Advanced Topics in Machine Learning - Semester Project

Aleksei Baidarov

June 20, 2018

1 Project introduction and task

The goal of the project is to analyze the APS Failure at Scania Trucks dataset and find suitable method or methods in order to predict if truck failures were caused by a failure of a specific component of the Air Pressure system (APS) or for reasons not related to APS. It is very important to be able to predict if a truck may fail because of the APS, because the pressurized air produced by the APS is used in such critical for the truck condition components as braking and gear changing. Being able to know which trucks may have APS failures would help Scania mechanics to double checks those trucks and avoid their breakdown, which would lead to much higher expenses that the cost of additional quality control.

We have two datasets: training dataset, which contains trucks, for which the class is known (positive ('pos') if the failure was caused by the APS, negative ('neg') otherwise), and test dataset, which contains trucks, for which we have to make our predictions. Since repairing a broken truck is much more expensive than a check in a workshop, we have two costs for misclassified trucks: one ($Cost\_1 = 10$) for trucks that are mistakenly classified as positive (false positives) and the other ($Cost\_2 = 500$) for trucks, that are mistakenly classified as negative (false negatives), which makes sense, because it is always better be safe than sorry. So our task is to make a prediction, that minimizes the total cost for the company:

$$Total\_Cost = Cost\_1 \cdot No\_Instances + Cost\_2 \cdot No\_Instances = \\ = 10 \cdot No\_FalsePositives + 500 \cdot No\_FalseNegatives \quad (1)$$

2 Dataset overview

We have two datasets: the training set and the test set, both of which are structured as CSV (comma separated values) formatted files. The training set contains 60 000 instances, the test set consists of 16 000 instances. Both sets have 171 features: all features, except class label, are numeric, so the only thing regarding the types of values that is required is to convert nominal (text) values of 'class' attribute to numeric. The structure of the training dataset is shown in the figure 1.

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	ee_008	ee_009	ef_000	eg_000
0	neg	76698	NaN	2.130706e+09	280.0	0.0	0.0	0.0	0.0	0.0	...	1240520.0	493384.0	721044.0	469792.0	339156.0	157956.0	73224.0	0.0	0.0	0.0
1	neg	33058	NaN	0.000000e+00	NaN	0.0	0.0	0.0	0.0	0.0	...	421400.0	178064.0	293306.0	245416.0	133654.0	81140.0	97576.0	1500.0	0.0	0.0
2	neg	41040	NaN	2.280000e+02	100.0	0.0	0.0	0.0	0.0	0.0	...	277378.0	159812.0	423992.0	409564.0	320746.0	158022.0	95128.0	514.0	0.0	0.0
3	neg	12	0.0	7.000000e+01	66.0	0.0	10.0	0.0	0.0	0.0	...	240.0	46.0	58.0	44.0	10.0	0.0	0.0	0.0	4.0	32.0
4	neg	60874	NaN	1.368000e+03	458.0	0.0	0.0	0.0	0.0	0.0	...	622012.0	229790.0	405298.0	347188.0	286954.0	311560.0	433954.0	1218.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
59995	neg	153002	NaN	6.640000e+02	186.0	0.0	0.0	0.0	0.0	0.0	...	998500.0	566884.0	1290398.0	1218244.0	1019768.0	717762.0	898642.0	28588.0	0.0	0.0
59996	neg	2286	NaN	2.130707e+09	224.0	0.0	0.0	0.0	0.0	0.0	...	10578.0	6760.0	21126.0	68424.0	136.0	0.0	0.0	0.0	0.0	0.0
59997	neg	112	0.0	2.130706e+09	18.0	0.0	0.0	0.0	0.0	0.0	...	792.0	386.0	452.0	144.0	146.0	2622.0	0.0	0.0	0.0	0.0
59998	neg	80292	NaN	2.130706e+09	494.0	0.0	0.0	0.0	0.0	0.0	...	699352.0	222654.0	347378.0	225724.0	194440.0	165070.0	802280.0	388422.0	0.0	0.0
59999	neg	40222	NaN	6.980000e+02	628.0	0.0	0.0	0.0	0.0	0.0	...	440066.0	183200.0	344546.0	254068.0	225148.0	158304.0	170384.0	158.0	0.0	0.0

60000 rows × 171 columns

Figure 1: APS Failure at Scania trucks training dataset structure.

The training set contains contains 59 000 negative instances and 1 000 positive instances, which means that the dataset is unbalanced and some sampling or balancing technique might be useful.

Many values are missing (have 'na' values), so some pre-processing is required. Also, since the number of features is relatively large, it would make sense to check if there are any redundant features and perform feature selection in order to make training faster and possibly improve the prediction results.

3 Approaches used to solve the task

We are going to use five algorithms for solving the task. All of them are known to work quite well for binary classification task. Since all our features are numeric, no data transformation is required (except of transformation of class labels to numbers).

1. Logistic Regression

   As a linear model, Logistic Regression has the following advantages:

   - Well learned and defined in terms of Maths
   - Works very well with large number of features, even if they are sparse
   - Works relatively fast compared to more complex model
   - Has only one parameter to tune (regularization parameter C)
   - Can make not linear boundary, if we use polynomial features

   The disadvantages of Logistic Regression are:

   - Does not work well, when the boundary is complex, non-linear
   - Despite very good theoretical background behind the algorithm, most practical implementations violate some of Gauss-Markov assumptions, that's why linear methods in practice often work worse than SVMs and ensemble methods for classification (and regression as well) tasks

   Logistic Regression is a case of Linear classifiers and is based on linear regression method and least squares methods. During training process it learn the weights of features using least square method. Then it estimates the value of function

   $$b(\vec{x}) = \vec{\omega}^T \vec{x}$$

   and uses sigmoid function

   $$\sigma(z) = \frac{1}{1 + exp^{-z}}$$

   to scale the values to 0 to 1 interval.

   To predict the class assignment for binary classification for each instance Logistic Regression estimates the probability that instance belongs to positive class as:

   $$p_+ = P(y_i = 1 | \vec{x_i}, \vec{\omega}) = \sigma(\vec{\omega}^T \vec{x_i})$$

2. Decision Tree

   Decision trees are very easy to understand and perceive by humans, even when there are many features, that's why we plan to apply this algorithm. It has the following advantages:

   - Easy to interpret, because of simple and clear rules of splitting nodes
   - Easy to visualize
   - Trains relatively fast compared to more complex algorithms
   - Has relatively small number of parameters to tune (but larger that Logistic Regression)
   - It works well not only with numeric features, but also categorical (although it's not important for our task, since all features are numeric)

   The disadvantages of the method are:

- Very sensitive to noise
- Prone to overfitting
- Not very stable: a little change in data can lead to significant change in decision tree
- Most practical implementations of the algorithm use greedy approach, which does not guarantee that global optimum is found

Decision tree uses greedy approach and maximizes Information Gain (in case of entropy criterion) - at each step it chooses the feature, whose split gives the maximum IG. Then the process is recursively repeated until the entropy reaches zero or some small values defined by user, if we want to avoid overfitting.

3. Random Forest

   Compared to single decision trees, Random Forest has several advantages that in most cases help to improve classification results, so in our opinion it makes sense to try an ensemble method and compare it to single decision tree methods and other non-ensemble algorithms.

   The advantages of Random Forests are:

   - Usually more accurate than linear algorithms
   - Almost insensitive to noise and outliers because of random sampling technique
   - Usually do not require much tuning of parameters and work well out of the box
   - Almost impossible to overfit, increasing the number of trees usually does not decrease the accuracy of the model

   But they still have some disadvantages:

   - Unlike single decision trees, random forests are not as easy to interpret
   - Work worse than linear methods when data has many sparse features
   - Still might overfit if there are a lot of noise or outliers

   Random Forest algorithm, that contains N trees has the following steps:

   - First for each $n = 1, \ldots, N$ it generates samples X_n using bootstrap.
   - Builds decision trees b_n for X_n (splitting the "best" feature on each step, while considering user defined parameters such as maximum depth of the tree, minimum number of instances in leaf nodes and others.
   - Final classifier looks like

     $$a(x) = \frac{1}{N} \sum_{i=1}^{N} b_i(x)$$

     which means "voting" for most popular decision, if we talk about classification tasks.

4. Support Vector Machines

   SVMs have their own advantages and disadvantages, but the reason we want to use this approach is to see how well (or not well) non-linear model can handle this task. Finding the best kernel and its parameters is still an open research question, so we are going to use radial-based function, polynomial and sigmoid kernels with different parameters. Of course, we will also use linear kernel, but we don't expect much difference compared to logistic regression in that case. Unlike other linear methods, SVMs maximize the margin between the classes, so they could be more robust than linear methods. Although logistic regression also allows using of non-linear features, SVMs usually are faster and more accurate. Just like linear methods, SVMs have small number of parameters to tune, if the kernel is known.

   SVMs try to find the hyperplane, that maximizes the margin $\frac{2}{||w||}$ (minimize $\frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \xi_i$, where ξ_i is a penalty variable and C is user defined parameter) between two classes, and have two important properties:

- Duality, which means that SVMs can be represented in a way in which data appears only within dot products
- They operate in a kernel induced feature space

$$f(x) = \sum_i \alpha_i y_i < \phi(x_i), \phi(x) > +b$$

which is a linear function in the feature space space implicitly defined by kernel K.

- Then the algorithm solves the optimization problem, which is to maximize

$$W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j < x_i, x_j >$$

subject to
$C \geq \alpha_i \geq 0$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$
with the solution:

$$w = \sum_{i=1}^{SV} \alpha_j y_i x_i^{SV}$$

5. LightGBM

   And just out of curiosity we are going to try LightGBM (Light Gradient Boosting Machine) algorithm in order to compare classic and time-tested algorithms to one of the recent ones. LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is supposed to be fast, memory-efficient, good at handling large datasets and have good accuracy. But since it is based on tree algorithms, it is prone to overfitting.

   Most decision tree learning algorithms grow trees by level (depth)-wise, but LightGBM grows trees leaf-wise (best-first). It will choose the leaf with max delta loss to grow. Holding number of leaves fixed, leaf-wise algorithms tend to achieve lower loss than level-wise algorithms.

   Leaf-wise may cause over-fitting when the dataset is small, so LightGBM includes the max_depth parameter to limit tree depth. However, trees still grow leaf-wise even when max_depth is specified.

   Detailed description of the algorithm can be found in the following webpage:

   https://github.com/Microsoft/LightGBM/blob/master/docs/Features.rst

4 Implementation, evaluation and comparison of results

4.1 Importing libraries and loading datasets

Before we start working with the data, we need to import into our project some libraries or modules from libraries, that help us work with data efficiently. The most important libraries we are going to use are:

- Pandas - for efficient work with dataframes (data tables).
- Scikit-learn - contains implementations of numerous machine learning algorithms, evaluation metrics and so on.
- Matplotlib - for plotting graphs.
- Seaborn - for visualization of confusion matrices, plots and so on.
- NumPy - makes work with Maths easier (working with matrices, statistics and so on).

After that we can load our datasets and begin data analysis.

4.2 Pre-processing data and exploratory analysis

Pre-processing is a very important step of data preparation, because very often data contains missed values, outliers, errors, which sometimes make it impossible to use the data as is or influence the accuracy of prediction in a negative way. Even very complex models with fine-tuned parameters might produce worse results than simple models when data is cleaned and pre-processed correctly.

Since all the features are numeric, it makes sense to convert class labels from strings to numbers ('pos' -> 1, 'neg' -> 0). Also we can notice, that the training dataset is very unbalanced, which usually has negative impact on prediction. That means that some sampling of the data or other balancing technique might be required.

In our case there are a lot of missing values (see Figure 2). Simply removing rows with missing values usually is not a good idea, because we lose some data doing so. Furthermore, in our case, as you can see on the Figure 2, some columns have very large proportion of missing data, for example, 'ab_000' has more than 46 000 'na' values out of 60 000, hence dropping instances with missing values would lead to dropping almost all instances from our dataset. Another approach to get rid of missing data is to drop columns that have high proportion of missing values, but again, we lose some data in that case and it might turn out that there is a correlation between those columns are class label. Sometimes drawing a correlation matrix (see figure 3) helps to determine if there are any interdependencies between features and find the columns that could be deleted.

```
In [8]: df_train.isnull().sum()

Out[8]: class         0
        aa_000        0
        ab_000    46329
        ac_000     3335
        ad_000    14861
        ae_000     2500
        af_000     2500
        ag_000      671
        ag_001      671
        ag_002      671
```

Figure 2: Checking how many missing values training dataset has.

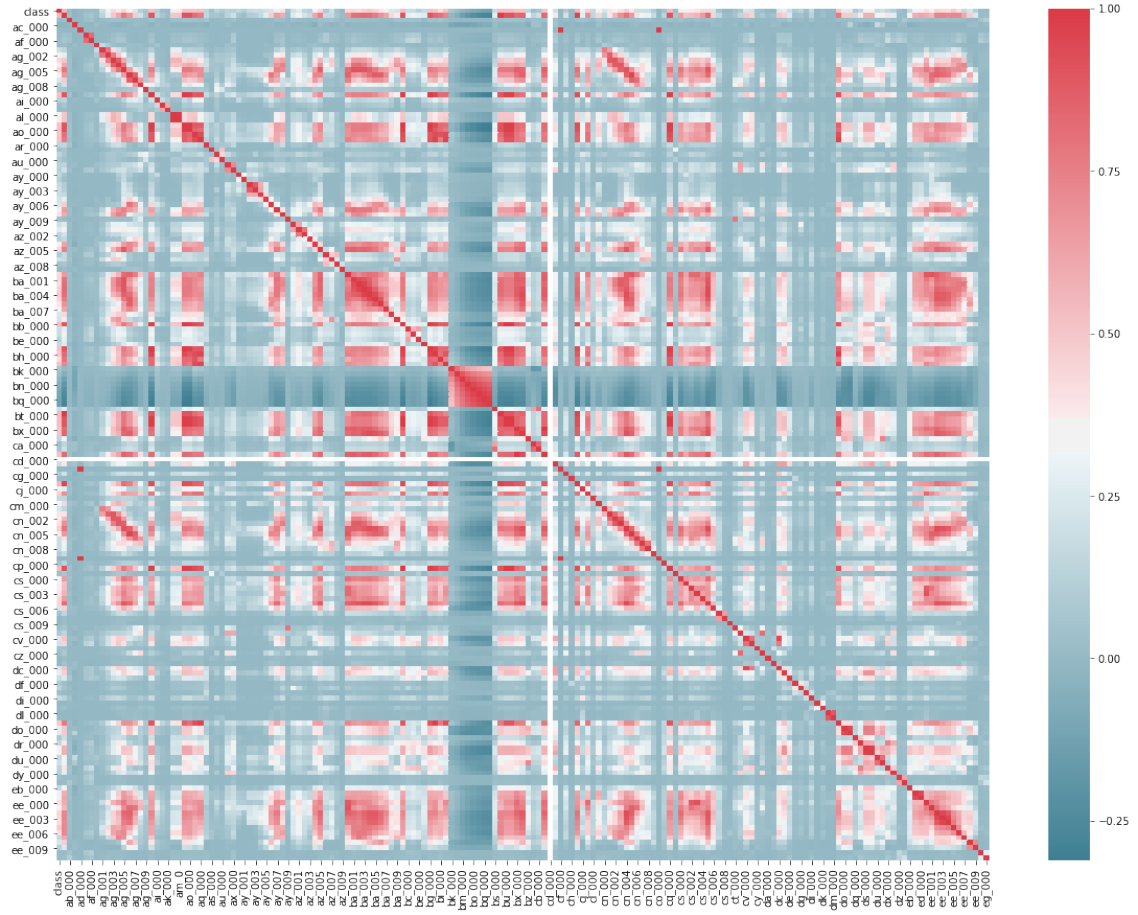

Figure 3: Correlation matrix.

In our case, considering quite big number of features, it is hard to see any redundant or useless (that depend on other features) features, so we can not see any obvious candidates for dropping, so we will use another technique to get rid of missing values: filling 'na' values with mean value of the respective column. That way we don't throw away any data without changing the mean values of columns. But of course that means that we make some assumptions about those values, so of course other approaches or heuristics can be useful and it is a trial and error process to find the best approach to handle missing data. After that we are ready to work with the data and can proceed to the next stage. After all pre-processing steps the dataset looks as shown in figure 4.

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005
0	0	76698	0.713189	2.130706e+09	280.000000	0.0	0.0	0.0	0.0	0.0	...	1240520.0	493384.0	721044.0	469792.0
1	0	33058	0.713189	0.000000e+00	190620.639314	0.0	0.0	0.0	0.0	0.0	...	421400.0	178064.0	293306.0	245416.0
2	0	41040	0.713189	2.280000e+02	100.000000	0.0	0.0	0.0	0.0	0.0	...	277378.0	159812.0	423992.0	409564.0
3	0	12	0.000000	7.000000e+01	66.000000	0.0	10.0	0.0	0.0	0.0	...	240.0	46.0	58.0	44.0
4	0	60874	0.713189	1.368000e+03	458.000000	0.0	0.0	0.0	0.0	0.0	...	622012.0	229790.0	405298.0	347188.0
5	0	38312	0.713189	2.130706e+09	218.000000	0.0	0.0	0.0	0.0	0.0	...	388574.0	288278.0	900430.0	300412.0
6	0	14	0.000000	6.000000e+00	190620.639314	0.0	0.0	0.0	0.0	0.0	...	168.0	48.0	60.0	28.0
7	0	102960	0.713189	2.130706e+09	116.000000	0.0	0.0	0.0	0.0	0.0	...	715518.0	384948.0	915978.0	1052166.0
8	0	78696	0.713189	0.000000e+00	190620.639314	0.0	0.0	0.0	0.0	0.0	...	699290.0	362510.0	1190028.0	1012704.0
9	1	153204	0.000000	1.820000e+02	190620.639314	0.0	0.0	0.0	0.0	0.0	...	129862.0	26872.0	34044.0	22472.0
10	0	39196	0.713189	2.040000e+02	170.000000	0.0	0.0	0.0	0.0	0.0	...	198386.0	99614.0	215734.0	189966.0
11	0	45912	0.713189	0.000000e+00	454.000000	0.0	0.0	0.0	0.0	0.0	...	495400.0	278660.0	544710.0	438256.0

Figure 4: Training dataset after pre-processing.

You can find the code for the previously explained steps in the part 1 of the source code: "Loading libraries and Pre-processing".

4.3 Feature selection

Since our data has relatively large number of features, it may be useful to execute feature selection procedure. Effective feature selection usually leads to two effects: reduced time for training (because we have fewer features than before) and improved accuracy (large number of features may lead to overfitting and building too complex model). There are several techniques to reduce the number of features: using univariate feature selection, removing features with variance, using recursive feature elimination, using principal component analysis and others. We used two first methods in our project:

1. Univariate feature selection

   This method selects best k (defined by user) features based on univariate statistical tests and then transforms the dataset accordingly. We experimented with different values of k, measured recall after training Logistic Regression and compared it with recall value without feature selection. For some reason not only prediction performance did not improve, but also the training time increased.

2. Removing features with low variance

   This method removes all features, whose variance doesn't meet some threshold (defined by user). We used the same approach as with Univariate feature selection and got the same result: performance and training time did not improve.

These two methods were not worth it, so we decided to not use any feature selection in our project. For the code see Part 2 of source code: "Experimenting with feature selection".

4.4 Applying machine learning algorithms and evaluating results

Before the start of training process we scaled the data using *StandardScaler* from *sklearn.preprocessing* module. It is not a necessary step, but since the dataset has many large numbers, scaling is useful in our case, because it helps to make computations faster and usually doesn't affect the performance of models in terms of prediction quality.

Also we used parameter *class_ weight*='balanced' for all classifiers (*is_ unbalance*=True for LightGBM) to balance the classes, since our dataset is unbalanced.

As an evaluation measure we used recall:

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

because the higher recall means the lower the number of FN and we want to have the number of FN as low as possible since the cost of FN is 50 times higher than the cost of FP.

4.4.1 Logistic Regression

To implement Logistic Regression, we used *LogisticRegression* classifier from *sklearn.linear_ model* module. The basic idea of training logistic regression was:

1. Defining a baseline model: training classifier with default or arbitrarily chosen hyperparameters and evaluating the baseline model.

2. Tuning classifier hyperparameters using 5-fold cross-validation search (*LogisticRegressionCV* - grid search optimized for logistic regression) for training set and recall as evaluation measure.

   The only parameter to tune for logistic regression is the regularization (inverted) parameter C: the higher the value of C, the "weaker" is the regularization and vice versa; higher values for C correspond to higher model capacity/complexity.

   we have tried 9 values for C: [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000] and the best value was $C = 0.001$.

3. Fitting the model with best hyperparameters, applying it on data set, computing confusion matrix and calculating total costs. After training the model with $C = 0.001$ and making prediction for the test set we have got the following results:

   Recall = 94.13%

   Total_Costs = 14840

   The confusion matrix is shown in figure 5.

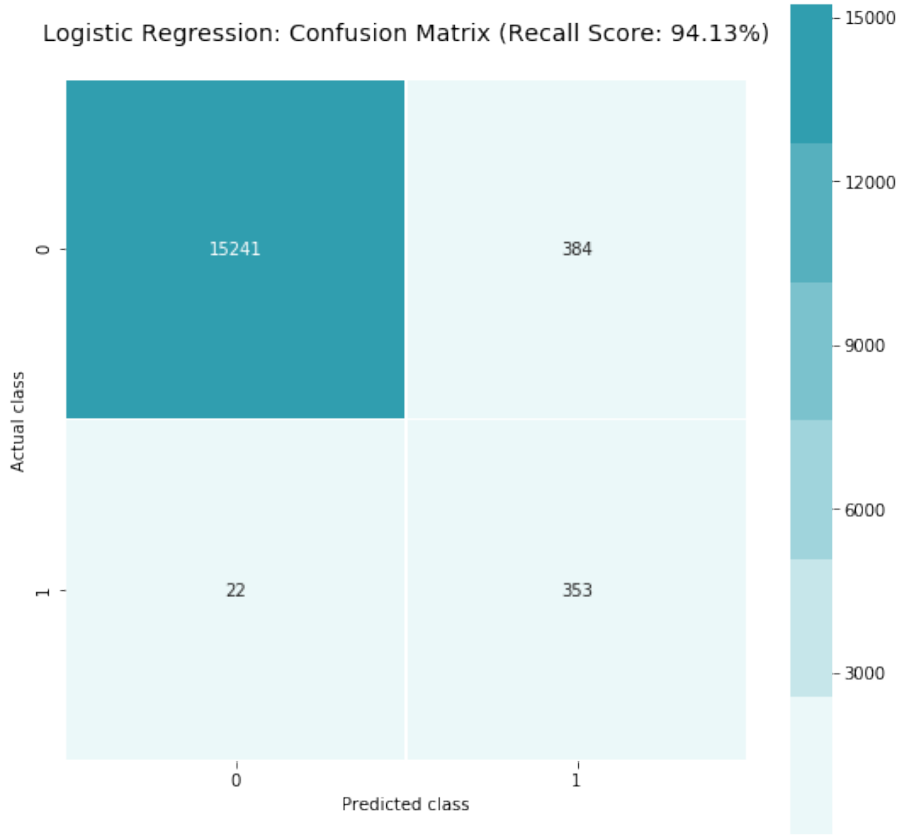

Figure 5: Confusion matrix (Logistic Regression).

You can find the code for Logistic Regression training in source code part 3.1: "Logistic Regression".

4.4.2 Decision Tree

To implement the Decision Tree algorithm we have used *DecisionTreeClassifier* from *sklearn.tree* module. The general sequence of steps was the same as in the previous algorithm:

1. Defining a baseline model: training classifier with default or arbitrarily chosen hyperparameters and evaluating the baseline model.

2. Tuning classifier hyperparameters using 5-fold cross-validation grid search for training set and recall as evaluation measure.

   Decision Tree has several important hyperparameters which we tuned during cross-validation:

   - *max_ depth* - the maximum depth of the tree
   - *max_ features* - the number of feature to consider when looking for a best split
   - *min_ samples_ leaf* - the minimum number of samples required to be in a leaf node
   - *criterion* - the function to measure the quality of a split (for example, 'entropy' measures information gain, 'gini' measures impurity)

There are a few more, but we have tuned only these four. The values we used for grid search CV were:

max_ depth: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, None]

max_ features: ['sqrt', 'log2', 20, 30, 50, 100, 150, 170, None]

min_ samples_ leaf: [1, 3, 5]

criterion: ['entropy', 'gini']

The best parameters that were found: *max_ depth* = 3, *max_ features* = 'sqrt', *min_ samples_ leaf* = 1, *criterion* = 'entropy'.

3. Fitting the model with best hyperparameters, applying it on data set, visualizing the tree, computing confusion matrix and calculating total costs.

   After training the model with these parameters and applying it on the test set we got the following results:

   Recall = 98.13%

   Total_Cost = 19010

   The confusion matrix is shown in figure 6.

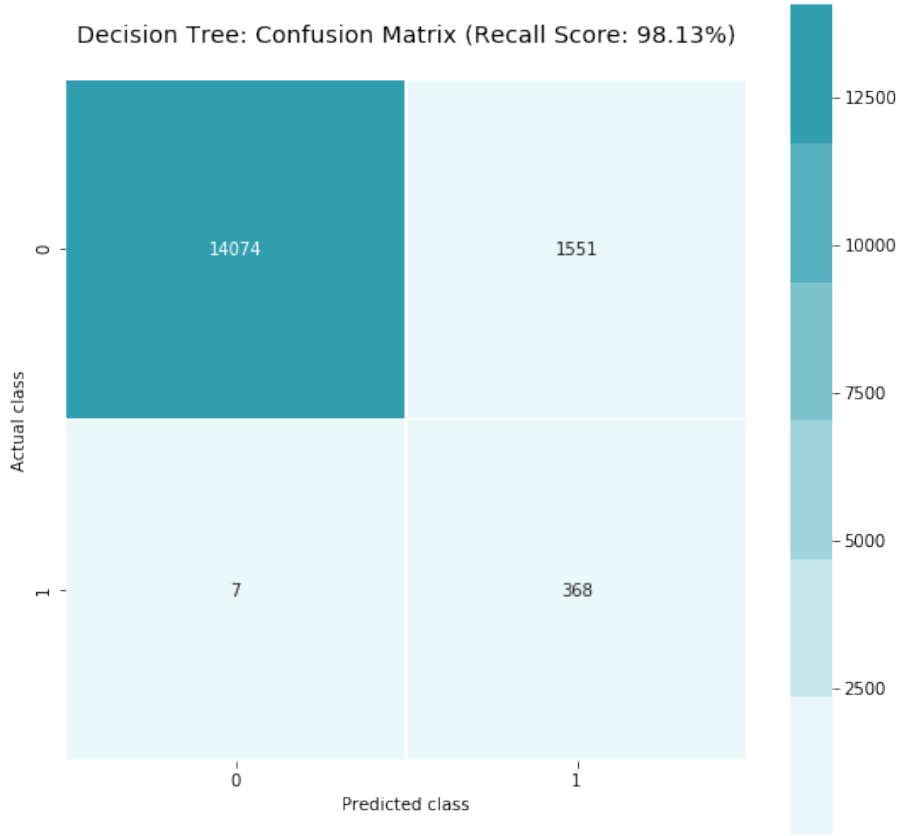

Figure 6: Confusion matrix (Decision Tree).

The decision tree itself is shown in figure 7.

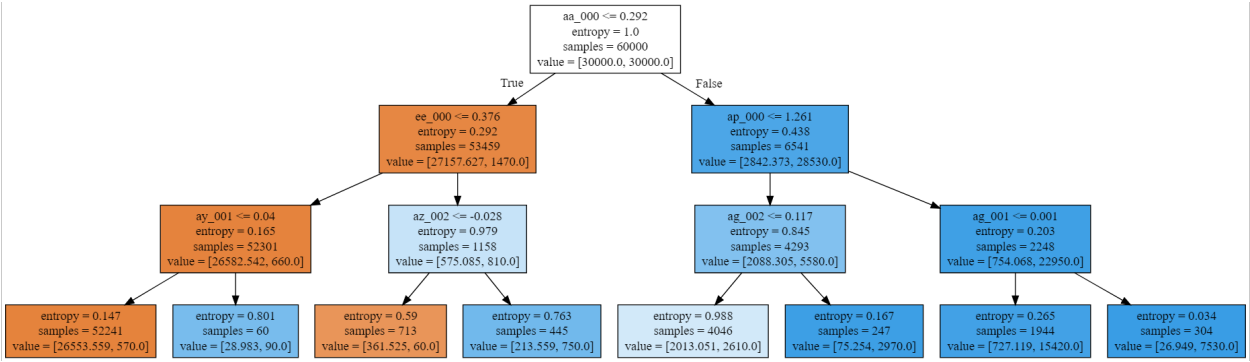

Figure 7: Decision Tree visualization.

You can find the code for Decision Tree training in source code part 3.2: "Decision Tree".

4.4.3 Random Forest

To implement the Random Forest algorithm we have used *RandomForestClassifier* from *sklearn.ensemble* module. The general sequence of steps was the same as in the previous algorithm:

1. Defining a baseline model: training classifier with default or arbitrarily chosen hyperparameters and evaluating the baseline model.

2. Tuning classifier hyperparameters using 5-fold cross-validation grid search for training set and recall as evaluation measure.

   Random Forest has several important hyperparameters which we tuned during cross-validation:

   - *n_estimators* - the number of trees in the forest
   - *max_features* - the number of feature to consider when looking for a best split
   - *min_samples_leaf* - the minimum number of samples required to be in a leaf node
   - *max_depth* - the maximum depth of the tree

   There are a few more, but we have tuned only these four. The values we used for grid search CV were:

   n_estimators: [100, 150, 200]

   max_features: ['sqrt', 'log2']

   min_samples_leaf: [1, 3, 5]

   max_depth: [20, 25, 30]

   The best parameters that were found: $n\_estimators = 200$, $max\_depth = 20$, $max\_features$ = 'log2', $min\_samples\_leaf = 5$.

3. Fitting the model with best hyperparameters, applying it on data set, computing confusion matrix and calculating total costs.

   After training the model with these parameters and applying it on the test set we got the following results:

   Recall = 84.27%

   Total_Cost = 30530

   The confusion matrix is shown in figure 8.

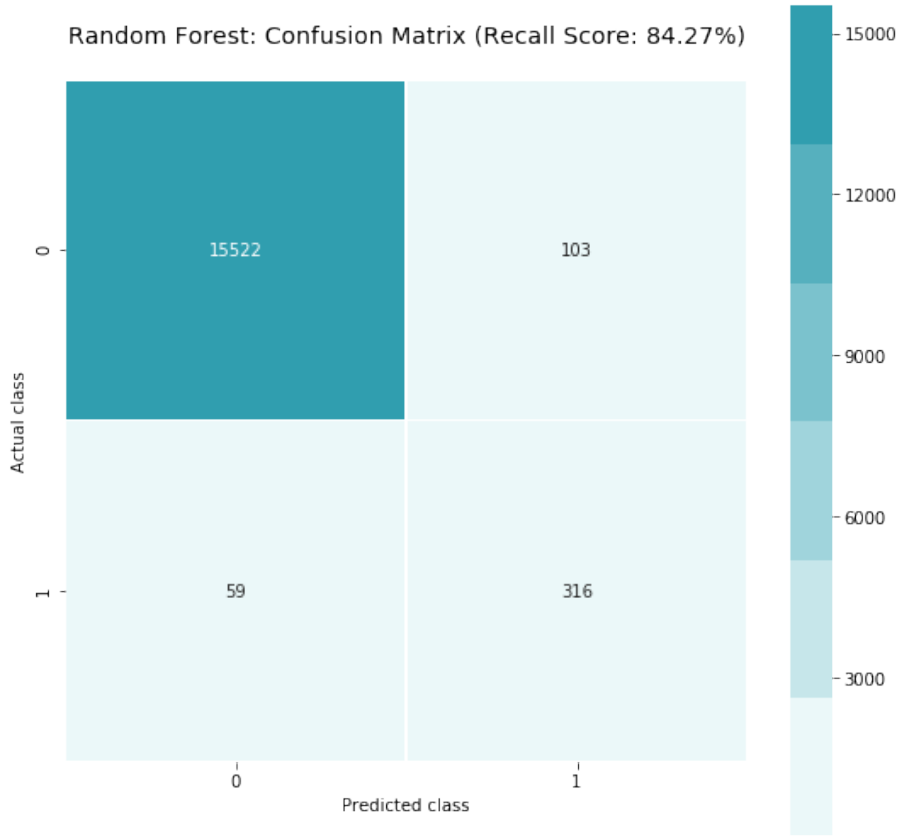

Figure 8: Confusion matrix (Random Forest).

You can find the code for Random Forest training in source code part 3.3: "Random Forest".

4.4.4 SVM

To implement SVM we used *SVC* classifier from *sklearn.svm* module to use with non-linear kernels, *LinearSVC* classifier from *sklearn.svm* module to use with linear kernel and also *SGDClassifier* (Stochastic Gradient Descent) from *sklearn.linear_model* (which acts as Linear SVM when parameter *loss* = 'hinge').

The overall framework was the same as for previous algorithms, except that they were repeated for the following three cases:

1. Non-linear kernels

   We have tried three kernels:

   - *RBF* - Radial Basis Function:

   $$K = exp(-\gamma||x - x'||^2)$$

   - *sigmoid*:

   $$K = tanh(\gamma < x, x' >)$$

   - *poly* - polynomial:

   $$K = (\gamma < x, x' >)^3$$

   For each of these kernels the only two parameters we tuned were *gamma* and regularization parameter C:

   gamma: [0.0001, 0.001]

C: [0.001, 0.01, 0.1, 1, 10, 100, 1000]

The best parameters found during grid search cross-validation were:

$kernel$ = 'poly'

$gamma$ = 0.001

C = 1000

After training the model with these parameters and applying it to the test set we have got the following results:

Recall = 74.93%

Total_Cost = 47500

The confusion matrix is shown in figure 9.

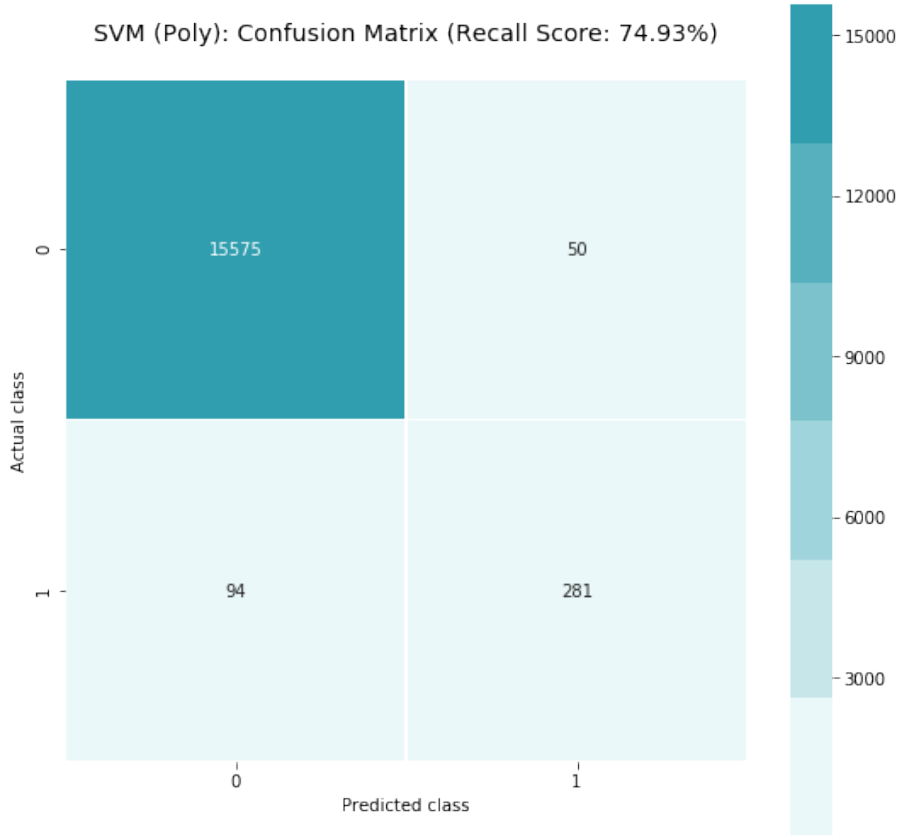

Figure 9: Confusion matrix (SVM - Poly).

2. Linear kernel: LinearSVC

$$K =< x, x' >$$

The only parameter to tune was C:

C: [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

The best parameter found during grid search cross-validation was:

C = 0.0001

And the final results after applying the model on the test set are:

Recall = 94.93%

Total_Cost = 13520

The confusion matrix is shown in figure 10.

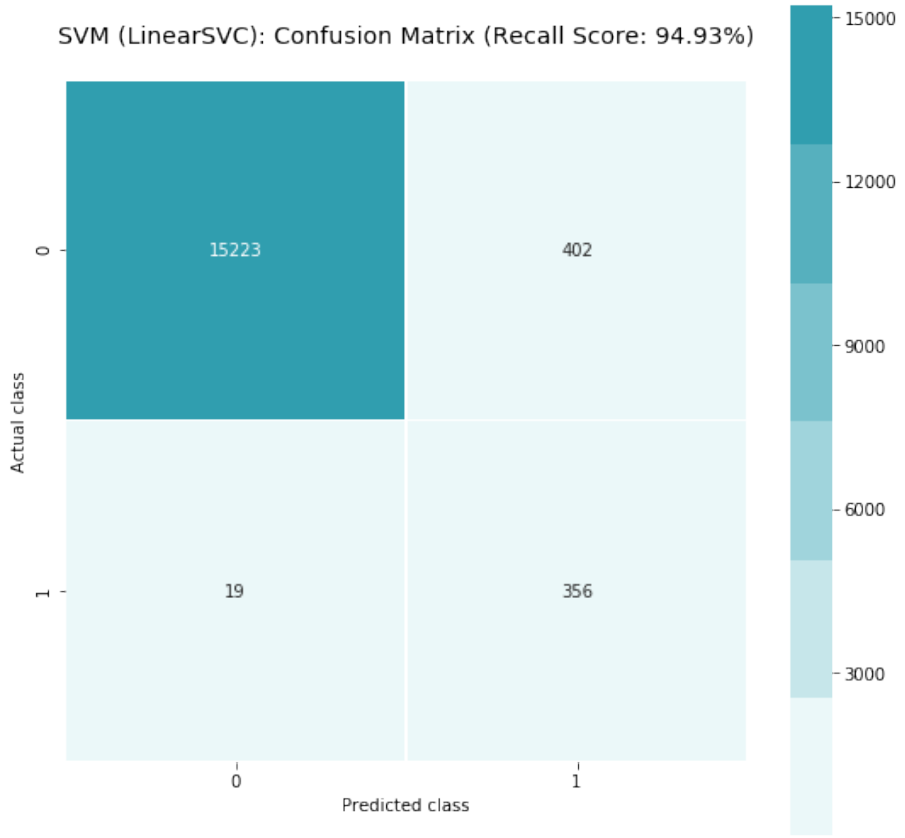

Figure 10: Confusion matrix (SVM - LinearSVC).

3. Linear kernel: SGDClassifier

   Just to compare two different implementations of Linear SVMs I have tried *SGDClassifier* with *loss*='hinge' in order to compare it with LinearSVC in terms of speed and performance.

   The only parameter to tune was *alpha*:

   alpha: [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]

   The best parameter found was *alpha* = 0.0001, which gave us the following results:

   Recall = 96.80%

   Total_Cost = 13140

   The confusion matrix is shown in figure 11.

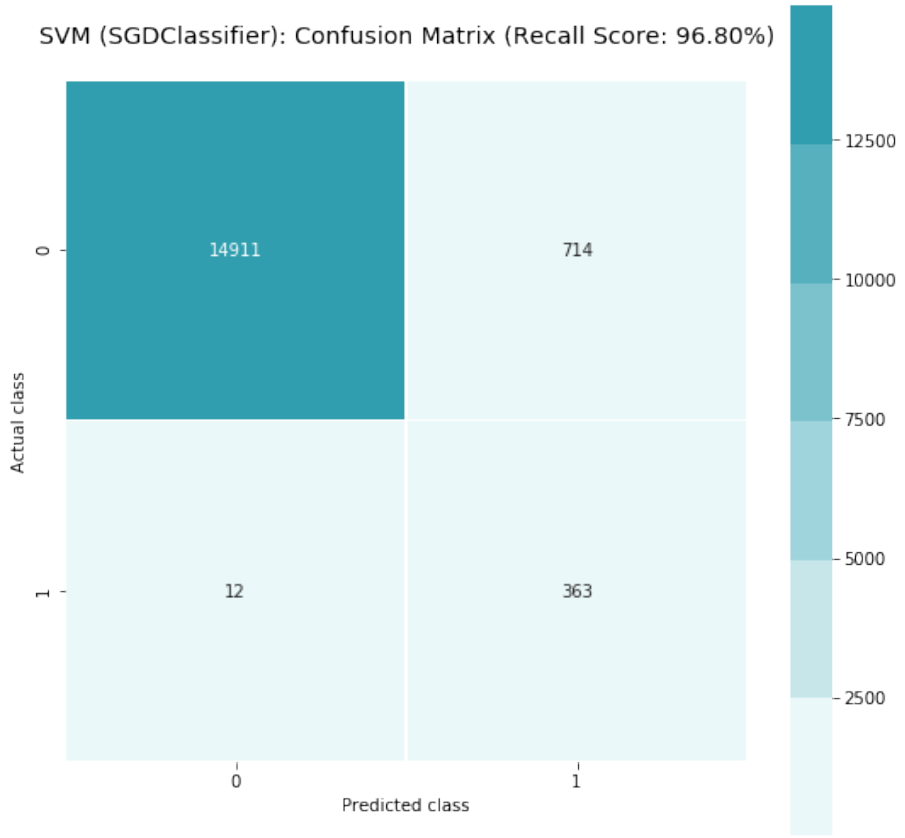

Figure 11: Confusion matrix (SVM - SGDClassifier).

As expected, two linear kernel have comparable performance with the same regularization parameter (SGDClassifier is a bit better though), but in terms of speed SGDClassifier is significantly faster than LinearSVC: 7.3 seconds vs 7.4 minutes on grid search CV with 40 fits.

The code for training SVMs can be found in Source code part 3.4: "SVM".

4.4.5 LightGBM

And finally we decided to try *LGBMClassifier* from *LGB* library. Installation guide of the LGB library can be found here: http://lightgbm.readthedocs.io/en/latest/Installation-Guide.html

The overall sequence of steps was the same as in previous algorithms. The parameters to tune were:

learning_rate: [0.001, 0.005, 0.01, 0.1, 1] - boosting learning rate.
n_estimators: [20, 50, 100] - number of boosted trees to fit.
num_leaves: [10, 20, 31] - maximum tree leaves for base learners.

The other parameters used for the classifier were:

boosting_type = 'gbdt' (we use Gradient Boosting Decision Tree)
objective = 'binary' (for binary classification)
is_unbalance = True (because our dataset is unbalanced)

The best parameters found during grid search CV are:

learning_rate = 0.001
n_estimators = 20
num_leaves = 10

And the final results on the test set are:

Recall = 97.60%
Total_Cost = 12880
The confusion matrix is shown in figure 12.

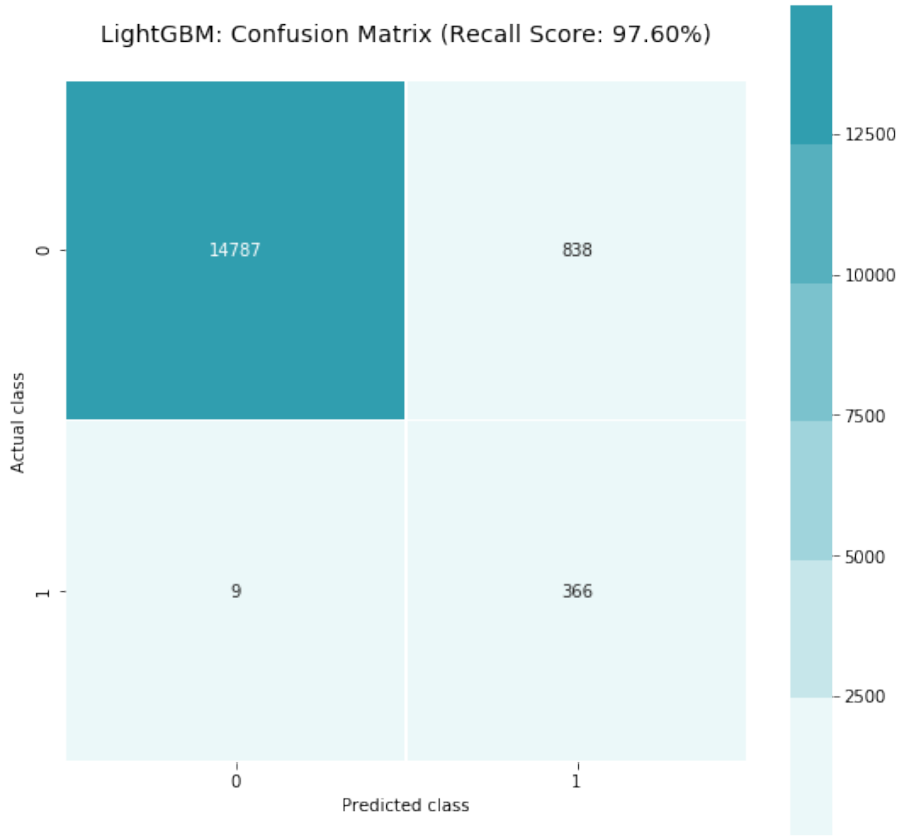

Figure 12: Confusion matrix (LightGBM Classifier).

The code for training LightGBM Classifier can be found in Source code part 3.5: "LightGBM".

4.5 Comparison of different algorithms

Algorithm	Recall	Total Cost	Time (GSCV)	Param_Comb	Time/Param_Comb
Logistic Regression	94.13%	14 840	37.6 sec	9	4.2 sec
Decision Tree	98.13%	19 010	2304 sec	648	3.6 sec
Random Forest	84.27%	30 530	9228 sec	108	85.4 sec
SVM (Poly)	74.93%	47 500	2136 sec	42	50.9 sec
SVM (LinearSVC)	94.93%	13 520	444 sec	8	55.5 sec
SVM (SGDClassifier)	96.80%	13 140	7.3 sec	8	0.9 sec
LightGBM	97.60%	12 880	378 sec	45	8.4 sec

Speaking about performance of the algorithms, linear classifiers and LightGBM produced better results than the other algorithms and demonstrated similar performance: Total_Cost was about 12 000 to 14 000. The other methods (Decision Tree, Random Forest and Non-linear SVMs) showed significantly worse results, but we think that the reason for that is not that these methods are not suited for the task, but in optimal parameters. Linear classifiers have only one hyperparameter, so it is much faster and easier to find the parameter, that leads us to more or less good results. Decision Trees and Random Forest have a few important parameters, which means it is needed to check hundreds or even thousands of parameter combinations in order to find the best ones. That is quite difficult and time consuming. Speaking of SVMs, finding a suitable kernel is difficult task itself and we only tried a few different values for a couple of parameters, which is definitely not enough to find the best parameters. The following table summarizes the performance and speed of applied algorithms. Time (GSCV) is the time needed to find the best parameters using Grid

Search cross-validation. Although it is not mathematically correct algorithms by comparing the time needed to fit different number of parameter combinations (because more fitting combinations needs more time), but from a practical point view it is a good indicator of algorithms performance. For example, despite that SVM Poly is faster than LinearSVC in terms of time needed to compute one fit, LinearSVC (8 combinations in 7.4 minutes to achieve 13 520 result) is definitely better than SVM Poly (42 combinations in 35.3 minutes to achieve 47 500 result), because we get better performance in shorter time. In terms of evaluation metric we used the best result was shown by Decision Tree, but due higher number of FP, the Total_Cost result was worse than results of some other algorithms. Overall, in terms of pure performance the winner is LightGBM, in terms of performance and speed the winner is SGDClassifier.

5 Conclusion

As we can see, Machine Learning algorithms can quite successfully solve classification tasks even when the data a lot of instances and dimensions and is not of the best quality. Predicting class label for 16 000 test instances takes less 10 minutes for some algorithms and the performance is quite good. That means that machine learning algorithms definitely can be useful in many real-world applications and help businesses save money, analyze and predict incoming risks and so on.

Of course, the results that we could achieve are not perfect and many things can be improved:

- Trying different heuristics to get rid of missing values, analyzing columns with high proportion of missing values individually.
- Using Total_Cost as a scoring function instead of Recall, because Precision still matters and it is cheaper to have one FN, than zero FN and 51 (or more) FP.
- It is possible that feature selection methods, that have not been tried in this project (recursive feature elimination and PCA) could improve both speed and performance.
- Trying more parameters and parameters combinations definitely would help to find better solutions.
- Trying different methods of finding best parameters could also have a positive effect (for example, using logspace instead of just a list of predefined values or Randomized Search).

So, there are a lot of things that could be improved or done better, but we could not realize all of them due to the large numbers of instances and features and big amount of time needed to implement all of those ideas. But nevertheless this project help us to get some practical experience with machine learning and understand some algorithms better. It was also very interesting to try not only traditional methods well-known methods, but some recently appeared algorithms like LightGBM as well. Good results produced by LightGBM show that machine learning field is successfully developing and we can expect even more efficient and accurate algorithms in the future.