CrowdFlower Search Results Relevance second place solution

Mikhail Trofimov, Stanislav Semenov, Dmitry Altukhov

July 16, 2015

Abstract

The goal of this competition was to create a predictive model which estimates the relevance of search results, given search query and respective response. The key ideas of our solution were: query expansion, model stacking and custom class separators.

Contents

1 Summary

The dataset provided by CrowdFlower team contains about 10k labeled samples and about 20k samples with unknown labels. Each sample is a (search query, product title, product description) triple. Label is an integer score from 1 (irrelevant) to 4 (relevant). Solutions were evaluated by quadratic weighted kappa metric[1]. Therefore two main difficulties of this competition were small amount of training data and non-standard evaluation metric.

Our solution consists of following steps:

- Text preprocessing
- Query expansion
- Feature extraction
- Model training and ensembling
- Optimizing class separators

2 Text preprocessing

First of all, we made some corrections in both query and title:

hardisk	hard drive
extenal	external
soda stream	sodastream
fragance	fragrance
16 gb	16gb
32 gb	32gb
500 gb	500gb
2 tb	2tb
shoppe	shop
refrigirator	refrigerator
assassinss	assassins
harleydavidson	harley davidson
harley-davidson	harley davidson

Table 1: List of corrections

After that all queries and titles were stemmed using nltk package.

In order to use word2vec (pretrained on Google News dataset[2]) in 3rd in 4th models, query, title and description were further lemmatized via nltk.stem.wordnet.WordNetLemmatizer().

3 Query expansion

Since search queries are very short, we decided to use information from relevant titles to make them longer. In order to do so we performed following operations:

- For each query concatenate that query and all respective titles with label 4

[1] https://www.kaggle.com/c/crowdflower-search-relevance/details/evaluation
[2] https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing

- Extract top n (we tried 10 and 15) most frequent words from it in descending order

We call those top n words 'expanded query'.

4 Feature extraction

We extracted following features:

1. Group 1:
   - number of words in query
   - number of words in title
   - whether description is present or not - boolean flag
   - number of words from query that are present in title
   - compression distance (lzma archiver) between query and title[3]
   - 1 - edit distance between query and title (using function SequenceMatcher from difflib package)
   - 1 - mean(maximum edit distance between word from query and every word from title)
   - whether last word from query is present in title - boolean flag
   - ratio of words from query that are present in title
2. Group 2 (built using expanded query based on top 10 words):
   - number of words from expanded query that are present in title
   - compression distance between expanded query and title
   - 1 - edit distance between expanded query and title
   - 1 - mean(maximum edit distance between word from expanded query and every word from title)
   - ratio of words from expanded query that are present in title
   - (26 - minimum index of expanded query word) if number of words from expanded query that are present in title > 0, else - 0. Since more frequent words are at the beginning of expanded query, this feature, basically, gives more weight to more frequent words
3. Group 3. Same as Group 2 but on top 15 words
4. Group 4. We calculate letter frequencies in query and title, and use the following ratios as features:
   (FreqQuery + A) / (FreqTitle + B) / (query length), where A and B is constants that maximize local score (obtained from cross-validation)
5. Group 5:
   - word2vec similarities between title and query
   - vector between mean of words in title and words in query (using word2vec pretrained embedding)
   - cosine similarities between query and title, query and description
   - tSNE(2dim) from SVD(200dim) from BoW+TfIdf on titles

[3] https://en.wikipedia.org/wiki/Normalized_compression_distance

5 Model training and ensembling

We average predictions of four different models.

5.1 Support Vector Regressor 1

Following feature groups are used:

- Group 1
- Group 2
- Group 4
- Group 5
- Concatenate expanded query (on top 10 words) and title, use TFIDF transformation on 'char' n-grams from 1 to 5, select first 300 components of SVD decomposition

All features are scaled to [0,1] range. Model is trained with 1/(1+variance) weights.

SVR parameters (estimated from grid search on 3-fold cross-validation):

- C = 8
- gamma = 0.15
- kernel = 'rbf'
- cache_size = 2048

5.2 Support Vector Regressor 2

Following feature groups are used:

- Group 1
- Group 3
- Group 4
- Group 5
- Concatenate expanded query (on top 15 words) and title, use TFIDF transformation on 'char' n-grams from 1 to 5, select first 400 components of SVD decomposition

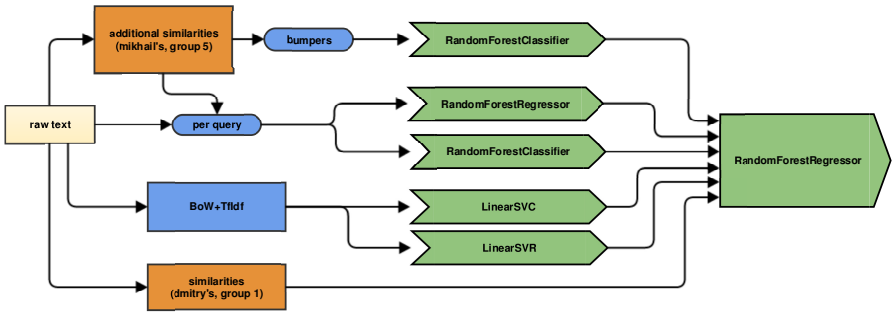
All features are scaled to [0,1] range. Model is trained with 1/(1+variance) weights.

SVR parameters (estimated from grid search on 3-fold cross-validation):

- C = 4
- gamma = 0.2
- kernel = 'rbf'
- cache_size = 2048

5.3 Stacked model 1

Model is build on original (not expanded) queries. Model architecture:



Apart from standard models trained on the whole dataset, we built small per-query models (261 - for every query). Those models are built only on BoW extracted from train+test union. Concept of 'bumpers' is also interesting. Since competition metric punishes severe misclassification, we create binary classifiers to distinguish {1} vs {2,3,4}, {1,2} vs {3,4} and {1,2,3} vs {4}. Probabilities from those classifiers were also used as metafeatures for final model.

In general, we build metafeatures from different 50/50 splits of training data.

5.4 Stacked model 2

Same as Stacked model 1 but built on expanded queries based on top 10 words.

6 Optimizing class separators

The most important part is transition from raw outputs to class labels. Straightforward rounding gives very poor results. Therefore optimal class borders were found via following routine for each cross-validation iteration:

- scale model outputs to [0,1] range
- perform exhaustive search over all reasonable borders with 0.01 step size

Final borders are [0.33, 0.6, 0.77]:

- if scaled output $<$ 0.33 - it's class 1
- if scaled output in [0.33,0.6) - it's class 2
- if scaled output in [0.6,0.77) - it's class 3
- if scaled output $>=$ 0.77 - it's class 4

7 Building final solution

1. Put train.csv, test.csv and all scripts in the working directory
2. Run scripts in the following order:
   - preprocessing_dmitry.py

- preprocessing_mikhail.py
- preprocessing_stanislav.py
- fit_models_dmitry.py
- fit_model1_mikhail.py
- fit_model2_mikhail.py
- blend.py

3. Solution is the final solution.csv file

It takes approximately 1-1.5 hours to build the submission file from scratch.

8 Dependencies

This code was run on machine with installed Ubuntu 14.04.2 and following software:

- python 2.7.6
- numpy 1.10
- pandas 0.16.0
- scikit-learn 0.16.1
- scipy 0.15.1
- nltk 3.0.3
- BeautifulSoup 4.3.2
- tsne 0.1.1 [4]
- gensim 0.11.1-1
- backports.lzma 0.0.3

9 Hardware

This code was developed on a machine with 32 cores and 60 gb ram (amazon cc2.8xlarge instance), however it is possible to build a solution even with the average laptop.

10 Single model score and main insights

All single models score in [0.703, 0.707] range depending on random seed and appropriate class separators. Models {1,2} and {3,4} produce very similar results, therefore it is enough to ensemble Model 1 and Model 4 to get a decent score.

Main insights:

- Query expansion. It helps a lot, because original query is very short. By expanding query we drastically increase the quality of query-title-similarity based features;
- n-grams on letters. It works better that word n-grams, because dataset is pretty small and we don't care much about word order;
- Custom class separators. It allows us to take advantage of evaluation metric.

[4] https://github.com/danielfrg/tsne