CommonLit Readability

Aditya Kallepalli Xing Yang Lan Quilvio Hernandez

Aditya Kallepalli Contribution:

- Brainstormed Ideas
- Organized Team Meetings
- Wrote Proposal
- NLP Research
- EDA (Wordcloud)
- Data Pre-Processing
- Data Cleaning
- Proofread Report

Xing Lang Lan Contribution:

- Brainstormed Ideas
- Wrote Proposal
- NLP Research
- Linear Models (Basic, CountVectorizer)

Quilvio Hernandez Contribution:

- Brainstormed Ideas
- Proposed Submitted Project Idea
- Organized Team Meetings
- Wrote Proposal
- NLP Research
- EDA (Everything with grey background)
- Data Cleaning
- Linear Models (Pipelines, TF-IDF, GridSearch)
- Tensorflow Research
- NN Models
- Wrote Report

Introduction

Currently, most educational texts can be matched to readers using commercially available readability methods or traditional formulas. However, these methods tend to have some issues when it comes to generating meaningful results. Namely, they lack construct and theoretical validity. CommonLit and Georgia State University are challenging Kagglers to develop algorithms that can improve the complexity rating of reading passages. We decided to center our final project around this research question and develop a model to predict the reading level of a passage of text.

We were able to get an RMSE value as low as 0.71234 using Ridge Regression and TF-IDF vectorization. Lasso performed poorly for the problem. Linear regression was surprisingly strong on the raw text, but the score fell off when using the cleaned data. No word embedding model (NN, CNN) we checked was able to best the results from our Ridge Regression. All of our models had an overfitting problem.

Methodology

Kaggle has provided training and test datasets. The training data set has 2834 unique excerpts with its reading ease score. The data can be found here.

The `target` variable is determined by aggregating scores among multiple raters for each excerpt. The `target` variable is a numerical value that ranges between -3.676 and 1.711. The scores follow an approximately normal distribution. The original size of the number of words used across all documents is 51,038. After preprocessing the data, including stemming, removing stopwords, and normalizing whitespace, this number is reduced down to 17233. The `standard error` variable is a measure of the spread among the raters. There are no missing values for either of these variables. Our performance metric is RMSE as outlined in the competition rules.

We only use the `except` variable to train our model. The `except` variable is a collection of passages from different time periods and reading ease scores. Modern texts are slightly overrepresented in the test set compared to the training set because these are the type of texts we want to generalize. Passage lengths range from 135 to 205 words. There are 51,038 words present in the dataset to train our model with; however, this drops down to 17,233 after preprocessing our data.

We used vectorized our data using three different methods: TF-IDF, Count, Tokenizer
TF-IDF determines word relevancy in the specific document compared to a collection of documents, or in our case – excerpts. The formula for it is essentially how often a word is used in the specific text divided by how often it is used across the entire corpus. Common words such as

“the”, “and”, and the like would have low scores, while unique words such as “world” and “war” would carry more weight.

A count vectorizer creates a matrix of “token” counts. Every word is treated as a “token”, so the array is a collection of words and their frequency across all passages. This matrix is very sparse because most words don’t get used in every excerpt.

The Keras Tokenizer vectorizes a corpus by mapping each token to an integer in a dictionary. This converts the text passages into a sequence of integers. There is an additional argument to deal with out of vocabulary tokens, that is words that aren’t present in the dictionary.

We tested our model on the following different linear regression models: Linear, Lasso, Ridge

Linear Regression finds the coefficients that minimize the residual sum of squares between the data points and the target variable. In our case, that would be assigning a certain weight to each token and taking the sum of those token weights to determine the reading ease score.

Lasso is a linear model with L1 regularization. This means it adds a penalty equal to the absolute value of the magnitude of coefficients with a tuning parameter alpha. Alpha is linearly proportional to bias, and inversely proportional variance. It is well-suited for models with high levels of muticollation.

Ridge is a linear model with L2 regularization. This is well-suited for problems where the number of variables exceeds the number of observations, a weakness of least squares regression. It also has a tuning parameter that controls the strength of the penalty.

We tested two neural networks architectures: neural network with 2 hidden layers and neural network with 2 hidden layers and one convolution layer.

The typical hidden layer is “dense” meaning every neuron in the layer is connected to every neuron in the previous layer. The dense layer is also performing matrix-vector multiplication to compute the values of the parameters that can be used to train and update in the backpropagation process. The output of the dense layer is of dimension n, so it can be used to change the dimension of the vector. The convolution layer solves the problem of dense layers in that dense layers can have too many parameters and lead to poor performance. Instead, it only maps to a sub-region of the previous layer.

Implementation Details

We preprocessed the excerpts by unpacking contractions, normalizing whitespace, converting text to lowercase, removing punctuation and stopwords, and stemming the words down to their

roots. After preprocessing we were able to vectorize them using TF-IDF, CountVectorizer, and Keras Tokenizer. We did not experiment much with the count vectorizer because TF-IDF outperformed it with every model, so we will disregard it for this section.

The data was split into 10 stratified folds, each containing 284 observations. 9 of these were used to train the model, while the outstanding fold was used for validation.

TF-IDF was implemented such that each word was a feature (unigrams). Rather than the traditional formula, we used a binary TF-IDF where all nonzero terms are set to 1. This made it easier to train our models. We also opted for the L-1 unit norm after performing a grid search and finding it to be the optimal parameter.

Using Keras Tokenizer, we created a dictionary of all the remaining words after preprocessing the data (approximately 17,000 words). We denoted any word that wasn't in our dictionary with a "<OOV>" (out of vocabulary) token. After converting the excerpts to sequences, we padded them to make all passages the same length. This is important when working with word embedding models. We decided to make them all 130 tokens long because no passage was over 130 words after preprocessing.

There were no parameters to play with for Linear regression and Lasso performed poorly, so we decided to focus on Ridge regression with the cleaned text. We executed a grid search to find the optimal hyperparameters and found the RMSE is minimized using 1000 iterations and an alpha of 0.01. The tested alpha values were .000001, .00001, .0001, .001, .005, .01, .05, .1, .5, 1, 10. The tested iteration values were 1000 and 100000. We also decided to ensure the model fit the intercept and normalized the regressors before training the model.

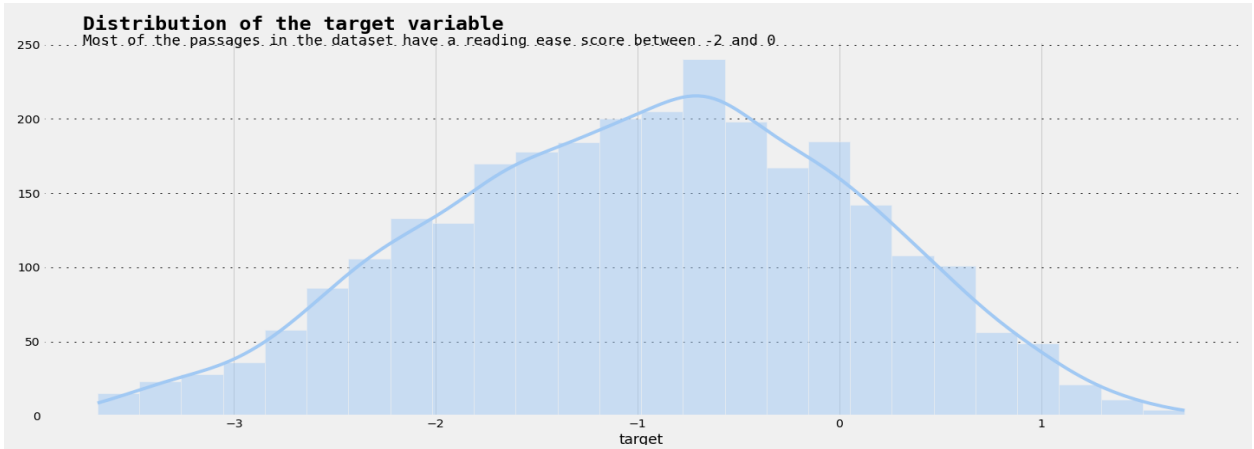
The embedding layer of our neural network had the same dimension as our dictionary (17233) and output dimension equal to the ceiling of the fourth root of the dictionary size (12). We chose this output dimension because in our research we found it was best practice for NLP problems like ours. The second layer was a global average pooling operation to average over the sequence dimension (130). This is a simple way for the model to handle input of variable length and output a fixed-length vector equal to the embedding dimension (12). After using a Hyperband tuner, we found the optimal number of units for the first dense layer to be 160. The second dense layer gives the output and has one unit. In our CNN model, the convolution layer comes before the global pooling layer with output size equal to the embedding dimensions (12) and kernel size of 3. All layers, where applicable, use the ReLU activation function and L-2 regularization with an alpha parameter of 0.01. We used Adam as our optimizer with a learning rate of 0.001.

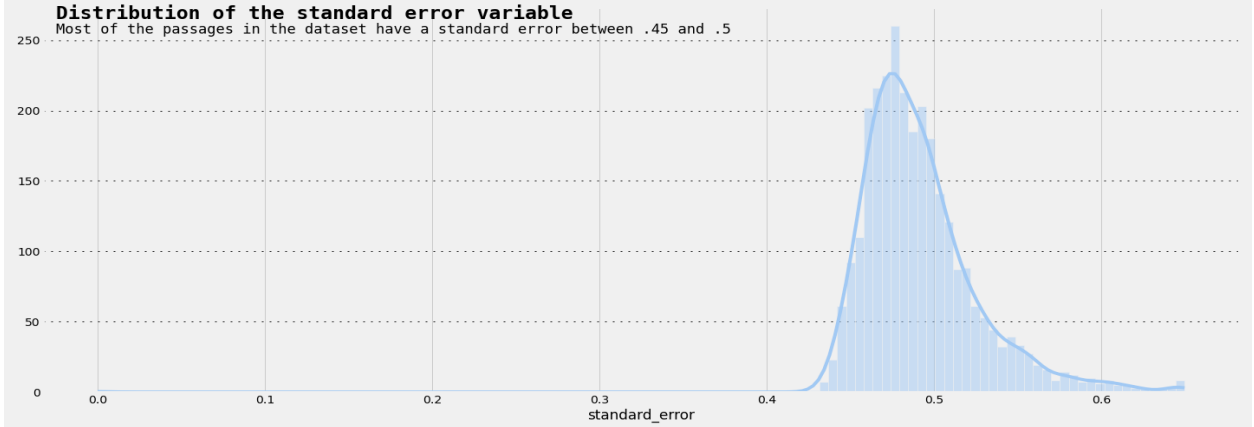
Results and Interpretation

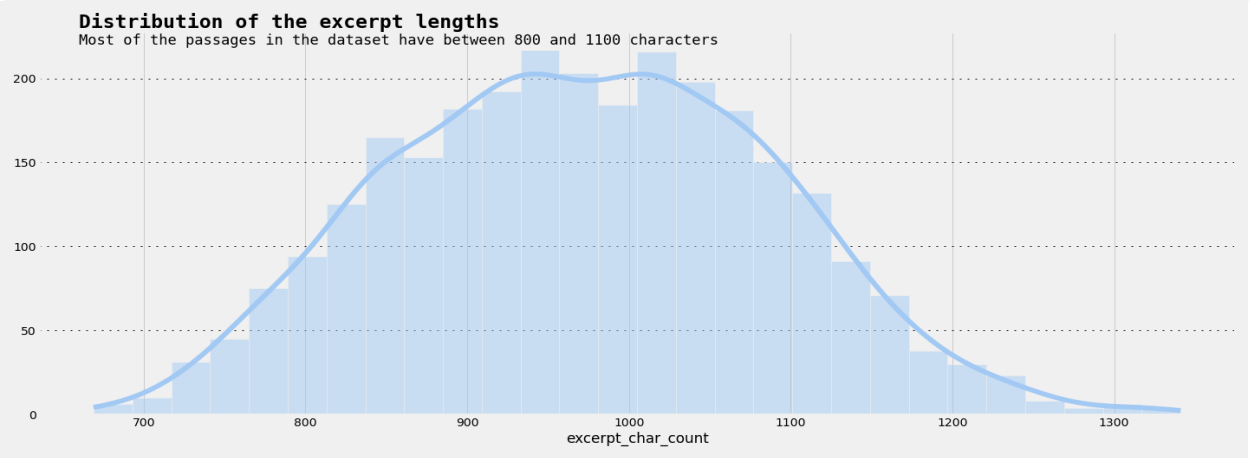
We experimented with Ridge, Lasso, and Linear regression on the raw and cleaned vectorized data. Ridge regression performed better on the cleaned data with an RMSE value of 0.71234 compared to 0.73107 on the raw data. Linear regression performed better on the uncleaned vectorized data with an RMSE value of 0.76888 and 0.78198 on the cleaned data. We believe this is because stopwords were more significant in the least squares regression case. Lasso performed poorly compared to our other models with an RMSE of 1.03147. We tested the CountVectorizer with our optimized Ridge regression and got an RMSE of 0.84287 so we did not test it with the other models.

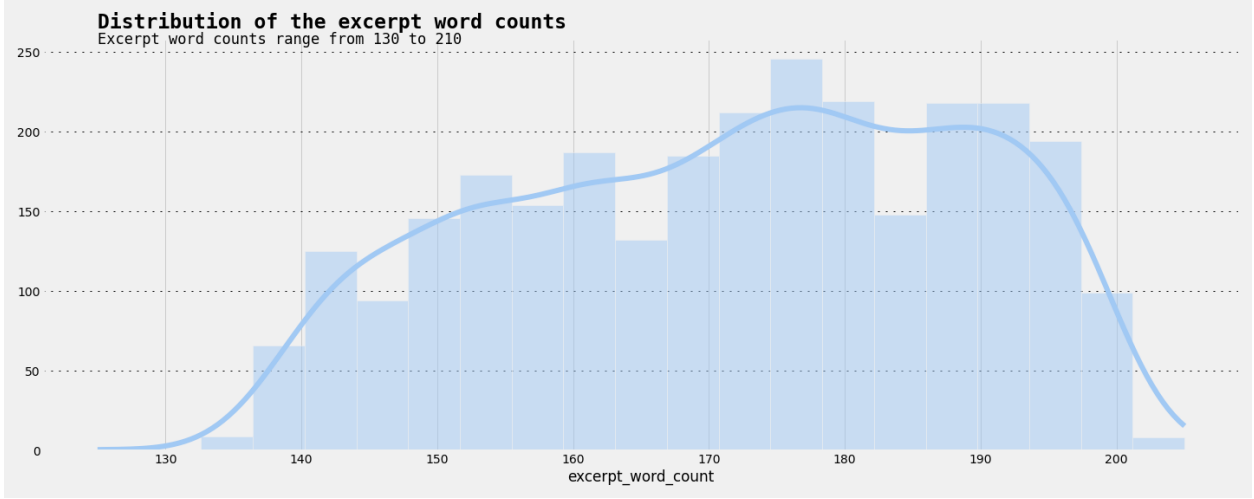
All the neural networks had an overfitting problem. They had very low training RMSE, but validation RMSE in line with the linear and ridge regression techniques. After tuning the hyperparamters, we got a validation RMSE of 0.7686. Our CNN model had a validation RMSE of 0.8401. However, the train RMSE of these models were 0.4008 and 0.1847, respectively.

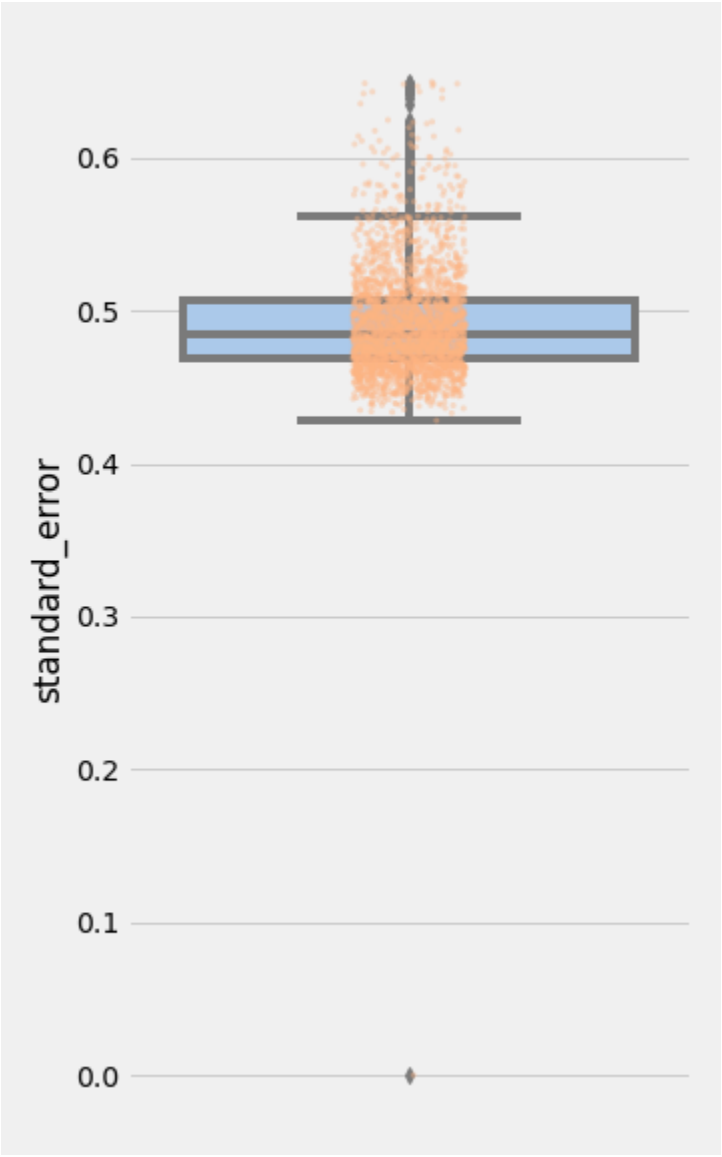
In doing this project we got a better grasp of wording with text data and applying NLP techniques. This included researching different NLP packages such as nltk and nlpretext. nltk is one of the most popular libraries but the pipeline and simplicity of nlpretext was perfect for our project. The difference in regression scores between the preprocessed data and the raw data showed us the importance of lemmatization, stemming, and cleaning text data in general. We were also able to familiarize ourselves with the Tensorflow and Keras framework. In class, we learned about PyTorch; however, we opted to go with Tensorflow because of it's beginner friendly framework. Tensorflow contains many high-level wrapper classes, as part of Keras. The ability to quickly add layers to a model with one line of code made it easy for us to implement a neural network without being too familiar with OOP. It also helped us better understand when to apply different regression techniques. Once we understood how Ridge regression worked, it became clear why it was performing far better than the other regression techniques tested. We feel confident we could apply these techniques to a similar problem.

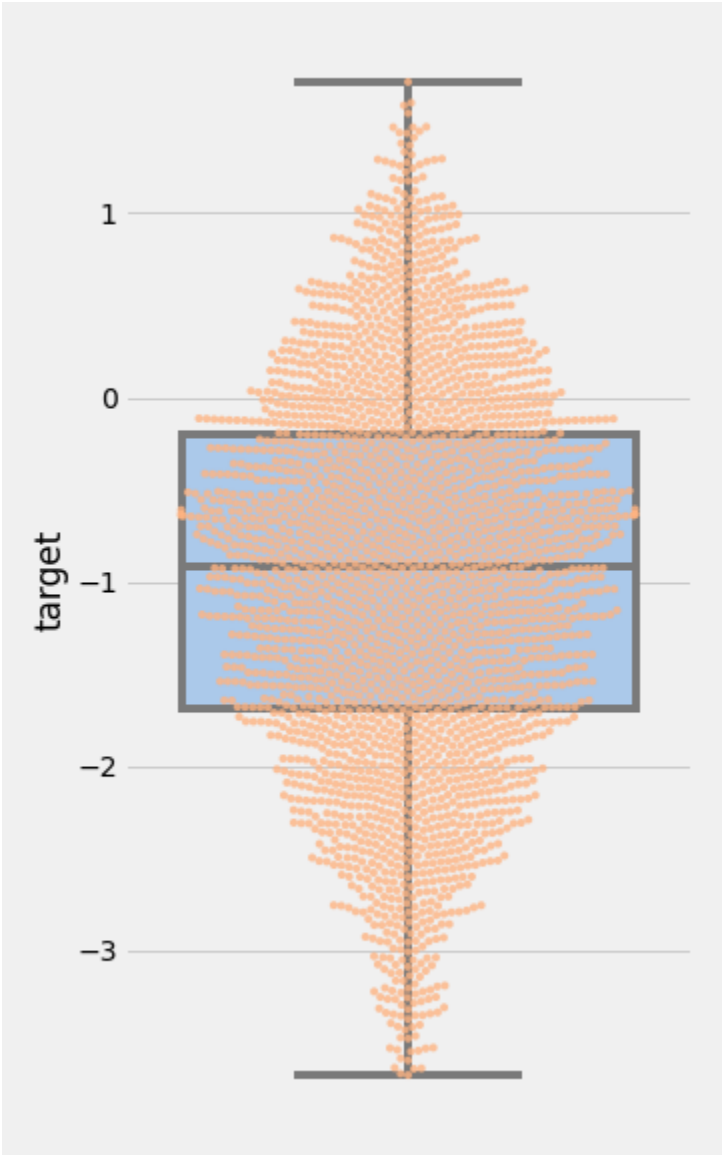

Distribution of the target variable
Most of the passages in the dataset have a reading ease score between -2 and 0
250
200
150
100
50
0
−3
−2
−1
0
1
target

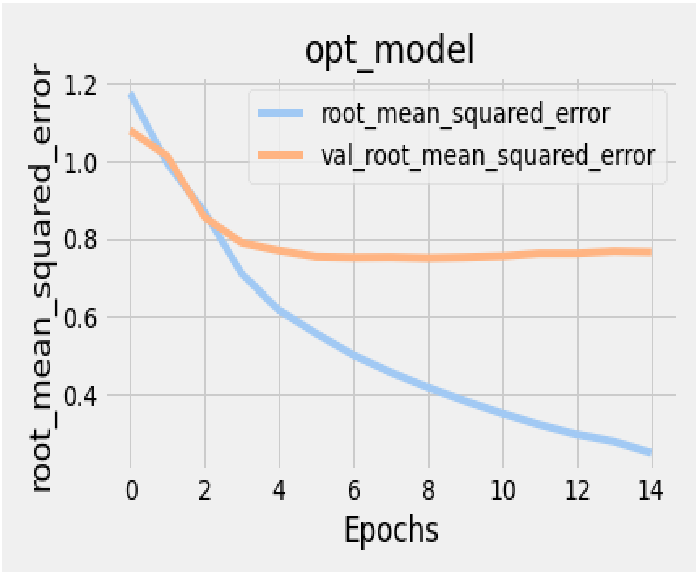

Distribution of the standard error variable
Most of the passages in the dataset have a standard error between .45 and .5
250
200
150
100
50
0
0.0
0.1
0.2
0.3
0.4
0.5
0.6
standard_error

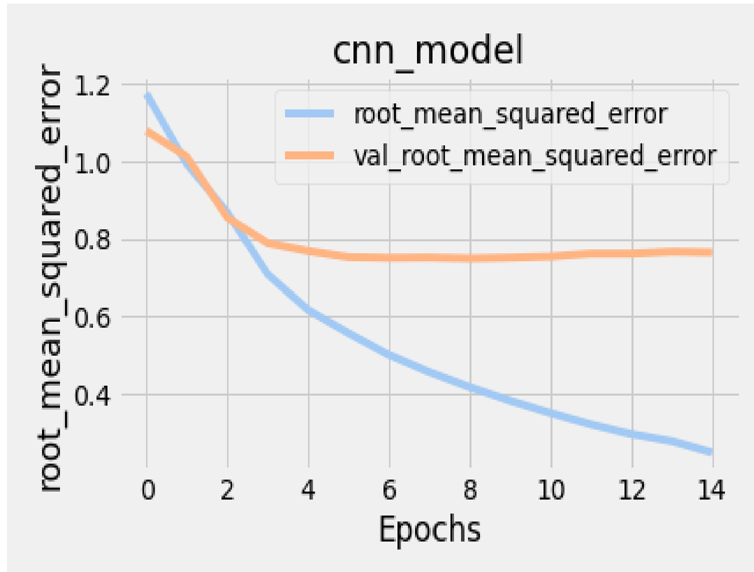

Distribution of the excerpt lengths
Most of the passages in the dataset have between 800 and 1100 characters
200
150
100
50
0
700
800
900
1000
1100
1200
1300
excerpt_char_count

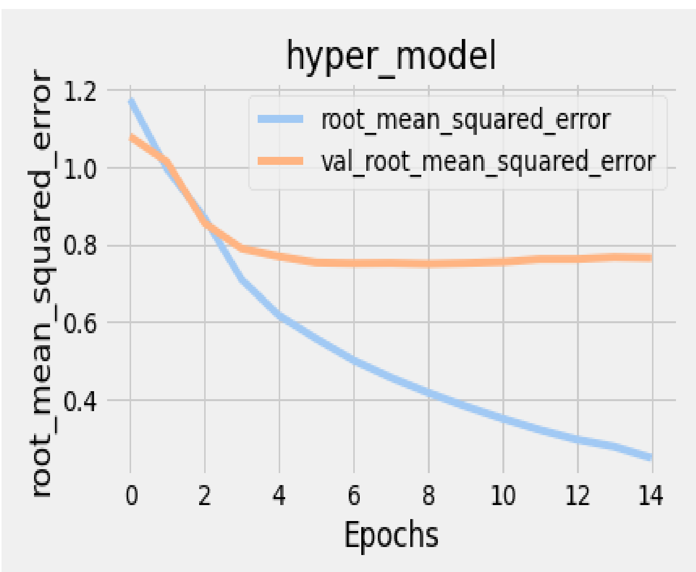

Distribution of the excerpt word counts
Excerpt word counts range from 130 to 210
250
200
150
100
50
0
130
140
150
160
170
180
190
200
excerpt_word_count

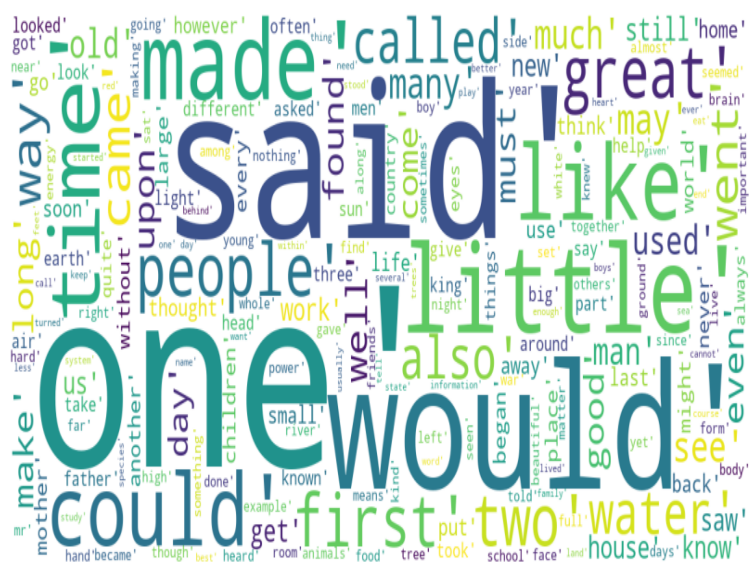

standard_error
0.6
0.5
0.4
0.3
0.2
0.1
0.0

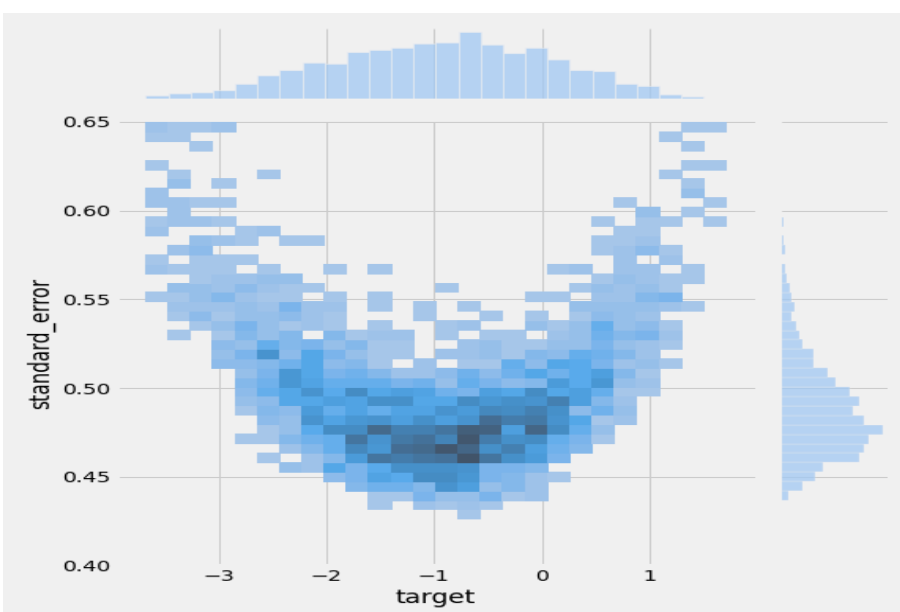

target
1
0
−1
−2
−3


opt_model
root_mean_squared_error
val_root_mean_squared_error
root_mean_squared_error
Epochs


cnn_model
root_mean_squared_error
val_root_mean_squared_error
root_mean_squared_error
Epochs


hyper_model
root_mean_squared_error
val_root_mean_squared_error
root_mean_squared_error
Epochs


said'
one'
would'
little'
made'
like'
time'
could'
people'
called'
great'
first'
two'
water'
also'
came'
went'
many'
much'
still'
way'
day'
even'
good'
see'
must'
well'
found'
upon'
make'
long'
old'
may'
used'
made'


standard_error
target