# **Overview**

**Home Depot Product Search Relevance**

<https://www.kaggle.com/c/home-depot-product-search-relevance/overview>

**Introduction**

Everything is moving to digitalization and so is the shopping experience. E-commerce has been on a rapid growth over the past few years; everything from buying stationary to buying a home can be done online. In any e-commerce website, product search plays the most vital role. For any search query, it's very important to show the most relevant product. A poor search result will lead to a disappointed customer and hence loss in business. Here we try to come up with a solution to this problem using Machine Learning.

**Business Problem**

For any search-query that the customer enters, he needs the correct result. It might not be the first product that is shown but it should be at the top few results (possibly the first page). A general customer doesn’t like to scroll down the results in search for a product and with every scroll the chances of a dissatisfied customer increases. An irrelevant search result can lead to a customer not being able to find the product which he/she could have bought and this can lead to huge loss in business. Not only does the customer want accurate results, but he wants it fast and hence there is this constraint of time.

The task is simple to understand; for any search-query that the customer enters, I need to find the most relevant products and show them to the user based on their relevance score i.e i need to rank them also; And i need to show this quickly. Now each product is stored as a text document which contains it’s information (like title, description, attributes, etc).And any query, if somehow I can calculate how closely related (or relevant) a product is to the query, then I can calculate this score for every product and then rank them on the basis of their scores. The measurement of the relevance between a search and a product is called product search relevance.

An intuition of similarity/relevance between a search and a product can be given by; Say if there were a large number of common terms between search and the product-document then one can say that they both are possibly related. But it’s not as simple as that; a search for “apple iphone” will match both with both iphone products and those related to the fruit “apple”, and then comes the problem of ranking them. Hence, a lot of things need to be considered.

**ML Formulation**

So the task can be formulated as: Given a search and the product text, find the relevance score between them. So if i had a lot of labeled data i.e a lot of (search-query, product-test) pairs with their relevance score then i can pose this as a supervised ML problem. The data we will be using is provided by Home Depot used in a [kaggle competition](https://www.kaggle.com/c/home-depot-product-search-relevance/overview).

Now in a real-world e-commerce search engine, the relevance score for every product is not possible as in any typical e-commerce website, the number of products are very large and hence it’s not feasible given the time-constraint. Thus first, we retrieve a few candidate products using some simple and fast retrieval model and then simpler retrieval models which permit fast query evaluation. And in the second phase, a more accurate but computationally expensive machine-learned model is used to re-rank these documents.

The problem that we will be solving is to calculate a relevance score for every query,product pair and not the ranking specifically.

**Real-World Constraints**:

Time - The customer needs to get the results quickly.

**Data Overview**

We have about 75k rows for training, each row corresponding to a pair of (search\_query, product\_title) and the relevance score telling how much that product is relevant to the search. The relevance score is a real number from 1 to 3 (1->not relevant & 3->most relevant). For each product, we generally have multiple rows i.e multiple search-query with their relevance score.

* product uid: Unique identifier of each product.
* product title: Title of each product.
* product description: A text description of each product.
* attributes (optional): Provide extended information about a subset of the products. Not every product will have attributes.

**Kaggle Objective and metric**

For every given (search\_query, product\_title), estimate the relevance of the product for the query given.

Now, this is all well and good but here they don’t take the real-world constraints i.e time into account.Thus while solving this problem, we need to consider the real-world constraints also.

Also here, we are given a pair (query, product) and we need to find the relevance score only for this pair.

In production, we would need to create an out-of-the-box search engine (e.g. Apache Solr or ElasticSearch) to retrieve a set of candidate products by matching user queries against properties text then apply a post-processing model to rerank the products.

For now, our focus is to just solve the Kaggle problem keeping the time constraint in mind.

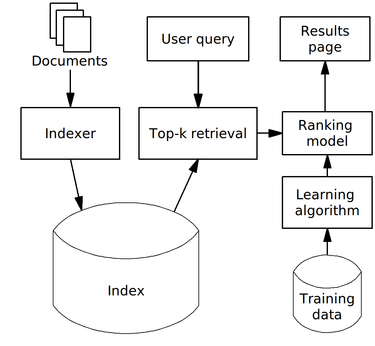
**Metric Used:** RMSE

# 

# **Research-Papers/Solutions/Architectures/Kernels**

The problem that we are dealing with is search relevance. This comes under the field of [Information Retrieval](https://en.wikipedia.org/wiki/Information_retrieval). Information Retrieval is what the name suggests - retrieving information. The most common example being search engines. You enter a query and then you get a list of documents matching that query ranked in the order of their ‘relevance’ or ‘similarity’ to the query. The simplest model could be an AND operator between query and document words. But this doesn’t rank them and ranking is very important. There are a number of non-machine-learned models like [BM25](https://en.wikipedia.org/wiki/Okapi_BM25) which can rank documents given a query.

In a typical machine learning search engine, you can’t check the relevance score for a given query on all the documents because of time constraints. Hence, first, a small number of potentially relevant documents are identified using simpler retrieval models which permit fast query evaluation, such as the [vector space model](https://en.wikipedia.org/wiki/Vector_space_model), [boolean model](https://en.wikipedia.org/wiki/Standard_Boolean_model), weighted AND, or [BM25](https://en.wikipedia.org/wiki/Okapi_BM25). In the second phase, a more accurate but computationally expensive machine-learned model is used to re-rank these documents.



In the home depot case study, our focus is on the second part i.e [learning to rank](https://en.wikipedia.org/wiki/Learning_to_rank).

The different kinds of information retrieval models can be classified into 4 groups:

* Set theoretic
* Algebraic
* Probabilistic
* Feature based Machine Learning models (Learning to Rank)

Our focus is on feature based machine learning models. Here feature functions are arbitrary functions of document and query and we can as such easily incorporate almost any other retrieval model as features.

**General Literature Survey**

1. **Literature Review of the common classical retrieval models** <https://www.youtube.com/c/VictorLavrenko/playlists>

<https://en.wikipedia.org/wiki/Information_retrieval>

**The set-theoretic models**

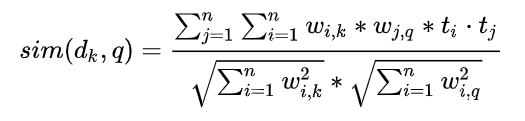
The set theoretic models like [Boolean model](https://en.wikipedia.org/wiki/Boolean_model_of_information_retrieval), [Fuzzy Retrieval](https://en.wikipedia.org/wiki/Fuzzy_retrieval), etc are based on representing queries and documents as sets and then applying some operation on them to get the relevance score. Some common features (simple set operations) that we learn from these model are

* AND, OR and NOT operators between a query and document
* Cosine coefficient, jacquard coefficient, inner product
* Set difference
* len(query), len(document)

**The algebraic models**

These generally represent documents and queries as vectors and then similarity between the query and the document is calculated as a scalar quantity. The variation occurs in how you set the weights of the vectors and then the use of similarity measures.

The [VSM](https://en.wikipedia.org/wiki/Vector_space_model) converts queries and docs into BOW or tf-idf vectors and then computes the similarity between them as the cosine of the angle between these two vectors.

The [GVSM](https://en.wikipedia.org/wiki/Generalized_vector_space_model) involves calculating summation of products of all weights of query vectors with every other weight of document vectors. For a document *dk* and a query *q* the similarity function now becomes:

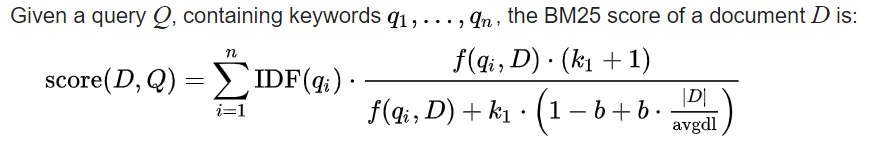
One more important technique used was [LSA](https://en.wikipedia.org/wiki/Latent_semantic_analysis#Latent_semantic_indexing) where truncated-SVD is applied to the term-document matrix such that the queries and documents are converted to vectors in the ‘concept’ space.

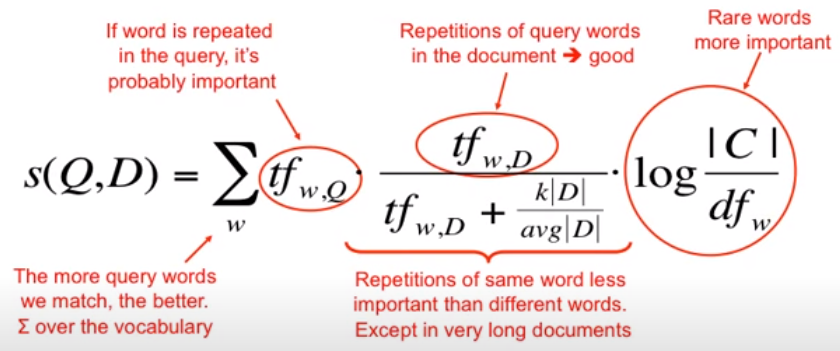
Note: In the vector models, the values of the vectors are called ‘weights’. In BOW these weights are tf of words, in tf-idf BOW these weights are tf\*idf values and so on. In the term-document matrix, we can have boolean or tf or tf\*idf weights as we like

**The Probabilistic Models**

Similarities/Relevance are computed as probabilities that a document is relevant for a given query. Probabilistic theorems like the Bayes Theorem are often used in these models.

The [Binary Independence model](https://en.wikipedia.org/wiki/Binary_Independence_Model) is very much like running a Naive Bayes classifier on the data.

[Okapi BM25](https://en.wikipedia.org/wiki/Okapi_BM25) is a very known model in which the relevance is defined by: 

The intuition for the formula can be given by 

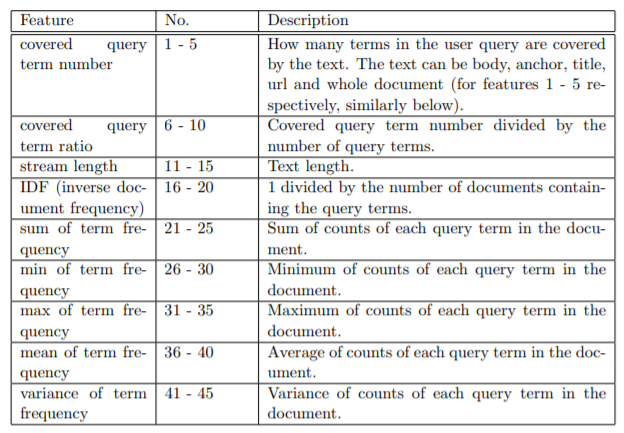
The only new parameters introduced are k1 and b which help to set how much of smoothing is required

Another class of probabilistic models include [Language Models](https://en.wikipedia.org/wiki/Language_model) where each document is proposed as a language model (probability distribution over sequence of words) and then documents are ranked based on the probability of the query q in the document’s language model Score = P(q|Md)

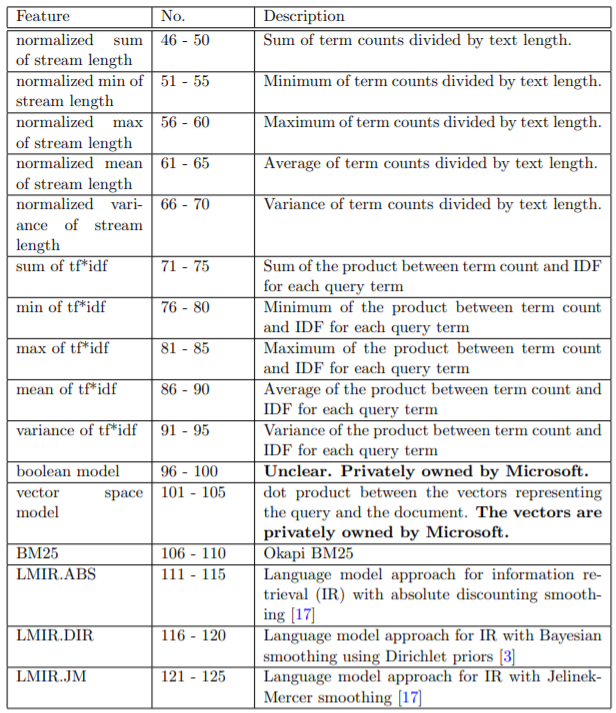
The [paper](https://dash.harvard.edu/bitstream/handle/1/25104739/tr-10-98.pdf;jsessionid=B8CB351674E40B3F232CBB113322C052?sequence=1) compares the different types of smoothing techniques with which a language model can be implemented. The deep learning models like Word2Vec also come under this category.

1. **Microsoft Learning to Rank** <https://www.microsoft.com/en-us/research/project/mslr/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fprojects%2Fmslr%2Ffeature.aspx>

<https://arxiv.org/pdf/1803.05127.pdf>



Microsoft implemented a LTR model on a web based dataset with the documents as web pages. Total 136 features were extracted by the team for every query-url pair. The features from 126 to 136 were all web-based like PageRank, outlink number, etc.. and are not useful to us. The features useful to us are mentioned below:

The features from 1 to 95 are very trivial features derived from classical retrieval models like tf, idf, and some basic operations on them - normalized tf, mean of tf, sum of tf-idf, etc. 

The features from 101 to 125 are a bit more complex each being a retrieval model in itself.

The boolean and the vector space model were not released to the public. We will discuss more on them and the possible extensions next.

1. **The Vector Space Model**

This [paper](https://dl.acm.org/doi/10.1145/361219.361220) is one of the pioneer papers in introducing vector space models to the field of information retrieval. Over the years, a lot of variations of it have been developed. As mentioned before, the vector space models represent queries and documents as vectors and then compute a scalar similarity value between them. The variation comes in as to how you set the weights in the vectors and which similarity you use.

Common weights used are tf, idf, tf-idf and the common similarity measures include cosine coefficient, jacquard coefficient and the inner product.

We will be exploring other possibilities of weights and similarity measures.

**Weights**

The [paper](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.104.3479&rep=rep1&type=pdf) compares the different term-weightings possibilities in the vector space model.

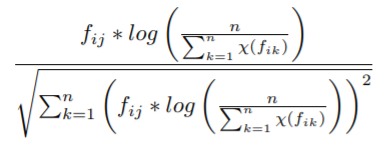
According to the paper, there are three components in a weighting scheme: aij = gi ∗ tij ∗ dj Where gi is the global weight of the ith term, tij is the local weight of the ith term in the jth document, dj is the normalization factor for the jth document.

The local weight tij deals with only the document at hand and is mainly variations of term-frequency tf. One important variant of this was using log(tf+1) because tf gives too much credit for weights that appear more. For instance, a word that appears ten times in a document is not usually ten times more important than a word that only appears one. Hence to reduce this effect one can use log(tf+1). Another variant to tackle this was the augmented-normalized-tf

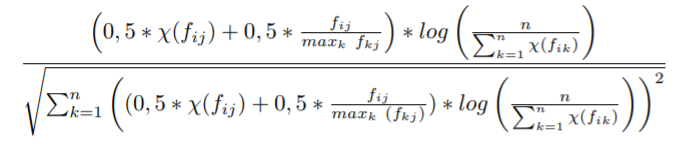
where (say k=0.5) it would give a value of 0.5 for a word appearing and a bonus (not more than 0.5) depending on the frequency. 

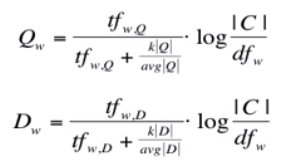
The global weight gi takes the whole corpus into account and helps determine the significance of the word. The IDF is the most effective global weight. Entropy is another measure that we can use which assigns a weight = 1 when the word only appears once in the whole corpus and weight = 0 when the document appears once in every document.

Dj is the normalization factor for a document's length. It's required as longer documents have undue advantage of matching more words but they may not be more relevant. The most common normalization technique is cosine normalization 

Salt and Buckley confirmed in the [paper](https://www.sciencedirect.com/science/article/abs/pii/0306457388900210) that most used weight is tf\*idf normalized by cosine.

In the [paper](https://conservancy.umn.edu/handle/11299/215454) they proposed (augmented normalized term frequency \* idf) normalized by cosine as the best term weighting scheme

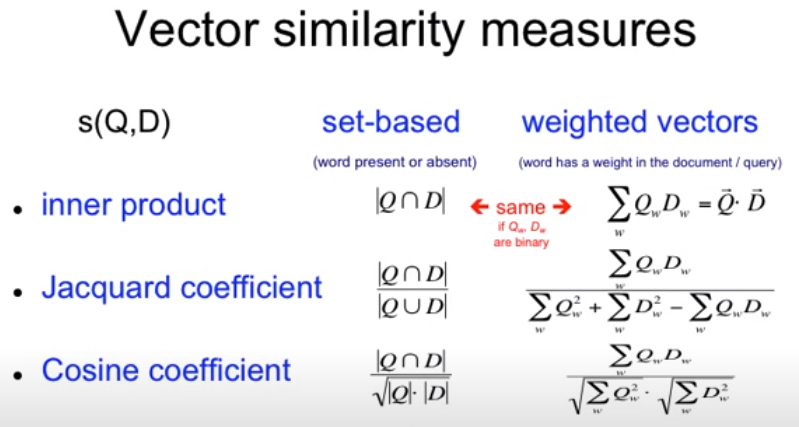


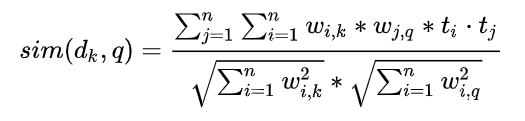
Another extension to the tf-idf weights was explained by Victor Lavrenko his [lectures](https://www.youtube.com/watch?v=nHnML6fauDg&list=PLBv09BD7ez_77rla9ZYx-OAdgo2r9USm4&index=10) in which the weights are given by

This is derived from the probabilistic model [Okapi BM25](https://en.wikipedia.org/wiki/Okapi_BM25) which is explained later.

**Similarity measures**

Common similarity measures include cosine coefficient, jacquard coefficient, inner product

The similarity function used in [GVSM](https://en.wikipedia.org/wiki/Generalized_vector_space_model) is another option 



Potential features:

* Representing query and document as vectors with different weights and then the similarity (scalar value) between them is treated as a feature.
* The vectors themselves can also be treated as features.
* The potential weights include tf, idf, tf-idf, tf-idf normalized by cosine, (augmented normalized term frequency \* idf) normalized by cosine
* Similarity measures include cosine coefficient, jacquard coefficient, inner product, GVSM similarity

1. **Latent Semantic Indexing**

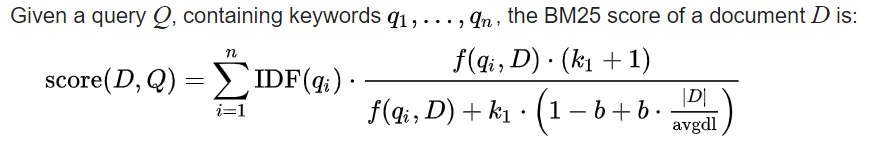
[LSA](https://en.wikipedia.org/wiki/Latent_semantic_analysis#Latent_semantic_indexing) (or LSI) is also one of the more common techniques used in general NLP which performs a truncated-SVD on the term-document matrix to map it to a ‘concept’ space and hence getting a better and dense representation of the text.

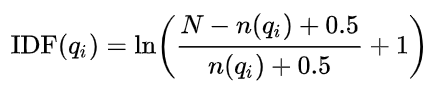
The [article](https://opensourceconnections.com/blog/2016/03/29/semantic-search-with-latent-semantic-analysis/) explains how LSA was implemented in [Solr Search Engine](https://prezi.com/z0dmaxdyuci0/equipping-solr-with-semantic-search-and-recommendation/). It also explains the common practices before using LSA like removing stopwords, using tf-idf and that it needs to be very finely tuned according to our problem to work well (like using better weights in the term document matrix like [BM25](https://en.wikipedia.org/wiki/Okapi_BM25) or using skipgrams)

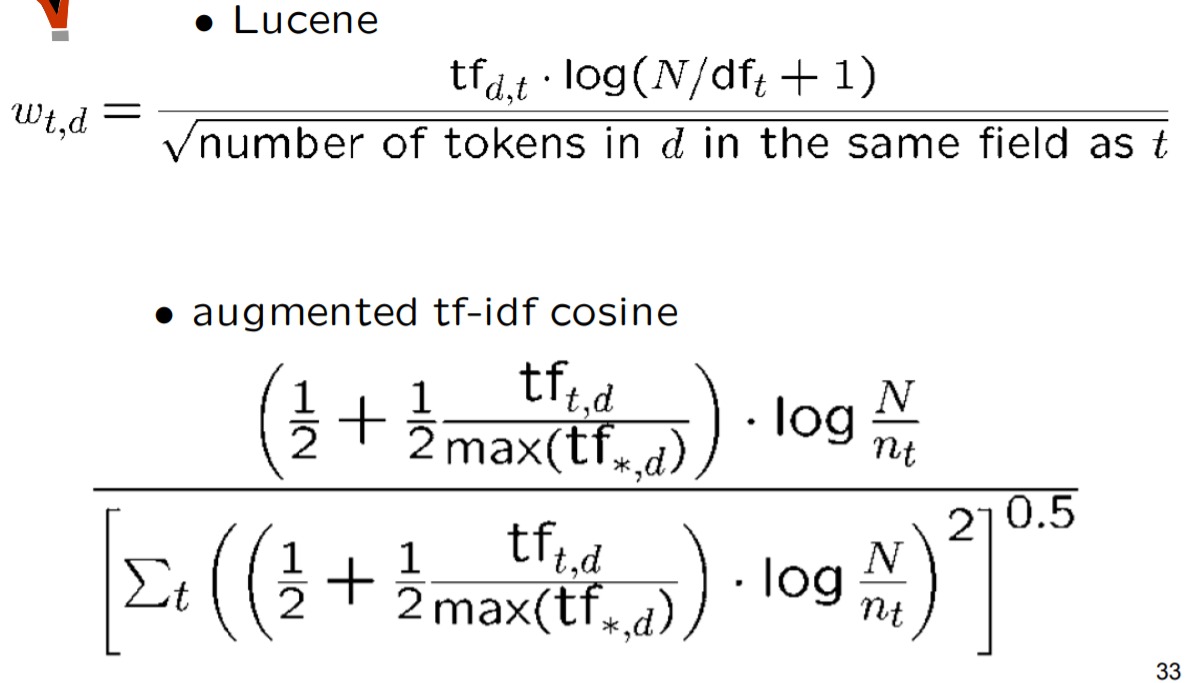
Potential feature:

* LSA with different weights and n-grams.

1. **BM25 and Extensions**

[Okapi BM25](https://en.wikipedia.org/wiki/Okapi_BM25) is one of the most popular ranking functions used in search engines since the 1980s. It is based on the [probabilistic relevance model](https://en.wikipedia.org/wiki/Probabilistic_relevance_model)for IR. 

Where the IDF is given by 

A couple of extensions of this were mentioned in [these lectures](http://www.ccs.neu.edu/home/jaa/CSG339.06F/Lectures/vector.pdf) 

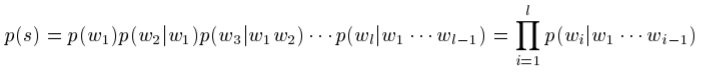
Potential features include:

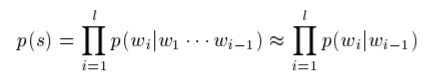
* BM25 ranking function
* Similarities using Lucene and augmented tf-idf cosine
* Score from Naive Bayes Classifier

1. **Language Models**

A [language model](https://en.wikipedia.org/wiki/Language_model#:~:text=From%20Wikipedia%2C%20the%20free%20encyclopedia,and%20phrases%20that%20sound%20similar.) is a type of a probabilistic model of IR. Very generally, for any collection of text D, and a given query q, it assigns the probability of the query q being randomly sampled from D i.e P(q|D). Here D is assumed to be a ‘language model’ i.e a probability distribution over the whole sequence of words.

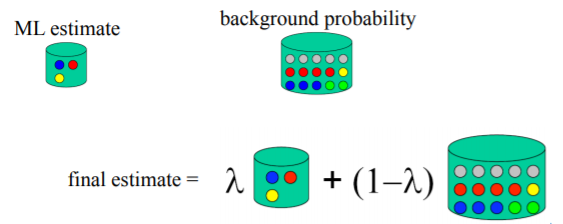
In the [lectures](https://www.youtube.com/watch?v=3DC5hlge_bs&list=PLBv09BD7ez_7Ke6U7yGBvfP4_Hau3ZGj2&index=1) by Victor Lavrenko he explains the concept behind coming up with such a model. In information retrieval, each document is considered a ‘language model’ and the probability P(q|d) is calculated on each document d which represents the relevance score. There are different kinds of language models like unigram, bigram, n-gram. It’s observed that uni-grams tend to perform the best.

The most widely-used language models, by far, are n-gram language models.Consider the case n = 2; these models are called bigram models.Let a sentence is composed of the words w1...wl then p(s) can be expressed as

In bigram models, we make the approximation that the probability of a word depends only on the identity of the immediately preceding word, giving 

One problem observed in the language models is data sparsity i.e a lot of terms in query don’t occur in the document which creates a big problem. For this we use smoothing. The [paper](https://dash.harvard.edu/bitstream/handle/1/25104739/tr-10-98.pdf;jsessionid=B8CB351674E40B3F232CBB113322C052?sequence=1) compares various smoothing techniques used for language models. It starts with *add-one smoothing* in which we assume every n-gram to occur one more time than it actually has. It then moves on to *additive smoothing* in which we pretend we’ve seen each n-gram δ times more than we have typically 0 < δ ≤ 1. It discusses a few more from *good-turing* to *modified-Kneser-Ney* which is considered to be the best. But these other smoothing techniques can be computationally expensive. We are gonna focus on the *Jelinek-Mercer* and *Dirichlet* smoothing.

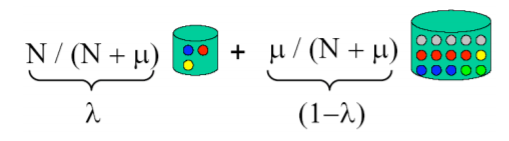
**Jelinek Mercer smoothing**

Jelinek-Mercer smoothing is based on the concept of interpolation. The problem with our ‘language model’ is that the data is very less in each document to call it a good approximation of the model which was the sole reason why we introduced smoothing. The idea here is to use a mixture of the whole data corpus and the document itself to calculate the probabilities. 

The ML estimate represents probability from the document that we are dealing with and the background-probability represents the probability calculated from the whole corpus. λ is a hyperparameter

**Dirichlet smoothing**

The problem with JM smoothing was that we use the same λ for each document. But we see that longer documents can be more reliable in providing better estimates and hence could get by with less smoothing. Hence to solve this we use Dirichlet smoothing which sets a bigger λ for documents with large lengths N. Here µ is a hyperparameter



Potential features include:

* Language model with Dirichlet and Jelinek Miller smoothing
* Word2Vec

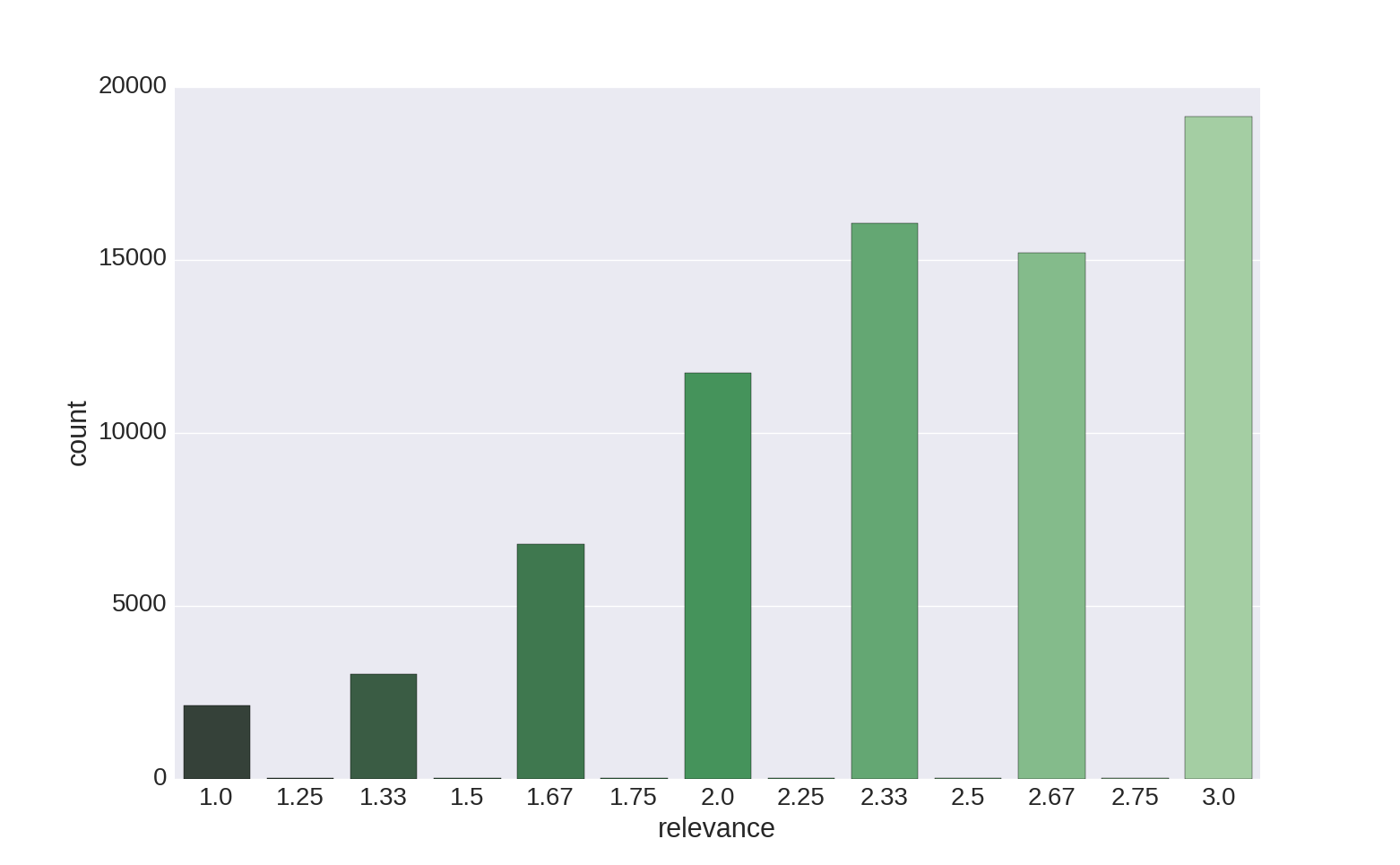
**Problem Specific Literature Survey**

1. **Overview of the different approaches**

<http://datasciencebar.github.io/blog/home-depot-produce-relevance-review/>

The blog gave an overview of the different approaches participants used to solve the problem.

**Data Exploration:**



* The relevance scores were an average of 3 people. Each person scored the relevance in categorical fashion only i.e {1 2 3}
* From here we can see that although the relevance scores were supposed to be real numbers, in the data we only find relevance scores in a categorical fashion.
* There is some overlap between products of train and test sets
* There is a considerable amount of overlap between search queries of product and test set. More than 80% of train queries are there in the test set. But a lot of test set queries are not in the train set which is a good thing.
* One more interesting observation was about the product\_id.

Large product id values were tending to have a relevance score lesser than the smaller product id values

Although this is something that was exploited by the kagglers, we should not be using it because it just doesn't make sense and was only used because it reduced the error.

**Pre-Processing**

* One problem was typos in the search queries. A simple approach to rectify this was, to build a list of terms that frequently occur in the queries, but don’t occur in the product description and manually fix these names. Other approaches were to implement your own spelling corrector or use Google's spelling corrector.
* units of measurement needed to be standardized eg. ft|foot|feet -> ft

pound|lb|pounds|lbs -> lb

* Stemming was implemented

**Feature Engineering**

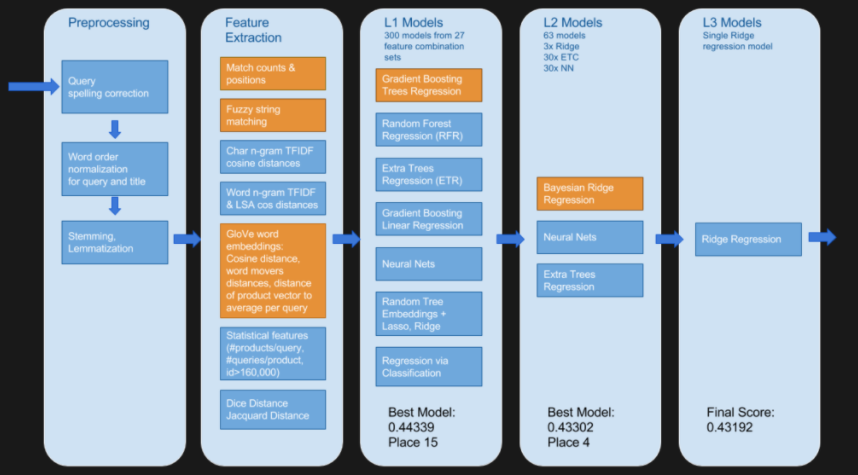
* One common feature was the similarity between description and query

This can be done using cosine similarities, BM-25, divergence from randomness sequential dependency model, etc

* To calculate similarities, we need to convert both into vectors. For this, common techniques were w2v and LSI.
* ‘word movers distance’ was used to calculate the similarity between longer pieces of text.
* Some other methods like fuzzy string matching, edit-distance were also used.
* POS tagger by nltk could also come in handy

**Algorithms**

* Most of the participants posed this as a regression problem. Many different models were used: from linear regression to random forest and gradient boosted regression trees.



* The winners used stacking of 3 layers of models. On the lowest layer they build 300 different regression models using various combinations of features. On the next layer 63 models combined predictions of the lowest layer models, and finally these 63 models were combined into a single model using ridge regression.
* Clearly, this should not be suitable for us because of real-world constraints

1. **2nd placed team interview**

<https://medium.com/kaggle-blog/home-depot-product-search-relevance-winners-interview-2nd-place-thomas-sean-qingchen-nima-68068f9f9ffd>

* The key to this competition was mostly preprocessing and feature engineering
* lot of the top teams had similar types of features, but different implementation
* Models

Started with random forest, extra trees and gbm-models

Furthermore xgboost and ridge were in our focus

The random forest and then extra trees did not help ensembles anymore.

So focused on xgboost, gbm and Ridge.

Best single model was a xgboost-model and scored 0.43347 on the public LB. The final ensemble consists of 19 models based on xgboost, gbm and Ridge. The xgboost-models were made with different parameters including binarizing the target, objective reg:linear, and objective count:poisson. Found that the Ridge Regression helped in nearly every case, so they included it in the final ensemble.

* Note that their best performance by a single model was 0.43347 which would rank 6th in the competition hence it’s possible to build a single model also with excellent performance. The ensemble models were mainly used for minute improvements.

1. **8th placed team interview**

<https://dimleve.medium.com/how-to-predict-search-terms-relevance-fc16217bdbf4>

Feature Generation :

* Distance/String Similarities

Cosine distance, n-gram overlapping distance, longest common subsequence, tf-idf distance and more similar distance measure were used between search\_query and (title, product, attributes) using several n-gram segmentation

* Other features

Frequency of query

Count of POS tags

Count of stop words and spelling errors in the query

Count number of attributes

Brand popularity

1st-last word similarities to the title

Query-products Intra similarity

Length of query, title, attributes

Model

* XGBoost, Bagging, Stacking

1. **Solution by Peng Xu**

<http://billy-inn.github.io/papers/cmput690.pdf>

**Preprocessing**

Parsing, stemming, removing Html content and punctuations, correcting common spelling mistakes, and so on. It was found that most attributes are extracted from the product description except the ‘brand’. So this brand attribute was created as a separate feature.

Three different schemes to handle the text information in the data were used :

* D1: Just apply basic operations to the text information.
* D2: Apply basic operations and remove the common stopwords.
* D3: Apply basic operations and remove the stopwords and numbers

**Feature Extraction**

Uses techniques from IR and NLP

Basic Features :

- brand feature

- Length of (query, title, description,brand)

- Common words between query and (title, description, brand)

- whether the last word in search\_query is in (title, description)

The above feature was taken because intuitively the last word of the query seems to be the most important word

- len of (title, description, brand) / len of query

Correct Search Terms : Lots of search terms had spelling mistakes. Those were corrected

Latent Semantic Indexing:

Applied the LSI on the term document matrices of search terms, title, description and brand separately and treat the low-rank approximations as features directly. Then, combine all the text information including title and description together for each product as documents and apply LSI on the corresponding term document matrices. Then, transform the query to the semantic space by multiplying Vk as illustrated above. Finally, treat the cosine distance between document vectors and query vectors as features

Query Expansion by w2v:

First, all the words in the query are converted to w2v vectors. Then let q be a user query represented by a bag of terms, q = [t1, t2, . . . , t|q| ].In order to expand a query q, we follow • For each t ∈ q, collect the k-most similar terms to t using w2v. Include the new terms into the query set giving us q’. After query expansion, we can use the expanded queries to recalculate all the basic features and LSI features, which were experimentally very effective.

Language Models : Indri Search

**Models**

- explored 3 different models : SVMs, RF and GBDT

**Results**

Table 1: CV results on different feature sets Feature (Using dataset D1)

Set Baseline F1 F2 F3 F4

RMSE 0.4844 0.4830 0.4799 0.4797 0.4702

• Baseline: Basic features only.

• F1: Basic features + Basic features based on corrected search terms.

• F2: Basic features + Basic features based on expanded search terms.

• F3: Basic features + Language model based relevance scores & rankings.

• F4: Basic features + Latent semantic indexing features.

Dataset D1 D2 D3

RMSE 0.4563 0.4589 0.4659

Models GBT RF SVR

RMSE 0.4563 0.4640 0.4653

1. **Solution by RR Iyer, et al.**

<https://arxiv.org/pdf/2001.04980.pdf>

Preprocessing

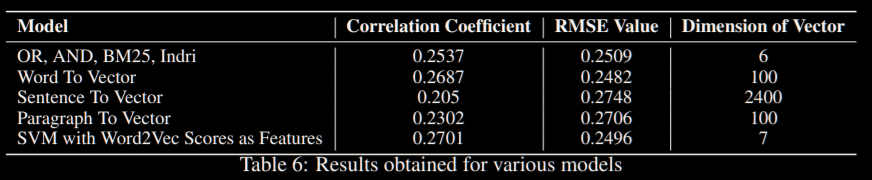
* Basic preprocessing + spelling correction

\*For modelling they used SVM

Feature Engineering:

* Tried a lot of different feature sets
* Unigram features for search, title and description
* OR and AND operators
* relevance scores from BM25 and Indri topic models
* W2V, Sent2Vec, Para2Vec

Results



Observations:

* Traditional topic models (BM25 and indri) combined with simple AND, OR operator gave a pretty decent RMSE score with the dimensions of input data being very less
* W2V gave a considerable improvement
* Sent2Vec and Para2Vec did not improve the model in fact lead to quite a decline in performance. One possible reason for this could be because the text data we have (search, title and description) are not very well written as the description mostly products attributes just put together in one corpus without any connection between sentences. Yet W2V did perform pretty decently.
* The scores from the w2v model were used as features in another model but the performance did not improve for some reason.
* The DL models were used on other datasets with more training data and the performance increased considerably. Hence more data could potentially reduce the rmse by a lot.