Practical Machine Learning, Project 1

Word Complexity Estimation

1. Feature Extraction

For feature extraction, I processed the sentences by transforming them to lowercase and getting rid of punctuation, then separating the sentences to lists of words. I chose 2 methods of feature extraction – the classic bag of words and tf-idf (the comparison will be discussed further).

I tried to experiment with both word stemming and tokenization for the bag of words features, which brought negative results on the test data compared to the usual bag of words.

I initially tested the different methods of feature extraction with MLPRegressor from the sklearn library, with SGD optimizer, 200 epochs and hidden layer size of (100,), and ReLU

In this documentation, I will refer to the private Kaggle scores, estimated on 25% of the data, as it was the only score I could take into consideration while working on the project.

The MLPRegressor, with the usual bag of words features, gave me a score of 0.05839 MSE. The same model, with stemming, gave me a score of 0.089 MSE, and with word tokenization – 0.06088 MSE. I have theorized upon the bad results of such word extraction practices for the word complexity estimation: stemming removes important word parts that are necessary for the experiment, like the '-ing' from a verb or the '-s' from the plural of a noun, and word tokenization doesn't separate compound words united through a dash, like 'long-term'; if 'long-term' is recognized as a single word, there is a high chance that its appearance in the data will be sparse, and the model will not be able to apply its learned complexity from the train data to the test data, or will not be able to predict its complexity in case it appeared only in the test data and not in the train data.

TF-IDF vs Bag-of-Words:

TF-IDF proved to give better results (0.05102 MSE), with the same MLPRegressor model with same parameters as above. I theorize that it is mostly because of TF, as it is equal to 1 divided to the number of words in the phrase. In most cases, the complexity grows parallel to the number of words in the phrase. Let's take 2 examples from the train data: 'coral', with 0.0 complexity, and 'coral outcrops', with 0.1 complexity. Of course, it is not entirely accurate for all examples. I think IDF didn't make a big change in the results. It focuses more on the relevance of a word to the rest of the text, by counting the number of appearances of it in the corpus of documents. While I believe it is a very useful practice in most NLP experiments, I think that the word complexity estimation is an exception. Whether it appears frequently or not in other sentences, most likely, doesn't play a role to the complexity perceived by the annotators.

2. How did I treat the problem of predicting the complexity?

I decided to predict the sum of the annotators who found the phrase to be difficult. Afterwards, in order to predict the complexity, I divided it to the sum of total number of annotators.

3. Picking the model and hyperparameters

I tried 2 sklearn models: neural_network.MLPRegressor and svm.SVR. In order to find the right combinations of hyperparameters, I decided to use sklearn.model_selection.validation_curve. I also chose a 5-fold cross-validation.

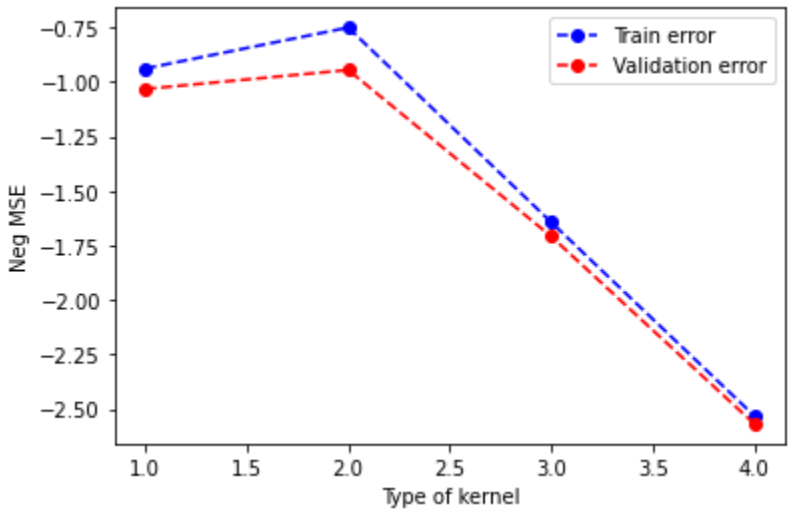
Validation_curve helped with hyperparameter tuning, by running the models with variating values for each hyperparameter, and giving me the train and validation negative mean squared error scores in a 5-fold cross-validation. Before plotting the results, I have calculated the mean of each 5 cv values.

Let's start with **SVR**. For most hyperparameters, I will test the model on 250 iterations max, for a faster convergence.

a. Kernel:

```
train_scores, valid_scores = validation_curve(estimator=svm.SVR(cache_size=50, verbose=True, max_iter=250),
                                              X=tf_idf_scaled, y=selected_annotators_scaled,
                                              scoring = 'neg_mean_squared_error', error_score="raise", param_name="kernel",
                                              param_range=['linear', 'poly', 'rbf', 'sigmoid'], cv=5)
```

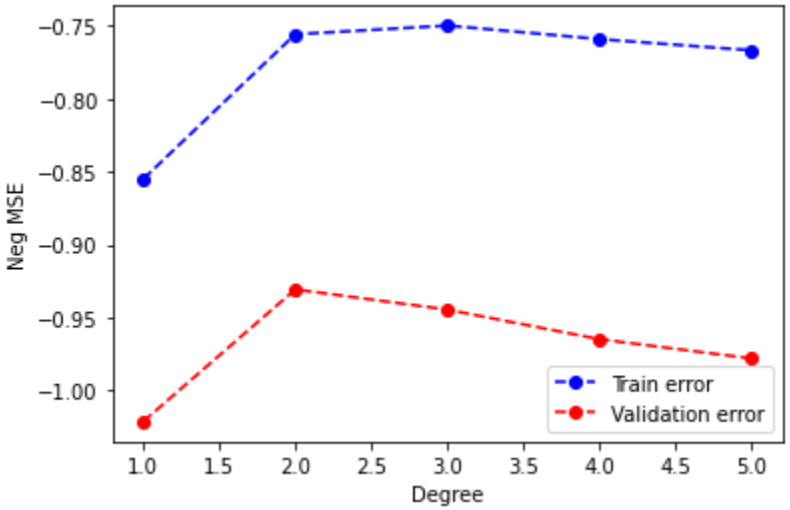
```
X = [1, 2, 3, 4]
plot_figure(train_scores_mean, valid_scores_mean, X, title=None, legend=['Train error', 'Validation error'], labels=['Type of kernel', 'Neg MSE'])
```



It seems like 'poly' kernel gave best results. Even though with poly, it seems to overfit most (the difference between the train and validation scores is bigger compared to the rest kernels, with the validation error being higher), however, I care more about the fact that 'poly' kernel gave smaller MSE error than the rest specifically on validation.

b. Degree (of the polynomial kernel function, ‘poly’). Default is 3. Wikipedia says that the most common degree for NLP practices is 2, since larger degrees tend to overfit. I tried a variation of 1-5. Let’s test it out:

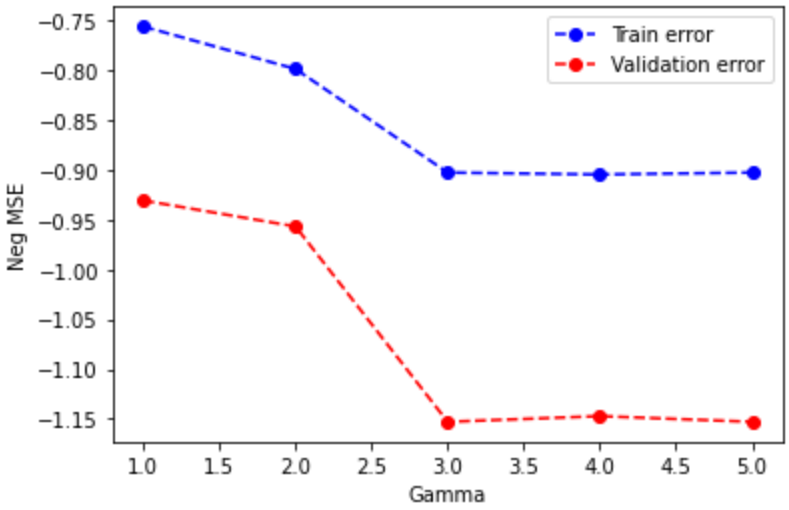
```
train_scores, valid_scores = validation_curve(estimator=svm.SVR(cache_size=50, verbose=True, max_iter=250, kernel='poly'),
                                              X=tf_idf_scaled, y=selected_annotators_scaled,
                                              scoring = 'neg_mean_squared_error', error_score="raise", param_name="degree",
                                              param_range=[1, 2, 3, 4, 5], cv=5)
```



Just as expected, degree 2 worked best on validation (even though degree 3 worked best on train, it just shows that it overfits on degree 3, because it’s not the best option for validation).

c. Gamma (kernel coefficient). Could be ‘scale’ or ‘auto’ for 2 pre-defined formulas of calculating gamma knowing the number of features, so I tried giving more values: 0.1, 0.5, 0.05:

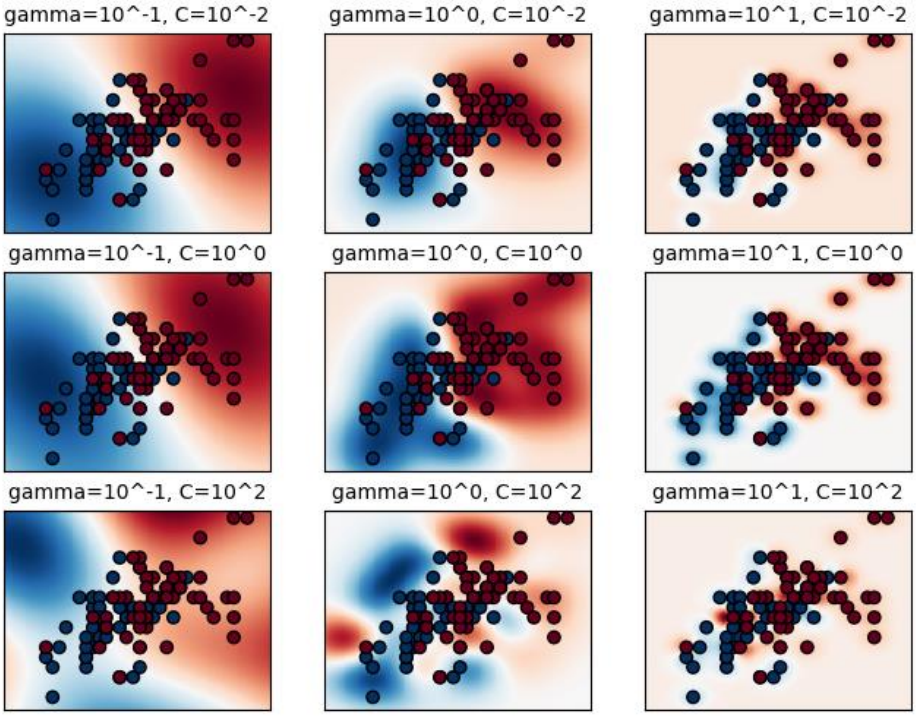
```
train_scores, valid_scores = validation_curve(estimator=svm.SVR(cache_size=50, verbose=True, max_iter=250, kernel='poly', degree=2),
                                              X=tf_idf_scaled, y=selected_annotators_scaled,
                                              scoring = 'neg_mean_squared_error', error_score="raise", param_name="gamma",
                                              param_range=['scale', 'auto', 0.1, 0.5, 0.05], cv=5)
```



It seems like gamma did best on 'scale', which is actually '1 / (n_features * X.var()) '.

d. C: it is a regularization parameter, which allows the model to not separate the data completely if it is not possible, or if separating the data completely does not give best results on test (in case there is a data-point which is an outlier compared to the rest, separating it correctly to the data will not give the best result).

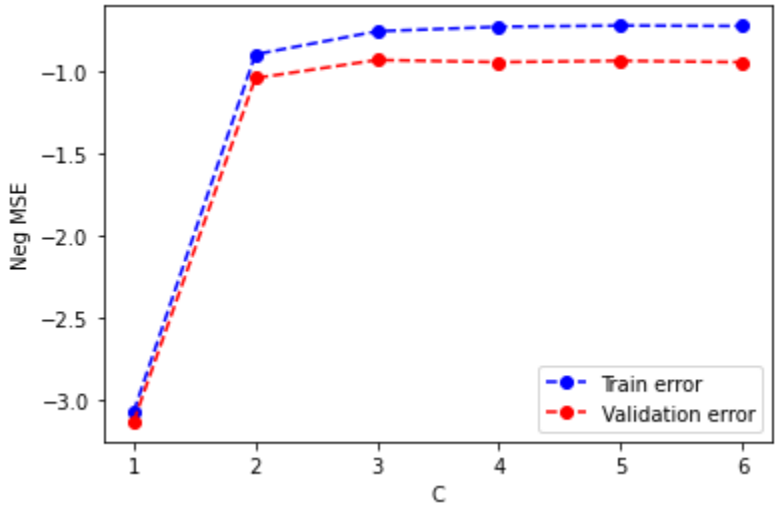
I am referencing this article on sklearn, which talks about choosing the best values for C and Gamma as to avoid overfitting and underfitting:



The 2nd row, 2nd column shows best results. The first column has more examples of underfitting, the last column – overfitting, as it can only predict the red and blue points, and not what color would be any point around them (theoretically, coming from the test data).

It seems like 10^-2 and 10^2 are too extreme for C. I chose a variation of [0.1, 0.5, 1, 5, 7, 10].

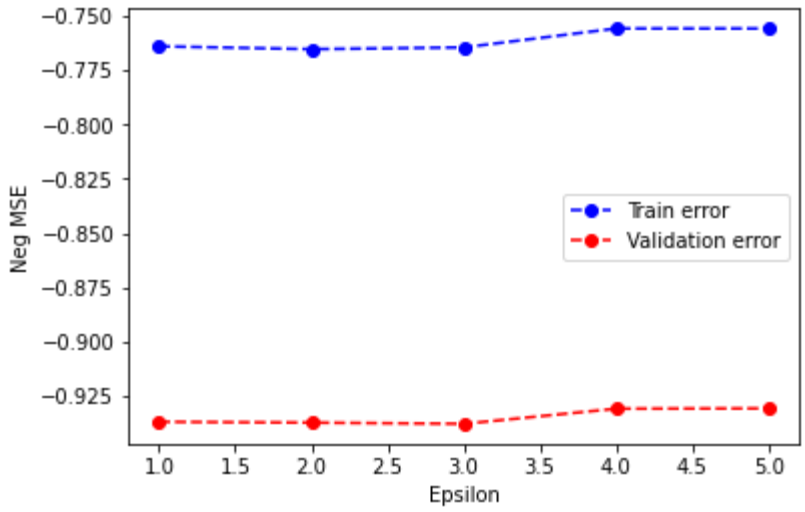
```
train_scores, valid_scores = validation_curve(estimator=svm.SVR(cache_size=50, verbose=True, max_iter=250, kernel='poly', gamma='scale',
                                                                degree=2),
                                              X=tf_idf_scaled, y=selected_annotato (parameter) error_score: Literal['raise']
                                              scoring = 'neg_mean_squared_error', error_score="raise", param_name="C",
                                              param_range=[0.1, 0.5, 1, 5, 7, 10], cv=5)
```



On validation, it did best on C=1.

e. Epsilon, default is 0.1. Similarly to C, it allows an error within which there will be no penalty. But it works differently. If the error on train/validation is smaller or equal to epsilon, the model receives no penalty. In some cases, an epsilon of 0 might not let the model converge, as it allows no error, or it could lead to overfitting. Not allowing an error would mean, as I think, to learn the training data too much. I gave it values in a variation of [10^-6, 10^-3, 10^-2, 0.1, 0].

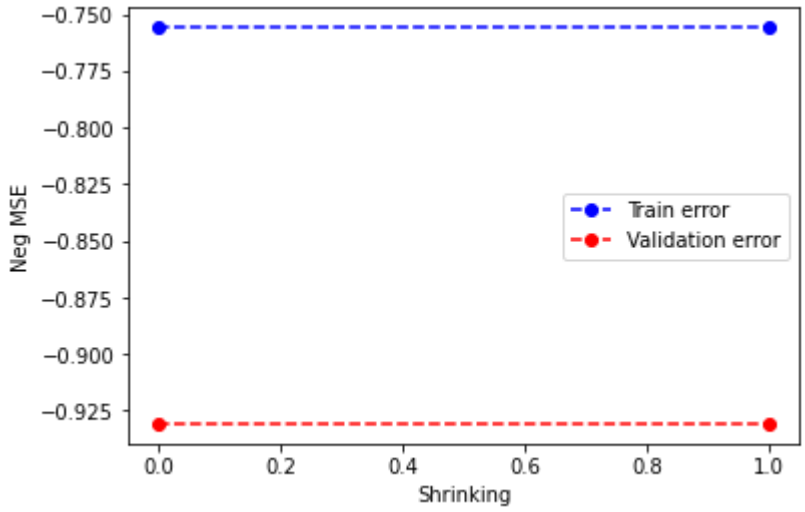
```
train_scores, valid_scores = validation_curve(estimator=svm.SVR(cache_size=50, verbose=True, max_iter=250, kernel='poly', gamma='scale',
                                                                degree=2, C=1),
                                              X=tf_idf_scaled, y=selected_annotators_scaled,
                                              scoring = 'neg_mean_squared_error', error_score="raise", param_name="epsilon",
                                              param_range=[10**-6, 10**-3, 10**-2, 0.1, 0], cv=5)
```



It did best on 0.1.

f. Shrinking: can be True or False. Shrinking is a heuristic that helps shortening the training time.

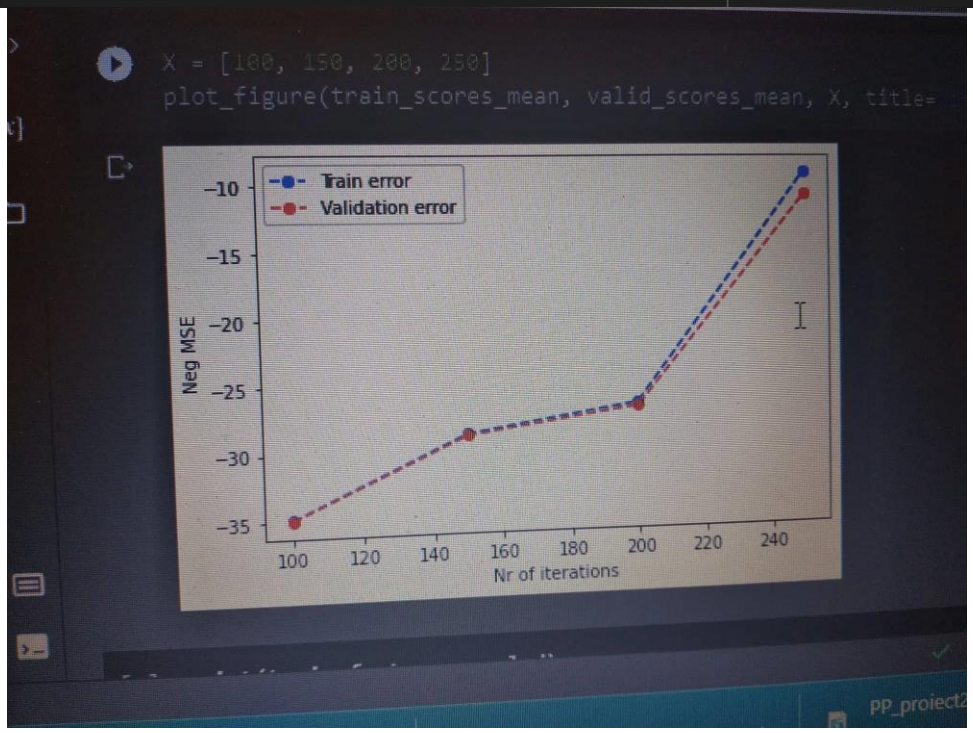
```
train_scores, valid_scores = validation_curve(estimator=svm.SVR(cache_size=50, verbose=True, max_iter=250, kernel='poly', gamma='scale',
                                                                degree=2, C=1, epsilon=0.1),
                                              X=tf_idf_scaled, y=selected_annotators_scaled,
                                              scoring = 'neg_mean_squared_error', error_score="raise", param_name="shrinking",
                                              param_range=[False, True], cv=5)
```



It didn't seem to make a difference. I assume it doesn't change the result the model gives, and can only change the time of the execution.

g. max_iter: Number of iterations at which the model needs to stop if it didn't manage to converge yet. Default is –1, which means the model stops when converges, and max_iter is infinite.

```
train_scores, valid_scores = validation_curve(estimator=svm.SVR(),
                                              X=tf_idf_scaled, y=selected_annotators_scaled,
                                              scoring = 'neg_mean_squared_error', error_score="raise", param_name="max_iter",
                                              param_range=[100, 150, 200, -1], cv=5)
```



It works best with an infinite number of iterations, until it converges.

The model, with kernel='poly', gamma='scale', degree=2, C=1, epsilon=0.1, and max_iter=-1 gave me a score of 0.06632 MSE. Still not as good as the initial MLPRegressor.

MLPRegressor:

a. solver/Optimizer:

I chose to work with SGD, because it gave me better results in the past. I still experimented with Adam, and got 0.06179 MSE, with ReLU as my activation function, 200 maximum iterations, hidden layer size of (100,). SGD gave a score of 0.05839 MSE.

b. max_iter and hidden_layer_sizes, with 200 and (100,) as default:

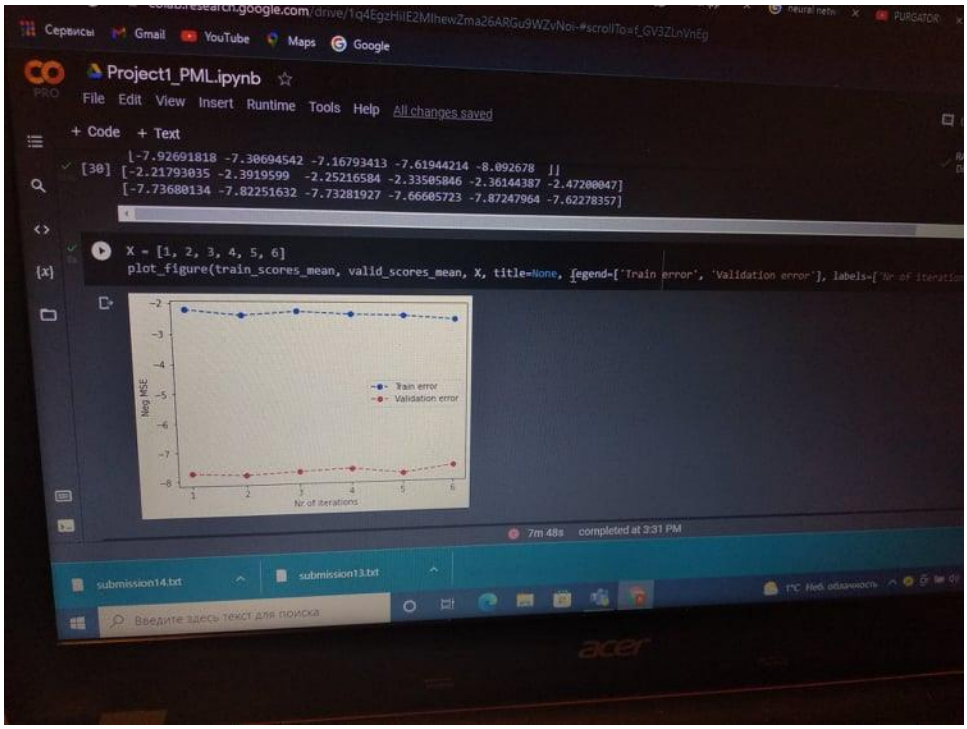
The reason why I chose both parameters is because they are more dependent on each other than the rest. For example, a certain number of iterations works well for a specific level of network complexity, and then as you change the hidden_layer_sizes, the same number of iterations/epochs might overfit or underfit.

Using the validation curve would be hard in this case, because you can only focus on one parameter at a time. However, I still tried a few variations and tested them by submitting on Kaggle:

- Changing the hidden layer size to (50, 30), something more complex, and leaving max_iter to 200 gave me the result of 0.12086 MSE, which is not too good. It made me believe the model was too complex.
- Simplifying the model a bit, and changing hidden_layer_sizes to (10, 20), but max_iter to 300 epochs as to avoid underfitting gave me a slightly better result, 0.06201 MSE.
- I decided to try and leave max_iter=300, but change the network, to (100, 3). I got a score of 0.08038 MSE.
- I tried going the other way, and lessened the number of epochs to 100, and used the default network of hidden (100,). The score was 0.06179 MSE.

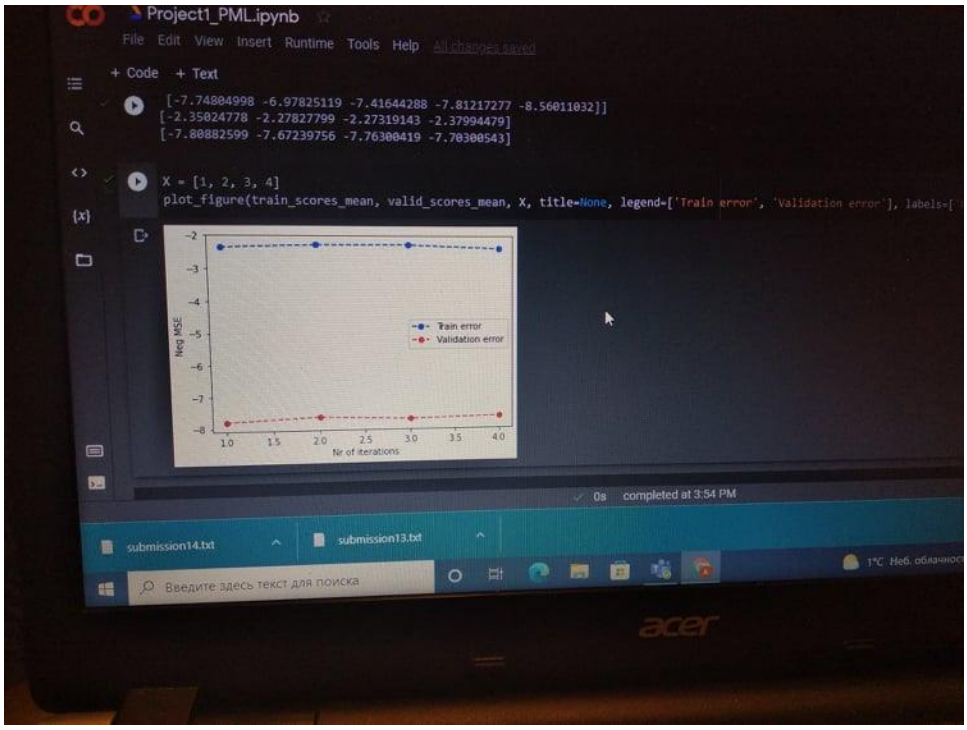
My last attempt was to stop at a more simplified network, of hidden layer size (10, 10). As I understand, it is fully connected, so the number of computations between the hidden layer is 10*10, similar to the default hidden layer size, which gave the best result so far. I used validation curve in order to play around with max_iter.

Firstly, I tried giving a smaller number of iterations, in a variation of [50, 100, 150, 200, 250, 300].



On the validation set, it was averaging at 8 MSE, and on the training set, at 2 (the plot is for Negative MSE). It looks a lot like overfitting to me, as the difference is about 6. The plots I made for parameter tuning for SVR showed a smaller difference, of 0.25 towards the end of experiment.

I still decided to see what I can get with a bigger number of iterations, so I tried [300, 500, 800, 1000].



I got a similar result, which made me believe the network was too complex.

I decided to stop at max_iter=200 and hidden_layer_sizes=(100,), the defaults. The best result I got was 0.05102. After the end competition ended, the final result was 0.06061, which was still the best compared to the other submissions. All of the submissions proved to give a worse score on the 75% of the data.

4. Picking the activation function:

Because I wanted to predict the sum of the number of annotators who found the sentence difficult, this number could be from 0 to total number of annotators. ReLU is the closest activation function to our predictions. I decided to use it for all layers.