Conceptual Model UML

The diagram 2.4 illustrates our conceptual model. The meaning of the attributes is described above (tables 2 and 3). The relationship between reviews and books is an composition (since no reviews exist without their books) and a simple many to one relationship for reviewers. One fact worth noting is the *5..** multiciplicity, from what was mentioned in 2.1.

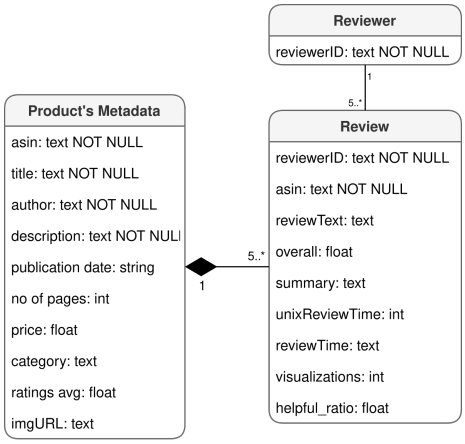

Figure 2: Conceptual diagram of the final database.

2.5 Data Characterization

The following sections describe how the database is populated, presenting each type of document, frequent patterns and other observations.

2.5.1 Reviews. Due to the fact that our project focus, to get the general knowledge on how to apply them in future search query results. One statistical analysis we made was applying word vectorization and determining the most used words (excluding common stopwords, such as "book", "title", "story", "author":

Word	Occurrences
love	45650
great	34561
time	25677
short	24756
enjoyed	16265
life	13547
romance	12460
written	11245
plot	5423
found	1003

Table 4: 10 most used words, in order.

This demonstrates a big advantage, given that we can use this type of words/contexts to infer costumers' satisfaction, having read the book for recommendations.

Meaning	Value
Avg review grade	4.34
Avg review word count	604

Table 5: Useful reviews statistics.

2.5.2 Metadata. The 15 most published authors are presented below(4).

We find that Literature and Fiction is the most common category in all of the store, which means given a query we are most likely to offer a book that belongs to this category. 5

We couldn't find any correlation between the average book rating, and its corresponding price. On the other hand, we noticed a clear tendency of the overall population to give 5 star review. This could create a biased result when we factor into account the rating while answering to queries. 6

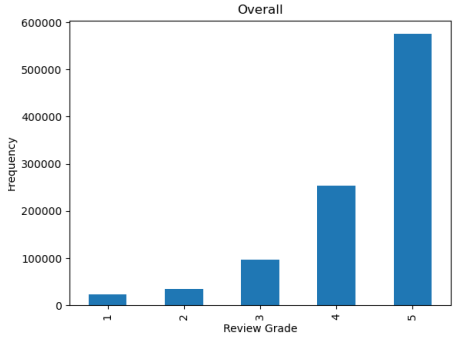

Figure 3: Frequency of overall ratings in the reviews dataset.

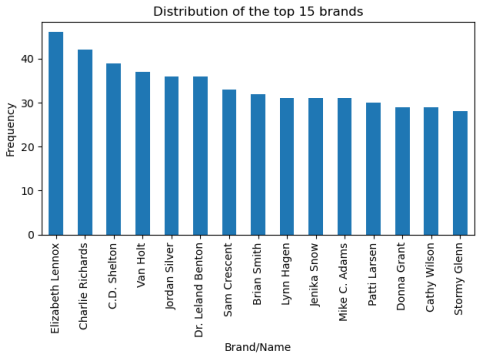

Figure 4: Distribution of authors' publications (Top 10).

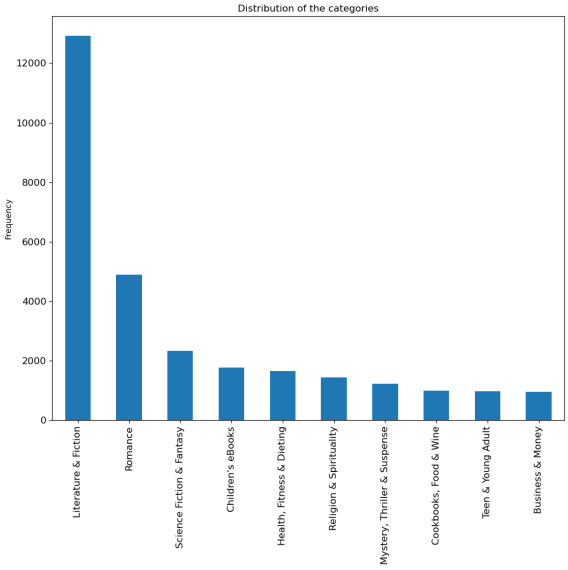

Figure 5: Distribution of categories (Top 10).

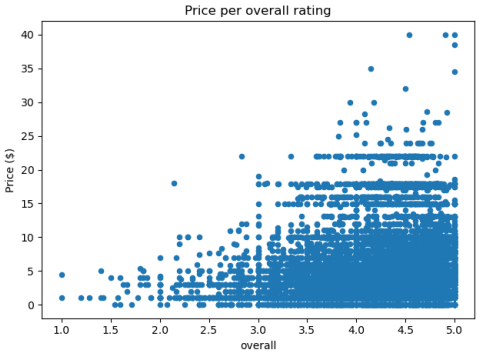

Figure 6: Price by average book rating (overall).

2.6 Search Scenarios

Search scenarios were oriented to provide interesting results in terms of book content and global appreciation. Thus,

they lean on the reviews each book has and the readers' opinion.

2.6.1 Possible query scenarios.

- Search the reviews of a given reviewer, with a time span interval.
- Search books in a specific category, and filter/sort by reviews rating.
- Search for a "light"/"heavy" book (or other equivalent colloquial synonyms).
- Search books from a given author, with a tie span interval.
- Filter reviews by ranking.

3 M1 IMPROVEMENTS

The following improvements were implemented in the previous stage of the process:

- *Makefile* refactor and improvement.
- Rearranged the Reviewer as a table in the Conceptual design (2.4).
- Small changes to the scripts we used for the previous milestone.
- Report alterations in accordance to the teacher's feedback.

4 COLLECTION + INDEXING

The following sections describe the collection and indexing stage of the developed platform, from the required changes to the documents to the final schema and analyzer layout.

All the processes described from now on will be about the SolR platform tool.

4.1 Defining documents

At the end of the last pipeline checkpoint (1), we were left with 2 csv files. To accommodate the file structure and attributes to the SolR, the following alterations had to be made:

- Converted both files back to json, since this format is more recommended for SolR.
- Id columns were renamed (in the case of the book's asin) and an index (simple ordinal identifier) was appended to the reviews table.
- Dates were converted to the following format. *YYYY-MM-DDTh:m:sZ*, more easily supported for calculations.

4.2 Indexing process

Originally, two cores were created for the indexing. This proved to be a bad decision (joining required a fromIndex attribute, which lead to some difficulties in joining the tables and was not recommended for efficiency). We opted to include both types of documents in the same kindle core, sharing the same schema and differentiated by a *type* attribute for filtering results. However, the join query filter still has to exist for querying relationship data. This vastly limits our query possibilities, for example, we can't gather the reviews of a specific book, for which we searched a term in the *q* field. No reviews can be returned in a book search. A workaround was reached by filtering the book title in a join filter of a review search, but no special treatment (boost, tokenizer, etc) is given to the term we are searching.

4.3 Indexed Files + Processing

Solr supplies a vast amount of Tokenizers and filters, used in both indexing and query operations. Tokenizers split the original string into smaller sections (following a rule set), and filters continuously parse the resulting tokens list, in a pipeline fashion. Our selection is illustrated in the following table 6, as well as the field type they pertain to:

Field Type (tokenizer)	Filters
text StandardTokenizerFactory	ASCIIFoldingFilterFactory LowerCaseFilterFactory
body ClassicTokenizerFactory	ASCIIFoldingFilterFactory LowerCaseFilterFactory ClassicFilterFactory StopFilterFactory PorterStemFilterFactory SynonymGraphFilterFactory RemoveDuplicatesFilterFactory FlattenGraphFilterFactory

Table 6: Field types and their Tokenizers and filters

The same collection of tokenizers and filters were chosen for the indexAnalyzer and queryAnalyzer sections, in both types.

4.3.1 Text. The text field type describes a generic text passage attribute in which we can search for terms and quotes (either an exact match or an included term), but no additional treatment is required, and if implemented would be computationally expensive for our objective. The indexing/-querying starts with ClassicTokenizer, which behaves like a StandardTokenizer with some extra rules about hyphens, period splitting, and internet domain matching.

After that, we apply a simple ASCIIFolding and LowerCase filter combination, for generic substitution of special characters to their ASCII equivalents (e.g. à to a), which could damage query results and enable the search for lower and upper case terms:

4.3.2 Body. On other hand, the body type describes a text-rich field, where we found that some type of language processing would be in some way advantageous to accomplish query tasks.

Our main goal was to expand the available vocabulary, linking it with synonym equivalents. Up untill the filter, all steps work to achieve a neutral baseline to more precisely match these synonyms. Previous data analysis demonstrated that the vast amount of included text is in English, so most of these steps are applied exclusively to this language.

- A classic filter, removing possessives *'s* and acronyms' periods.
- A stop word filter, from the most common used in the english language [5].
- A Stemming algorithm for suffix removal (in this case Porter's algorithm [5]). Words such as loved, loving, love are left as their root stem lov.

The synonym stem mapping selects a set of words (chosen according to the context of multiple queries), replaces them with an appropriate term, and finally a duplicate removal. In this case, we are aware that removing duplicates could limit term frequency matching, but it appears solr uses different names for synonym substitute types.

4.4 Schema

As previously mentioned, the schema encompasses both types of documents, books, and their reviews. Besides including all available attributes, there were trade-offs and additions: a *type* attribute now describes which document type we want to retrieve (review or book). The overall remained as float, albeit an integer is more suited for the overall field in reviews. Text which is not indexed can be fetched as a keyword or that won't take part in a complex full-text search, is stored as a simple default string. The asin attribute is also declared as stored and multiValued.

[5]https://countwordsfree.com/stopwords

type	name	indexed
text	reviewerName	true
	brand	true
	category	true
	title	true
body	description	false
	reviewText	true
	summary_ratio	true
string	type	true
	imgUrl	false
	asin	true
	reviewerID	false
pdate	publication_date	true
	reviewTime	false
pfloat	price	false
	overall	true
	helpful_ratio	true
pint	no_pages	true
	visualization	false

Table 7: Document attributes, types, and indexing status

5 RETRIEVAL

The following sections explain our thought formulation in fabricating the queries, implementing and optimizing them, and concluding with how well they matched our expectations and fulfillment of the objective. Seeing that our document collection is split into two separate types, we tried to orient the queries separately. When possible (e.g. 5.3), we took advantage of the implicit data relationship they have with one another, including it in a "workaround" filter.

The first step was brainstorming a list of possible information needs a user could have, (having as a baseline the ones mentioned previously in the report 2.6). After that, we pruned the options by analyzing how feasible and complex they were and how many results they would generically produce, in terms of quantity and quality. We then modeled them into the SolR search fields and fined tuned them to our specifications.

The first query used the *Lucene* search type given its simplicity in the q field. The following were performed in *DisMax*, considering the vast amount of customizable fields, in which we can include different boosts and other filters.

5.1 Query 1 - Top 10 books written by "Francis"

In this query, we are trying to retrieve books written by all authors having Francis embedded in their names. The query searched for the term "Francis" in the brand, to match any first, middle, or last name. Sorting ensures the top 10 ten books, overall.

5.1.1 Query Specifics.

- *q*: "brand: francis"
- *q,op*: OR
- *sort*: overall desc
- *rows*: 10

5.1.2 Conclusions. This query works as our simplest query, serving as a foundation to evaluate the different systems. From the results in figure 8, we see that all systems behaved exactly the same. However, this is a simple query and we will not exaggerate extrapolating results.

5.1.3 Example. The top 10 documents retrieved are on table 9. This query retrieved 16 documents on total.

5.2 Query 2 - Small soccer and football books.

In this query, we are trying to retrieve small books that discuss football (or soccer in American English). We consider a book to be small if it has less than 150 pages. To check if a book discusses football, we searched for "football" and "soccer" in the title, brand and description.

The boosted query has the following boosts:

- Field boost of 2x on title

5.2.1 Query Specifics.

- *q*: soccer football
- *q,op*: OR
- *qf*: title brand description
- *fq*: no-pages:[0 TO 150]

5.2.2 Conclusions. In this query, we used defType "edismax" which was then used during the following queries. This allowed us to filter for the number of pages, for example. From the results in figure 9 we see that all systems behaved similarly.

5.2.3 Example. The top 10 documents retrieved are on table 10. This query retrieved 14 documents on total.

5.3 Query 3 - Positive reviews from books of the author Dr. Leland Benton

In this query, we are trying to retrieve positive reviews from the books of Dr. Leland Benton. The query term to be searched in is "good", taking advantage of the synonyms filter on the custom field type. We thought about including "book" as a required close term but after searching for the reviews, most of the most prominent reviews didn't have that word. We needed to join the reviews and the books to know which reviews were from the wanted author.

To find the more relevant positive reviews, we had a filter that only accepted reviews with an overall more than 4.0. We sorted the reviews by visualizations and helpful rate in a descending order.

The boosted query has the following boosts:

- Field boost of 2x on summary

5.3.1 Query Specifics.

- *q*: "good"
- *q,op*: OR
- *fq*: {!join from=id to=asin}brand:"Dr. Leland Benton" & overall:[4.0 TO *]
- *sort*: helpful_rate desc, visualization desc
- *fl*: reviewText summary

5.3.2 Conclusions. We chose this query to show how we can join (this join works as an inner join) different structured documents of the same core (something called by the SolR documentation as *Combined Documents*) and filter the desired documents by their parent document's attributes. The downside of this join operator is that it can only be used on the *filter query* option, which does not operate with the other SolR query parsers like *lucene* or *dismax* so we cannot use query operators and functions to enable fuzziness or wildcards on the author name. After this discovery, we restrained from using this kind of join, since they were not of very use.

5.3.3 Example. The top 10 documents retrieved are on table 11. This query retrieved 199 documents on total.

5.4 Query 4 - Reviews recommending books for vacations done in vacation period of 2013

In this query, we are trying to retrieve the reviews that recommend a book to be read during summer/winter vacations, depending on the hemisphere. The text we will use as query terms is "vacation read" since we are using the synonyms and stem filter, this searches a lot of expressions like "holidays" and "trip". In this query, we also used a proximity search of degree one to allow us to get more results. We also, during data characterization, noticed that it was a common practice of reviewers to recommend books for vacation purposes during vacation times. The vacation time we used to filter the search was from 1-06-2013 to 31-08-2013. To find the more relevant positive reviews, we had a filter to only accept reviews with an overall more than 4.0, an overall we consider positive, and sorted the reviews by visualizations and helpful rate in descending order.

The boosted query has the following boosts:

- Field boost of 2x on summary

5.4.1 Query Specifics.

- *q*: "vacation read"
- *q,op*: OR
- *fq*: type:"review" & reviewDate:[2013-06-01T00:00:00Z TO 2013-09-31T00:00:00Z]
- *sort*: helpful_rate desc, visualization desc
- *qs*: 1
- *fl*: reviewText summary

5.4.2 Conclusions. This query highlights our usage of the proximity search. We thought about using more than 1 on the proximity field, but we noticed that it only would add more noise to our search and make the top searches more irrelevant. We wanted to include every vacation period from 1992 to 2013 in this query filter. However, we discovered we could only use ranges, starting and ending at a certain date. It made our query more simple, but no alternative was found.

5.4.3 Example. The top 10 documents retrieved are on table 12. This query retrieved 540 documents on total.

5.5 Query 5 - Reviews that depict the book as boring, difficult to read or long

In this query, we are trying to retrieve reviews that characterize a book as too boring to read, difficult, or simply too long to finish. We will use the terms "dull", "tough" and "endless". According to our synonym mapping filter, these are substituted by boring (the more colloquial term for dull), difficult and long, in their respective order, matching with the already parsed strings. We search them in the reviewText and summary attributes. The summary, as the name says, reinforces the reader's mood, most of the time even more dramatically than the reviewText, which tends to be more rational and thoughtful.

To retrieve the absolute worst reviews, we filtered the result by selecting only the ones up to the 2.0 value rating.

The boosted query has the following boosts:

- query becomes dull $\hat{4}$ tough $\hat{2}$ endless.
- query field (*qf*) summary with 1.5 boost.
- boost function (*bf*) using *termfreq* for "not" in the reviewText field.
- phrase field (*pf*) boost reviewText.
- phrase field slop (*ps*) boost 10.

The objective of this boosts is to enforce or emphasise the idea of the original query. The terms we search for are ordered by (our subjective) relevance. A boring book is much worse than a long book, the latter can be done with pleasure, if it is a good and entertaining book. The phrase field boost increases the score of reviewTexts where the 3 terms appear spaced by 10 tokens. This was not done to the summary, since it is usually much smaller in length and therefore rarelly includes all the terms. Finally a termfreq counts and boosts the score of the reviewText with the not word in it. It is a crude attempt of semantic analysis, finding negative context segments included in the sentences, for example, "not finish" or "not good".

5.5.1 Query Specifics.

- *q*: dull difficult endless
- *q.op* OR
- *fq* type:review, overall:[* TO 2.0]
- *fl*: id summary reviewText asin overall

5.5.2 Conclusions. The original approach included a sorting query field on the overall attribute, but somehow it removed the boosting termfreq aspects, overwriting them in the final presented order. The termfreq boost is not a recommended or optimal semantic analyzer, but since we verified a large number of negative reviews, that included the "not <adjective>" characterization, we opted to leave it. The phrase boost made an almost unnoticeable difference to the results since the original query arrangement already returned and ordered the reviewTexts that matched this criterion.

5.5.3 Example. The top 10 documents retrieved are on table 13. This query retrieved 6894 documents on total.

6 EVALUATION

Evaluating the documents a query retrieves is an important part of creating a good search engine. Everyone wants a good search engine to return the most relevant documents and to make sure it happens, it is important to evaluate with what consistency the queries we developed, do it.

The first thing we needed to create to be able to evaluate our queries is to identify their *qrel*. A *qrel* is the collection of the most relevant documents a query should return. These are collected from all the documents in the collection and require a data understanding of the database.

In our case, we needed to pick these by hand from our datasets. This task not easy to accomplish when we are talking about more than thirty thousand reviews and we needed to go one by one to check if they are relevant. We decided that in queries where the results would be ampler, we would use a sample of our dataset, where going one by one would be possible. In other cases, we searched on the *CSV* representation of the dataset for words/expressions/values we believe/knew would lead us to the reviews/products we wanted and added those to the *qrel* file of the query.

After defining the *qrel* of each query, we decided the parameter we would use. Those parameters were:

- *P@10* - Precision of the first 10 documents retrieved
- *AvP* - Average precision
- *Precision-Recall curves*

In the end, we decided to evaluate three scenarios to check how our changes improve the search results relevance:

(1) Scenario where we have a schema where all the fields have a default SolR type;
(2) Scenario where we use our custom schema but the queries are without boost;
(3) Scenario where we use our custom schema and the queries are boosted.

6.1 Evaluation Results and Discussion

In this section, we will reveal the results of our evaluation and discuss what we think about them.

The following table 8 has the results of P@10 and AvP of our queries on the 3 different scenarios.

In these queries, we have certain situations:

- A flat line with precision on 1: This indicates such query retrieved fewer documents than the total number of relevant docs and all of them were relevant
- A flat line with precision on 0: This indicates such query retrieved 0 relevant queries.
- Precision starting in a high value like 1 and decaying to a low value when reaching a recall value of 1: This is the standard behavior of a precision-recall curve and the area beneath the curve is a good indicator of performance. The bigger the area, the better the performance.

With these points in mind we can look query by query and check how our improvements on the filterless scenario improved our results:

- *Query 1* (fig 8): On query 1, we always get almost max precision through the different scenarios.
- *Query 2* (fig 9): From 0 to 0.3 recall, the situation persists for all 3 scenarios, however, after 0.3 recall, the area of scenario 3 is higher than the area of scenario 2 and scenario 1. It allows us to conclude that the filters and boosts positively affected query performance.
- *Query 3* (fig 10): On query 3, we have a standard curve on the three scenarios. However, we can check that the area of scenario 3 is higher than the area of scenario 2 and scenario 1. We can check that the filters and boosts improved this query performance.
- *Query 4* (fig 11): On query 4, we have S-shaped curve with a very high precision on the filterless and an similiar curve on the other two scenarios. With this, we can assume the filters were essential to get the majority of the results since the P@10 is 0.4, a small number. However, the boosts were ineffective, since the curve and metrics were the same.
- *Query 5* (fig 12): On query 5, we find discrepancies in the search scenarios. On the one hand, the filterless and the unboosted queries performed poorly, considering the small area under the curve. The score of the mentioned scenarios is very bad for both precision and recall across all thresholds. They start with a very small precision value, even for lower recalls, meaning all data is irrelevant and misidentified. Boosting improved the initial threshold values, the precision starts at a good point but decreases rapidly, after reaching a recall value of 0.2, implying a small enhancement in retrieving good results. For greater recalls, all cases stagnate, which means that no matter how many more relevant documents are returned (turning false negatives into retrieved true positives, [1]), more irrelevant documents are appended in comparison (false positives), therefore precision stays at around 0.4-0.5.

Query	**Scenario 1**		**Scenario 2**		**Scenario 3**	
	AvP	P@10	AvP	P@10	AvP	P@10
1	0.94	0.80	0.97	0.90	1.0	1.0
2	0.60	0.40	0.74	0.6	0.94	0.90
3	0.59	0.40	0.74	0.70	0.78	0.70
4	0.76	0.70	0.64	0.60	0.51	0.40
5	0.20	0.30	0.30	0.4	0.57	0.6

Table 8: Average precision and P@10 queries scores.

7 IMPROVEMENTS OF THE SEARCH SYSTEM

The work progress we laid out in the previous sections already includes most of the features Solr provides relative to term searching (such as boosts and filters). Our goal in this stage was to correct unnoticed errors in the previous delivery, improve results and further explore Solr tools and possibilities.

7.1 Interface

Most of the extra features Solr offers are destined for real-life implementation of a search engine, helping the user to choose adequate results for their search intentions (**??**). To show the diversity of these options, we opted to create a simple GUI page, in which the user can search for book titles.

7.1.1 Front-end. The front end was done using the React library [6] and Axios for HTTP requests. It displays a single page that includes: a search bar, a filter collection, pagination buttons, and finally the book display section. In this section, information stored in the kindle core is placed in each book card. The ASIN reference (the book's index id), can be used to retrieve the actual Amazon website for the product, [4]. The store has a built-in GET URL path (i.e. http://www.amazon.com/exec/obidos/ASIN/B00BHUBHMW), that allows us to redirect the user, once the book card is *clicked*.

7.1.2 Back-end. Initially, we planned on making the requests exclusively from the front end (loaded webpage). However, a CORS [7] protection in the Solr running service prevented this type of request. Several unofficial (changing the web.xml file [8]) and official solutions (whitelisting the IPs [9]) were tested to circumvent this scenario, but revealed to complex or unfruitful.

[6] https://reactjs.org/
[7] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[8] https://laurenthinoul.com/how-to-enable-cors-in-solr/
[9] https://solr.apache.org/guide/8_10/securing-solr.html

The simplest solution was implementing a proxy server (pseudo API) that forwards URL requests to Solr. The front end creates the desired URL and sends it to the intermediary wrapped in a POST request, listening for the answer that includes the results.

7.2 Facets/Filters

The facets query filters can be described as:

> the dynamic clustering of items or search results into categories that let users drill into search results (or even skip searching entirely) by any value in any field. Each facet displayed also shows the number of hits within the search that match that category. [6]

In our case, we chose to elect the 'category' field as the filter option. We realized it is the most useful and has the most intuitive and interesting outcomes. The facet command is introduced with a *"facet=true"* parameter in the URL regularly used to make the search (and even specialized with *facet.field=category*. This request then returns all categories available in the database and the number of results from the original query that fall into each "bucket":

```
"facet_counts": {
    "facet_fields": {
        "category": ["Fiction", 45....]
    }}
```

We list them as filter options (check boxes), that the users can pick from to narrow down their searches, according to their preferences. These are later introduced as filter queries (used previously in 5) when a new search is made.

7.3 Paging

Paging is the process of sectioning or partitioning data and retrieving a chunk. When a database reaches a large volume, fetching and collecting the whole indexed rows is unfeasible, in terms of memory usage and network overloading (when API requests are involved). The user interface is also affected. Paging relieves these weights by allowing a desired amount of results to be fetched from the server. Solr offers two types of paging:

- *start* and *row* [10]: placed in the URL query, but the client side has the responsibility of maintaining the current page, leading to other problems (the client side doesn't have the knowledge of the actual state of the database, creating offset problems [7]).
- *Cursor* [11]: a common alternative (with relative indexes), often used in SQL systems. The server keeps track of the selected chunk issuing a key for the next available slice.

We used the latter in our implementation. The cursors start off as the default '*' selection, and as new responses come with the new *cursorMark*, appended to the next request. Unfortunately, the previous cursor checkpoint is not available as a Solr parameter in the response, so a "stack-like" structure stores already-fetched cursors. This is necessary in order to supply the front end with the back page option, completing the pagination UI.

7.4 Suggester/SpellChecker

We wanted a system to show our users suggestions of books based on what they are typing on the search bar. With this idea on our minds, we discovered Solr has a Search Component called "SuggestComponent" that enables us to do that. This component needs some parameters to execute this task but the most important ones are:

- **lookupImpl**: This parameter is the lookup implementations. Solr has several possible implementations to look up terms in the suggested index and we choose the *FuzzyLookupFactory*. The first option we opted for was *AnalyzingLookupFactory* which first analyzes the incoming text and adds the analyzed form to a weighted FST (finite state transducers) and then does the same thing at lookup time. For general purposes, an FST is a finite-state machine with two memory tapes: an input tape and an output tape. This difference makes an FST able to define relations between sets of strings. This was great at first but the search was an exact-string match. Searching for more options, we found *FuzzyLookupFactor* which does the same thing as the previous LookupFcatory but is fuzzy by nature, which is better in our opinion, since covers misspelled search terms.
- **dictionaryImpl**: This parameter represents the dictionary implementation we used. A dictionary implementation defines how terms are stored. We explored some of them with different weights and payloads but found almost no difference between the returned results and ended up using the default one: *DocumentDictionaryFactory*.
- **field**: This parameter represents the attribute where the suggester will search for books. The one we used is the book title.

After this, we added the request handler '/suggest' which allowed us to make requests to Solr and get suggestions on our interface.

When it comes to spelling, we noticed a bare-bones definition of a spelling search component is already built. However, we did not know how to insert such a feature on our interface and connect it with the other features. We had

[10] https://solr.apache.org/guide/6_6/pagination-of-results.html#basic-pagination

[11] https://solr.apache.org/guide/6_6/pagination-of-results.html#fetching-a-large-number-of-sorted-results-cursors

to decide upon Suggester or Spelling and ended up using Suggester.

7.5 Highlight

Highlighting is the art of emphasizing a word/expression and we wanted the users to be able to check if the word/-expression they searched for appears in the results. Using Solr innate Highlighting, we were able to accomplish this by inserting on the query request handler if we wanted to highlight ('hl=on') and on what field we wanted the highlight to appear ('hl.fl=title'). The term to be highlighted is the query term. With this simple way, we added a highlight system.

7.6 Corrections/fixes

When it comes to corrections and fixes, we decided to:

- Remove from the schema the filter 'RemoveDuplicatesTokenFilterFactory' since we decided to improve our queries with boost frequency in some attributes like 'description' and 'review_text'. This is important because when we want to quantify how positive or negative a message is, the frequency of a related term is important, and with that filter, messages like "good, good, and good" were cut down to only "good". These two messages have a difference when it comes to relevancy and we wanted that difference to persist;
- The list of stop words we used had some words we thought were important when it comes to evaluating terms. For example, the term 'not' was on our previous list of stop words, being it a word we needed back to search for negative scenarios like "not good", etc. Knowing this, we removed some words from this list;
- The synonyms list was vastly improved from the previous iteration. We started by searching for a complete synonyms list [12]. From this list, we wanted to apply lemming to each word of the list, but since we couldn't do it in the QueryAnalyser like we wanted and the results there are stemmed, we decided to apply stem. From this list of stemmed words, we formatted the list to the format we wanted: 'stemmed synonyms => regular word to converge'

7.7 New evaluation

Using the previously referenced tables (9.1) as evaluated results, new metrics composed with the official calculation spreadsheet can be found in the annexes (9.2).

Although the evaluation turned out more precise and liable, this new approach didn't change the outlook we had on each query, thus the previous conclusions still remain up to date **??**.

[12] https://github.com/hansonrobotics/hr-solr/blob/master/synonyms.txt

7.8 New additions to the dataset/Scrapping

These are the changes we made to the dataset in this period:

- Added the 'rank' attribute to the metadata. This attribute increases the number of queries we can do or filter when it comes to books.
- Noticed a trend where a significant percentage of books had '- Kindle edition' on their title. These were removed.

8 CONCLUSION AND FUTURE WORK

In sum, the results of the evaluation were satisfactory since we got a good and well-structured document storage, indexing on what we perceive as relevant and optimizable. Generally the average precision for all queries in a given scenario stayed around 0.8, which we considered a good value in cases where the user performs all the available queries. More concise and specific queries generated the needed results, scoring well in evaluation. More complex and vague searches, turned out to be a coin-toss scenario, while the positive reviews query (5.3) met and surpassed our expectations with a 0.78 AvP, the boring reviews search (5.5) maxed out at 0.57 AvP, unable to achieve our goals.

One question we stumbled across when evaluating our queries was determining the relevance of a given result. It was unclear to us how to choose the most adequate reviews or books to calculate the recall and precision. Given that we are not independent ob- servers, every query result we selected or searched for while taking a "blind" strategy, may sometimes include a bias towards the query we constructed. We opted to sample the results, embedding some relevant objects into it, guaranteeing a filled query response. This allowed us to gain control of the overall pool of data, and more easily debug and evaluate queries.

In hindsight, Solr proved to be a very powerful tool in terms of searching capability. It lacks clear documentation, which is enough to fluster any user and deter its usage. We aimed and managed to extract the full potential of its tools and customization, while also learning how to set up the best query parameters to achieve the best possible results. Rethinking our steps, we didn't realize how important the initial dataset choice was for future stages (had wrong expectations of the work ahead). The initial cleaning and analysis were essential to figure out what type of queries we could make in the future, and we found out we were severely restricted by the data at hand, too late.

In conclusion, a well-constructed implementation and analytical discussion of the progress made throughout the semester is what we think to have achieved.

8.0.1 Future Work. Future work, to be completed:

- Try out other features Solr includes (i.e. OpenNLP lemming);

- Index the documents with a nested structure. An attempt was eventually successful, however effectively making this change would require some alterations to the existing queries and an introduction of a new one, which highlighted this fact;
- Scrape more data to have a more complete dataset (descriptions could be obtained on other websites or even with Amazon's API, which paid, unfortunately);
- Continue to improve query results and metrics;

9 ANNEXES

9.1 Query results

position	Id	Title
1	B00AZGEDVG	"All of My Love (Grayson Friends Book 9)"
2	B008B0AAU6	"The Education of a White Boy"
3	B00CRTAXQS	"Truth Hurts, Lies Kill 2 - Kindle edition"
4	B005VD8QJ8	"The Wish: A Holiday Story - Kindle edition"
5	B00FND3CO0	"Love Is Relative (Defining Love Series, Book One)"
6	B0046ZS31I	"Fury From Within"
7	B00CXVVGNO	"Terra Two eBook"
8	B00A4CSGDW	"Kindle Fire HD Apps"
9	B00ANSE582	"Tilly Lake's Road Trip - Kindle edition"
10	B00DTNV0YA	"The Highlander's Bride - Kindle edition"

Table 9: Retrieved documents on query 1.

position	Id	Title
1	B00EP0A71I	"Soccer Games - Jessica's First Soccer Goal - Kindle edition"
2	B00F1Q2YFW	Football Fun with Jake - The Rookie Football Season - Kindle edition"
3	B00CBMBIQA	"Secret Football Games - World Cup Plan"
4	B007WM3IBW	"The Demon You Know: A Demon Hunting Soccer Mom Short Story"
5	B00BG3EN3A	"Moon Wreck"
6	B00EBUC5G2	"All's Fair in Love and Football - Kindle edition"
7	B00CXACPZ8	"Christine Becomes The Football Quarterback A Prudence Periwinkle Book - Kindle edition"
8	B00EBUC8KU	"All's Fair in Love and Football 2 - Kindle edition"
9	B00E9M5FX2	"Feudlings in Sight (Novella) (Fate On Fire) eBook"
10	B005I57MXK	"The Bathtub Spy (Kindle Single) eBook"

Table 10: Retrieved documents on query 2.

9.2 Query relevance analysis

KindleMeSome

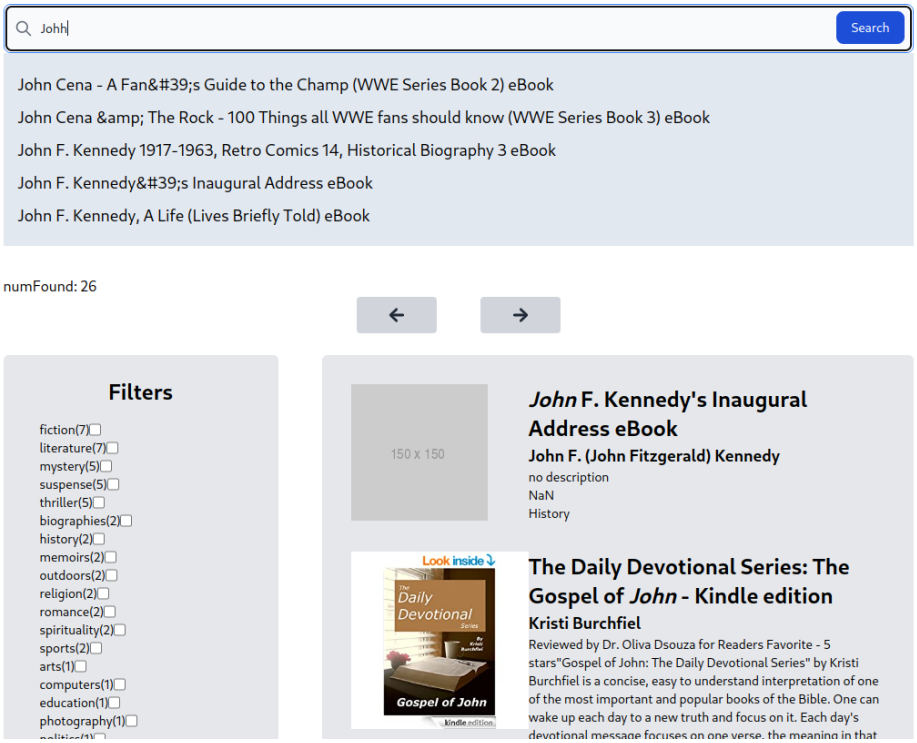

Figure 7: Interface.

position	reviewId	summary
1	201099	"A Great Survival Guide"
2	201106	"Survival Guide Necessity!
3	201708	"Good Analysis!"
4	202148	"Why We Do What We Do"
5	185099	"Thinking of investing? Read 'How to Choose a Good Trading System' First"
6	197312	"Great Resource for video marketing"
7	206546	"no fluff, just great advice"
8	122480	"Boat Load of Info! Thanks!"
9	209715	"Survival Planning tool"
10	201702	"Pleasantly Surprised"

Table 11: Retrieved documents on query 3.

position	reviewId	summary
1	277030	"A Truly Great Summer Read!"
2	152230	"The Perfect Holiday Escape!"
3	337902	"Kicking off the holidays with red shoes"
4	264764	"Good vacation read!"
5	161607	"A quick fun holiday read!!"
6	274620	"Sweet & Tasty, with a Bite of Sassy Flavor"
7	99523	"Great Story"
8	338014	"Break Away From the Mundane Holiday Reads!"
9	109068	"I liked the story"
10	312645	"I really enjoyed this one."

Table 12: Retrieved documents on query 4.

position	reviewId	summary
1	165678	"Found it boring in the long run"
2	70680	"Long and boring"
3	139386	"way too long and boring"
4	103992	"Too long"
5	31788	"Hard To Believe"
6	3761	"The long, long, long road"
7	11162	"A long boring story !"
8	275818	"Very boring"
9	82281	"Boring, boring, boring"
10	200442	"Boring, boring, boring....."

Table 13: Retrieved documents on query 5.

position	Scen 1	Scen 2	Scen 3
1	1	1	1
2	1	1	1
3	1	1	1
4	1	1	1
5	1	1	1
6	1	1	1
7	0	1	1
8	1	0	1
9	1	1	1
10	0	1	1

Table 14: Relevance of the results in query 1 (for 3 scenarios).

position	Scen 1	Scen 2	Scen 3
1	1	1	1
2	1	1	1
3	0	0	1
4	0	1	1
5	1	1	1
6	0	0	0
7	0	1	1
8	1	0	1
9	0	0	1
10	0	1	1

Table 15: Relevance of the results in query 2 (for 3 scenarios).

position	Scen 1	Scen 2	Scen 3
1	1	1	1
2	1	1	1
3	0	0	1
4	0	1	0
5	0	0	0
6	1	1	1
7	0	1	1
8	1	0	1
9	0	1	0
10	0	1	1

Table 16: Relevance of the results in query 3 (for 3 scenarios).

position	Scen 1	Scen 2	Scen 3
1	1	1	1
2	0	0	0
3	1	1	1
4	1	0	0
5	1	1	0
6	1	1	1
7	1	1	0
8	0	0	0
9	1	0	0
10	0	1	1

Table 17: Relevance of the results in query 4 (for 3 scenarios).

position	Scen 1	Scen 2	Scen 3
1	0	0	1
2	0	0	0
3	1	1	1
4	0	1	0
5	0	0	0
6	0	0	1
7	0	0	0
8	1	1	1
9	1	0	1
10	0	1	1

Table 18: Relevance of the results in query 5 (for 3 scenarios).

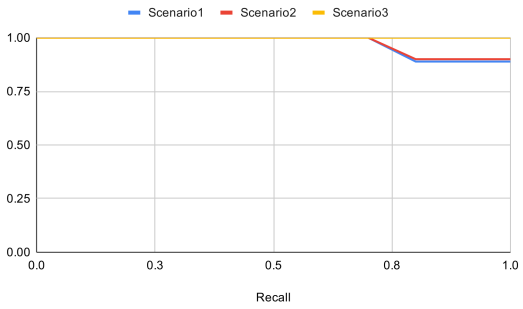

Figure 8: Precision-recall curve of query 1.

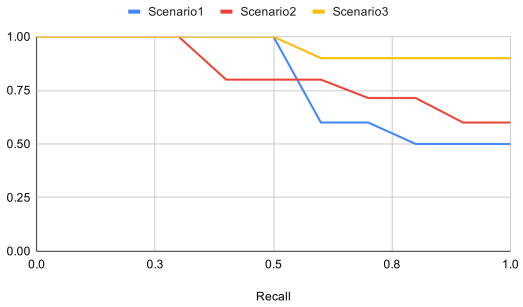

Figure 9: Precision-recall curve of query 2.

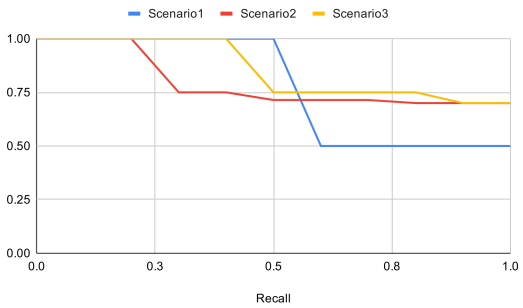

Figure 10: Precision-recall curve of query 3.

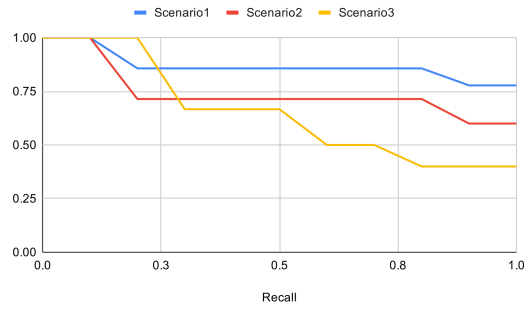

Figure 11: Precision-recall curve of query 4.

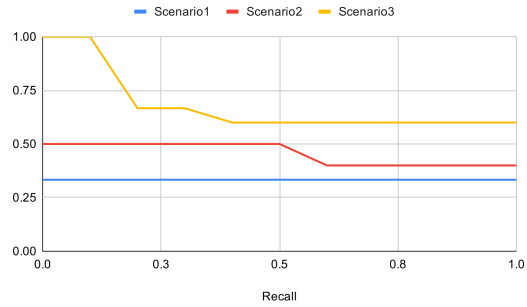

Figure 12: Precision-recall curve of query 5.

REFERENCES

[1] 2008. *Modern algebra of information retrieval.* (2nd ed.). O'Reilly Media. Chap. 4 Basics of Information Retrieval Technology, 92–93. ISBN: 978-3-540-77658-1.

[2] History.com Editors. 2015. Amazon opens for business. Retrieved October 9, 2022 from https://www.history.com/this-day-in-history/amazon-opens-for-business.

[3] Julian McAuley Jianmo Ni Jiacheng Li. 2019. Justifying recommendations using distantly-labeled reviews and fine-grained aspects, 10 pages. https://cseweb.ucsd.edu/~jmcauley/pdfs/emnlp19a.pdf.

[4] O'Reilly Media. 2020. Jump to a product using its asin. Retrieved May 4, 2020 from https://www.oreilly.com/library/view/amazon-hacks/0596005423/ch01s05.html.

[5] 1997. *An algorithm for suffix stripping. Readings in Information Retrieval.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 313–316. ISBN: 1558604545.

[6] Yonik Seeley. 2020. Faceted search with solr. Retrieved Fri, 11 Nov 2022 from https://lucidworks.com/post/faceted-search-with-solr/.

[7] Apache Solr. 2017. Pagination of results. Retrieved 08 Jun 2017 from https://solr.apache.org/guide/6_6/pagination-of-results.html#basic-pagination.