Potential Pulsars Classification

Maxim Dergunov

2020

Introduction

Pulsar is a highly magnetized rotating neutron star that emits beams of electromagnetic radiation with a certain frequency (from 640 impulses per second to 1 impulse per 5 seconds). They are of interest not only in the field of theoretical physics. For example, a group of Russian physicists proposed to use pulsars as a time standard, due to their accuracy and the fact, that they will work for millions of years. Also, there are space orientation systems, based on pulsars location, which are being developed. And of course, there are golden plates installed on “Pioneer” and “Voyager” space probes, on which Earth location is shown relative to 14 nearest pulsars.

In this project, we are dealing with a binary classification problem. The main problem is imbalanced class distribution – there is only a little less than 10% of pulsars in that dataset. Data passed through the model will be reviewed by a human expert, so our goal is not only to build a model that correctly classifies the highest amount of objects but to build a model that made as few type 2 errors as possible.

Overview of approach

In this project, I was trying to find not only the best model but also the best way for data preprocessing. To do so I have created additional datasets:

- With the usage of Principal Component Analysis to decrease dimension amount;
- With the usage of Robust Scaling method;
- With usage of 3 methods of undersampling – random undersampling, NearMiss-3, and Condensed Nearest Neighbor;
- With the usage of 3 methods of oversampling – random oversampling, SMOTE and ADASYN;

And different combinations of these methods. Eight machine learning algorithms have been trained on that datasets - decision tree, random forest, extremely randomized trees, logistic regression, K-nearest neighbors, support vector machines, adaptive boosting, and gradient boosting. Each time they have been trained with standard hyperparameters and with the usage of grid search cross-validation.

The original dataset has been split to train and test parts in a 7:3 ratio. Resulted train set consisted of 12529 samples (with 1129 pulsars) and the test set got 5369 samples (with 510 pulsars).

Cohen’s Kappa has been chosen to be a quality metric. Precision, recall, F1-score, and Matthews correlation coefficient has also been collected.

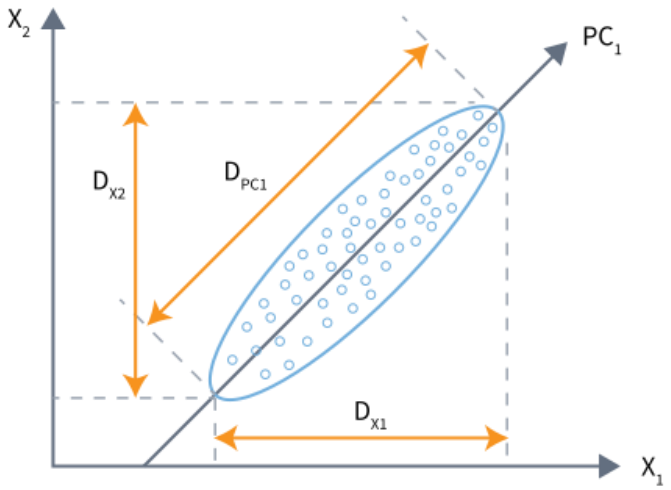
Below is an overview of the used methods, algorithms, and metrics.

Preprocessing

Principal Component Analysis

Principal component analysis is the main method of dimension reduction. The main idea is to find the principal components of the data. Geometrically speaking that components represent the directions of the data that explain a maximum amount of variance. The larger the variance carried by the line, the larger the dispersion along the line and the larger is dispersion along the line the more information it has.

In the picture, you can see the process of dimension reduction from 2-dimensional to 1-dimensional. The first principal component PC_1 is oriented along with the greatest elongation of the original data ellipsoid. That principal component PC_1 is less than the sum of original axes D_{x1} and D_{x2} which means that we were unable to express data variance with one principal component. So we'll have to build a second one, a third one, and so on until we can express the whole data variance.



Picture 1. Principal component analysis

Note that the variance along the principal components is not necessarily equal, thus the information that the principal component has can be more or less than others has. So, knowing how much information each component has we can keep only a given amount of components removing ones with the minimum amount of information.

Using PCA we can speed up calculations at the cost of the slight drop of quality.

Standardization

Standardization is a transformation that centers the data by removing the mean value of each feature and then scaling it by dividing features by their standard deviation. After that transformation, the mean of the data μ will be zero and the standard deviation σ will be 1.

Our data is unevenly distributed and has some outliers, so a regular standardization method may not work well. In that project, I used a Robust Scaler method. It removes the median and scales data using a quantile range:

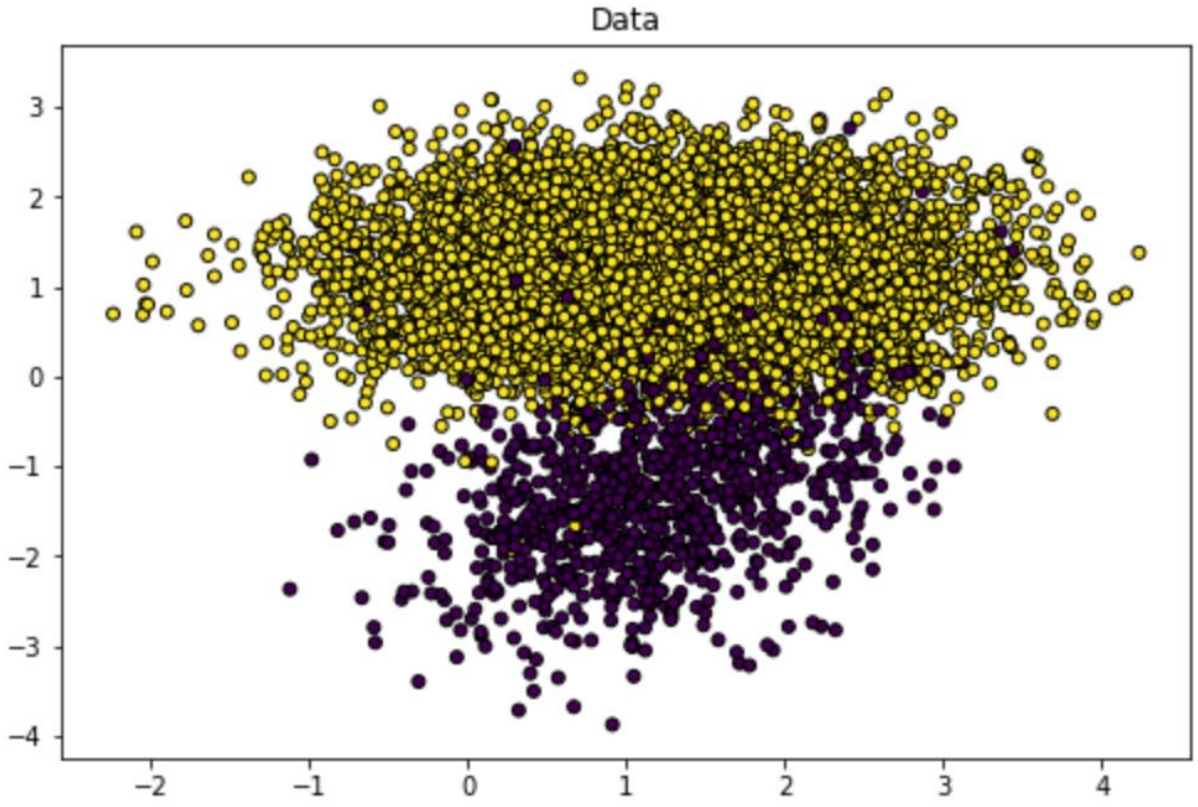
$$\frac{x_i - Q_1(x)}{Q_3(x) - Q_1(x)}.$$

A quartile is a type of quantile that divides the data into four equal parts. So the first quartile Q1 is the middle number between the smallest number and the median of the dataset. The third quartile Q3 is the middle value between the median and the biggest number in the dataset.

Data resampling

The classes in a given dataset are unevenly distributed – 16259 negative samples and 1639 positive samples. This is a problem because most machine learning algorithms have been designed to improve accuracy by reducing the error. Thus, they do not take into account the class distribution. To solve that issue I used three methods of undersampling, in which some samples of negative classes have been removed, and three methods of oversampling, in which synthetic samples of the positive class have been created.

To illustrate these methods with `sklearn.datasets.make_classification` I generated a new dataset. The parameters of the generator is: *`n_samples=10000, n_features=2, n_informative=2, weights=[0.1,0.9], class_sep=1.2.`*

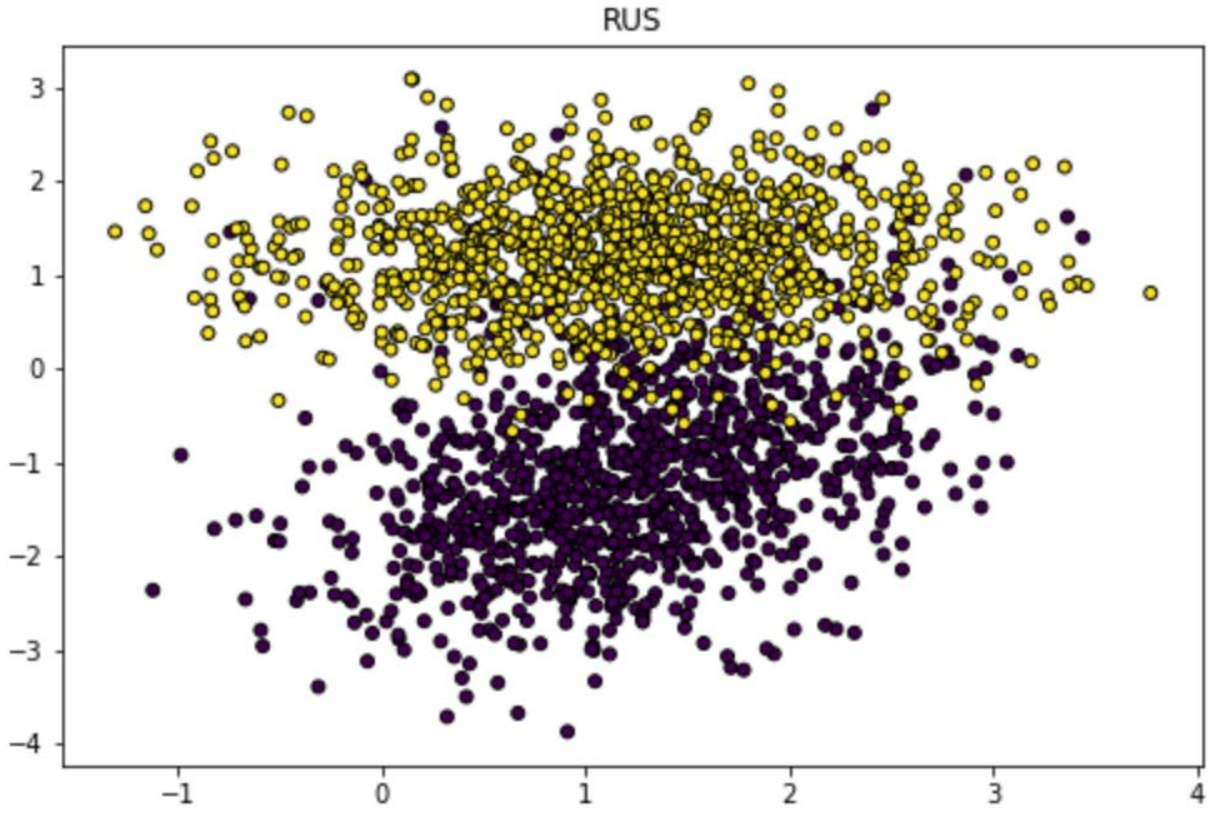


Picture 2. Synthetic dataset

Undersampling

Random Undersampling

Undersampling of majority class by randomly picking and removing negative samples. In that project, the number of positive and negative samples was equalized.

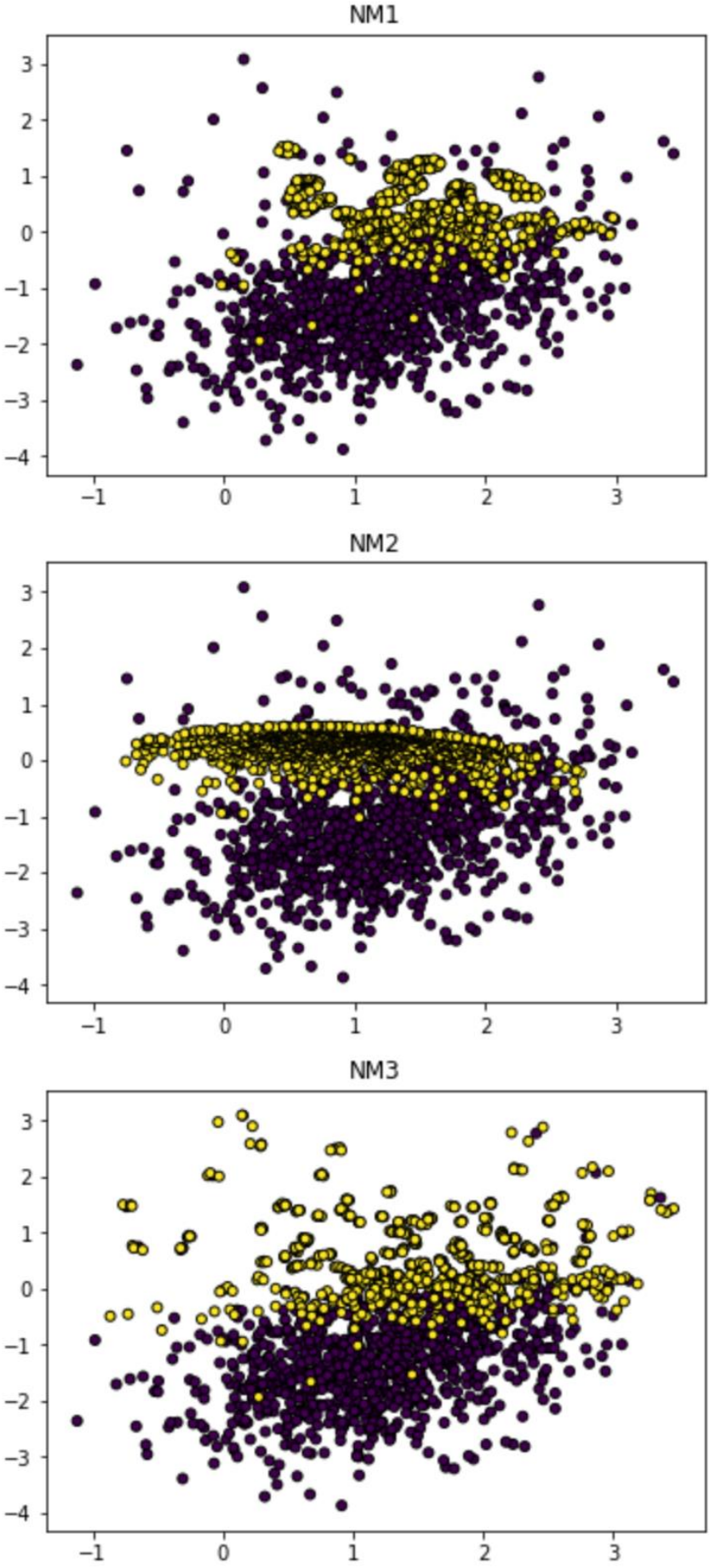


Picture 3. Random undersampling results

Near Miss

There are 3 versions of that algorithm. The first one keeps majority samples for which the average distance to the k closest minority samples is the smallest. In that project $k = 3$. The second version keeps majority samples for which the average distance to the k farthest minority samples is the smallest. Models working with these two versions showed poor results, so I will not use them in this project.

Near Miss-3 is a 2-step algorithm. First, for each positive sample, their n nearest neighbors will be kept. Then the negative samples selected are the one for which the average distance to the k nearest neighbors is the largest. In that project $n = 3$ and $k = 3$. Interesting thing is that in the resulting dataset there are more pulsars than non-pulsars!



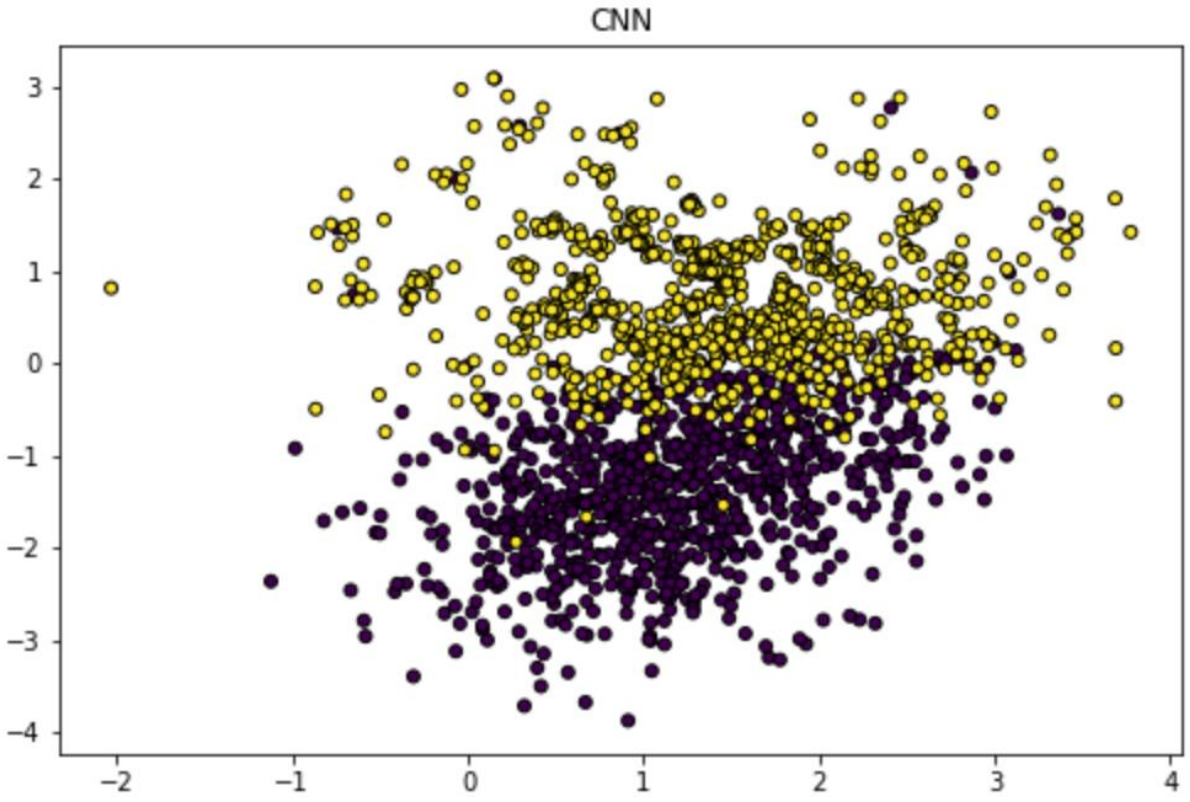
Picture 4. The results of the 3 versions of the NearMiss algorithm

Condensed Nearest Neighbors

This method uses a 1 nearest neighbor rule to iteratively decide if a sample should be removed or not. The algorithm works as follows:

1. Get all minority samples in a set C.
2. Add a sample from the targeted class (class to be under-sampled) in C and all other samples of this class in a set S.
3. Go through the set S, sample by sample, and classify each sample using a 1 nearest neighbor rule.
4. If the sample is misclassified, add it to C, otherwise do nothing.
5. Reiterate on S until there are no samples to be added.

The main flaw of that method is that it has very low speed due to reiteration through the whole dataset again and again. Also because of the random selection of the first sample results may vary each time this method is applied.

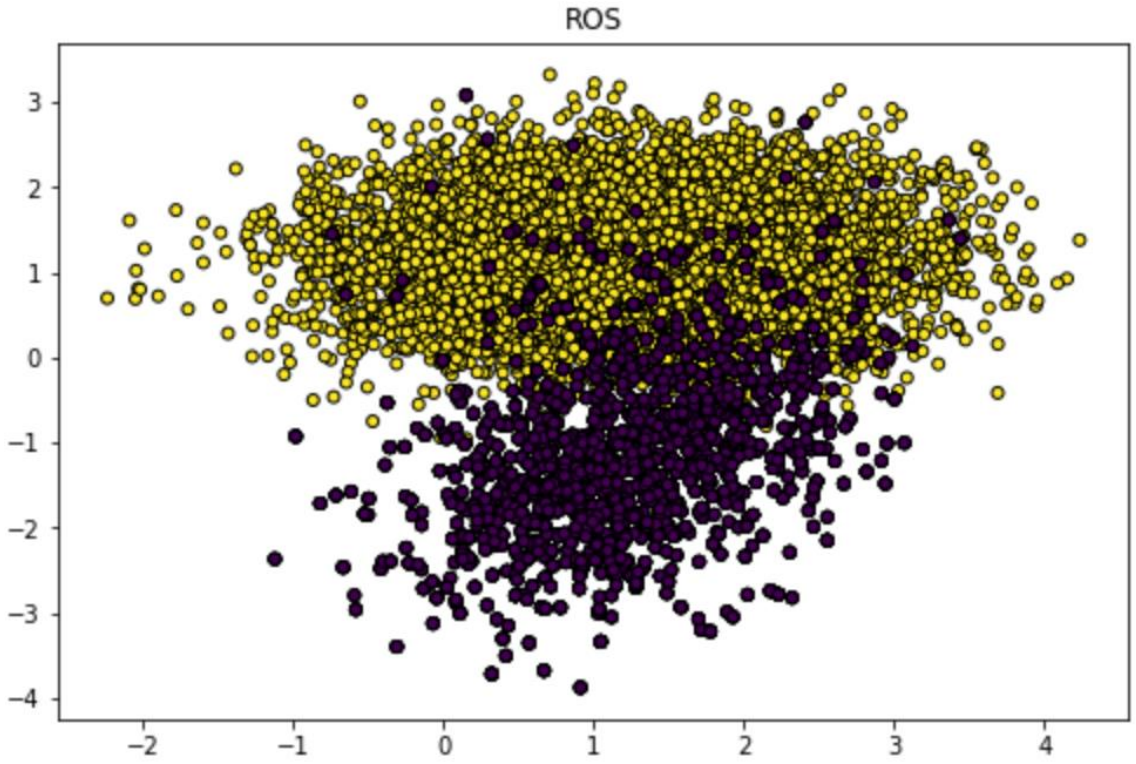


Picture 5. Condensed nearest neighbors results

Oversampling

Random Oversampling

This is a pretty naïve method that simply duplicates random minority samples. This method could lead to overfitting.



Picture 6. Random oversampling results

SMOTE (Synthetic Minority Oversampling TEchnique)

Instead of simple duplication of random samples, this method generates new samples based on existing ones: at first random samples are chosen, then the algorithm finds their nearest neighbors, and a new sample is generated between the original one and its neighbor.

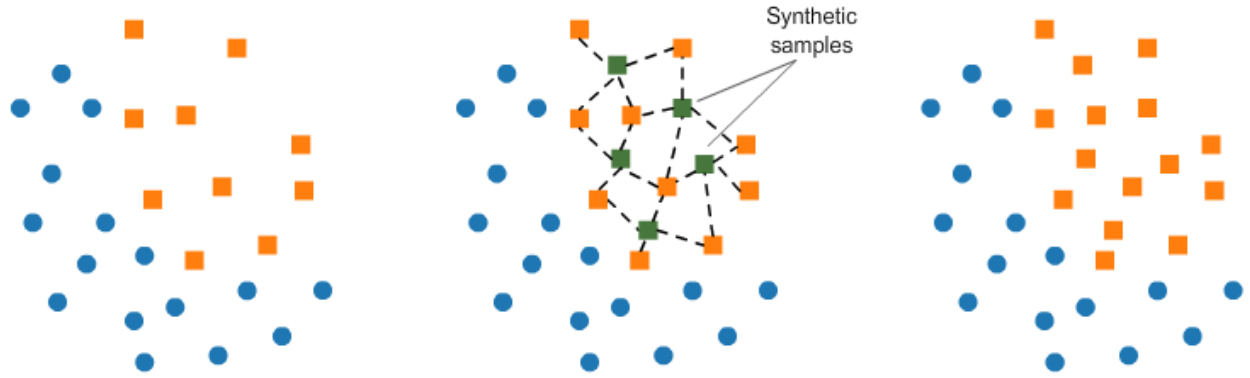
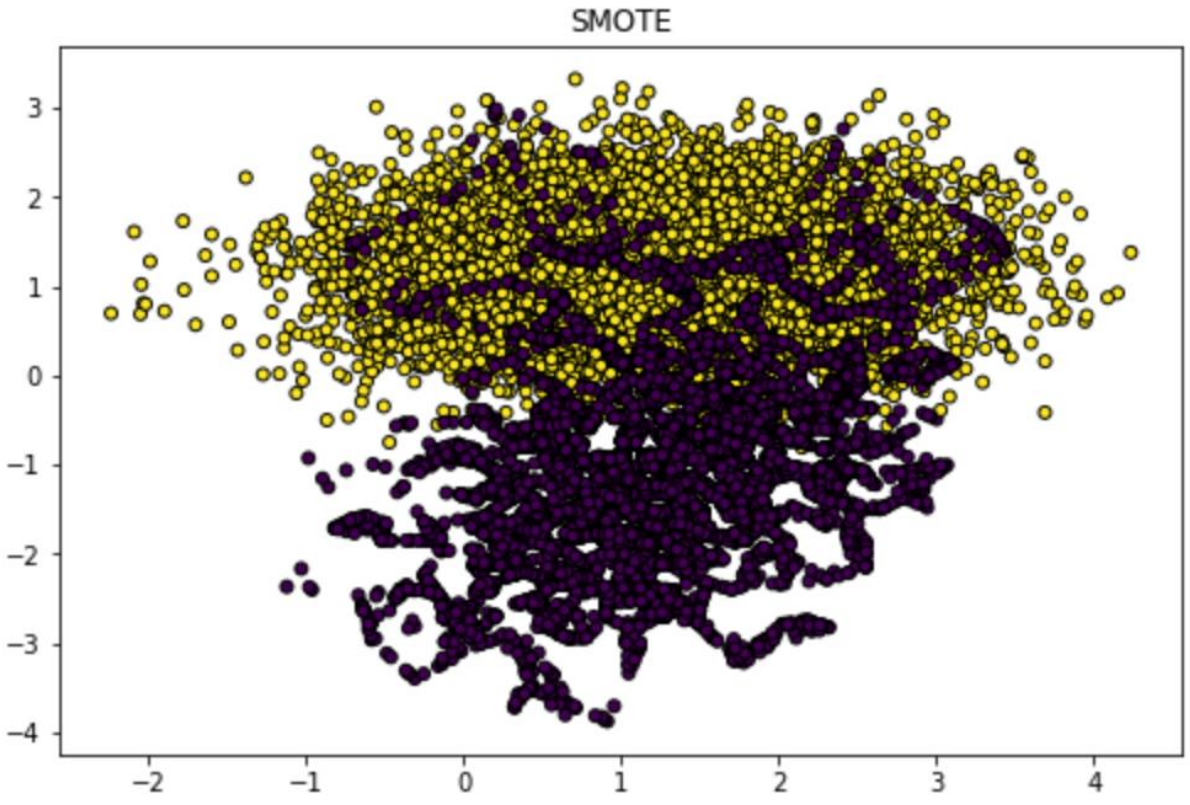


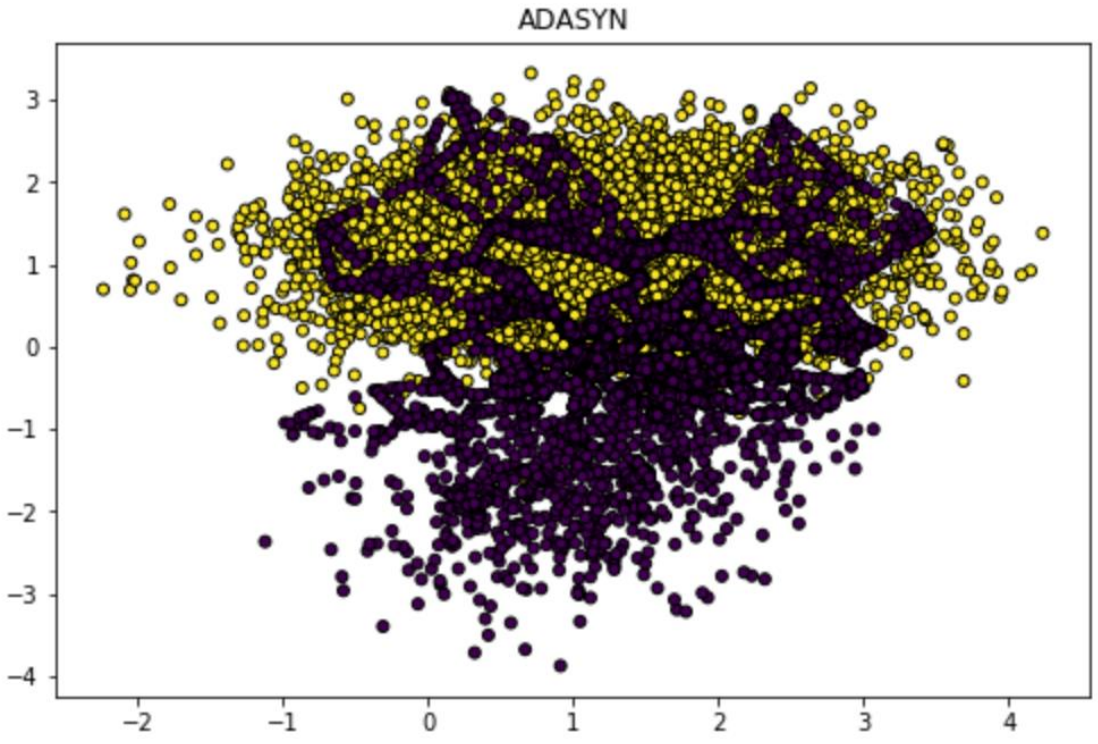
Рисунок 7. SMOTE algorithm illustration



Picture 8. SMOTE algorithm results

ADASYN (Adaptive Synthetic)

The last method, ADASYN, is based on SMOTE – the algorithm is finding the nearest neighbors too, but it takes into consideration a density distribution, or, in other words, majority and minority ratio. For samples, that are hard to classify, the amount of generated samples will be higher.



Picture 9. ADASYN algorithm results

Table 1. Processed data class distribution

Machine learning algorithms

K-Nearest Neighbors

This is one of the simplest methods. It assigns a class based on object's k nearest neighbors. For example, if $k = 3$ then for each object the algorithm will check classes of its 3 nearest neighbors. Geometric distance between two points in multidimensional space (also known as Euclidean Distance) is calculated by the formula:

$$d_{xy} = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}.$$

Support Vector Machine

This algorithm puts all feature vectors on an imaginary multidimensional plot (in our case its 8-dimensional) and draws an imaginary 8-dimensional hyperplane, that separates positive examples from negative examples. The hyperplane equation is given by two parameters: vector w, which has the same dimensionality as our dataset, and a real number b:

$$wx - b = 0,$$

where wx means $w_{(1)}x_{(1)} + w_{(2)}x_{(2)} + \ldots + w_{(D)}x_{(D)}$ and D is the number of dimensions of our dataset.

Now we can predict the classes like this:

$$y = sign(wx - b),$$

Where $sign$ is a mathematical operator that takes any value as input and returns +1 if the input is a positive number and –1 if the input is a negative number.

Thus the main objective of the algorithm is to find the optimum values for parameters w and b.

Logistic regression

Despite the name, the logistic regression is a classification algorithm. The name comes from statistics and is explained by the fact, that the mathematical formulation of logistic regression is similar to linear regression.

This method allows us to learn the probability of object affiliation to some class. To do that we need to find the best $wx_i + b$ parameters combination, but also to put it into a function whose codomain is (0, 1). In that case, we will assign a positive class if a function returns a value closer to 1, otherwise we will assign a negative class. One such function is a standard logistic function (also known as a sigmoid function):

$$f(x) = \frac{1}{1 + e^{-x}},$$

where e is an Euler's number.

Thus a logistic regression model looks like this:

$$f(x) = \frac{1}{1+e^{-(wx+b)}}.$$

Having found the parameters, we can use the algorithm. It will return a value p, where $0 < p < 1$. Thus we get the probability of sample affiliation to positive class and if it is more than some threshold (by default it is 0.5) then the sample is labeled with positive class.

Decision Tree

It is easy to illustrate this method with the reference to the "20 questions" game – one person thinks of a celebrity and the others are trying to guess by asking "yes" or "no" questions. What question should be first? The one that narrows down the number of remaining options the most. Asking "Is it Carl Sagan" will, in case of the negative response, remove only one person from the array of options, while asking "Is it a man" will reduce the possibility roughly by two. In other words, the "gender" feature divides the celebrity dataset much better than features like "sportsman", "grows bonsai", etc. Thus we approaching the concept of information gain based on entropy.

For a system with N possible states, Shannon's entropy is defined as:

$$S = -\sum_{i=1}^{N} p_i log_2 p_i,$$

where p is a probability function. The higher the entropy the more chaos there is in the system and vice versa. By measuring the entropy in split data we can measure that split efficiency:

$$IG(Q) = S_0 - \sum_{i=1}^{q} \frac{Ni}{N} S_i,$$

where Q is a condition, on which split occurs, q – the number of resul sets, N_i – the number of samples in that sets.

The algorithm finds the most effective split variants and it keeps splitting until there is nothing to split (which can lead to overfitting) or until the tree reaches the maximum depth given by the analyst.

Random Forest

Random Forest is one of a few universal algorithms. It can be used not only for classification problems but also for regression, clustering, anomaly finding, etc.

To build random forest consist of N trees:

- For each $n = 1, ..., N$:
    - Create a subset X_n from the original train set;
    - Build a decision tree b_n based on X_n subset;
        - Pick the best feature based on given criteria;
        - Split the subset by this feature to create a new tree level. Repeat this procedure until the subset is exhausted or until the tree reaches a given maximum depth;
        - For each split randomly pick m features from the n original ones and search for the best split only among these m features.

The final result is decided by the majority voting.

Extremely randomized trees

The main difference of this method from the random forest is in the splitting part. Just as in the random forest, we are using some subset of original features, but instead of looking for the optimal threshold, we use a random one for each feature. The best one among them is used to split the node.

Adaptive boosting (AdaBoost)

In this algorithm, we are building not the fully grown trees, but stumps or, in other words, trees with only one split. Each stump doing a terrible job classifying samples by himself, but in the resulting forest weights of the votes of that stumps is not equal. More than that, in the usual random forest each tree is created independently, but in an adaptive boosting algorithm each new tree takes into account the errors of previous trees.

First, the same weight $\frac{1}{number\ of\ samples}$ is assigned to each sample. Then we create the first stump that will look for the best split for each feature. We can measure split quality with Gini impurity:

$$Gini = 1 - \sum_k (p_k)^2,$$

where k is a number of classes and p_k is a probability of assigning that class. We choose the stump with the lowest Gini impurity.

After that, we need to calculate the stump's weight. To do that we have to calculate the overall error – the summarized weight of all misclassified samples. Having done that we can get stump's weight:

$$stump's\ weight = \frac{1}{2}\log\left(\frac{1-overall\ error}{overall\ error}\right).$$

Thus if the overall error is low, then the weight will be a big positive number. If the error is big, then the weight will be a negative number or in other words, if that stump will decide that the sample must be labeled as a pulsar, then its vote will be counted as a non-pulsar vote.

Now we need to redistribute sample weights. We will start with misclassified samples – they will get new weights according to the formula:

$$new\ weight = old\ weight \times e^{stump's\ weight}$$

We need to lower the weights for correctly classified samples:

$$new\ weight = old\ weight \times e^{-stump's\ weight}$$

Now we normalize these new weights (each one is divided by the total sum of weights). Then we create a new dataset with the same dimensions. We randomly choose samples from the previous dataset. Note that samples with higher weights have more chances to be chosen and they can be duplicated. Now once again we assign the same weights to our samples in a new dataset. Having duplicate samples will increase the cost of misclassification. We repeat this algorithm until datasets will stop changing or until we reach the given amount of trees.

Now, after our stump forest is completed, the voting takes place. Each stump makes his prediction and we count the summarized weights of stumps, that assigns pulsar label, and those, which assigns non-pulsar label. The biggest sum wins.

Gradient Boosting

In this project, I've been using the XGBoost library's gradient boosting implementation. The resulting forest should show us the sample's positive label probability.

First, we should set an initial prediction. By default, it is set to 0.5. After that, we should count a residual – the difference between the observation and an initial prediction. At that very first step, the residual will be equal to either 0.5 or -0.5. Now we build a tree that will predict these residuals. But we can't simply use these residuals as an output value – we have to transform them:

$$\frac{(\sum(Residual_i)^2}{\sum previous\ probability_i \times (1 - previous\ probability_i) + \lambda},$$

where λ – regularization parameter. We make this calculation for every leaf in our tree.

Now, when our tree is completed, we can update our probabilities. First, we need to transform our initial predictions to logarithmic form:

$$log\left(\frac{p}{1-p}\right),$$

For the default value 0.5, the result will be 0. Now we add it with the product of a tree's output value and learning rate. We put the result into a sigmoid function and get a new probability. Now we can get a new residual and start building a new tree.

We repeat this algorithm until the probabilities will stop changing or until we build a given number of trees. Finally, each tree makes its prediction. Each prediction is multiplied with the learning rate and they are summed with a logarithmic form of an initial prediction. The result is processed by sigmoid function and we receive a sample's probability of being a pulsar.

Choosing a metric

The most obvious and interpretable classification quality metric – accuracy – can't be used in this project because of the imbalanced class distribution. That's why I chose Cohen's Kappa. I have also collected precision, recall, f1-score, and Matthews correlation coefficient values.

First, it is important to understand what is the confusion matrix. In the case of binary classification classifier can make not only True Positive (TP, when its positive label both on prediction and observation) or True Negative (TN, when its negative label both on prediction and observation) predictions, but also a False Positive (FP, when the sample is positive on prediction, but negative on observation) and False Negative (FN, when the sample is negative on prediction, but positive on observation). The confusion matrix looks like this:

	$y = 1$	y = 0
$\hat{y} = 1$	*True Positive*	*False Positive*
$\hat{y} = 0$	*False Negative*	*True Negative*

Here $\hat{y}$ – algorithm's prediction and y – true label.

Precision

$$Precision = \frac{TP}{TP + FP}$$

This metric can be interpreted as a ratio of correct positive predictions to the total predicted positives.

Recall

$$Recall = \frac{TP}{TP + FN}$$

The ratio of correct positive predictions to the total positive examples.

F1-score

$$F = 2\frac{Precision \times Recall}{Precision + Recall}$$

F1-score can be interpreted as a weighted average of precision and recall.

Matthews Correlation Coefficient

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

This coefficient takes into account every value of the confusion matrix. MCC is in essence a correlation coefficient value between -1 and +1, where a coefficient of +1 means a perfect classifier, 0 is a random prediction and -1 is an inverse prediction.

Cohen's Kappa

Cohen's Kappa measures the agreement between the prediction and the observation. First, we need to calculate the observed agreement. For the binary classification it is counted as:

$$P_0 = \frac{TP + TN}{TP + TN + FP + FN}$$

Now we need to calculate the random agreement probability:

$$P_e = \frac{TP + FP}{TP + TN + FP + FN} \times \frac{TP + FN}{TP + TN + FP + FN} + \frac{FN + TN}{TP + TN + FP + FN} \times \frac{FP + TN}{TP + TN + FP + FN}$$

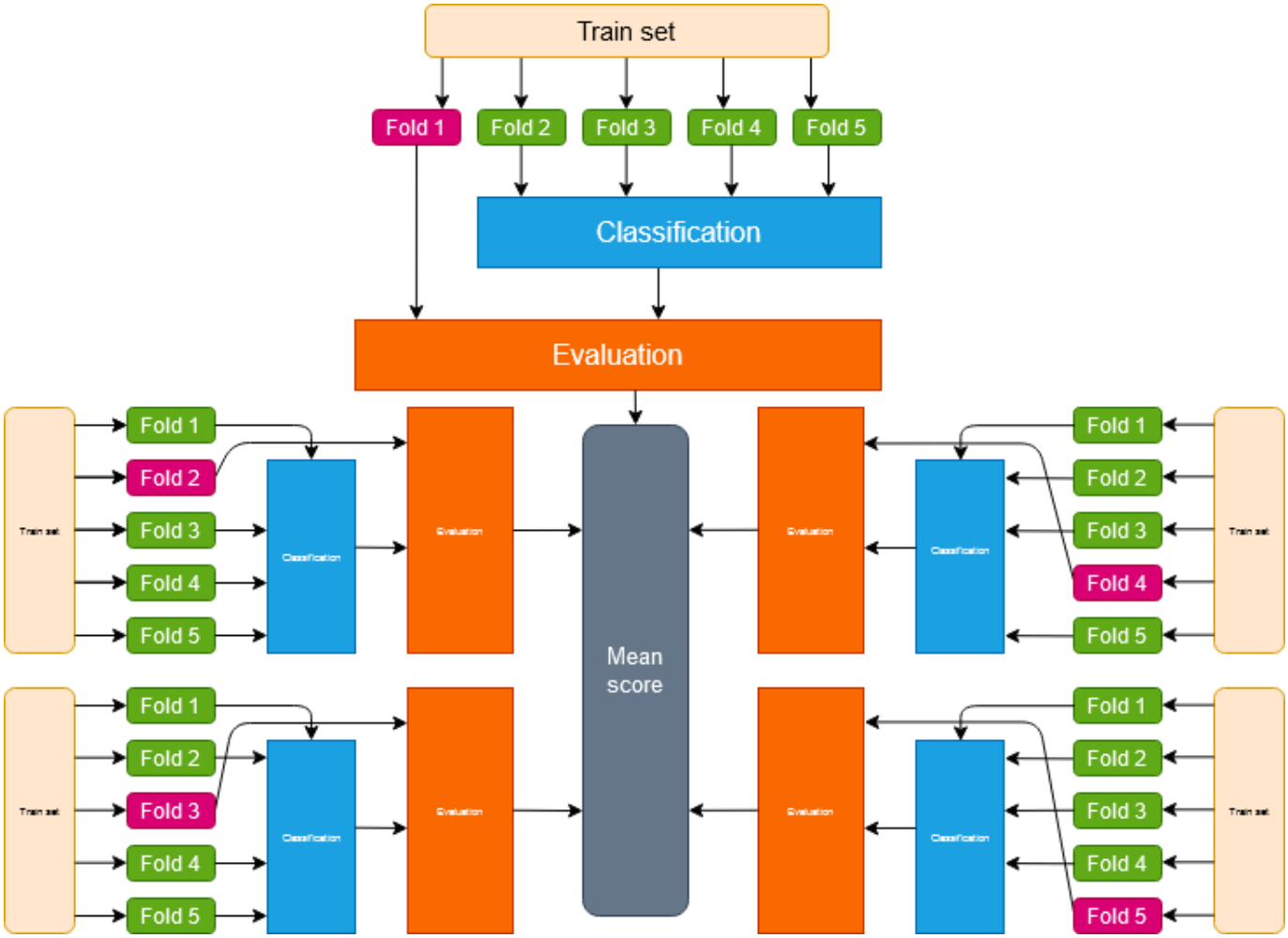
Finally, we can calculate Cohen's Kappa:

$$k = \frac{P_0 - P_e}{1 - P_e}$$

Just as Matthew's correlation coefficient, Cohen's kappa's value can range between -1 and 1.

Hyperparameters tuning

To tune hyperparameters, I used a grid search algorithm. First, for each algorithm, we assign a set of its hyperparameters and their values (or values range). We split the training dataset into five folds. Each fold is stratified, meaning that in each fold the class distribution is the same as in the full training dataset. Now for each hyperparameters combination, we build five models – in one we will use the first fold as a validation set and four other for training, in the second we will use the second fold for validation and four other for training, etc. We measure the quality of the model with a chosen metric. We use a mean metric value of all five folds to get a final evaluation of this combination of hyperparameters.



Picture 10. 5-fold cross-validation

Table 2. Hyperparameters grid

Support vector machine	*C = np.geomspace(0.8, 1.2, 5)* *kernel = ['linear', 'poly', 'rbf']* *gamma = ['scale', 'auto']*
Logistic regression	*penalty = ['l1', 'l2']* *C = np.linspace(0.1, 10, 50),* *solver = ['liblinear', 'newton-cg', 'lbfgs']* *max_iter = [50, 100, 500]*
Decision tree	*splitter = 'best', 'random'* *max_depth = range(3, 9)* *max_features = ['auto', 'log2', None]*
Random dorest	*n_estimators = [400, 200, 100]* *criterion = ['entropy']* *max_depth = [10, 9, 8, 7]* *max_features = ['auto', 'log2']*
Extremely randomized trees	*n_estimators = [50, 100, 500, 1000]* *criterion = ['entropy']* *max_depth = range(5, 9)* *max_features = ['auto', 'log2', None]*
Adaptive boosting	*n_estimators = range(500, 1100, 100)* *algorithm = ['SAMME', 'SAMME.R']* *learning_rate = np.linspace(0.01, 0.3, 5)*
Gradient boosting	*max_depth = [3, 4]* *learning_rate = [0.3]* *booster = ['dart']* *subsample = [0.7]* *colsample_bylevel = [0.7, 0.6],* *colsample_bynode = [0.7, 0.6]* *colsample_bytree = [0.7, 0.6]* *gamma = [0]* *n_estimators = [1000, 100, 50]* *tree_method = [‘exact’]* *‘rate_drop’ = [0.03]*
K-nearest neighbors	*n_neighbors = range(3, 15)* *weights ='uniform', 'distance'* *algorithm= 'ball_tree', 'kd_tree', 'brute'*

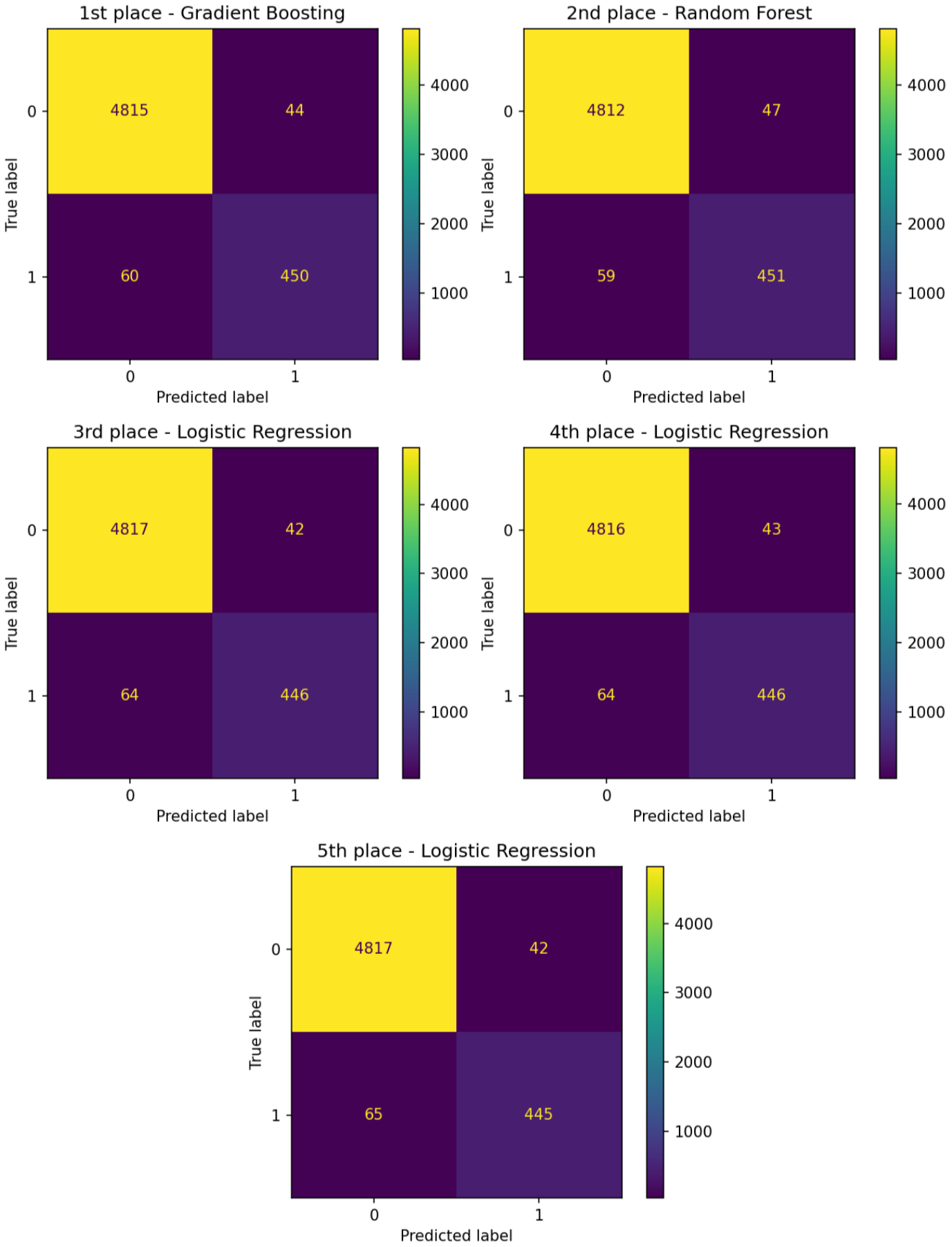
Results

I have built 312 models in total – 48 variants for each algorithm except gradient boosting and support vector machines. Due to their long training time, I decided to remove variants with PCA and without scaling. The reason is simple – other algorithms working with PCA or without scaling showed not very impressive results. For example, the best model using PCA took 66th place, and the best model working with unscaled data took 103rd place.

Below is a table showing the parameters and metrics of top-10 models and confusion matrices for these models.

Table 3. Top-10 models final results

Algorithm	Principal component analysis	Hyperparameters tuning	Scaling	Sampling	Weighted precision	Weighted recall	Weighted F1	Model's parameters	Matthews correlation coefficient	Каппа Коэна
Gradient boosting	False	True	True	SMOTE	0.980401	0.980630	0.980491	*max_depth = 4 learning_rate = 0.3 booster = 'dart' subsample = 0.7 colsample_bylevel = 0.7 colsample_bynode = 0.7 colsample_bytree = 0.7 gamma = 0 n_estimators = 1000 tree_method = 'exact' rate_drop = 0.03*	0.886	0.886
Random forest	False	True	True	SMOTE	0.980073	0.980257	0.980152	*n_estimators = 400 criterion = 'entropy' max_depth = 10 max_features = 'log2'*	0.884	0.884
Logistic regression	False	True	False	Random oversampling	0.979958	0.980257	0.980062	*penalty = 'l2' C = 0.98 solver = 'newton-cg' max_iter = 50*	0.883	0.883
Logistic regression	False	True	True	Random oversampling	0.979939	0.980257	0.980043	*penalty = 'l2' C = 3.74 solver = 'lbfgs' max_iter = 50*	0.883	0.883
Logistic regression	False	False	True	Random oversampling	0.979758	0.980071	0.979864	*penalty = 'l2' C = 1 solver = 'lbfgs max_iter = 100*	0.882	0.882
Logistic regression	False	False	True	SMOTE	0.979758	0.980071	0.979864	*penalty = 'l2' C = 1 solver = 'lbfgs' max_iter = 100*	0.882	0.882
Random forest	False	True	True	NM-3	0.979758	0.980071	0.979864	*n_estimators = 200 criterion = 'entropy' max_depth = 8 max_features = 'auto'*	0.881	0.882
Logistic regression	False	True	True	SMOTE	0.979671	0.980071	0.979752	*penalty = 'l1' C = 0.7 solver = 'liblinear' max_iter = 50*	0.881	0.880
Adaptive boosting	False	True	True	SMOTE	0.979444	0.979698	0.979544	*n_estimators = 1000 algorithm = 'SAMME.R' learning_rate = 0.08*	0.881	0.880
Extremely randomized trees	False	True	True	SMOTE	0.979655	0.980071	0.979714	*n_estimators = 500 criterion = 'entropy' max_depth = 8 max_features = None*	0.880	0.880



Picture 11a. Confusion matrices

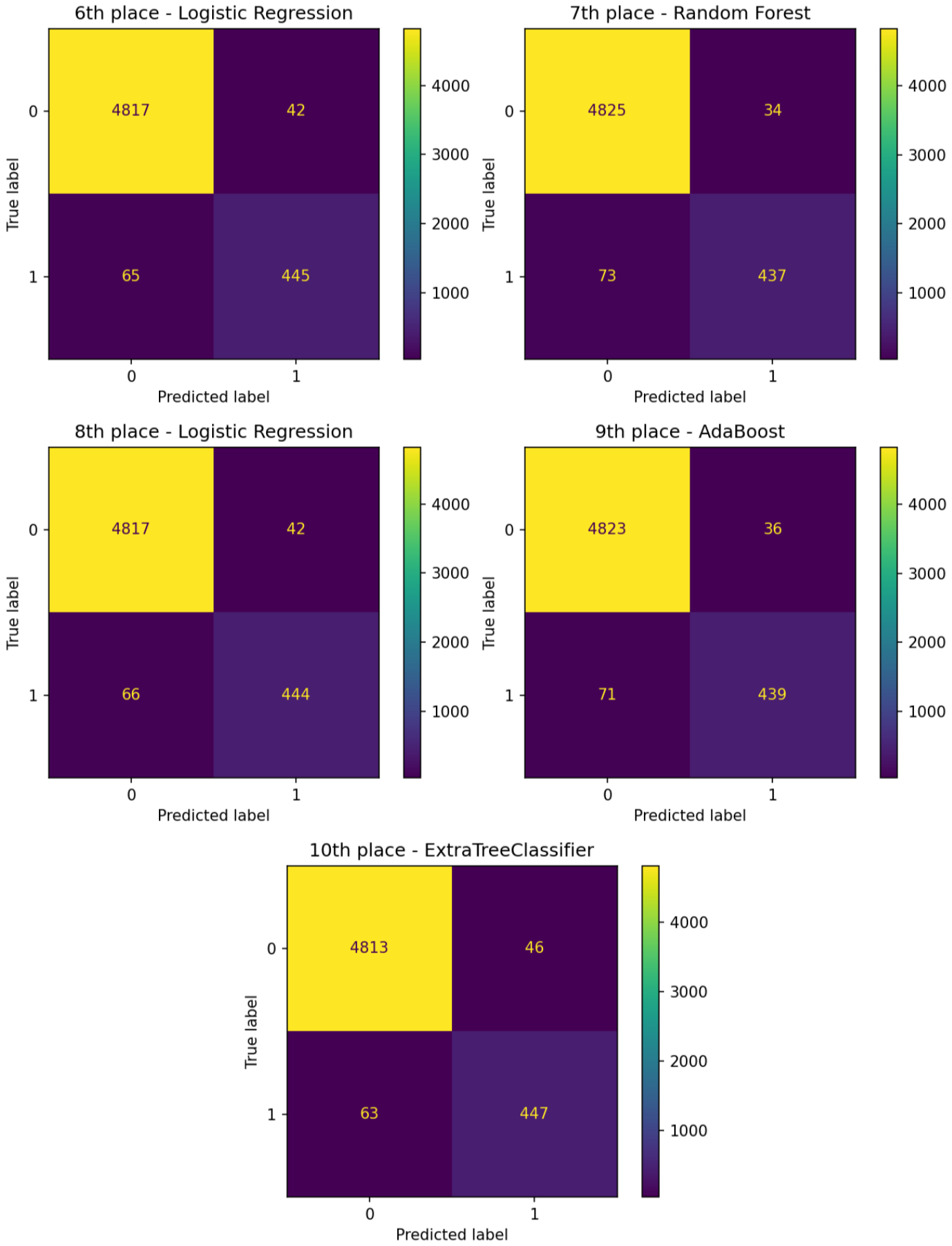


Рисунок 11b. Confusion matrices

As it was said at the beginning, it is important not to miss pulsars by assigning a negative label for them. Samples classified as pulsar will be checked by a human specialist, so we should pay more attention to the second row in a confusion matrix which shows the number of True Positives and False Negatives.

Models ranked first and second showed similar results. The gradient boosting model took first place by having correctly classified more negatives. The random forest that took second place made three more False Positives (which will be rejected later by a human) but found one more True Positive and in this project that would be much more important. Thus it is suggested to use the Random Forest model.

Conclusion

The main goal of this project was to build the best machine learning binary classification model based on the HTRU2 dataset. I gave an overview of Principal Component Analysis and Robust Scaler methods, various undersampling and oversampling methods, popular binary classification quality metrics, and eight machine learning algorithms. I used the grid search method to find the best hyperparameters for each model. The Random Forest using Robust Scaler and SMOTE undersampling proved to be the best.