"Relevance Prediction"

Klearchos Stavrothanasopoulos
International Hellenic University,
Thessaloniki, Greece
k.stavrothanasopoulos@ihu.edu.gr

Iordanis Papoutsoglou
International Hellenic University
Thessaloniki, Greece
i.papoutsoglou@ihu.edu.gr

Abstract- The purpose of this paper is to describe the developing process of a model that can accurately predict the relevance of search results with the actual product.

Keywords: Relevance Prediction, search term, product

I. INTRODUCTION

Search relevancy is the practice of turning a search engine into a helpful sales associate. In the same way the associate understood what clients meant when looking for a specific product in the store, relevant search can do the same for an online store. Poor relevancy is the modern equivalent of a lazy sales associate that seems unwilling or unable to help. Good search relevancy, on the other hand, keeps users on the site. They're delighted by what comes up and they want to come back for more. [1] However good search relevancy is not an easily achievable goal. Natural languages, synonymy, homonymy, term frequency, norms, relevance model, boosting, performance, subjectivity are some of the reasons why search relevancy still remains a hard problem

II. DATA AND PROBLEM DESCRIPTION

Our dataset consists of 74067 search terms-queries. There are two more available data sets: one which contains a textual description of each different product and a third one which contains additional attributes for some of the products. The challenge is to predict a relevance score for the provided combinations of search terms and products. A text mining model has been built in order to solve this regression problem.

III. PERFORMANCE EVALUATION

Given a search term and the product title, we wish to predict whether there is any relevance among them or not and define the exact score. . The relevance score is a real number between 1 (not relevant) and 3 (highly relevant).

The regression performance is evaluated by the **root-mean-square deviation (RMSD) or root-mean-square error (RMSE)**

$$\mathrm{RMSD} = \sqrt{\frac{\sum_{t=1}^{T}(\hat{y}_t - y_t)^2}{T}}$$

In our model, we use K-Folds cross-validator (K=10) for the validation of our performance. This method provides train/test indices to split data in train/test sets. Dataset is splited into k consecutive folds, each fold is then used once as a validation while the k - 1 remaining folds form the training set [3]

IV. REGRESSOR SELECTION

Throughout the implementation of this project, we considered 3 regressors:

- **Random Forest Regressor (RFR)** fits a number of classifying decision trees on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size [4]
- Our best result using RFR was: **50,0158%**

```
In [123]: """Training the model using KFold validation (K=10)"""
     ...: from sklearn.model_selection import KFold
     ...: kf=KFold(n_splits=10, shuffle=True, random_state=False)
     ...: outcomesRf=[]
     ...: outcomesRf=[]
     ...: for train_id, test_id in kf.split(x,y):
     ...:     X_train, X_test = x[train_id], x[test_id]
     ...:     y_train, y_test = y[train_id], y[test_id]
     ...:     rf_best.fit(X_train,y_train)
     ...:     predictions = rf_best.predict(X_test)
     ...:     accuracy = fmean_squared_error(y_test, predictions)
     ...:     n=n+1
     ...:     print(n,accuracy)
     ...:     outcomesRf.append(accuracy)
     ...:
     ...: print(np.mean(outcomesRf))
1 0.49630515863521846
2 0.49453139632259735
3 0.4976332908072487
4 0.5019087631488313
5 0.5034748244208058
6 0.5037232994795944
7 0.5004600768861794
8 0.5056360982776805
9 0.49753342358668886
10 0.5003792336644145
0.500158556522926

In [124]:
```

- **Extra Trees Regressor (ETR)** fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. [5]

Main differences in regards with the RFR are that RFs are often more compact than ETs. ETs are generally cheaper to train from a computational point of view but can grow much bigger.

Our best result using ETR was: **50,9907%**

```
In [125]: """Training the model using KFold validation (K=10)"""
     ...: from sklearn.model_selection import KFold
     ...: kf=KFold(n_splits=10, shuffle=True, random_state=False)
     ...: outcomesRf=[]
     ...: outcomesRf=[]
     ...: for train_id, test_id in kf.split(x,y):
     ...:     X_train, X_test = x[train_id], x[test_id]
     ...:     y_train, y_test = y[train_id], y[test_id]
     ...:     rf_best.fit(X_train,y_train)
     ...:     predictions = rf_best.predict(X_test)
     ...:     accuracy = fmean_squared_error(y_test, predictions)
     ...:     n=n+1
     ...:     print(n,accuracy)
     ...:     outcomesRf.append(accuracy)
     ...:
     ...: print(np.mean(outcomesRf))
1 0.5091512592138133
2 0.5057921663483478
3 0.5019582605682911
4 0.5136370042218199
5 0.5161365845816075
6 0.5093123223841096
7 0.5096718221317961
8 0.5120718648996117
9 0.5074134280678103
10 0.5139229254391415
0.5099067637856349
```

- **Gradient Boosting Regressor** builds an additive model in a forward stage-wise fashion. It allows for the optimization of arbitrary differentiable loss functions. In each stage a regression tree is fit on the negative gradient of the given loss function. [6]

Our best result using ETR was: **47,4243%**

```
In [127]: n=0
     ...:
     ...: """Training the model using KFold validation (K=10)"""
     ...: from sklearn.model_selection import KFold
     ...: kf=KFold(n_splits=10, shuffle=True, random_state=False)
     ...: outcomesRf=[]
     ...: outcomesRf=[]
     ...: for train_id, test_id in kf.split(x,y):
     ...:     X_train, X_test = x[train_id], x[test_id]
     ...:     y_train, y_test = y[train_id], y[test_id]
     ...:     rf_best.fit(X_train,y_train)
     ...:     predictions = rf_best.predict(X_test)
     ...:     accuracy = fmean_squared_error(y_test, predictions)
     ...:     n=n+1
     ...:     print(n,accuracy)
     ...:     outcomesRf.append(accuracy)
     ...:
     ...: print(np.mean(outcomesRf))
1 0.4723455789004026
2 0.47008065861809767
3 0.4683325026130444
4 0.4781157833636667
5 0.4781578928172266
6 0.475218018084708
7 0.47538386997714444
8 0.47809706867438084
9 0.4731698080383188
10 0.473528416299907005
0.4742429597386059
```

In order to gain the best results out of each regressor by tuning their hyper-parameters (parameters that are not directly learnt within the estimator) we have used the **Grid Search** method.

V. DATA PREPROCESSING AND FEATURE ENGINEERING

A. *Data preprocessing*

Data preprocessing is the data mining procedure that involves transforming raw data into an understandable format. Our main action for this part was the **stemming** method implementation. Stemming is a crude heuristic process that removes the ends of words and often includes the removal of derivational affixes, ending up with the stem [7] Prior to this action we have used some extra techniques. Firstly, we have converted text to lowercase. Afterwards we replaced a group of words with one common word (i.e amperes/ampere/amps to amp)

In addition, **tokenization** has been applied. Tokenization is a step which splits longer strings of text into smaller pieces, or tokens. In this project larger chunks of text were tokenized into sentences, sentences were tokenized into words.

B. *Feature Engineering*

Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work. If feature engineering is done correctly, it increases the predictive power of machine learning algorithms by creating features from raw data that help facilitate the machine learning process.

A series of features have been extracted to optimize our results.

- **len_of_query :** the number of words for each query
- **len_of_title:** product title's number of words
- **len_of_description:** product description's number of words
- **len_of_brand:** product brand name's number of words
- **query_in_title:** whole query exists in title or not
- **query_in_description:** whole query exists in product description
- **word_in_title:** number of query's words that exists in title
- **word_in_description:** number of query's words that exists in product description
- **ratio_title:** ratio of the query's words existing in product description over the total query's words
- **word_in_brand :** number of query's words that exists in brand name of the product
- **ratio_brand:** ratio of the query's words existing in brand over the total brand's words

VI. CONCLUSION

We used a total of 11 features in our model. Most of them were based on the number of common words found in the tittle,description,brand.The best score we obtained was 0.4742 with a gradient boosting regressor with its hyper-parameters tuned by the grid search method. From all the regressors we tested, we observed that boosting technique gives us the greatest result.

[1] https://opensourceconnections.com/blog/2014/06/10/what-is-search-relevancy/

[2] https://findwise.com/blog/improve-search-relevance-using-machine-learning-statistics-apache-solr-learning-rank/

[3] http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

[4] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

[5] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesRegressor.html

[6] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

[7] https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html