Introduction to machine learning

Project 3

BERNARD Simon (s161519)
KLAPKA Ivan (s165345)
SCHOFFENIELS Adrien (s162843)

1st year of master in computer science and engineering
Academical year 2019-2020

1 Different approaches

(a) Pre-processing

A first important step to obtain a good classification is to have a good learning sample : enough relevant data as well as good set of features. In our case, it was important to see the fact that we were facing a very imbalanced dataset (ratio negative:positive of 1:36). In this subsection we will see the different approaches we tried on this provided dataset.

i) Fingerprint choice

One of the most interesting approach we tried was the choice of our molecule representation. Two things were to be determined.

- The method to convert molecule into bits
- The number of bits to represent the molecule

For the method, three representations have been tried.

Morgan : this was the initial representation (the one from the `toy_example.py` provided). It allows any number of bits.

Avalon : we were able to use this representation thanks to the `pyAvalonTools.GetAvalonFP()` method from `rdkit.Avalon`. It requires a multiple of 8 as number of bits

Daylight : we were able to use this representation thanks to the `rdFingerprintGenerator()` method from `rdkit.Chem`. However, this method can not be tuned over the number of bits which is set at 2048.

Switching between Morgan and Avalon was quite easy, and we often tried both (as soon as we started looking at other representations). However, increasing to thousands of bits while using the daylight representation, take too much time to be tried often. So as the daylight was not really satisfying, it was dropped after a few tests, and increasing to thousands of bits was reserved to already effective methods. Our final solution involves the Avalon representation with 3200 bits. Our model with the best score on the private leaderboard (that we unfortunately have not selected) involved a 2048 bits Morgan representation.

ii) Resampling

Another approach tried is the resampling (after splitting between training and validation set), which aims to help the model to face imbalanced datasets. We tried the four main existing methods available in the `imblearn` package.

Random over-sampling : it simply adds randomly new iterations of existing samples of the minority class.

Random under-sampling : it simply removes randomly samples of the majority class

SMOTE : it adds new synthetic iterations of the minority class, interpolating the new feature from existing samples of the minority class in order to have them with more minority neighbours than majority.

ADASYN : it works the same way as SMOTE, but adding a bit of randomness on the features to simulate in a better way the real world.

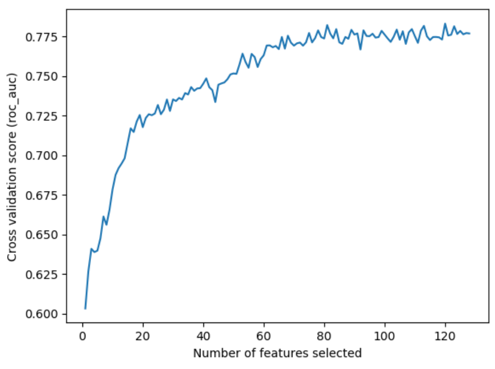
First we tried to combine random over-sampling and under-sampling. We wanted to not add too much times the same point, not obtaining a 1:1 proportion only with over-sampling to avoid overfitting, but also not decreasing to much the majority class with under-sampling (which is a loss of information). This did not really give us good result.
We then tried to do only over-sampling with random over-sampling, SMOTE and ADASYN. But this was not better either and also increased the computation time (nearly doubling the size of the learning set).
However, a lot of models with weights gives the possibility to tune the class weight, and it was most of the time better to choose a 'balanced' class weight[1] in order to counteract the imbalanced dataset.

iii) Feature selection

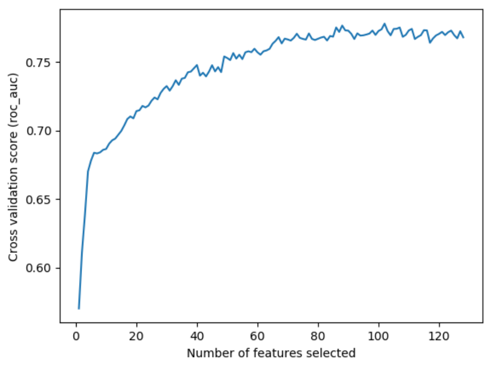
The last approach we tried is feature selection. The goal is to select the most important bits of the fingerprint representation. To have a good look on what is going on, we used a recursive feature elimination with cross-validation (with `RFECV()` from `sklearn`), which is nice for bioinformatics. We therefore have to choose an estimator.
For this estimator several choices were possible. The linear support vector machine is a nice model for that, however it took too many time to compute and we were not able to use it. The second choice was the decision tree which is nice to discriminate features, but it has the drawback to be too much related to the dataset. So in order to lower the variance due to the dataset, we used a random forest classifier as estimator. For the hyper-parameters, we took the best ones we had at this time which were with 300 estimators, a minimum impurity reduction of 0.0005 in order to reduce more the overfitting, and a balanced class weight to deal with the imbalanced dataset.
We then plot[2] the evolution of the AUC score when decreasing the number of features for different representations and different numbers of bits.

[1]One can also often custom the weights with a dictionary in those models (to take more or less than balanced weight), but it did not gave better results.

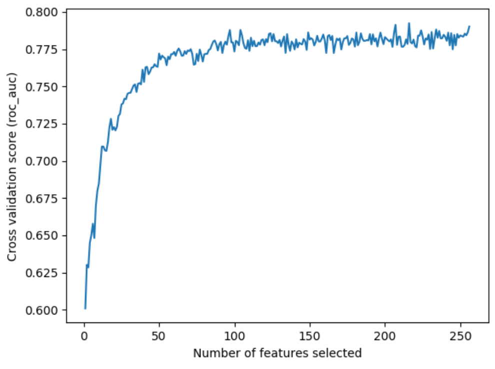
[2]we based our code to make those graph on `https://scikit-learn.org/stable/auto_examples/feature_selection/plot_rfe_with_cross_validation.html` (with some changes such as the scoring measure represented)



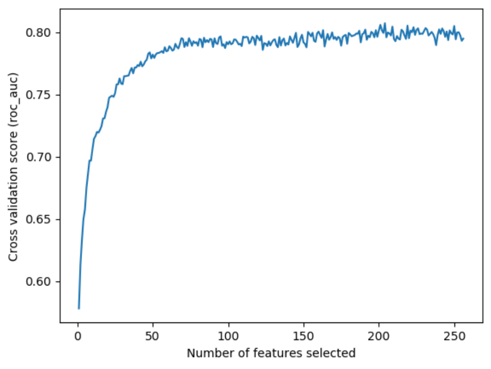
RFECV Morgan representation 128 bits



RFECV Avalon representation 128 bits



RFECV Morgan representation 256 bits



RFECV Avalon representation 256 bits

As it can be seen, removing features did not seem to increase the AUC score for the random forest. However, it is quite stable down to 50 features so we tried to reduce the number of feature for other method than the random forest (with `RFE` also from `sklearn`), but the results did not seem to improve. As the goal of the challenge was to have the best score without especially evaluating the computation time, we dropped the feature selection and favored a high number of bits. Doing recursive feature elimination for thousands of bits was not really possible due to our hardware limitations.

(b) Model selection

A lot of model have been tried. In order to compare them, we used the process presented in the section 3 to get the best of these model, then we were able to compare quite fairly their efficiency. The different methods are named in the section 2 in the table or in the note after.

(c) Joining of several models

In order to improve the results, we tried to join several methods. Therefore, we either did the mean of the results of different models, or we used the stacking method provided in the `vecstack` package. For a long moment we tried to stack different models, but without satisfying results. Then, we decided to compute the mean of two models (KNN and random forest classifier). This gave us a bad AUC estimation but a correct public result (the highest we ever achieved at this time of the competition). Therefore, we tried to stack these two models. This gave us even better results. We also tried to compute the mean of other models, for example the mean of KNN and a gradient boosting model, but without giving satisfying results on the public leaderboard. The best result we found was by computing the mean of KNN and a random forest classifier. Therefore, our last models are all based on a stacking of at least KNN and a random forest classifier models. The differences were the preprocessing techniques, the addition of other models to stack, the parameters of the stacked models and the last model to classify based on the new features given by the stacking technique.

2 Performance of the different approaches

In order to not overload the report, we will only discuss the performances of the methods that helped us to determine our final model. Indeed, we submitted a lot of models on Kaggle (41). Sometimes they were just tests to see if the estimated AUC score was correct, or if they were not overfitting the learning dataset.

The models we chose to include are listed and detailed below (they appear in chronological order) :

1. 5 nearest neighbors with feature selection;
2. 10 nearest neighbors with random oversampling (ratio 1:5) followed by random undersampling (ratio 1:3);
3. Random forest classifier with 1500 trees, balanced weights, 0.0005 minimum impurity decrease and entropy as criterion;
4. Support vector classifier with balanced weights and Hamming kernel;
5. Support vector classifier with balanced weights and a common substructures kernel;
6. Support vector classifier with balanced weights, a common substructures kernel and a 30 features selection;
7. 70 nearest neighbors with random oversampling (ratio 1:2) followed by random undersampling (ratio 1:1);
8. Stacking (10 nearest neighbors, logistic regression, gradient boosting) with gradient boosting as final model;
9. Gradient boosting classifier with learning rate of 0.1, 250 estimators and max depth of 6;
10. Mean of 3 models : gradient boosting classifier with max depth of 4, 10 nearest neighbors and random forest classifier with 250 trees;
11. Mean of 2 models : 10 nearest neighbors and random forest classifier with 250 trees;

12. Mean of 2 models : 10 nearest nearest neighbors and gradient boosting with max depth of 4;
13. 3 folds stacking of 2 models (random forest classifier of 450 trees with 20 nearest neighbors), with logistic regression as final model, with 160 bits Avalon's fingerprints;
14. 3 folds stacking of 3 models (random forest classifier with 450 trees, 20 nearest neighbors and support vector classifier with a common substructure kernel), with logistic regression as final model with 2048 bits Avalon's fingerprints;
15. 3 folds stacking of 3 models (random forest classifier with 450 trees, 20 nearest neighbors and support vector classifier with a common substructure kernel), with logistic regression as final model with 2048 bits Morgan's fingerprints;
16. 3 folds stacking of 3 models (random forest classifier with 450 trees, 20 nearest neighbors and support vector classifier with a common substructure kernel), with logistic regression as final model with 3200 bits Avalon's fingerprints;

All the models used are functions from the `Scikit-learn` open library, except for the gradient boosting that comes from the `XGBoost` package, and the stacking function that comes from the `vecstack` function.

Method	Validation AUC Score	Public AUC score	Private AUC score
1	0.73213	0.71687 (- 0.01526)	0.74731 (+0.01518)
2	0.74113	0.71967 (- 0.02146)	0.77474 (+0.03361)
3	0.79405	0.71678 (- 0.07727)	0.78226 (- 0.01179)
4	0.71546	0.64931 (- 0.06615)	0.67372 (- 0.04174)
5	0.71542	0.64920 (- 0.06622)	0.67361 (- 0.04181)
6	0.67361	0.61919 (- 0.05442)	0.70125 (+0.02764)
7	0.78904	0.69846 (- 0.09058)	0.80387 (+0.01483)
8	0.80113	0.69578 (- 0.10535)	0.77039 (- 0.03074)
9	0.77078	0.67015 (- 0.10063)	0.76624 (- 0.00454)
10	0.76039	0.71478 (- 0.04561)	0.79340 (+0.03301)
11	0.69591	0.74985 (+0.05394)	0.79995 (+0.10404)
12	0.76691	0.70707 (- 0.05984)	0.79008 (+0.02317)
13	0.78878	0.76273 (- 0.02605)	0.77802 (- 0.01076)
14	0.80168	0.78312 (- 0.01856)	0.80616 (+0.00448)
15	0.81583	0.75238 (- 0.06345)	0.82143 (+0.00560)
16	**0.80895**	**0.79984** (- 0.00911)	**0.80531** (- 0.00364)

Table 1 – Estimated, public and private AUC score, with deviation from estimation, of the models listed above

There is an important thing to notice on this table. As we can see, the private leaderboard seems to be more similar to our learning set as the different methods were in general nicely estimated. In the other hand, the public leaderboard, which has probably a class distribution much different from the two other, lead to often to worse AUC score. However, our best method seems to be quite robust to these kind of changes as it is the only one which have an estimated score that differs from both leaderboard by less than a percent (in addition to have the best public score).

Note that there are also a lot of models that we tested but that we never submitted to Kaggle since the submissions were restricted to 5 a day. Below is a list of other methods and techniques that we tried :

- Linear discriminant analysis;
- Adaboost;
- Gaussian naive Bayes;
- Stochastic gradient descent;
- Bagging;
- Stacking a lot of different models.

3 Parameters optimization and validation technique

In order to tune the parameters of the different models, we implemented a piece of code that tests several models in one execution, and prints the estimated AUC score for the different models tried. We used it for testing the same models but with different parameters and we kept the models with the parameters giving the best estimated AUC score. Once we had the best parameters, we re-estimated the AUC score with the process described in section 4.

This second estimation (mean + standard deviation) allowed us to have a strong feeling about the result it would have on the public leaderboard on Kaggle. If we had a good feeling (good mean and a low variance), we submitted our result on Kaggle, which determined if we kept the model or not. For example, we gave up the use of the gradient boosting model (from `XGBoost` and from `Scikit-learn`), which gave a really bad score on the public leaderbord regarding our estimated AUC (we realised at the end that our estimations using these models were not so bad when tested on the private leaderboard). This also encouraged us to used the nearest neighbor model and the random forest model. Indeed, these two models gave individually and coupled a good estimation and a good result on the public leaderbord. All the submissions we made at the end of the challenge used at least these two models.

4 Estimation of the AUC

In order to estimate the AUC score of our models, we divided randomly the learning set into a training set and a validation (test) set. The validation set was composed of 33 % of the learning set. Then, the models were fitted on the training set, then evaluated with the `roc_auc_score` function provided by the `scikit-learn` open-source library. For each model, this procedure was applied ten times. Then the mean and the standard deviation were computed using the `NumPy` package. The mean value is the final AUC score that we predicted and wrote on the submission files. The standard deviation allowed us to have a better feeling about the behavior of the model (eg. stability).

Our estimation technique gave quite good estimations. For example, our final model (the one selected to test on the private leaderboard) gave us the following results :

Estimated AUC : 0.80896

Public leaderboard AUC : 0.79984

Private leaderboard AUC : 0.80531

Since the public leaderboard was composed of more or less one third of the data, and the private leaderboard on the remaining data, we can compute our AUC on the whole dataset :

$$AUC = \frac{1}{3} \cdot 0.79984 + \frac{2}{3} \cdot 0.80531 = 0.80349$$

Regarding the small differences between the estimated AUC score, the "true" AUC score computed and the ones from the public and private leaderboards, we can state that our final model is quite robust and that the difference seems to small to be considered as overfitting. maybe another 10 simulations of the model would have changed the estimated AUC slightly.

Sources

- Theoritical course by P. Geurts and L. Wehenkel `http://www.montefiore.ulg.ac.be/~lwh/AIA/`
- `Scikit-learn`: `https://scikit-learn.org/stable/`
- `vecstack`: `https://github.com/vecxoz/vecstack`
- `XGBoost` : `https://xgboost.readthedocs.io/en/latest/`
- `rdkit` : `https://www.rdkit.org/`
- `https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/`
- `https://medium.com/@gurkamaldeol/predicting-environmental-carcinogens-with-logistic-regression-knn-gradient-boosting-and-7973f88eb8b3`
- `https://www.kaggle.com/rafjaa/resampling-strategies-for-imbalanced-datasets`
- `https://www.kaggle.com/residentmario/undersampling-and-oversampling-imbalanced-data`
- `https://academic.oup.com/bioinformatics/article/23/19/2507/185254`