Relevance Prediction using Scala and Spark

Georgios Arampatzis
School of Informatics
Aristotle University of Thessaloniki
Greece
garampat@csd.auth.gr

Orestis Lampridis
School of Informatics
Aristotle University of Thessaloniki
Greece
lorestis@csd.auth.gr

1 Introduction

In this project, we will tackle a problem that involves text mining, natural language processing, feature extraction from text, machine learning and the use of various data mining and information retrieval techniques.

The particular task is focused on predicting the quality results of a query at the site of Home Depot. Past queries from users were evaluated on their quality from the users themselves. Thus, for each query we will have search query, the result product and the evaluation of that query query (with values that range from 1 to 3). We will work in two different flavors of the problem. The supervised case, where the ground truth information is available and the unsupervised case, in which there is no ground truth information to use (no labels).

We are going to utilize Apache Spark with the Scala programming language, in order to parallelize the work and thus make it run more efficiently.

2 Dataset

The dataset that we were given contained three separate csv files each with their own set of data. These included:

- **train.csv**, which contains the products, searches and their relevance score..
- **product_descriptions.csv**, which contains a detailed text description of every product listed
- **attributes.csv**, which provides additional information for all the given product ids (e.g. material, brand name, height, weight, dimensions etc.)

In more detail, the three tables contain the following fields:

- id, which is a unique id field that represents a pair between the search_term and the product_uiid
- product_uid, an id for products.
- product_title, which is the title of every product
- search_term, which includes the search term used by the customers
- relevance, which indicates how much the customer was pleased with the result of his/hers search (an evaluation of 1 in the relevance score field equals to an unsatisfactory result, whereas an evaluation of 3 indicates the most expected result for the customer).
- product_description, which includes a text description of the products
- name, the attribute's name
- value, the attribute's value

For our purposes we had to choose which attributes to use from the attributes.csv file. After careful consideration, we decided to use only the brand names included in this file because the brand name has a high probability of appearing in the text of the queries that users make.

Next, we had to create a unified representation comprising of all our data. To that end, we created a spark dataframe for each of the csv files and joined them all together on the product id. In this fashion, we ended up with a single spark dataframe comprising of the following columns: product_uid (the product id), product_title, search_term (the query), product_description, brand and relevance. Like this we are able to move to the next step, which requires the transformation of our text data in order to uncover the underlying information.

3 Text pre-processing

The transformation of unstructured text into a structured set of data is not a straigh-forward task and the areas of text mining and natural language processing offer a wide variety of different tools and approaches to face the irregularities which materialize in natural language.

The pipeline of text pre-processing consists of a number of stages that aim to improve further the results of the machine learning algorithms. The following methods were used for the pre-processing of the data used:

- Lowercasing
- Remove special characters
- Tokenization
- Stop word removal
- Stemming

As a first step, we used the lower() method in the columns of our dataframe in order to lowercase our text data. Then, we proceeded by removing all the special characters from the text, including punctuation with the use of a special function we created. After that, we proceeded by using tokenization, which helped us split the text into tokens. That way, the text became more “clear”, helping us distinguish each individual token as a separate word which helped us achieve the following steps. On top of that, we also removed stop words. Those words are commonly used in their respective language and often do not bear any useful information or additional meaning to the sentence. Up until now, there is not a universal list of stop words to be removed, so depending on the tool used the results may vary. For our purposes, we used StopWordsRemover from org.apache.spark.ml.feature library.

Afterwards, we wanted to stem the words to transform them into their root forms. For this reason, we decided to use Spark-NLP[1], an NLP library built on Apache Spark, used for aiding in various text-preprocessing steps such as stemming, lemmatization and spell check among others.

Stemming[2] is the process of reducing inflection in words to their root forms such as mapping a group of words to the same stem even if the stem itself is not a valid word in the Language.

Lemmatization[2], unlike Stemming, reduces the inflected words properly ensuring that the root word belongs to the language. In Lemmatization root word is called Lemma. A lemma (plural lemmas or lemmata) is the canonical form, dictionary form, or citation form of a set of words.

In our case we chose stemming since it is a simpler and faster method than lemmatization, as it does not require the use of a dictionary.

Then, we were ready to transform our text into vectors. We used three different methods to accomplish this transformation. These were the following:

- **TF-IDF**: a feature vectorization technique which works by summarizing how often a given word appears within a given document.
- **Word2Vec**: which takes as input sequences of words representing documents and trains a Word2Vec model. The model maps each word to a unique fixed-size vector.
- **CountVectorizer**: which takes as input a collection of text documents and converts them to vectors of token counts.

4 Feature Engineering

Feature engineering is the process of extracting features from raw data, in our case text, with the use of various techniques. The extracted features can then be used to train machine learning models. Extracting and selecting good quality features allows a more accurate representation of the underlying structure of the data and therefore helps in the creation of more accurate machine learning models.

[1] "Spark NLP - John Snow Labs." https://nlp.johnsnowlabs.com/.

[2] "Deep Learning Pipeline | SpringerLink." https://link.springer.com/book/10.1007/978-1-4842-5349-6.

We decided to extract a mix of different features from our data including:

- **Vector representations**: the TF-IDF and Word2Vec vector representations of the product title, the query, the product description and the brand name.
- **Counting features**: the length of the product title, the length of the query, the length of the product description, the length of the brand name, the number of common words between the query and the title, the number of common words between the query and the description, the number of common words between the query and the brand.
- **Statistical features**: ratio of title length to query length, ratio of description length to query length.
- **Similarity features**: the cosine similarity and the euclidean distance between the vectors of the product title and the query.

In this manner, we finally end up with 19 extracted features.

After creating our features, we tried to use Normalization. Normalizer[3] is a Transformer which transforms a dataset of Vector rows, normalizing each Vector to have unit norm. It takes parameter p, which specifies the p-norm used for normalization. (p=2 by default.) This normalization can help standardize the input data and improve the behavior of learning algorithms.

Before starting to run our experiments, we tried using the ChiSqSelector, to help us select the most important features by pointing out which have the most predictive power. A chi-square test is used in statistics to test the independence of two events. ChiSqSelector stands for Chi-Squared feature selection. ChiSqSelector uses the chi-square test of independence to decide which features to choose. We used the parameter numTopFeatures, which chooses a fixed number of top features according to a chi-squared test.

The results of the ChiSqSelector pointed out that, the majority of our counting features, especially those that count the length, are of lower importance.

The last step in our preprocessing of the data was to use the VectorAssembler. VectorAssembler[4] is a transformer that combines a given list of columns into a single vector column. It is very useful for combining raw features and features generated by different feature transformers into a single feature vector. This step was needed in order to train all the machine learning models used in the experiments.

[3] "Extracting, transforming and selecting features - Spark 2.4.4 ...." https://spark.apache.org/docs/latest/ml-features.

[4] "VectorAssembler - Apache Spark." https://spark.apache.org/docs/latest/api/python/pyspark.ml.html?highlight=vectorassembler.

5 Supervised Case

To run our experiments we used a variety of machine learning algorithms for both regression and classification.

5.1 REGRESSION

Carefully examining the dataset in question, we concluded that the problem needs to be addressed with regression techniques, since the output variable (relevance) takes continuous values.

The list of the algorithms we used in our experiments are listed below:

- Linear Regression
- Generalized Linear Regression
- Decision Tree Regression
- Random Forest Regression
- Gradient Boosted Tree Regression
- Isotonic Regression

All the algorithms were implemented with the help of the documentation provided by Apache Spark and the default parameters were used.

5.2 CLASSIFICATION

Wanting to investigate further, we wanted to know if the given problem could be solved using classification techniques. In order to do that, we had to split the dataset into classes. The most logical way seems to split them based on the relevance feature. Upon examining the dataset we found that the field's values range from [1.00, 3.00], having 13 unique rates.

The approaches we chose were to split the dataset in:

- Two classes. One with values in the relevance field of 1.5 and greater and the other one with values less than 1.5. Thus isolating all the completely "bad" searches in our data.
- Two classes. One with values in the relevance field of 2.5 and greater and the other one with values less than 2.5. Thus, isolating all the completely "good" searches in our data.
- Four classes, those range from [1.00, 1.67), [1.67, 2.33), [2.33, 2.67) and [2.67, 3.00].
- Thirteen classes, which have all the unique relevance field's values.

During our experiment we tried to use a dimensionality reduction technique in order to reduce the number of variables under consideration. Since we tried this technique to increase the results of a classification problem, we used the Principal component analysis (PCA[5]). PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. A PCA class trains a model to project vectors to a low-dimensional space using PCA.

The algorithms used in the experiments are listed below:

- Random Forest Classification.
- One-vs-Rest
- Naive Bayes

All the algorithms were implemented with the help of the documentation provided by Apache Spark and the default parameters were used.

6 Unsupervised Case

In the second case of the problem we don't have access to ground truth data and thus we assume that the relevance information does not exist. We assume that any relevance score less than a particular value *n* is considered as 0 and any relevance score equal or larger than *n* is considered as 1. To solve this problem, we only worked with the queries and the product titles returned as a result in order to predict if the result is relevant to the query or not. We thought of three different approaches to solve this problem either by using cosine similarity or locality sensitive hashing (LSH).

LSH is a class of hashing techniques, which work by hashing data points into buckets, so that the data points which are close to each other are in the same buckets with high probability, while data points that are far away from each other are very likely in different buckets. Since similar items end up in the same buckets, this technique can be quite useful for data clustering and approximate nearest neighbor search.

6.1 Cosine Similarity

The first approach of solving this problem is to use the attributes of the cosine similarity measure. The thought process behind our idea is that two vector representations of text are highly likely to have a high cosine similarity value if they are relevant. By using the Word2Vec vector representations for the product title and the search term respectively, we are able to calculate the cosine similarity between them using a UDF that we created. The UDF takes as parameters two vectors and returns their cosine similarity.

Afterwards we use a threshold value which can be set to any value between 0 and 1. Any pair of title and query that has a cosine similarity value higher or equal than that of the threshold is considered as being relevant and any pair that has a value lower than that of the threshold is considered as not being relevant. This threshold is a parameterized value and we explore the different results given different values in the next section.

6.2 MinHash

MinHash is an LSH family for Jaccard distance where input features are sets of binary numbers, in our case the vectors representing the text. Jaccard distance *d* of two sets *A* and *B* is defined by the cardinality of their intersection and union.

$$d(\mathbf{A},\mathbf{B}) = 1 - \frac{|\mathbf{A}\cap\mathbf{B}|}{|\mathbf{A}\cup\mathbf{B}|}$$

For representing the text into vectors we chose to use the count vectorizer. Then to implement the minhash method we followed closely the code provided in the Apache Spark documentation. We created a MinHashLSH() model with setNumHashTables(5). After fitting our model based on the title DF and transforming both the title and query DFs, we used approxSimilarityJoin(dataset1_LSH, dataset2_LSH, 1, "JaccardDistance"). This approximately joins dataset1_LSH and dataset2_LSH on Jaccard distance smaller than 1 into a new DF called jaccardDF. Consequently we end up with all the possible pairs of both dataframes.

Next, after having at our disposal the joined jaccardDF which contained the values of all possible pairs, we needed to throw away all the pairs that were not pairs originally. For this reason, we kept and index of every title and every query so that each pair of title and query would have the same index. Thus, we used a filter to keep only the rows where the index of the first dataframe had to be equal to the index of the second dataframe. Like this, we end with all the original title-query pairs, along with their jaccard distance.

Similarly to the cosine similarity approach, we use a threshold value which can be set to any value between 0 and 1. However, because we are handling the jaccard distance and not the jaccard similarity we needed to be careful. Contrary to the cosine similarity approach, any pair of title and query that has a Jaccard distance value higher or equal than that of the threshold is considered as being not relevant and any pair that has a value lower than that of the threshold is considered as being relevant. Again, this threshold is a parameterized value and we explore the different results given different values in the following section.

It should also be noted that due to repeated memory problems in Spark when using the complete datasets, we decided to only utilize a smaller sample of our dataframes. Thus, for MinHash and for the following approach (Bucketed Random Projection), we only used 500 rows from both of our dataframes (title DF and query DF).

6.3 Bucketed Random Projection

Bucketed Random Projection is another LSH family for Euclidean distance. The Euclidean distance is defined as follows:

$$d(\mathbf{x},\mathbf{y}) = \sqrt{\sum_i (x_i - y_i)^2}$$

To a large extent, this method is similar to the previous one. As before, we used the count vectorizer to turn the text into vectors and for the implementation, we followed closely the code provided in the Apache Spark documentation. We created a BucketedRandomProjectionLSH() model with setBucketLength(2.0) and setNumHashTables(3). After fitting our model based on the title DF and transforming both the title and query DFs, we used approxSimilarityJoin(dataset1_LSH, dataset2_LSH, 1000, "EuclideanDistance"). This approximately joins dataset1_LSH and dataset2_LSH on Euclidean distance smaller than 1000 into a new DF called jaccardDF. Consequently we end up with all the possible pairs of both dataframes along with their Euclidean distances.

For eliminating non-pair titles and queries we used the same method as in MinHash. We also noticed that the highest value of Euclidean Distance was 5.0, however this should be investigated further. As in the previous approaches and assuming that the highest value of Euclidean distance is 5, we thought of using a threshold value which can be set to any value between 0 and 5. Any pair of title and query that has a Euclidean distance value higher or equal than that of the threshold is considered as being not relevant and any pair that has a value lower than that of the threshold is considered as being relevant. Once again, this threshold is a parameterized value and we explore the different results given different values in the following section.

7 Results

7.1 Supervised

In order to evaluate our approach we split the data into 60% for the training set and 40% for the testing set. The metric that we used for the evaluation is the Mean Squared Error (MSE).

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

When given an estimator, the MSE is a loss function which works by measuring the average of the squares of its errors. In other words, it measures the average squared difference between the estimated values and the actual value.

7.1.1 Regression

First we tried running the methods without any preprocessing on the data, in order to be able to verify if our experiments actually benefit the algorithms. In order to conduct our experiments we merged the “train.csv” and the “description.csv”. We left the “attributes.csv” for a later stage of the experiments.

The features used in this part were the following:

- The length of the search term
- The length of the product title
- The length of the product description
- The ratio of product description to search term
- The ratio of product title to search term

Algorithms:	MSE
Linear Regression	0.2814276866130841
Generalized Linear Regression	0.28163355248001204
Decision Tree Regression	0.27707391734179737
Random Forest Regression	0.27658712079183545
Gradient Boosted Tree Regression	0.2766612112837516
Isotonic Regression	0.28547393462661086

As a first step in preprocessing the data, we started by removing all the special characters used in the data. In addition, we also removed all the stop words included in the data, trying to filter out any not useful words.

Algorithms:	MSE
Linear Regression	0.2810170710300641
Generalized Linear Regression	0.28111293671008786
Decision Tree Regression	0.2778852581223346
Random Forest Regression	0.2771243486530173
Gradient Boosted Tree Regression	0.2776181783543509
Isotonic Regression	0.2865967802870633

By doing so, we noticed a slight improvement in Linear and Generalized Linear Regression methods, but the results on the other methods were a little worse.

In order to counter, we thought of adding some new features:

- Common Words between Search term and Product Title
- Common Words between Search term and Product Description

But for this method to work, we also had to convert all the words in our data in lower case. We also thought to examine which of these two new features is more important. In order to do that, we ran our test three times. The first time (1) we used only the common words in search term and product title. The second time (2) we used only the common words in search term and product description and the third time (3) we used both features.

(1)

Algorithms:	MSE
Linear Regression	0.26408113815984635
Generalized Linear Regression	0.2675157870701835
Decision Tree Regression	0.2549001489553074
Random Forest Regression	0.2575681947913835
Gradient Boosted Tree Regression	0.25448222169390544
Isotonic Regression	0.2861668505044218

(2)

Algorithms:	MSE
Linear Regression	0.26685518965935195
Generalized Linear Regression	0.2695523959329641
Decision Tree Regression	0.2591219614842563
Random Forest Regression	0.2612775324602847
Gradient Boosted Tree Regression	0.2592586562534526
Isotonic Regression	0.286185946313925

(3)

Algorithms:	MSE
Linear Regression	0.26012980367811134
Generalized Linear Regression	0.26318145650028935
Decision Tree Regression	0.2503378682971965
Random Forest Regression	0.2519542372429813
Gradient Boosted Tree Regression	0.24913706067502314
Isotonic Regression	0.2856167083527193

Comparing those results, we can clearly see that the feature of "common words between search term and product title" is more important than the "common words between search term and description title", since it gives better results. In addition, we can determine that those features actually do help the process since they improve the results of the algorithms in comparison with the earlier results. Moreover, while using both features in addition to the five previous ones, we see that all the algorithms presented give the best results yet. More specifically, the Gradient Boosted Tree Regression algorithm has a Mean Squared Error of 0.2491. Our next step was to normalize the features used in our experiments.

Algorithms:	MSE
Linear Regression	0.26584211916206674
Generalized Linear Regression	0.26928594566384356
Decision Tree Regression	0.262594739502341
Random Forest Regression	0.25788658700259054
Gradient Boosted Tree Regression	0.2523432532715279
Isotonic Regression	0.2817897753442357

After examining the results, we notice that when running our tests with normalized features, there is a slight increase in the mean squared error in all cases apart from the Isotonic Regression algorithm. Due to that, we concluded that normalizing our data is not the optimal path, so we excluded this step for our future experiments with this dataset. Next, we chose to add the cosine similarity between the product title and the search term and also their Euclidean distance. We run our experiments three different times. Like before, once while using only the cosine similarity as feature (1), a second time while using only the Euclidean distance as a feature (2) and a third time while using both features (3).

(1)

Algorithms:	MSE
Linear Regression	0.2565475904848559
Generalized Linear Regression	0.2623703927163111
Decision Tree Regression	0.2570093781983407
Random Forest Regression	0.2517208538768476
Gradient Boosted Tree Regression	0.2425664693423298
Isotonic Regression	0.2831595729411892

(2)

Algorithms:	MSE
Linear Regression	0.2536920195532292
Generalized Linear Regression	0.258472782078485
Decision Tree Regression	0.2448988452900881
Random Forest Regression	0.2469193715228414
Gradient Boosted Tree Regression	0.2433650261713034

Isotonic Regression	0.2921622839849574

(3)

Algorithms:	MSE
Linear Regression	0.2462765350810221
Generalized Linear Regression	0.2503853133198072
Decision Tree Regression	0.2407157567926998
Random Forest Regression	0.2417962467218619
Gradient Boosted Tree Regression	0.236841959030064
Isotonic Regression	0.2912443974528835

After examining the results we concluded that, while the test using the Euclidean distance produce better results than the one with the cosine similarity, the most satisfying results came when using both features. After that, we thought of comparing the previous results with two new features. So, we excluded the last two features added (Cosine similarity and Euclidean distance) and added the vector representations of the title and query using the Word2Vec.

Algorithms:	MSE
Linear Regression	0.2475489732984553
Generalized Linear Regression	0.2528471739280578
Decision Tree Regression	0.244253035320332
Random Forest Regression	0.2454466312039993
Gradient Boosted Tree Regression	0.2324886708765216
Isotonic Regression	0.2916432766301844

The results, though not better, only deteriorated slightly. So, our next (and most logical) step was to add all those features together.

Algorithms:	MSE
Linear Regression	0.2423600007636339
Generalized Linear Regression	0.2467810959275811
Decision Tree Regression	0.2411524371792415
Random Forest Regression	0.2414011841013734
Gradient Boosted Tree Regression	**0.2305398901781349**
Isotonic Regression	0.2912443974528795

As it appears, the MSE has dropped even further, reaching a value of 0.23 in the case of Gradient Boosted Tree Regression, an algorithm that produced the best results in every case so far.

At this point, we tried merging all the possible information we had from the data, including the “attributes.csv”. First we tried running the experiments only using counting, statistical and similarity features.

Algorithms:	MSE
Linear Regression	0.2445729182811379
Generalized Linear Regression	0.2484444459766509
Decision Tree Regression	0.2397796061124222
Random Forest Regression	0.2398143436434514
Gradient Boosted Tree Regression	0.2359678682706382
Isotonic Regression	0.2887855573225136

The results were extremely satisfactory having MSE in most cases less than 0.25. Next, we thought of using only the vector representation of our data.

Algorithms:	MSE
Linear Regression	0.2695314343065376
Generalized Linear Regression	0.2704946481606484
Decision Tree Regression	0.272149738385558
Random Forest Regression	0.2662635087090829
Gradient Boosted Tree Regression	0.2565222367317001
Isotonic Regression	**0.2804209331533791**

The results of all algorithms were far worse, indicating that only using the vectors created by the wod2vec is not enough. So as our next step we combine all of the features, including the TF-IDF vector representations. Thus, we use all of our 19 extracted features. After training our Linear Regression model, we found out that the MSE got far worse than before with a value of 0.3011453501148741. It was evident that the TF-IDF features were not useful. Consequently, we decided to continue our experiments using all the features excluding the 4 TF-IDF vector representations.

Algorithms:	MSE
Linear Regression	**0.2399439889370123**
Generalized Linear Regression	**0.2441751713022523**
Decision Tree Regression	**0.2413968357955297**
Random Forest Regression	**0.2389871305466019**
Gradient Boosted Tree Regression	0.2319681582001096
Isotonic Regression	0.2887855573225136

As it appears, using all features (excluding TF-IDF vectors) from all three files, produce our best results. Since the number of features is now 15, we thought of using the ChiSqSelector to determine which are the most useful ones in predicting our results. The test concluded that those were:

- length of the search term
- ratio between the title and the search term
- common words between search term and title
- common words between search term and description
- common words between search term and attribute’s name

So, we tried running again our test using only those 5 top features.

Algorithms:	MSE
Linear Regression	0.2514379892229894
Generalized Linear Regression	0.25573509826693
Decision Tree Regression	0.2412049328298799
Random Forest Regression	0.2423200507105696
Gradient Boosted Tree Regression	0.2398237659497898
Isotonic Regression	0.3772560037564629

Finally, we can tell that using only the features suggested by the ChiSqSelector does not improve the results of the algorithms, thus making it clear that we need all possible information to have the lowest MSE.

7.1.2 Classification

In the case of classification, the algorithms that were used to test our results use as an evaluation metric the accuracy of the respective algorithms. The accuracy ranges from [0.00, 1.00] and a value of 1 equals to 100% of correct predictive results.

Using the Naïve Bayes classifier, we tried with and without normalizing the features we used, while using the two “2 classes” approaches. We also made some test trying with the 13 unique classes.

Naive Bayes		**Accuracy**
No normalize	2 classes (≥ 1.5)	0.9331359377762065
	2 classes (≥ 2.5)	0.5889605798126215
	13 unique classes	0.00795474633197808
With Normalize	2 classes (≥ 1.5)	09278387738774185
	2 classes (≥ 2.5)	0.5765423369277002

Based on the results it is clear that splitting the dataset in 13 classes is not an optimal option and that only the binary classes’ approaches give results that are more satisfying. In more detail, only when splitting the two classes in 1.5 relevance the classifier had an accuracy over 92%, while on the other case is a little below 60%. This is due to the nature of the dataset, having only a few data with relevance below 1.5.
In addition, like in the previous experiments, we observed that when we normalize the feature used to calculate the results, the accuracy of the method used slightly drops.

Next we used One-vs-Rest Classifier and Random Forest Classifier to test our data with more classification algorithms.

First we tried running the tests using only counting and statistical features. The following results include experiments that were done in a “2 classes” approach, splitting the data once in 1.5 and once in 2.5. We also used the “4 classes” approach.

In addition to different class approaches, we also tried running the test with and without using PCA. When using PCA we tried reducing our features once to 3 and once to 5.

Class	PCA feat.	One-vs-Rest Classifier	Random Forest Classifier
2cl ≥1.5	-	0.7449941462673148	0.7304838651339088
	3	**0.9250928053738731**	**0.9251398786567905**
	5	0.9248718401979849	0.9251369984090507
2cl ≥2.5	-	**0.6240498497436804**	**0.6434063991514938**
	3	0.5724765776913558	0.5890931589181545
	5	0.6159183312709917	0.47732897295386245
4cl	3	0.42297153968534557	0.43895746898545178
	5	**0.44011843733427614**	**0.46299594548876548**

Judging by the results we can tell that PCA definitely helps to improve the results of our algorithms. Although, we only get satisfactory results when the dataset is split in 1.5 relevance. In addition, the results with 4 classes proved one again that this particular task is not a classification problem and that the regression methods are far superior.

7.2 Unsupervised

Before we started running the tests for any of the unsupervised approaches that we used, we had to convert the relevance score in a binary manner. For that we chose three different approaches.

- 2.0, in this case, since the relevance values range from 1 to 3, we split them right in the middle separating the with the unwanted searches.
- 1.5, in this case we tried excluding only the really unwanted results
- 2.5, where we exclude only the most satisfactory results based on the opinions of the customers

In order to evaluate our results we used the F1 score. This metric considers both the precision and the recall of a test in order to produce its score. The recall and precision are calculated based on the actual value and the predicted value (True Positives, False Positives, False Negatives, True Negatives) .

$$\textbf{Precision} = \frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

$$\textbf{Recall} = \frac{\text{True Positive}}{\text{True Positive + False Negative}}$$

$$F_1 = \left(\frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}}\right) = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

So, the values of the F1 - Score range from [0,1], with a higher value indicating better results.

7.2.1 Cosine Similarity

As mentioned in the previous section, before running the tests, we had to choose a threshold (θ), which will indicate when we predict that two texts are relevant. The higher the threshold, the more strict and difficult to find a match. We make the assumption that two texts are relevant when their cosine similarity is greater than or equal to θ. During our experiments we tried a variety of thresholds, trying to find a balance between a strict value and satisfactory results. Following are the F1 results for different values of θ and for different values of relevance score to convert to binary.

θ	1.5	2.0	2.5
0.9	0.1731577274	0.1823834196	0.2066881966
0.8	0.4751645194	0.4876812749	0.4537557233
0.7	0.6694885182	0.6733609621	0.5507178526
0.6	0.7899707232	0.7766339711	0.5869967183
0.5	0.8629004987	0.8364374376	0.6001167018
0.4	0.9058481707	0.8693377011	0.6041437042
0.1	0.9540358638	0.9008658352	0.6005435822

Examining the results we can see that, in all cases, when trying to find matches for a cosine similarity higher than 0.9, the results are extremely poor, failing in most cases. The case in which we concluded that we have our best results, are for a cosine similarity higher than 0.5.

7.2.2 MinHash

Likewise, before running the tests, we had to choose a threshold (θ), which will indicate when we predict that two texts are relevant. Contrary to the previous approach and because we are dealing with jaccard distance and not similarity, the lower the threshold, the more strict and difficult to find a match. We assume that two texts are relevant when their Jaccard distance is less than or equal to θ. Following are the F1 results for different values of θ. Also note that these results are not representative of the entire dataset, since due to memory problems we had to limit our dataframes only to 500 rows.

θ	f1-score
0.5	0.052830188679245
0.6	0.137184115523465
0.7	0.417177914110429
0.8	0.723716381418092
0.9	0.927710843373494
0.95	0.977186311787072

7.2.3 Bucketed Random Projection

One last time, we had to choose a threshold (θ), which will indicate when we predict that two texts are relevant. As explained in section 6.3, our values this time range from [0,5]. We assume that two texts are relevant when their Euclidean distance is less than or equal to θ. Following are the F1 results for different values of θ. Again, note that these results are not representative of the entire dataset.

θ	f1-score
2.5	0.3379174852652259
3.0	0.6687402799377917
3.5	0.8274932614555256
4.0	0.9214026602176542
4.5	0.9481132075471699
5.0	0.958139534883721

8 Conclusion

After running all our tests, we can conclude that in the supervised case, when trying to use regression algorithms the results were extremely satisfactory. We evaluated our results using the Mean Squared Error and in some cases reached a value of 0.23 in the case of Gradient Boosted Tree Regression. In addition, we can tell that our results were getting better almost every time we added new features. On the other hand, when we tried to use classification techniques we started to notice that the results were far worse than we expected. The algorithms had trouble pointing out the correct class, thus indicating that this problem is not a classification problem.

For the unsupervised case, we were surprised to see such good results when given such low values of cosine similarity or such high values of Jaccard or Euclidean distance. Upon further investigation, we discovered that the dataset was highly imbalanced as it contained a much larger amount of 1 (relevant documents), compared to 0 (non relevant documents). To prove that this was the case we decided to split the dataset into a different threshold as was done in the cosine similarity approach. The results speak for themselves. When choosing a threshold of 2.5 the dataset is almost evenly split and thus we can have a more meaningful evaluation. Looking at the table we decided that the best value to choose as a cosine similarity threshold would be 0.5. Future work could include the same principle applied on the LSH methods we used to further fortify our concept. Finally, focusing on the LSH methods, we noticed much better results with the use of Bucketed Random Projection compared to MinHash.