Word Complexity Estimation

Zăvelcă Miruna-Andreea

December 29, 2021

1 Introduction

The scope of this project was to train a regression model that could best determine the complexity of a word. The input consists of several lines containing id number, a sentence where the word / phrase can be seen in use, the position where the target starts in the sentence as well as the position where it ends, the target word / phrase, number of native and non-native annotators together with the mean complexity score of each group and the final score of the target (overview).

For code simplicity I have decided to only train each model using the final complexity score, ignoring the separation by native / non-native annotators.

I have also ignored the target position in the example sentences. The position could have been used for finding parts of speech, word dependencies and everything that comes with it, but that would require additional training for finding these characteristics.

2 Preprocessing

Before training a model on the data set I had to choose an embedding. For this purpose I tested tfidf, word2vec and a custom embedding on different data sets to see which one works better. I tested the custom embedding both alone and in combination with other embeddings, to see which works best.

For embedding I first separated each sentence in lowercase tokens (*nltk.tokenize.word_tokenize*) and removed all characters except for letters, apostrophes and hyphens.

For embedding target phrases, I separate each phrase in words, preprocess them with the aforementioned approach and then calculate the mean for each feature in the embedding.

All of these embeddings were trained on the joined set of sentences from both the training set and the test set, with the final result being a pandas data frame with 2 columns: ”data” and ”label”.

2.1 TFIDF

Tfidf cares about the frequency of a word in a document, so I only need each separate word for its training.

2.2 Word2Vec

I first tried training this vectorizer only on the input set of sentences, but the results were not great. It obviously needed a bigger training set, so I added sentences from nltk.corpus.brown.

Word2vec also seemed to work better in combination with the custom embedding.

2.3 Custom

Inspiration [1]

Calculated a custom set of features to be considered for embedding a word. The final vector contains information about length, percentage of vowels in the word, number of double letters and maximum consecutive consonants.

Tried both this model alone and in combination with other embeddings. It seems to have way too little features to work alone, but can bring a plus to other embeddings.

Model	TFIDF	Word2vec	Custom	TFIDF + custom	Word2vec + custom
LinearSVR	train: 0.035 test: 0.081	train: 0.098 test: 0.104	train: 0.089 test: 0.094	train: 0.045 test: 0.086	train: 0.153 test: 0.158
KNeighborsRegressor	train: 0.043 test: 0.079	train: 0.038 test: 0.074	train: 0.097 test: 0.106	train: 0.041 test: 0.086	train: 0.039 test: 0.07

Table 1: Results for each embeddings on the tested models with their best coeficients

3 Scaling

I used the StandardScaler to normalize the input data, since word vectorizers were used and feature scaling could make a notable difference in model training. Moreover, the features obtained from the embedding were also sometimes combined with the custom features extracted from words, which makes normalizing the data even more important.

It seemed to have better results on word2vec after training it on a bigger data set, but since the data sets I used mostly were really small it didn't make a notable difference.

4 Models

4.1 LinearSVR

According to [2]

The Linear Support Vector Regression model tries to define a hyperplane to fit in the data. It is very flexible in terms of how much error it accepts and as such is a good choice for a linear regression approach. Instead of being constrained by only optimizing a specific function, this provides the freedom to fine tune its parameters so as to find an error tolerance that gives the best overall results.

For the parameters, first of all, there is epsilon, which helps shape the hyperplane that will fit the data. It represents the maximum error that will be accepted for points that fit inside

the hyperplane's boundaries. The regularization parameter, C, helps define the tolerance the model has for points outside of the epsilon range. I have tried various combinations for the 2 aforementioned parameters (epsilon in the range 0 – 0.1 and C in the range 0 – 100) to see what works best. For the loss parameter, which defines which loss function is used, I left the default option (L1 loss) and did the same for the tol parameter, as the default is fairly standard.

Since compilation time was quite fast, I also used this model for testing different embeddings.

C / **epsilon**	1	5	10	50	100
0	train: 0.039 test: 0.072	**train: 0.043** **test: 0.073**	train: 0.066 test: 0.096	train: 0.053 test: 0.081	train: 0.05 test: 0.082
0.05	train: 0.062 test: 0.087	train: 0.127 test: 0.149	train: 0.063 test: 0.09	train: 0.073 test: 0.1	train: 0.064 test: 0.091
0.1	train: 0.088 test: 0.108	train: 0.09 test: 0.111	train: 0.076 test: 0.097	train: 0.09 test: 0.111	train: 0.087 test: 0.108

Table 2: Results for LinearSVR with different values for C and epsilon

I had the best result with C=5, tfidf embedding and no scaling.

4.2 NuSVR

According to [3]

The Nu Support Vector Regression model is similar to the one described above, with the main difference that it has a nu parameter that replaces the epsilon parameter from before. This new parameter is used to control the number of support vectors used.

Since training takes a lot of time, I only tested it with C=10 and coef0=1.0 (parameters found in this paper [1]). These were also my best results in the competition (tfidf embedding, no scaling).

4.3 KNeighborsRegressor

According to [4]

This is a regression model based on the k-nearest neighbours technique. It is a fairly simple model, but quite powerful when little noise is expected, so it was worth trying on the given data.

One of the main parameters that I tried tuning was n_neighbours, the number of neighbours to be taken into consideration for each point. I have mostly tried values in the 1-10 range, as anything larger would start giving very bad results (perhaps the data points are moderately clustered). The other parameter that I have tweaked was weights, which defines which weight function to be used in the prediction. The best overall results were given by 'uniform', but I have also tried 'distance' for larger values of n_neighbours, in the off chance that it would balance out the negative effects of outliers.

embedding / **n_neighbours**	tfidf	word2vec + features
2	train: 0.04 test: 0.067	**train: 0.039** **test: 0.067**
5	train: 0.058 test: 0.072	train: 0.056 test: 0.072
7	train: 0.064 test: 0.076	train: 0.062 test: 0.075
10	train: 0.069 test: 0.078	train: 0.068 test: 0.077

Table 3: Results for KNeighborsRegressor with different embeddings and number of neighbours

I had the best results with n_neighbors=2, word2vec + custom embedding and scaled.

4.4 RandomForestRegressor

According to [5]

The Random Forest Regressor is a model that uses the average estimation of multiple classifying decision trees to make a prediction. Thanks to bagging and random feature selection, it is less prone to overfitting. It also handles high dimensionality well, which is why it seemed like a good option to try on the input, given the high number of features it has as a result of word vectorization.

The 2 parameters I have tweaked were n_estimators (the number of trees to be generated for the forest) and max_depth (the maximum depth a decision tree could have). The number of trees that I used was generally high, since the main disadvantage of having more is the code running slower, while they improve the accuracy of the estimator. I have tried both lower and higher numbers for the maximum depth, as increasing it might improve the accuracy, but make the model more prone to overfitting.

I had some promising results early on in the competition, but it started taking longer and longer time as I added features to my embeddings.

I had the best results with n_estimators=1000 and max_depth=300, tfidf embedding, no scaling.

References

[1] Andrei M. Butnaru and Radu Tudor Ionescu. Unibuckernel: A kernel-based learning method for complex word identification. https://arxiv.org/pdf/1803.07602.pdf.

[2] sklearn. Linearsvr, https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVR.html.

[3] sklearn. Nusvr, https://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVR.html.

[4] sklearn. Kneighborsregressor, https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighb

[5] sklearn. Randomforestregressor, https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.Randc