University of Granada

Business Intelligence

Practice 3: Competition on Kaggle: used car price prediction (multiclass)

Author:
Mikhail Raudin
Practice Group 1
mraudin@correo.ugr.es

Lecturer:
Dr. Daniel Molina
dmolinac@ugr.es

January 4, 2021

Contents

List of Figures

1 Introduction

The goal of this practice is to solve a multiclass classification for a kaggle competition[1]. The dataset contains data about cars sold, and the attribute to predict is the car price that has been classified into five categories (1- represents the cheapest, 2- represents those cheaper then average, 3- represents those who are in average price, 4- represents those who are more expensive than the average, and 5- which represents the most expensive cars). The training set contains around 6000 samples and 13 attributes, both categorical and numerical data. The class distribution is unbalanced. As an evaluation measure the prediction accuracy is used.

The best results were obtained by the XGBoost Classifier (Extreme Gradient Boosting Algorithm). A score of 0.80845 is achieved on the test set (0.8401 on the training set).

The documentation will describe the different methods, including the data preprocessing applied in the best result and mention other strategies that were tried out during the competition. At the end of the document are fig.15, fig.16, fig.17 containing the table with information about every submission made to kaggle.

2 Data Preprocessing

This section will describe in detail the methods used to obtain the best score of 0.80845, which are

- imputing missing values with the most frequent value
- Label Encoding
- Feature Selection (Correlation Matrix, Chi-Square, mutual information gain)
- oversampling the minority class using BorderlineSMOTE

Furthermore, other techniques used such as

- removing characters from attributes,
- imputing missing values with the mean value,
- Binary and One Hot Encoding,
- and oversampling the minority class using SMOTE

will be mentioned.

[1] `https://www.kaggle.com/c/ugrin2020-vehiculo-usado-multiclase/leaderboard`

Removing Characters

Fig.1 shows that the attributes "Potencia", "Consumo" and "Motor CC" are numerical attributes, but the numerical values of those attributes contain their meaning as well, such as "bhp" for Potencia, "kmpl" for Consumo and "CC" for "Motor CC". So this numerical attributes are of the type String, or in our case a python object at the moment. To handle this three attributes as purely numerical, a function is created that removes the chars of meaning from the values in the columns. The result is seen in fig. 1.

Consumo	Motor_CC	Potencia
23.4 kmpl	1248 CC	74 bhp
20.51 kmpl	998 CC	67.04 bhp
25.32 kmpl	1198 CC	77 bhp
18.5 kmpl	1197 CC	80 bhp
18.7 kmpl	1199 CC	88.7 bhp

Consumo	Motor_CC	Potencia
23.40	1248.0	74.00
20.51	998.0	67.04
25.32	1198.0	77.00
18.50	1197.0	80.00
18.70	1199.0	88.70

Figure 1: The values of the columns "Potencia", "Consumo" and "Motor CC" after removing the characters "bhp" , "kmpl" and "CC".

Missing Values

The first idea was to impute the missing values of numerical and categorical attributes separately. For the numerical attributes, the missing values are exchanged by the mean value of the attribute in the training set. For the categorical attributes the most frequent value is used. Due to unsolved complications using the SimpleImputer class of scikit-learn, which did not remove all NaN values at once and so at the second iteration original values were changed. For this reason, as well as a mistake applying normalization, the first four submissions to kaggle reached only a score around 0.20.

When I discovered the error source, I decided to use the pandas mode() function, another way to impute missing values. This time the missing value is imputed by the most frequent value for the numerical as well as the categorical attributes. This technique is used in all the submissions after the fourth one and also leads to the best score obtained.

Labeling Algorithms

Due to the fact that categorical attributes are part of the dataset, a labeling strategy is needed to transform them into numerical values, interpretable by the

algorithms. Three encoding techniques were tried out:

- Label Encoding
- Binary Encoding
- One Hot Encoding

The best result is obtained by performing the Label Encoding on all of the attributes. A Label Encoding is a very simple method, which maps each unique value of an attribute to a number starting from 0. For example, the attribute "Ciudad" has 11 categories described by letters ('G', 'I', 'F', 'E', 'H', 'C', 'J', 'L', 'B', 'K', 'D'). Those are transformed to numbers ranging from 0 to 10, each one representing a letter.

The problem of Label Encoding is that its solution implies an order/ranking in the attribute values. Moreover, the distance between city G and D is 10, but between G and I it is 1, implying a bias that they are more similar, which does not have to be the case. In our dataset, we have several categorical attributes with no order, so the Label Encoding creates unwanted side effects. The attributes "Nombre", "Ciudad", "Combustible" and "Tipo marchas" are effected. The only categorical attribute with an order is the "Mano" (how many previous owners it had).

To overcome the issues of the Label Encoding, a popular solution is the One Hot Encoding algorithm. One Hot Encoding creates a new feature column for every unique value of the attribute. For our example attribute "Ciudad", 11 new features are created, each one of them representing one of the unique values. A vector is created, and every training sample has 0 for all the new feature columns representing "Ciudad" besides the one representing the original "Ciudad" value, this column has a 1. Fig. 2 shows the One Hot Encoding of "Ciudad".

Surprisingly, the One Hot encoded training data doesn't improve the results. Also, this method has a drawback in our case: the data has 5 categorical attributes and the attribute "Nombre" has 1600 unique values. Applying One Hot Encoding leads to a dimensional explosion, increasing the training time of the algorithms.

Binary Encoding is tested to reduce the dimensionality problem of the One Hot Encoding solution. Binary Encoding has a very similar functionality to One Hot Encoding, the important difference is that it encodes each unique value in a binary representation with base 2. So a 3 would become "011" and an 8 "1000". For each binary digit a new feature column is added to the dataset. In our case we end up with 33 columns in total (20 new columns, and just 11 new columns for the attribute "Nombre") instead of 1905 columns with One Hot Encoding. Therefore, it creates less new dummy feature columns and is more memory efficient. The final results did not improve applying Binary Encoding.

To sum up, Label Encoding creates the best results. The reason could be that the decision tree based algorithm XGBoost is used for the predictions, not taking into account the distance between values. One Hot Encoding generates sparsity to the dataset and when the tree algorithm tries to split a categorical attribute, it is left with only two values, 0 and 1, to split on.

Ciudad_B	Ciudad_C	Ciudad_D	...
0.0	0.0	0.0	...
0.0	0.0	0.0	...
0.0	0.0	0.0	...
0.0	0.0	0.0	...
0.0	0.0	0.0	...
...	...	...	...
0.0	0.0	0.0	...
0.0	1.0	0.0	...
0.0	0.0	0.0	...
1.0	0.0	0.0	...
1.0	0.0	0.0	...

Figure 2: Some of the new dummy feature columns created by One Hot Encoding for the attribute "Ciudad".

2.1 Feature Selection

Correlation Matrix

Fig. 3 shows the correlation matrix of the numerical attributes. Looking at the last row we can see how high the correlation is between each attribute and the target we try to predict, the price category. "Descuento" (Discount) has the highest correlation, but it also has more then 75% of its values missing, so we remove this feature because a missing value imputation does not make sense. "Ano" (Year) has the highest correlation apart from "Descuento" so we keep the feature. We can see that "Kilometros" correlation with the price category is almost 0, so it does not provide any useful information for the prediction. Before leaving the feature "Kilometros" out, other techniques are observered to support this hypothesis.

Chi-Square Test

We use the Chi-Square Test to calculate the degree of association between the categorical features and the target value. If a feature has a low Chi-Square score, then it is independent from the target value and not useful for predicting it. Higher Chi-Square scores mean that the feature is useful for model training, because the target value is dependent on it. Fig. 4 shows that "Ciudad" and "Mano" have the lowest values and "Nombre" the highest.

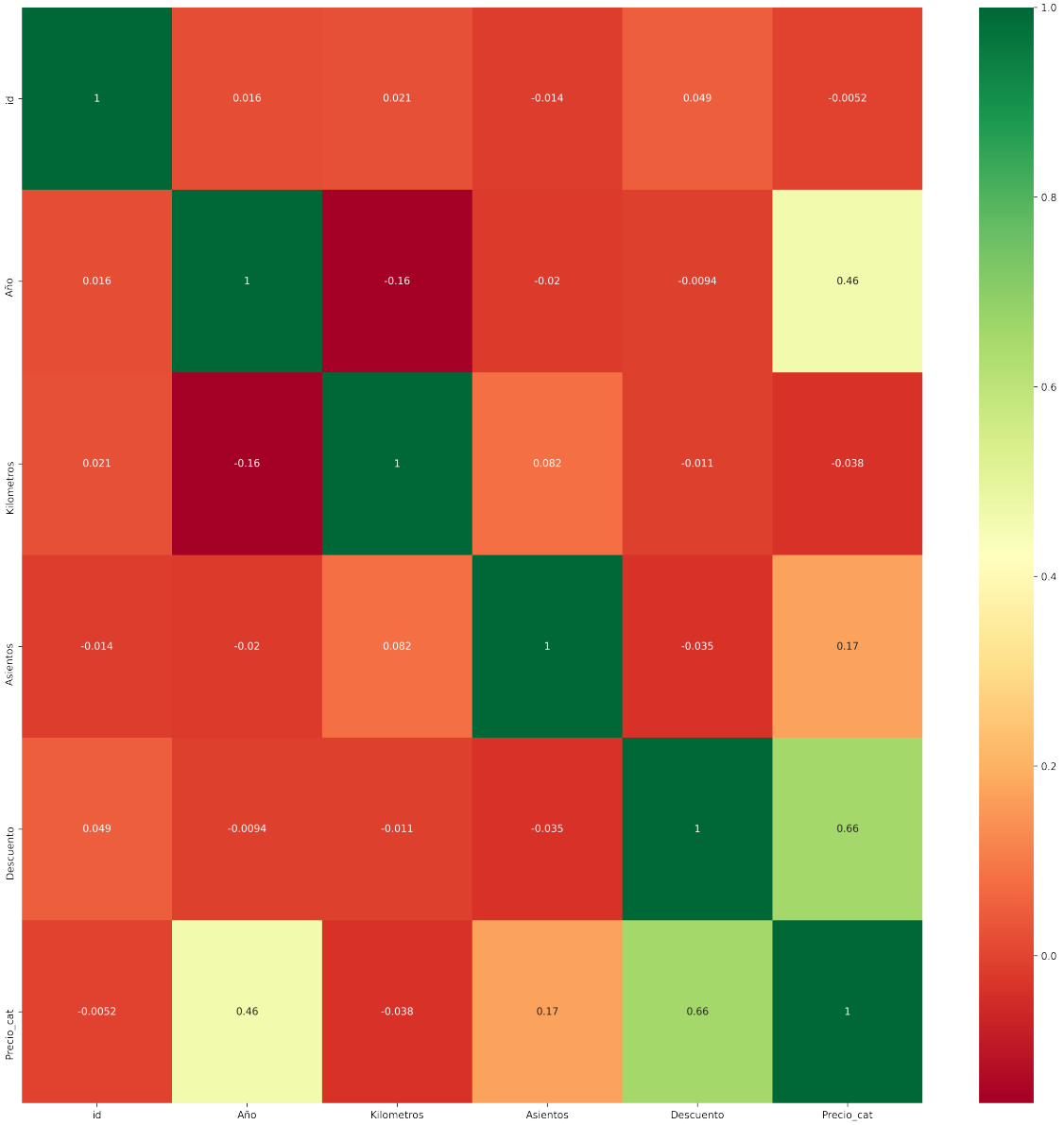

Figure 3: The correlation matrix of the numerical attributes.

	Specs	Score
0	Nombre	17788.838354
2	Combustible	744.179186
3	Tipo_marchas	528.977476
4	Mano	380.397925
1	Ciudad	22.114929

Figure 4: The Chi-Square scores of the categorical attributes.

Mutual information

Another measurement we apply to explore which features are important is the mutual information, the application of information gain from information theory. It describes the reduction in the level of uncertainity for the target value given a known value of a feature. So larger mutual information values are better. For the categorical features fig. 5 shows that again "Ciudad" and "Mano" have the worst scores. This measurement can also be applied to numerical features and fig. 6 shows that "Kilometros" provides the least information for the price category, supporting the results of the correlation matrix.

	Specs	Score
0	Nombre	0.778054
3	Tipo_marchas	0.182767
2	Combustible	0.100694
1	Ciudad	0.052041
4	Mano	0.015367

Figure 5: The mutual information scores of the categorical attributes.

	Specs	Score
4	Potencia	0.636697
2	Consumo	0.518869
3	Motor_CC	0.498039
0	Año	0.162878
5	Asientos	0.076943
1	Kilometros	0.020727

Figure 6: The mutual information scores of the numerical attributes.

Feature Importance XGBoost

Lastly, the feature importance calculated by the trees of the XGBoost Algorithm is shown in fig. 7. It supports our previous observation, as we can see that "Kilometros" has the lowest importance, followed by "Mano" and "Ciudad".

Table 1 shows the training accuracy scores of a cross validation on different subsets of the training data based on this analysis. Leaving out the not relevant features "Kilometros" and "Mano" improves the score and reduces the model complexity.

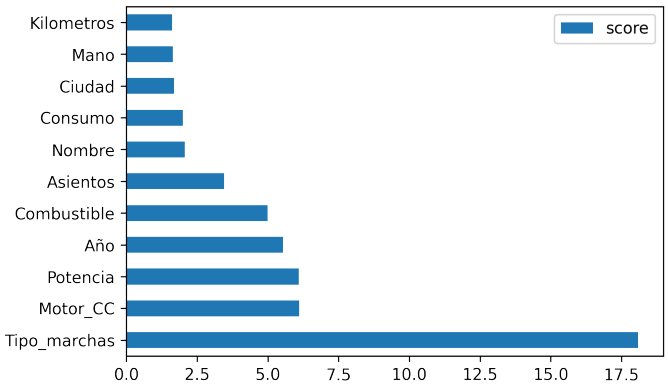

Figure 7: The feature importance returned by the XGBoost Alogrithm based on information gain (the average gain across all splits the feature is used in).

Table 1: This table compares the accuracy scores of the XGBoost algorithm on different subsets of the training sets.

subset	accuracy score
no Kilometros, Mano	0.8419
no Kilometros	0.8394
no Mano	0.8371
full training set	0.8369
no Ciudad	0.8172
no Kilometros, Ciudad	0.8137
no Kilometros, Mano, Ciuadad	0.8116

2.2 Balancing Data

Looking at the class distribution of the training samples, we see that there are 2211 samples in class 3, 978 in class 3, 759 in class 5, 602 in class 2 and only 269 in class 1. In the minority class (class 1) there are only 12% of the amount of samples in the majority class (class 5). This is a moderate class imbalance and handling it could improve the classification results. Table 2 sums up the results of the different balancing approaches which are explained in this subsection, the approaches being numbered as followed:

- 1: no balancing
- 2: SMOTE default parameters
- 3: SMOTE, oversampling only minority class to 400, k-neighbors=3
- 4: BorderlineSMOTE, oversampling only minority class to 400, k-neighbors=2

- 5: SMOTE, oversampling only minority class to 400, k-neighbors=3; and RandomUnderSampler, undersampling only majority class to 2000

The result scores are obtained by performing the oversampling within the cross-validation (for each of the k=5 folds), fig. 8.

```
xgb = XGBClassifier(objective="multi:softmax",scale_pos_weight=1,learning_rate=0.1,
                    colsample_bytree = 0.8,subsample = 0.8, n_estimators=400,
                     reg_alpha = 0.3, max_depth=4, gamma=0.6)

samples = {1: 400, 3: 2211, 2: 602, 4: 978, 5: 759}
samples_under = {1: 400, 3: 2000, 2: 602, 4: 978, 5: 759}

over = BorderlineSMOTE(sampling_strategy=samples, k_neighbors=3,
                       n_jobs=-1, random_state = 42)
#under = RandomUnderSampler(sampling_strategy=samples_under)

steps = [('over', over),
         #("under", under),
         ('model',xgb)]
pipeline = Pipeline(steps=steps)

# evaluate pipeline
cv = RepeatedStratifiedKFold(n_splits=5, n_repeats=1, random_state=1)
scores = cross_val_score(pipeline, X_noK_noM, y, scoring='accuracy', cv=cv, n_jobs=-1)
```

Figure 8: The application of oversampling with SMOTE within cross validation.

Table 2: This table compares the accuracy scores of the XGBoost algorithm combined with different class balancing techniques. Strategy: 1: no balancing. 2: SMOTE default parameters, 3: SMOTE, oversampling only minority class to 400, k-neighbors=3. 4: BorderlineSMOTE, oversampling only minority class to 400, k-neighbors=2. 5: SMOTE, oversampling only minority class to 400, k-neighbors=3; and RandomUnderSampler, undersampling only majority class to 2000

strategy	train accuracy	test accuracy
1	0.8424	0.80155
2	0.8095	- (no submission)
3	0.8425	0.80241
4	0.83918	0.80845
5	0.83856	- (no submission)

SMOTE

SMOTE is short for Synthetic Minority Oversampling Technique and a type of data augmentation, because new samples are synthesized from existing data,

instead of just copying existing data to create more samples but not gain any information. SMOTE creates new samples by randomly selecting a sample from the minority class, finding its k-nearest minority class neighbors, then selecting randomly one of those neighbors and creating a new synthesized sample at a random point in the feature space between the two picked samples.

In the first try, we over-sample all classes with the default parameters of SMOTE, so each class has 2211 samples after the application. This reduced the training accuracy score significantly to 0.8095, probably because more noise is created then new useful information. After more research on the internet, it seems more useful to only oversample the minority class and to play around with the sampling strategy, providing a dictionary with the number of samples per class wanted after using SMOTE. After trying some values for all of the classes besides the majority, that means, oversampling all classes a bit, and also trying different values for just the number of minority class samples desired, the best score is obtained by oversampling only the minority class from 269 to 400 samples, nearly doubling the amount. Besides that, the parameter k for the nearest neighbors was tuned between 1 and 7, the best results gives k=2 (0.8425).

BorderlineSMOTE

BorderlineSMOTE is a version of SMOTE that is more selective about the samples from the minority class that are chosen to create new synthesized samples. It picks only instances of the minority class that are close to the decision boundary and that were misclassified. Only this data points are oversampled, providing more information in the critical region, where lots of misclassification can happen. The samples selected by BorderlineSMOTE are more important for the classification process and therefore we hope to improve our algorithm predictions applying this method instead of plain SMOTE.

After the knowledge gained from applying SMOTE on the car dataset, we keep the sampling strategy the same, that is, oversampling minority class 1 from 269 to 400 samples. Only the hyper parameter k for the nearest neighbors is tuned, the best result gives k=2.

Further techniques

The SMOTE algorithm might perform better when combined with under sampling the majority class. So we add the RandomUnderSampler to the model pipeline and after oversampling the minority class with SMOTE to 400 samples, we undersample the majority class from 2211 to 2000 samples. The number of final majority samples was varied, and 2000 provides the best accuracy score.

Moreover, SVMSMOTE and ADASYN are tried for the oversampling, but do not improve the results.

3 Algorithms

This section describes the XGBoost algorithm that is used to generate the best predictions, as well as the hyper-parameter tuning process. The explanation of early stopping, applied to reduce overfit, as well as a brief summary of other algorithms put into practice is provided.

3.1 XGBoost Classifier

The XGBoost algorithm (eXtreme Gradient Boosting) is an optimized version of the Gradient Boosting Algorithm. Gradient Boosting is an ensemble technique using multiple models, in our case decision trees. The core point is Boosting, a technique that builds models in a sequential manner, by minimizing the errors from the previously build models in the ensemble. At the same time, it tries to increase ("boost") the influence of the best-performing models within the ensemble. This way new models are added until no further improvements are made or the number of maximum trees in the ensemble is reached. Gradient Boosting is just a version of Boosting that uses the gradient descent algorithm for the error minimization of the sequential models. XGBoost then optimizes the Gradient Boosting method, by using parallel processing, tree-pruning and regularization to avoid overfitting and bias. XGBoost was very successful in a wide range of kaggle competitions and machine learning benchmark datasets. Further, it performs well on structured/tabular data, which fits to our car dataset.

3.2 Hyper-parameter Tuning

We need to optimize the parameters of XGBoost to fit to our dataset and problem context. We will run a grid search with a 5-fold cross validation for each parameter combination. There are a lot of parameters, and in the first run we just focus on the two main ones:

- number of trees in the ensemble (n estimators): more trees are usually better. Search range: from 30 to 150, steps range: 20.
- learning rate: creates a more robust model by slowing down the learning in the gradient boosting to prevent overfitting. Search range: [0.1,0.2,0.3,0.4,0.5].

The best combination is 70 estimators for the number of trees and a learning rate of 0.5.

The second run consists of trying out more important parameters that need to be tuned in order to reduce the overfit and bias of the algorithm, so it generalizes well and predicts great on the unseen test data.

- maximum depth of each tree: Higher values allow the model to create very specific trees for particular data samples, hence lower values avoid overfitting and simplify the model. Search range: 2-10.

- minimum child weight: defines the minimum sum of weights of all samples required in a child, prevents overfitting to particular data samples. High values can cause underfitting. Search range: [1,3,5,7]
- gamma: a regularization parameter that specifies the minimum loss reduction required to make split, prevents overfitting.
- colsample bytree: the number of columns randomly selected to be used as samples for each tree. Search range: [0.3,0.4,0.5,0.7]

Keeping the learning rate=0.5 and estimators=70, the best combination is maximum depth=4, minimum child weight=3, gamma=0 and cosample bytree=0.7. It achieves an accuracy of 0.8373 on the training data.

To further improve the prediction, another grid search over the number of estimators and learning rate is done, this time focusing on small learning rates in combination with a higher number of estimators (which small learning rates require).

- number estimators search range: [100, 200, 300, 400, 500]
- learning rate search range: [0.0001, 0.001, 0.01, 0.1]

The best combination is: learning rate=0.1, estimators=400. Now a refined grid search is run to investigate values close to the wide grid search optimal values.

- number estimators search range: [480, 500, 510, 525, 550]
- learning rate search range: [0.1]

The optimal value for estimators is found at 510, the accuracy score is 0.8448. Moreover, the following parameters are tuned in the last step again, as well as two new ones:

- colsamply bytree: [0.7,0.8,0.9]
- gamma: [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
- reg alpha: L1 regularization on weights, makes the model more conservative. Search range: [0, 0.1, 0.2, 0.3, 0.4]
- subsample: the amount of rows to be random samples for each tree, values smaller then 1 prevent overfitting. [0.6, 0.7, 0.8, 0.9]

The final result of the hyper-parameter tuning shows table 3. Configuration 1 achieved the best score of all attempts on the training set and was tried in submission 9, but only 0.79982 on the test set. Configuration 2 performs better on the test set and was used in submission 8, 10 and 11.

Table 3: This table shows the best parameter configurations of XGBoost.

parameter	configuration 1	configuration 2
n estimators	510	400
learning rate	0.1	0.1
max depth	4	4
gamma	0.6	0.6
reg alpha	0.3	0.3
subsample	0.8	0.8
colsample bytree	0.8	0.8
reg alpha	0.3	0.3
accuracy score	0.84478	0.8419
test score	0.79982	0.80155

Early Stopping

Early stopping is a method used to prevent overfitting the model during training time. In order for it to work, we need a hold out set that we use as a validation set. The performance of the model on this validation set is monitored and once there is no improvement within a fixed number of training iterations, the training is stopped. We set the number of iterations without improvement to 10, that means after 10 iterations without an improvement of the loss function for multi-classification the XGBoost training comes to an end.

Fig.9 shows that after around 20 iterations, the multi-class log loss (cross entropy) does not go down further on the validation set. Similarly it can be observed in fig.10 that the classification error does not improve much after 20 iterations either. So it makes sense to apply early stopping in our training process.

Fig. 11 shows the model training with early stopping. Whereas on the training set the performance is around the same with and without early stopping, on the test set in kaggle, the performance dropped to 0.76445 applying this method. The reason is that we had to split the training data into a training and validation set in order for the early stopping to work, but this lead to data or information loss, because the final model is trained only on a subset of data.

Figure 9: The development of the multi-class log loss over the training iterations on the train and hold out test set for XGBoost.

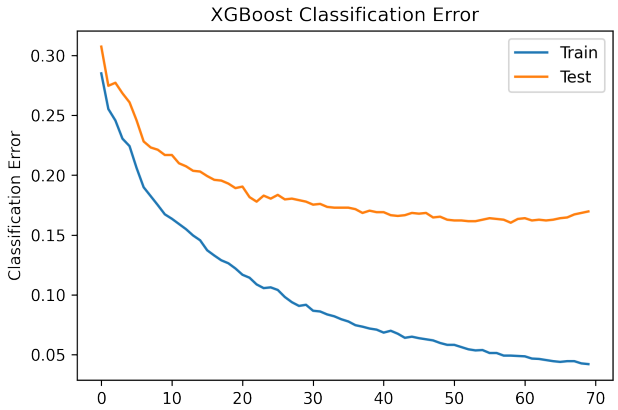

Figure 10: The development of the classification error over the training iterations on the train and hold out test set for XGBoost.

```
# fit model with EARLY STOPPING
model = XGBClassifier(objective="multi:softmax", scale_pos_weight=1,
                    learning_rate=0.5, colsample_bytree = 0.8,
                      subsample = 0.8, n_estimators=70,
                     reg_alpha = 0.3, max_depth=4, gamma=0.6)
eval_set = [(X_test, y_test)]
# stop after 10 rounds with no improvement
model.fit(X_train, y_train, early_stopping_rounds=10,
          eval_metric="mlogloss", eval_set=eval_set, verbose=True)
# make predictions for test data
y_pred = model.predict(X_test)
predictions = [round(value) for value in y_pred]
# evaluate predictions
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: %.4f" %  (accuracy))
```

Figure 11: Fitting the XGBoost algorithm with early stopping.

Other Algorithms

Table 4 shows all the different algorithms that were tried out during the competition.

Table 4: This table shows the cross-validation results on the training data of all the algorithms used during the competition.

algorithm	configuration	train accuracy
XGBoost	default	0.831
LightGBM	default	0.83046
RandomForest	default	0.8193
GradientBoosting	default	0.8099
AdaBoost	default	0.686
OneVsRest(LogisticRegression)	default,multi-class=”ovr”	0.62336
Multi-layer Perceptron	tuned	0.6044
Multi-layer Perceptron	hidden layersizes = [100]*3	0.5287
LogisticRegression	default, multi-class=”multinomial”	0.4864
OneVsRest(linear SVM)	default	0.4555

None of them performed better then the XGBoost algorithm in default configuration. We can see a clear trend towards decision tree algorithms. The default parameters are used in order to find a good starting algorithm to further tune, this turned out to be XGBoost.

Another promising approach is stacking. In stacking, different base-models make predictions on the data in the first step. In the second step, a meta-model makes a final prediction based on the base-models predictions. This classifier is trained on the output of the first step, the predictions. A model stack is created with the LGBM algorithm with optimized hyper-parameters and the

RandomForest algorithm with the default parameters (fig.12, fig.13). XGBoost is used in the second step of the stacking pipeline as a meta-model and the hyper-parameters are tuned using a grid search. The final training accuracy score of the stacking model using cross validation is 0.8282 (fig.14).

```
#STACKING with vecstack
models = [
    lgbm.LGBMClassifier(colsample_bytree= 0.7, learning_rate= 0.05,
                        max_depth=4, n_estimators=350),
    RandomForestClassifier()]
```

```
X_train, X_test, y_train, y_test = train_test_split(X_noK_noM, y)
```

```
S_train, S_test = stacking(models, X_train, y_train, X_test,regression=False,
                           mode='oof_pred_bag', needs_proba=False, save_dir=None,
                           metric=accuracy_score, n_folds=5, stratified=True,
                           shuffle=True,random_state=0,verbose=2)
```

Figure 12: Defining the stacking algorithm.

```
task:         [classification]
n_classes:    [5]
metric:       [accuracy_score]
mode:         [oof_pred_bag]
n_models:     [2]

model  0:     [LGBMClassifier]
    fold  0:  [0.81466113]
    fold  1:  [0.80912863]
    fold  2:  [0.85200553]
    fold  3:  [0.82019364]
    fold  4:  [0.82132964]
    ----
    MEAN:     [0.82346371] + [0.01491887]
    FULL:     [0.82346431]

model  1:     [RandomForestClassifier]
    fold  0:  [0.80774550]
    fold  1:  [0.78838174]
    fold  2:  [0.80774550]
    fold  3:  [0.81189488]
    fold  4:  [0.79501385]
    ----
    MEAN:     [0.80215630] + [0.00892268]
    FULL:     [0.80215827]
```

Figure 13: Running base models.

```
model = XGBClassifier(random_state=0, n_jobs=-1, learning_rate=0.01,
                      n_estimators=40, max_depth=3, silent=True, verbose=0)

model = model.fit(S_train, y_train)
y_pred = model.predict(S_test)
print('Final prediction score: [%.5f]' % accuracy_score(y_test, y_pred))
```

Figure 14: cross-validation with meta-model.

Submission No.	Date	Leader board position	Local train set score (cross validation)	Prediction score on test set	Data Preprocessing	Algorithm	Parameters of Algorithm
1	29/12/2020, 14:00	Last place	0.4576	0.2010	1. Drop column „id" and „Descuento" 2. Remove chars from numerical attributes of dtype string (Motor CC, Consumo, Potencia). E.g. 74 bhp -> 74. 3. Impute missing values of numerical attributes with mean value 4. Impute missing values of categorical attribute with most frequent value 5. Binary Encoding of categorical attributes 6. Normalizing numerical attributes	Random Forest	max_features=0.5,max_depth=10,min_samples_leaf=2, random_state=42
2	29/12/2020, 16:00	Last place	0.8594	0.20362	Same as in submission 1, except for **One Hot Encoding** instead of Binary Encoding in step 4	Random Forest	max_features=0.5,max_depth=10,min_samples_leaf=2, random_state=42
3	30/12/2020, 10:00	Last place	0.8470	0.23123	Same as in submission 2 (One Hot Encoding)	MLP Classifier (Multi-layer Perceptron)	hidden_layer_sizes = [100]*3
4	30/12/2020, 14:00	Last place	0.40133	0.22001	Same as in submission 1, except for Label Encoding of **all** the variables in step 4	XGB Classifier (Extreme Gradient Boosting from xgboost library)	default parameters
5	30/12/2020, 21:45	23	0.8319	0.76790	1. Drop columns „id" & „Descuento" 2. Impute missing values of **all** attributes using the most frequent value 3. Use Label Encoding on **all** attributes	XGB Classifier	default parameters

Figure 15: Kaggle submissions 1-5.

Submission No.	Date	Leader board position	Local train set score (cross validation)	Prediction score on test set	Data Preprocessing	Algorithm	Parameters of Algorithm
6	31/12/2020, 11:00	23	0.8373 on full data 0.8360 on hold out validation set with early stopping	0.76445	Same as in submission 5	XGB Classifier Create hold out set from training set, to use early stopping with the hold out set as the validation set during model fitting.	objective="multi:softmax", min_child_weight=1, learning_rate=0.5, colsample_bytree = 0.7, n_estimators=70, max_depth=4
7	31/12/2020, 12:52	11	0.8373	0.79378	Same as in submission 5	XGB Classifier without early stopping: fit on full train set	Same as in submission 6
8	31/12/2020, 14:38	8	0.8424	0.80155	Same as in submission 5 + Feature Selection: drop attributes „Kilometros" and „Mano"	XGB Classifier	Same as in submission 6
9	01/01/2021, 22:00	11	0.8448	0.79982	Same as in submission 8	XGB Classifier	objective="multi:softmax", scale_pos_weight=1, **learning_rate=0.1**, colsample_bytree = 0.8, subsample = 0.8, **n_estimators=510**, reg_alpha = 0.3, max_depth=4, gamma=0.6

Figure 16: Kaggle submissions 6-9.

Submission No.	Date	Leaderboard position	Local train set score (cross validation)	Prediction score on test set	Data Preprocessing	Algorithm	Parameters of Algorithm
10	01/01/2021, 00:30	10	0.8425	0.80241	Same as in submission 8 + Class Balancing: Oversampling **the minority class** using SMOTE algorithm (to 400 samples instead of 269)	XGB Classifier	XGB: same as in sub. 9, n_estimators = 400 SMOTE: k_neighbors =2 , sampling_strategy = samples -> samples = {**1:400**, 2: ...}
11	01/01/2021, 00:43	8	0.8392	0.80845	Same as in submission 8 + Class Balancing: Oversampling **the minority class** using **BorderlineSMOTE** algorithm (to 400 samples instead of 269)	XGB Classifier	XGB: same as in sub. 9, n_estimators = 400 BorderlineSMOTE: k_neighbors =3, sampling_strategy = samples -> samples = {**1:400**, 2: ...}

Figure 17: Kaggle submissions 10 and 11.

References

[1] https://towardsdatascience.com/chi-square-test-for-feature-selection-in-machine-learning-206b1f0b8223 *Chi-Square Test for Feature Selection in Machine learning*, last access 04-01-2021.

[2] https://machinelearningmastery.com/feature-selection-with-categorical-data/ *How to Perform Feature Selection with Categorical Data*, last access 04-01-2021.

[3] https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/ *SMOTE for Imbalanced Classification with Python*, last access 04-01-2021.

[4] https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/ *SMOTE for Imbalanced Classification with Python*, last access 04-01-2021.

[5] https://miro.medium.com/max/700/1*QJZ6W-Pck_W7RlIDwUIN9Q.jpeg *Evolution of XGBoost Algorithm from Decision Trees*, last access 04-01-2021.

[6] https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/ *Complete Guide to Parameter Tuning in XGBoost with codes in Python*, last access 04-01-2021.

[7] https://machinelearningmastery.com/avoid-overfitting-by-early-stopping-with-xgboost-in-python/ *Avoid Overfitting By Early Stopping With XGBoost In Python*, last access 04-01-2021.

[8] https://machinelearningmastery.com/avoid-overfitting-by-early-stopping-with-xgboost-in-python/ *Avoid Overfitting By Early Stopping With XGBoost In Python*, last access 04-01-2021.

[9] https://machinelearningmastery.com/tune-learning-rate-for-gradient-boosting-with-xgboost-in-python/ *Tune Learning Rate for Gradient Boosting with XGBoost in Python*, last access 04-01-2021.

[10] https://machinelearningmastery.com/avoid-overfitting-by-early-stopping-with-xgboost-in-python/ *Avoid Overfitting By Early Stopping With XGBoost In Python*, last access 04-01-2021.