INTERNATIONAL HELLENIC UNIVERSITY

RELEVANCE PREDICTION

ASSIGNMENT

Christou Christos
Latsiou Konstantina

Academic Year 2020-2021

Natural Language Processing and Text Mining

Professors
Berberidis Christos
Papadopoulos Apostolos

Contents

The assignment

The task that had to be performed was to predict the relevance of a result with respect to a query based on three datasets (*train.csv, product_descriptions.csv, attributes.csv*). Each query is represented by a search term or terms and the respective result, which is a specific product. Based on the relevance between the query and the answer, a relevance score is assigned. The higher the score the better the relevance of the answer. The relevance is a real number in the range [1,3] (in steps of 0.5), where 1 denotes minimum relevance and 3 is the maximum relevance.
The evaluation measure used was the Root-Mean-Square Error (RMSE).

The datasets

As mentioned above, for the relevance prediction that had to be performed there were three datasets available. The datasets in more detail are the following; The **train.csv** contains pairs of queries -the search terms-, the answer to the query and the corresponding relevance score. The second dataset is **product_descriptions.csv,** which contains a textual description for each product. The last dataset is the **attributes.csv** and contains additional attributes for some of the products.

Pre-processing

In our implementation, we used all the files. In detail, we used all columns from both the *train.csv* and the *product_descriptions.cs*v. From the *attributes.csv* we only kept attributes with name “MFG Brand Name”, in a data frame called “*brand*”. The three DataFrames were joined on the “*product_uid*” column, which was the common feature for all three datasets.

In the second step, we conducted various text preprocessing tasks, aiming to improve the performance of our models. Some tasks were specific to the datasets, while others were more general. The process that we followed for this part of the assignment is as follows:

First, we cleaned up the strings with the help of using Regular Expressions.

- Removed redundant spaces for dataset-specific patterns
- Standardized various measurement units instances like ‘inches’ & ‘in’ or ‘volts’ & ‘v’ into a particular string for each unit
- Dealt with URL encoding, like turning ‘ ’ into a space
- Removed HTML syntax

Furthermore, we tokenized the sentences and consequently stemmed words and converted them to lowercase.

At this point, it's worth mentioning that we used pickles extensively throughout our code. In Python, pickle is primarily used for serializing and deserializing an object structure. In other words, it's the process of converting a Python object into a byte stream to store it in a file/database, maintain program state across sessions or transport data over the network. Basically, it is a useful module for anything that requires a lot of processes and ends in an object, with the end goal of minimizing the time of the entire procedure.

Similarly, we used it to save the state of the dataset for important 'milestones'. For instance, we noticed that preprocessing was taking between 1.5 hours and 2 hours to complete. Saving the pre-processed data in a pickle allowed us to load the pre-processed data in a matter of seconds. Consequently, we were able to iteratively try news ideas at a much faster pace.

Text representation

Cleaning up the text is not enough in NLP applications. In order to train machine learning models, we have to somehow vectorize the text. For that reason, we applied four different methods, which are the following:

1. Levenshtein distance
2. TF-IDF
3. Word2Vec
4. Doc2Vec

Levenshtein distance

In our research for text similarity, we found FuzzyWuzzy, a Python library used for string matching. Fuzzy string matching is the process of finding strings that match a given pattern. It uses Levenshtein Distance to calculate the differences between sentences in a simple-to-use package.
We calculated the full as well as the partial similar ratio between the various columns of the original datasets. These rations were used as features in model training.

TF-IDF

One of the most popular methods to represent text is through the TF-IDF scheme, which is used as a weighting factor in text mining applications. In simple terms, TF-IDF attempts to highlight

important words which appear frequently in a document but not across documents. The terms are briefly explained below:

1. Term Frequency (TF): This summarizes the normalized Term Frequency within a document.
2. Inverse Document Frequency (IDF): This reduces the weight of terms that appear a lot across documents.

However, in TF-IDF sooner or later a big problem occurs, which is matrix size. Since TF-IDF matrix size is determined by the number of words in the dictionary, the bigger the dictionary the bigger the matrix size of the method which means more time and resources needed to process the matrix.[1]

As hyperparameters of our TF-IDF implementation, we set the ngram_range to have a lower boundary equal to one and an upper boundary equal to two. Additionally, we combined the truncated Singular Value Decomposition (SVD aka LSA) to achieve dimensionality reduction to the number of input variables in the training data.

Word2Vec

Having mentioned the above problem of the matrix size of the TF-IDF method, the solution to this is the Word2Vec method, which reduces the matrix dimension. Another advantage of the Word2Vec feature representation is that it captures the relationship between words. So for example, if we have the word ‘airplane’, Word2Vec will be able to tell us that the similar words are ‘airport’, ‘pilot’, and ‘turbine’.

We built several Word2Vec models, trying different combinations of hyperparameters. The first variation is which training algorithm to use; Skip-gram vs Continuous Bag-of-Words (CBOW). The second variation is the vector size of the model; 300 or 100.

Throughout all variations, the window size and minimum count were set to 5 and 2 respectively. The reason we chose a relatively small min count was the documents’ size, especially the search term. Having a larger required term frequency would not be very useful.

After building the Word2Vec model, we needed to somehow create sentence-level embeddings. So, we decided to formulate them, by taking the mean of the constituent word embeddings.

Finally, we used the cosine distance to calculate the similarity between the vectors and generate features.

Doc2Vec

Another method we used was Doc2Vec, which expands the idea of the Word2Vec method. The difference is that the former includes document ids in the process of creating the embeddings. The goal of Doc2Vec is to create a numeric representation of a document, regardless of its length.

We trained 2 different models, both with a vector size of 300. Their difference lies in the algorithm used; PV-DM (Paragraph Vector - Distributed Memory) vs PV-DBOW (Distributed BOW).

Model training and evaluation

Models

After processing and vectorized the text with the four above methods, it was about time to apply machine learning models. For that reason we chose the following four supervised models:

1. Random Forests
2. Ridge Regression
3. Gradient Boosting Regressor
4. XGBoost

Random Forests

Random Forest classifier is a supervised learning algorithm and an ensemble method. It trains several decision trees with bootstrapping followed by aggregation, jointly referred to as bagging. Bootstrapping indicates that several individual decision trees are trained in parallel on various subsets of the training dataset, using different subsets of available features. Bootstrapping ensures that each decision tree in the random forest is unique, reducing the RF classifier's overall variance. For the final decision, the RF classifier aggregates the decisions of individual trees; consequently, the RF classifier exhibits good generalization.

As for hyperparameters, we set the *n_estimators*, the number of trees in the forest, to 100. For the *max_depth*, which is the depth of the tree, we set it to 6. Lastly, we set the *random_state* to be equal to a specific number, in our case equal to zero, in order to have the subsets of train and test be always the same, every time we run the code.

Ridge Regression

Ridge regression is a special case of Tikhonov regularization, in which all parameters are regularized equally. Ridge regression is particularly useful to mitigate the problem of multicollinearity in linear regression, which commonly occurs in models with large numbers of parameters. When multicollinearity occurs, least squares estimates are unbiased, but their variances are large so they may be far from the true value. In general, the method provides improved efficiency in parameter estimation problems in exchange for a tolerable amount of bias. By adding a degree of bias to the regression estimates, ridge regression reduces the standard errors.[2]

For the Ridge model, we only used the *alpha* parameter, which is used for regularization strength reasons. We set the *alpha* parameter to 0.1.

Gradient Boosting Regressor

The 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.

For the hyperparameters, we chose the *n_estimators* to be set to its default value, which is 100. The *n_estimators* is the number of boosting stages to be performed. Gradient boosting is fairly robust to over-fitting, so a large number usually results in better performance. We also left the *learning_rate* to its default value of 0.1. So, the contribution of each tree is shrunk by 0.1. The *max_depth*, which limits the number of nodes in the tree, was set to 1 because it had the best interaction with the input variables. Once again we set in this model as in the Random Forest one the random_state to be equal to an arbitrary number, in our case we chose 0. Lastly, we set the *loss* parameter to be 'ls', which refers to least squares regression.

XGBoost

XGBoost is short for Extreme Gradient Boosting. It is a library designed and optimized for boosted tree algorithms. Its main goal is to push "the extreme" of the computation limits of machines to provide scalability, portability, and accuracy for large-scale tree boosting.

For the hyperparameters, we left the *n_estimators* to its default value, which is 100. The *n_estimators* is the number of boosting stages to be performed. Besides that, we adjusted the *learning_rate* to be equal to 0.08. We left the *gamma* parameter, which sets the minimum loss

reduction required to make a further partition on a leaf node of the tree, to its default value of 0. The larger gamma is, the more conservative the algorithm will be. Furthermore, we fixed the subsample parameter to 0.75. Setting it to 0.75 means that XGBoost would randomly sample three-quarters of the training data before growing trees and this prevented overfitting. The *colsample_bytree* is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed. We left that parameter to its default value. Lastly, for this model, we set the *max_depth* (maximum depth of a tree) equal to 7.

Evaluation

By looking at the below table, someone can notice that across all text representing methods, the one with the smallest RSME was the Levenshtein distance one, regardless of the model. In addition, the best performing learning model among all the different methods was XGBoost.

The table contains two more columns. The first is the time that took to build the representation. The second is the time to train them for all the learning methods. The fastest representation to built was the Levenshtein distance method. The Doc2Vec PV-DBOW 300 approach achieved the smallest training time. However, it required substantially more time to be built, compared to the Levenshtein distance one.
Below, we provide the tabular representation of the results of all the different methods in combination with all the different models that were used.

Method	RMSE				Time (s)	
	Random Forest	Ridge	Gradient Boosting	XGBoost	Build[1]	Train
Levenshtein distance	.4941	.4968	.4951	**.4892**	28.980	10.587
TF-IDF	.5166	.5246	.5183	**.5124**	169.885	17.786
WordVec Skip-gram 300[2]	.5035	.5108	.5055	**.5017**	453.759	18.683
WordVec CBOW 300[2]	.499	.5032	.5001	**.4988**	305.079	16.822
WordVec Skip-gram 100[2]	.5031	.5086	.5045	**.5018**	383.075	17.001
WordVec CBOW 100[2]	.5002	.5033	.5009	**.4997**	254.279	16.709
Doc2Vec PV-DBOW 300[2]	.5300	.5300	.5300	.5300	301.630	2.120
Doc2Vec PV-DM 300[2]	.5300	.5300	.5300	.5302	425.354	2.251

[1]. Does not include preprocessing
[2]. Vector size

References

[1] https://himang27s.medium.com/text-to-features-using-python-bb27e48f8d78 [Accessed 6th May 2021].

[2] https://en.wikipedia.org/wiki/Tikhonov_regularization#cite_note-1 [Accessed 10th May 2021].