Solution for the Search Results Relevance Challenge

Chenglong Chen

July 12, 2015

Abstract

In the Search Results Relevance Challenge, we were asked to build a model to predict the relevance score of search results, given the searching queries, resulting product titles and product descriptions. This document describes our team's solution, which relies heavily on feature engineering and model ensembling.

Personal details

- Name: Chenglong Chen
- Location: Guangzhou, Guangdong, China
- Email: `c.chenglong@gmail.com`
- Competition: Search Results Relevance[1]

[1] `https://www.kaggle.com/c/crowdflower-search-relevance`

Contents

1 Summary

Our solution consisted of two parts: feature engineering and model ensembling. We had developed mainly three types of feature:

- counting features
- distance features
- TF-IDF features

Before generating features, we have found that it's helpful to process the text of the data with spelling correction, synonym replacement, and stemming. Model ensembling consisted of two main steps, Firstly, we trained model library using different models, different parameter settings, and different subsets of the features. Secondly, we generated ensemble submission from the model library predictions using bagged ensemble selection. Performance was estimated using cross validation within the training set. No external data sources were used in our winning submission. The flowchart of our method is shown in Figure 1.

The best single model we have obtained during the competition was an XGBoost model with linear booster of Public LB score **0.69322** and Private LB score **0.70768**. Our final winning submission was a median ensemble of 35 best Public LB submissions. This submission scored **0.70807** on Public LB (our second best Public LB score) and **0.72189** on Private LB. [2]

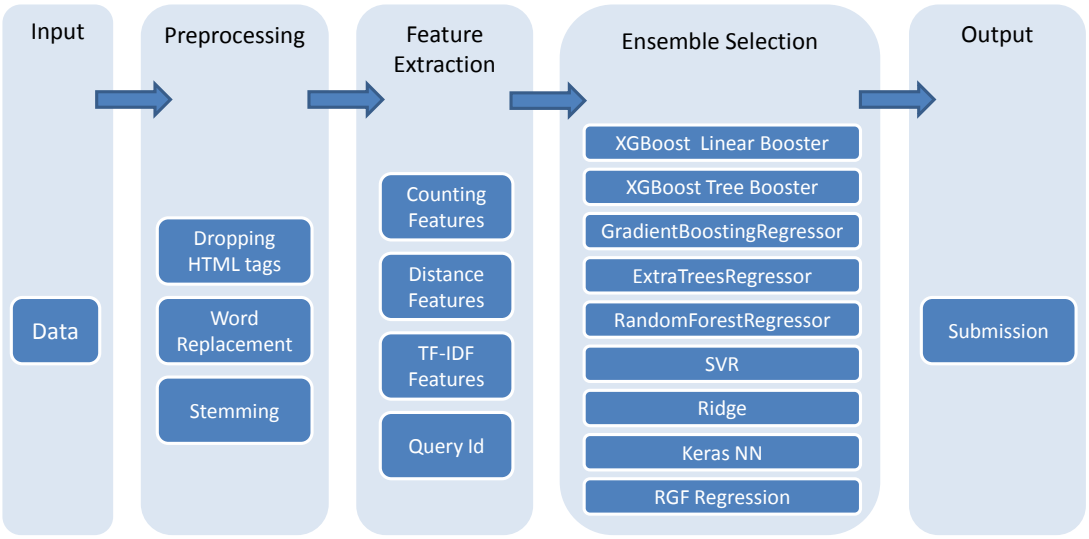

Figure 1: The flowchart of our method.

[2]The best Public LB score was **0.70849** with corresponding Private LB score **0.72134**. It's a mean ensemble version of those 35 LB submissions.

2 Preprocessing

A few steps were performed to cleaning up the text.

2.1 Dropping HTML tags

There are some noisy HTML tags in `product_description` field, we used the library `bs4` to clean them up. It didn't bring much gain, but we have kept it anyway.

2.2 Word Replacement

We have created features similar as "how many words of query are in product title", so it's important to perform some word replacements, e.g., spelling correction and synonym replacement, to align those words with the same or similar meaning. By exploring the provided data, it seems CrowdFlower has already applied some word replacements in the searching results.

2.2.1 Spelling Correction

The misspellings we have identified are listed in Table 1. Note that this is by no means an exhaustive list of all the misspellings in the provided data. It is just the misspellings we have found while exploring the training data during the competition. This also applies to Table 2 and Table 3.

Table 1: Spelling Correction

misspellings	correction
refrigirator	refrigerator
rechargable batteries	rechargeable batteries
adidas fragance	adidas fragrance
assassinss creed	assassins creed
rachel ray cookware	rachael ray cookware
donut shoppe k cups	donut shop k cups
extenal hardisk 500 gb	external hardisk 500 gb

2.2.2 Synonym Replacement

Table 2 lists out the synonyms we have found within the training data.

2.2.3 Other Replacements

Apart from the above two types of replacement, we also replace those words listed in Table 3 to align them. [3]

[3]For a complete list of all the replacements, please refer to file `./Data/synonyms.csv` and variable `replace_dict` in file `./Code/Feat/nlp_utils.py`

Table 2: Synonym Replacement

synonyms	replacement
child, kid	kid
bicycle, bike	bike
refrigerator, fridge, freezer	fridge
fragrance, perfume, cologne, eau de toilette	perfume

Table 3: Other Replacement

original	replacement
nutri system	nutrisystem
soda stream	sodastream
playstation	ps
ps 2	ps2
ps 3	ps3
ps 4	ps4
coffeemaker	coffee maker
k-cup	k cup
4-ounce	4 ounce
8-ounce	8 ounce
12-ounce	12 ounce
ounce	oz
hardisk	hard drive
hard disk	hard drive
harley-davidson	harley davidson
harleydavidson	harley davidson
doctor who	dr who
levi strauss	levis
mac book	macbook
micro-usb	micro usb
video games	videogames
game pad	gamepad
western digital	wd

2.3 Stemming

We also performed stemming before generating features (e.g., counting features and BOW/TF-DF features) with Porter stemmer or Snowball stemmer from NLTK package (i.e., `nltk.stem.PorterStemmer()` and `nltk.stem.SnowballStemmer()`).

3 Feature Extraction/Selection

Before proceeding to describe the features, we first introduce some notations. We use tuple (q_i, t_i, d_i) to denote the i-th sample in `train.csv` or `test.csv`, where q_i is the `query`, t_i is the `product_title`, and d_i is the `product_description`. For `train.csv`, we further use r_i and v_i to denote `median_relevance` and `relevance_variance`[4], respectively. We use function ngram(s, n) to extract string/sentence s's n-gram (splitted by whitespace), where $n \in \{1, 2, 3\}$ if not specified. For example

$$\text{ngram}(\text{bridal shower decorations}, 2) = [\text{bridal shower}, \text{shower decorations}]^5$$

All the features are extracted for each run (i.e., repeated time) and fold (used in cross-validation and ensembling), and for the entire training and testing set (used in final model building and generating submission).

In the following, we will give a description of the features we have developed during the competition, which can be roughly divided into four types.

3.1 Counting Features

We generated counting features for $\{q_i, t_i, d_i\}$. For some of the counting features, we also computed the ratio following the suggestion from Owen Zhang [1].

The file to generate such features is provided as **genFeat_counting_feat.py**.

3.1.1 Basic Counting Features

- **Count of n-gram**
  count of ngram(q_i, n), ngram(t_i, n), and ngram(d_i, n).
- **Count & Ratio of Digit**
  count & ratio of digits in q_i, t_i, and d_i.
- **Count & Ratio of Unique n-gram**
  count & ratio of unique ngram(q_i, n), ngram(t_i, n), and ngram(d_i, n).
- **Description Missing Indicator**
  binary indicator indicating whether d_i is empty.

3.1.2 Intersect Counting Features

- **Count & Ratio of a's n-gram in b's n-gram**
  Such features were computed for all the combinations of $a \in \{q_i, t_i, d_i\}$ and $b \in \{q_i, t_i, d_i\}$ $(a \neq b)$.

[4]This is actually the standard deviation (std).

[5]Note that this is a list (e.g., `list` in Python), not a set (e.g., `set` in Python).

3.1.3 Intersect Position Features

- **Statistics of Positions of a's n-gram in b's n-gram**
  For those intersect n-gram, we recorded their positions, and computed the following statistics as features.
  - minimum value (0% quantile)
  - median value (50% quantile)
  - maximum value (100% quantile)
  - mean value
  - standard deviation (std)
- **Statistics of Normalized Positions of a's n-gram in b's n-gram**
  These features are similar with above features, but computed using positions normalized by the length of a.

3.2 Distance Features

Jaccard coefficient

$$\text{JaccardCoef}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

and Dice distance

$$\text{DiceDist}(A, B) = \frac{2|A \cap B|}{|A| + |B|} \tag{2}$$

are used as distance metrics, where A and B denote two sets respectively. For each distance metric, two types of features are computed.

The file to generate such features is provided as **genFeat_distance_feat.py**.

3.2.1 Basic Distance Features

The following distances are computed as features

- $D(\text{ngram}(q_i, n), \text{ngram}(t_i, n))$
- $D(\text{ngram}(q_i, n), \text{ngram}(d_i, n))$
- $D(\text{ngram}(t_i, n), \text{ngram}(d_i, n))$

where $D(\cdot, \cdot) \in \{\text{JaccardCoef}(\text{set}(\cdot), \text{set}(\cdot)), \text{DiceDist}(\text{set}(\cdot), \text{set}(\cdot))\}$, and $\text{set}(\cdot)$ converts the input to a set.

3.2.2 Statistical Distance Features

These features are inspired by Gilberto Titericz and Stanislav Semenov's winning solution [2] to Otto Group Product Classification Challenge on Kaggle. They are computed for `product_title` and `product_description`, respectively. Take `product_title` for examples. They are computed in the following steps.

1. group the samples by `median_relevance` and (`query`, `median_relevance`).

$$G_r = \{i \,|\, r_i = r\} \tag{3}$$

$$G_{q,r} = \{i \,|\, q_i = q, r_i = r\} \tag{4}$$

where $q \in \{q_i\}$ (i.e., all the unique `query`) and $r \in \{1, 2, 3, 4\}$.

2. compute distance between each sample and all the samples in each `median_relevance` level. Note that we excluded the current sample being considered when computing the distance. For $G_{q,r}$, we considered the group with same query as the current sample.

$$S_{i,r,n} = \{D(\text{ngram}(t_i, n), \text{ngram}(t_j, n)) \,|\, j \in G_r, j \neq i\} \tag{5}$$

$$SQ_{i,r,n} = \{D(\text{ngram}(t_i, n), \text{ngram}(t_j, n)) \,|\, j \in G_{q_i,r}, j \neq i\} \tag{6}$$

where $r \in \{1, 2, 3, 4\}$ and $D(\cdot, \cdot) \in \{\text{JaccardCoef}(\cdot, \cdot), \text{DiceDist}(\cdot, \cdot)\}$.

3. for $S_{i,r,n}$ and $SQ_{i,r,n}$, respectively, compute statistics such as
   - minimum value (0% quantile)
   - median value (50% quantile)
   - maximum value (100% quantile)
   - mean value
   - standard deviation (std)
   - more can be added, e.g., moment features and other quantiles

   as features.

3.3 TF-IDF Based Features

We extracted various TF-IDF features and the corresponding dimensionality reduction version via SVD (i.e., LSA). We also computed the (basic) cosine similarity and statistical cosine similarity.

3.3.1 Basic TF-IDF Features

The file to generate such features is provided as **genFeat_basic_tfidf_feat.py**.

- **TF-IDF Features**
  We extracted TF-IDF features from $\{q_i, t_i, d_i\}$, respectively. We considered unigram & bigram & trigram (in Sklearn's `TfidfVectorizer`, set `ngram_range`=(1,3).)
  - **Common Vocabulary**
    Note that to ensure the TF-IDF feature vectors of $\{q_i, t_i, d_i\}$ are projected into the same vector space, we first concatenated $\{q_i, t_i, d_i\}$, and then fit a TF-IDF transformer to obtain the common vocabulary. We then used this common vocabulary to generate TF-IDF features for $\{q_i, t_i, d_i\}$, respectively.

  - **Individual Vocabulary**
    We fit TF-IDF transformer for $\{q_i, t_i, d_i\}$, separately, with individual vocabulary.

- **Basic Cosine Similarity**
  With previous generated TF-IDF features (using common vocabulary), we computed the cosine similarity of
  - q_i and t_i
  - q_i and d_i
  - t_i and d_i
- **Statistical Cosine Similarity**
  Since cosine similarity is a distance metric, we also computed statistical cosine similarity as in Sec. 3.2.2.
- **SVD Reduced Features**
  We performed SVD to the above TF-IDF features to obtain a dimension reduced feature vector. Such reduced version was mostly used together with non-linear models, e.g., random forest and gradient boosting machine.
  - **Common SVD**
    We first concatenated the TF-IDF vectors of $\{q_i, t_i, d_i\}$ (using common vocabulary), and fit a SVD transformer.
  - **Individual SVD**
    We fit a SVD transformer for TF-IDF vectors of $\{q_i, t_i, d_i\}$, separately.
- **Cosine Similarity Based on SVD Reduced Features**
  We computed cosine similarity based on SVD reduced features (using common SVD).
- **Statistical Cosine Similarity Based on SVD Reduced Features**
  We computed statistical cosine similarity based on SVD reduced features.

3.3.2 Cooccurrence TF-IDF Features

We extracted TF-IDF for cooccurrence terms between

- `query` unigram/bigram and `product_title` unigram/bigram
- `query` unigram/bigram and `product_description` unigram/bigram
- `query` id (`qid`) and `product_title` unigram/bigram
- `query` id (`qid`) and `product_description` unigram/bigram

We give an example to explain what's cooccurrence terms. Consider sample with `id` = 54 in `train.csv` (see Table 4). For this sample, we have (after converting to lowercase)

- cooccurrence terms for `query` unigram and `product_title` unigram is
  [silver fremada, silver sterling, silver silver, silver freeform, silver necklace, necklace fremada, necklace sterling, necklace silver, necklace freeform, necklace necklace]

Table 4: One sample in `train.csv`

id	query	product_title
54	silver necklace	fremada sterling silver freeform necklace

- cooccurrence terms for `query` bigram and `product_title` unigram is [silver necklace fremada, silver necklace sterling, silver necklace silver, silver necklace freeform, silver necklace necklace]

We have found that such features are very useful for linear model (e.g., XGBoost with linear booster). We suspect it is because these features add nonlinearity to the model. We also performed SVD to such features though we haven't found much gain using the corresponding SVD features.

The file to generate such features is provided as **genFeat_cooccurrence_tfidf_feat.py**.

3.4 Other Features

3.4.1 Query Id

one-hot encoding of the `query` (generated via `genFeat_id_feat.py`)

3.5 Feature Selection

For feature selection, we adopted the idea of "untuned modeling" as used in Marios Michailidis and Gert Jacobusse's 2nd place solution [3] to Microsoft Malware Classification Challenge on Kaggle. The same model is always used to perform cross validation on a (combined) set of features to test whether it improves the score compared to earlier feature sets. For features of high dimension (denoted as "High"), e.g., feature set including raw TF-IDF features, we used XGBoost with linear booster (MSE objective); otherwise, we used `ExtraTreesRegressor` in Sklearn for features of low dimension (denoted as "Low").

Note that with ensemble selection, one can train model library with various feature set and rely on ensemble selection to pick out the best ensemble within the model library. However, feature selection is still helpful. Using the above feature selection method, one can first identified some (possible) well performed feature set, and then trained model library with it. This helps to reduce the computation burden to some extent.

4 Modeling Techniques and Training

4.1 Cross Validation Methodology

4.1.1 The Split

Early in the competition, we have been using `StratifiedKFold` on `median_relevance` or `query` with $k = 5$ or $k = 10$, but there was a large gap between our CV score and

Table 5: CV score and LB score

CV Mean	CV Std	Public LB	Private LB	CV Method	Repeated Time
0.642935	0.003694	0.63773	0.66185	3-fold CV	10
0.661263	0.008021	0.66529	0.69208	3-fold CV	3
0.664184	0.008027	0.66775	0.69596	3-fold CV	3
0.668797	0.008394	0.67020	0.69509	3-fold CV	3
0.669313	0.007969	0.67166	0.69267	3-fold CV	3
0.669399	0.006669	0.67275	0.69135	3-fold CV	3

Public LB score. We then changed our CV method to `StratifiedKFold` on `query` with $k = 3$, and used *each 1 fold as training set* and *the rest 2 folds as validation set*. This is to mimic the training-testing split of the data as pointed out by Kaggler @Silogram. With this strategy, our CV score tended to be more correlated with the Public LB score (see Table 5).

4.1.2 Following the Same Logic

Since this is an NLP related competition, it's common to use TF-IDF features. We have seen a few people fitting a TF-IDF transformer on the stacked training and testing set, and then transforming the training and testing set, respectively. They then use such feature vectors (**they are fixed**) for cross validation or grid search for the best parameters. They call such method as semi-supervised learning. In our opinion, if one is taking such method, he should refit the transformer using only the whole training set in CV, following the same logic.

On the other hand, if one fit the transformer on the training set (for the final model building), then in CV, he should also refit the transformer on the training fold only. This is the method we used. Not only for TF-IDF transformer, but also for other transformations, e.g., normalization and SVD, one should make sure he is following the same logic in both CV and the final model building.

4.2 Model Objective and Decoding Method

In this competition, submissions are scored based on the *quadratic weighted kappa*, which measures the agreement between two ratings. This metric typically varies from 0 (random agreement between raters) to 1 (complete agreement between raters).

Results have 4 possible ratings, $\{1, 2, 3, 4\}$. Each search record is characterized by a tuple (e_a, e_b), which corresponds to its scores by Rater A (human) and Rater B (predicted). The quadratic weighted kappa is calculated as follows. First, an $N \times N$ histogram matrix O is constructed, such that $O_{i,j}$ corresponds to the number of search records that received a rating i by A and a rating j by B. An $N \times N$ matrix of weights, w, is calculated based on the difference between raters' scores:

$$w_{i,j} = \frac{(i-j)^2}{(N-1)^2} \tag{7}$$

An $N \times N$ histogram matrix of expected ratings, E, is calculated, assuming that there is no correlation between rating scores. This is calculated as the outer product between each rater's histogram vector of ratings, normalized such that E and O have the same sum.

From these three matrices, the quadratic weighted kappa is calculated as:

$$\kappa = 1 - \frac{\sum_{i,j} w_{i,j} O_{i,j}}{\sum_{i,j} w_{i,j} E_{i,j}} \tag{8}$$

4.2.1 Classification

Since the relevance score is in $\{1, 2, 3, 4\}$, it is straightforward to apply multi-classification to the problem (using softmax loss). To convert the raw prediction (i.e., probabilities of four classes) to a single integer score, we can set it to the class label with the highest probability (i.e., `argmax`). However, we can achieve better score via the following strategy.

1. convert the four probabilities to a score via: $s = \sum_i iP_i$, i.e., weighted sum of the four probabilities.
2. calculate the pdf/cdf of each `median_relevance` level, 1 is about 7.6%, $1 + 2$ is about 22%, $1 + 2 + 3$ is about 40%, and $1 + 2 + 3 + 4$ is 100%.
3. rank the raw prediction in an ascending order.
4. set the first 7.6% to 1, $7.6\% - 22\%$ to 2, $22\% - 40\%$ to 3, and the rest to 4.

In CV, the pdf/cdf is calculated using training fold only, and in final model training, it is computed using the whole training data.

This also applies to One-Against-All (OAA) classification, e.g., `LogisticRegression` in Sklearn.

4.2.2 Regression

Classification doesn't take into account the weight $w_{i,j}$ in κ, and the magnitude of the rating. With $w_{i,j}$'s form, it is convincing to apply regression (with mean-squared-error, MSE) to predict the relevance score. In prediction phase, we can convert the raw prediction score to $\{1, 2, 3, 4\}$ following step 2-4 as in Sec. 4.2.1.

It turns out that MSE is the best objective among all the alternatives we have tried during the competition. For this reason, we mostly used regression to predict `median_relevance`.

4.2.3 Pairwise Ranking

We have tried pairwise ranking (LambdaMart) within XGBoost, but didn't obtain acceptable performance (it was worse than softmax).

4.2.4 Oridinal Regression

We have also tried to treat the task as an ordinal regression problem, and have implemented the following two methods: EBC and COCR (and the corresponding decoding method). It turned out COCR has superior performance than EBC, but is on a similar edge with softmax.

- **Extended Binary Classification (EBC)**
  This method is implemented within the XGBoost framework using customized objective. The objective and the corresponding decoding method of this method are in file `./Code/Model/utils.py`: `ebcObj` and `applyEBCRule`, respectively. For details of the EBC method, please refer to [6].
- **Cost-sensitive Ordinal Classification via Regression (COCR)**
  This method is implemented within the XGBoost framework using customized objective too. The objective and the corresponding decoding method of this method are in file `./Code/Model/utils.py`: `cocrObj` and `applyCOCRRule`, respectively. For details of the COCR method, please refer to [9].

4.2.5 Softkappa

Wse have tried to maximize κ directly. To that goal, we re-write it in a soft-version using class probabilities.[6] The objective is in file `./Code/Model/utils.py`: `softkappaObj`. The decoding method is the same as softmax and the performance is similar.

4.3 Sample Weighting

We are provided with the variance of the relevance scores given by raters. Such variance can be seem as a measure of the confidence of the ratings, and utilized to weight each sample. We have tried to weight the samples according to their variance, and it gives about 0.003 improvement. We have found the following weighting strategy works well in our models

$$w_i = \frac{1}{2}(1 + \frac{\hat{v}_m - \hat{v}_i}{\hat{v}_m}) = 1 - \frac{\hat{v}_i}{2\hat{v}_m} \tag{9}$$

where $\hat{v}_i = \sqrt{v_i}$ and $\hat{v}_m = \max_i \hat{v}_i$.

Most of our models (see Table 6) used weighted data, and a few didn't as to

- generate diverse predictions for the ensemble;
- sample weighting is not supported, e.g., `Lasso` in Sklearn.

4.4 Ensemble Selection

For the ensemble, we use bagged ensemble selection [7]. One interesting feature of ensemble selection is its ability to build an ensemble optimized to an arbitrary metric, e.g., quadratic weighted kappa used in this competition. We have also made some modifications to the original algorithm. Firstly, the model library is built with parameters

[6] Due to time and space constraints, we are not able to provide the detailed derivation in this version.

Table 6: Model Library

Package	Model		Feature	Weighting
XGBoost	`gblinear`	MSE	High/Low	Yes
		COCR		
		Softmax		
		Softkappa		
	`gbtree`	MSE	Low	Yes
		COCR		
		Softmax		
		Softkappa		
Sklearn	`GradientBoostingRegressor`		Low	Yes
	`ExtraTreesRegressor`		Low	Yes
	`RandomForestRegressor`		Low	Yes
	`SVR`		Low	Yes
	`Ridge`		High/Low	Yes
	`Lasso`		High/Low	No
	`LogisticRegression`		High/Low	No
Keras	NN Regression		Low	No
RGF	Regression		Low	No

of each model guided by a parameter searching algorithm. Secondly, model weight optimization is allowed in the procedure of ensemble selection. Thirdly, we used random weight for ensembling model similar as `ExtraTreesRegressor`. In the following, we will detail our ensemble methodology.

4.4.1 Model Library Building via Guided Parameter Searching

Ensemble selection needs a model library contains lots (hundreds or thousands) of models trained used different algorithm (e.g., XGBoost or NN, see Table 6 for the algorithms we used) or different parameters (how may trees/layers/hidden units) or different feature sets. For each algorithm, we specified a parameter space, and used TPE method [5] in Hyperopt package [4] for parameter searching. It not only find the best parameter setting for each algorithm, but also create a model library with various parameter settings guided or provided by Hyperopt.

During parameter searching, we trained a model with each parameter setting on training fold for each run and each fold in cross-validation, and saved the rank of the prediction of the validation fold to disk. Note that such rank was obtained using the corresponding decoding method as in step 2-4 of Sec. 4.2.1. They were used in ensemble selection to find the best ensemble. We also trained a model with the same parameter setting on the whole training set, and saved the rank of the prediction of the testing set. Such rank predictions were used for generating the final ensemble submission.

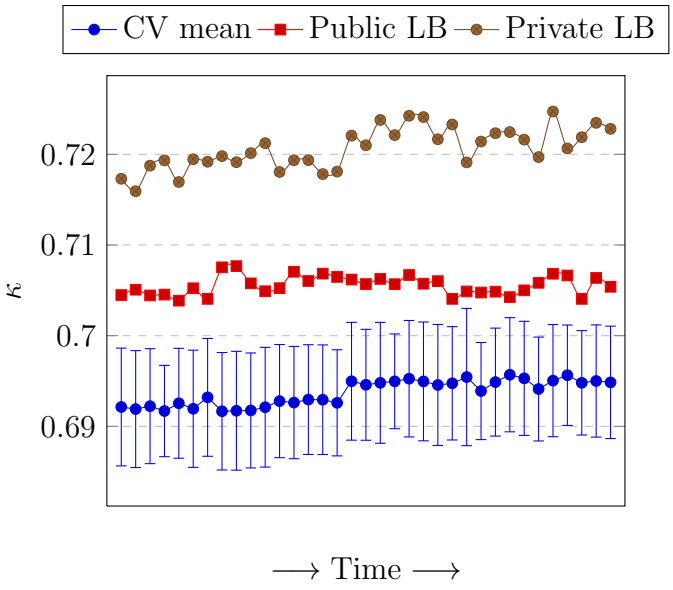

Figure 2: CV mean, Public LB, and Private LB scores of our 35 best Public LB submissions generating with randomized ensemble selection. One standard deviation of the CV score is plotted via error bar.

4.4.2 Model Weight Optimization

In the original ensemble selection algorithm, the model is added to the ensemble with hard weight 1. However, this is not guaranteed for best performance. We have modified it to allow weight optimized for each model when adding to the ensemble. The weight is optimized with Hyeropt too. This gives better performance than hard weight 1 in our preliminary comparison.

4.4.3 Randomized Ensemble Selection

The final method we used to generate the winning solution is actually without model weight optimization. On the contrary, we replaced weight optimization with **random weight**. This is inspired by the `ExtraTreesRegressor` to reduce the model variance (or the risk of overfitting). Figure 2 shows the CV mean, Public LB, and Private LB scores of our 35 best Public LB submissions generated with this method. As shown, CV score is correlated with the Public LB and Private LB, while it's more correlated with the latter. As time went by, we have trained more and more different models, which turned out to be helpful for ensemble selection in both CV and Private LB (as shown in Figure 2). Finally, the winning solution that scored **0.70807** on Public LB and **0.72189** on Private LB is just a median ensemble of these 35 best Public LB submissions.

5 Code Description

The implementation is organized in the following three parts.

5.1 Setting

- **param_config.py**: This file provides parameter configurations for the project.

5.2 Feature

All the files are in the folder `./Code/Feat`.

- **ngram.py**: This file provides functions to compute n-gram & n-term.
- **replacer.py**: This file provides functions to perform synonym & antonym replacement. Such functions are adopted from [8] (Chapter 2, Page 39-43.)
- **nlp_utils.py**: This file provides functions to perform NLP task, e.g., TF-IDF and POS tagging.
- **feat_utils.py**: This file provides utils for generating features.
- **preprocess.py**: This file preprocesses data.
- **gen_info.py**: This file generates the following info for each run and fold, and for the entire training and testing set.
  1. training and validation/testing data
  2. sample weight
  3. cdf of the `median_relevance`
  4. the group info for pairwise ranking in XGBoost
- **gen_kfold.py**: This file generates the `StratifiedKFold` sample indices which will be kept fixed in **ALL** the following model building parts.
- **genFeat_id_feat.py**: This file generates the following features for each run and fold, and for the entire training and testing set.
  1. one-hot encoding of query ids (qid)
- **genFeat_counting_feat.py**: This file generates the counting features described in Sec. 3.1 for each run and fold, and for the entire training and testing set.
- **genFeat_distance_feat.py**: This file generates the distance features described in Sec. 3.2 for each run and fold, and for the entire training and testing set.
- **genFeat_basic_tfidf_feat.py**: This file generates the basic TF-IDF features described in Sec. 3.3.1 for each run and fold, and for the entire training and testing set.
- **genFeat_cooccurrence_tfidf_feat.py**: This file generates the cooccurrence TF-IDF features described in Sec. 3.3.2 for each run and fold, and for the entire training and testing set.

- **combine_feat.py**: This file provides modules to combine features and save them in svmlight format.
- **combine_feat_[LSA_and_stats_feat_Jun09]_[Low].py**: This file generates one combination of feature set (Low).
- **combine_feat_[LSA_svd150_and_Jaccard_coef_Jun14]_[Low].py**: This file generates one combination of feature set (Low).
- **combine_feat_[svd100_and_bow_Jun23]_[Low].py**: This file generates one combination of feature set (Low).
- **combine_feat_[svd100_and_bow_Jun27]_[High].py**: This file generates one combination of feature set (High). **Such features are used to generate the best single model with linear model**, e.g.,
  - XGBoost with linear booster (MSE objective)
  - `Ridge` in Sklearn
- **run_all.py**: This file generates all the features and feature sets in one shot.

5.3 Model

- **utils.py**: This file provides functions for
  - various customized objectives used together with XGBoost
  - various decoding method for different objectives
    - MSE
    - Pairwise ranking
    - Softmax
    - Softkappa
    - EBC
    - COCR
- **ml_metrics.py**: This file provides functions to compute quadratic weighted kappa. It is adopted from `https://github.com/benhamner/Metrics/tree/master/Python/ml_metrics`.
- **train_model.py**: This file trains various models.
- **generate_best_single_model.py**: This file generates the best single model.
- **model_library_config.py**: This file provides model library configurations for ensemble selection.
- **generate_model_library.py**: This file generates model library for ensemble selection.
- **ensemble_selection.py**: This file contains ensemble selection module.
- **generate_ensemble_submission**: This file generates submission via ensemble selection.

6 Dependencies

We used Python 2.7.8, with the following libraries and modules:

- os, re, csv, sys, copy, cPickle
- NumPy 1.9.2
- SciPy 0.15.1
- pandas 0.14.1
- nltk 3.0.0
- bs4 4.3.2
- sklearn 0.16.1
- hyperopt (Developer version, `https://github.com/hyperopt/hyperopt`)
- keras 0.1.1 (`https://github.com/fchollet/keras/releases/tag/0.1.1`)
- XGBoost-0.4.0 (Windows Executable, `https://github.com/dmlc/XGBoost/releases/tag/v0.40`)
- ml_metrics (`https://github.com/benhamner/Metrics/tree/master/Python/ml_metrics`)

In addition to the above Python modules, we used

- rgf1.2 (Windows Executable, `http://stat.rutgers.edu/home/tzhang/software/rgf/`)
- libfm-1.40.windows (Windows Executable, `http://www.libfm.org/`)

7 How To Generate the Solution (aka README file)

1. download data from the competition website and put all the data into folder `./Data`.
2. run `python ./Feat/run_all.py` to generate feature set. This will take a few hours.
3. run `python ./Model/generate_best_single_model.py` to generate the best single model submission. In our experience, it only takes a few trials to generate model of best performance or similar performance. See the training log in `./Output/Log/[Pre@solution]_[Feat@svd100_and_bow_Jun27]_[Model@reg_xgb_linear]_hyperopt.log` for example.
4. run `python ./Model/generate_model_library.py` to generate model library. This is quite time consuming. **But you don't have to wait for this script to finish: you can run the next step once you have some models trained.**
5. run `python ./Model/generate_ensemble_submission.py` to generate submission via ensemble selection.

8 Additional Comments and Observations

Some interesting insights we got during the competition:

- spelling correction and synonyms replacement are very useful for searching query relevance prediction
- linear models can be much better than tree-based models or SVR with RBF/poly kernels when using raw TF-IDF features
- linear models can be even better if you introduce appropriate nonlinearities
- ensemble of a bunch of diverse models helps a lot
- Hyperopt is very useful for parameter tuning, and can be used to build model library for ensemble selection

9 Simple Features and Methods

Without any stacking or ensembling, the best (Public LB) single model we have obtained during the competition was an XGBoost model with linear booster. It is with Public LB score: **0.69322** and Private LB score: **0.70768**. Apart from the counting features and distance features, it used raw basic TF-IDF and raw cooccurrence TF-IDF.

To reproduce the best single model, run

```
> python ./Code/Feat/combine_feat_[svd100_and_bow_Jun27].py
```

to generate the feature set we used, and

```
> python ./Code/Model/generate_best_single_model.py
```

to train the XGBoost model with linear booster. Note that due to randomness in the Hyperopt routine, it won't generate exactly the same score, but a score very similar or even better. Note that, you can also try other linear models, e.g., `Ridge` in Sklearn.

10 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. We would also like to thank the Kaggle team and CrowdFlower for organizing this competition.

References

[1] http://nycdatascience.com/featured-talk-1-kaggle-data-scientist-owen-zhang/.

[2] https://www.kaggle.com/c/otto-group-product-classification-challenge/forums/t/14335/1st-place-winner-solution-gilberto-titericz-stanislav-semenov.

[3] https://www.kaggle.com/c/malware-classification/forums/t/13863/2nd-place-code-and-documentation.

[4] `http://hyperopt.github.io/hyperopt/`.

[5] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems: Proceedings of the 2011 Conference (NIPS '11)*, pages 2546–2554, 2011.

[6] Ling Li and Hsuan-Tien Lin. Ordinal regression by extended binary classification. In *Advances in Neural Information Processing Systems: Proceedings of the 2006 Conference (NIPS '06)*, pages 865–872, 2006.

[7] Alexandru Niculescu-Mizil, Rich Caruana, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. *Proceedings of International Conference on Machine Learning*, pages 137–144, 2004.

[8] Jacob Perkins. *Python Text Processing with NLTK 2.0 Cookbook*. Nov. 2010.

[9] Yu-Xun Ruan, Hsuan-Tien Lin, and Ming-Feng Tsai. Improving ranking performance with cost-sensitive ordinal classification via regression. *Information Retrieval*, 17(1):1–20, 2014.