AMD-SMML Joint Project

Analysis of "IBM Transactions for Anti Money Laundering" dataset

Federico Bruzzone, Ivan Mazzon

October 2023

Github repository: `https://github.com/FedericoBruzzone/anti-money-laundering/`

We declare that this material, which we now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should we engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by us or any other person for assessment on this or any other course of study.

Contents

1 Introduction

The money laundering detection is a considerable problem in machine learning. The difficulty of this problem is due to high imbalanced available data, in detail the shortage of examples of illegal operations, that we will call “positive” instances. In this project we have provided an implementation of three tree learning algorithm for binary classification to face this problem: ID3 [1], C4.5 [2] and a custom algorithm inspired by techniques used in ID3. Each algorithm is also used to induce a random forest using the Spark framework [3].

2 Preliminary Dataset Analysis

“IBM Transactions for Anti Money Laundering” provides several dataset each of which consists in transactions data.

In every dataset, the value of every attribute is known, then, we will assume that every observation given to the model, in both fitting and training processes, will be complete. Nevertheless, a handling method for unseen values in training phase will be required. Since the training set will be extract randomly from a whole dataset, it could happen that each attribute does not take all possible unique values and in the test set there might be unseen values.

Every dataset of the repository is affected by unbalanced labeling. It means that instances that belong to positive class are very infrequent. It has observed in literature that unbalanced labeling induce a proclivity in traditional learning models to learn more from examples belonging to the majority class and consequently most of new instances will be classified with that class [4, 5]. Some provided datasets have a higher illicit ratio, but it’s not enough because the ratio between number of positive and negative is however very low. For example, that ratio in `HI-Small_Trans` dataset is $5 \cdot 10^{-4}$.

To cope with high unbalanced labeling of examples in the training set, we have chosen to evaluate some preprocessing techniques to mitigate the problem and confidently obtain higher performance of classification. They are the following:

- undersampling: it consists in discarding random examples of majority class in order to balance the training set.
- oversampling: it consists in adding multiple copies of every example of minority class in the training set.

It can be noticed that the dataset contains several categorical attributes. Generally, during the fitting procedure of a machine learning model, many comparisons between values of attributes are needed. Since equality checking between integers is more efficient than between strings, we have implemented an encoding system to convert values of attributes from strings to integers to obtain a homogeneous type representation.

The datasets have an attribute `Timestamp` which can be split in three attributes `Date`, `Hour` and `Minute`, trying to generate more useful data to process.

3 Implemented Algorithms

In this section we explain the algorithms we have implemented. To this aim, we introduce the following notation. Let T the considered training set. The instances in T are in the form $(\boldsymbol{x}, y)$, where $\boldsymbol{x} = (x_1, \ldots, x_n)$ is the vector of n values of $X_1, \ldots, X_n$ attributes and $y \in \{0, 1\}$ is the classification given a priori, since we are operating in a binary classification context. Let $N_c(S)$ the number of class $c \in C = \{c^+, c^-\}$ instances over set S.

3.1 ID3

Introduced by J.R. Quinlan in 1986, ID3 [1] is an algorithm that induces tree predictors adopting a greedy divide-et-impera strategy. In detail, when a node is considered, starting by the root, the algorithm selects the best split option at the moment and subdivides the problem in smaller ones.

The information gain criterion. ID3 uses an index called *information gain* as split quality indicator. The information gain of an attribute X over a set of instances T is calculated in the following way:

$$gain(X) = info(T) - info_X(T) \tag{1}$$

where

$$info(T) = -\sum_{c \in C} \frac{N_c(T)}{|T|} \cdot \log_2 \frac{N_c(T)}{|T|} \tag{2}$$

is the entropy function and, being $T_1, \ldots, T_s$ the partitions of T consisting of instances grouped by the value of the attribute X (s distinct values),

$$info_X(T) = \sum_{i=1}^{s} \frac{|T_i|}{|T|} \cdot info(T_i)\ . \tag{3}$$

The gain criterion selects the attribute that maximize the information gain.

The ID3 algorithm thus can be described by pseudocode as follows:

Algorithm 1 ID3 - fitting procedure

```
Input
    nd  node of decision tree
    T   training set
    Z   set of attributes already considered for a split

if all the samples of $T$ belongs to the same class $C$ then
    $nd$ is a leaf that classifies instances in class $C$
    return
end if
if all the attributes belong to $Z$ then
    return
end if
for each attribute $X \notin Z$ do
    compute $gain(X)$ (information gain)
end for
$split\_attribute$ ← attribute with higher information gain
set the split condition of $nd$
for each $T'$ in splitting of $T$ do
    add child to $nd$
    ID3($child, T', Z \cup \{split\_attribute\}$)
end for
```

Our implementation. The ID3 algorithm, in its original definition, can't handle continuous values of attributes. To cope with splitting on continuous attributes, we have introduced the hyperparameter `continous_attr_groups` $=: \nu$. It represents the maximum number of splits over continuous attributes. In other words, for a continuous attribute many thresholds as $\nu - 1$ are computed to operate the partition in groups. A threshold θ_i, with $1 \leq i \leq \nu$, is calculated as the quantile of order a such that $a = \frac{i}{\nu}$. Let x a value of attribute X, the training set T will be splitted in at most ν groups described by conditions $\{x \mid x \leq \theta_1\}, \{x \mid \theta_1 < x \leq \theta_2\}, \ldots, \{x \mid \theta_{\nu-1} < x \leq \theta_\nu\}, \{x \mid x > \theta_\nu\}$.

The original ID3 also can't handle, during the prediction phase, values of attributes not already seen in training set. Some methods to solve this problem were proposed in [1]. We chose to follow the most common value approach for efficiency reasons. Thus, a new instance with an unknown attribute value will be considered as the most common value for that attribute.

Lastly, we have introduced a hyperparameter ϕ to limit the maximum depth of the tree.

3.2 Custom Algorithm

We propose in this project a custom learning algorithm inspired by the concept of information gain of ID3. This new algorithm provides a binary split for each node of the tree, with either categorical and numerical attributes. The choice of the attribute to consider for split is made in the following way.

For each attribute X_k, the information gain is calculated for several split conditions. If X_k is categorical, for each its possible values $x_1, \ldots, x_n$, the information gain is calculated considering the split condition $T_1 = \{x \mid x = x_i\}, T_2 = T \setminus T_1$. If X_k is numerical, a search for the best information gain over a pool of τ possible thresholds is performed. The pool is defined as quantiles q_a where $a = \frac{i}{\tau}$ for $i = 0, \ldots, \tau$. The condition on continuous attributes will have the form $T_1 = \{x \mid x \leq \theta_i\}, T_2 = T \setminus T_1$. The condition that overall maximize the information gain is chosen for the split on the current node.

We have provided the option to change the calculation metric of information gain. The available choices are Shannon entropy, scaled entropy and Gini index.

The implementation provides another two hyperparameters: ϕ (`max_depth`) used to set the maximum depth of the tree, and σ (`min_samples_split`), used to set the minimum number of samples required to split an internal node.

3.3 C4.5

C4.5 [2] algorithm is an extension of ID3. It was introduced in 1993 by J.R. Quinlan and subsequently has become one of most influential algorithms in data mining [6]. The structure of C4.5 is similar to ID3 with the following changes. The algorithm uses a different criterion of choice of best split attribute. Instead using the information gain, it uses the *information gain ratio* criterion.

The gain ratio is the quantity that represents the potential information generated splitting T in s subsets over an attribute X and it's computed in the following way:

$$gain\ ratio(X) = \frac{gain(X)}{split\ info(X)}$$

where

$$split\ info(X) = -\sum_{i=1}^{s} \frac{|T_i|}{|T|} \cdot \log_2\left(\frac{|T_i|}{|T|}\right) \ .$$

Tests on continuous attributes. The specification of C4.5 provides the following solution to handle continuous values of attributes. It consists in finding a threshold to have a binary test on the attribute. Considering the training set T and an attribute X, let $x_1, x_2, \ldots, x_m$ be the sorted unique values of X. Thus there are $m-1$ possible choice of thresholds. For $i \in [1, m-1]$ the midpoint value between couples of adjacent values

$$\theta_i = \frac{x_i + x_{i+1}}{2}$$

induce the binary splitting $T_1^{\theta_i} = \{x \mid x \leq \theta_i\}, T_2^{\theta_i} = \{x \mid x > \theta_i\}$. For each value θ_i the information gain on attribute X, $gain_{\theta_i}(X)$, is computed considering the splitting above. The threshold is chosen as follows:

$$\theta = \arg\max_{\theta_i}\{gain_{\theta_i}(X)\} \ .$$

Finally, the information gain ratio of attribute X is calculated over the split $T_1^{\theta}, T_2^{\theta}$ to compete with others for the selection of the splitting attribute.

Optimized implementation. This approach is computationally intensive, mainly due to the sorting of attribute values. The calculation of the information gain for all possible split determined by the $m-1$ thresholds, as just hinted in [2], can be done in linear time updating formula's terms on the fly. In this paragraph we describe the method that we used for this update. Let θ_j be, with j from 1 to $m-1$, the threshold of which we want to calculate the information gain. We can observe that in the information gain formula 1 that the term $info(T)$ is independent from the choice of threshold. Thus that term can be computed one time and cached for next calculations. The other term of the formula is threshold dependent. We will indicate it with the notation $info_X^{\theta_j}(T)$.

To the aim of point out components to be updated, we can apply some substitutions in formula 3 using 2 as follows. For notation simplicity, we will indicate subsets originated by the split over the current threshold θ_j as T_i, without specifying the threshold as previously done.

$$\begin{aligned}
info_X^{\theta_j}(T) &= \sum_{i=1}^{s} \frac{|T_i|}{|T|} \cdot info(T_i) \\
&= \frac{|T_1|}{|T|} \cdot info(T_1) + \frac{|T_2|}{|T|} \cdot info(T_2) \\
&= \frac{|T_1|}{|T|} \cdot \left(- \sum_{c \in C} \frac{N_c(T_1)}{|T_1|} \cdot \log_2 \frac{N_c(T_1)}{|T_1|} \right) + \frac{|T_2|}{|T|} \cdot \left(- \sum_{c \in C} \frac{N_c(T_2)}{|T_2|} \cdot \log_2 \frac{N_c(T_2)}{|T_2|} \right) \\
&= - \frac{1}{|T|} \left[\left(\sum_{c \in C} N_c(T_1) \cdot \log_2 \frac{N_c(T_1)}{|T_1|} \right) + \left(\sum_{c \in C} N_c(T_2) \cdot \log_2 \frac{N_c(T_2)}{|T_2|} \right) \right] \\
&= - \frac{1}{|T|} \left[\left(N_+(T_1) \cdot \log_2 \frac{N_+(T_1)}{|T_1|} + N_-(T_1) \cdot \log_2 \frac{N_-(T_1)}{|T_1|} \right) \right. \\
&\qquad \left. + \left(N_+(T_2) \cdot \log_2 \frac{N_+(T_2)}{|T_2|} + N_-(T_2) \cdot \log_2 \frac{N_-(T_2)}{|T_2|} \right) \right]
\end{aligned}$$

From this expanded formula we can understand how to compute $info_X^{\theta_j}$ basing it on the previous calculation of $info_X^{\theta_{j-1}}$. Ideally, a function Δ can update the computation as follows:

$$info_X^{\theta_j}(T) = info_X^{\theta_{j-1}}(T) + \Delta(info_X^{\theta_{j-1}}(T)) \, .$$

We can observe that, since $x_1, \ldots, x_n$ were sorted unique values of X, also the thresholds $\theta_1, \ldots, \theta_{m-1}$, by definition, are sorted in ascending order. When every threshold is chosen, following the ascending order, for information gain calculation, one and only one element having x_j value of attribute X will switch from T_2 to T_1. Let $y_j \in \{0, 1\}$ be the negative/positive classification of this instance and $\mathcal{N}^+_{T_x} = N_+(T_x^{\theta_{j-1}})$, $\mathcal{N}^-_{T_i} = N_-(T_i^{\theta_{j-1}})$, respectively, the number of positive and negative instances in set T_i. Considering finally $\mathcal{N}_{T_i} := \mathcal{N}^+_{T_i} + \mathcal{N}^-_{T_i}$, we can define:

$$\begin{aligned}
info_X^{\theta_j}(T) = \frac{1}{|T|} \cdot \Bigg[& \mathcal{N}^+_{T_1} + y_j \cdot \log_2 \frac{\mathcal{N}^+_{T_1} + y_j}{\mathcal{N}_{T_1} + 1} \\
& + \mathcal{N}^-_{T_1} + (1 - y_j) \cdot \log_2 \frac{\mathcal{N}^-_{T_1} + (1 - y_j)}{\mathcal{N}_{T_1} + 1} \\
& + \mathcal{N}^+_{T_2} - y_j \cdot \log_2 \frac{\mathcal{N}^+_{T_2} - y_j}{\mathcal{N}_{T_2} + 1} \\
& + \mathcal{N}^-_{T_2} - (1 - y_j) \cdot \log_2 \frac{\mathcal{N}^-_{T_2} - (1 - y_j)}{\mathcal{N}_{T_2} + 1} \Bigg]
\end{aligned}$$

In detail, it can be implemented using the dynamic programming approach, caching and updating the following quantities at every iteration:

$$
\begin{aligned}
N_+(T_1^{\theta_j}) &= N_+(T_1^{\theta_{j-1}}) + y_j \\
N_-(T_1^{\theta_j}) &= N_-(T_1^{\theta_{j-1}}) + (1 - y_j) \\
N_+(T_2^{\theta_j}) &= N_+(T_2^{\theta_{j-1}}) - y_j \\
N_+(T_2^{\theta_j}) &= N_+(T_2^{\theta_{j-1}}) - (1 - y_j)
\end{aligned}
$$

3.4 Random Forest

Random forests is a machine learning model that consists in training of multiple decision trees. Every decision tree is fitted in parallel with a subset of the initial training set. When a new observation is submitted for classification in a random forest, it is sent to every decision tree that makes it up. Thus, each decision tree returns a prediction. The random forest classifies the observation in the class that occurs the most. The algorithm can be summarized as follows.

Algorithm 2 Random forest - fitting procedure

```
Input
    $T$    training set
    $n$    number of trees in the forest

$F \leftarrow \emptyset$
for $i \in \{1, \dots, n\}$ do
    $T^{(i)} \leftarrow$ A bootstrap sample from $T$
    $f_i \leftarrow \textsc{TreeFit}(T^{(i)})$
    $F \leftarrow F \cup f_i$
end for
```

Usage of Spark framework. The random forest model is used in this project in a distributed configuration. This is possible through the usage of the Apache Spark framework [3] and its Python library `pyspark`. The Spark framework allows to handle data parallelism and distributed computing.

Its functioning is based on the abstraction of RDD, *resilient distributed dataset*, which is a collection of elements partitioned across the workers that can be operated on in parallel. In our case, every worker is responsible of fitting its own decision tree given a partition of the training set passed in RDD format.

For the prediction phase, we can use the MapReduce paradigm to submit every row of test set to every tree (each one in a worker) and aggregate results. We can illustrate the procedure through the following formalism. Let T' be the test set and

h be the tree predictor owned by the worker, then:

$$\begin{aligned}
\forall\ t_i \in T' &\xrightarrow{\text{M}} (i,\ h(t)) \\
\forall\ (i, h(t)) &\xrightarrow{\text{M}} ((i,\ h(t)),\ 1) \\
((i,\ h(t)),\ (1, \ldots, 1))) &\xrightarrow{\text{R}} ((i,\ h(t)),\ \text{sum}(1, \ldots, 1) =: a) \\
\forall\ ((i,\ h(t)),\ a)) &\xrightarrow{\text{M}} (i,\ (h(t),\ a)) \\
(i,\ ((h(t_1),\ a_1), \ldots, (h(t_n),\ a_n))) &\xrightarrow{\text{R}} (i,\ (b^-, b^+)) \\
\forall\ (i,\ (b^-, b^+)) &\xrightarrow{\text{R}} \begin{cases} 1 & \text{if } b^+ \geq b^- \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

At the end of this chain of MapReduce operations, we have the classification of every observation t_i of the test set which is the most frequent class generated by trees.

4 Experimental Results and Discussion

In this section experiments we have done are described to evaluate performance of the models implemented. Initially, each algorithm have been tested with an example dataset, a Diabetes prediction dataset [7], to verify their correctness. Although we have obtained excellent classification performance on this dataset, since its analysis is not a objective of this project, we don't report experiments on this dataset in this paper.

With the aim of fitting the decision tree model on a single machine, for the following experiments we have considered the `HI-Small_Trans` dataset. In fact, we can reasonably assume that it can be stored entirely in main memory.

4.1 Decision trees

4.1.1 ID3

As said previously, our implementation of ID3 has the hyperparameter ν, that is the number of groups for the splitting over continuous attributes and ϕ, which represents the maximum depth of the tree. The approach we used for this experiment is a grid search based on the two hyperparameters. We have chosen to evaluate the subsets of values $\nu \in \{2, 4, 6\}$ and $\phi \in \{4, 8, 12\}$. The table 1 shows the classification performance of the model for these configurations.

First of all, we can observe the phenomenon described in [5], i.e. without measures to balance the dataset (the non-preprocessed way) the model classified every instances of the test set as negative. In fact, for each configuration of hyperparameters, no true positive are obtained by the model, as suggested by F_1 score equals to zero in every case. For next experiments the non-preprocessed configuration will not be reported

		Non-preprocessed		Undersampled		Oversampled	
ν	ϕ	F_1	*Acc*	F_1	*Acc*	F_1	*Acc*
2	4	0	0.999	0.0148	0.883	0.0148	0.883
	8	0	0.999	0.0148	0.883	0.0148	0.883
	12	0	0.999	0.0147	0.882	0.0148	0.883
4	4	0	0.999	0.013	0.866	0.0148	0.883
	8	0	0.999	0.014	0.874	0.0146	0.879
	12	0	0.999	0.0141	0.874	0.0145	0.879
6	4	0	0.999	0.014	0.868	0.0146	0.881
	8	0	0.999	0.014	0.868	0.0146	0.88
	12	0	0.999	0.0136	0.869	0.0146	0.88

Table 1: Classification performances of ID3 by varying ν and ϕ using different preprocessing techniques

since we have observed and it is already proved [5] that it leads to worse performances in terms of true positives.

In both undersampled and oversampled configurations, we have noted that with increasing of ν the performance tends to decrease. The variation of hyperparameter ϕ does not affect the results very significantly.

The execution of fitting using overfitting (the case with the higher number of considered examples) and prediction procedure takes about respectively 5 and 2 minutes to complete.

4.1.2 Custom Algorithm

For the custom algorithm we followed the same approach used in analysis of ID3. We have operated a grid search among some possible values of its hyperparameters. The hyperparamers chosen for this experiment are $\phi \in \{4, 7, 10\}$, the maximum depth of the tree, $\tau \in \{2, 4, 6\}$, the number of thresholds for numerical binary split and $\sigma \in \{2, 100\}$, the minimum number of samples that allows the split. By default, the Shannon entropy metric is used for the information gain calculation.

The table 2 shows the classification performance of the model for these configurations. We can observe that the performance in terms of both F_1 score and accuracy obtained through the oversampled training set are averagely better than those achieved with undersampled training set. Furthermore, an increase of classification performances for higher values of ϕ with a maximum of the F_1 score of 0.027. The variation of σ affects the results reasonably using undersampling preprocessing for higher values of ϕ.

For example purposes, figure 1 shows a tree classifier generated by the custom algorithm.

The execution of fitting using overfitting and prediction procedure takes about respectively 60 and 3 minutes to complete.

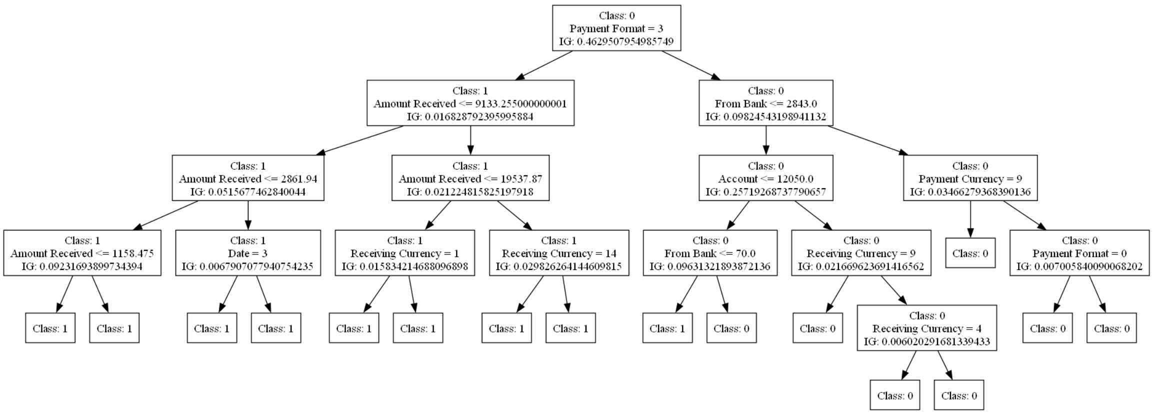
Class: 0 Payment Format = 3 IG: 0.4629507954985749
Class: 1 Amount Received <= 9133.255000000001 IG: 0.016828792395995884
Class: 0 From Bank <= 2843.0 IG: 0.09824543198941132
Class: 1 Amount Received <= 2861.94 IG: 0.0515677462840044
Class: 1 Amount Received <= 19537.87 IG: 0.021224815825197918
Class: 0 Account <= 12050.0 IG: 0.25719268737790657
Class: 0 Payment Currency = 9 IG: 0.03466279368390136
Class: 1 Amount Received <= 1158.475 IG: 0.09231693899734394
Class: 1 Date = 3 IG: 0.0067907077940754235
Class: 1 Receiving Currency = 1 IG: 0.015834214688096898
Class: 1 Receiving Currency = 14 IG: 0.029826264144609815
Class: 0 From Bank <= 70.0 IG: 0.09631321893872136
Class: 0 Receiving Currency = 9 IG: 0.021669623691416562
Class: 0
Class: 0 Payment Format = 0 IG: 0.007005840090068202
Class: 1
Class: 1
Class: 1
Class: 1
Class: 1
Class: 1
Class: 1
Class: 1
Class: 1
Class: 0
Class: 0
Class: 0 Receiving Currency = 4 IG: 0.006020291681339433
Class: 0
Class: 0
Class: 0
Class: 0

Figure 1: A representation of tree classifier generated with the custom algorithm using in the configuration with undersampling and $\phi = 20, \tau = 2, \sigma = 100$.

			Undersampled		Oversampled	
ϕ	τ	σ	F_1	Acc	F_1	Acc
4	2	2	0.012	0.843	0.023	0.929
		100	0.012	0.843	0.023	0.929
	4	2	0.015	0.885	0.02	0.917
		100	0.015	0.884	0.02	0.917
	6	2	0.015	0.882	0.021	0.919
		100	0.015	0.882	0.021	0.919
7	2	2	0.019	0.916	0.021	0.924
		100	0.02	0.915	0.021	0.924
	4	2	0.014	0.875	0.019	0.915
		100	0.014	0.875	0.019	0.915
	6	2	0.014	0.869	0.019	0.91
		100	0.014	0.867	0.02	0.91
10	2	2	0.012	0.911	0.023	0.932
		100	0.02	0.917	0.023	0.932
	4	2	0.014	0.878	0.023	0.932
		100	0.014	0.88	0.021	0.928
	6	2	0.016	0.888	0.021	0.928
		100	0.014	0.876	0.019	0.92
20	2	2	0.017	0.914	0.027	0.966
		100	0.02	0.918	0.027	0.966
	4	2	0.016	0.897	0.025	0.963
		100	0.015	0.886	0.025	0.963
	6	2	0.015	0.896	0.024	0.964
		100	0.014	0.877	0.024	0.964

Table 2: Classification performances of custom algorithm using different preprocessing techniques.

4.1.3 C4.5

Maintaining the same analysis approach, we have evaluate a grid search over the hyperparameters $\phi \in \{4, 8, 12\}$ and $\sigma \in \{2, 100\}$. These hyperparameters have the same functions as in the previous algorithm.

The table 3 shows the classification performance of the model for these configurations. We can observe that every pair of hyperparameters leads to very similar performances respectively in undersampling and oversampling layouts.

The execution of fitting using overfitting and prediction procedure takes about respectively 45 and 2 minutes to complete.

		Undersampled		Oversampled	
ϕ	σ	F_1	*Acc*	F_1	*Acc*
4	2	0.0138	0.87	0.015	0.889
	100	0.0136	0.869	0.015	0.889
8	2	0.013	0.862	0.015	0.883
	100	0.013	0.862	0.015	0.883
12	2	0.013	0.862	0.015	0.883
	100	0.013	0.862	0.015	0.883

Table 3: Classification performances of C4.5 by varying ν and σ using different preprocessing techniques

4.2 Random forest

The experiments concerning random forests are structured as follows. A random forest is trained for each kind of previously analyzed decision tree considering the best configuration depending on preprocessing method and hyperparameters values. Thus, these configurations are evaluated with the oversampling preprocessing method and the following hyperparameters values:

- ID3: $\phi = 8,\ \nu = 2$;
- Custom: $\phi = 20,\ \tau = 2,\ \sigma = 100$;
- C4.5: $\phi = 4,\ \sigma = 100$.

We have considered as training set the 10% of the original dataset due to execution time reasons. Furthermore, we have used a resampling technique called bootstrap sampling [8], which consists in randomly select $k = |T|$ observations from the dataset to obtain a new bootstrap dataset. The resampling is performed with replacement, which means that the same observation can occur more than once in the bootstrap dataset.

These settings, used to induce random forests, lead to performances reported in table 4. In only one case, using the custom algorithm, the generation of a random forest allows to obtain slightly better results respect to the single tree setting.

The execution of random forest in the worst case takes about 7 minutes to complete.

Algorithm	F_1	Acc
ID3	0	0.998
Custom	0.003	0.996
C4.5	0.001	0.958

Table 4: Classification performances of random forest using different tree induction algorithms.

References

[1] J. R. Quinlan. "Induction of decision trees". In: *Machine Learning* 1.1 (Mar. 1986), pp. 81–106. ISSN: 1573-0565. DOI: 10.1007/BF00116251. URL: https://doi.org/10.1007/BF00116251.

[2] J. R. Quinlan. *C4.5: programs for machine learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1-55860-238-0. URL: http://portal.acm.org/citation.cfm?id=152181.

[3] URL: https://spark.apache.org/.

[4] Charles X. Ling and Victor S. Sheng. "Class Imbalance Problem". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 171–171. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_110. URL: https://doi.org/10.1007/978-0-387-30164-8_110.

[5] Megahed Fadel M. et al. "The class imbalance problem". In: *Nature Methods 18*. 2021, pp. 1270–1272. URL: https://doi.org/10.1038/s41592-021-01302-4.

[6] Xindong Wu et al. "Top 10 algorithms in data mining". In: *Knowledge and Information Systems* 14.1 (Jan. 2008), pp. 1–37. ISSN: 0219-3116. DOI: 10.1007/s10115-007-0114-2. URL: https://doi.org/10.1007/s10115-007-0114-2.

[7] Mohammed Mustafa. *Diabetes prediction dataset*. URL: https://www.kaggle.com/datasets/iammustafatz/diabetes-prediction-dataset.

[8] Gareth James et al. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013. Chap. 5.2.