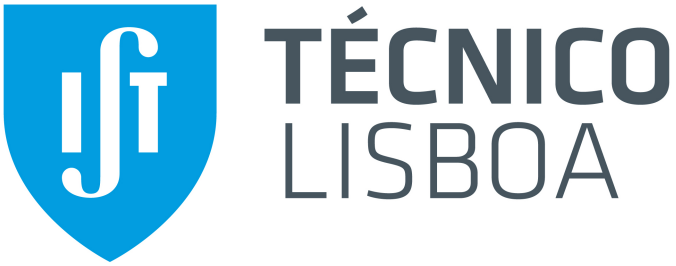

Instituto Superior Técnico

MEEC

Machine Learning

Machine Learning Project

Work done by:	*Number:*
Luís Sousa	90130
Rodrigo Contreiras	90183

Group 4

16 of October 2022

Contents

Introduction

The first part of the project comprises of a linear regression problem, which is to be solved using a Machine learning Algorithm of our choice. Furthermore, it is divided into two deliverables.

The first deliverable consists of training a basic linear predictor using the provided training set, with no outliers, using the sum of squared errors (SSE) as a performance metric. An independent set of data is applied to the calculated predictor to complete the algorithm performance evaluation (test set). The teaching team then receives the test set's expected results for their final SSE comparison-based evaluation.

The second deliverable has the same goals as the first, but in this case the training set contains outliers. Because of this, the developed algorithm must be robust enough to achieve good results in spite of the outliers.

For the second part of the project, we had to focus on Image Analysis, specifically the analysis of images of butterfly wings. The data was treated the same way as on the first part of the project.

The first deliverable of this part consists on a binary problem to distinguish between images of the spots and the eyespots of the butterfly wings. The evaluation metric for this part was the F_1 score.

The second deliverable consists of a multiclass segmentation problem, where we had to predict in which of the three distinct areas of the eyespot the pixels belongs to: 1) the white ring, 2) the black and gold ring combined, and 3) the background. The evaluation metric for this part was the balanced accuracy.

1 Regression With Synthetic Data

1.1 Basic Linear Regression

The numpy data from the provided files was first uploaded to the appropriate variables. Using this data, we were able to train and test a regression model. As a result, the information from the training data was saved on the X and Y, and the information from the test data was saved on the Xtest_geral.

The next step was to scale the data, which was done using a Standard Scaler, because there was no domain knowledge about the upper and lower boundaries of the data, therefore it wasn't advisable to use the MinMaxScaler(). This way both the mean and standard deviation of X were used to scale it and Xtest_geral.

To create a better model for the data, three different algorithms were used. Then, by cross reference, the model with the lowest SSE was going to be used to predict the final values of the Xtest_geral set.

In all models a K-fold cross validation technique was used to better fine tune the

parameters and prevent our models from overfitting the data. The X and Y variables were split on a training and test set and in each fold the SSE was calculated. In the end the average of the SSE from all folds was computed and compared with the ones obtained on the rest of the models.

Linear Regression Model

The supervised machine learning model known as "linear regression" seeks the best linear relationship between the independent and dependent variables. In this dataset, the results obtained, for the SSE where in general very good. This comes with the fact that there where no outliers present.

In figure 1 it's represented the values for the SSE in each fold done on the cross validation:

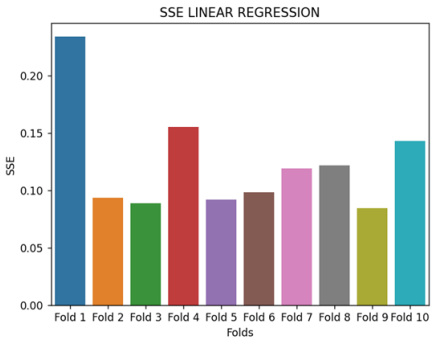

Figure 1: SSE for the linear regression model over the 10 cross validation folds.

The mean of the SSE obtain from this folds was 0.12295 with a standard deviation of 0.04344.

Ridge Regression Model

This model functions similarly to a linear regression model, but it introduces an additional bias to prevent the model from overfitting the training data. In this manner, the model offers a marginally weaker fit to the training dataset, but in the end, it yields superior long-term predictions.

Ridge Regression use L2 Regularization to introduce this bias, which increases the "squared magnitude" of the coefficient as a penalty term in the loss function. The more it increases, the more the model underfit the data; if the alpha value that determines the regularization strength is very low, it essentially has no effect and we have a linear regression model.

In this case, since we also had to tune the best alpha to use, in each cross validation fold of the training data, we did another cross validation to see which alpha value was the best. In figure 2 and 3 we can see the values of the best SSE obtained for each cross fold and the alphas that where used:

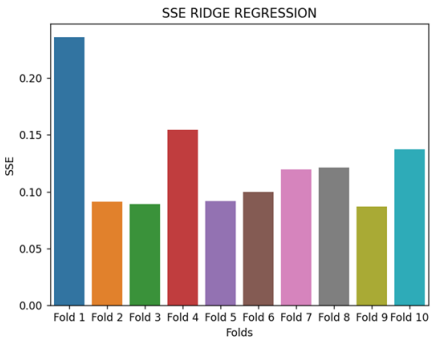

Figure 2: SSE for the ridge regression model over the 10 cross validation folds.

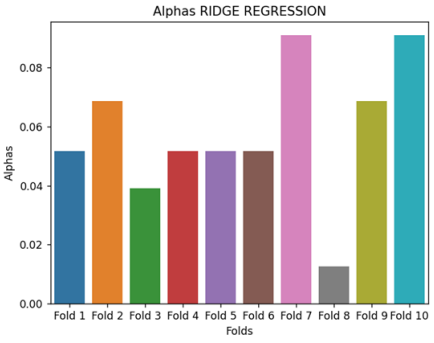

Figure 3: Alphas for the ridge regression model over the 10 cross validation folds.

Next we used the mean of the alphas which was 0.05783 to calculate the mean for the SSE, again across the cross validation of the training set. The final value for the SSE was 0.12281, which was going to be used to compare with the values obtained on the other models.

Lasso Regression Model

This model functions similarly to the Ridge Regression model explained earlier, with the exception that Lasso reduces the coefficient of the less significant characteristic to zero, eliminating certain features entirely.

Lasso Regression uses L1 Regularization to introduce its bias, which adds absolute value of magnitude of coefficient as penalty term to the loss function. The more it increases, the more the model underfit the data, by making coefficients zero; if the alpha value that determines the regularization strength is very low, again it essentially has no effect and we have a linear regression model.

As in Ridge regression the same model to tune the value of the alpha was employed, as it is visible in figure 4 and 5:

Figure 4: SSE for the ridge regression model over the 10 cross validation folds.

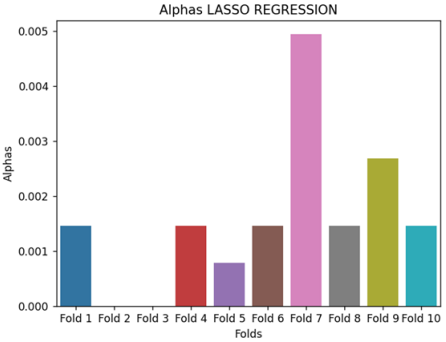

Figure 5: Alphas for the ridge regression model over the 10 cross validation folds.

Next we used the mean of the alphas which was 0.0015697 to calculate the

mean for the SSE, again across the cross validation of the training set. The final value for the SSE was 1214.0919771936037, which was going to be used to compare with the values obtained on the other models. This value is too high, because we didn't manage to properly tune the value for alpha.

Final Model

Then to choose the final model to be implemented, the mean of the SSE obtained for each of the models was compared and the model with the smallest value was finally fitted to the entire dataset (X and Y variables) and was used to predict and store the values for the Xtest_geral.

1.2 Linear Regression With Outliers

In this section, a linear regression problem was provided, with the added detail that the training set contains outliers, which will skew the trained model if a simple linear regressor is used. Due to this, other models were sought in order to obtain a good fitting model in spite of the outliers.

Isolation Forest

The first model that was implemented was the Isolation Forest model, which yielded very poor results in comparison to the other models, even with parameter tuning using GridSearchCV. These poor results might have been due to our implementation of the model but since it proved poor results it was not considered.

Huber Regressor

Another model that was implemented was the Huber Regressor. This model uses a regression technique that is robust to outliers.

In order to tune the model parameters, the model was tested with different values for the epsilon parameter, the values ranging from 1 to 5 in steps of 0.05. Initially the training data SSE was calculated with the full set without any alterations.

The observed SSE using this method was very high in comparison to the results obtained in the basic regression problem. This high SSE turned out to be due to outliers skewing the SSE. Even though the model obtained with the Huber regressor was a good predictor, when calculating the SSE the error in the outlier points was very high since the predicter "ignored" these points in training. So the SSE was very high solely due to the error in the outliers.

In order to obtain a reliable metric of the success of the model as a predictor for this problem, the outliers had to be removed from the training data in order to calculate the SSE of the predictor without the outliers, which will be more effective in predicting the success of the model in the test data since it has no outliers. To achieve this, the RANSAC algorithm was used to identify the outliers in the validation set of the training data, which were subsequently removed from this set.

The resulting SSE of the training data (using the training-test split) from this model, with the optimum epsilon was very low (between 0.001 and 0.01) and similar to the first problem. The SSE for the filtered data for each epsilon value was plotted and can be seen in figure 22, with the optimum epsilon highlighted in red.

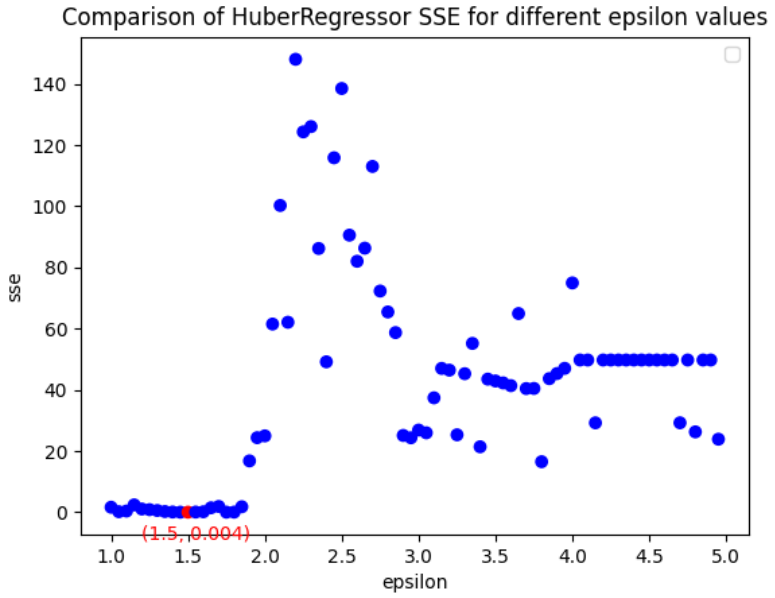

Figure 6: Caption

Ridge Regressor with outlier removal using RANSAC Algorithm

The final approach that was implemented was to remove the outliers from the training set, using the RANSAC algorithm, to then perform some type of linear regression model to predict the output.

Using the default RANSAC model from the *sklearn* library, the outliers were identified and then removed. After this filtering process, a ridge regression model was used to predict the training data, using the same method as in section 1.1. This is, the model was tested with the training data for each alpha value using k-fold validation, after which the optimum alpha value (the value with the lowest SSE) was selected. The training data SSE was then calculated doing the k-fold validation again and doing the mean of the folds.

The value obtained for the training data SSE was satisfactory but relatively higher than the one in the basic regression problem, ranging from 0.5 to 1.5.

Final Model

Considering the training data SSE's obtained for each model, we can see that the Huber Regressor that was implemented yielded the best results in predicting the output for this problem, so this model was selected to predict the test set output. One aspect that might worsen the performance this regressor at predicting new data

is the fact that a training-validation split wasn't implemented, so the model might be overfit to the training data, which might cause it's performance as predictor on the training data to be higher than the one obtained in the test data.

2 Image Analysis

2.1 First task

Functions Implemented and Code Structure

In this task we had to deal with imbalanced data. To do this after uploading the data, we oversampled the eyespot ocurrences, seeing that there where 3131 images of this kind and 5142 spot images. The oversampling process was done by the "*overSample()*" and "*tweakImage()*" functions.

The first function was intended to take the difference between the biggest set of images and use that number to create new images, with the second function, on the lowest set of images. In this case, 2011 new images where created by randomly rotating and mirroring random images taken from the eyespot set.

After this oversampling was done, there where two processes that we had to do. The first was the tuning of the hyperparameters for each of the models used. In this phase, we used the Gridsearch portion of the code and we tested for each model separately. After the tuning was done for all of the models, the second phase took part. In this phase, we run and predicted all the models on the training set at the same time. Then at the end, the model that obtained the best f1 score, compared with the rest, was used to predict the final values to be evaluated.

The Algorithms employed where carefully chosen, based on an extended research about the topic of classifiers for Image Analysis.

Multi-layer Perceptron classifier

The first algorithm employed was a Neural network. On the hyperparameter tunning, we focused on the hidden_layer_sizes, max_iter, random_state, activation, solver and the learning_rate. At the end the best result obtained for both the hyperparameters and the F_1 score was:

```
Model name: nn
model: MLPClassifier(hidden_layer_sizes=(100, 50, 20), max_iter=2000, random_state=4)
score: 0.855799883321511
```

Figure 7: Hyperparameters and the F_1 score for the NN algorithm.

The classification report and the confusion matrix where as shown:

```
BINARY PROBLEM SCORES NN:
              precision    recall  f1-score   supp

           0       0.92      0.78      0.85
           1       0.81      0.93      0.87

    accuracy                           0.86
   macro avg       0.86      0.86      0.86
weighted avg       0.86      0.86      0.86
```

Figure 8: Classification Report for the NN algorithm.

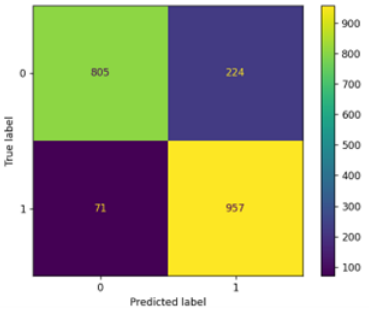

Figure 9: Confusion Matrix for the NN algorithm.

C-Support Vector Classifier

For the hyperparameter tunning, we focused on the C, kernel, random_state, degree and class_weight. At the end the best result obtained for both the hyperparameters and the F_1 score was:

```
Model name: svc
model: SVC(C=10, class_weight='balanced', random_state=4)
score: 0.8701550071457196
```

Figure 10: Hyperparameters and the F_1 score for the C-SVC algorithm.

The classification report and the confusion matrix where as shown:

```
              precision    recall  f1-score   supp

           0       0.88      0.85      0.87
           1       0.86      0.89      0.87

    accuracy                           0.87
   macro avg       0.87      0.87      0.87
weighted avg       0.87      0.87      0.87
```

Figure 11: Classification Report for the C-SVC algorithm.

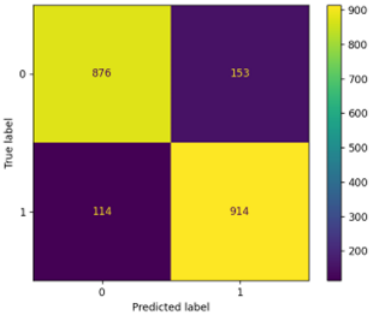

Figure 12: Confusion Matrix for the C-SVC algorithm.

NU-Support Vector Classifier

For the hyperparameter tunning, we focused on the nu, kernel, random_state, degree and class_weight. At the end the best result obtained for both the hyperparameters and the F_1 score was:

```
Model name: nusvc
model: NuSVC(class_weight='balanced', degree=4, nu=0.2, random_state=4)
score: 0.8716205865498636
```

Figure 13: Hyperparameters and the F_1 score for the NU-SVC algorithm.

The classification report and the confusion matrix where as shown:

	precision	recall	f1-score
0	0.89	0.85	0.87
1	0.86	0.89	0.87
accuracy			0.87
macro avg	0.87	0.87	0.87
weighted avg	0.87	0.87	0.87

Figure 14: Classification Report for the NU-SVC algorithm.

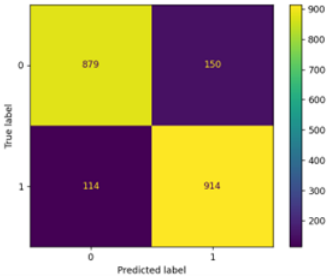

Figure 15: Confusion Matrix for the NU-SVC algorithm.

k-Nearest Neighbors Classifier

For the hyperparameter tunning, we focused on the n_neighbors, weights and leaf_size. At the end the best result obtained for both the hyperparameters and the F_1 score was:

```
Model name: kneighbors
model: KNeighborsClassifier(n_jobs=-1, n_neighbors=6, weights='distance')
score: 0.8416862331376094
```

Figure 16: Hyperparameters and the F_1 score for the k-Nearest Neighbors algorithm.

The classification report and the confusion matrix where as shown:

	precision	recall	f1-score	support
0	0.88	0.80	0.83	1029
1	0.81	0.89	0.85	1028
accuracy			0.84	2057
macro avg	0.84	0.84	0.84	2057
weighted avg	0.84	0.84	0.84	2057

Figure 17: Classification Report for the k-Nearest Neighbors algorithm.

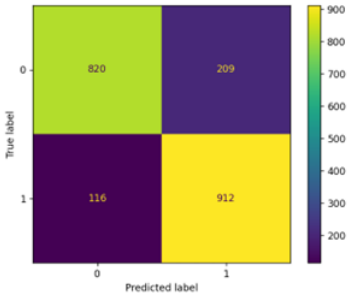

Figure 18: Confusion Matrix for the k-Nearest Neighbors algorithm.

Decision Trees Classifier

For the hyperparameter tunning, we focused on the criterion, class_weight and random_state. At the end the best result obtained for both the hyperparameters and the F_1 score was:

```
Model name: decisiontrees
model: DecisionTreeClassifier(class_weight='balanced', criterion='entropy',
                              random_state=4)
score: 0.7937492666588444
```

Figure 19: Hyperparameters and the F_1 score for the Decision Trees algorithm.

The classification report and the confusion matrix where as shown:

	precision	recall	f1-score	support
0	0.83	0.74	0.78	1029
1	0.77	0.85	0.80	1028
accuracy			0.79	2057
macro avg	0.80	0.79	0.79	2057
weighted avg	0.80	0.79	0.79	2057

Figure 20: Classification Report for the Decision Trees algorithm.

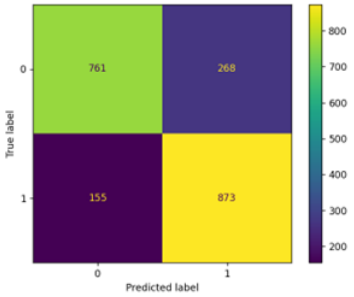

Figure 21: Confusion Matrix for the Decision Trees algorithm.

Gaussian Naive Bayes Classifier

For the hyperparameter tunning, we focused on the var_smoothing. At the end the best result obtained for both the hyperparameters and the F_1 score was:

```
Model name: gaussian
model: GaussianNB(var_smoothing=533.6699231206302)
score: 0.6531548241324656
```

Figure 22: Hyperparameters and the F_1 score for the Gaussian Naive Bayes algorithm.

The classification report and the confusion matrix where as shown:

	precision	recall	f1-score	support
0	0.70	0.55	0.62	1029
1	0.63	0.77	0.69	1028
accuracy			0.66	2057
macro avg	0.67	0.66	0.65	2057
weighted avg	0.67	0.66	0.65	2057

Figure 23: Classification Report for the Gaussian Naive Bayes algorithm.

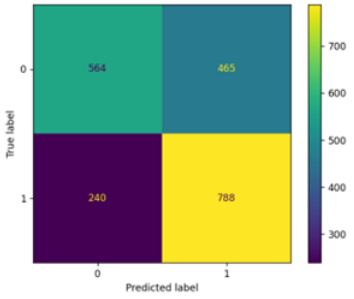

Figure 24: Confusion Matrix for the Gaussian Naive Bayes algorithm.

First Task Final Results and Conclusions

As seen from the previous sections, the best model was the NU-Support Vector Classifier. Therefore this was the model implemented to produce the final delivery results.

From this task, there where a couple of assumptions that we took and implemented on the second one.

We realized that we needed to make the code more autonomous, so that the hyperparameters tuning was easier to implement and the score testing could be done after this part was over, without the need to change the code.

Then the two biggest assumptions that we made was that we needed to reduce the size of the number of samples, in order for us to test more hyperparameters and in addition to create more accurate models, without being constrained to the long running time of the code. The other assumption was that before choosing the best model based solely on one testing run, we would test it on different sets and see which model would perform better, most times.

2.2 Second task

There where two big "issues" with the data that needed to be taken care of. The first was the fact that the data was extremely imbalanced and the second was that in 50700 images, 8694 were from the white ring (1), 40438 from the black and gold ring (2) combined, and 1568 from the background (3).

To handle with this fact we implemented three functions to balance the data: the *underSample_all_to_min*, *overSample_underSample_to_mid* and *overSample_all_to_max*.

The models that were implemented as predictors for this problem were the Decision Trees Classifier, the K-Neighbours Classifier and the NU-Support Vector Classifier.

Parameter tuning for the models

The first function, *underSample_all_to_min*, undersampled the two biggest sets to the size of the lowest, in this case both the white ring and the black and gold ring sizes were reduced to 1568, the size of the background. This was done in order to have a smaller set of data to tune the hyperparameters of each model, as this was done using *GridSearchCV()* which is time intensive, so the running time of the program would not be too long. The parameter tuning was done by using *GridSearchCV()* for 10 different training-validation splits, for each model. The best parameters were saved for each iteration for each model, 30 configurations in total.

For the K-Neighbours Classifier, the parameters that were tuned were: n_neighbours, leaf_size and p. For the Decision Trees Classifier, the parameters that were tuned were the crieterion, max_features and max_features. Finally, for the NU-SVC Classifier, the parameters that were tuned were the kernel and degree.

Final model selection

The *overSample_underSample_to_mid* function was implemented in order to have a substantial data set in order to train the 30 different models with the best hyperparameters obtained in the previous method. This function does this by oversampling the smallest data set to the size of the one in the middle, and by undersampling the largest one to the same size. The oversampling process was done as in the previous section, by adding random images from the respective data set after rotating and mirroring them randomly, until the data set has the desired size. In the case of the training data, the background set was oversampled to 8694 images and the black and gold ring set was undersampled to the same size.

These 30 models were tested in 5 different training-validation splits, and the model that proved to have the best balanced accuracy score was the K-Neighbours classifier with the following parameters:

```
Model name: kneighbors_fold2
best params: {'leaf_size': 20, 'n_neighbors': 2, 'p': 1, 'weights': 'distance'}
```

Figure 25: Best model obtained and its parameters

Final model training

Finally, the *overSample_all_to_max* function oversampled the 2 smallest sets to the size of the largest one in order to train the model which proved the best balanced accuracy score in the previous method, with the biggest dataset possible. The final model resulted in a balanced accuracy score of 0.974, which is very high. The confusion matrix and the precision, recall and f1 scores of each class can be seen in figure 26

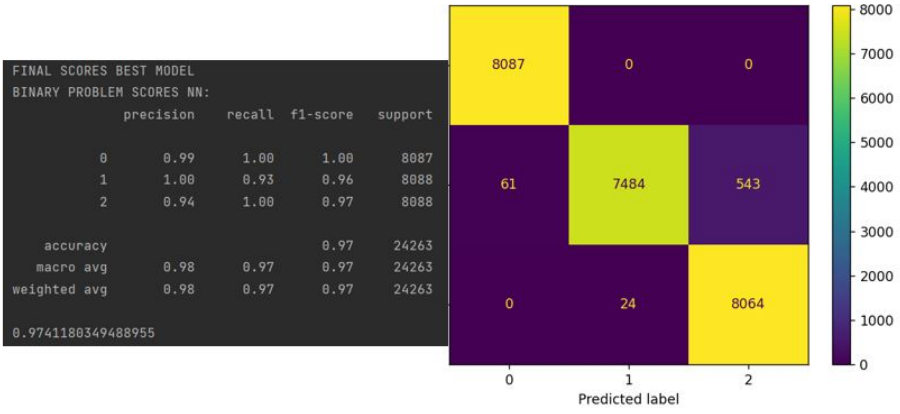

Figure 26: Best model results

2.2.1 Second task final results and conclusions

The model proved to have a significantly lower balanced accuracy score in the test set provided by the teaching staff than in the training set. This might be due to the fact that the oversampling on the training data before training the final model repeated the data countless times. For example for the background pixels, 13870 images of synthetic data were created, which created a lot of redundancy in that class since the new images were the same images of the original set but rotated and/or mirrored randomly. This might have caused the model to have overfit to the data, mainly in the classes that were overfit, since they had a lot of repeat data, even with the training-validation splits.