Project 1 PML – Visual Complexity Competition

Author: Marinescu Alexandru

Group: 407

Table of Contents:

1. Introduction

This is the documentation for a Kaggle Competition about **visual complexity** for a dataset containing English sentences.

The purpose is to label as accurately as possible different kinds of sentences with the score that will reflect how **visual** that sentence is. The score is supposed to quantify and measure how much visual information that sentence can suggest for us.

Example:

A sandwich and french bread sit on a cutting board with an apple. is assigned a score of: -0.(6)

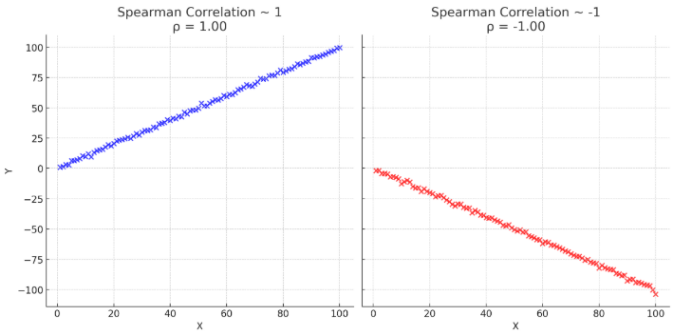
OLD BLACK AND WHITE PHOTO OF AN ALL BOYS SCHOOL. is assigned a score of: 1,(6)

The continuous range of scores means that this is a **regression problem**. This is also a **supervised task**, meaning that the contestants where provided with:

- Training data (labeled)
- Validation data (labeled)
- Testing data

The predictions will be done on the testing data and will be compared with the real values using the **Spearman's Rank Correlation** as a metric. This is a measure of linear dependency between 2 variables. The variables in this case are not the actual scores, but their indexes after one of the sets is sorted. It can take values between -1 and 1:

- 1 means they're completely dependent on each other (practically, after sorting, the order doesn't change)
- -1 means they're inversely dependent on each other (practically, we get the reverse order after sorting)



The baseline was 0.09677, but an ideal score would be closer to 1, since that is the closest to the real scores.

2. Related Work

While the rules mention to not use any pre-trained model, it's also a good idea to investigate if any similar work was done in the past. Since it's difficult to find an exact area of work, a good starting point is to simply the problem to a *"regression score on text"*.

One such example is **Assessing English language sentences readability using machine learning models**. This work want to predict the level of readability of sentences in English, and correlate it to the level of literacy from a target audience. The techniques mentioned there are: KNN (k nearest neighbors), SVM (support vector machine), LR (linear regression), NN (neural networks) and more

Since this was my first Kaggle competition, I decided to look for similar past competitions. A popular one is **CommonLit Readability Prize**. The purpose of this was to predict a level of complexity of certain passages in English for grades 3-12. While many models implied the transformer architecture which is our of my knowledge reach, some known solutions were also mentioned like SVR (support vector regression) and Ridge Regression, which proved efficient for the task.

3. Method

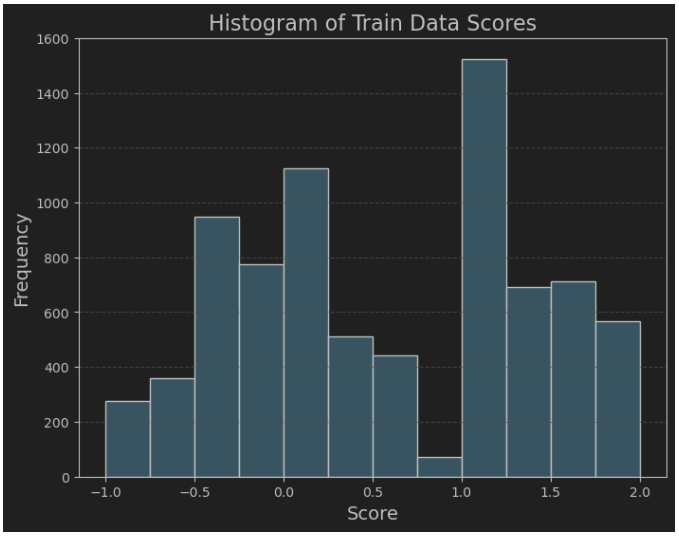
Dataset

As mentioned in the introduction section, the contestants where provided with the following:

- Training data (labeled)
- Validation data (labeled)
- Testing data

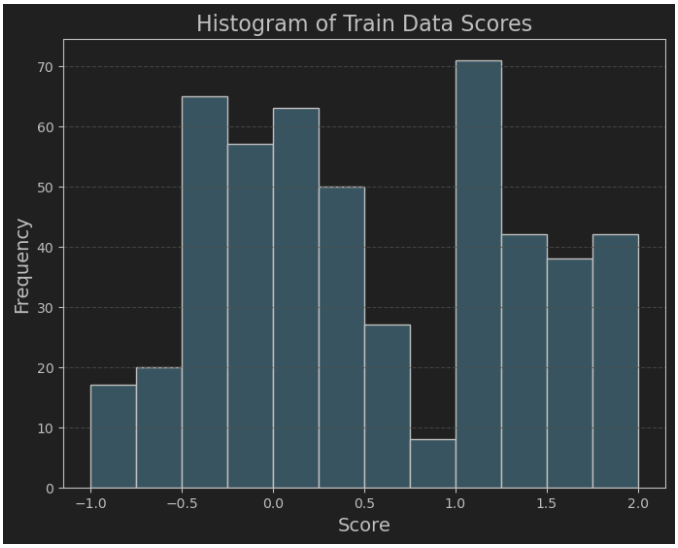
From this I started to explore each of the labeled data:

For training data, I found out that there are **8000** samples. Looking at the labels (the scores of each sentence) of these samples, I found out that they have a mean of **0.52079** and a standard deviation of **0.81712**. This is a histogram for the training data with bin size of 0.25:



The observation of *"2 valleys"* indicates that the data isn't uniformly distributed and we should be careful about overfitting and thus missing to generalize the less frequent cases.

For training data, I found out that there are **500** samples. Looking at the labels (the scores of each sentence) of these samples, I found out that they have a mean of **0. 48046** and a standard deviation of **0.815415**. This is a histogram for the training data with bin size of 0.25:



The mean and standard deviation are very close to the training data. More so, the histogram also looks very similar (there are a lot less samples, but a better measure would be to analyze the difference in the ratio of each bin value relative to total count:

$$\frac{\alpha_i}{8000} \; and \; \frac{\beta_i}{500} \; where \begin{matrix} \alpha_i \; is \; the \; i^{th} \; bin \; value \; in \; training \; data \\ \beta_i \; is \; the \; i^{th} \; bin \; value \; in \; validation \; data \end{matrix}$$

For example, for 3rd bin (-0.5 <-> -0.25) for training data we have the ratio *0.1205* and for the same bean in the validation data we have the ratio *0.13.* While the ratios differ, the overall distribution seems to the same with training data and won't help in generalizing the data (more on that later).

Test data size is **500**, which gives an overall split of: **8000-500-500**.

Preprocessing

Since the raw data is represented by sentences:

id	text	score
196112	A sandwich and french bread sit on a cutting b...	-0.666667
514558	A baseball player is standing on the field whi...	0.000000
54244	A casserole served at a restaurant in a brown…	1.333333

Features have to be extracted from the text data in order to have a better representation and reduce the noise.

Using **nltk** library tools (*word_tokenize*, *stopwords* and *WordNetLemmatizer*), the preprocessing steps considted in:

1. Not using capital letters (lower)
2. Tokenizing the sentence (split in words)
3. Excluding stopwords (“a”, “the”, “and”)
4. Excluding numbers
5. Lemmatizing the word
6. Joining back the tokens for a new sentence

After applying these steps for each raw data set, a new vocabulary is obtained with sizes:

- Training Vocabulary Size: 4107
- Validation Vocabulary Size: 956
- Testing Vocabulary Size: 1291

And the most common words for each set are:

- Training: [('man', 819), ('white', 708), ('two', 671), ('sitting', 536), ('standing', 511)]
- Validation: [('man', 54), ('two', 37), ('standing', 34), ('sitting', 33), ('white', 32), ('woman', 31)]
- Testing: [('two', 40), ('sitting', 40), ('red', 37), ('man', 33), ('large', 30)]

Out of the first 5 words, 3 are common in all lists, which gives us hope that the testing data might be similar to the validation set.

The final step is vectorizing the data. Because I tried multiple approaches (including a neural net), I used 2 methods depending on the approach.

For the neural network choice (using **Tensorflow)**, the data was vectorized using *TextVectorization* which is a layer type from the *keras* API, with:

- max_tokens = 5000
- sequence_length = 100

And this was adapted to the original training data.

However, for the non-neural network attempts, I used the *TfidfVectorizer* from **sklearn**.

TF-IDF or (Term Frequency(TF) — Inverse Dense Frequency(IDF)) is a technique which is used to find meaning of sentences consisting of words and cancels out the incapabilities of Bag of Words technique which is good for text classification.

Won’t delve in the actual formula, but the score for each token is calculated as a product of ***how important is that token for a specific document (sentence)*** and ***how common is the token among all documents (text set of sentences)***. The parameters that gave the most success were:

- max_features = 4000
- ngram_range = (1, 2)

Methods

Neural Network

Many possible architectures were tried (including *convolutional networks*). The one that gave the most success is the following:

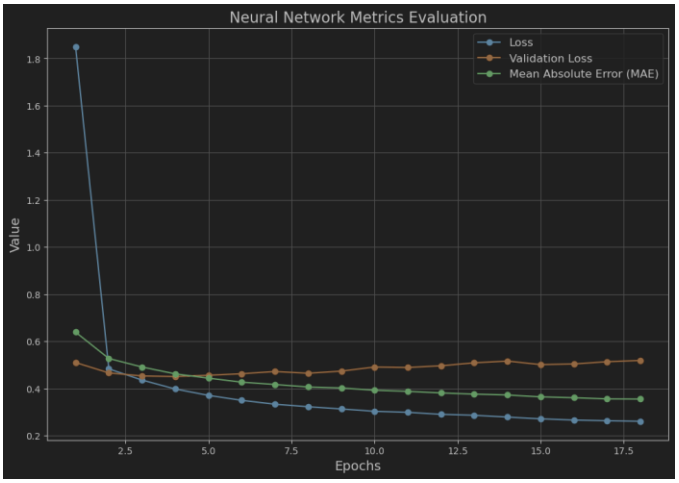
Layer (type)	**Output Shape**	**Param #**
text_vectorization_1 (TextVectorization)	(32, 100)	0
embedding_4 (Embedding)	(32, 100, 128)	2560000
bidirectional_2 (Bidirectional) using LSTM	(32, 100, 256)	263168
global_average_pooling1d (GlobalAveragePooling1D)	(32, 256)	0
dense_3 (Dense)	(32, 64)	16448
dropout_4 (Dropout)	(32, 64)	0
dense_4 (Dense)	(32, 1)	65

The first 2 layers are part of the preprocessing step, in the end obtaining the necessary *Embeddings* (vectors) that better represent the relationship between the words (those with similar “meaning” should be “closer”).

LSTM (Long-Short Term Memory) layer had the hope to extract the dependencies not just between the words next to each other, but from the whole context. And the improvement would be to do this both for past and future tokens, which is why the *Bidirectional* layer was added (hopefully).

After averaging the results in a single vector, the general technique of using a *Dense* layer at the end proved useful (in the hope of generalizing more and reduce overfitting, *Dropout* was introduced).

ADAM was the choice for optimizer and *mse* (mean square error) for the loss function. Because of other past attempts and overfitting, a scheduler for the learning was introduced. I hoped that gradually reducing the learning rate as training progressed would ensure the stability in the loss of validation data (it did, but not by much).



The above results are obtained after 18 epochs with a batch size of 32. As you can see, the loss for validation data slowly converges to 0.5, which isn't good. The evaluation comparison will be done using spearman metric on validation predictions in the Evaluation section.

Random Forest

Decision trees wouldn't help much in a regression problem, but *Random Forest* technique can combine multiple such trees. The implementation from **sklearn** was used (*RandomForestRegressor*). Question would be: how many trees (estimators) should be used? Too few would miss features, but too many may include unnecessary features that can add noise.

For this *GridSearchCV* method from **sklearn** was used. This will try all possible combinations of given hyperparameters and does cross-validation for performance between each combination. For this specific search, the following setup was used:

- RandomForestRegressor as the model
- 'n_estimators': [10, 50, 100, 150, 200, 250]
- 'neg_mean_squared_error' as scoring
- the vectorized training data features for fit

The best parameter was 100. After this the model was fit with the vectorized training data features.

Ridge Regression

Ridge Regression is a linear regression method, but which adds another parameter to the loss function for regularization (transforming into a optimization problem). The hope would be to reduce overfitting especially with a small amount of training samples.

Since the α (alpha) parameter is basically the regularization strength (like a weight coefficient in the sum from predictive function), *GridSearchCV* method was used here as well to establish it, with:

- Ridge from **sklearn** as the model
- 'alpha': [0.001, 0.1, 1.0, 10, 20, 30, 40, 50, 60, 70, 100]
- 'neg_mean_squared_error' as scoring

The best alpha was 10, but further attempts on the vectorized training features, got better results on 6.

Lasso Regression

This is something I have discovered while finding out more things about Ridge Regression. *Lasso Regression* is similar in the sense that it also adds a regularization to the loss function, but this is an L1

regularization, not L2 like in the Ridge regression. Actually, the loss functions aren't as similar as expected, but I need to explore this further.

In the **sklearn** implementation (Lasso), the α (alpha) parameter is just as important as in the Ridge Regression. Thus, *GridSearchCV* method was used here as well to establish it, with the setup:

- Lasso as the model
- 'alpha': [0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0, 10, 20, 30, 40, 50, 60, 70, 100]
- scoring='neg_mean_squared_error'
- the vectorized training data features for fit

Using max_iter = 10000 for the model, the result of the search was 0.0001, but further attempts on the vectorized training data features, got better results with 0.0005.

Support Vector Regression (SVR)

Instead of finding a hyperplane and use it for class separation (SVM), *Support Vector Regression* (SVR) estimates a function that fits the data within a margin of error (tolerance - ε – epsilon). In order to avoid noisy data, slack variables were introduced and we can control this level of regularization using another parameter – C.

For the **sklearn** implementation *SVR, GridSearchCV* method was used to find the best combination, in:

- SVR as the model
- "C": [0.01, 0.1, 1]
- "epsilon": [0.01, 0.1, 1]
- 'neg_mean_squared_error' as scoring
- the vectorized training data features for fit

The best results were: {'C': 0.1, 'epsilon': 0.1}, which were also used in the final spearman evaluation for validation data. A few more tries were made, but nothing better was obtained here.

Combined Attempt

As a curiosity, a weighted averaged of all non-neural attempts was created. The idea was:

- evaluate the spearman score for all 4 models on validation data and get scores
- sum those scores
- get a weight for each model: $\frac{model\_score_i}{total\_score}$
- get the final predictions by a weighted averaged with the above weights and the respective model predictions $\sum_{i=1}^{4} model\_weight_i * model\_prediction_i$

In practice we would need to obtain to the weights as mentioned above (basically on validation data), make predictions on test data using each model, and submit the final prediction as a weighted sum.

This is computationally expensive, yet the purpose of the competition was to obtain the best score on testing data.

Evaluation

As mentioned in the Introduction, the predictions will be compared with the real values using the **Spearman's Rank Correlation**. Locally, I used the implementation from **scipy**: *spearmanr*.

For every method mentioned, predictions were generated on validation data and the spearman score was obtained with those and validation score

Method	Spearman Score
Neural Network	0.5451729578500569
Random Forest	0.5858997956891087
Ridge Regression	0.6097673865466267
Lasso Regression	0.5707103375985584
Support Vector Regression	0.59971866035178
Combined Attempt	0.6093642375972341

It seems that the best methods (at least for validation data, seem to be the **Ridge Regression** and **Combined Attempt**. What matters are the test predictions, but this will serve as baseline.

4. Conclusion

Test submission results

During the Kaggle competition, I had 30 submission coming from all methods mentioned in the above section.

Before showing some scores from the submissions, I have some observations about the best methods:

Ridge Regression:

From 500 samples: Mean: 0.55271, SD: 0.375691

Combined Attempt:

From 500 samples: Mean: 0.58089, SD: 0.390392

This is quite significant, given the similar scores on validation data and the small number of samples.

Now, the submission scores based on randomized 20% of the test data (before the final day):

Method	Spearman Score
Neural Network	0.44847
Random Forest	0.46124
Ridge Regression	0.55080
Lasso Regression	N/A
Support Vector Regression	0.52788
Combined Attempt	0.50688

I showcased the best score from each method. Out of the 30 submission, many predictions were done with older and inefficient models (especially using NNs). For the final submissions, I used 2 **Ridge** generated samples which scored each 0.55080 and 0.54307.

After final submission day, when all data was evaluated from those 2 attempts, the final score was 0.53938, placing on 52/149 competitors.

Possible Improvements

First of all, I shouldn't have spent so much time on neural networks. Even with smaller count of nodes, kernels, etc. the scores have been worse than other methods. One of the key reasons for this is the small amount of data used in training. Just 8000 samples haven't been enough and should have been a sign to use other method earlier.

Other methods like data augmentation could have been used in training (mixing the words in a sentence and just use 1-gram to eliminate context), in order to generate more training data.

Otherwise, I feel that any improvements would go around how to process and represent the data.

Code implementations

Final code file will be attached to this document, including other attempts, not just the best model.

5. Bilbiography

1. Maqsood, Shazia & Shahid, Abdul & Afzal, Muhammad & Roman, Muhammad & Khan, Zahid & Nawaz, Zubair & Haris, Muhammad. (2022). Assessing English language sentences readability using machine learning models. PeerJ Computer Science. 7. e818. 10.7717/peerj-cs.818.

2. Agnes Malatinszky, Aron Heintz, asiegel, Heather Harris, JS Choi, Maggie, Phil Culliton, and Scott Crossley. CommonLit Readability Prize. https://kaggle.com/competitions/commonlitreadabilityprize, 2021. Kaggle.

3. Madan, Rohit. "TF-IDF/Term Frequency Technique: Easiest Explanation for Text Classification in NLP Using Python (Chatbot Training on Words)." *Analytics Vidhya*, 30 May 2019, https://medium.com/analytics-vidhya/tf-idf-term-frequency-technique-easiest-explanation-for-text-classification-in-nlp-with-code-8ca3912e58c3.

4. Jamieson, Andrew. "Convolutional Neural Networks (CNN) with Text." *Medium*, 19 Oct. 2020, https://asjamieson.medium.com/convolutional-neural-networks-cnn-with-text-f1b438be5466.

5. Olah, Christopher. "Understanding LSTM Networks." *Colah's blog*, 27 Aug. 2015, https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

6. Ionescu, R. (2024–2025). *Practical Machine Learning*. Faculty of Mathematics and Computer Science, University of Bucharest., All courses, https://practical-ml-fmi.github.io/ML/