# Approach for the Quora Question Pairs Challenge

YesOfCourse

June 9, 2017

**Abstract**

In the Quora Question Pairs Challenge, we were asked to build a model to classify whether question pairs are duplicates or not (multiple versions of the same question). This documents describes our team's solution which can be divided into diffrent parts: Pre-processing, Feature Engineering, Modeling and Post-processing.

## Personal details

Team Name: **YesOfCourse**

Members:

- **Liang Pang**
  Email: `pangliang@gmail.com`
- **Yixing Fan**
  Email: `fanyixing@gmail.com`
- **Jianpeng Hou**
  Email: `houjp1992@gmail.com`
- **Xinyu Yue**
  Email: `yuexinyu@gmail.com`
- **Guocheng Niu**
  Email: `niuox@gmail.com`

Competition: Quora Question Pairs

# Contents

## 1 Summary

Our solution consisted of four main parts: Pre-processing, Feature Engineering, Modeling and Post-processing. What's more, we developed a light weight Machine Learning framework **FeatWheel** to help us to finish ML jobs, such as feature extraction, feature merging and so on.

In pre-processing, we process the text of data with text cleaning, word stemming, removing stop words and shared words and can form different versions of original data. In feature engineering, we extracted features based on various versions of data. The features can be classified in to three categoriesStatistical Features, NLP Features and Graph Features. In modeling, we build deep models, boosting models (using XGBoost, LightGBM) and linear models (Linear Regression) and build a multi-layer stacking system to ensemble different models together. As we all know, the distribution of the training data and test data are quite different, so we made post-processing on the prediction results. We cut the data into different parts according to the clique size and rescale the results in different parts.

### 1.1 Flowchart

The flowchart of our method is shown in Fig. 1.

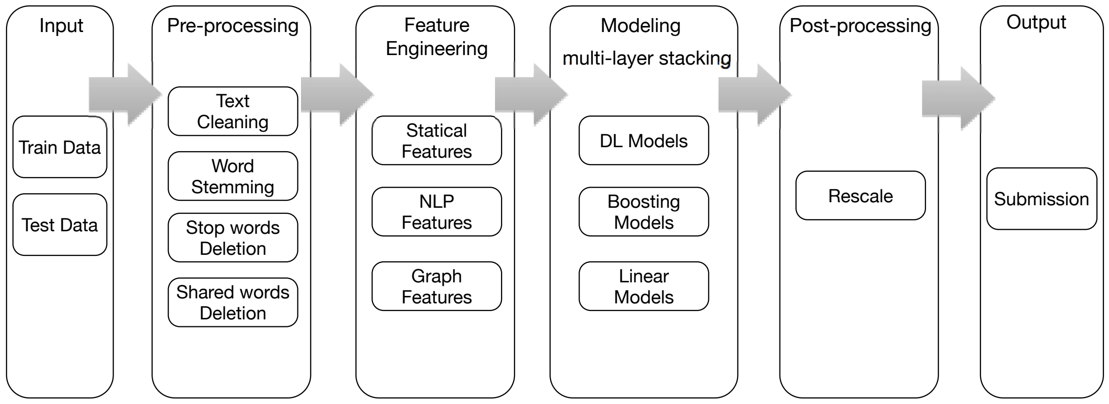

Figure 1: The flowchart of our method.

### 1.2 Submission

Submissions were evaluated on the log loss between the predicted values and the group truth. In specific, the best single model we have obtained during the competition was an XGBoost model with tree booster of Public LB score **0.12653** and Private LB score **0.13067** (without post-process). Our final submission was a stacking result of multiple models. This submission scored **0.11450** on Public LB and **0.11768** on Private LB (with post-process), ranking **4** out of **3396** teams.

## 2 Pre-processing

Pre-processing

## 3 Feature Engineering

Feature Engineering

## 4 Modeling

Modeling

## 5 Post-processing

Post-processing

## 6 ML Framework

ML Framework

## 7 Acknowledgement

Acknowledgement

A few steps are applied to clean up the text. These steps were done separately by the two parts of our team. Unless otherwise noted, Igor&Kostia's part is implemented in `homedepot_functions.py` and `text_processing.py`. For Chenglong's part please see `data_processor.py`.

### 7.1 Initial steps

They included making the text lowercase, replacement or cleaning of some symbols and some pattern replacement. Some examples of the replacements are shown in the table below:

| Symbol | Replacement |
|---|---|
| &amp; | & |
| &nbsp; | |
| ; | . |
| : | . |
| + | |
| * | x |

Other text cleaning included:

- Removed words in parentheses or replace parentheses with dots.

- Removed commas between digits (for example, $10,000$ was replaced with 10000).
- Added dot and space between a digit (left) and a letter (right).
- Added space between at least three letters (left) and a digit (right).
- Replaced \ or / between letters with a space.

The punctuation was important for extraction of the most important words from the text or tagging words using *NLTK.pos_tagger()* function. Otherwise, it was removed at all.

We also noticed that product description is produced by concatenation of text lines, so that some words have capital letters in the middle (like *firstSecond*). We added a space in such words, but only in those that are not found in product title. So, we would transform the string *firstSecond* to *first Second*, but leave *InSinkErator* as it is.

## 7.2 Spell Correction

It is mostly done for search term which contains many spelling errors.

- Replacement of search terms using **the Google dictionary** from the forum [1].
- **Manual replacements** posted on the forum.
- **Automatic spell checker** (Igor&Kostia). It is implemented in `text_processing.py`. It tries to find replacement for words from search term which are not found in product title (in this paragraph we name them *'misspelled'* words). First, the algorithm checks whether the misspelled word is actually a combination of two separate words. Second, it searches for a good replacement in product titles that are matched to queries with the misspelled words. Finally, all product title is used as a vocabulary to find an appropriate replacement.

## 7.3 Concatenation of Numbers with Measure Units

This part is best illustrated by the following code:

```
s = re.sub('(?<=[0-9])[\ ]*pound[s]*(?=\ |$|\.)', '-lb ', s)
s = re.sub('(?<=[0-9])[\ ]*lb[s]*(?=\ |$|\.)', '-lb ', s)
s = re.sub('(?<=[0-9])[\ ]*gallon[s]*(?=\ |$|\.)', '-gal ', s)
s = re.sub('(?<=[0-9])[\ ]*gal(?=\ |$|\.)', '-gal ', s)
```

## 7.4 Thesaurus Replacement

In total, we made about 200 replacements that might be described as follows:

- **'Stemming':** *winterizing* $\rightarrow$ *winterizer.*
- **Synonym replacement:** *lamp* $\rightarrow$ *light.*
- **Abbreviation convertion:** *bbq* $\rightarrow$ *barbeque.*

- **Uniform spelling.** Many words can be spelled differently in the dataset: *mail box* and *mailbox*, *fibre* and *fiber*, *chain saw* and *chainsaw*, *aluminium* and *aluminum*. We made the spelling of some of such words uniform throughout all text columns.
- **Brands and materials:** *lg electronics* $\rightarrow$ *lg* and *medium density fiberboard* $\rightarrow$ *mdf*

## 7.5 Part-of-Speech Tagging

It was done by *NLTK.pos_tagger()* function.

## 7.6 Extracting information about brands and materials

We used appropriate rows from file `attributes.csv`. Then we extracted brand and materials from the available docs (search term, product title, product description, attribute bullets). Later we will use these docs with or without brand/material information to generate different sets of features.

It should be noted that some 'true brand names' are long: for example, *'BEHR Premium Plus'*. But it is likely to be as simple as 'BEHR' in the query. So, once we found such a long brand name within a doc, we would replace it with a shorter version. In total, we replace 131 such long brand names.

## 7.7 Dealing with Different Parts of Query or Product Title

We noticed a structure in product title which helped us to find the most important part of the document. For example, in product title *'Husky 52 in. 10-Drawer Mobile Workbench with Solid Wood Top, Black'* the product is workbench, not wood top. To deal with multiple patterns present in product title, we had to elaborate a complex algorithm, which extracted important words.

We also extracted the top trigram [2]. Igor and Kostia started this work independently, so in their notation the trigrams words are denoted as (*before2thekey*, *beforethekey*, *thekey*), where *thekey* is the last word in the trigram.

We separately dealt with text after words *with*, *for*, *in*, *without*.

## 7.8 Final Steps

To generate the most of the features we used stemmed text as an input.

## 7.9 Other

These part describes some of the processing done on Chenglong's side. Most of the processing are similar as Igor&Kostia's, including lower case conversion, unit conversion, removing comma between digits, manual word replacement, google spelling correction, lemmatizing, stemming etc. However, they are done in different way, which is helpful for introducing diversity. In addition, we would like to mention a few words about the following parts.

### 7.9.1 Splitting Lower Case Words and Upper Case Words

In the production_description, there are words concatenated without spacing. For example,

- hidden from *viewDurable* rich *finishLimited* lifetime *warrantyEncapsulated* panels
- Dickies quality has been built into every *product.Excellent visibilityDurable.*

This might due to that while creating the product description file, someone concatenated all the lines in a description file without adding spaces in between. To deal with such cases, we split the 'lower[.?!]UPPER' pattern. For the above two examples, they become

- hidden from *view Durable* rich *finish Limited* lifetime *warranty Encapsulated* panels
- Dickies quality has been built into every *product. Excellent visibility Durable.*

### 7.9.2 Mapping English Number to Digit

We map English numbers, e.g., "one" and "two" to 1 and 2.

### 7.9.3 Extracting Product Name

We extract product name from both search_term and product_title using a similar method described in the 3rd place solution for CrowdFlower competition [2].

### 7.9.4 Home-made Spelling Checker

We have built our own home-made spelling checker, which is inspired by Peter Norvig's post [3]. Though not used in the final submission, it also helps to identify possible misspellings. Please see `PeterNovigSpellingChecker` and `AutoSpellingChecker` classes in `spelling_checker.py` for details. Following is some of the output

```
Run AutoSpellingChecker at search_term
Building word corpus
Building error correction pairs
Building query correction map
                original query                  corrected query
                    linen tile                       liner tile
  ekena millwork cherry corbel     ekena millwork cherry core
    potting soil chemical free     potting pool chemical free
       ge dishwasher stainless        ge dishwasherstainless
    led shop light makita light    led shop light making light
     500 ft. thhn stranded blue     500 ft. thwn stranded bare
 roman shade cord idler pulley   roman shade come idler pulley
```

| | |
|---|---|
| `gold panning pan` | `gold hanging pan` |
| `green quadrato pot` | `green quadro pot` |

# 8 Feature Extraction

## 8.1 Chenglong's Part

As in CrowdFlower, we have many pairwise features that measures the similarity/distance between search_term and product_title, or search_term and product_description. So we implement a base class `BaseEstimator` that takes an `obs` (e.g., search_term) and `target` (e.g., product_title) as input. For some features, e.g., intersected position in CrowdFlower or ngram level similarity/distance, we need to use various aggregation methods to generate statistics such as max and mean.[1] `BaseEstimator` also serves this purpose. Most of our feature generators are based on `BaseEstimator`.

On the top, we have `StandaloneFeatureWrapper` and `PairwiseFeatureWrapper` for generating standalone features and pairwise features respectively, using various feature generators, between various `obs` and `target` (e.g., search_term vs. product_title and search_term vs. product_description). Please find details for these classes in the module `feature_base`.

All the similarity/distance measurements are in the module `./utils/dist_utils`.

### 8.1.1 Basic Features

We generated some basic descriptive features with file `feature_basic.py`.

- **DocId**: numerical encoding of `obs` (e.g., search_term, product_title; only used with tree model)
- **DocIdOneHot**: onehot encoding of `obs` (e.g., search_term, product_title; only used with linear model)
- **DocIdEcho**: return the original Id of `obs`; this is only for product_uid
- **ProductUidDummy1**: dummy 1 for product_uid
- **ProductUidDummy2**: dummy 2 for product_uid
- **ProductUidDummy3**: dummy 3 for product_uid
- **DocLen**: length of `obs`
- **DocFreq**: frequency of `obs` in the corpus
- **DocEntropy**: entropy of `obs`
- **DigitCount**: count of digit in `obs`
- **DigitRatio**: ratio of digit in `obs`
- **UniqueCount_Ngram**: count of unique word ngram in `obs`
- **UniqueRatio_Ngram**: ratio of unique word ngram in `obs`

[1]For some features, we also use double aggregation.

- **AttrCount**: count of attributes in product_attribute_list
- **AttrBulletCount**: count of bullet attributes in product_attribute_list
- **AttrBulletRatio**: ratio of bullet attributes in product_attribute_list
- **AttrNonBulletCount**: count of non-bullet attributes in product_attribute_list
- **AttrNonBulletRatio**: ratio of non-bullet attributes in product_attribute_list
- **AttrHasProductHeight**: whether or not contains product height in product_attribute_list
- **AttrHasProductWidth**: whether or not contains product width in product_attribute_list
- **AttrHasProductLength**: whether or not contains product length in product_attribute_list
- **AttrHasProductDepth**: whether or not contains product depth in product_attribute_list
- **AttrHasIndoorOutdoor**: whether or not contains indoor/outdoor info in product_attribute_list

For using with linear model, we apply `np.log1p` to some of the above features, e.g., `DocLen` and `DocFreq` etc.

### 8.1.2 Distance Features

We generated the following features with file `feature_distance.py`.

- **JaccardCoef_Ngram**: jaccard coefficient of `obs` word ngram and `target` word ngram
- **DiceDistance_Ngram**: same as above but using dice distance
- **CompressionDistance**: compression distance between `obs` and `target` (document level)
- **EditDistance**: edit distance between `obs` and `target` (document level)
- **EditDistance_Ngram**: same as above, but in word ngram level with double aggregation. To see what double aggregation means, we take for example the feature: `EditDistance_Bigram_Max_Min_search_term_x_product_title`. It means: for each word bigram in search_term, compute the maximum value of the edit distance between this bigram and every word bigram in product_title; then take the minimum value of those maximum values as output feature. *Double aggregation allows to capture the similarity between `obs` and `target` (e.g., search_term and product_title) in word level.*

### 8.1.3 Doc2Vec Features

We generated the following features using file `feature_doc2vec.py`.

- **Doc2Vec_Vector**: doc2vec vector of `obs` (not used in final submission)
- **Doc2Vec_Vdiff**: difference between doc2vec vector of `obs` and doc2vec vector of `target` (not used in final submission)

- **Doc2Vec_CosineSim**: cosine similarity between doc2vec vector of `obs` and doc2vec vector of `target`
- **Doc2Vec_RMSE**: RMSE between doc2vec vector of `obs` and doc2vec vector of `target`

Doc2Vec model are trained using `embedding_trainer.py` with the provided data, i.e., search_term, product_title, product_description, and product_attribute.

#### 8.1.4 First and Last Ngram Features

We generated the following features using file `feature_first_last_ngram.py`.

- **FirstIntersectCount_Ngram**: how many word ngram in `target` closely matches with the first ngram of `obs`. We choose closely match instead of exactly match in the consideration of that there might be some misspelling characters in the word, e.g,. *conditionar* (which should probably be *conditioner*). The closeness is measured with edit distance.
- **LastIntersectCount_Ngram**: same as **FirstIntersectCount_Ngram** but considering last ngram of `obs`
- **FirstIntersectRatio_Ngram**: same as **FirstIntersectCount_Ngram** but normalized with the number of word ngram in `target`
- **LastIntersectRatio_Ngram**: same as **LastIntersectCount_Ngram** but normalized with the number of word ngram in `target`

Compared to the **Intersect Count Features** described below, such features are designed to give more weight on the first and last ngram. One a more general level, one can add weighting according to the position of the word ngram in the search_term or product_title.

#### 8.1.5 Group Relevance Based Distance Features

We generated the following features using file `feature_group_distance.py`.

- **GroupRelevance_Ngram_Jaccard**: group by (search_term, relevance) based distance features; (single) aggregation with various methods, e.g., max and mean etc. (see Sec 3.2.2 in Chenglong's Solution of CrowdFlower for details).

However, such features are NOT used in the final submission as we haven't achieved much improvement with these features. In CrowdFlower (where such features are reported with success in the 1st and 3rd solutions), there are about 260 unique queries and for each query and relevance combination, there are enough samples to obtain a reliable group based distance. However, in this competition, there are approximately 7.5 samples for each search_term. Considering that there are 7 major relevance levels, there is not enough samples for such group based distance. This might be the reason why such features fail to work here. A possible way out would be to cluster similar search_term first. For example, *furnace air filters* and *furnace filters with sensor* might be considered as within the same group/cluster.

### 8.1.6 Group Relevance Features

We generated the following features using file `feature_group_relevance.py`.

- **GroupRelevance**: group by search_term/product_title and compute some statistics (e.g., max and mean) of the relevance in the group.

However, such features are currently NOT used in model building due to similar reason mentioned in Sec. 8.1.5.

### 8.1.7 Intersect Count Features

We generated the following features using file `feature_intersect_count.py`.

- **IntersectCount_Ngram**: count of word ngram of `obs` that closely matches any word ngram of `target`.
- **IntersectRatio_Ngram**: same as **IntersectCount_Ngram** but normalized with the total number of word ngram in `obs`
- **CooccurrenceCount_Ngram**: count of closely matching word ngram pairs between `obs` and `target`.
- **CooccurrenceRatio_Ngram**: same as **CooccurrenceCount_Ngram** but normalized with the total number of word ngram pairs between `obs` and `target`.

As mentioned in Sec. 8.1.4, you can weight such feature according to the position of the word ngram in the search_term or product_title, besides extracting the intersect position as features as described in the following section.

### 8.1.8 Intersect Position Features

We generated the following features using file `feature_intersect_position.py`.

- **IntersectPosition_Ngram**: see Sec 3.1.3 in Chenglong's Solution of CrowdFlower for details
- **IntersectNormPosition_Ngram**: see Sec 3.1.3 in Chenglong's Solution of CrowdFlower [4] for details

### 8.1.9 Match Features

We generated the following features using file `feature_match.py`.

- **MatchQueryCount**: count of search_term that occurs in `target`
- **MatchQueryRatio**: same as **MatchQueryCount** but normalized with the length of `target`
- **LongestMatchSize**: longest match size between `obs` and `target`
- **LongestMatchRatio**: same as **LongestMatchSize** but normalized with the minimum length of `obs` and `target`
- **MatchAttrCount**: count of attribute in product_attribute_list that matches search_term

- **MatchAttrRatio**: ratio of attribute in product_attribute_list that matches search_term
- **IsIndoorOutdoorMatch**: whether the indoor/outdoor info in seach_term and product_attribute_list match

#### 8.1.10 Query Quality Features

We generated the following features using file `feature_query_quality.py`.

- **QueryQuality**: measures the quality of search_term using the edit distance among various versions, e.g., original search_term and cleaned search_term.
- **IsInGoogleDict**: whether the search_term is in the Google spelling correction dictionary or not.

#### 8.1.11 Statistical Cooccurrence TFIDF Features

We generated the following features using file `feature_tfidf.py`.

- **StatCoocTF_Ngram**: for each word ngram in `obs`, count how many word ngram in `target` that closely matches it and then aggregate to obtain various statistics, e.g., max and mean.
- **StatCoocTFIDF_Ngram**: same as **StatCoocTF_Ngram** but weight the count with IDF of the word ngram in the `obs` corpus
- **StatCoocBM25_Ngram**: same as **StatCoocTFIDF_Ngram** but with BM25 weighting

All the feature above used closely matching.

#### 8.1.12 Vector Space Features

We generated the following features using file `feature_vector_space.py`.

- **LSA_Word_Ngram**: LSA of `obs` word ngram
- **LSA_Char_Ngram**: LSA of `obs` char ngram
- **LSA_Word_Ngram_Cooc**: LSA of cooccurrence word ngram between `obs` and `target`; see Sec. 3.3.2 in Chenglong's Solution of CrowdFlower [4] for how we generate cooccurrence terms.
- **LSA_Word_Ngram_Pair**: LSA after concatenating TFIDF of `obs` word ngram and TFIDF of `target` word ngram
- **LSA_Word_Ngram_CosineSim**: cosine similarity between LSA of `obs` word ngram and LSA of `target` word ngram
- **LSA_Char_Ngram_CosineSim**: same as **LSA_Word_Ngram_CosineSim** but for char ngram
- **TFIDF_Word_Ngram_CosineSim**: same as **LSA_Word_Ngram_CosineSim** but use TFIDF instead of LSA

- **TFIDF_Char_Ngram_CosineSim**: same as **LSA_Word_Ngram_CosineSim** but use TFIDF instead of
- **TSNE_LSA_Word_Ngram_Cooc**: 2-D TSNE of **LSA_Word_Ngram_Cooc**
- **TSNE_LSA_Word_Ngram_Pair**: 2-D TSNE of **LSA_Word_Ngram_Pair**
- **CharDistribution_Ratio**: ratio between the char distribution of `obs` and char distribution of `target`
- **CharDistribution_CosineSim**: same as **CharDistribution_Ratio** but use cosine similarity as similarity measurement
- **CharDistribution_KL**: same as **CharDistribution_Ratio** but use KL divergence as distance measurement

### 8.1.13 Word2Vec Features

We generated the following features using file `feature_word2vec.py`.

- **Word2Vec_N_Similarity**: n_similarity between `obs` and `target`
- **Word2Vec_Centroid_RMSE**: RMSE between the centroid vector of `obs` and centroid vector of `target`
- **Word2Vec_CosineSim**: double aggregation features. Take for example the feature `Word2Vec_CosineSim_Max_Min_search_term_x_product_title`. It means: for each word in search_term, compute the maximum value of the cosine similarity between this word and every word in product_title; then take the minimum value of those maximum values as output feature.

Word2Vec model are trained using `embedding_trainer.py` with the provided data, i.e., search_term, product_title, product_description, and product_attribute. We also use pretrained models trained with Google News corpus and Wikipedia corpus as listed here.

### 8.1.14 WordNet Similarity Features

We generated the following features using file `feature_wordnet_similarity.py`.

- **WordNet_Path_Similarity**: double aggregation features. Take for example the feature `WordNet_Path_Similarity_Max_Min_search_term_x_product_title`. It means: for each word in search_term, compute the maximum value of the similarity between this word and every word in product_title; then take the minimum value of those maximum values as output feature. The similarity between two words is measured as following: for each word, we first obtain all its synset set then compute the maximum `path_similarity` between these synset sets.
- **WordNet_Lch_Similarity**: same as **WordNet_Path_Similarity** but use `lch_similarity` instead of `path_similarity`
- **WordNet_Wup_Similarity**: same as **WordNet_Path_Similarity** but use `wup_similarity` instead of `path_similarity`

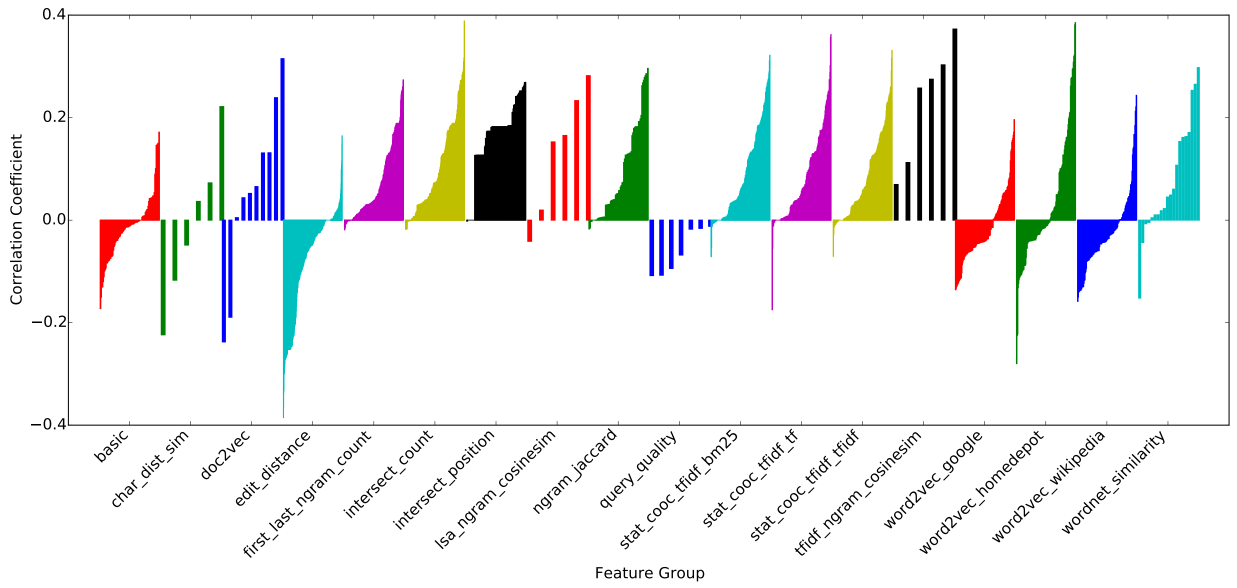

Figure 2: Correlation coefficients of Chenglong's features with target relevance for each feature group.

Such features are quite time consuming. A possible way to reduce the computation time would be to only consider those *important word ngrams* as described in Igor&Kostia's part, i.e., Sec. 8.2.6.

### 8.1.15 Feature Correlation with Target

Fig. 2 shows the correlation coefficients of our features with target relevance for each feature group. As shown, most of our feature groups exhibit high correlation with the target relevance indicating strong predictive power. Three feature groups, i.e., `edit_distance`, `intersect_count`, and `word2vec_homedepot`, contain features with absolute correlation coefficient around 0.39. For the `word2vec` feature group, we have observed that features generated using model trained with Homedepot provided data exhibits higher correlation with the target, than those extracted with pre-trained Google News model and Wikipedia model.

## 8.2 Igor&Kostia's Part

Unless otherwise noted, the feature generation is done in file `feature_extraction1.py`. Feature importances of 20 top features for our benchmark GradientBoostingRegressor model are shown in Fig. 3.

### 8.2.1 Basic Features

- **len_of_*strings***: number of words in different available strings (e.g. search term, search term without brand, search terms without brand and material, brands/materials

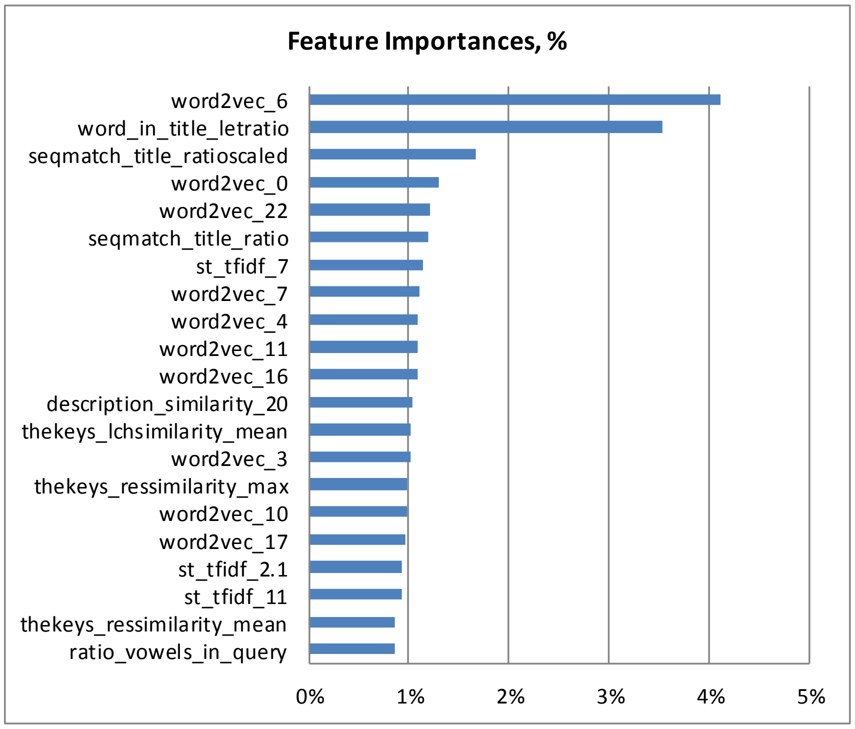

Figure 3: Feature importances of top 20 features for benchmark GradientBoostingRegressor model (Igor&Kostia).

Note: **word2vec_6** was calculated as *n_similarity()* between stemmed search term and stemmed product title.

in search terms, words with digits in search term, the most important words in search term, nouns/verbs/adjectives in search terms, words after *'for'* and *'with'* in search terms and after *'without'* in product titles, and similar variables for product title, product descriptions, product attributes etc.)

- **size_of_brand/material_in_*strings***: number of distinct brands and materials in search term, product title, product description, attribute bullets.
- **perc_digits_in_*strings***: percentage of digits in different text variables.
- **has_attributes_dummy**.
- **no_bullets_dummy**.

The following three variables, which were inspired by [5], are also used to measure the 'amount' of information in the search term:

- **Ratio of meaningful words in search term**: the word is meaningful if WordNet knows synsets for the word.
- **Average length of word in search term**.
- **Ratio of vowels in search terms**.

### 8.2.2 Distance Features

For many stemmed variable pairs *str1* and *str2* (e.g. stemmed search term and stemmed product title) we generated a tuple of up to five features:

- ***variable_name*_num**: number of unique common words/bigrams in *str1* and *str2*.
- ***variable_name*_sum**: number of all (non-unique) common words/bigrams.
- ***variable_name*_let**: number of characters in the unique common words/bigrams.
- ***variable_name*_numratio**: ratio of unique common words to the total number of words in *str1*.
- ***variable_name*_letratio**: ratio of the number of characters in unique common words to the total number of characters (excluding spaces) in *str1*.

Here *str1* could be search term, search term without digits, words after *'for'/'with'* in search term, important/not important nouns in search term, verbs/gerunds/adjectives/adverbs in search term, words after *'without'* in product title.

Similar features were also calculated for common 1-4 grams and 2-3 terms in `grams_and_terms_features.py` in the same way as in Chenglong's CrowdFlower winning solution. The Jaccard of Dice coefficients were used to measure the distance between strings.

We also calculated similar distance features, but instead of using the common word as a measure of closeness of the strings, we considered that the strings have similar words if the Damerau-Levenshtein distance between those words is not large (please see files `grams_and_terms_features.py` and `dld_features.py`). We consider that the distance between $word_1$ and $word_2$ is not very large if it is less than

$$\min\left(3, \max\left(\text{len}(word_1) - 3, 1\right)\right).$$

For example, words *'alumanam'* and *'aluminum'* are distinct, but their Damerau-Levenshtein distance is small (equal to 2 because only two letters are different), therefore we consider the words the same when we calculate the appropriate distance features. This approach helped us to extract much information from misspelled words.

Another way to deal with misspelled strings was using the similarity in terms of character sequences. In the features above, the intersection between *'trafficmaster'* and *'traffic master'* would be zero. But if we compare the character sequences with the help of *difflib.SequenceMatcher()* function, the similarity would be high. We used this approach to compare search terms to the text from the other variables.

### 8.2.3 Brands and materials

For the lists of brands/materials in the search term and another text variable (e.g. product title) or brands/materials from `attributes.csv` we calculated:

- **Number of fully matched brands/materials**: the number of brands/materials from search term found in the text variable.
- **Number of partially matched brands/materials**: the match is considered partial if the brand/material from search term is a part of the brand/material for the other text variable (e.g. *'BEHR'* and *'BEHR Premium'*).
- **Number of brands/materials with assumed match**: the match is assumed if the first word of brand/material from search term is a part of the brand/material for the other text variable (e.g. *'American Standard'* and *'American Woodmark"*).
- **Number of brands not found the text**.

We also tried to incorporate the information from those features into one *convoluted* feature, which output would be:

- 3 if all brands/materials are fully or partially matched;
- 2 if the match for all brands/materials is full or partial or assumed;
- 1 if the match for at least one brand/material is full or partial or assumed, but also at least one brand is not found in the text;
- 0 if there are no brands in search term;
- $-1$ if there no brands/materials in the text variable;
- $-2$ if all brands are mismatched.

### 8.2.4 TFIDF features

We calculated TFIDF features from 1-4 gramms and 2-3 terms in the intersection between search term and the other text variables. The corresponding code is in `grams_and_terms_features.py`.

Also, in `feature_extraction1.py` we used TFIDF to assess how much information there is in search term or intersection between search term and the other text. We used product title, product description or attribute bullets as vocabulary for TFIDF and calculated the weights for the corresponding words. Then we used the corresponding sum of weights to

(a) measure the **total** amount of information in the search term in terms of words found in the particular text variable;

(b) measure the amount of information in the **intersection** between the search term and the text variable.

We also calculated similar features by multiplying the TFIDF weights by the number of characters in the corresponding words (the names of these features end with *_let* instead of *_num*). Examples of the features are:

- **tfidf_title_num, tfidf_title_let:** amount of information in the search term in terms of words found in product title;
- **tfidf_matchtitle_num, tfidf_matchtitle_let:** amount of information in the intersection between search term and product title measured in terms of words found in product title;
- **tfidf_matchtitle_stringonly_num, tfidf_matchtitle_stringonly_let:** the same as previous, but only words without digits are used;
- **tfidf_title_querythekey_num, tfidf_title_querythekey_let:** amount of information in the most important words (*thekeys*) measured in terms of words found in product title;
- **tfidf_nn_important_in_title_num, tfidf_nn_important_in_title_let:** amount of information in important nouns measured in terms of words found in product title.

The other similar features are calculated with vocabularies from product description and attribute bullets instead of product title, and for other parts of search term like unimportant nouns or adverbs and adjectives, etc.

In file `tfidf_by_st_features.py` TFIDF features were calculated for the docs related to the same unique search term (instead of all docs in the dataset described above). We calculated this features manually and did not use any specific libraries. So, these features might differ from the exact implementation of TFIDF.

### 8.2.5 Word2vec

We built four different vocabularies: for raw/processed text using two different ways of building the vocabularies (adding all parts of the available documents successively or all search terms at the first step, all product titles at the second step and so on). We trained four models on these vocabularies and we did not use any pretrained models. Then we calculated **n_similarity()** between

- stemmed search term and one of four stemmed variables (product title, product description, all attributes, all text);
- not stemmed search term and all text;
- stemmed product title and product description,

thus ending up with 24 word2vec features.

### 8.2.6 WordNet Similarity Features

This is similar to Chenglong's approach in section 8.1.14. We take a pair of words for the most important trigrams in search term and product title, extract the lists of the associated synsets from WordNet, then use WordNet to calculated path similarity, Leacock-Chodorow similarity and Resnik similarity for all combination of synsets in the list. For the model we take the maximum and the mean value of the corresponding similarity measure. Example of the features:

- **beforethekey_thekey_pathsimilarity_mean:** mean similarity between synsets for *beforethekey* in search term and *thekey* in product title; path similarity is used a measure of similarity;
- **thekey_before2thekey_lchsimilarity_max:** maximum similarity between synsets for *thekey* in search term and *before2thekey* in product title; Leacock-Chodorow similarity is used a measure of similarity; and so on.

### 8.2.7 Query Expansion

In this competition we were required to determine the relevance between the search query and the matched product. Since each search term has only a few words, the amount of information contained in the search term might be too low for our task to be performed efficiently. However, we found a way to use some information from product description as a separate input (so we did not need to compare product description with the search term) for our algorithm.

The idea is to group similar queries together and then estimate how common the product description is. We observed that the relevance distribution is significantly skewed to 3, so in each small subset the majority of relevances will be closer to 3 than to 1. We need to assume that the more *common* is the product description for a particular group of search terms, the higher is the corresponding relevance.

For example, let us consider search queries asking for doors. The product descriptions for such queries usually describe different attributes of doors like height, width and material. But if one of the queries for door is matched with another product like refrigerator, the product description would feature very unusual properties for doors like voltage, temperature or volume. It is very likely that such an unusual description would lead to a low relevance.

We considered the rows to be in the same group if the two last words in the top trigram for search term were identical. Then we generated the list of the most common 30 words in the product description for each group and found their frequencies (e.g. $word_1$ is found in $p_1$% of product descriptions, $word_2$ is found in $p_2$% of product descriptions, etc.). In the next step we generated the following features:

- **description_similarity_*nN*:** $\sum_{i=n}^{N} p_i$ if $word_i$ in product description, where $N\epsilon(10, 20, 30)$ and $n$ is normally equal to 1, but could also be 11 or 21;
- **description_similarity_*nN*rel:** the same as the previous but divided by $\sum_{i=n}^{N} p_i$ ;
- **dummies:** if the number of rows in a group is above 8 and above 15. The correlation of the features above with the actual relevance would be too volatile

if the size of the group is small, so we needed to account for the size of the group. Once we included the size of the group as a feature into the model and found it very predictive, but eventually we decided to remove the feature which might be a major cause of overfitting.

### 8.2.8 Dummies: Brands, Materials, Keywords

We generated about 500 such features. Some of them proved to be informative.

# 9 Model Building

## 9.1 Chenglong's Part

### 9.1.1 Split

We implement an advanced splitter `HomedepotSplitter` by taking into consideration the following aspects:

- search_term
  - number of search_term **only** in training set
  - number of search_term **both** in training and testing set
  - number of search_term **only** in testing set
- product_uid
  - number of product_uid **only** in training set
  - number of product_uid **both** in training and testing set
  - number of product_uid **only** in testing set
- number of samples in training and testing set.

Inspired by [6], we compare different splits of product_uid and search_term via Venn-Diagram in Fig. 5. In specific, we compare three splits:

1. the actual train/test split
2. the naive $0.69 : 0.31$ random shuffle split
3. the advanced `HomedepotSplitter`

As shown, the proposed `HomedepotSplitter` yields a similar split as the actual train/test split on both the search_term and product_uid.

Using this splitter, the gap between local CV and public LB is consistent around $0.001 \sim 0.002$ which is within the 2-std range of CV for both single models and ensembled models. This can be verified by Fig. 5, which shows the CV RMSE together with the corresponding public & private LB RMSE for some of Chenglong's submissions.

We use `HomedepotSplitter` to generate splits for all our models (including base models and ensembled models).

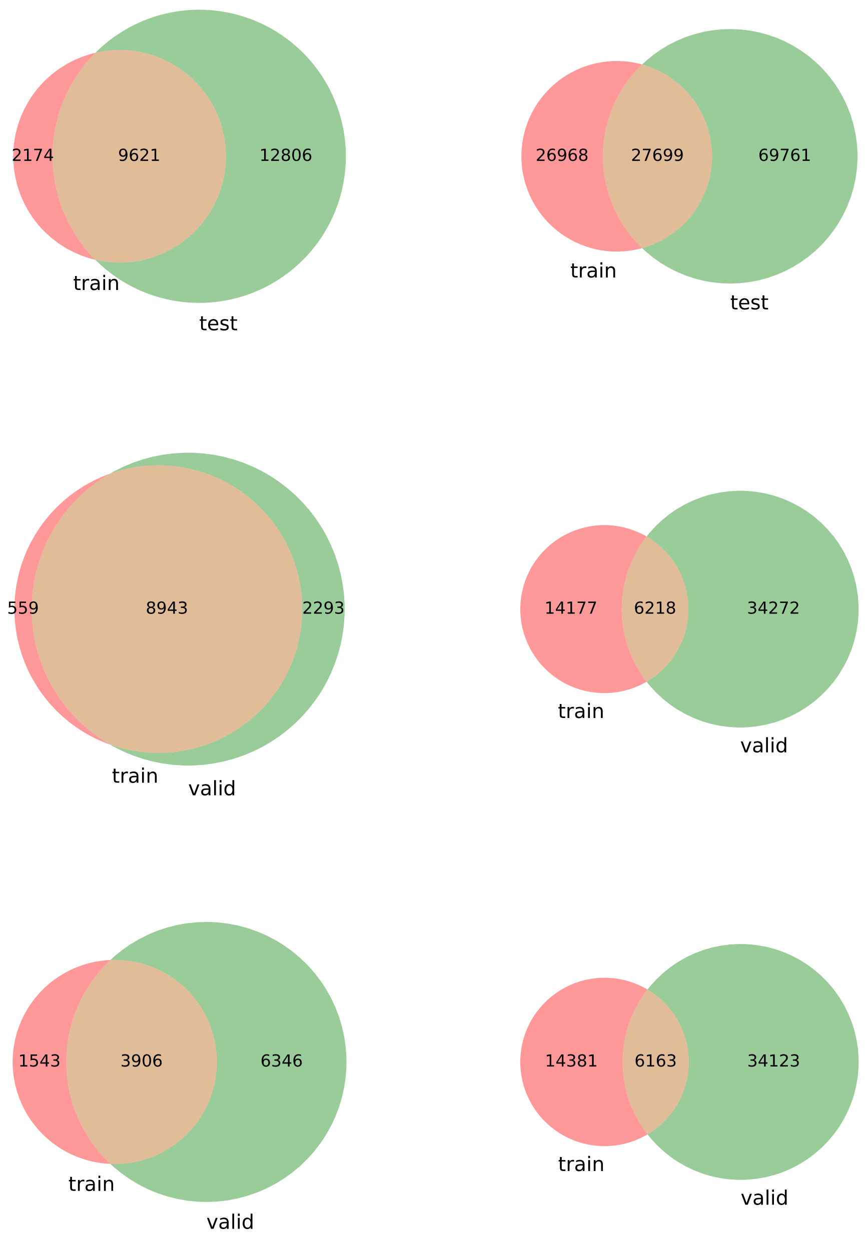

Figure 4: Comparison of different splits on search_term (left column) and product_uid (right column). From top to bottom, shown are actual train/test split, naive 0.69 : 0.31 random shuffle split, and the proposed split.

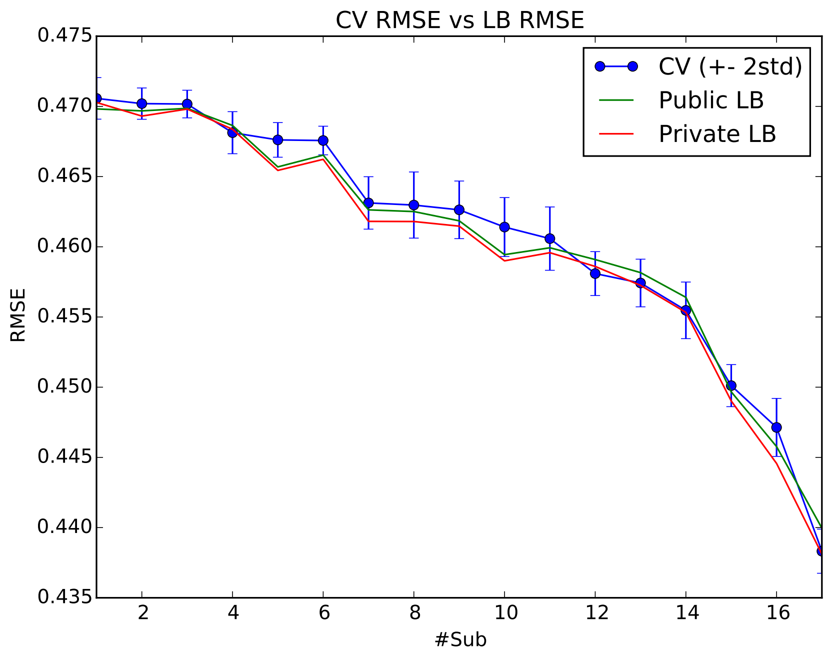

Figure 5: CV RMSE vs LB RMSE for some of Chenglong's submissions.

#### 9.1.2 Learners

The learners we used are listed in 1.

#### 9.1.3 Base Models

We build base models with the above learners on various feature subsets. In most of the case, we use correlation to the relevance target as a judgement to include feature into model building or not. In specific, we only consider those features with a correlation higher than a threshold e.g., 0.05. This strategy might be way too simple. Other alternative choices may include using the feature importance returned by GBDT or RF, greedy forward/backward feature selection etc.

#### 9.1.4 Ensembling

As first, we mainly focused on using (extreme) ensemble selection which is also used in CrowdFlower. While it still improves the performance of single models, it becomes very time consuming to generate an ensemble as the model library becoming larger and larger. On the other side, since the evaluation metric of the HomeDepot competition is RMSE (which can be directly optimized using gradient method), ensemble selection doesn't offer much edge over simple linear blending and stacking. So we turn to stacking eventually. One notable advantage of stacking is that, besides using the predictions from base models as features, we can also include other meta features, e.g., features used to build base models.

Table 1: Learners of Chenglong

| Package | Model | Module |
|---|---|---|
| xgboost | XGBoostRegressor | utils.xgb_utils |
| sklearn | GradientBoostingRegressor | |
| | AdaBoostRegressor | utils.skl_utils |
| | ExtraTreesRegressor | |
| | RandomForestRegressor | |
| | LinearSVR | utils.skl_utils |
| | Ridge | |
| | Lasso | |
| keras | KerasDNNRegressor | utils.keras_utils |
| rgf | RGFRegressor | utils.rgf_utils |

## 9.2 Igor&Kostia's Part

### 9.2.1 Cross Validation

Throughout the competition we used *StratifiedShuffleSplit* with test_size = 0.65. The idea was to mimic the split between train and test (the test accounts for 69% of the total observations). Such a split gave us a consistent gap between our CV and LB: LB RMSE − CV RMSE $\approx 0.002$.

But for our ensembling we wanted that each observation be proportionally present in the training and validation sets, which can not be guaranteed by StratifiedShuffleSplit for any finite number of splits. That is why for ensembling we switched to *StratifiedKFold* with $k = 3$ and used only one part for training and two parts for validation, thus implementing the Chenglong's approach for CrowdFlower [4].

### 9.2.2 Learners and Models

The list of our learners is presented in table 2.

We found it beneficial to run xgboost as a base model for *BaggingRegressor*. This combination improved RMSE by $0.001 \ldots 0.002$ (we used n_estimators = 10). For any other our model *BaggingRegressor* would not improve performance.

We built several models on different feature sets and with different parameters. To reduce the number of features, we combined the following approaches:

- Generated feature importances from our benchmark *GradientBoostingRegressor* model and kept features only above certain importance threshhold.
- Removed highly correlated features.

### 9.2.3 Ensembling

As an output from the modelling steps we had prediction for validation set and test set. We used linear regression on predictions for validation set to find optimal coefficients for our ensemble and then calculated ensemble predictions for the test set.

Table 2: Learners of Igor&Kostia

| Package | Model | File |
|---|---|---|
| xgboost, sklearn | BaggingRegressor(XGBoostRegressor()) | generate_models.py<br>generate_model_wo_google.py |
| sklearn | GradientBoostingRegressor | generate_models.py |
|  | ExtraTreesRegressor |  |
|  | RandomForestRegressor |  |
|  | SVR |  |

#### 9.2.4 Ensembling2

Also we would like to describe another ensemble which we use for the final predictions. This ensemble slightly improved our private LB score from 0.43723 to 0.43704 and has some positive and negative sides. The biggest problem was a skyrocketing gap between CV and LB due to poor cross validation split (*StratifedKFold* with $k = 3$ and two parts used for training and one part used for validation and split was not stable). Nevertheless, we think this ensemble has some interesting ideas and could perform better with the correct split method.

Algorithm description step by step:

**Level 1 stages** (`ensemble_script_random_version.py`):

1. Select list of models for level 1.
2. Randomly split all features into two subset (first and second part).
3. Fit and make predictions for each model and each part of features 4 times: 3 for validation and 1 for test (here we use *StratifedKFold* with $k = 3$).
4. Repeat steps 2 and 3 if computation capacity is available (it requires a lot of time).

**Level 2 stages** (`model_selecting.py`):

5. Use *StratifedKFold* split with $k \geq 5$ for validation predictions of models on level 1. $k-1$ parts are used for fitting level 2 models and 1 part is used for validation.
6. Only some of the generated models are beneficial to the ensemble. Now we need to identify the list of 'good' models. Start with the empty list.
7. Add the best (by its validation RMSE) model from level 1 to 'good model list'.
8. For each level 1 prediction which is not yet in 'good model list', fit level 2 model on the model predictions and predictions from models in 'good model list'. Add that model to 'good model list' which provides the largest reduction to the ensemble RMSE (We use linear regression/ridge regression as level 2 model).
9. Repeat 8 while the ensemble RMSE is improving.

Advantages of this algorithm are:

- random feature's subsets are good source of variation which improves the ensemble performance;

Table 3: Weights for the Two Final Submissions

| | Weight Chenglong for $part_1$ and $part_2$ | Weight Chenglong for $part_3$ | Public RMSE | Private RMSE |
|---|---|---|---|---|
| Submission 1 | 0.75 | 0.8 | 0.43443 | 0.43271 |
| Submission 2 | 0.6 | 0.3 | 0.43433 | 0.43271 |

Note: The weights for Chenglong and Igor&Kostia add up to 1.

- greedy selection of models for level 2 performs better on LB than the straightforward ensembling of all available level 1 models;
- the solution can be well automated without any 'hand tuning' or other human intervention (we still need to define the model list for the first step).

Disadvantages:

- in our case inferior split lead to overfitting on level 2.

### 9.3 Final Blend

So, we had two ensembles prepared using different methodologies. We observed that our ensembles behave differently in different parts of the datasets ($part_1$: id $\leq$ 163700, $part_2$: 163700 $<$ id $\leq$ 221473, $part_3$: id $>$ 221473 on Fig. 7 and other figures in Appendix). Since we observed regular patterns in the data as well (see Figs. 9-12), we thought that one of the ensembles might be especially prone to overfitting in some parts. So, while blending our ensembles for final submissions, we made different bets assuming that in some parts one of the models would behave much worse in private than in public.

Our two final submission were produced with the weights from Table 3. They scored the same 0.43271 on the private leaderboard.

It should be noted that the organizers included "poisoned" test data (representing 33% of the test set) to discourage hand labeling. In fact, the $part_3$ was totally formed of poisoned data which was not scored neither in public nor in private.

## 10 Code Description

### 10.1 Chenglong's Part

All the code are in `./Code/Chenglong`.

#### 10.1.1 Config

All the pathes, parameters, and other configurations are set via `config.py`.

#### 10.1.2 Data Processing

- `data_preparer.py`: generate raw dataframe data

- `data_processor.py`: process data
  - a bunck of processing
  - automated spelling correction
  - query expansion
  - extract product name for search_term and product_title
- `google_spelling_checker_dict.py`: contains google spelling correction dictionary from this Kaggle forum post
- `spelling_checker.py`: home-made spelling checker inspired by Peter Norvig's post [3] (though not used in the final submission)

### 10.1.3 Feature Generation

- `feature_base.py`: base class for feature generation, including `BaseEstimator`, `StandaloneWrapper` and `PairwiseWrapper`
- `feature_combiner.py`: combine a bunch of single feature (specified via a feature conf file) into a feature matrix
- `feature_transformer.py`: various feature transformations
- `feature_[basic|distance|doc2vec|...].py`: various feature generators; see 8.1 for details
- `embedding_trainer.py`: script for training word2vec and doc2vec models with the provided data
- `get_feature_conf_*.py`: generate feature conf file as input for `feature_combiner.py`
- `convert_pkl_lsa_to_csv_lsa.py`: convert `.pkl` format LSA features to `.csv` format for using `Rtsne` package in R
- `convert_csv_tsne_to_pkl_tsne.py`: convert `.csv` format TSNE features to `.pkl` format

### 10.1.4 Task

- `task.py`: definitions for
  - learner & ensemble learner
  - feature & stacking feature
  - task & stacking task
  - task optimizer

  It is also the wrapper for running various tasks with various features and various learners.
- `model_param_space.py`: contains parameter searching space for various learners
- `run_[test_ridge|test_xgb|stacking_ridge].py`: some scripts for running the following procedure in one shot
  - generate feature conf

    - generate feature matrix
    - run task

  This is for accelerating the circle of change-validate-check.

- `extreme_ensemble_selection.py`: generate submission via extreme ensemble selection

### 10.1.5 Other

- `run_data.py`: generate data and features in one shot
- `splitter.py`: generate splitter for all the models including 1st, 2nd and 3rd level models
- `turing_test_converter.py`: convert `.csv` format dataframe features (from Igor&Kostia) to `.pkl` format features
- `./utils`: utils for various modules
    - `dist_utils.py`: utils for distance computation
    - `keras_utils.py`: utils for Keras models
    - ...

## 10.2 Igor&Kostia's Part

All the code are in `./Code/Igor&Kostia`.

### 10.2.1 Config

All the paths are set via `config_IgorKostia.py`.

### 10.2.2 Data Processing

Since the same text processing steps should be performed in order to generate different features, we decided to do all text processing before any feature generation. This approach helped us to calculate the result a few days faster if compared with step-by-step feature generation from the input text files.

- `homedepot_functions.py`: contains many functions used for text processing and in some other steps.
- `google_dict.py` contains google spelling correction dictionary from [1].
- `text_processing.py` and `text_processing_wo_google.py` implement the same text processing steps, except the latter file does not uses replacement from the Google dictionary from the forum.

### 10.2.3 Feature Generation

- `feature_extraction1.py` and `feature_extraction1_wo_google.py`: the same logic, but input produced with or without Google dictionary. These files generate basic features, some distance features, brand and material features, some TFIDF features, WordNet similarity features, query expansion features and dummies (see section 8.2)
- `grams_and_terms_features.py`: generates a part of the distance features (see section 8.2.2).
- `dld_features.py`: generates the final part of the distance features (see section 8.2.2), namely involving the Damerau-Levenshtein distances.
- `word2vec.py` and `word2vec_without_google_dict.py`: generate word2vec features (see section 8.2.5).

### 10.2.4 Modelling

- `generate_feature_importances.py`: supplies all generated feature into our benchmark *GradientBoostingRegressor* model and gets the list of feature importances.
- `generate_models.py` and `generate_model_wo_google.py`: create appropriate feature lists, fit a number of models on those features, generate validation results and prediction results for test dataset.
- `generate_ensemble_output_from_models.py`: reads the output from the previous step and generates ensemble submission.

### 10.2.5 Modelling2

This is the necessary code to reproduce our results, although this part is not crucial for our performance because of inferior cross-validation split (*StratifiedKFold* with $k = 3$, two part used for training and one for validation). However, we tried a few advanced techniques for model selection here, which might be promising.

- `ensemble_script_random_version.py`: selects random feature subsets and generates validation and test predictions using a different split.
- `ensemble_script_imitation_version.py`: reproduces the output from previous step used in our submission.
- `model_selecting.py`: selects models and generates ensemble submission.

# 11 Simple Features and Methods

As suggested by the Kaggle documentation template, we generated a model with only 10 features and got private RMSE of 0.44949 ( 31st place on the leaderboard). Please see section 1.2 to compare this with our best submissions.

Feature importances for the simplified model are shown on Fig. 6

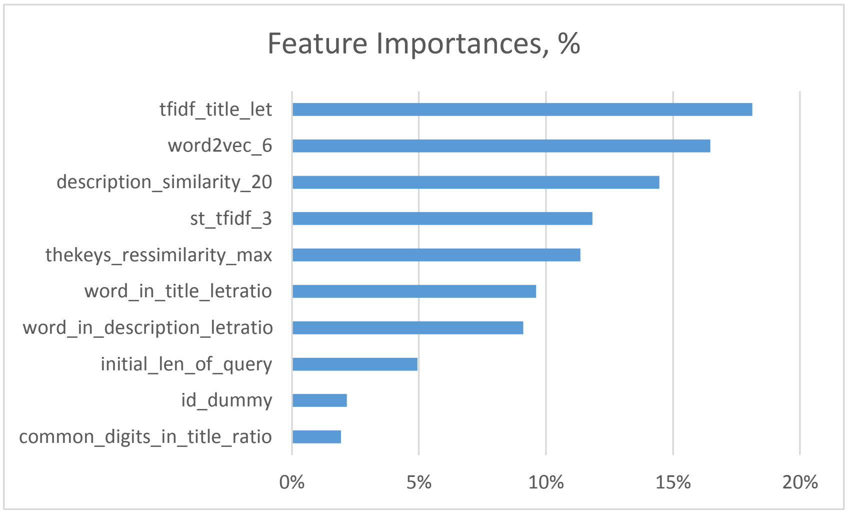

Figure 6: Feature importances for simplified model with 10 features (Igor&Kostia). Note: **word2vec_6** was calculated as *n_similarity()* between stemmed search term and stemmed product title. **description_similarity_20** is the feature generated from query expansion to 20 most common words (section 8.2.7). **st_tfidf_3** is just the number of rows in the dataset with a particular search term.

## 12 Additional Comments and Observations

Some interesting insights we got during the competition:

- Spelling correction/synonyms replacement/text cleaning are very useful for searching query relevance prediction as also revealed in CrowdFlower.
- In this competition, it is quite hard to improve the score via ensemble and stacking. To get the most out of ensemble and stacking, one should really focus on introducing diversity. To that goal, try various text processing, various feature subsets, and various learners. Team merging also contribute a lot of diversity!
- That said, know how to separate the effect of improved algorithm from the effect of increased diversity. Careful experiments are necessary. If some clear and effective solution does not work as well as your old models, do not discard it until you compare both approaches in the same conditions.
- Keep a clean and scalable codebase. Keep track/log of the change you have made, especially how you create those massive crazy ensemble submission.
- Set up an appropriate and reliable CV framework. This allows to try various local optimizations on the features & models and getting feedback without the need to submit them. This is important for accelerating the change-validate-check cycle.
- After team merging, we have spent some effort on figure out the best way to ensemble our results. Due to the fact that we do not use the same CV split, we cannot run something like 2nd level stacking. In the end, we go with the simple

weight average method. We might have better results if we have merged early on. But that might also reduce the diversity of our models, which might be harmful for ensemble as well. But who knows.

- We were thinking that features measuring how close is the { length, width, height, depth, volume, weight, color} of the product in search_term and product_title would be predictive. However, we did not manage to spare out some time to explore these things.

## 13 Acknowledgement

We would like to thank the DMLC team for developing the great machine learning package xgboost, François Chollet for developing package keras, James Bergstra for developing package hyperopt, Radim Řehůřek for developing package gensim. We would also like to thank the Kaggle team and Homedepot for organizing this competition.

Igor would like to thank his wife Anastasiia for her support and inspiration during this competition.

## References

[1] `https://www.kaggle.com/steubk/home-depot-product-search-relevance/fixing-typos`.

[2] `http://blog.kaggle.com/2015/07/22/crowdflower-winners-interview-3rd-place-team-quartet`

[3] `http://norvig.com/spell-correct.html`.

[4] `https://github.com/ChenglongChen/Kaggle_CrowdFlower`.

[5] `https://www.kaggle.com/c/dato-native/forums/t/16626/beat-the-benchmark-0-90388-with-simple-model`.

[6] `https://www.kaggle.com/c/home-depot-product-search-relevance/forums/t/19463/what-s-the-lb-shakeup-potential/116689#post116689`.

[7] `https://github.com/manasRK/glove-gensim`.

# Appendix A Model Execution Time

- **What software did you use for training and prediction?**
  Please see section Appendix B.
- **What hardware (CPUS spec, number of CPU cores, memory)?**
  Chenglong: 8-Core i7-4790 3.60GHz, 16GB RAM running Windows; 16-Core E5520 2.27GHz, 36GB RAM running Linux;

  Igor&Kostia: 4-Core i5 2.8 GHz, 16GB RAM running Windows.
- **How long does it take to train your model?**
  Chenglong: Feature generation part takes 1 $\sim$ 2 days to finish, depending on the computational power. This is for Chenglong's feature alone. Prediction and training for the best single model from Chenglong takes a few minutes to run. To reproduce Chenglong's best ensemble submission which relies on a few hundreds of 1st level models, it will take a few days. The `AdaBoostRegressor`, `GradientBoostingRegressor`, and `RGFRegressor` are the most time consuming models to train. But including a few of them is useful for the ensemble.

  Igor&Kostia: Text processing and feature generation takes a few days. Prediction and training for a single model can take from $\sim$ 5 minutes (xgboost) to $\sim$ 5 hours (SVR).
- **How long does it take to train the simplified model (referenced in section 11)?**
  We did not try to generate the used features very fast. Training and prediction takes a few seconds.

# Appendix B Dependencies

## B.1 Chenglong's Part

### B.1.1 Python

We used Python 3.5.1 and modules comes with Anaconda 2.4.1 (64-bit). In addition, we also used the following libraries and modules:

- gensim 0.12.4
- hyperopt 0.0.3.dev
- keras 0.3.2
- matplotlib-venn 0.11.3
- python-Levenshtein 0.12.0
- regex 2.4.85
- xgboost 0.4

### B.1.2 R

We used the following packages installed via `install.packages()`:

- data.table
- Rtsne

### B.1.3 Other

We used the following thirdparty packages:

- rgf 1.2

## B.2 Igor&Kostia's Part

### B.2.1 Python

We used Python 2.7.11 on Windows platform and modules comes with Anaconda 2.4.0 (64-bit), including:

- scikit-learn 0.17.1
- numpy 1.10.1
- pandas 0.17.0
- re 2.2.1
- matplotlib 1.4.3
- scipy 0.16.0

In addition, we also used the following libraries and modules:

- NLTK 3.1 (use *nltk.download()* command)
- gensim 0.12.2
- xgboost 0.4

Some descriptive analysis was also done in Excel 2007 and Excel 2010.

# Appendix C How To Generate the Solution (aka README file)

## C.1 Chenglong's Part

Before proceeding, one should place all the data from the competition website into folder `./Data`. Note that in the following, all the commands and scripts are executed and run in directory `./Code/Chenglong`.

### C.1.1 Step 1. Install Dependencies

Make sure you have installed all the dependencies listed in B.1.

### C.1.2 Step 2. Prepare External Data

**1. Pre-trained Word2Vec Model**
We used pre-trained Word2Vec models listed in this Github repo. In specific:

- Google News
- Wikipedia+Gigaword 5

We used glove-gensim [7] to convert GloVe vectors into Word2Vec format for easy usage with Gensim. After that, put all the models in the corresponding directory (see `config.py` for detail).
**2. Other**
We also used the following external data:

- Color data from this Kaggle forum post, i.e., `./Data/dict/color_data.py`.
- Google spelling correction dictionary from this Kaggle forum post, i.e., `google_spelling_checker_dict.py`.
- Home-made word replacement dictionary, i.e., `./Data/dict/word_replacer.csv`.
- NLTK corpora and taggers data downloaded using `nltk.download()`, specifically: `stopwords.zip`, `wordnet.zip` and `maxent_treebank_pos_tagger.zip`.

### C.1.3 Step 3. Generate Features

To generate data and features, one should run `python run_data.py`. While we have tried our best to make things as parallelism and efficient as possible, this part might still take 1 ∼ 2 days to finish, depending on the computational power. So be patient :)

Note that various text processing are useful for introducing diversity into ensemble. As a matter of fact, one feature set (i.e., `basic20160313`) from our final solution is generated before the Fixing Typos post, i.e., not using the Google spelling correction dictionary. Such version of features can be generated by turning off the `GOOGLE_CORRECTING_QUERY` flag in `config.py`.

After team merging with Igor&Kostia, we have rebuilt everything from scratch, and most of our models used different subsets of Igor&Kostia's features. For this reason, you should also need to follow Sec. C.2.3 to generate their features. Since Igor&Kostia's features are in `.csv` dataframe format, we provide a converter `turing_test_converter.py` to convert them to the format we use, i.e., `.pkl`.

### C.1.4 Step 4. Generate Feature Matrix

In step 3, we have generated a few thousands of features. However, only part of them will be used to build our model. For example, we don't need those features that have very little predictive power (e.g., have very small correlation with the target relevance.) Thus we need to do some feature selection. In our solution, feature selection is enabled via the following two successive steps.
**1. Regex Style Manual Feature Selection**
This approach is implemented as `get_feature_conf_*.py`. The general idea is to include or exclude specific features via *regex* operations of the feature names. For example,

- one can specify the features that he want to **include** via the `MANDATORY_FEATS` variable, despite of its correlation with the target
- one can also specify the features that he want to **exclude** via the `COMMENT_OUT_FEATS` variable, despite of its correlation with the target (`MANDATORY_FEATS` has higher priority than `COMMENT_OUT_FEATS`.)

The output of this is a feature conf file. For example, after running the following command:

```
python get_feature_conf_nonlinear.py -d 10 -o feature_conf_nonlinear_201605010058.py
```

we will get a new feature conf `./conf/feature_conf_nonlinear_201605010058.py` which contains a feature dictionary specifying the features to be included in the following step.

One can play around with `MANDATORY_FEATS` and `COMMENT_OUT_FEATS` to generate different feature subset. We have included in `./conf` a few other feature confs from our final submission. Among them, `feature_conf_nonlinear_201604210409.py` is used to build the best single model.

**2. Correlation based Feature Selection**

With the above generated feature conf, one can combine all the features into a feature matrix via the following command:

```
python feature_combiner.py -l 1 -c feature_conf_nonlinear_201604210409 -n
basic_nonlinear_201604210409 -t 0.05
```

The `-t 0.05` above is used to enable the correlation base feature selection. In this case, it means: drop any feature that has a correlation coef lower than `0.05` with the target relevance.

TODO(Chenglong): Explore other feature selection strategies, e.g., greedy forward feature selection (FFS) and greedy backward feature selection (BFS).

### C.1.5 Step 5. Generate Submission

**1. Various Tasks**

In our solution, a `task` is an object composite of a specific `feature` (e.g., `basic_nonlinear_201604210409`) and a specific `learner` (e.g., `XGBoostRegressor` from xgboost. The definitions for `task`, `feature` and `learner` are in `task.py`.

Take the following command for example.

```
python task.py -m single -f basic_nonlinear_201604210409 -l reg_xgb_tree -e
100
```

- It runs a `task` with `feature` `basic_nonlinear_201604210409` and `learner` `reg_xgb_tree`.
- The `task` is optimized with hyperopt for 100 evals for searching the best parameters for learner `reg_xgb_tree`.
- The `task` performs both CV and final refit. CV in this case has two purposes: 1) guide hyperopt to find the best parameters, and 2) generate predictions for each CV fold for further (2nd and 3rd level) stacking.
- For all the available learners and the corresponding parameter searching space, please see `model_param_space.py`.

During the competition, we have run various tasks (i.e., various features and various learners) to generate a diverse 1st level model library. Please see `./Log/level1_models` for all the tasks we have included in our final submission.

**2. Best Single Model**

After generating the `feature basic_nonlinear_201604210409` (see step 4 how to generate this), run the following command to generate the best single model:

```
python task.py -m single -f basic_nonlinear_201604210409 -l reg_xgb_tree_best_single_model
-e 1
```

This should generate a submission with local CV RMSE around 0.438 $\sim$ 0.439 (and private LB RMSE around 0.438).

**3. Best Ensemble Model**

After building **some diverse** 1st level models, run the following command to generate the best ensemble model:

```
python run_stacking_ridge.py -l 2 -d 0 -t 10 -c 1 -L reg_ensemble -o
```

This should generate a submission with local CV RMSE around 0.436.

## C.2 Igor&Kostia's Part

Before proceeding, one should specify correct paths in file `config_IgorKostia.py` and place all the data from the competition website into folder specified by variable `DATA_DIR`. To reproduce our Ensemble$_B$ from section C.2.5 one should place the used feature sets into folder specified by variable `FEATURESETS_DIR`. Note that in the following, all the commands and scripts are executed and run in directory `./Code/Igor&Kostia`.

### C.2.1 Step 1. Install Dependencies

Make sure you have installed all the dependencies listed in B.2.

### C.2.2 Step 2. Text Preprocessing

We do all text preprocessing before any feature generation and save the results to files. It helped us save a few computing days since the same preprocessing steps are necessary to generate different features.

- Run `text_processing.py`;
- Run `text_processing_wo_google.py`

The necessary replacement data is loaded automatically from files `homedepot_functions.py` and `google_dict.py`.

### C.2.3 Step3. Feature Generation

We need to run consequently the following files:

- `feature_extraction1.py`.
- `grams_and_terms_features.py`.

- `dld_features.py`.
- `word2vec.py`.

To generate features without using the Google dictionary, we also need to run:

- `feature_extraction1_wo_google.py`.
- `word2vec_without_google_dict.py`.

As a result, we will have a few csv files with the necessary features for model building. Please see more information in section 10.2.

#### C.2.4 Step 4. Generate Benchmark Model with Feature Importances

- Run `generate_feature_importances.py`.

#### C.2.5 Step 5. Generate Submission File

One part of the ensemble (Ensemble$_A$) is generated from the following code:

- `generate_models.py`.
- `generate_model_wo_google.py`.
- `generate_ensemble_output_from_models.py`.

To get the other part (Ensemble$_B$), we need to run these files:

- `ensemble_script_imitation_version.py` (It just reproduces the selection of random features generated from `ensemble_script_random_version.py`. You do not need to run `ensemble_script_random_version.py` again).
- `model_selecting.py`.

These two parts can be generated in parallel. Our final submission from Igor&Kostia was then prodused in Excel as:

$$\text{Output} = 0.75 \times \text{Ensemble}_A + 0.25 \times \text{Ensemble}_B$$

## C.3 Blending Two Ensembles into the Final Submissions

This step was done in Excel using the weights described in section 9.3.

# Appendix D Some charts

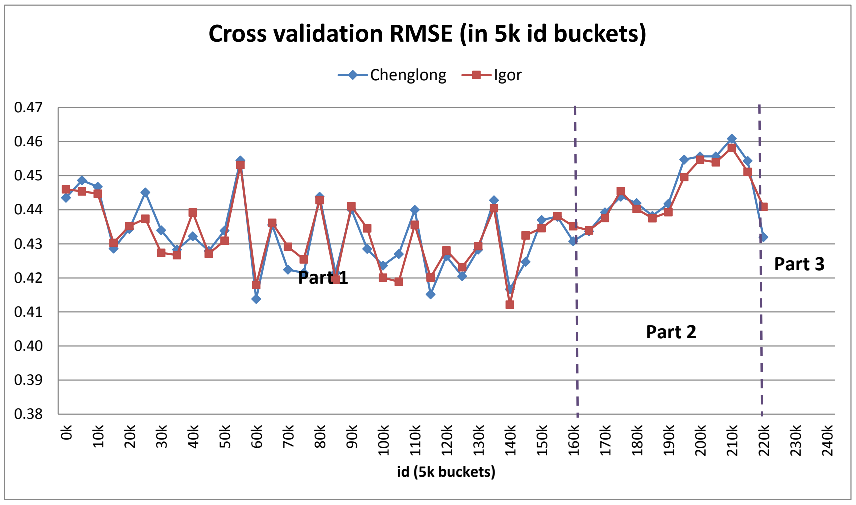

Figure 7: Cross validation RMSE of ensembles.

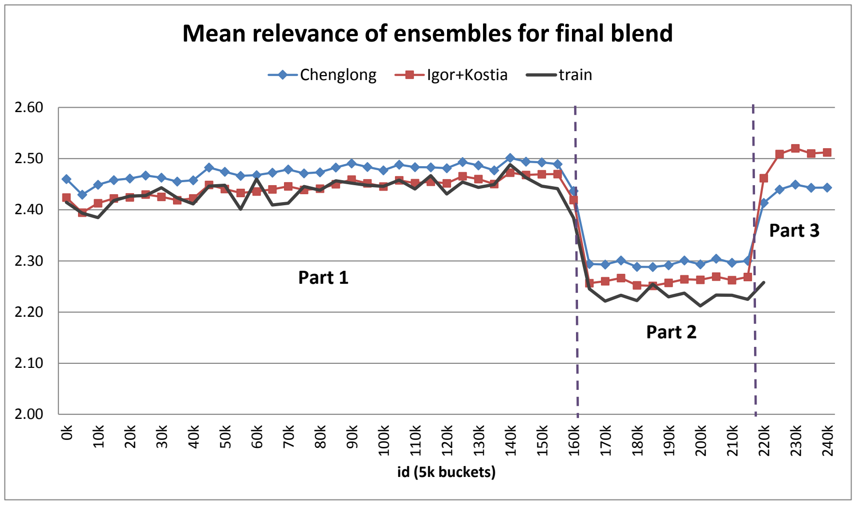

Figure 8: Mean relevance of ensembles for final blend.

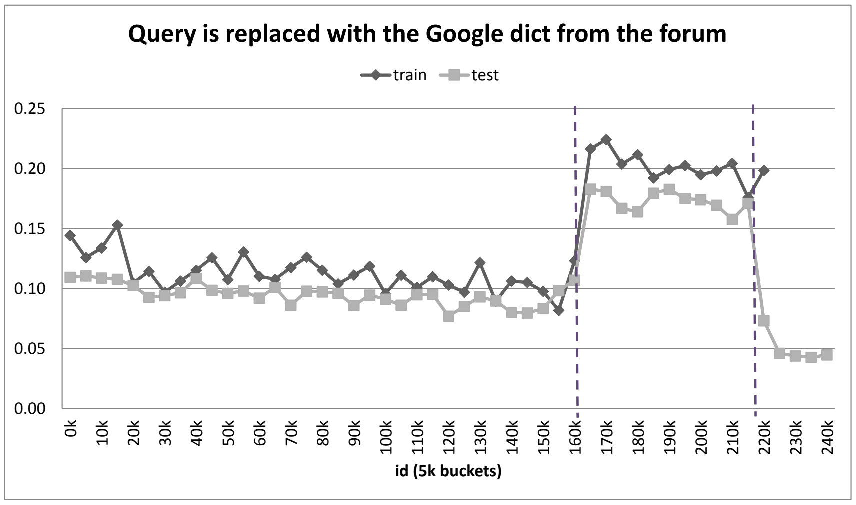

Figure 9: Query is replaced with the Google dictionary from the forum. (Note the difference between train and test).

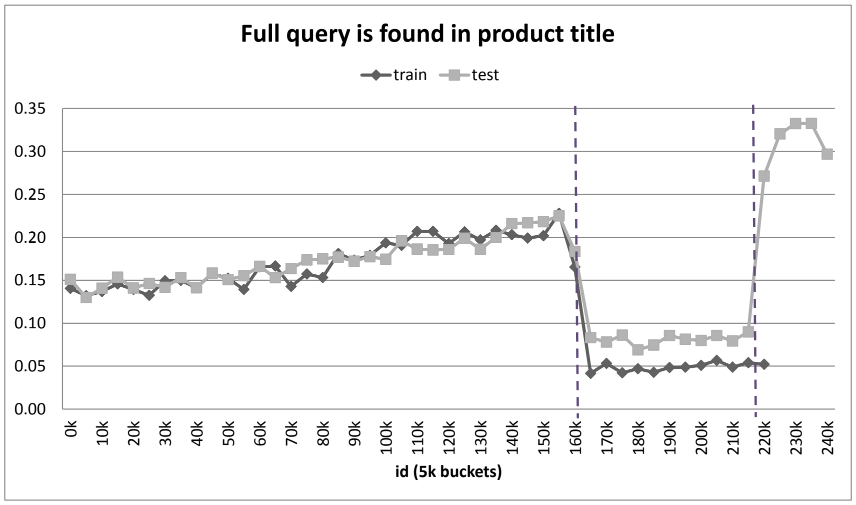

Figure 10: Full query is found in product title.

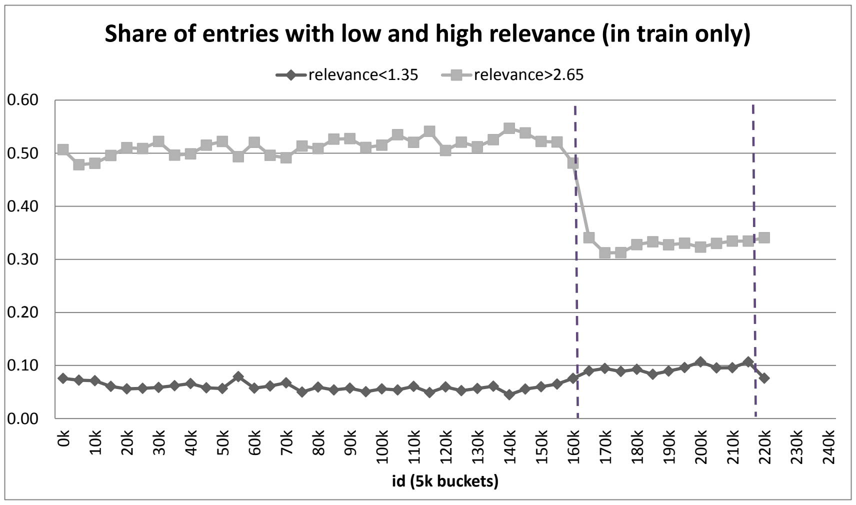

Figure 11: Share of entries with low and high relevance (in train only).

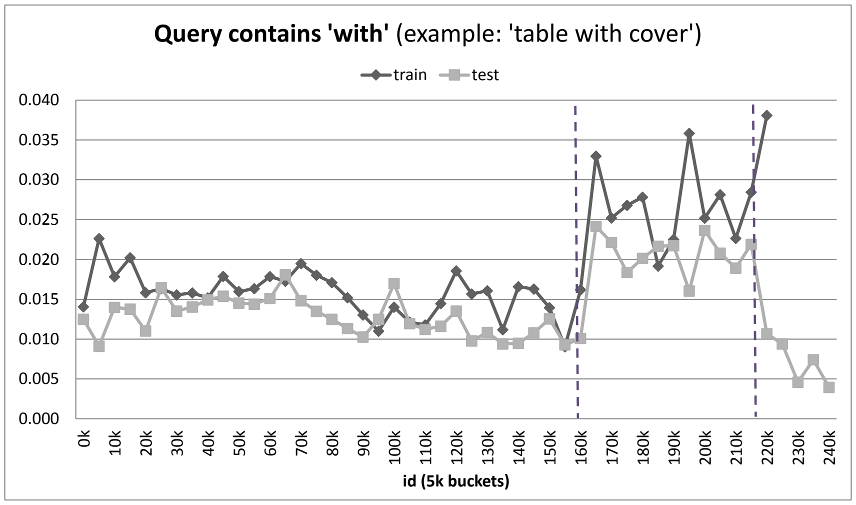

Figure 12: Query contains *'with'* (example: *'table with cover'*).