Introduction

This section describes our most promising approaches to handling the high-dimensionality of the dataset using machine learning techniques, and our steps to building a final ensemble model that is able to predict future data. Some more conclusions about the dataset and the effectiveness of some machine learning techniques on the dataset are also found while we try different approaches to find the best way. There are also a lot more approaches that we tried, but since they did not help us to draw any insight, or their performances were very poor, we decided to not put them in the repository nor describe them in this section.

Specifications

All of the results, insights, and conclusions drawn in this part are based on the executed code from the four reproducible Jupyter notebooks named "KMeansSMOTE\_and\_ML\_models", "Models\_and\_Feature\_selection", "Machine\_learning\_approaches\_(draft)", and "Machine\_learning\_approaches\_(final)" (in chronological order) in the "Code" directory.

The notebooks were run on Google Colaboratory without GPU or TPU acceleration. The hardware specifications and package versions of all notebooks are specified in the last section "Information" of the "Machine\_learning\_approaches\_(final)" notebook. It is worth noting that the notebooks used Python 3.7.13.

Oversampling techniques

K-means SMOTE is “a simple and effective oversampling method based on k-means clustering and SMOTE (synthetic minority oversampling technique), which avoids the generation of noise and effectively overcomes imbalances between and within classes” [1]. In simple words, the technique applies k-means clustering before over-sampling using SMOTE [2].

In the notebook "KMeansSMOTE\_and\_ML\_models", we used the class "KMeansSMOTE" of the "imblearn.over\_sampling" package [2]. This class has three parameters that we tried to tune, which are "kmeans\_estimator", "cluster\_balance\_threshold" and "k\_neighbors".

For the "kmeans\_estimator" parameter, it should be an instance of the class "KMeans" of the "sklearn.cluster" package. The class "KMeans" is an implementation of the clustering algorithm k-means in scikit-learn [3]. Therefore, we tried to tune the "n\_clusters" hyperparameter of this class. This parameter accepts a positive integer as its input for the number of clusters, so we tried with 1, 2, 3, 5, 7, 10, 13, 19, 26, 37, 51, 71, and 100, which approximately are "numbers spaced evenly on a log scale" [4] from 1 to 100 (including). The result shows that, except for 1, trying all the other numbers caused the same error "no clusters found with sufficient samples of class 1" (Figure 1). After that, we specifically set the "cluster\_balance\_threshold" to 0.0001 so all of them worked. To evaluate the generalization quality of the over-sampling method, we first split the original dataset into two sets, using one for training and the other for testing. Then, for each dataset created from the training dataset using a specific "n\_clusters" value, we trained a random forest [3] on that over-sampled dataset, then used it to predict the testing dataset, and finally evaluated the f2 score [3] between the real testing results and the predicted results. So the quality of the over-sampled dataset can be approximately measured by that f2 score. The output showed that all of the values of "n\_clusters" were bad but 1. We drew the conclusion that the k-means SMOTE over-sampling technique is not applicable in this dataset since it only works with "n\_clusters" equals 1.

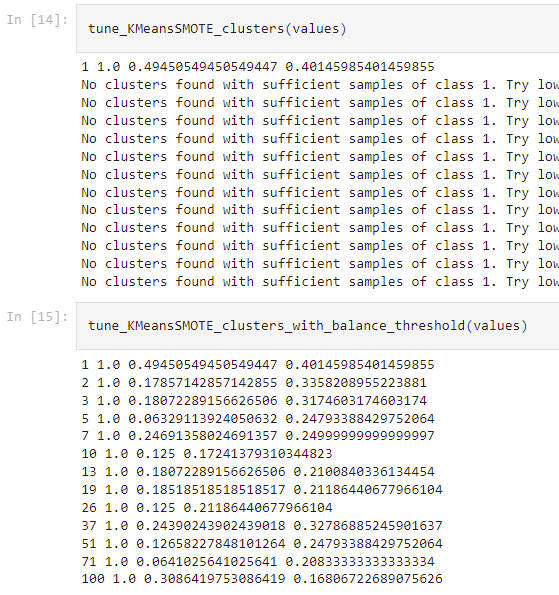


Figure 1. KMeansSMOTE tuning results on "kmeans\_estimator" and "cluster\_balance\_threshold".

However, we are still in doubt if SMOTE is applicable, so we hard fixed "n\_clusters" to be 1, let "cluster\_balance\_threshold" be "auto", and tried different values for "k\_neighbors". Though the results in Figure 2 showed really big differences (or high variance) between the tested f2 values, those differences are really random with different random state values. Therefore, we drew two other conclusions. Firstly, SMOTE is applicable, and it is even good. Secondly, there is no best value for "k\_neighbors".

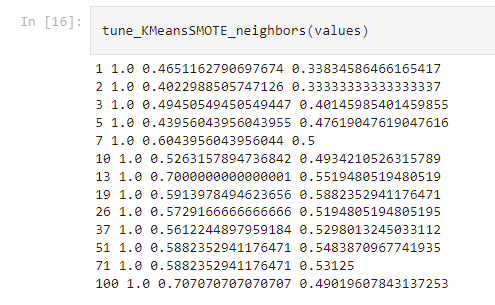


Figure 2. KMeansSMOTE tuning results on "k\_neighbors".

In conclusion, we decided to use SMOTE without tuning the number of neighbors.

Machine learning models

In the notebook "Models\_and\_Feature\_selection", we split the original dataset into two datasets (training and test), then applied SMOTE (with the parameter "k\_neighbors" set to 5 by default) to over-sample the training dataset and used that as the new training dataset. From this over-sampled dataset, we created two other training datasets. The first one is named the resampled dataset, which has a number of data points of twice as much as the over-sampled dataset, and it was resampled (with replacement) [3] from the over-sampled dataset above, this dataset is used to remove outliers and noise. The second one is named the filtered dataset, which has filtered many columns (features) after we applied Lasso regression [5] and removed all features with the coefficients of the predictor function equal 0. For evaluation purposes, we also created a new filtered test set, in which all of the same columns are also filtered. We clearly show all those datasets in Figure 3.

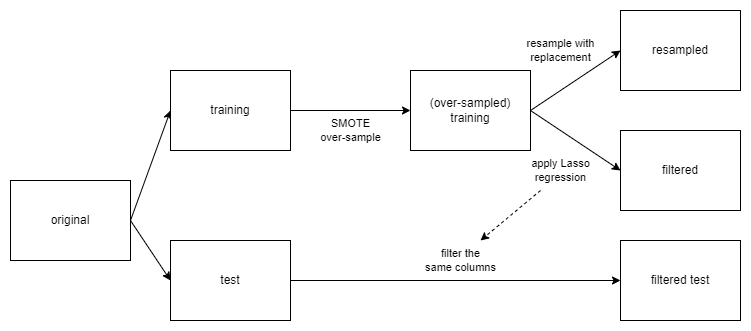


Figure 3. Datasets used in "Models\_and\_Feature\_selection".

With the three datasets (training, resampled, and filtered) in hand, we tried to apply some of the most promising machine learning models. They are neural networks, bagging, random forests, AdaBoost, and gradient boosting. For all hyperparameters, we also used 1, 2, 3, 5, 7, 10, 13, 19, 26, 37, 51, 71, and 100 as their values. For each model, for each dataset, and for each hyperparameter value, we trained all of them separately, with the exception that bagging and random forests were not trained on the resampled dataset, since the two algorithms already used bootstrap sampling to cope with noise [3].

For each figure from Figure 4 to Figure 16, the x axis is for hyperparameter values, and the y axis is for training and testing f2 scores. In particular, the orange line represents the training scores, and the blue line represents the testing scores.

# Neural networks

Each neural network has two layers. The two layers have the same number of neurons, and this number is the parameter we were trying to tune. Figure 4, Figure 5, and Figure 6 are the results of neural networks on each dataset.

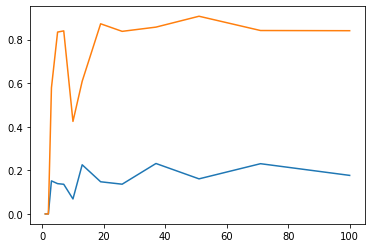


Figure 4. Performance of neural networks on the training dataset.

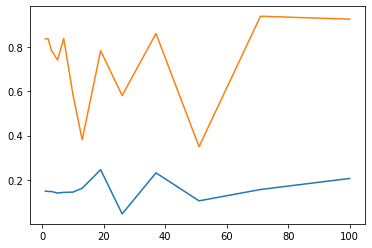


Figure 5. Performance of neural networks on the filtered dataset.

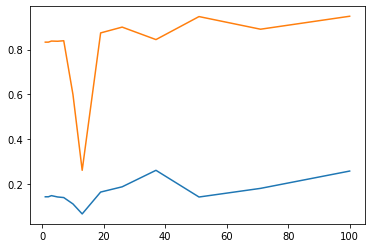


Figure 6. Performance of neural networks on the resampled dataset.

The results may vary a lot as we tried with different random seeds. They were so inconsistent that we decided to not use neural networks in our final model. Later on, we found that our dataset had too few data points for a good neural network, even after over-sampling or resampling. Feature selection with Lasso did not help at all.

# Bagging

"A bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregates their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it" [3].

We used the default base estimator of scikit-learn, which is the decision tree classifier. The hyperparameter to tune is "n\_estimators", which is the number of decision trees used in the bagging classifier. Figure 7 and Figure 8 are the results of bagging on two datasets.

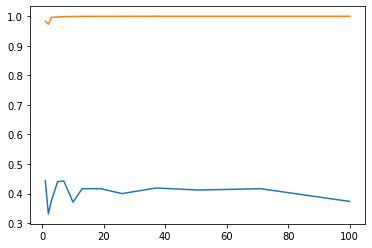


Figure 7. Performance of bagging on the training dataset.

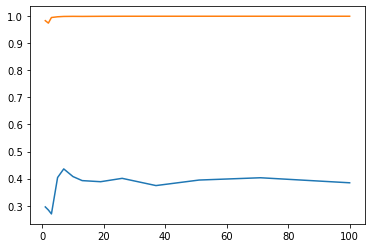


Figure 8. Performance of bagging on the filtered dataset.

Providing consistent testing scores, bagging is a good classifier in this case. However, feature selection using Lasso generally made the generalization scores worse.

# Random forests

"A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting" [3].

Compared to bagged decision trees, "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" [6].

"n\_estimators" is the hyperparameter to tune in our work. Figure 9 and Figure 10 represent the results.

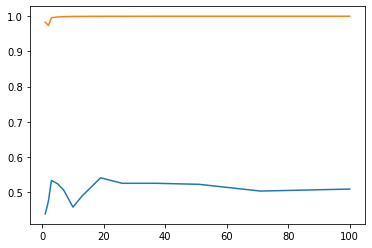


Figure 9. Performance of random forests on the training dataset.

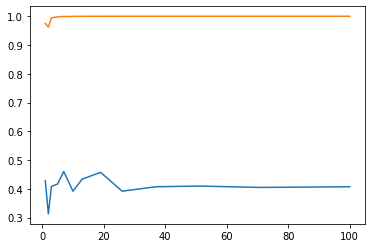


Figure 10. Performance of random forests on the filtered dataset.

As with bagging, random forests are good, but the Lasso feature selection made the scores worse.

# AdaBoost

"An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases" [3].

As with bagging, the default decision tree classifier is used as the base estimator of AdaBoost and the tuned hyperparameter is "n\_estimators". However, AdaBoost classifiers do not use any sampling method. Therefore, all three datasets are used. Their results are represented in Figure 11, Figure 12, and Figure 13.

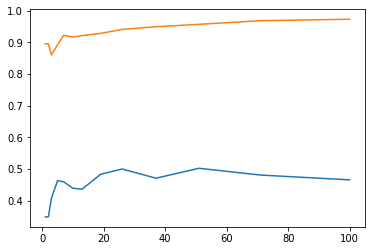


Figure 11. Performance of AdaBoost on the training dataset.

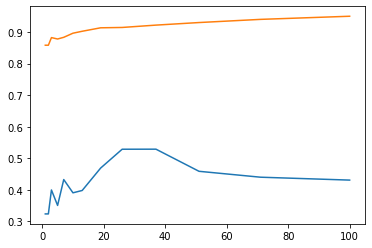


Figure 12. Performance of AdaBoost on the filtered dataset.

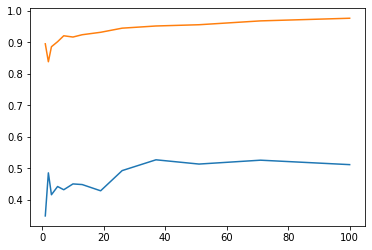


Figure 13. Performance of AdaBoost on the resampled dataset.

Among three datasets, the f2 scores of AdaBoost on the resampled dataset were the best. The performance on the training dataset was only slightly worse, and on the Lasso filtered dataset was clearly the worst, since it was really inconsistent between different random seeds. In exchange for only slightly better performance, AdaBoost took a much longer time to run on the resampled dataset compared to on the training dataset. In conclusion, using the resampled dataset on AdaBoost was not worth the time.

# Gradient boosting

Gradient boosting "builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage n\_classes\_ regression trees are fit on the negative gradient of the loss function, e.g. binary or multiclass log loss. Binary classification is a special case where only a single regression tree is induced" [3].

We tuned the "n\_estimators" hyperparameter, which is the number of trees, on the gradient boosting classifiers. As with AdaBoost, we also needed to apply gradient boosting on all three datasets. Results are shown in Figure 14, Figure 15, and Figure 16.

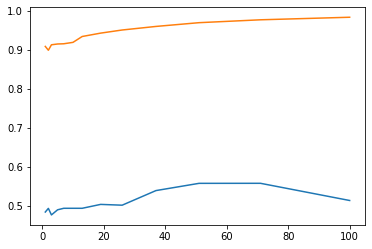


Figure 14. Performance of gradient boosting on the training dataset.

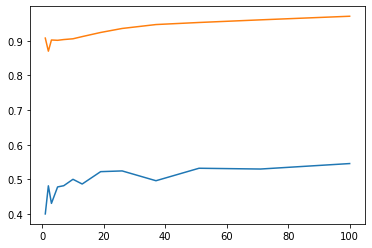


Figure 15. Performance of gradient boosting on the filtered dataset.

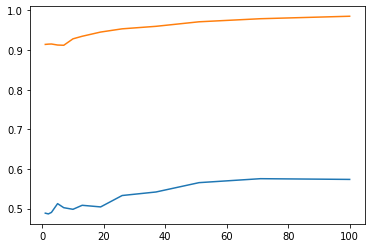


Figure 16. Performance of gradient boosting on the resampled dataset.

The testing f2 scores of gradient boosting on the filtered dataset were inconsistent at first but started to make some improvements and stabilize, though it was still the worst dataset amongst the three datasets. We did not see that much of a difference between the performance of the training dataset and the resampled dataset when we tried different random seeds. (However, the graphs generated by the notebook with the random seed of 42 do not seem like that, but to keep the notebook reproducible and fair, we still put them in this report.) On the resampled dataset, gradient boosting took a really long time to train, so it was not worth the time.

# Conclusion

After all the experiments above, we decided to use the following machine learning models: bagging, random forests, AdaBoost, and gradient boosting. For the dataset, we would use the training dataset (or the SMOTE over-sampled original training dataset).

Building the final prediction model

In the notebook "Machine\_learning\_approaches\_(final)", we first split the original dataset into two datasets, named "training and validation" and "test". We created a new dataset named "over-sampled training and validation" from the "training and validation" dataset using SMOTE. Afterwards, we split the "training and validation" dataset into two datasets, which are the training and the validation datasets. Finally, we SMOTE over-sampled the training dataset. The two datasets, which are the over-sampled training dataset and the "over-sampled training and validation" dataset, would be used for training purposes. The other two datasets, which are the validation dataset and the test dataset, would be used for testing and evaluating purposes. Figure 17 summarizes the process.

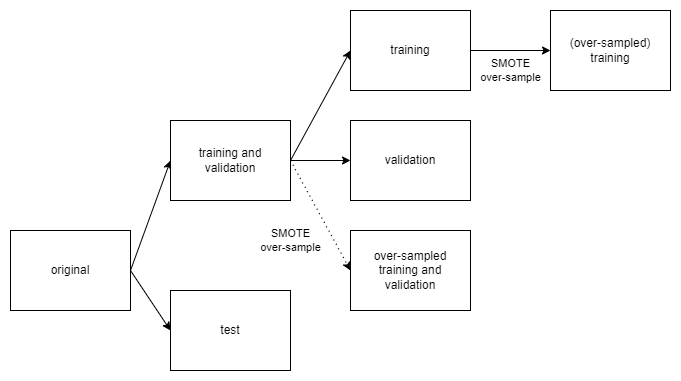


Figure 17. Datasets used in "Machine\_learning\_approaches\_(final)".

## Finding the best component models

Halving grid search cross-validation (halving grid search CV) is a method to search for the best combination of hyperparameters. "The search strategy starts evaluating all the candidates with a small amount of resources and iteratively selects the best candidates, using more and more resources" [3].

All the classifier classes of scikit-learn have their own default hyperparameters' values. Those default values are usually good in many cases. Therefore, in order to find the best values for hyperparameters possible without costing too much time, for each machine learning model, we choose one particular hyperparameter to tune, and then we search for every possible integer from half to double the default value of that hyperparameter using halving grid search CV. For instance, in random forest, we tuned the "n\_estimators" hyperparameter by trying all integers from 50 to 200 since the default value of "n\_estimators" is 100. Finally, we evaluate the generalization f2 score of the best model that the algorithm found on the validation dataset.

For random forest classifiers, after searching for the values of "n\_estimators" from 50 to 200, the best estimator we found has a number of trees that equals 157. Its generalization score is 0.6862745098039215. The results of the searching process are represented in Figure 18.

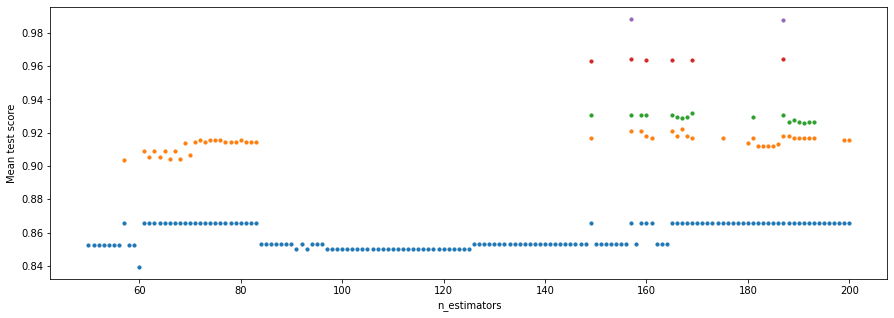


Figure 18. Results of all the halving grid search CV iterations while tuning "n\_estimators" of random forest.

For random forest only, we took a look at the "feature\_importances\_" attribute of the best model (with "n\_estimators=157") that the algorithm found above (Figure 19), and drew the conclusion that nearly all the attributes of the dataset are important, so that might be the reason why filtering the data features reduced the scores in the "machine learning models" section.

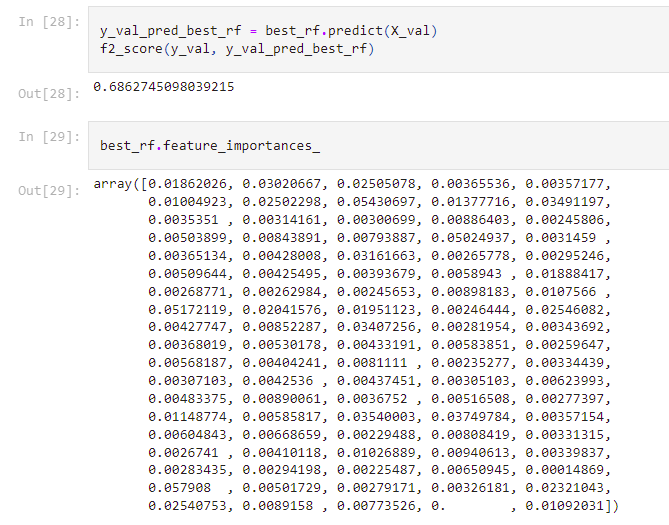


Figure 19. The "feature\_importances\_" attribute of the best random forest found.

For bagging classifiers, after searching for the values of "n\_estimators" from 5 to 20, the best estimator we found has a number of trees that equals 17. Its generalization score is 0.46391752577319584. The results of the searching process are represented in Figure 20.

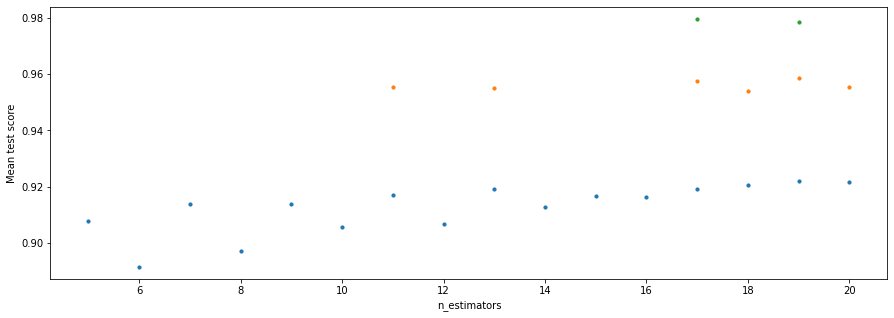


Figure 20. Results of all the halving grid search CV iterations while tuning "n\_estimators" of bagging.

For AdaBoost classifiers, after searching for the values of "n\_estimators" from 25 to 100, the best estimator we found has a number of trees that equals 97. Its generalization score is 0.5701754385964912. The results of the searching process are represented in Figure 21.

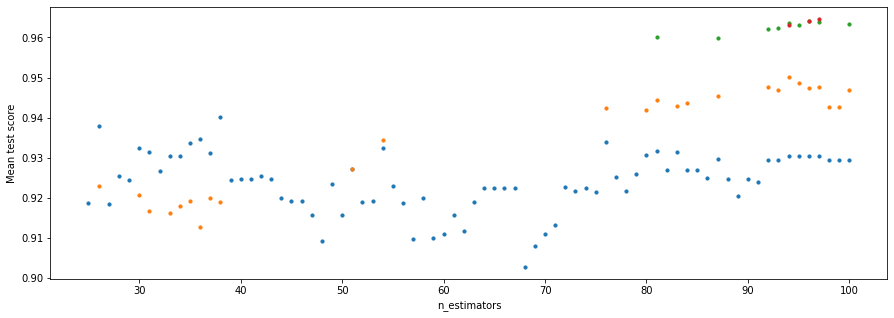


Figure 21. Results of all the halving grid search CV iterations while tuning "n\_estimators" of AdaBoost.

For gradient boosting classifiers, after searching for the values of "n\_estimators" from 50 to 200, the best estimator we found has a number of trees that equals 170. Its generalization score is 0.7017543859649122. The results of the searching process are represented in Figure 22.

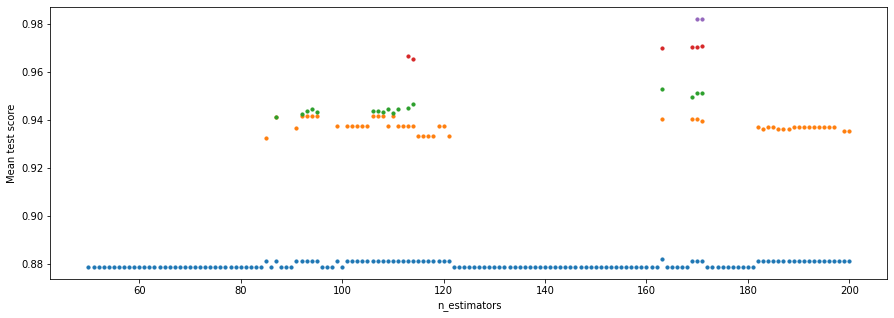


Figure 22. Results of all the halving grid search CV iterations while tuning "n\_estimators" of gradient boosting.

## Ensembling the best models into one model

To prevent biased predictions, we only use a simple voting classifier [3]. The four best models above are put into the classifier and retrained using the "over-sampled training and validation" dataset to get the final voting classifier. Finally, this classifier is tested on the test dataset to check its generalizability (in other words, check if its predictions will be good in the future). Its generalization f2 score is 0.6168831168831169 (Figure 23), which is pretty good for a hard-to-predict problem like this company bankruptcy prediction.



Figure 23. Testing result of the final voting classifier.

References

[1] Last, F., Douzas, G., & Bacao, F. (2017). Oversampling for imbalanced learning based on k-means and smote. arXiv preprint arXiv:1711.00837.

[2] Lemaître, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. The Journal of Machine Learning Research, 18(1), 559-563.

[3] Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., ... & Varoquaux, G. (2013). ECML PKDD Workshop: languages for data mining and machine learning. API Design for Machine Learning Software: Experiences from the Scikit-Learn Project, 108-122.

[4] Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825), 357-362.

[5] Wikipedia contributors. (2022d, July 7). Lasso (statistics). Wikipedia. <https://en.wikipedia.org/wiki/Lasso_(statistics)>

[6] Brownlee, J. (2021, January 4). Bagging and random forest for imbalanced classification. Machine Learning Mastery. Retrieved July 12, 2022, from <https://machinelearningmastery.com/bagging-and-random-forest-for-imbalanced-classification/>