Classification and regression trees

Paulo S. A. Sousa

2023-05-10

Abstract

Classification and regression trees are introduced. Their tendency to overfitting and possible solutions is discussed.

The problem of imbalanced datasets and possible solutions are presented.

Python code for working with trees is developed.

1 Introduction

Trees can be used for both classification and regression, being a powerful prediction model.

2 The algorithm for classification

The algorithm underlying the creation of a classification tree is based on the idea of recursively splitting the data into more homogenous subsets. For instance, if our problem is a one of binary classification, with classes A and B, the algorithm will try to split the data into subsets that are the purest possible, i.e., with only data points of class A or with only data points of class B.

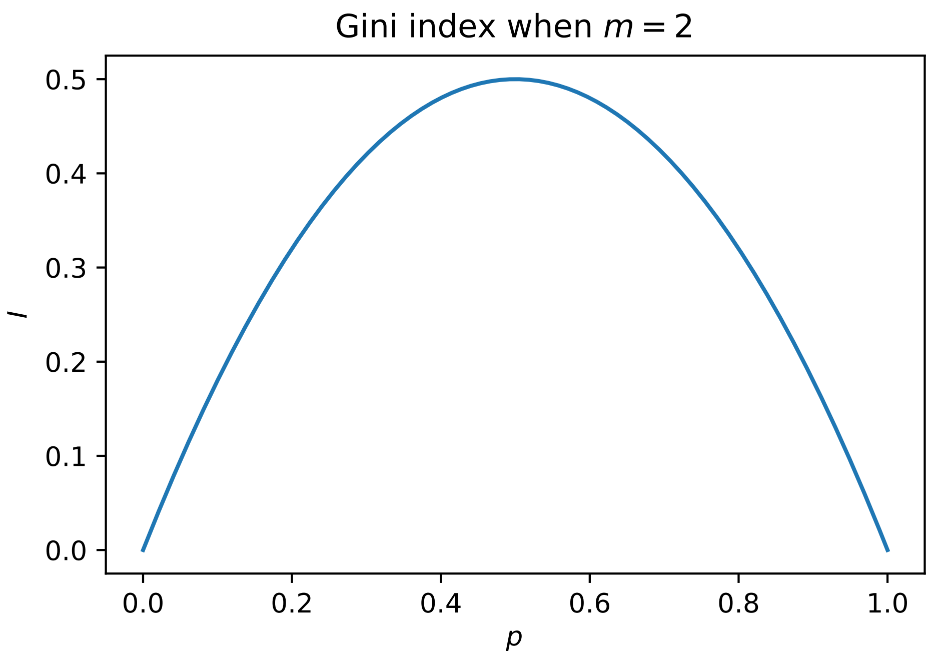
There are several measures of purity. However, the most popular ones, in the context of trees, are the Gini index and the entropy. In our course, we will focus only on the Gini index.

Considering the outcome variable can be any class k, with $k = 1, \cdots, m$, this index, for a subset S, is given by the formula

$$I(S) = 1 - \sum_{k=1}^{m} p_k^2,$$

where p_k is the proportion of the data points of class k in the subset S.

It is easy to plot the graph of the Gini index when $m = 2$ (only two classes):



As we can see from the plot, the Gini index is maximum when $p = 0.5$, i.e., when there is the same proportion of class A and B, and it is minimum when there is only one class. The conclusion that the Gini index is zero when there is only a single class in the set is also valid when $m > 2$ (that is when the outcome variable has more than two classes).

Let us now consider the following tiny dataset, to illustrate how the algorithm works:

	X1	X2	Y
0	1	2	0
1	2	3	1
2	3	4	1
3	4	5	0

We start by calculating the Gini index for each possible split in the predictor X1. As a rule, for each split, we consider the midpoint, always.

- Split at X1 = 1.5:

  We will have two subsets: the first one with 1 data point of class 0 and another one with 1 data point of class 0 and 2 data points of class 1. Then, the overall Gini index will be equal to (each subset will be weighted by its size in the calculation of the Gini index):

$$\frac{1}{4} \times 0 + \frac{3}{4} \times \left(1 - \left(\frac{2}{3}\right)^2 - \left(\frac{1}{3}\right)^2\right) \approx 0.33.$$

- Split at X1 = 2.5:

  We will have two subsets: the first one with 2 data points, one of class 0 and another one of class 1, and another one with 2 data points, one of class 0 and another one of class 1. Then, the overall Gini index will be equal to (each subset will be weighted by its size in the calculation of the Gini index):

$$\frac{2}{4} \times \left(1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2\right) + \frac{2}{4} \times \left(1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2\right) = 0.5.$$

- Split at X1 = 3.5:

  We will have two subsets: the first one with 1 data point of class 0 and another one with 1 data point of class 0 and 2 data points of class 1. Then, the overall Gini index will be equal to (each subset will be weighted by its size in the calculation of the Gini index):

$$\frac{1}{4} \times 0 + \frac{3}{4} \times \left(1 - \left(\frac{2}{3}\right)^2 - \left(\frac{1}{3}\right)^2\right) \approx 0.33.$$

We can therefore select, as the first split of the tree, the midpoint X1 = 1.5 – it corresponds to the lowest overall Gini index (*ex aequo* with split at X1 = 3.5, which we could have instead selected).

When X1 < 1.5, the resulting subset has only an element of the same class, and, consequently, there is no need to proceed further with splits in that direction – we cannot purify further the subset.

The set associated to X1 > 1.5 is the following:

	X1	X2	Y
0	2	3	1
1	3	4	1
2	4	5	0

We start by calculating the Gini index for each possible split in the predictor X1:

- Split at X1 = 2.5:

  We will have two subsets: the first one with 1 data point of class 1, and another one with 1 data point of class 0 and 1 data points of class 1. Then, the overall Gini index will be equal to (each subset will be weighted by its size in the calculation of the Gini index):

$$\frac{1}{3} \times 0 + \frac{2}{3} \times \left(1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2\right) \approx 0.33.$$

- Split at X1 = 3.5:

  We will have two subsets: the first one with 2 data points of class 1, and another one with 1 data point of class 0. Then, the overall Gini index will be equal to (each subset will be weighted by its size in the calculation of the Gini index):

$$\frac{2}{3} \times 0 + \frac{1}{3} \times 0 = 0.$$

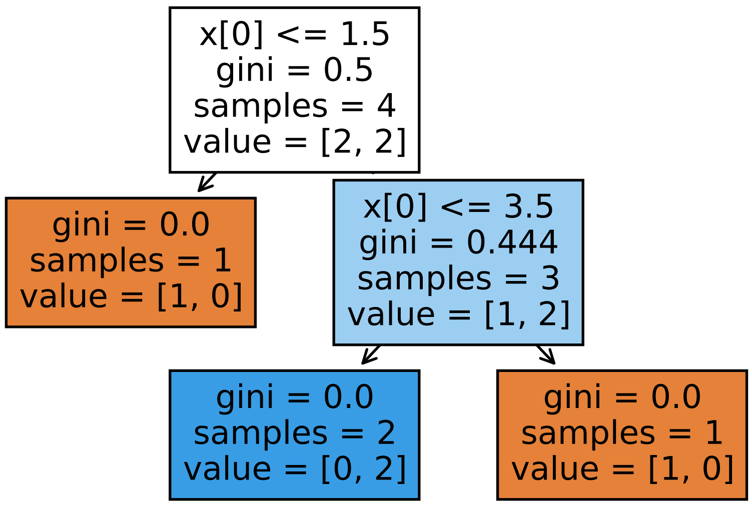
Consequently, the second split of the tree will be at the midpoint $X1 = 3.5$ – it corresponds to the lowest overall Gini index.

The splits in predictor X2 lead to similar Gini indexes. Therefore, we do not need to examine them. Otherwise, we should also study the splits on X2 (actually, on all predictors, if there were more than two predictors).

2.1 Implementation in Python

We can obtain the same classification tree by using Python:

```python
from matplotlib import pyplot as plt
from sklearn.tree import DecisionTreeClassifier, plot_tree

X, y = df.drop('Y', axis=1), df['Y']

# random_state controls randomness and therefore we
# get always the same tree
tree = DecisionTreeClassifier(random_state=45)

tree.fit(X, y)

plot_tree(tree, filled=True)

plt.show()
```



The process of obtaining the prediction for a new case follows the following steps:

1. Verifies if the new case satisfies the logical condition in the first node of the tree (the tree root).
2. If the new case satisfies the said logical condition, let us go to the left and come to the next node of the tree.
3. If the new case does not satisfy the said logical condition, let us go right and reach the next node of the tree.
4. In the reached node, the respective logical condition is verified against the new case and the process is repeated again and again until a terminal node (tree leaf) is reached.
5. In the reached tree leaf, the majority class will be taken by the tree prediction for the new case.

As an example, we are going to use the built tree to classify new case (X1=3, X2=4). Since X1 does not satisfy the logical condition of the root (x[0] <= 1.5; x[0] corresponds to the first column of `df`, x[1] to the second column of `df`, and so on), we go to the right. The logical condition in the reached node is satisfied by the new case (X1 <= 3.5). We take the left and reach a leaf of the tree. The majority class in the reached leaf is 1 and therefore the prediction for the new case is 1.

We can obtain the same prediction by using Python:

```
X_new = pd.DataFrame({
  'X1': [3],
  'X2': [4]}
  )

tree.predict(X_new)
```

```
array([1])
```

3 The algorithm for regression

The goal of the regression tree is also to partition the feature space into regions that are as homogeneous as possible in terms of the target variable. Therefore, the algorithm works similarly to the one used for classification. At each node of the tree, we split the data based on the value of one of the features, and we assign a predicted value to each region defined by the split. The predicted value for a region is simply the mean value of the target variable for the samples in that region.

We can use the MSE as a measure of the quality of the split, and we want to minimize it at each step of the tree construction. Here's how we can calculate the MSE for a split:

1. Calculate the mean target value for each subset resulting from the split.
2. Calculate the MSE for each subset as the average squared difference between the target values and their corresponding mean value.
3. Calculate the overall MSE for the split as the weighted sum of the MSEs for each subset, where the weights are the proportions of samples in each subset relative to the total number of samples.

	X1	X2	Y
0	2.0	1.0	10
1	3.0	1.5	10
2	4.0	2.0	12
3	5.0	3.0	18

We start by calculating the MSE for each possible split in the predictor X1:

- Split at X1 = 2.5:

  We will have two subsets: the first one with 1 data point with Y=10 and another one with 3 data points, with Y=10, Y=12 and Y= 18, respectively, which yields $\overline{Y} \approx 13.33$. The MSE is given by:

$$\frac{1}{4} \times 0 + \frac{3}{4} \times \frac{1}{3} \left(\left(10 - \overline{Y}\right)^2 + \left(12 - \overline{Y}\right)^2 + \left(18 - \overline{Y}\right)^2 \right) \approx 8.66.$$

- Split at X1 = 3.5:

  We will have two subsets: the first one with 2 data points with Y=10 and another one with 2 data points, with Y=12 and Y= 18, respectively, which yields $\overline{Y} = 15$. The MSE is given by:

$$\frac{2}{4} \times 0 + \frac{2}{4} \times \frac{1}{2} \left(\left(12 - \overline{Y}\right)^2 + \left(18 - \overline{Y}\right)^2 \right) = 4.5.$$

- Split at X1 = 4.5:

  We will have two subsets: the first one with 1 data point with Y=18 and another one with 3 data points, with Y=10, Y=10 and Y= 12, respectively, which yields $\overline{Y} \approx 10.66$. The MSE is given by:

$$\frac{1}{4} \times 0 + \frac{3}{4} \times \frac{1}{3}\left(\left(10-\overline{Y}\right)^2 + \left(10-\overline{Y}\right)^2 + \left(12-\overline{Y}\right)^2\right) \approx 0.33.$$

We can therefore select, as the first split of the tree, the midpoint X1 = 4.5 – it corresponds to the lowest overall mean squared error.

When X1 > 4.5, the resulting subset has only an element, and, consequently, there is no need to proceed further with splits in that direction – we cannot improve further the subset.

The set associated to X1 <= 4.5 is the following:

	X1	X2	Y
0	2.0	1.0	10
1	3.0	1.5	10
2	4.0	2.0	12

We can now start by calculating the MSE for each possible split in the predictor X1:

- Split at X1 = 2.5:

  We will have two subsets: the first one with 1 data point with Y=10 and another one with 2 data points, with Y=10 and Y=12, respectively, which yields $\overline{Y} = 11$. The MSE is given by:

$$\frac{1}{3} \times 0 + \frac{2}{3} \times \frac{1}{2}\left(\left(10-\overline{Y}\right)^2 + \left(12-\overline{Y}\right)^2\right) \approx 0.66.$$

- Split at X1 = 3.5:

  We will have two subsets: the first one with 2 data points, both with Y=10 and one with 1 data point with Y=12. The overall MSE is 0.

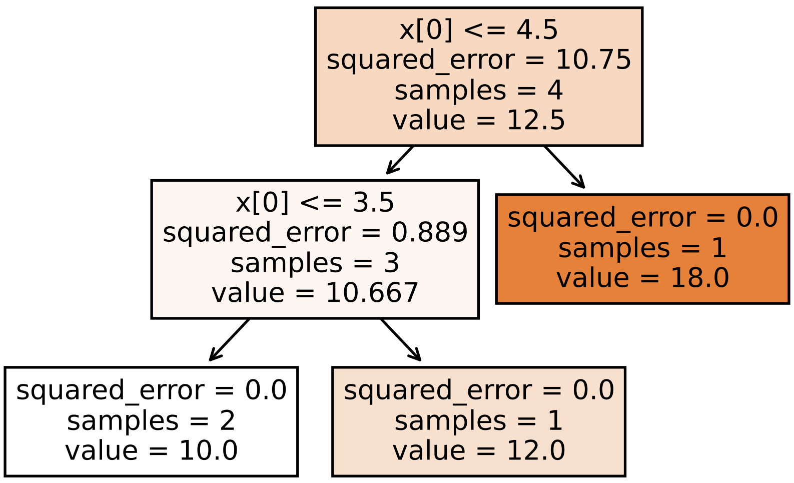
Consequently, the second split of the tree will be at the midpoint X1 = 3.5 – it corresponds to the lowest overall MSE.

The splits in predictor X2 lead to similar MSE. Therefore, we do not need to examine them. Otherwise, we should also study the splits on X2 (actually, on all predictors, if there were more than two predictors).

3.1 Implementation in Python

We can obtain the same regression tree by using Python:

```python
from matplotlib import pyplot as plt
from sklearn.tree import DecisionTreeRegressor, plot_tree

X = df.drop('Y', axis=1)
y = df['Y']

# random_state controls randomness and therefore we
# get always the same tree
tree = DecisionTreeRegressor(random_state=45)
tree.fit(X, y)

plot_tree(tree, filled=True)
plt.show()
```



The process of obtaining the prediction for a new case follows similar steps as with a classification tree, except that the prediction taken from the leaves is the `value` of the leaf (mean of the outcome variable associated with the leaf).

As an example, we are going to use the built tree to predict the outcome variable for the new case (X1=4, X2=3). Since X1 satisfies the logical condition of the root (x[0] <= 4.5), we go to the left. The logical condition in the reached node is not satisfied by the new case (X1 <= 3.5). We take the right and reach a leaf of the tree. The `value` in the reached leaf is 12 and therefore the prediction for the new case is 12.

We can obtain the same prediction by using Python:

```python
X_new = pd.DataFrame({
  'X1': [4],
  'X2': [3]}
  )
```

```
tree.predict(X_new)
```

```
array([12.])
```

4 Overfitting

As it turns out from the study of the algorithm of tree creation, the trees can grow up to a stage of full purity. A so pure tree will produce perfect predictions for the training set – and this is very problematic! The reason why this is so problematic is that the data usually contain noise, statistical noise, and such a perfect tree will fit the noise and not only the meaningful information of the data – that is *overfitting*.

Since the noise of the test set is not the same as contained in the training set, the result is that the prediction for the test set (and then for real new data) will tend to be very poor – and our tree useless.

4.1 An example of overfitting

The problem of overfitting emerges when we try to solve the problem of the bank marketing campaign with classification trees. Let us remember the code we used:

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import recall_score
from sklearn.metrics import accuracy_score, confusion_matrix

# adjust the path to the file as appropriate in your case
df =
 ↪  pd.read_csv('/home/x/programas/r/pythonL/datasets/bank_mark_campaign.csv',
 ↪  sep=';')

df = df.replace('unknown', np.nan)

col_nan = df.columns[df.isna().any(axis=0)].to_list()
col_num = df.describe().columns.to_list()
df.columns.difference(col_nan + col_num)
col_cat = df.columns.difference(col_nan + col_num + ['y']).to_list()
```

```
na_treat = Pipeline([
    ('imp', SimpleImputer(strategy='most_frequent')),
    ('oneh', OneHotEncoder(drop='first'))])

preprocessor = ColumnTransformer([
    ('na_tr', na_treat, col_nan),
    ('cat_tr', OneHotEncoder(drop='first'), col_cat),
    ('scale_tr', StandardScaler(), col_num)],
    remainder='passthrough')

pipe = Pipeline([
    ('pre', preprocessor),
    ('tree', DecisionTreeClassifier())])

X = df.drop('y', axis=1)
y = df['y']

X_train, X_test, y_train, y_test = train_test_split(X, y,
 ↪  test_size=0.2, random_state=45)

pipe.fit(X_train, y_train)

y_pred = pipe.predict(X_train)

acur = accuracy_score(y_train, y_pred)
print(f'Accuracy= {acur}')
cm = confusion_matrix(y_train, y_pred, labels=['yes', 'no'])
print(cm)
recall = recall_score(y_train, y_pred, pos_label='yes')
print(f'Recall= {recall}')
```

```
Accuracy= 1.0
[[ 3700     0]
 [    0 29250]]
Recall= 1.0
```

The result, for the training set, is 100% of accuracy and recall score!

Remember that recall score, also known as sensitivity or true positive rate, is a performance metric used to evaluate the ability of a classification model to identify relevant instances correctly. It is especially useful when dealing with imbalanced datasets, where the proportion of one class is significantly lower than the other. Recall score is calculated using the following formula:

Recall = True Positives / (True Positives + False Negatives)

In this formula, “True Positives” are the instances that the model correctly identified as positive, and “False Negatives” are the instances that the model incorrectly identified as negative when they were actually positive.

Let us use our bank marketing campaign dataset as an example. In this case, the positive class is `yes` (clients who subscribed), and the negative class is `no` (clients who didn’t subscribe).

Let us check now the accuracy and the recall score of the model for the test set:

```
y_pred = pipe.predict(X_test)

acur = accuracy_score(y_test, y_pred)
print(f'Accuracy= {acur}')
cm = confusion_matrix(y_test, y_pred, labels=['yes', 'no'])
print(cm)
recall = recall_score(y_test, y_pred, pos_label='yes')
print(f'Recall= {recall}')
```

```
Accuracy= 0.8826171400825443
[[ 480  460]
 [ 507 6791]]
Recall= 0.5106382978723404
```

It is manifest that both the accuracy and recall scores dropped considerably for the test set, especially the recall score, which fall to one-half!

When we face a so sharp discrepancy between the training set and test set regarding the performance of a model, that is a signal that the model is overfitting.

To prevent overfitting, we need to impose constraints on the growth of the tree. There are several hyperparameters that we can manipulate, which are later enumerated in a proper section. Notwithstanding, there is a hyperparameter that we can use to control the complexity of the tree, by pruning the tree. The complexity of a tree is determined by the number of terminal nodes (leaves) and the depth of the tree, and, therefore, complex trees are prone to overfitting.

The `ccp_alpha` hyperparameter represents the complexity cost (penalty) added to the model’s error for each additional split. It is a non-negative value, where higher values of `ccp_alpha` result in more aggressive pruning (simpler trees), and lower values result in less pruning (more complex trees).

If `ccp_alpha` is set to 0, no pruning will be performed, and the tree will grow to its maximum depth, potentially leading to overfitting. If `ccp_alpha` is set to a positive value, the tree will be pruned, and the optimal tree size will be determined based on the trade-off between the model’s complexity and its performance on the training data. The optimal value of `ccp_alpha` varies depending on the specific problem and dataset. We can find the best value by using techniques like cross-validation or grid search to determine the best `ccp_alpha` value.

To search for a good value for `ccp_alpha`, we can use grid search:

```
from sklearn.model_selection import GridSearchCV

hyper = {
    'ccp_alpha': [0.001, 0.01, 0.3, 0.5]
}

pipe = Pipeline([
    ('pre', preprocessor),
    ('grid', GridSearchCV(DecisionTreeClassifier(), hyper, cv=5))])

pipe.fit(X_train, y_train)

y_pred = pipe.predict(X_train)

acur = accuracy_score(y_train, y_pred)
print(f'Accuracy= {acur}')
cm = confusion_matrix(y_train, y_pred, labels=['yes', 'no'])
print(cm)
recall = recall_score(y_train, y_pred, pos_label='yes')
print(f'Recall= {recall}')
```

```
Accuracy= 0.9136267071320182
[[ 1928  1772]
 [ 1074 28176]]
Recall= 0.5210810810810811
```

Now, the results are not so stellar! Let us check whether they are similar to the one obtained by using the test set:

```
y_pred = pipe.predict(X_test)
acur = accuracy_score(y_test, y_pred)
print(f'Accuracy= {acur}')
cm = confusion_matrix(y_test, y_pred, labels=['yes', 'no'])
print(cm)
recall = recall_score(y_test, y_pred, pos_label='yes')
print(f'Recall= {recall}')
```

```
Accuracy= 0.9129643117261471
[[ 493  447]
 [ 270 7028]]
Recall= 0.524468085106383
```

Indeed, now the results for both sets are similar and, therefore, there is no suspicion of overfitting.

5 Imbalanced datasets

Imbalanced datasets refer to datasets where the distribution of instances across different classes is unequal. In other words, one or more classes have significantly fewer instances compared to others. This imbalance can lead to biased models that favor the majority class, as they are more inclined to optimize their predictions for the more frequently occurring class. This can result in poor performance on the minority class, especially in applications such as fraud detection, rare disease diagnosis, or spam filtering, where the minority class is often of greater interest.

The dataset that we are using is imbalanced. We can confirm that by:

```
# the parameter normalize=True is to convert the counts to
 ↪  percentages

y.value_counts(normalize=True)
```

```
y
no     0.887346
yes    0.112654
Name: proportion, dtype: float64
```

Only 11% of the cases in the dataset correspond to the `yes` class. The `yes` class is the class that interests us most, as we want to predict the customers more likely to accept our marketing campaign (to reduce wasting time with customers that will likely not accept the campaign).

The `class_weight` parameter in the decision tree classifier is a way to handle imbalanced datasets by assigning different weights to different classes. It allows you to adjust the learning process so that the model pays more attention to the minority class during training. This helps mitigate the bias towards the majority class, which may arise when working with imbalanced datasets.

When using the `class_weight` parameter in a decision tree classifier, the algorithm adjusts the splitting criterion (Gini impurity) to consider the class weights. During the tree construction, it seeks to minimize the weighted impurity of each node, making the model more sensitive to the minority class. The option `class_weight='balanced'` automatically adjusts the class weights based on the number of samples in each class.

Let us try that:

```
from sklearn.model_selection import GridSearchCV

hyper = {
    'ccp_alpha': [0.001, 0.01, 0.3, 0.5]
}

pipe = Pipeline([
    ('pre', preprocessor),
    ('grid',
↪   GridSearchCV(DecisionTreeClassifier(class_weight='balanced'),
↪   hyper, cv=5))])

pipe.fit(X_train, y_train)

y_pred = pipe.predict(X_train)

acur = accuracy_score(y_train, y_pred)
print(f'Accuracy= {acur}')
cm = confusion_matrix(y_train, y_pred, labels=['yes', 'no'])
print(cm)
recall = recall_score(y_train, y_pred, pos_label='yes')
print(f'Recall= {recall}')
```

```
Accuracy= 0.8877086494688923
[[    0  3700]
 [    0 29250]]
Recall= 0.0
```

Surprisingly, the situation not only did not improve – it has got much worse, as the recall score is zero! Consequently, we have to look for new ways to improve the recall score. Using the receiver operating characteristic curve is a possibility.

6 Receiver Operating Characteristic

In a binary classification problem, a model generates a probability score for each instance to predict its class. This probability score is then compared to a classification threshold to assign the instance to either the positive or negative class. The classification threshold is a value between 0 and 1 that determines the boundary between the two classes. By default, the threshold is typically set to 0.5, which means that instances with a probability score above 0.5 are classified as the positive class, and those with a score below 0.5 are classified as the negative class.

A ROC (Receiver Operating Characteristic) curve is a graphical representation of this relationship, plotting the true positive rate (TPR) against the false positive rate (FPR) at various classification thresholds.

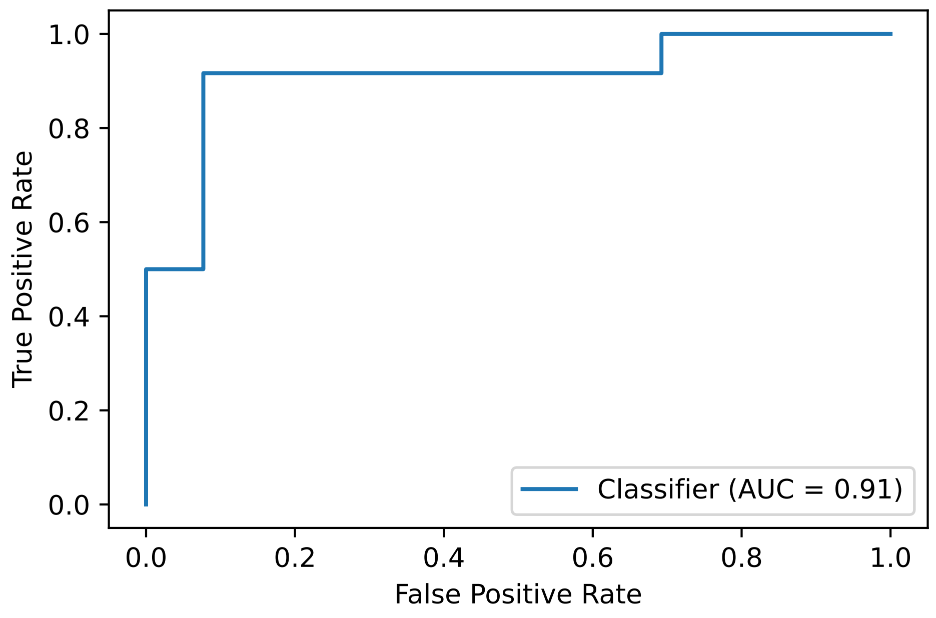
The True Positive Rate (TPR) is the proportion of actual positive cases that are correctly identified by the model:

TPR = True Positives / (True Positives + False Negatives).

The False Positive Rate (FPR) is the proportion of actual negative cases that are incorrectly identified as positive by the model:

FPR = False Positives / (False Positives + True Negatives).

An example of a ROC curve is depicted below:



The area under the ROC curve (AUC-ROC) is a popular metric used to evaluate the overall performance of a binary classifier, with higher AUC-ROC values indicating better classification performance. An ideal binary classifier would correspond to a unitary AUC-ROC, as, for each value of the False Positive Rate, the True Positive Rate would be always 1 (the maximum possible).

Fortunately, we can change the `GridSearchCV`'s optimizing goal: Instead of using the accuracy score (the default option), we can use the `roc_auc` score, in the expectation that, by thus proceeding, we are optimizing the recall score, by changing the level of the complexity of the tree (by trying different values for the parameter `ccp_alpha`).

We try that now:

```
from sklearn.model_selection import GridSearchCV

hyper = {
    'ccp_alpha': [0.001, 0.01, 0.3, 0.5]
}

pipe = Pipeline([
    ('pre', preprocessor),
    ('grid',
 ↪  GridSearchCV(DecisionTreeClassifier(class_weight='balanced'),
 ↪  hyper, cv=5, scoring='roc_auc'))])

pipe.fit(X_train, y_train)
```

```
y_pred = pipe.predict(X_train)

acur = accuracy_score(y_train, y_pred)
print(f'Accuracy= {acur}')
cm = confusion_matrix(y_train, y_pred, labels=['yes', 'no'])
print(cm)
recall = recall_score(y_train, y_pred, pos_label='yes')
print(f'Recall= {recall}')
```

```
Accuracy= 0.8336874051593324
[[ 3519   181]
 [ 5299 23951]]
Recall= 0.951081081081081
```

The results appear to be fantastic: The recall score is about 95% for the training set! However, we still have to check whether the recall score is similar when using the test set.

```
y_pred = pipe.predict(X_test)
acur = accuracy_score(y_test, y_pred)
print(f'Accuracy= {acur}')
cm = confusion_matrix(y_test, y_pred, labels=['yes', 'no'])
print(cm)
recall = recall_score(y_test, y_pred, pos_label='yes')
print(f'Recall= {recall}')
```

```
Accuracy= 0.8319980577810148
[[ 873   67]
 [1317 5981]]
Recall= 0.9287234042553192
```

The results are also great! And now we have a good prediction model that will not miss many potential `yes` cases, saving a lot of time and effort for the bank, that we will contact only a small fraction of its customers (and not all of them), where almost all customers, that are most likely to accept the marketing campaign, are.

7 Hyperparameters

Some examples of hyperparameters, that can be adjusted to control the growth and structure of a decision tree model, are:

- `max_depth`: This controls the maximum depth of the tree. The default value is `None`, which means the tree is grown until all leaves are pure or until all leaves contain less than `min_samples_split` samples.

- `min_samples_split:` This specifies the minimum number of samples required to split an internal node. The default value is 2.
- `min_samples_leaf:` This specifies the minimum number of samples required to be at a leaf node. The default value is 1.
- `max_leaf_nodes:` This sets the maximum number of leaf nodes in the tree. The default value is `None`, which means the number of leaf nodes is not constrained.
- `criterion:` This specifies the impurity criterion used to measure the quality of a split. The default value is `gini`, but `entropy` can also be used.

These hyperparameters can be set when initializing the `DecisionTreeClassifier` or `DecisionTreeRegressor` object in `scikit-learn`. They can also be optimized using techniques such as cross-validation and grid search to find the optimal values for a given problem.