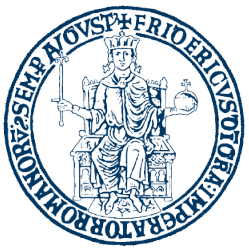
EUSTEMA CHALLENGE

Dipartimento di Fisica "Ettore Panicini"

Via Cinthia 21, Napoli



UNIVERSITÀ DEGLI STUDI DI NAPOLI

FEDERICO II

Corso di Laurea Magistrale in Data Science

Ammirati Vincenzo, Ettari Adriano, Ferrara Andrea, Zighed Ismael

Contents

1 Abstract

This report focuses on the techniques we applied in order to make predictions on the **Outcome**, the **Duration** and the **Settlement** variables in two different datasets (*train_s* and *train_od*). The observations in these datasets are *300.000 Italian lawsuits*. We deeply focused on the preprocessing step, considering the null values and the anomalies for each variable (not only by looking at the variable distributions, but also analyzing the relation among variables and also their meaning); removing the least useful variables and applying format transformation that are needed to run the models. Moreover, we deal with the unbalancing problem by applying both downsampling and upsampling. Eventually, we tried several models like **Random Forest**, **MARS** and **Neural Networks** both for Regression (predict *Settlement* and *Duration*) and Classification task (predict Outcome).

2 Introduction

According to the strategy of learning by doing, we tackle the challenge presented by EUSTEMA S.P.A. which is an Italian company active in the field of Information and Communication Technology that offers consulting services and software engineering. The challenge is about predicting the *Outcome*, the *Duration* in number of days and the *Settlement* for around 300.000 lawsuits produced by several tribunals of the "Giudice di Pace" scattered around Italy.
Eustema supplied 4 csv file:

- Training Set for *Outcome* and *Duration* prediction (train_od): containing all the data fields and the 2 target variables.
- Blind Test Set for *Outcome* and *Duration* prediction (test_od): containing all the data fields but no target variables.
- Training Set for *Settlement* prediction (train_s): containing all the data fields and the target variable.
- Blind Test Set for *Settlement* prediction (test_s): containing all the data fields but no target variable.

. In particular the following data fields are present:

- ID : an unique identifier to each row of the dataset
- Case Identifier : an unique identifier for the case. The first number represents the one of the case in order of time of the year (second number) in one particular office. It represents the number of the lawsuit in an office in a year.
- Judge Identifier : an unique ID given to each Judge.
- Object : a code which identifies the type of the lawsuit.
- Date : the date in which the Judge Identifier was assigned, and hence also the date when the case started.
- Section : the judge's tribunal section. Where the judge's office is.
- Value : the monetary value of the lawsuit.
- Tax Related : it indicates if the case is tax related.
- Unified Contribution : it is the monetary cost for starting the lawsuit. It is related to *Value*. To each Value corresponds a scheduled Unified Contribution (two tables before and since 25/06/2014). There are also particular cases.
- Primary Actor : it is the primary actor of the lawsuit, i.e. the accusing part.
- Secondary Actor : it is the secondary actor of the lawsuit, i.e. the accusing part.
- Primary Defendant : is the primary defendant of the lawsuit.

- Secondary Defendant : it is the secondary defendant of the lawsuit
- Number of Lawyers : the number of lawyers involved in the lawsuit, attack plus defense.
- Number of Legal Parties : the number of legal parties involved in the lawsuit.
- Number of Person : the number of persons involved in the lawsuit, only defense (lawyers excluded).
- City of the Judge's Office : the city in which the judge's office is located.
- Outcome : the outcome of the lawsuit (the first target variable).
- Duration : the duration of the lawsuit in number of days (the second target variable).
- Settlement : the final settlement of the lawsuit (the third target variable). In particular, it is the cost of the lawsuit, how much the lawyers have to be paid.

Besides the following metrics will be used to evaluate your predictions:

- *Outcome* classification metric : micro-averaged F1 Score.
- *Duration* regression metric: Mean Absolute Error (MAE).
- *Settlement* regression metric: Mean Absolute Error (MAE).

The first section is the Section 3, and it is the preprocessing. This step consists in manipulating, analyzing and standardizing the data.
Then, in Section 4, we introduce different models and all the parameters that we set for each one.
In section 5, we show the results of the best model, why we have chosen it and some possible improvements.

3 Preprocessing and EDA

The first section of this article is designated to the 'Preprocessing', a preliminary step to prepare raw data to be analyzed.
We start considering the train_s dataset. The first step is to delete the first column *Unnamed:0* because it is a redundant information since there is the second column *ID* which identifies uniquely each row in the dataset.
To understand the direction to take, some exploratory data analysis is done, e.g. the cases are not balanced in the cities since Milan (the city where the majority of cases are debated) shows almost a double number of them respect to Naples (the second classified) and then the differences between cities become not so significant as the first.

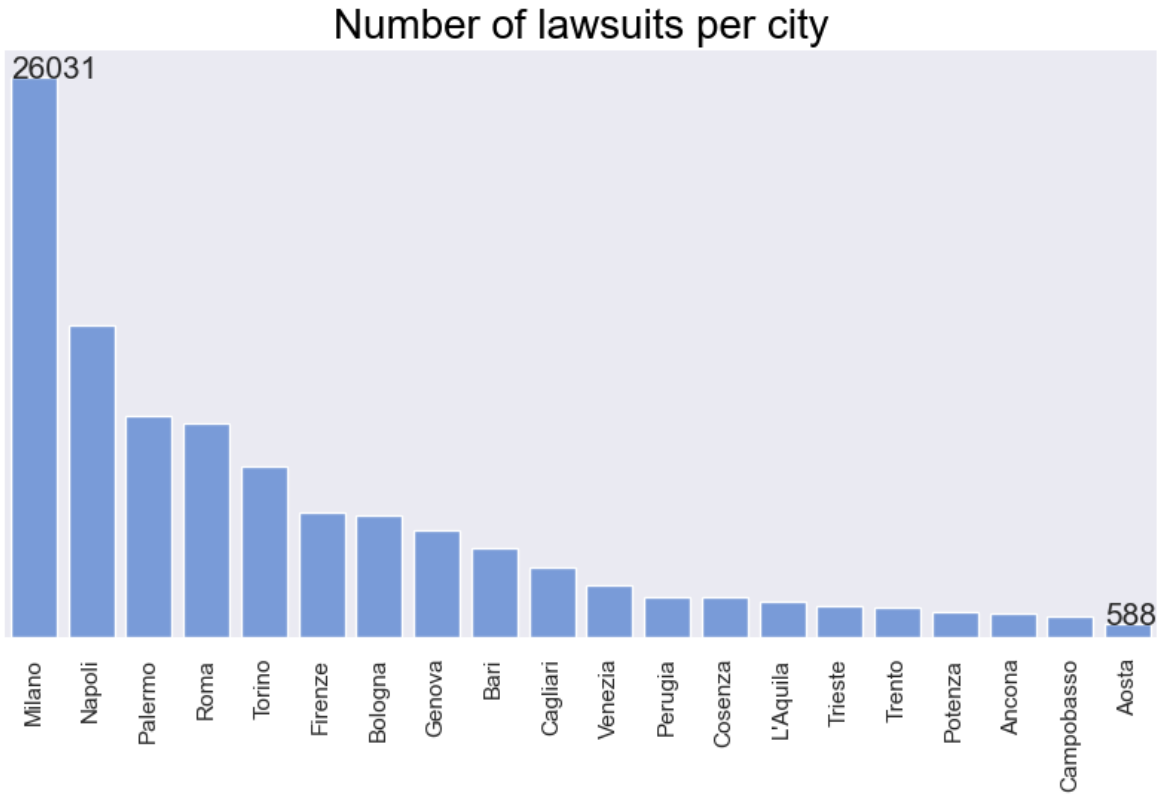


Figure 1: *Number of lawsuits for city*

Analysing all the variables individually, it stands out that *Tax related* was characterized by TRUE and NaN values, so a transformation of this variable into a boolean 0/1 and a count have been done.
The greatest preprocessing action has been taken on the columns *Value* and *Unified Contribution*. Firstly we transform them into numeric cells. Then, we apply an anomaly detection checking if *Unified Contribution* corresponds to the scheduled values of the Lawsuit Value (tables 1 and 2). Besides, *Unified Contribution* can also assume values that are not present in these two tables (when they are halved, doubled or added to its half). All the other 'strange' values have been eliminated.
To agree upon the next step to do, we build Spearman (not linear) correlation between the numerical columns of the dataset. We can see that there are some columns which are highly correlated. For example, the *Settlement* is highly correlated with *Value*.

Lawsuit Value	Unified Contribution
0€	43€ 237€ 98€
1€ - 1100€	43€
1100.01€ - 5200€	98€
5200.01€ - 26000€	237€
26000.01€ - 52000€	518€
52000.01€ - 260000€	759€
260000.01€ - 520000€	1214€
520000.01€ -	1686€

Table 1: *Since 25/06/2014 (First Grade)*

Lawsuit Value	Unified Contribution
0€	37€ 85€ 206€ 450€
1€ - 1100€	37€
1100.01€ - 5200€	85€
5200.01€ - 26000€	206€
26000.01€ - 52000€	450€
52000.01€ - 260000€	660€
260000.01€ - 520000€	1056€
520000.01€ -	1466€

Table 2: *Before 25/06/2014 (First Grade)*

Therefore, we analyze these two variables together. What we can evaluate is that *Value* is quite always greater or equal to *Settlement*, except for some observations. In particular, there are some observations where *Settlement* is much greater than *Value*. That's why we decide to analyze in detail these observations.

In fact, the last preprocessing step is the anomaly detection for *Settlement*. We remove the values close to the corresponding *Value* multiplied by $10, 100, 1'000, 10'000$ and in the range $10'000 < Settlement < 1'000'000$. Moreover, it is unlikely to observe values of *Settlement* greater than 1'000'000, so we decide to remove them too.

In train_od we repeat the same procedures of train_s.

Figure 2: *Correlation between numerical variables*

The only difference is the presence of *Duration* and *Outcome* instead of *Settlement*. We remove all the observations where *Duration* is lower than 0 (4% of the overall observations) because it can't be a negative number. Besides, we dropp also the observations in which the date plus the duration exceed the 15 of October 2022 (when the data was provided to us). As regard *Outcome*, we have not preprocessed it but we have noticed that the classes are not balanced. In particular, there are 140295 where Duration is equal to *Accepted*, 80739 where it is equal to *Declined*, 20140 where it is equal to *Partially Accepted* and 10940 where it is equal to *Ceased Matter*.

3.1 One Hot Encoding and Standardization

This section works on a filtered dataset with only the variables we are interested in. *ID*, *Primary Actor*, *Secondary Actor*, *Primary Defendant*,*Secondary Defendant*, *Judge Identifier*, *Date*, *Section* have been removed.
From this moment on, we will work with this filtered datasset. The first action to take here is to convert the categorical variables *City of the judge's office* and *Object* into numerical. For doing this we apply One Hot Encoding.

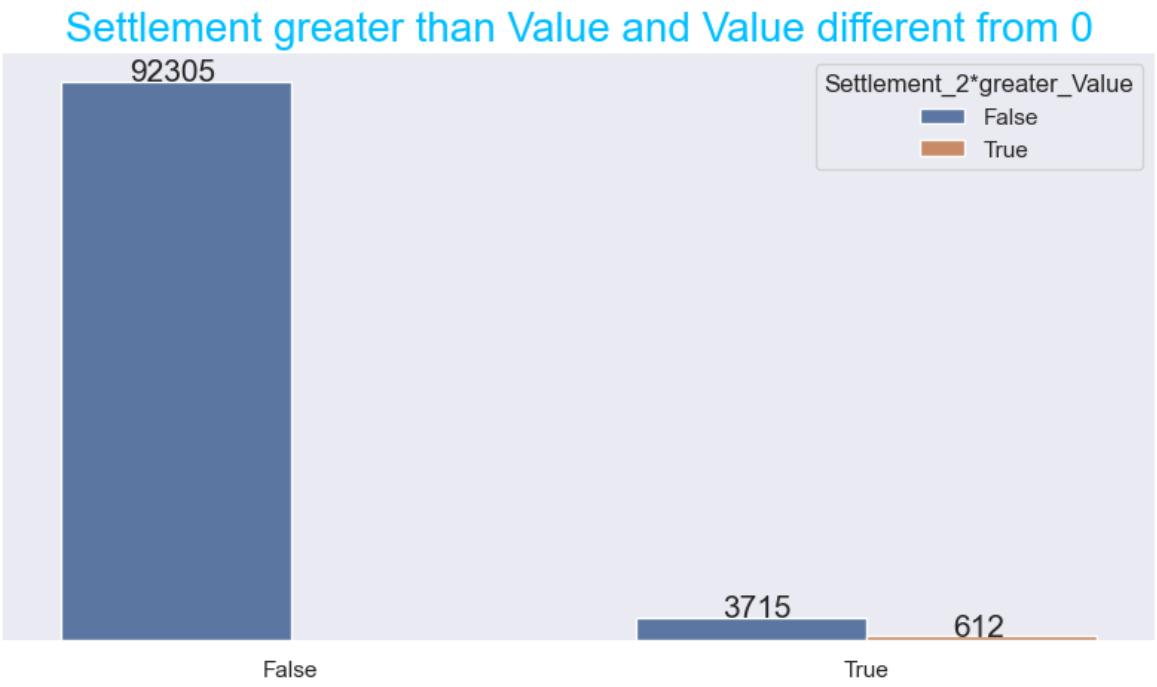


Figure 3: *Settlement greater than value and value different from 0*

As regard train_od, the target variable*Outcome* is categorical. It has 4 classes:

- *Accepted*,
- *Ceased Matter*,
- *Declined*,
- *Partially Accepted*.

In order to apply the model we have to transform *Outcome* into a numerical variable and we did this with the above-mentioned procedure. In particular, *Accepted* will be equal to 0; *Ceased Matter* will be equal to 1, *Declined* will be equal to 2 and *Partially Accepted* will be equal to 3. Finally, the last step before applying the models is the Standardization of the quantitative variables. These are *Number of Lawyers*, *Number of Legal Parties*, *Value*, *Unified Contribution*.
Last but not least we divide the dataset into train set (70%), validation set (20%), test set (10%) to build and check models.

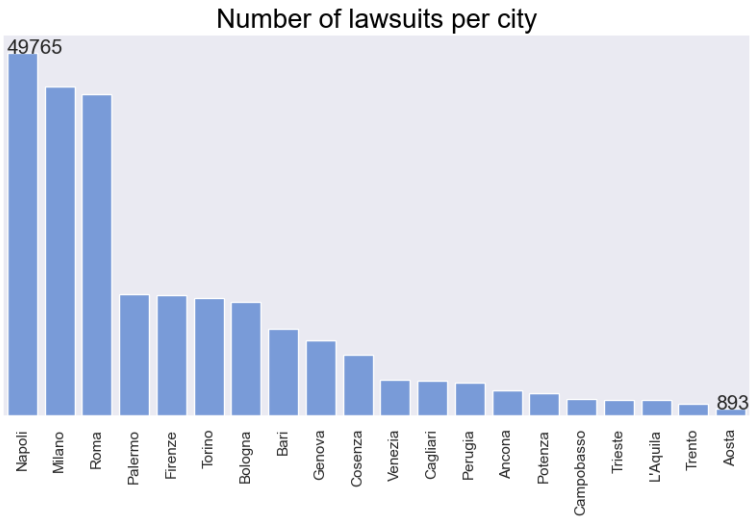


Figure 4: *Number of lawsuits for city in train_od*

4 Models

In Machine learning, it is better to analyze different models to obtain the best results. We put the attention on the most famous Ensemble Methods and on the Neural Networks. Firstly, we implement Ensemble methods because they are very suitable for complex and large datasets. Secondly, we apply Neural Networks because they are versatile, powerful and scalable, making them ideal to tackle large and highly complex Machine Learning tasks.

4.1 Ensemble Methods

The first models we decide to apply are Random Forests and Gradient Boosting which belong to the class of the Ensemble Methods. In order to explain the above-mentioned methods we have to give a brief introduction of Decision Trees.

4.1.1 Introduction to Decision Trees

They are Machine Learning algorithms that can perform both Classification and Regression tasks. They are powerful methods capable of fitting complex datasets. In particular, one of the many qualities of Decision Trees is that they require very little data preparation. In fact, they don't require feature scaling.
Besides, they are intuitive and their decision are easy to interpret. Such models are called white box models; in contrast, Neural Networks are generally considered black box models because you can't easily check the calculation that they perform to make predictions.
We start at the root node: this node asks a question. If it is True, that specific observation moves down to the root's left child node; if it is False, that observation moves down to the root's right child node.
We consider the Classification and Regression Tree (CART) algorithm to train Decision Trees. It produces only binary trees: non-leaf nodes always have two

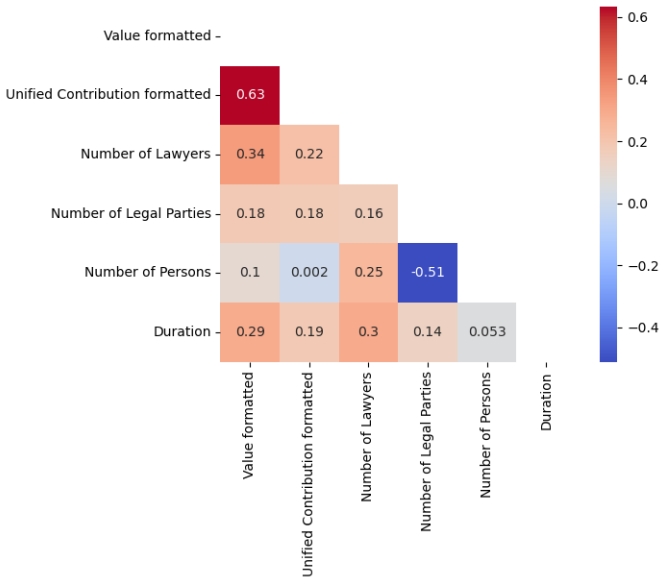


Figure 5: *Correlation between numerical variables in train_od*

children. The algorithm works by first splitting the training set into two subsets using a single feature K and a threshold t_k. How does it choose k and t_k?
In a Classification setting it searches for the pair (k, t_k) that produces the purest subsets (weighted by their size). Impurity is measured by the Gini index. A node is pure if all training instances ,in that node, belong to the same class.
In a Regression setting it searches for the pair (k, t_k) that minimizes the deviance.
Once the CART algorithm has successfully split the training set in two, it splits subset using the same logic, then the sub-subsets, and so on, recursively. That's why, we can set different parameters in order to stop the split.
These parameters are :

- max_depth: The maximum depth of the tree.
- min_sample_leaf : The minimum number of samples required to be at a leaf node.
- min_sample_split: The minimum number of samples required to split an internal node.
- max_leaf_nodes: The maximum number of leaf-nodes.

As you can imagine the CART algorithm is a greedy algorithm: it greedily searches for an optimum split at the top level, then repeats the process at each subsequent level. It does not check whether or not the split will lead to the lowest possible impurity several levels down.
Decision Trees make very few assumptions about the training data (non parametric models), so the model structure is free to stick closely the data. That's why there is a high risk of over-fitting. To avoid this problem, there are two directions. The first one is to restrict the Decision Tree's freedom during training. This step is called regularization. In order to do it we can use the above-mentioned parameters. For instance, reducing max_depth will regularize the model and thus reduce the risk of over-fitting. This solution is faster but less accurate.

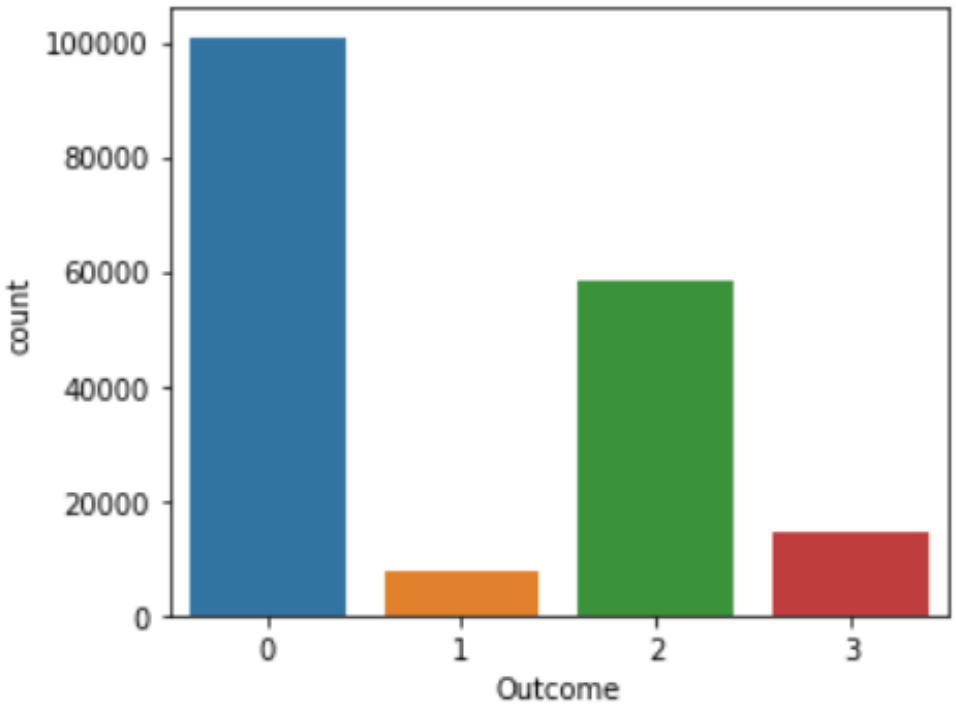


Figure 6: *Number of observation for each class*

The second one is to implement Ensemble methods. The main idea is that if you aggregate the result of a group of predictors (each predictor is a Decision Tree) you will often get better predictions than with the best individual predictor. The aggregation function is typically the statistical mode (the most frequent prediction) for Classification, or the average for Regression. This technique is called Ensemble Learning and an Ensemble Learning algorithm is called Ensemble method. The main advantage of these methods is that when you aggregate the results of different predictors, you reduce the variance and therefore also the risk of over-fitting. However this is only true if all predictors are independent.

4.1.2 Random Forests

Random forests provide an improvement over bagging, by way of a small tweak so that the trees are not correlated anymore. As in bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, only a random sample of m predictors is chosen as split candidates from the full set of p predictors. Typically we choose m equal to the square root of p. This may sound crazy, but it has a clever rationale. Suppose that there is one very strong predictor in the dataset, along with a number of other moderately strong predictors. Then in the collection of bagged trees, most or all of the trees will use this strong predictor in the top split. Consequently, all of the bagged trees will look quite similar to each other. Hence the predictions from the bagged trees will be highly correlated. Unfortunately, averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities. This means that bagging will not

lead to a substantial reduction in variance over a single tree in this setting. Random forests overcome this problem by forcing each split to consider only a subset of the predictors. We can think of this process as "decorrelating" the trees, thereby making the average of the resulting trees less variable and hence more reliable. Yet another great quality of Random Forests is that they make it easy to measure the relative importance of each feature. This is very useful, in particular if you need to perform feature selection.

At this point let's see how we built this model for train_s and for train_od.

train_s : our response variable is numerical, so we build a *RandomForestRegressor* model. In particular, we try several values and several combinations for the parameters. At the end the best parameters are:

- n_estimators = 50,
- max_depth = 10,
- min_sample_leaf = 20,
- min_sample_split = 20,
- random_state = 42,
- criterion = absolute_error.

. We use the combination of the training and the validation set and for testing the model we use the test set. For the test set we obtain a mean absolute error of 765.

train_od : we apply this model both for the Regression and the Classification task. In the Regression task our response variable (*Duration*) is numerical so, as we do for train_s, we build a *RandomForestRegressor* model. The best parameters are:

- n_estimators = 300,
- random_state = 42.

. Also in this case, we use the combination of the training and the validation set and for testing the model we use the test set. For the test set we obtain a mean absolute error of 233.
As regard the Classification task, the response variable (*Outcome*) is categorical. That's why,in the preprocessing, we transform it into a numerical one. In this case we build a*RandomForestClassifier* model. In this situation, the best parameters are:

- n_estimators = 100,
- min_samples_leaf = 100,
- min_samples_split = 150,
- max_features = sqrt,
- max_depth = 15,
- class_weight = balanced.

. We set class_weight equal to balanced because the class are not balanced. For the test set we obtain a micro-averaged f1 score = 0.38.

4.1.3 Gradient Boosting

Boosting refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. The most popular boosting methods are:

- Ada-Boost
- Gradient Boosting

When you train an Ada-Boost classifier, the algorithm first trains a base classifier (such as a Decision Tree) and uses it to make prediction on the training set. The algorithm then increases the relative weight of mis-classified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on.
As you can understand, one way for a new predictor to correct its predecessor is to pay more attention to the training instances that the predecessor under-fitted. This is the technique used by Ada-Boost (it adds predictors to the ensemble, gradually making it better).
Another popular boosting algorithm is Gradient Boosting. It works sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like Ada-Boost does, this method tries to fit the new predictor to the residual errors made by the previous predictor.
For both Ada-Boost and Gradient Boosting the learning rate is a very important parameter. If you set it to a low value, such as 0.01, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better.
In order to solve our two tasks we implement Gradient Boosting which can be applied importing a very famous library called XGBoost. It offers several nice features, such as automatically taking care of early stopping. Here the main idea is to stop the training when the validation error does not improve for a certain number of iterations (defined by the early_stopping_rounds parameter). This can reduce the risk of over-fitting because it prevents the error from increasing.

train_s : we start with a simple architecture where we set just few parameters. Then in order to improve the results we tried different combinations of them. In conclusion, the best parameters are:

- booster = gbtree,
- max_depth = 10,
- verbosity = 0,
- eta = 0.01,
- learning_rate = 0.05,
- eval_metric = mae,
- tree_method = exact,
- objective = reg:linear,

- subsample: 0.8,
- colsample_bytree = 0.5.

. We use the training set for fitting the model and the validation set for evaluating it. We set the the early_stopping_rounds parameter equal to 50, then, the test set is used for testing the model. For the test set we obtain a mean absolute error of 805.

train_od : as we have done for Random Forests, we apply this model both for Regression and Classification task. In the Regression task the best parameters are:

- booster = gbtree,
- learning_rate = 0.05,
- objective = reg:linear,
- eval_metric = mae,
- eta = 0.01,
- subsample = 0.8,
- colsample_bytree = 0.5,
- tree_method = exact,
- verbosity = 0,
- max_depth : 10.

As we did for train_s, we use the training set for fitting the model and the validation set for evaluating it. When we fit the model we set the early_stopping_rounds parameter equal to 50. Instead, the test set is used for testing the model. For the test set we obtain a mean absolute error of 214.
As regard the Classification task, XGBoost doesn't have a class_weight parameters that takes into account that the dataset is unbalanced. So, in order to obtain a balanced dataset, we perform downsampling. The main idea of this technique is to remove observations from the majority classes, so that all the classes have the same size (of the class which has the lowest number of observations (class 1)).

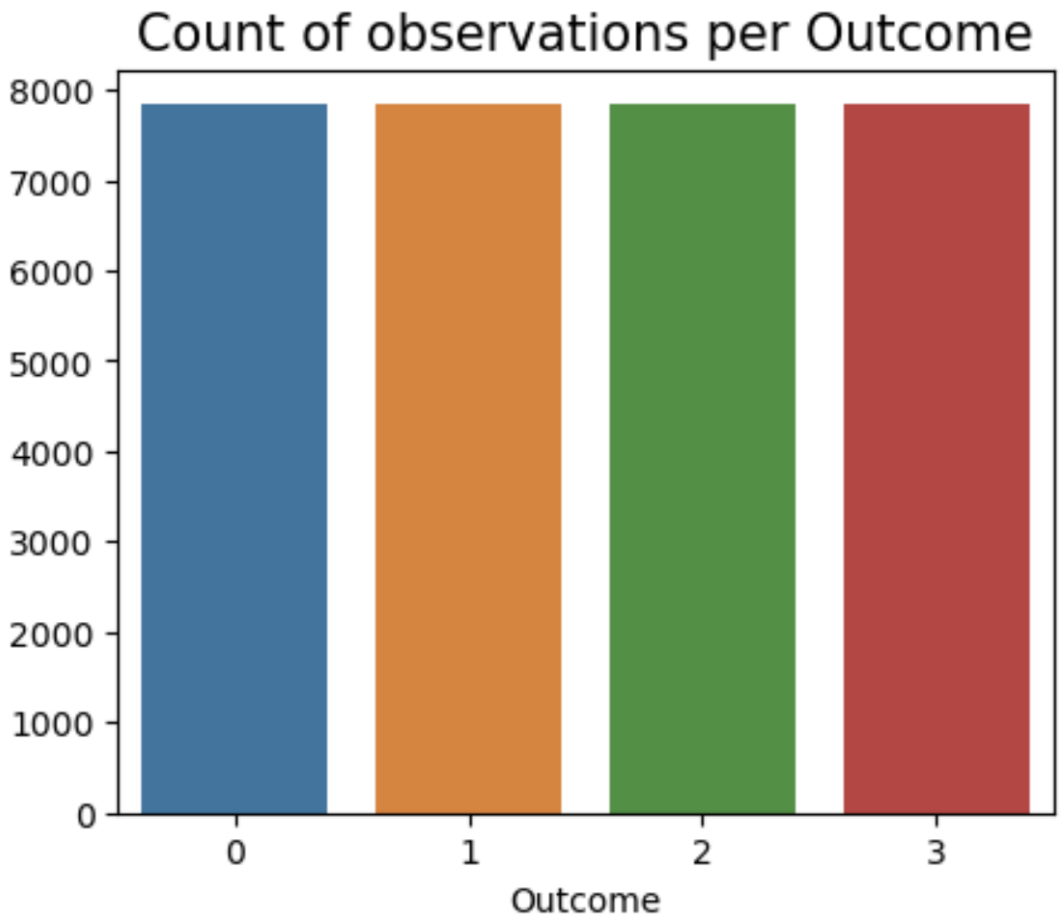


Figure 7: *Count of the number of observations for class in the balanced dataset*

Once we have created this balanced dataset, we build a gradient boosting model. These are the best parameters:

- booster = gbtree,
- num_class = 4,
- objective = multi:softmax,
- eval_metric = merror,
- eta = 0.01,
- subsample = 1,
- colsample_bytree = 0.5,
- tree_method = exact,
- verbosity = 0,
- max_depth : 10.

. Also in this case, for the test set we have a micro-averaged f1 score = 0.38.

4.2 Neural Networks

Artificial Neural Networks (ANNs) are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex Machine Learning tasks requiring a certain capacity of abstraction over data. There are numerous variants of Neural Networks. Convolutional Neural Networks perform well for image recognition; LSTM networks fit well time-series problems and speech recognition problems, just to name a few.
In our problem, we consider a particular architecture known as Multilayer Perceptron (MLP). An MLP can be considered the simplest form of ANN, but it can reach the desired level of complexity and therefore abstraction to solve both Regression and Classification problems such as classifying hand-written digits. An MLP is composed of one input layer, one or more hidden layers and one output layer. All the neurons in a layer are connected to every neuron in the previous layer (fully connected layer or Dense layer). Every layer except the output layer includes a bias neuron and is fully connected to the next layer. With the non linear activation function called here Φ, we can represent the full transition of activation from one layer to the next with :

$$a^{(1)} = \Phi(Wa^{(0)} + b) \tag{1}$$

Where :

- $a^{(1)}$ are the value contained in the layer $n + 1$
- $a^{(0)}$ are the value contained in the layer n
- W is the weight matrix
- b is the bias vector

The signal flows from input to output, allowing us to call this architecture, an example of feed-forward Neural Networks. This architecture is often trained with back-propagation algorithms using Steepest Gradient Decent (SGD) or SGD-like methodologies.Weights and biases are initialised randomly and are tweaked to minimise a certain cost-function. In a Regression problem the loss function is typically the mean-squared-error (MSE) or the mean absolute error (MAE). In a Classification setting the loss function is typically the cross-entropy function. In this articular case, if the response variable is made up of more than two classes, and if each instance can belong only to a single class, we need one output neuron per class activated by the softmax activation function. It ensures that all the estimated probabilities are between 0 and 1 and that they add up to 1. This case is called multiclass Classification. In Regression, in general you don't use any activation function for the outputs neurons so they are free to output any range of values. However, if you want to guarantee that the output will always be positive, then you can use the ReLU activation function in the output layer and if you want that the predictions will fall within a given range of values, you can use the logistic function.
We have built a Regression Neural Network in order to predict *Settlement* (task 1) and *Duration* (task 2) and a Classification Neural Network in order to predict *Outcome* (task 2)

```
model = keras.models.Sequential([
    keras.layers.Dense(64, activation="relu", kernel_initializer="he_normal",input_shape=X_train.shape[1:]),
    keras.layers.Dense(128, activation='relu', kernel_initializer="he_normal"),
    keras.layers.Dense(64, activation='relu', kernel_initializer="he_normal"),
    keras.layers.Dense(30, activation="relu", kernel_initializer="he_normal"),
    keras.layers.Dense(1)
])

model.compile(loss='mae', optimizer=keras.optimizers.Adam(learning_rate=0.015))
```

Figure 8: *MLP chosen architecture*

train_s : We apply the following architecture : As you can see from the *figure 8*, there are 4 hidden layers, one input layer and 1 ouput layer. We have chosen the ReLu activation function. In particular, the skeleton of the MLP has been compared with numerous proven architectures, and has been empirically chosen from the benchmark. We notice that the model has not prompt to overfit and we consider inappropriate the use of "Dropout" gates even in order to dampen a surplus of complexity that could come from the irregularity and the inconsistency of the original dataset. We purposely pick a small learning rate to reduce the risk of overshooting a local minimum of the loss function that has been defined as *MAE*. The optimizer is the SGD-like optimizer *Adam*, often mentioned in the literature and used in the state-of-the-art of similar problems. For the sake of reproducibility, we set a random seed. The different parameters are displayed in *figure 9* :

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_5 (Dense)             (None, 64)                2432

 dense_6 (Dense)             (None, 128)               8320

 dense_7 (Dense)             (None, 64)                8256

 dense_8 (Dense)             (None, 30)                1950

 dense_9 (Dense)             (None, 1)                 31

=================================================================
Total params: 20,989
Trainable params: 20,989
Non-trainable params: 0
_________________________________________________________________
```

Figure 9: *MLP parameters*

We use the training set for fitting the model and the validation set for evaluating it. This Neural Network reaches, on the test set, a MAE lower than 760, surpassing the performance of every other model implemented so far.

train_od : As we have said before, we apply a Neural Network both for Regression and Classification task. In order to predict *Duration* we have built a little bit more complex Neural Network than the one built for train_s. (*figure 10*) .

```
NN_reg = keras.models.Sequential([
    keras.layers.Dense(128,activation = 'relu',kernel_initializer='he_normal',
                       input_shape = X_train.shape[1:]),
    keras.layers.Dense(256,activation = 'relu',kernel_initializer='he_normal'),
    keras.layers.Dense(512, kernel_initializer='he_normal',activation='relu'),
    keras.layers.Dense(256,activation = 'relu',kernel_initializer='he_normal'),
    keras.layers.Dense(128,activation='relu',kernel_initializer='he_normal'),
    keras.layers.Dense(1,activation = 'relu') #relu because i want only positive numbers
])
```

Figure 10: *MLP chosen architecture*

Besides the different parameters are displayed in *figure 11*

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               5376

 dense_1 (Dense)             (None, 256)               33024

 dense_2 (Dense)             (None, 512)               131584

 dense_3 (Dense)             (None, 256)               131328

 dense_4 (Dense)             (None, 128)               32896

 dense_5 (Dense)             (None, 1)                 129

=================================================================
Total params: 334,337
Trainable params: 334,337
Non-trainable params: 0
_________________________________________________________________
```

Figure 11: *MLP's parameters*

Here again, the selection of hyper-parameters, especially the number of neurons and the number of hidden layers has been done empirically by benchmarking

different architectures. The loss and the optimizer are the same used for the Regression task in train_s. In the test set we reach a mean absolute error of 212. For the Classification task we consider the same data set that we used for XGBoost (balanced dataset created with downsampling). The architecture is empirically chosen (*figure 12*).

```
NN_class = keras.models.Sequential([
    keras.layers.Dense(256,activation = 'relu',kernel_initializer='he_normal',
                       input_shape = X_train.shape[1:]),
    keras.layers.Dense(512,activation = 'relu',kernel_initializer='he_normal'),
    keras.layers.Dense(256,activation = 'relu',kernel_initializer='he_normal'),
    keras.layers.Dense(128,activation = 'relu',kernel_initializer='he_normal'),
    keras.layers.Dense(64,activation = 'relu',kernel_initializer='he_normal'),
    keras.layers.Dense(32,activation = 'relu',kernel_initializer='he_normal'),
    keras.layers.Dense(4,activation = 'softmax')
])
```

Figure 12: *MLP chosen architecture*

As you can see there are 6 hidden layer, 1 input layer and 1 output layer with 4 neurons.
The loss used is *"sparse_categorical_crossentropy"*, the optimizer is *"Adam"*, the activation function of the output layer is *"Softmax"* and the metric used is the accuracy. In this case, for the test set we have a micro-averaged f1 score = 0.39.

4.3 MARS

The last model that we have applied (just for train_s) is The Multivariate Adaptive Regression Splines (MARS). MARS is an adaptive procedure for Regression, and it is well suited for high-dimensional problems. It can be viewed as a generalization of stepwise linear Regression. It is an algorithm that creates a piece-wise linear model which provides an intuitive stepping block into non-linearity after grasping the concept of multiple linear Regression. Basically, it tries to capture non-linear relationships in the data by assessing knots similar to step functions.
Here is shown the model for our dataset: we tune a grid search and the best parameters are nprune = 40 / degree = 3 if we consider Mean Squared Error as the metric, and nprune = 5 / degree = 2 if it is Root Mean Squared Error. Last but not least, the method used for pruning is Cross Validation.
We have applied Bagging and calculated the MAE, obtaining a result of 847.1232 for the test set.
Finally, we have trained the model considering the out-of-bag and recalculated the MAE, obtaining a result of 846.4701.

5 Results

It's time to take a look to the results for both the datasets.

train_s : We apply different models such as Random Forests, Gradient Boosting, Neural Network and MARS. The model with the best results is Random Forests and for this reason we have used it for making predictions in the blind test set. Moreover, these 3 models show a very high error when *Value* is between 5200 and 26000. For this reason, we try to build an algorithm that is able to reduce the error when Value is in the above-mentioned range.

```
ranges_value
(1000,1100]         5369
(1100,5200]        48556
(26000,52000]         61
(5200,26000]        9539
(52000,260000]         8
(520000,inf)           2
[0-1)                748
[1,1000]           32308
dtype: int64
```

Figure 13: *Number of observations for each range*

As we can see from this picture, there are only few observations where *Value* is in this range. To make the algorithm capable of predicting these observations, we apply upsampling. So, we build an upsampled dataset (the (5200, 26000] class will be upsampled to the (1100,5200] range, where there is the highest number of observations). The model we have chosen is a Neural Network which has a very similar architecture to the one introduced in the previous section. Unfortunately, the results are not so good as we expect. We also try to downsample the dataset but with the same negative results.

train_od : Regarding the Regression task, we apply Random Forests, Gradient Boosting, Neural Network. Respect to the train_s dataset, the best model is the Neural Network, then, used for the predictions in the blind test set.
Let's put the attention on the Classification task. The models in section 4 are built considering a balanced dataset. In particular, the best model is the Neural Network.
Moreover, we also consider the case where the datasset is not balanced. Here, the results seem to be very promising. In fact, the micro-average f1 score for a Neural Network (unbalanced dataset) is 0.59 which is much greater than the micro-average f1 score 0.39 in the previous Neural Network (balanced dataset).
However, as we can notice from the confusion matrix, this model (Neural Newtork - unbalanced dataset) is not able to predict the classes correctly, except for class 0 which is the easiest to predict because the most frequent.

NN Confusion Matrix			
11878	20	2055	112
869	**28**	209	13
5313	12	**2609**	118
1270	2	417	**272**

Table 3: *Neural Network (Unbalanced) - Confusion Matrix*

It is important to underline that when we deal with an unbalanced Classification problem, it is very easy to build an algorithm that predicts always the most frequent class. For this reason, we prefer to consider a model with a lower micro-average f1 score but able to predict all the classes, as we can see from the confusion matrix (Neural Network - balanced dataset).

NN Confusion Matrix			
4607	2990	4079	2389
133	**621**	227	138
1323	1542	**3565**	1622
261	222	430	**1048**

Table 4: *Neural Network (Balanced) - Confusion Matrix*

6 Explainer

This section is meant to be a useful hand to understand our model and which variables mainly affect the output prediction.

6.1 SHAP

One technique is the SHAP method and we use it to give a general look to the predictions. It shows the contribution of each variable but it does not evaluate the quality of the predictions.

train_s :

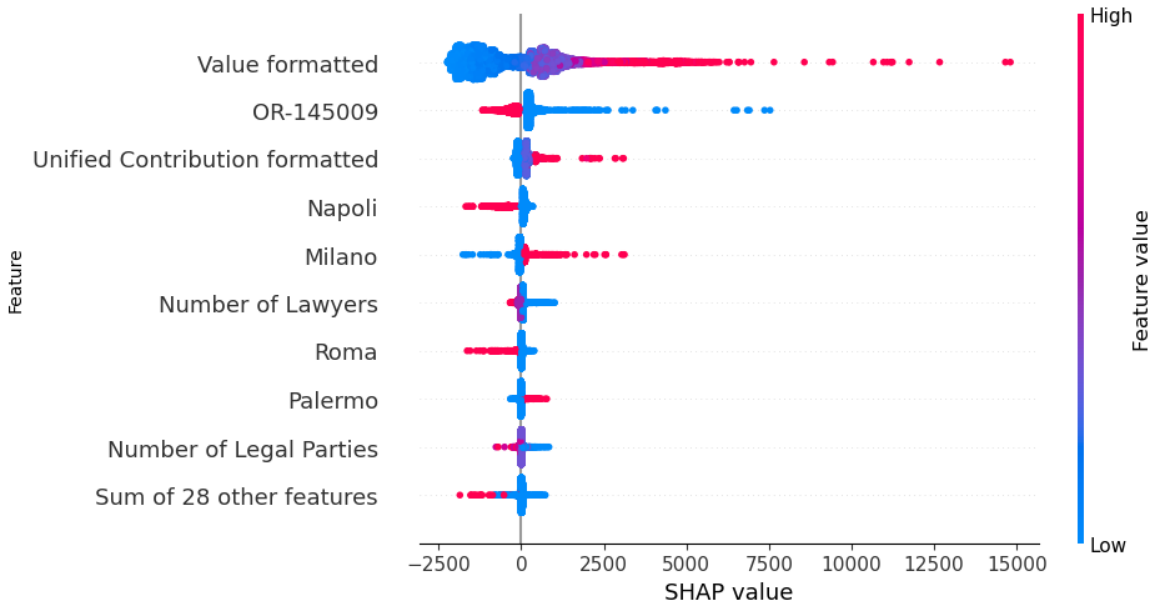


Figure 14: *SHAP beeswarm plot for a general overview*

We can notice that *Value formatted* is the main variable that affect the predictions. On the right we found how to interpret the colors, so points are blue if the value of the variable is low (e.g. *Value formatted* = 5) and it changes until becoming red which represents an high value (e.g. *Value formatted* = 52000). On the x axis there is the SHAP value which represents how much a variable is important. It is necessary to underline that it is scaled with the scale of the variable itself, which might lead to distortions and misinterpretations.
It is curious to notice that *Unified Contribution formatted* follows a similar trend to *Value formatted* but squashed and it can be due to their relationship (table 1 and 2).

6.2 LIME

Another explainer technique is the LIME method and we use it to look inside the model and analyze which variables mainly affect a single prediction. This method provides local model interpretability. While the model may be very complex globally, it is easier to approximate it around the vicinity of a particular instance and it is done considering a very small area where it is possible to approximate the model to a linear one.

train_s : In this plot, on the x axis we find the coefficients (linear regression) of the variables and on the y axis there is the sorted by importance list of features.

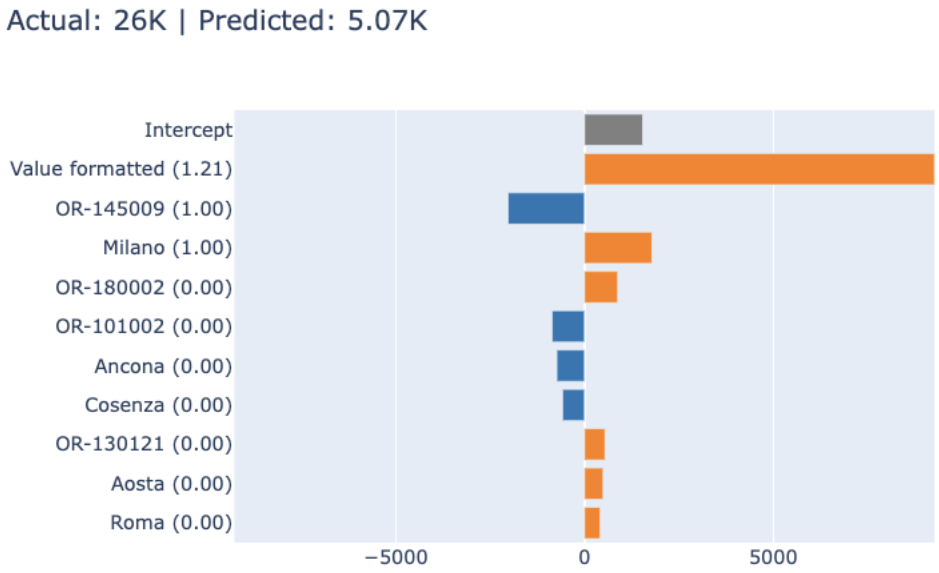


Figure 15: *LIME waterfall plot for predictions with a high residual*

Figure 16 shows the plot of a bad prediction (high residual with respect to the true value). It is evident that *Value formatted* is the variable that impact most on this prediction and the others are quite "silent". It is a trend which is repeated for predictions which show a high residual. Another pattern that pools these predictions is the "positive" direction of *Value formatted*.

On the other part, regarding predictions with a low residual, *Value formatted* is not always the most important feature and neither the only one which significantly affects them. Another difference we can notice is that for these predictions, *Value formatted* has a "negative" direction.

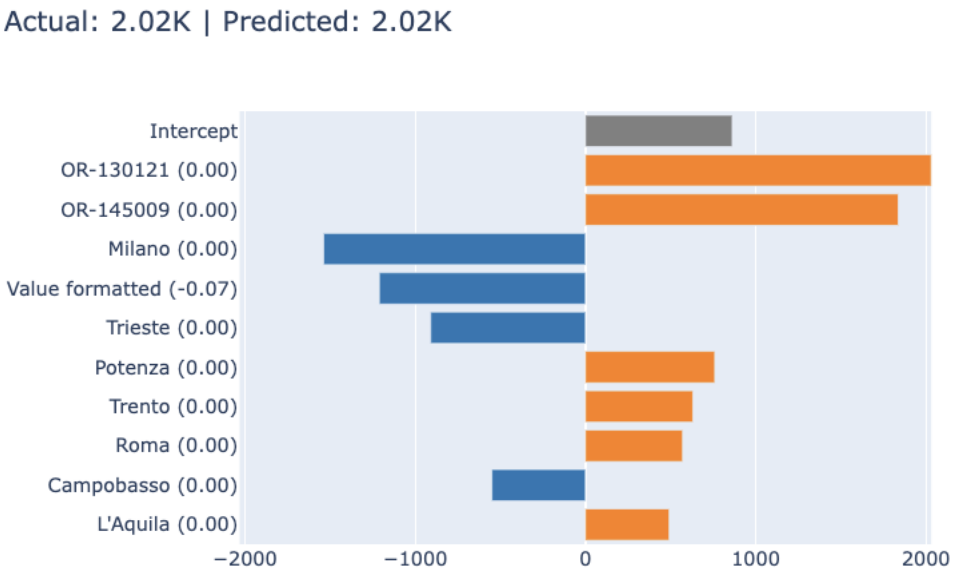


Figure 16: *LIME waterfall plot for predictions with a low residual*

train_od : For train_od we have built a plot (different only in aesthetics from the train_s ones) and we notice that "Venezia" (always "negative" direction) is the most important feature in most of the predictions and it can be confirmed by looking at the *Cities of the Judge's Office* with the highest average *duration*. There are other cities that recur often in the first places as "L'Aquila".

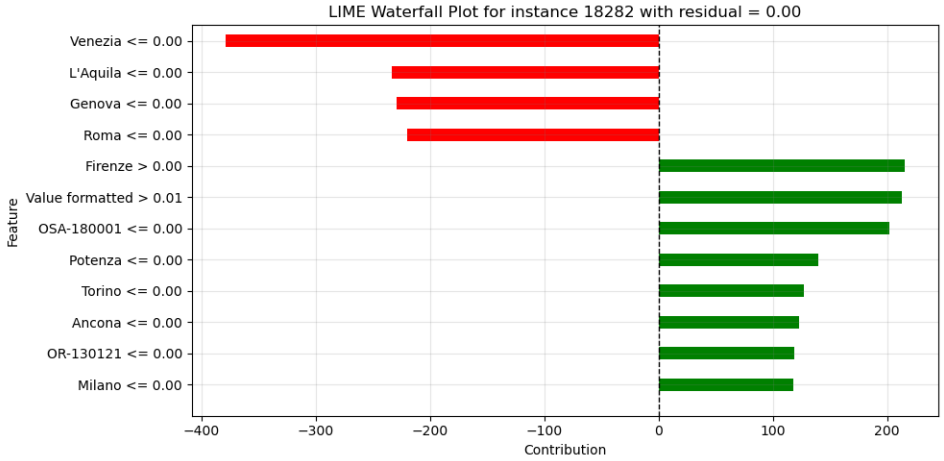


Figure 17: *train_od: LIME waterfall plot for predictions with a low residual*

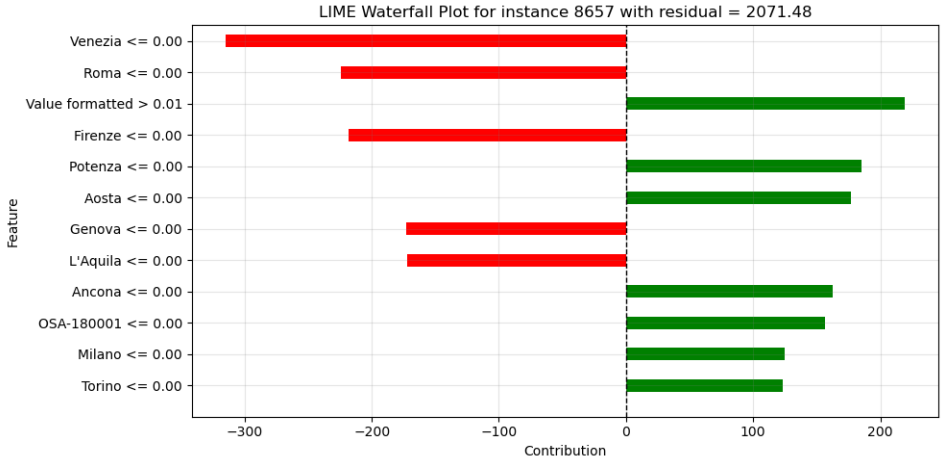


Figure 18: *train_od: LIME waterfall plot for predictions with a high residual*

It is important to highlight the comparison between "Venezia" and "Milano" because we notice that Lombard city has a "positive" direction and to confirm this, it shows a very lower mean than the Venetian city.

Figure 19 and 20 show two LIME waterfall plots, one representing predictions of Accepted lawsuits and the other Denied ones. These two plots are apparently very similar and a domain specialist would have been necessary to understand better how to deduce the maximum from them. However, we can highlight that there are often the same variables in the first positions (cities as "Aosta", "Trento" and objects like "OR-145009" and "145999"), therefore they are the most meaningful ones.

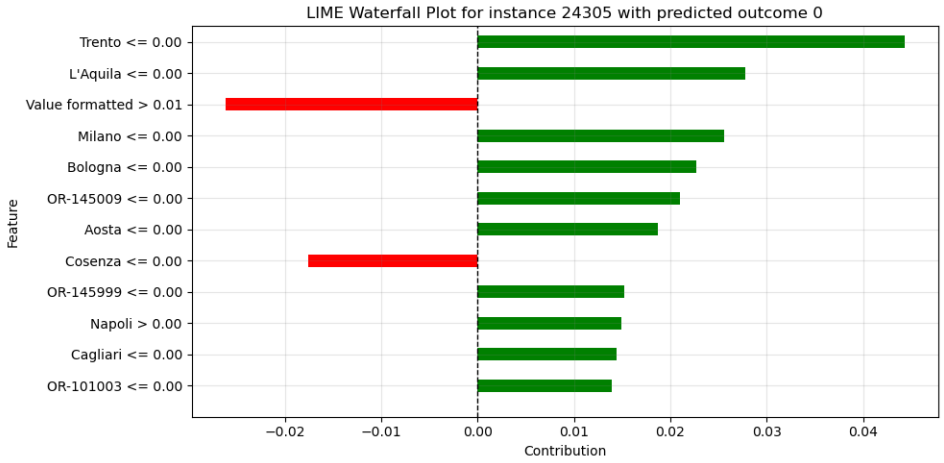


Figure 19: *train_od: LIME waterfall plot for predictions of Outcome = 0 (Accepted)*

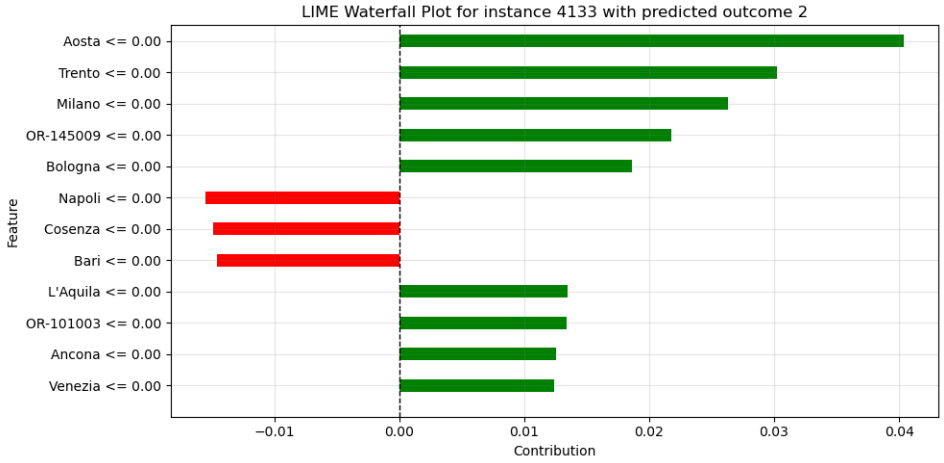


Figure 20: *train_od: LIME waterfall plot for predictions of Outcome = 2 (Denied)*

References

[1] Hastie, T., Tibshirani, R. and Friedman, J. H. (2009) *The Elements of Statistical Learning: data mining, inference, and prediction*. 2nd ed. New York, Springer.

[2] Aurélien Géron (2019) *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly

[3] Bradley Boehmke and Brandon Greenwell (2020) *Hands-On Machine Learning with R*. 1st ed. Chapman and Hall/CRC;

[4] https://towardsdatascience.com/