Handling Bias-Variance Trade-off

Margarita Kilinkarov 337904619, Mor Yaacov 208934778

March 2022

1 Abstract

It is important to understand prediction errors when it comes to accuracy in any machine learning algorithm. Prediction errors can be decomposed into two main subcomponents of interest: error from bias, and error from variance. Many times, steps taken to fight one (either High Bias or High Variance) can end up worsening the other. Besides, many data-scientists tend to evaluate the model using only error evaluation metrics such as MSE, RMSE, R square etc. In our project we will try to find the optimal model that deals with both over- and under-fitting, and maximizes the predictions by dealing with high bias and/or high variance. In our work, we used many prediction algorithms and techniques (and combination of those) in order to decide which gives the optimal bias variance balance. The bias and variance were measured by bias_variance_decomp() from mlxtend.evaluate library. The purpose of this project is to reach the ratio of 1 between them (equal). In our results we found some good models that maximize the prediction by handling bias-variance trade-off but it shows that we can't find the only optimal model for different datasets. The best results that we got were using such techniques as cross-validation & decision trees and regularizations.

2 Problem description

Data science pipeline includes many stages. Our project mainly focuses on the modeling stage but uses every prior stage to figure how to create the best model with bias variance ratio, according to their results. In this stage we create models. Models are built with algorithms and different learning techniques, which is used as a predictive tool.

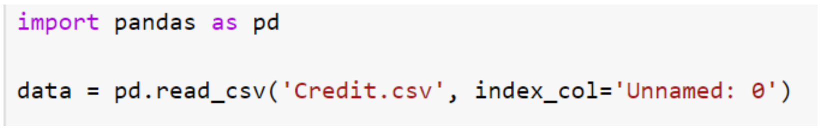
What is bias? Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. Model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on training and test data.

What is variance? Variance is the variability of model prediction for a given data point or a value which tells us the spread of our data. Model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn't seen before. As a result, such models perform very well on training data but have high error rates on test data. The center of the target represents the model that predicts correct values. And as we become farther from it, then our predictions are getting worse.

Underfitting happens when a model is unable to capture the patterns of the data and its connections between the data and the results of predictions. These models usually have high bias and low variance. Overfitting happens when our model captures the "noise" that includes in the data, does not capturing the patterns, but the examples themselves.

3 Solution overview

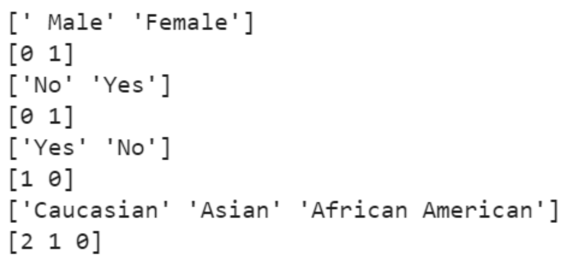
First of all, we load our data from a CSV file into a Pandas DataFrame using pd.read_csv() function in Pandas:



```
import pandas as pd

data = pd.read_csv('Credit.csv', index_col='Unnamed: 0')
```

Figure 1: Reading the Credits.csv dataset

We drop the index column from the DataFrame in order to replace it with the standard sequential indexing. In many Machine-learning or Data Science activities, the data set might contain text or categorical values (basically non-numerical values) but most of the algorithms expect numerical values to achieve state-of-the-art results. In order to properly use the categorical features, we use LabelEncoder(). It is used to convert text or categorical data into numerical data which the model expects and performs better with. Labels are arranged in alphabetical order and a unique index is assigned to each label starting from 0. The output will be:



```
[' Male' 'Female']
[0 1]
['No' 'Yes']
[0 1]
['Yes' 'No']
[1 0]
['Caucasian' 'Asian' 'African American']
[2 1 0]
```

Figure 2: Label's encoding

As we mentioned in our proposal, we split our data to train and test set to be 80% and 20% by using train_test_split() from sklearn.model_selection. It's a function in Sklearn model selection for splitting data arrays into two subsets: for training data and for testing data. After that we create x_train and x_valid using DataFrame.drop from Pandas - the drop function that allows us to remove the column "Balance" (the data that we are going to predict) from our DataFrame. And we use y_train and y_valid that will hold "Balance" data.

At this point we proceed to our data modeling part. In our base model we used the **KNN algorithm** for Linear Regression's predictions. KNN is one of the simplest models since it is a non-parametric and lazy learning method. In the case of Linear Regression, the output is based on the mean of the k-nearest neighbors in the feature space. After trying all kinds of values between 10 and 30 we found that the optimal value of neighbors is 25. The error rate that we got was about 200 after a number of different runs of the dataset, and the estimation of the quality of predictions was done by RMSE (Root Mean Squared Error). Handling bias-variance trade-off: the bias will be 0 when K=1, however, when it comes to new data (in test set), it has a higher chance to be an error, which causes high variance. When we increase K, the training error will increase (increase bias), but the test error may decrease at the same time (decrease variance).

The next method that we used is **regularization**. We tried both L1 (Lasso) and L2 (Ridge regression) regularization that in most of the cases gave us almost the same results. Both methods gave us an improvement of about 40% over our base model. What Regularization does to overfit models is that it negates or minimizes the effect of predictor columns with large outliers, by penalizing their regression coefficients. The result is a smoother model which can work well on other test data sets with similar kinds of data. Regularization attempts to reduce the variance of the estimator by simplifying it, something that will increase the bias, in such a way that the expected error decreases.

The next technique that we tried was decision tree's algorithms, in particular Bagging and Random Forest. Let's start with **Bagging** (= bootstrap aggregation). The bootstrap is a powerful statistical method for estimating a quantity from a data sample. The algorithm of bootstrap procedure:

1. Create many (e.g. 1000) random sub-samples of our dataset with replacement (meaning we can select the same value multiple times).
2. Calculate the mean of each sub-sample.
3. Calculate the average of all of our collected means and use that as our estimated mean for the data.

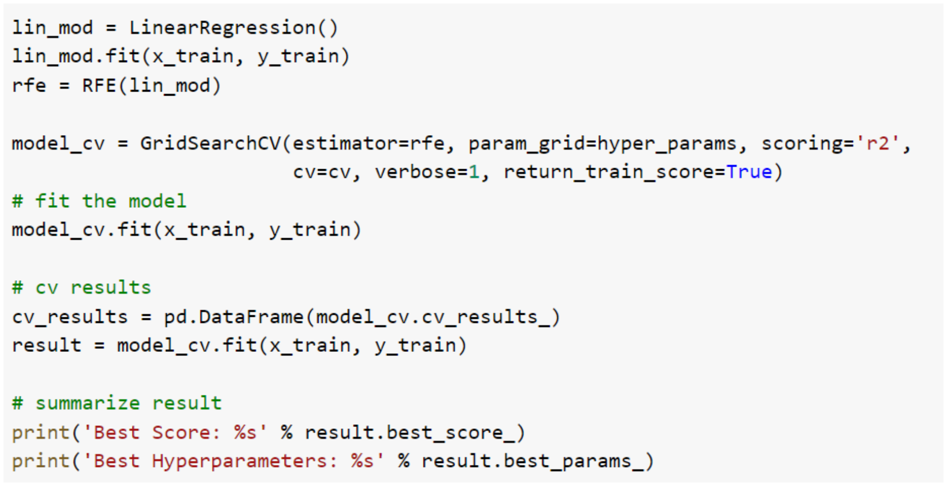
Bagging is the application of the Bootstrap procedure that can be used to reduce the variance for those algorithms that have high variance, typically decision trees. Bagging of some algorithm would work as follows:

1. Create many (e.g. 100) random sub-samples of our dataset with replacement.
2. Train a model on each sample.
3. Given a new dataset, calculate the average prediction from each model.

Random forest is an extension of bagging that also randomly selects subsets of features used in each data sample. The main difference is that all features (variables or columns) are not used; instead, a small, randomly selected subset of features (columns) is chosen for each bootstrap sample.

This has the effect of de-correlating the decision trees (making them more independent), and in turn, improving the ensemble prediction. The main idea behind random forest is that lots of high variance and low bias trees combine to generate a low bias low variance forest. Since it is distributed over different trees and each tree sees a different set of data, therefore random forests in general do not overfit. And since they are made of low bias trees, underfitting also does not happen. Our results show the improvement of about 40-50% over the base model.

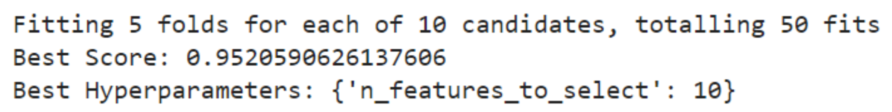
Another method for handling high bias/variance that we used is **Cross-Validation**. It is a resampling procedure that uses different portions of the data to test and train a model on different iterations. In our algorithm we used K-Fold Cross-Validation when the k is 5. In K Fold cross validation, the data is divided into k subsets. The method is repeated k times, such that each time, one of the k subsets is used as the test set/ validation set and the other k-1 subsets are put together to form a training set. This significantly reduces bias as we are using most of the data for fitting, and also significantly reduces variance as most of the data is also being used in validation set. We also added RFE (Recursive Feature Elimination - a popular feature selection algorithm) for this model. RFE works by searching for a subset of features by starting with all features in the training dataset and successfully removing features until the desired number remains. So we explored the model using both methods (CV and RFE):



```
lin_mod = LinearRegression()
lin_mod.fit(x_train, y_train)
rfe = RFE(lin_mod)

model_cv = GridSearchCV(estimator=rfe, param_grid=hyper_params, scoring='r2',
                        cv=cv, verbose=1, return_train_score=True)
# fit the model
model_cv.fit(x_train, y_train)

# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
result = model_cv.fit(x_train, y_train)

# summarize result
print('Best Score: %s' % result.best_score_)
print('Best Hyperparameters: %s' % result.best_params_)
```

Figure 3: Cross-Validation model using RFE method

And the results that we got were:

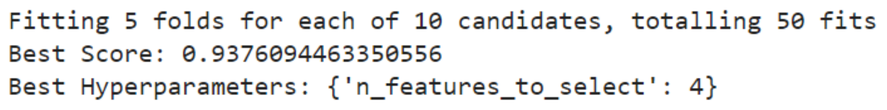


```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best Score: 0.9520590626137606
Best Hyperparameters: {'n_features_to_select': 10}
```

Figure 4: Exploring the results - the optimal number of features is 10

So in the combined model of these two methods we used all 10 features and the final results were good enough over the base model but not a big difference compared to other methods.

The final model that we tried and that gave us the best results in most of the runs was a combination of the previous method **(CV and RFE) and Random-Forest algorithm**. So we took the model that we used in the Random-Forest part and activated the code of CV and RFE on it. After the exploring this model we got such results:



```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best Score: 0.9376094463350556
Best Hyperparameters: {'n_features_to_select': 4}
```

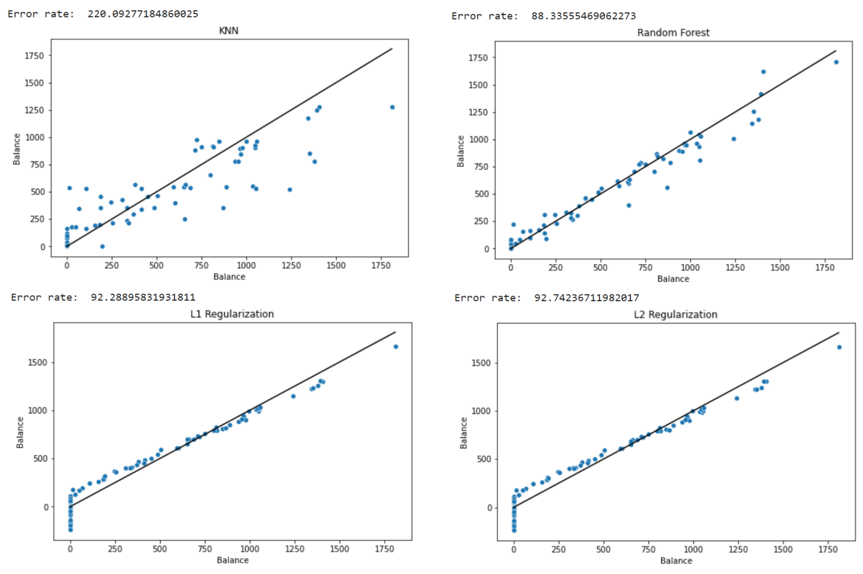
Figure 5: Exploring the results - the optimal number of features is 4

Thus, in our final model for this dataset we chose 4 features and improved our previous method of CV+RFE combination, which in turn gave the best results over all previous models in most of the cases. Unfortunately, we couldn't find any "universal" model for all datasets. After we ran the same actions for the other 3 dataset we got different results, but the main two models that gave us the best results are CV+RFE and Random-Forest algorithm, and Regularization techniques.

4 Experimental evaluation

In order to evaluate which model is the best one we used such tools as RMSE (Root Mean Squared Error) - estimator of the quality of prediction - and function bias_variance_decomp() from mlxtend.evaluate library.

In the following graphs the black line is the real balance values and the blue dots are our model's prediction values:



```
Error rate:  220.09277184860025
Error rate:  88.33555469062273
Error rate:  92.28895831931811
Error rate:  92.74236711982017
```

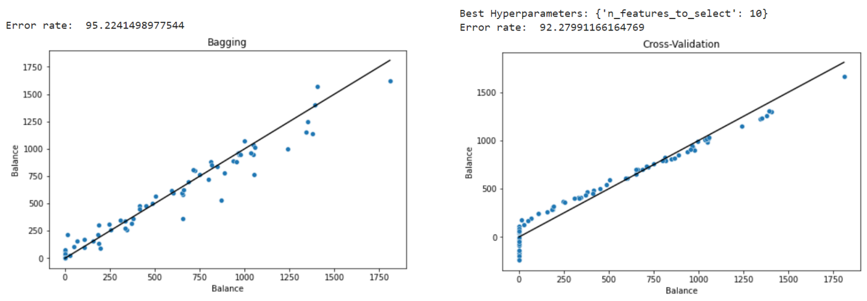

```
Error rate:  95.2241498977544
```
```
Best Hyperparameters: {'n_features_to_select': 10}
Error rate:  92.27991166164769
```

Figure 6: Comparison between our model's prediction values and real balance values

As we can see, the smaller the error - the closer our prediction to the real values. And our final model:

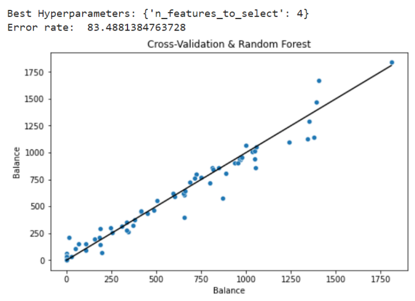

```
Best Hyperparameters: {'n_features_to_select': 4}
Error rate:  83.4881384763728
```

Figure 7: Comparison between our model's prediction values and real balance values in the final model

We also measured the values of Bias and Variance with respect to RMSE. How are these values measured? MSE is measured by the following formula:

$$MSE = E[(\hat{f}(x) - f(x))^2]$$

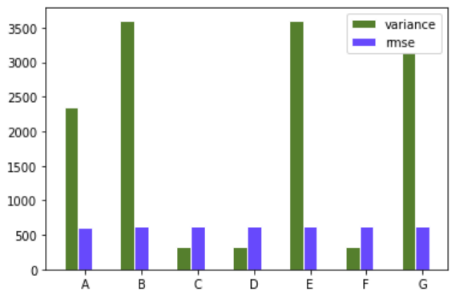
This error may then be decomposed into bias and variance components:

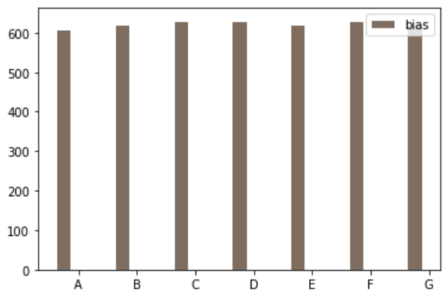
$$MSE = Bias^2 + Variance$$

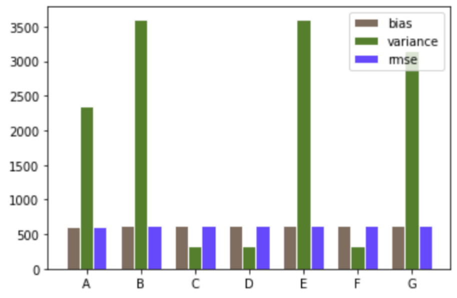
The proof:

$$E[(\hat{f}(x) - f(x))^2] = E[\hat{f}(x)^2] + f(x)^2 - 2E[\hat{f}(x)]f(x)$$
$$Bias^2(\hat{f}(x), f(x)) = (E[\hat{f}(x)] - f(x))^2 = E^2[\hat{f}(x)] + f(x)^2 - 2E[\hat{f}(x)]f(x)$$
$$Variance(\hat{f}(x)) = E[\hat{f}(x)^2] - E^2[\hat{f}(x)]$$

So on the following graphs we can see the relation of bias and variance to RMSE (after we calculate square root from Bias and MSE).







5 Related works

Most of the works that we found are trying to focus on one or two specific techniques. So we decided to use these works in our project and combined the ideas of some of them into the whole one. These are several of the articles:

- Understanding the Bias-Variance Tradeoff
- K-Nearest Neighbors and Bias-Variance Tradeoff
- Work about Regularization and the bias-variance trade-off
- Cross-Validation with Linear Regression using RFE

6 Conclusion

The purpose of our project was to find the optimal model that manages to deal with bias-variance trade-off. We couldn't not find one but we found out that apparently there is no such model at all. Our conclusion is that we need to focus on every dataset separately by using the techniques that were described in this project. It is important to take into consideration not only error estimator but all the metrics that we talked about in this article.