Introduction to Tree-based methods (with external dependencies)

Written by:

- Manuel Szewc (School of Physics, University of Cincinnati)
- Philip Ilten (School of Physics, University of Cincinnati)

This notebook wants to implement decision trees, random forests and gradient boosting. A lot of it is based on [Aurelien Geron's lectures](#).

```
import os
import sys

# To generate data and handle arrays
import numpy as np

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

%matplotlib inline
mpl.rc("axes", labelsize=14)
mpl.rc("xtick", labelsize=12)
mpl.rc("ytick", labelsize=12)


import pandas as pd
import cv2  # pip install opencv-python

%matplotlib inline
from scipy.stats import norm, multivariate_normal

# Useful classes for data manipulation
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

# Useful classes for model evaluation and selection
from sklearn.model_selection import train_test_split, cross_val_predict, GridSearchCV
from sklearn.metrics import (
    accuracy_score,
    recall_score,
    confusion_matrix,
    mean_squared_error,
)
```

```
# a baseline classifier
from sklearn.linear_model import LinearRegression, RidgeCV

# The necessary models
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
```

Theory

Decision Tree, most usually based on the **Classification and Regression Tree** (CART) framework, work by **recursively partitioning** the input space through a series of **binary decisions** in order to predict a target (either for classification or regression). Once an appropriate partitioning of the feature space is achieved, a new prediction is computed by following the set of binary splittings.

Under the CART framework, at a given decision step m, we split a **node** m containing N_m instances into two by looking at all available features $\vec{x} \in \mathbb{R}^D$. A decision tree finds a feature i and the best cut θ_m in **one** of the features such that when the data is divided in two new nodes according to $x_i \leq \theta$ the weighted sum of a given metric H evaluated over the two candidate nodes is optimized:

$$G(\theta_m, i) = \frac{N_{m,x_i \leq \theta_m}}{N_m} H(\text{instances with } x_i \leq \theta) + \frac{N_{m,x_i > \theta_m}}{N_m} H(\text{instances with } x_i > \theta_m)$$

That is,

$$\theta_m^*, i^* = \arg\min_{\theta_m, i} \sum_{n=1}^{N_m} G(\theta_m, i)$$

The resulting two nodes are called **children**. The initial node is called a **root** and the nodes which have no children, and are thus final, are called **leaves**.

For K class classification problems, the usual metric H is either the `Gini` or the `entropy` defined as

$$\text{Gini} = \sum \sum_{k=1}^{K} p_{m,k}(1 - p_{m,k})$$

$$\text{Entropy} = -\sum_{k=1}^{K} p_{m,k} \ln p_{m,k}$$

where p_k are the fraction of instances belonging to class k in the node.

For regression problems, it's usually the mean squared error defined as

$$\text{MSE} = \frac{1}{N_m} \sum_{n=1}^{N_m} (y_m - \bar{y}_m)^2$$

where $\bar{y}_m$ is the average target value in the node $\bar{y}_m = \frac{1}{N_m} \sum^{N_m} y_i$

where $\bar{y}_m$ is the average target value in the node $\bar{y}_m = \frac{1}{N_m} \sum_{n=1} y_m$.

Decision Trees are **greedy** algorithms, in that all binary partitions are decided based on how well they perform, without regard to **global strategies**.

From `sklearn`:

Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualized.
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Some tree and algorithm combinations support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. However, the scikit-learn implementation does not support categorical variables for now. Other techniques are usually specialized in analyzing datasets that have only one type of variable. See algorithms for more information.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximations as seen in the above figure. Therefore, they are not good at extrapolation.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree

learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.

- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

Classification with Decision Trees

We'll use the `sklearn` implementation of decision trees.

Let's use a dataset as an example

```
!wget -q -N https://gitlab.com/mcgen-ct/tutorials/-/raw/main/.full/ml/datasets/season-1
```

```
# https://datahub.io/sports-data/english-premier-league and https://www.football-data.c
df = pd.read_csv("season-1112.csv")
```

This file has all matches of the 2011-2012 English Premier League season. For each match, we have local and away goals both at half-time and at the end of the match. We also have the number of shots, shots on goal, fouls, yellow cards, red cards and betting odds from some known sites.

```
df.head()
```

	Div	Date	HomeTeam	AwayTeam	FTHG	FTAG	FTR	HTHG	HTAG	HTR	...	BbMx>2.5
0	E0	13/08/11	Blackburn	Wolves	1	2	A	1	1	D	...	2.06
1	E0	13/08/11	Fulham	Aston Villa	0	0	D	0	0	D	...	2.21
2	E0	13/08/11	Liverpool	Sunderland	1	1	D	1	0	H	...	1.92
3	E0	13/08/11	Newcastle	Arsenal	0	0	D	0	0	D	...	1.89
4	E0	13/08/11	QPR	Bolton	0	4	A	0	1	A	...	2.27

5 rows × 71 columns

```
len(df)
```

```
380
```

```
column_names = df.columns
```

```
column_names
```

```
Index(['Div', 'Date', 'HomeTeam', 'AwayTeam', 'FTHG', 'FTAG', 'FTR', 'HTHG',
       'HTAG', 'HTR', 'Referee', 'HS', 'AS', 'HST', 'AST', 'HF', 'AF', 'HC',
       'AC', 'HY', 'AY', 'HR', 'AR', 'B365H', 'B365D', 'B365A', 'BWH', 'BWD',
       'BWA', 'GBH', 'GBD', 'GBA', 'IWH', 'IWD', 'IWA', 'LBH', 'LBD', 'LBA',
       'SBH', 'SBD', 'SBA', 'WHH', 'WHD', 'WHA', 'SJH', 'SJD', 'SJA', 'VCH',
       'VCD', 'VCA', 'BSH', 'BSD', 'BSA', 'Bb1X2', 'BbMxH', 'BbAvH', 'BbMxD',
       'BbAvD', 'BbMxA', 'BbAvA', 'BbOU', 'BbMx>2.5', 'BbAv>2.5', 'BbMx<2.5',
       'BbAv<2.5', 'BbAH', 'BbAHh', 'BbMxAHH', 'BbAvAHH', 'BbMxAHA',
       'BbAvAHA'],
      dtype='object')
```

We can play with this.

One possibility is trying to predict a winner based on all other features.

Let's make a copy before further processingHagamos una copia antes de empezar

```
df_copy = (
    df.copy()
)  # [['HomeTeam','AwayTeam', 'FTHG','FTAG','FTR','HTHG', 'HTAG', 'HTR','HS','AS','HST'
```

```
df_copy_train, df_copy_test = train_test_split(df_copy)
```

Let's explore the data to try and understand how a Decision Tree works.

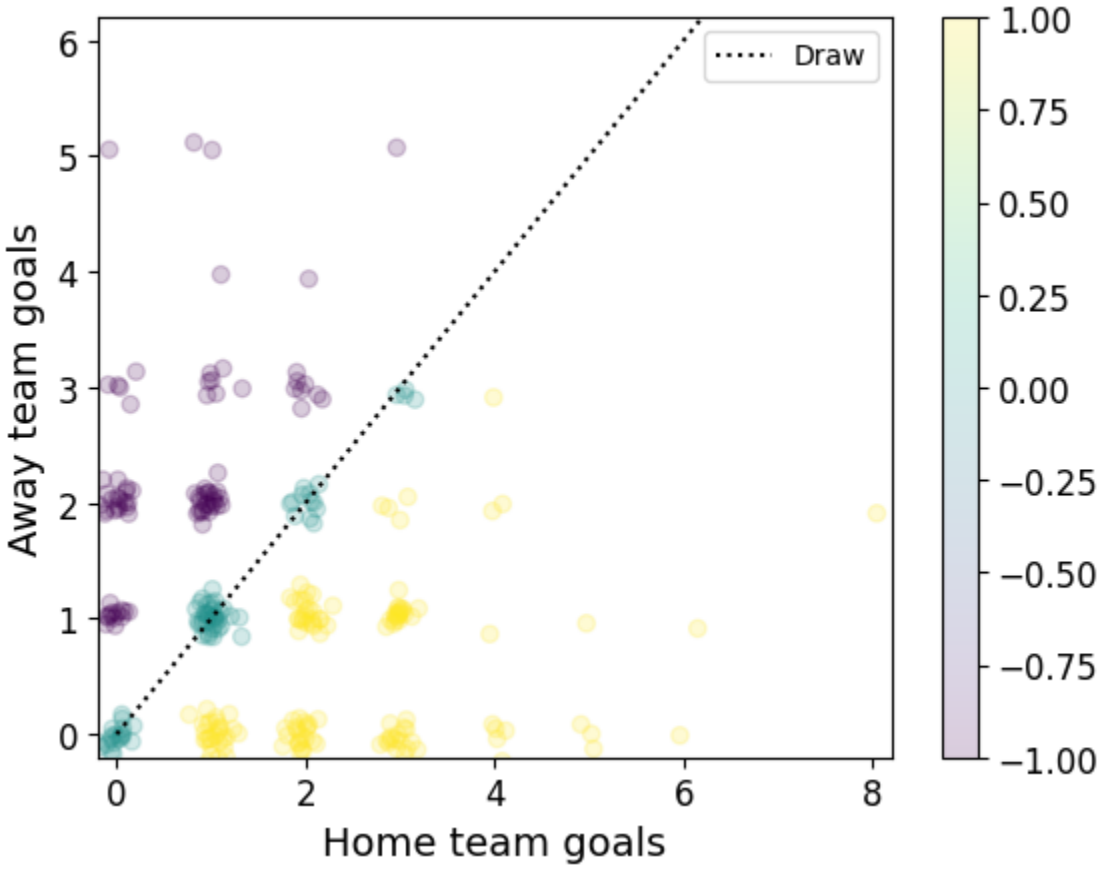
To plot, let's look at only two features for now.

```
target_train = np.zeros(len(df_copy_train))
target_train[df_copy_train["FTR"] == "H"] = 1.0
target_train[df_copy_train["FTR"] == "D"] = 0.0
target_train[df_copy_train["FTR"] == "A"] = -1.0
features_train = np.asarray(df_copy_train[["FTHG", "FTAG"]])

target_test = np.zeros(len(df_copy_test))
target_test[df_copy_test["FTR"] == "H"] = 1.0
target_test[df_copy_test["FTR"] == "D"] = 0.0
target_test[df_copy_test["FTR"] == "A"] = -1.0
features_test = np.asarray(df_copy_test[["FTHG", "FTAG"]])
```

```
features_scatter = features_train + 0.1 * np.random.randn(len(features_train), 2)
plt.scatter(features_scatter[:, 0], features_scatter[:, 1], c=target_train, alpha=0.2)
plt.colorbar()
xvals = np.linspace(0.0, 8.0, 10)
plt.plot(xvals, xvals, linestyle="dotted", color="black", label="Draw")
plt.xlabel("Home team goals")
```

```
plt.xlabel("Home team goals")
plt.ylabel("Away team goals")
plt.xlim(-0.2, 8.2)
plt.ylim(-0.2, 6.2)
plt.legend(loc="upper right")
```

```
<matplotlib.legend.Legend at 0x7a55adc4b710>
```



Let's forget about all the DT hyperparameters for now and just train a naive classifier:

```
dt = DecisionTreeClassifier(max_depth=None)
dt.fit(features_train, target_train)
```

```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

And let's see how it works:

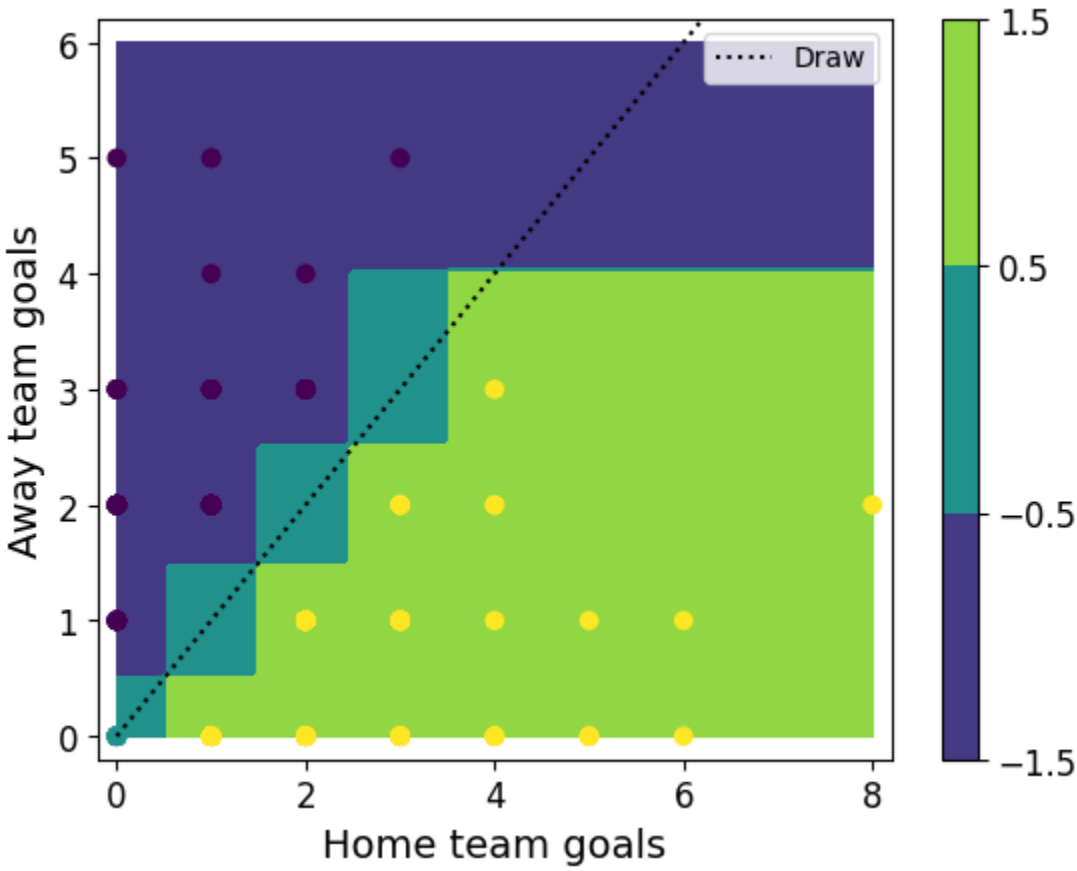
```
dt.predict(np.asarray([1.0, 2.0]).reshape(1, -1))
```

```
array([-1.])
```

```
xvals = np.linspace(0.0, 8.0, 100)
yvals = np.linspace(0.0, 6.0, 100)
X, Y = np.meshgrid(xvals, yvals)
```

```
Z = dt.predict(np.c_[X.ravel(), Y.ravel()]).reshape(X.shape)
plt.contourf(xvals, yvals, Z, levels=[-1.5, -0.5, 0.5, 1.5], label="DT")
plt.colorbar()
plt.scatter(features_train[:, 0], features_train[:, 1], c=target_train)
plt.plot(xvals, xvals, linestyle="dotted", color="black", label="Draw")
plt.xlabel("Home team goals")
plt.ylabel("Away team goals")
plt.xlim(-0.2, 8.2)
plt.ylim(-0.2, 6.2)
plt.legend(loc="upper right")
```

```
/tmp/ipython-input-15-3502440609.py:5: UserWarning: The following kwargs were not u
  plt.contourf(xvals, yvals, Z, levels=[-1.5, -0.5, 0.5, 1.5], label="DT")
<matplotlib.legend.Legend at 0x7a55ab802210>
```



It's really overfitting! Weird looking curves.

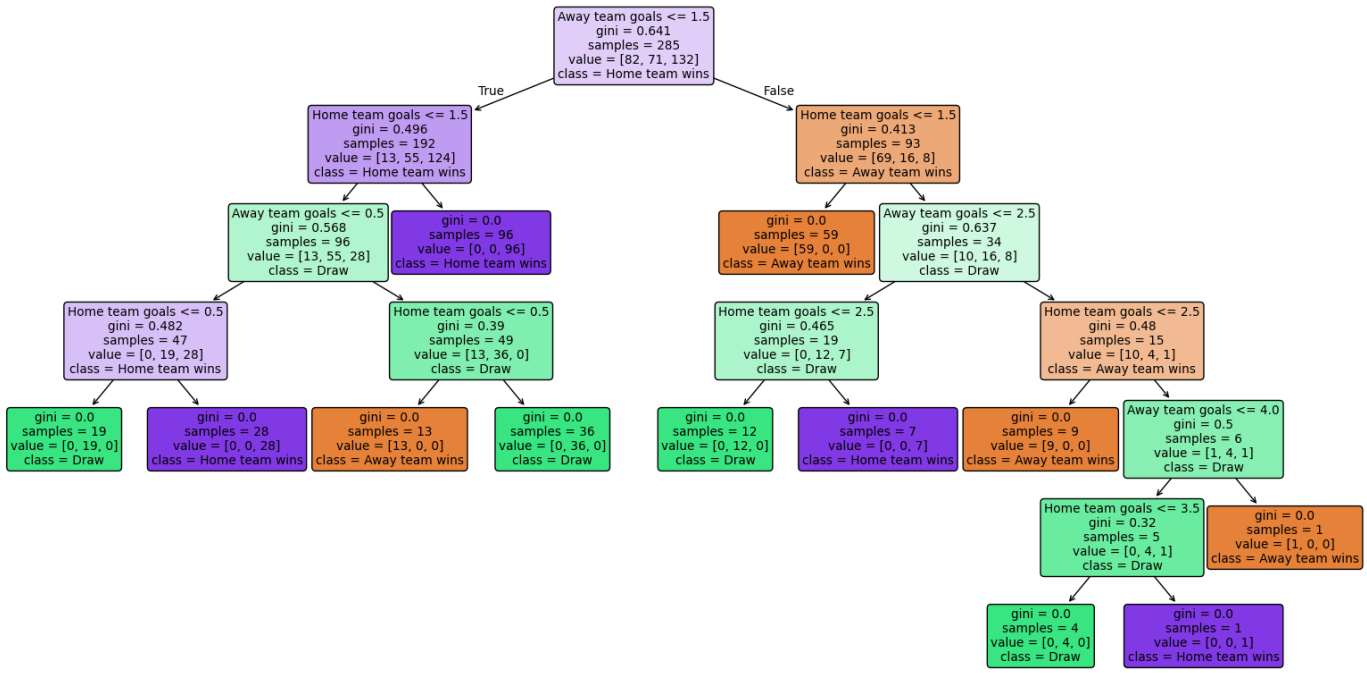
We wouldn't be able to tell from the confusion matrix though...

```
print(confusion_matrix(target_train, dt.predict(features_train)))
print(confusion_matrix(target_test, dt.predict(features_test)))
```

```
[[ 82   0   0]
 [  0  71   0]
 [  0   0 132]]
[[34  0  0]
 [ 0 20  2]
 [ 0  0 39]]
```

But an inspection of the defined tree would show it:

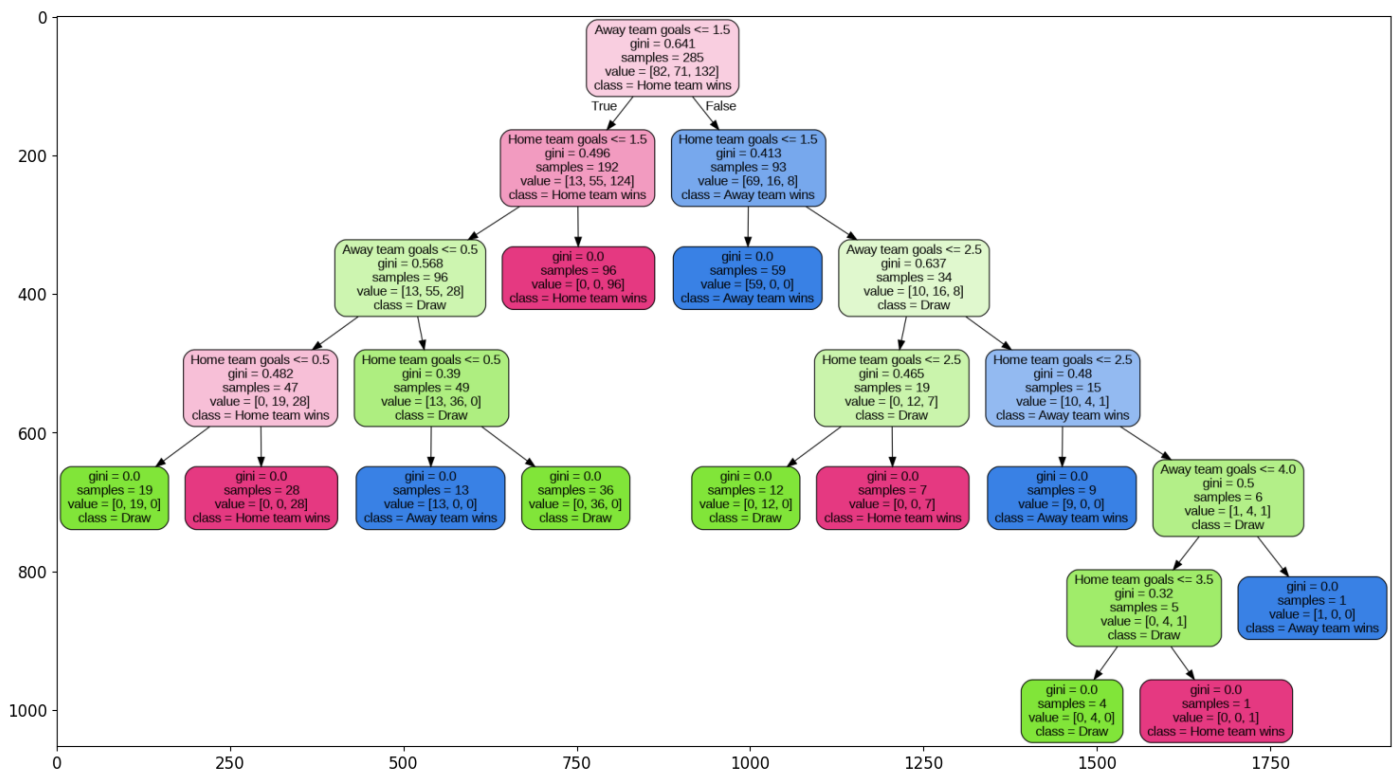
```
plt.figure(figsize=(20, 10))
tree.plot_tree(
    dt,
    filled=True,
    rounded=True,
    feature_names=["Home team goals", "Away team goals"],
    class_names=["Away team wins", "Draw", "Home team wins"],
)
plt.show()
```



This plot can also be exported as a `.dot` file and saved as `.png`.

```
tree.export_graphviz(
    dt,
    out_file="futbol.dot",
    feature_names=["Home team goals", "Away team goals"],
    class_names=["Away team wins", "Draw", "Home team wins"],
    rounded=True,
    filled=True,
)

# dot to png
if "google.colab" in sys.modules:
    !apt-get install graphviz

! dot -Tpng futbol.dot -o futbol.png

# Plot the image
img = cv2.imread("futbol.png")
plt.figure(figsize=(20, 10))
plt.imshow(img)
```

```
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
graphviz is already the newest version (2.42.2-6ubuntu0.1).
0 upgraded, 0 newly installed, 0 to remove and 35 not upgraded.
<matplotlib.image.AxesImage at 0x7a55a9e13ed0>
```



The DT can only do cuts in the individual features. Thus, it looks at home and away goals separately. But we know that in futbol the only important thing is to score more than the other team. The fact that it can only perform cuts on individual features can be a problem for DTs (but also the reason why we do not need to preprocess the features to remove units).

We can do some feature engineering

```
features_train = df_copy_train[["FTHG", "FTAG"]]
features_train["Local - Visitante"] = features_train["FTHG"] - features_train["FTAG"]
features_train = np.asarray(features_train)

features_test = df_copy_test[["FTHG", "FTAG"]]
features_test["Local - Visitante"] = features_test["FTHG"] - features_test["FTAG"]
features_test = np.asarray(features_test)
```

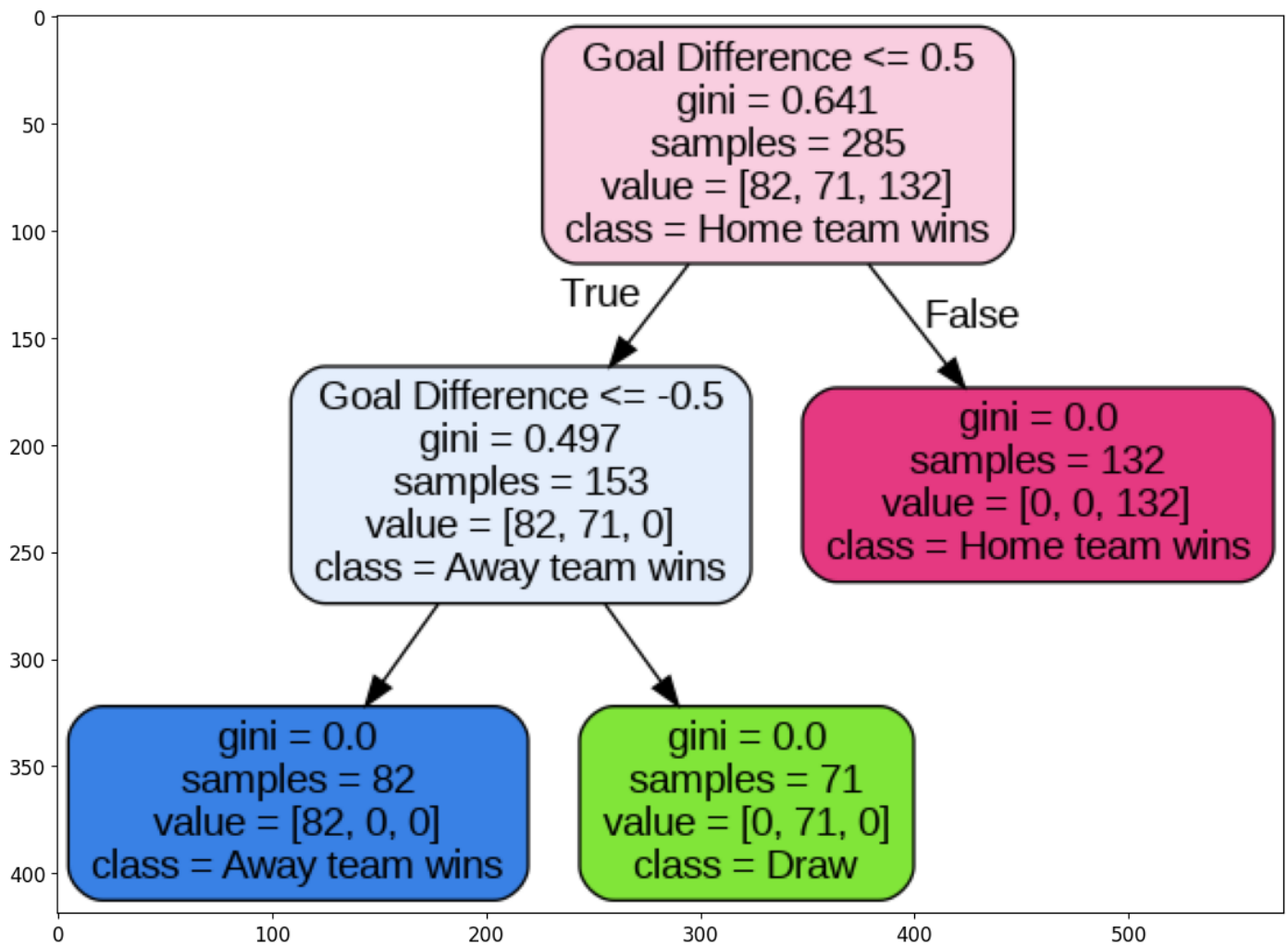
```
/tmp/ipython-input-19-4236061890.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/
  features_train["Local - Visitante"] = features_train["FTHG"] - features_train["FT
/tmp/ipython-input-19-4236061890.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/
  features_test["Local - Visitante"] = features_test["FTHG"] - features_test["FTAG"
```

```
dt = DecisionTreeClassifier()
dt.fit(features_train, target_train)
```

```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

```
tree.export_graphviz(
    dt,
    out_file="futbol.dot",
    feature_names=["Home", "Away", "Goal Difference"],
    class_names=["Away team wins", "Draw", "Home team wins"],
    rounded=True,
    filled=True,
)

# Convierto el dot a png
! dot -Tpng futbol.dot -o futbol.png

# Ploteamos el png
img = cv2.imread("futbol.png")
plt.figure(figsize=(20, 10))
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7a55a9ee6ed0>
```



Much better!

For such an easy example, DTs are not particularly useful. But now let's look at all the features that are less obvious in relation to wins. Let's remove the betting scores also.

```
names_train = df_copy_train[["HomeTeam", "AwayTeam"]]
features_train = df_copy_train.drop(
    [
        "Div",
        "Date",
        "Referee",
        "HomeTeam",
        "AwayTeam",
        "FTHG",
        "FTAG",
        "FTR",
        "HTHG",
        "HTAG",
        "HTR",
        "B365H",
        "B365D",
        "B365A",
        "BWH",
        "BWD",
        "BWA",
        "GBH",
        "GBD",
        "GBA",
        "IWH",
        "IWD",
        "IWA",
        "LBH",
        "LBD",
        "LBA"
```

```
        "LBA",
        "SBH",
        "SBD",
        "SBA",
        "WHH",
        "WHD",
        "WHA",
        "SJH",
        "SJD",
        "SJA",
        "VCH",
        "VCD",
        "VCA",
        "BSH",
        "BSD",
        "BSA",
        "Bb1X2",
        "BbMxH",
        "BbAvH",
        "BbMxD",
        "BbAvD",
        "BbMxA",
        "BbAvA",
        "BbOU",
        "BbMx>2.5",
        "BbAv>2.5",
        "BbMx<2.5",
        "BbAv<2.5",
        "BbAH",
        "BbAHh",
        "BbMxAHH",
        "BbAvAHH",
        "BbMxAHA",
        "BbAvAHA",
    ],
    axis=1,
)

names_test = df_copy_test[["HomeTeam", "AwayTeam"]]
features_test = df_copy_test.drop(
    [
        "Div",
        "Date",
        "Referee",
        "HomeTeam",
        "AwayTeam",
        "FTHG",
        "FTAG",
        "FTR",
        "HTHG",
        "HTAG",
        "HTR",
        "B365H",
        "B365D",
```

```
        "B365D",
        "B365A",
        "BWH",
        "BWD",
        "BWA",
        "GBH",
        "GBD",
        "GBA",
        "IWH",
        "IWD",
        "IWA",
        "LBH",
        "LBD",
        "LBA",
        "SBH",
        "SBD",
        "SBA",
        "WHH",
        "WHD",
        "WHA",
        "SJH",
        "SJD",
        "SJA",
        "VCH",
        "VCD",
        "VCA",
        "BSH",
        "BSD",
        "BSA",
        "Bb1X2",
        "BbMxH",
        "BbAvH",
        "BbMxD",
        "BbAvD",
        "BbMxA",
        "BbAvA",
        "BbOU",
        "BbMx>2.5",
        "BbAv>2.5",
        "BbMx<2.5",
        "BbAv<2.5",
        "BbAH",
        "BbAHh",
        "BbMxAHH",
        "BbAvAHH",
        "BbMxAHA",
        "BbAvAHA",
    ],
    axis=1,
)


features_train
```

	HS	AS	HST	AST	HF	AF	HC	AC	HY	AY	HR	AR
152	17	3	6	0	11	11	9	1	1	1	0	0
112	17	9	7	8	17	10	9	5	4	2	0	0
66	19	17	13	10	10	7	4	4	2	3	0	0
226	23	9	12	5	9	17	14	2	1	2	0	0
62	17	13	9	6	9	6	10	3	1	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...
247	18	7	9	3	7	9	4	3	2	1	0	0
269	19	5	8	1	9	11	10	4	2	0	0	0
8	26	8	19	5	9	2	7	4	0	0	0	0
206	16	7	8	4	6	14	10	5	0	1	0	0
178	7	12	4	7	9	12	5	11	1	3	0	0

285 rows × 12 columns

Next steps: Generate code with `features_train` | View recommended plots | New interactive sheet

We took off the team names since we don't care about them in order to predict. The DT could use them if we turn them into a categorical variable.

Let's now look at the DT hyperparameters to regularize the algorithm. In particular, we can choose whether it uses Gini or Entropy to calculate the impurity of a split. Generally, there is no difference, but by definition, Gini may favor the most frequent class more. The advantage is that it is faster.

Looking at the other hyperparameters, the options we have in `sklearn` are:

`max_depth`: By default, this is `None`; it controls the depth of the tree. `min_samples_split`: Sets the minimum number of samples a node must have to continue splitting it. `min_samples_leaf`: The minimum number of samples a leaf (i.e., the end node) must have. `min_weight_fraction_leaf`: The minimum weighted fraction of samples a leaf must have. `max_leaf_nodes`: Maximum number of leaves. `max_features`: Maximum number of features evaluated in a split.

If you raise the minimum values or lower the maximum values, you are restricting the tree and regularizing the model.

There are other regularization methods, such as pruning, in which you train without restrictions and then remove unnecessary nodes.

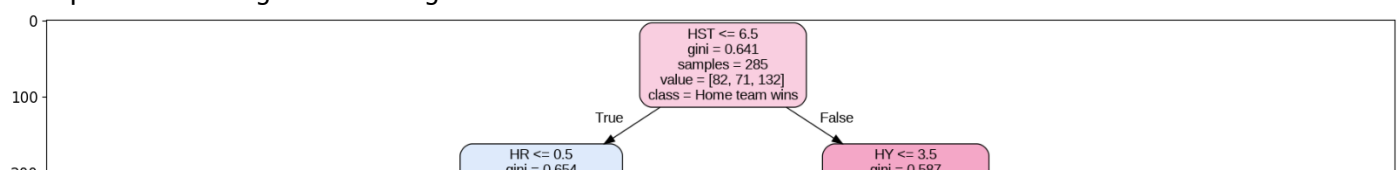
```
dt = DecisionTreeClassifier()
```

```
# dt?
```

Let's play:

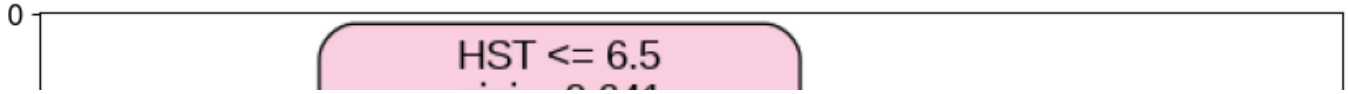
```
dt = DecisionTreeClassifier(max_depth=3)
dt.fit(features_train, target_train)
tree.export_graphviz(
    dt,
    out_file="futbol.dot",
    feature_names=features_train.columns,
    class_names=["Away team wins", "Draw", "Home team wins"],
    rounded=True,
    filled=True,
)

# Convierto el dot a png
! dot -Tpng futbol.dot -o futbol.png

# Ploteamos el png
img = cv2.imread("futbol.png")
plt.figure(figsize=(20, 10))
plt.imshow(img)
```

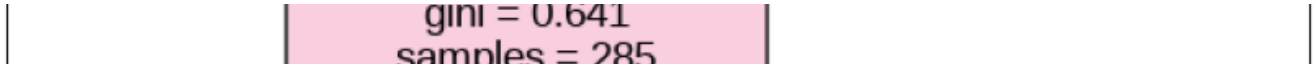
```
<matplotlib.image.AxesImage at 0x7a55a9da6310>
```




```
dt = DecisionTreeClassifier(min_samples_leaf=50, max_depth=100)
dt.fit(features_train, target_train)
tree.export_graphviz(
    dt,
    out_file="futbol.dot",
    feature_names=features_train.columns,
    class_names=["Away team wins", "Draw", "Home team wins"],
    rounded=True,
    filled=True,
)

# Convierto el dot a png
! dot -Tpng futbol.dot -o futbol.png

# Ploteamos el png
img = cv2.imread("futbol.png")
plt.figure(figsize=(20, 10))
plt.imshow(img)
```

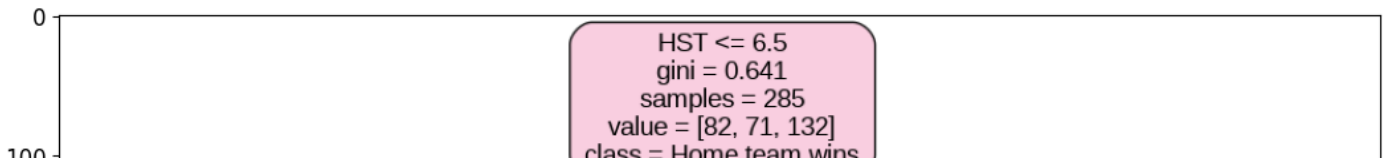
```
<matplotlib.image.AxesImage at 0x7a55a9c248d0>
```






```
dt = DecisionTreeClassifier(max_leaf_nodes=6)
dt.fit(features_train, target_train)
tree.export_graphviz(
    dt,
    out_file="futbol.dot",
    feature_names=features_train.columns,
    class_names=["Away team wins", "Draw", "Home team wins"],
    rounded=True,
    filled=True,
)

# Convierto el dot a png
! dot -Tpng futbol.dot -o futbol.png

# Ploteamos el png
img = cv2.imread("futbol.png")
plt.figure(figsize=(20, 10))
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7a55a9c8ce90>
```



Now let's really optimize things

```
dt = DecisionTreeClassifier()
params = {
    "max_depth": [2, 3, 5],
    "min_samples_leaf": [10, 50],
    "max_leaf_nodes": [3, 4, 5],
}
grid = GridSearchCV(dt, params, cv=10, scoring="accuracy")
grid.fit(features_train, target_train)
```

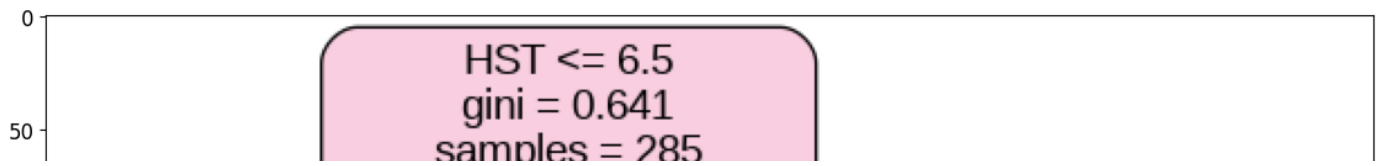
```
▸ GridSearchCV
▸ best_estimator_: DecisionTreeClassifier
  ▸ DecisionTreeClassifier
```

```
grid.best_params_
```

```
{'max_depth': 2, 'max_leaf_nodes': 3, 'min_samples_leaf': 50}
```

```
model = grid.best_estimator_


tree.export_graphviz(
    model,
    out_file="futbol.dot",
    feature_names=features_train.columns,
    class_names=["Away team wins", "Draw", "Home team wins"],
    rounded=True,
    filled=True,
)

# Convierto el dot a png
! dot -Tpng futbol.dot -o futbol.png

# Ploteamos el png
img = cv2.imread("futbol.png")
plt.figure(figsize=(20, 10))
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7a55a9e36510>
```




```
predicts = cross_val_predict(model, features_train, target_train, cv=5)
print(confusion_matrix(target_train, predicts))
print(
    recall_score(
        np.where(target_train == -1.0, 1.0, 0.0), np.where(predicts == -1.0, 1.0, 0.0)
    )
)
print(
    recall_score(
        np.where(target_train == 0.0, 1.0, 0.0), np.where(predicts == 0.0, 1.0, 0.0)
    )
)
print(
    recall_score(
        np.where(target_train == 1.0, 1.0, 0.0), np.where(predicts == 1.0, 1.0, 0.0)
    )
)

print(
    accuracy_score(
        np.where(target_train == -1.0, 1.0, 0.0), np.where(predicts == -1.0, 1.0, 0.0)
    )
)
print(
    accuracy_score(
```

```
        np.where(target_train == 0.0, 1.0, 0.0), np.where(predicts == 0.0, 1.0, 0.0)
    )
)
print(
    accuracy_score(
        np.where(target_train == 1.0, 1.0, 0.0), np.where(predicts == 1.0, 1.0, 0.0)
    )
)

print(confusion_matrix(target_test, model.predict(features_test)))
```

```
[[38  0 44]
 [29  0 42]
 [33  0 99]]
0.4634146341463415
0.0
0.75
0.6280701754385964
0.7508771929824561
0.5824561403508772
[[12  0 22]
 [ 8  0 14]
 [11  0 28]]
```

```
print(model.predict_proba(features_train[:3]))
print(np.argmax(model.predict_proba(features_train[:3]), axis=1) - 1)
print(model.predict(features_train[:3]))
```

```
[[0.42268041 0.30927835 0.26804124]
 [0.39215686 0.19607843 0.41176471]
 [0.39215686 0.19607843 0.41176471]]
[-1  1  1]
[-1.  1.  1.]
```

```
print(np.where(model.predict(features_train[:3]) == -1.0, 1.0, 0.0))
print(np.where(model.predict(features_train[:3]) == 0.0, 1.0, 0.0))
print(np.where(model.predict(features_train[:3]) == 1.0, 1.0, 0.0))
```

```
[1. 0. 0.]
[0. 0. 0.]
[0. 1. 1.]
```

```
thresholds = [0.2, 0.4, 0.6, 0.8]
for threshold in thresholds:
    print("Threshold " + str(threshold) + "\n")
    y_pred_away = np.where(
        model.predict_proba(features_train)[:, 0] >= threshold, 1.0, 0.0
    )
    y_pred_draw = np.where(
        model.predict_proba(features_train)[:, 1] >= threshold, 1.0, 0.0
    )
    y_pred_home = np.where(
        model.predict_proba(features_train)[:, 2] >= threshold, 1.0, 0.0
```

```
    )
    print(accuracy_score(np.where(target_train == -1.0, 1.0, 0.0), y_pred_away))
    print(accuracy_score(np.where(target_train == 0.0, 1.0, 0.0), y_pred_draw))
    print(accuracy_score(np.where(target_train == 1.0, 1.0, 0.0), y_pred_home))
    print("\n")
```

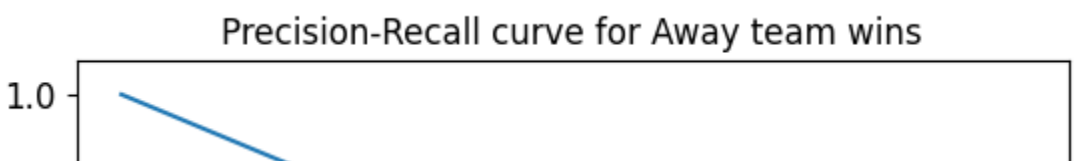
```
Threshold 0.2

0.6210526315789474
0.35789473684210527
0.4631578947368421


Threshold 0.4

0.6596491228070176
0.7508771929824561
0.6210526315789474


Threshold 0.6

0.712280701754386
0.7508771929824561
0.6526315789473685


Threshold 0.8

0.712280701754386
0.7508771929824561
0.5368421052631579
```

```
print(
    recall_score(
        np.where(target_train == -1.0, 1.0, 0.0),
        np.where(np.argmax(model.predict_proba(features_train), axis=1) == 0, 1.0, 0.0)
    )
)
print(
    recall_score(
        np.where(target_train == 0.0, 1.0, 0.0),
        np.where(np.argmax(model.predict_proba(features_train), axis=1) == 1, 1.0, 0.0)
    )
)
print(
    recall_score(
        np.where(target_train == 1.0, 1.0, 0.0),
        np.where(np.argmax(model.predict_proba(features_train), axis=1) == 2, 1.0, 0.0)
    )
)
```

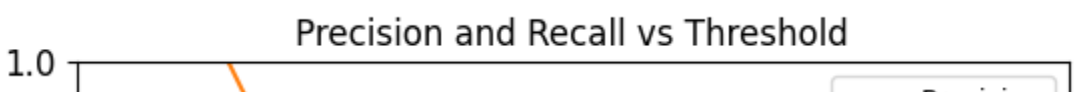
```
0.5
```

```
0.0
0.803030303030303
```

```
from sklearn.metrics import precision_recall_curve

class_names = ["Away team wins", "Draw", "Home team wins"]
for nclass_label, class_label in enumerate([-1.0, 0.0, 1.0]):
    precision, recall, thresholds = precision_recall_curve(
        target_train, model.predict_proba(features_train)[:, 0], pos_label=class_label
    )
    plt.plot(precision[:-1], recall[:-1])
    plt.title("Precision-Recall curve for " + str(class_names[nclass_label]))
    plt.xlabel("Precision")
    plt.ylabel("Recall")
    # plt.xlim(0.0,1.0)
    # plt.ylim(0.0,1.0)
    plt.show()
```




```
plt.plot(thresholds, precision[:-1], label="Precision")
plt.plot(thresholds, recall[:-1], label="Recall")
plt.title("Precision and Recall vs Threshold")
plt.xlabel("Threshold")
plt.ylabel("Metric")
plt.legend()
plt.xlim(0.0, 1.0)
plt.ylim(0.0, 1.0)
plt.show()
```



It's hard to predict draws!

Exercise:

Add bets as features and optimize the DT. What do you find? Can you assess feature importance? In particular, is it more important to see at "in-game" info or "pre-game" bets?

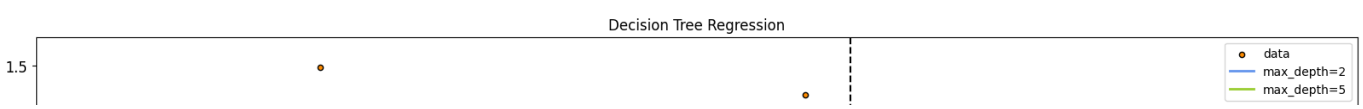
Regression

Let's see how DTs can be used for regression with a synthetic dataset:

```
# Create a random dataset
rng = np.random.RandomState(1)
X = np.sort(5 * rng.rand(80, 1), axis=0)
y = np.sin(X).ravel()
y[::5] += 3 * (0.5 - rng.rand(16))

X_test = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]


# Plot the results
plt.figure()
plt.scatter(X, y, s=20, edgecolor="black", c="darkorange", label="data")
plt.xlabel("data")
plt.ylabel("target")
# plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

We can see how DTs operate by exploring different depths:

```
# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=2)
regr_2 = DecisionTreeRegressor(max_depth=5, min_samples_leaf=5)
regr_1.fit(X, y)
regr_2.fit(X, y)
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)


# Plot the results
plt.figure(figsize=(20, 10))
plt.scatter(X, y, s=20, edgecolor="black", c="darkorange", label="data")
plt.plot(X_test, y_1, color="cornflowerblue", label="max_depth=2", linewidth=2)
plt.plot(X_test, y_2, color="yellowgreen", label="max_depth=5", linewidth=2)
plt.axvline(3.133, linestyle="dashed", color="black")
plt.axhline(0.571, linestyle="dashed", color="black")
plt.axhline(-0.667, linestyle="dashed", color="black")
plt.xlabel("X")
plt.ylabel("t")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```



The tree decides on a predicted value by doing cuts in feature space. `max_depth` controls the number

The tree decides on a predicted value by doing cuts in feature space. "max_depth" controls the number of cuts the algorithm makes. Let's see how the target is assigned:

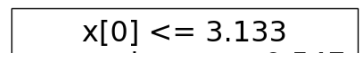
```
np.mean(y)
```

```
np.float64(0.12215899268094885)
```

```
mean_squared_error(np.mean(y) * np.ones(len(y)), y)
```

```
0.5471130002937142
```

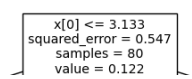
```
plt.figure(figsize=(20, 10))
tree.plot_tree(regr_1)
plt.show()
```




```
y_first_cut = y[(X[:, 0] <= 3.133)]
print(np.mean(y_first_cut), np.mean(y[(X[:, 0] > 3.133)]))
print(mean_squared_error(np.mean(y_first_cut) * np.ones(len(y_first_cut)), y_first_cut)
```

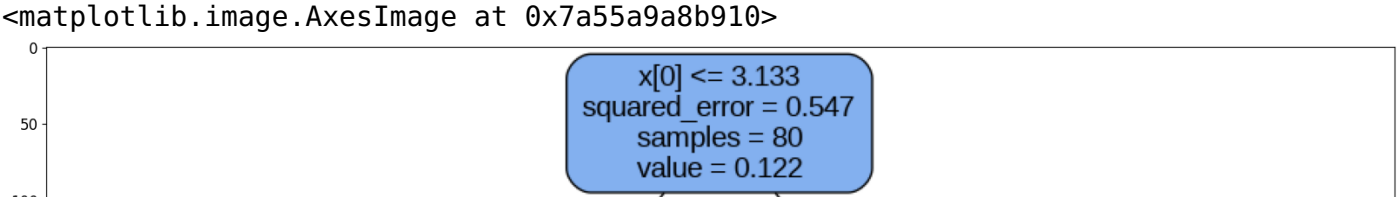
```
0.5711567593351029 -0.6674577693660114
0.23136965662280937
```

```
plt.figure(figsize=(20, 10))
tree.plot_tree(regr_2)
plt.show()
```




```
tree.export_graphviz(regr_1, out_file="reg_tree.dot", rounded=True, filled=True)

# Convierto el dot a png
! dot -Tpng reg_tree.dot -o reg_tree.png

# Ploteamos el png
img = cv2.imread("reg_tree.png")
plt.figure(figsize=(20, 10))
plt.imshow(img)
```

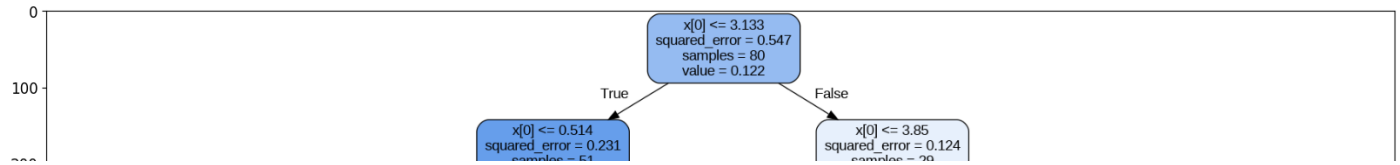
```
<matplotlib.image.AxesImage at 0x7a55a9a8b910>
```




```
tree.export_graphviz(regr_2, out_file="reg_tree.dot", rounded=True, filled=True)
```

```
# Convierto el dot a png
! dot -Tpng reg_tree.dot -o reg_tree.png

# Ploteamos el png
img = cv2.imread("reg_tree.png")
plt.figure(figsize=(20, 20))
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7a55a9837790>
```



To select the cut, it does not consider neither Gini nor entropy, it uses the MSE! Additionally, it assigns as predicted target the mean of all features before the cut is made.

Exercise:

Let's consider the California dataset. Train a DT to predict the house price. Optimize the hyperparameter and report the RMSE and a predicted vs actual house value.

```
HOUSING_PATH = "datasets"
import pandas as pd


def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)


### from Geron

if "google.colab" in sys.modules:
    import tarfile

    DOWNLOAD_ROOT = "https://github.com/ageron/handson-ml2/raw/master/"
    HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

    !mkdir -p ./datasets/housing

    def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
        os.makedirs(housing_path, exist_ok=True)
        tgz_path = os.path.join(housing_path, "housing.tgz")
        # urllib.request.urlretrieve(housing_url, tgz_path)
        !wget {HOUSING_URL} -P {housing_path}
        housing_tgz = tarfile.open(tgz_path)
        housing_tgz.extractall(path=housing_path)
        housing_tgz.close()
```

```
        housing_tgz.close()

    # Corramos la función
    fetch_housing_data()

else:
    print("Not running on Google Colab. This cell is did not do anything.")
```

```
--2025-07-04 13:29:46--  https://github.com/ageron/handson-ml2/raw/master/datasets/
Resolving github.com (github.com)... 20.27.177.113
Connecting to github.com (github.com)|20.27.177.113|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/ageron/handson-ml2/master/datasets/hous
--2025-07-04 13:29:47--  https://raw.githubusercontent.com/ageron/handson-ml2/maste
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.109.133,
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.109.133
HTTP request sent, awaiting response... 200 OK
Length: 409488 (400K) [application/octet-stream]
Saving to: ‘datasets/housing.tgz’

housing.tgz         100%[===================>] 399.89K  2.07MB/s    in 0.2s

2025-07-04 13:29:47 (2.07 MB/s) - ‘datasets/housing.tgz’ saved [409488/409488]
```

```
housing_pre = load_housing_data()
from sklearn.model_selection import StratifiedShuffleSplit

housing_pre["income_cat"] = pd.cut(
    housing_pre["median_income"],
    bins=[0.0, 1.5, 3.0, 4.5, 6.0, np.inf],
    labels=[1, 2, 3, 4, 5],
)

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=445543)
for train_index, test_index in split.split(housing_pre, housing_pre["income_cat"]):
    california_housing_train = housing_pre.loc[train_index]
    california_housing_test = housing_pre.loc[test_index]

for set_ in (california_housing_train, california_housing_test):
    set_.drop("income_cat", axis=1, inplace=True)


housing = california_housing_train.copy()

problematic_columns = ["median_house_value", "housing_median_age", "median_income"]
max_values = []
for col in problematic_columns:
    max_value = housing[col].max()
    print(
        f"{col}: {sum(housing[col] == max_value)} districts with {col} = {max_value} ({
    )
    max_values.append(max_value)

housing_clean = housing.copy()
```

```
housing_clean = housing.copy()
for col, max_value in zip(problematic_columns, max_values):
    housing_clean = housing_clean[housing_clean[col] != max_value]

housing_test = california_housing_test.copy()
housing_test_clean = housing_test.copy()
for col, max_value in zip(problematic_columns, max_values):
    housing_test_clean = housing_test_clean[housing_test_clean[col] != max_value]
```

```
median_house_value: 762 districts with median_house_value = 500001.0 (4.61%).
housing_median_age: 997 districts with housing_median_age = 52.0 (6.04%).
median_income: 42 districts with median_income = 15.0001 (0.25%).
```

```
housing_clean["rooms_per_household"] = (
    housing_clean["total_rooms"] / housing_clean["households"]
)
housing_clean["bedrooms_per_room"] = (
    housing_clean["total_bedrooms"] / housing_clean["total_rooms"]
)
housing_clean["population_per_household"] = (
    housing_clean["population"] / housing_clean["households"]
)

housing_test_clean["rooms_per_household"] = (
    housing_test_clean["total_rooms"] / housing_test_clean["households"]
)
housing_test_clean["bedrooms_per_room"] = (
    housing_test_clean["total_bedrooms"] / housing_test_clean["total_rooms"]
)
housing_test_clean["population_per_household"] = (
    housing_test_clean["population"] / housing_test_clean["households"]
)


housing_labels = housing_clean["median_house_value"].copy()
housing_clean = housing_clean.drop(
    "median_house_value", axis=1
)  # drop labels for training set
housing_num = housing_clean.drop("ocean_proximity", axis=1)

housing_test_labels = housing_test_clean["median_house_value"].copy()
housing_test_clean = housing_test_clean.drop(
    "median_house_value", axis=1
)  # drop labels for training set
housing_test_num = housing_test_clean.drop("ocean_proximity", axis=1)
```

Some useful preprocessing

```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
```

```
from sklearn.preprocessing import OneHotEncoder

num_pipeline = Pipeline(
    [
        ("imputer", SimpleImputer(strategy="median")),  # hay mas opciones aca
        ("std_scaler", StandardScaler()),
    ]
)

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer(
    [
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ]
)

housing_prepared = full_pipeline.fit_transform(housing_clean)
housing_test_prepared = full_pipeline.transform(housing_test_clean)
```

Another nice example

This is verbatim from `sklearn` documentation:

```
from sklearn.datasets import fetch_olivetti_faces
from sklearn.utils.validation import check_random_state

# Load the faces datasets
data, targets = fetch_olivetti_faces(return_X_y=True)
```

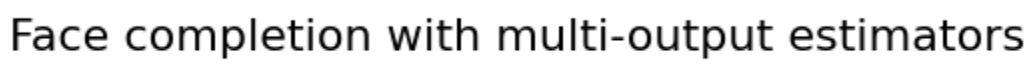
```
downloading Olivetti faces from https://ndownloader.figshare.com/files/5976027 to /
```

We can try to predict the lower half of a face using the upper half:

```
train = data[targets < 30]
test = data[targets >= 30]  # Test on independent people

# Test on a subset of people
n_faces = 5
rng = check_random_state(4)
face_ids = rng.randint(test.shape[0], size=(n_faces,))
test = test[face_ids, :]

n_pixels = data.shape[1]
# Upper half of the faces
X_train = train[:, : (n_pixels + 1) // 2]
# Lower half of the faces
```

```
# Lower half of the faces
y_train = train[:, n_pixels // 2 :]
X_test = test[:, : (n_pixels + 1) // 2]
y_test = test[:, n_pixels // 2 :]

# Fit estimators
ESTIMATORS = {
    "Decision Trees": DecisionTreeRegressor(),
    "Linear regression": LinearRegression(),
    "Ridge": RidgeCV(),
}

y_test_predict = dict()
for name, estimator in ESTIMATORS.items():
    estimator.fit(X_train, y_train)
    y_test_predict[name] = estimator.predict(X_test)

# Plot the completed faces
image_shape = (64, 64)

n_cols = 1 + len(ESTIMATORS)
plt.figure(figsize=(2.0 * n_cols, 2.26 * n_faces))
plt.suptitle("Face completion with multi-output estimators", size=16)

for i in range(n_faces):
    true_face = np.hstack((X_test[i], y_test[i]))

    if i:
        sub = plt.subplot(n_faces, n_cols, i * n_cols + 1)
    else:
        sub = plt.subplot(n_faces, n_cols, i * n_cols + 1, title="true faces")

    sub.axis("off")
    sub.imshow(
        true_face.reshape(image_shape), cmap=plt.cm.gray, interpolation="nearest"
    )

    for j, est in enumerate(sorted(ESTIMATORS)):
        completed_face = np.hstack((X_test[i], y_test_predict[est][i]))

        if i:
            sub = plt.subplot(n_faces, n_cols, i * n_cols + 2 + j)

        else:
            sub = plt.subplot(n_faces, n_cols, i * n_cols + 2 + j, title=est)

        sub.axis("off")
        sub.imshow(
            completed_face.reshape(image_shape),
            cmap=plt.cm.gray,
            interpolation="nearest",
        )

plt.show()
```

```
plt.show()
```



Bagging and Random Forests

Bagging is a particular type of **ensemble** training. Ensemble methods combine different estimators to build a better one, usually reducing the variance and overfitting. In bagging, which originates from **bootstrapping agreggating**, we bootstrap the data and train a model for each bootstrapped dataset. The overall model is then an average of the trained predictors.

A **RandomForest** is a bagging model where the base estimator is a Decision Tree and where additionally **feature bagging** is performed. That is, at each decision step for each bootstrapped dataset, only a subset of features chosen at random is considered to select the optimal cut. This further increases the variability of the ensembled models. The added stochasticity can make the decision frontier more irregular, but usually increases performance.

Let's see this using an example.

```
# Let us define a couple of useful functions (if in colab, otherwise, take from utils m

### From Rodrigo Diaz


def plot_clasi(
    x,
    t,
    ws,
    labels=[],
    xp=[-1.0, 1.0],
    thr=[
        0,
    ],
    spines="zero",
    equal=True,
    join_centers=False,
    margin=None,
):
    """
    Figura con el resultado del ajuste lineal
    """
    assert len(labels) == len(ws) or len(labels) == 0
    assert len(ws) == len(thr)

    if margin is None:
        margin = [False] * len(ws)
```

```
else:
    margin = np.atleast_1d(margin)
assert len(margin) == len(ws)

if len(labels) == 0:
    labels = np.arange(len(ws)).astype("str")

# Agregemos el vector al plot
fig = plt.figure(figsize=(9, 7))
ax = fig.add_subplot(111)

xc1 = x[t == np.unique(t).max()]
xc2 = x[t == np.unique(t).min()]

ax.plot(*xc1.T, "ob", mfc="None", label="C1")
ax.plot(*xc2.T, "or", mfc="None", label="C2")

for i, w in enumerate(ws):
    # Compute vector norm
    wnorm = np.sqrt(np.sum(w**2))

    # Ploteo vector de pesos
    x0 = 0.5 * (xp[0] + xp[1])
    ax.quiver(
        0,
        thr[i] / w[1],
        w[0] / wnorm,
        w[1] / wnorm,
        color="C{}".format(i + 2),
        scale=10,
        label=labels[i],
        zorder=10,
    )

    # ploteo plano perpendicular
    xp = np.array(xp)
    yp = (thr[i] - w[0] * xp) / w[1]

    plt.plot(xp, yp, "-", color="C{}".format(i + 2))

    # Plot margin
    if margin[i]:
        for marg in [-1, 1]:
            ym = yp + marg / w[1]
            plt.plot(xp, ym, ":", color="C{}".format(i + 2))

if join_centers:
    # Ploteo línea que une centros de los conjuntos
    mu1 = xc1.mean(axis=1)
    mu2 = xc2.mean(axis=1)
    ax.plot([mu1[0], mu2[0]], [mu1[1], mu2[1]], "o:k", mfc="None", ms=10)

ax.legend(loc=0, fontsize=12)
if equal
```

```
    if equal:
        ax.set_aspect("equal")

    if spines is not None:
        for a in ["left", "bottom"]:
            ax.spines[a].set_position("zero")
        for a in ["top", "right"]:
            ax.spines[a].set_visible(False)

    return


def makew(fitter):
    # # Obtengamos los pesos y normalicemos
    w = fitter.coef_.copy()

    # # Incluye intercept
    if fitter.fit_intercept:
        w = np.hstack([fitter.intercept_.reshape(1, 1), w])

    # # Normalizon
    # w /= np.linalg.norm(w)
    return w.T


# Utility from Geron
def plot_decision_regions(
    clf,
    X,
    t,
    axes=None,
    npointsgrid=500,
    legend=False,
    plot_training=True,
    figkwargs={"figsize": [12, 8]},
    contourkwargs={"alpha": 0.3},
):
    """
    Plot decision regions produced by classifier.

    :param Classifier clf: sklearn classifier supporting XXX
    """

    fig = plt.figure(**figkwargs)
    ax = fig.add_subplot(111)

    if axes is None:
        dx = X[:, 0].max() - X[:, 0].min()
        dy = X[:, 1].max() - X[:, 1].min()
        axes = [
            X[:, 0].min() - 0.1 * dx,
            X[:, 0].max() + 0.1 * dx,
            X[:, 1].min() - 0.1 * dy,
            X[:, 1].max() + 0.1 * dy
```

```
            X[:, 1].max() + 0.1 * dy,
        ]

    # Define grid for regions
    x1s = np.linspace(axes[0], axes[1], npointsgrid)
    x2s = np.linspace(axes[2], axes[3], npointsgrid)
    x1, x2 = np.meshgrid(x1s, x2s)

    # Make predictions on points of grid; reshape to grid format
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)

    # custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    ax.contourf(x1, x2, y_pred, **contourkwargs)

    #     custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
    #         plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)

    if plot_training:
        for label in np.unique(t):
            ax.plot(
                X[:, 0][t == label], X[:, 1][t == label], "o", label="C{}".format(label
            )

    # Axis
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.ylabel(r"$x_2$", fontsize=18, rotation=0)

    if legend:
        plt.legend(loc="lower right", fontsize=14)

    plt.show()
    return fig
```

Example with Moons dataset

Let's use a simple non-linearly separable dataset to exemplify this:

```
from sklearn.datasets import make_moons

X, t = make_moons(n_samples=400, noise=0.25, random_state=1234)
```

```
plot_clasi(X, t, [], [], [], [], spines=None)
```


```
# Split
X, X_test, t, t_test = train_test_split(X, t, test_size=0.2)
```

Simple RF training

```
from sklearn.ensemble import RandomForestClassifier
```

```
rf = RandomForestClassifier(n_estimators=100, max_depth=2, n_jobs=6)
rf.fit(X, t)
```

```
▾ RandomForestClassifier
RandomForestClassifier(max_depth=2, n_jobs=6)
```

```
fig = plot_decision_regions(
    rf,
    X,
    t,
    legend=True,
    npointsgrid=500,
    figkwargs={"figsize": [12, 8]},
    contourkwargs={"alpha": 0.5, "levels": 5, "cmap": "viridis"},
)
```

This will not generalize well...

```
from sklearn.metrics import accuracy_score

y_train = rf.predict(X)
y_test = rf.predict(X_test)
print("Accuracy (train): {:.3f}".format(accuracy_score(t, y_train)))
print("Accuracy (test): {:.3f}".format(accuracy_score(t_test, y_test)))
```

```
Accuracy (train): 0.906
Accuracy (test): 0.887
```

Below you can solve this by optimizing the Random Forest using `GridSearchCV`.

Another feature of RFs is their interpretability. Since it's based on a white box algorithm, Decision Trees, we can use to study the learned properties. In particular, we can gauge feature importance by inspecting the fitted DTs. For a given DT, the most important features are closer to the root. We can perform statistics on the feature importances by averaging over the fitted DTs.

`sklearn` stores this through `feature_importances_`

```
print(rf.feature_importances_)
for name, score in zip(["x_1", "x_2"], rf.feature_importances_):
    print(name, score)
```

```
[0.47884546 0.52115454]
x_1 0.47884546065207095
x_2 0.521154539347929
```

Exercise

Train an optimize Random Forest by exploring the possible hyperparameters. Compare with a simpler classifier like an optimized polynomial Logistic Regressor or a optimized Decision Tree.

Boosted and Boosted Decision Trees

Boosting methods are another example of **ensemble** methods. They also combine different instances of a base estimator. However, in boosting each successive instance learns both from the data and from the previous estimator. That is, it learns to "correct" the previous estimator.

- It usually **greatly improves** the performance of **weak predictors**.
- It's not easily paralellizable.
- It's greedy. Each step seeks to be as good as possible without thinking of global strategies.

We'll see two types of boosting: AdaBoosting and GradientBoosting.

```
from sklearn.ensemble import (
    AdaBoostClassifier,
    AdaBoostRegressor,
    GradientBoostingRegressor,
    GradientBoostingClassifier,
)
```

AdaBoost

In AdaBoost, at each step the data points are weighted according to the performance of the previous estimator (they are initiated to 1)

That is, for steps $i = 1, \ldots, N$

1. We train a predictor h_i.
2. We update the per sample weight $w_{n,i} = f(w_{n,i-1}, h_i)$ The final predictor combines all N predictors

predictors.

The two `sklearn` classes are `AdaBoostClassifier` and `AdaBoostRegressor`, with algorithm specific hyperparameters:

The AdaBoost-specific hyperparameters are:

- `estimator`: The weak predictor used. By default, it is a `DecisionStump` (a `DecisionTree` with `max_depth=1`).
- `n_estimators`: How many estimators to use.
- `learning_rate`: The learning rate when taking a new estimator. The lower the learning_rate, the more estimators are needed to fit the data. This is a regularizer for the algorithm.
- `loss`: Exclusively for regression. This is the loss function used by the algorithm. The options are `linear`, `square`, and `exponential`.

From the fitted class, you can obtain:

- `estimators_`: The list of estimators.
- `estimators_weights_`: The weights of each estimator. 1 for `SAMME.R` classification, not equal to 1 for regression and classification with `SAMME`.
- `estimators_errors_`: The error of each estimator when evaluated on the dataset. This is not the error when applying the ensemble.
- `feature_importances_`: The importance of the features.

In addition, AdaBoost has the `.staged_` function that allows the ensemble to be evaluated at each step as if it were complete.

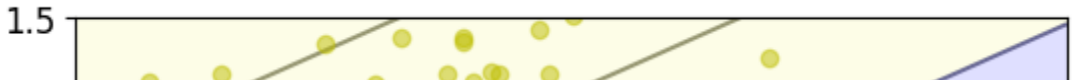
Example

```
X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

```
from matplotlib.colors import ListedColormap


def plot_decision_boundary(
    clf, X, y, axes=[-1.5, 2.45, -1, 1.5], alpha=0.5, contour=True
):
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(["#fafab0", "#9898ff", "#a0faa0"])
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap)
    if contour:
        custom_cmap2 = ListedColormap(["#7d7d58", "#4c4c7f", "#507d50"])
```

```
        plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
    plt.plot(X[:, 0][y == 0], X[:, 1][y == 0], "yo", alpha=alpha)
    plt.plot(X[:, 0][y == 1], X[:, 1][y == 1], "bs", alpha=alpha)
    plt.axis(axes)
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
```

```
ridgeCV = RidgeCV()
ridgeCV.fit(X_train, y_train, sample_weight=np.where(X_train[:, 1] > -0.5, 100.0, 1.0))

plot_decision_boundary(ridgeCV, X, y)
# plt.axhline(-0.5)
```




```
n_estimators = 300
# AdaBoostClassifier(base_estimator=SVC/DT/Perceptron/RL,n_estimator= cuantos voy a con
ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1),
    n_estimators=n_estimators,
    algorithm="SAMME",
    learning_rate=0.5,
    random_state=42,
)

ada_clf.fit(X_train, y_train)
plot_decision_boundary(ada_clf, X, y)
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
```


```
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_val_score, cross_val_predict

preds = cross_val_predict(ada_clf, X_train, y_train)
cm = confusion_matrix(y_train, preds)  # ,ada_clf.predict(X_train))
print(cm)
print(accuracy_score(y_train, preds))  # ada_clf.predict(X_test)))
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
```

```
[[170  19]
 [ 15 171]]
0.9093333333333333
```

Let's look at the individual estimators, their weights and loss function, computed as

$$\text{Loss} = \sum_i w_i \text{Loss}_i$$

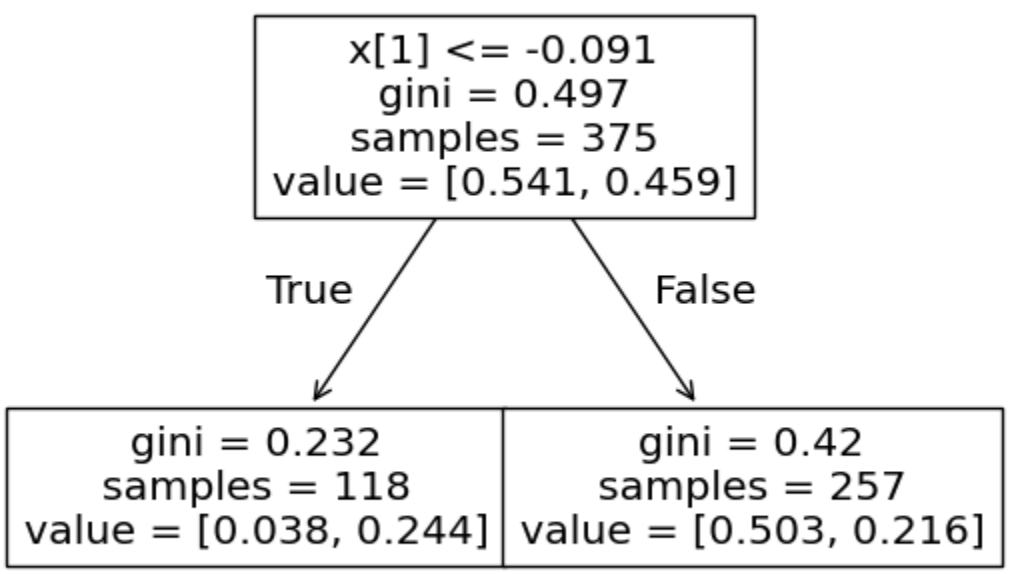
```
print(np.asarray(ada_clf.estimators_).shape)
```

```
(300,)
```

```
from sklearn.tree import plot_tree
```

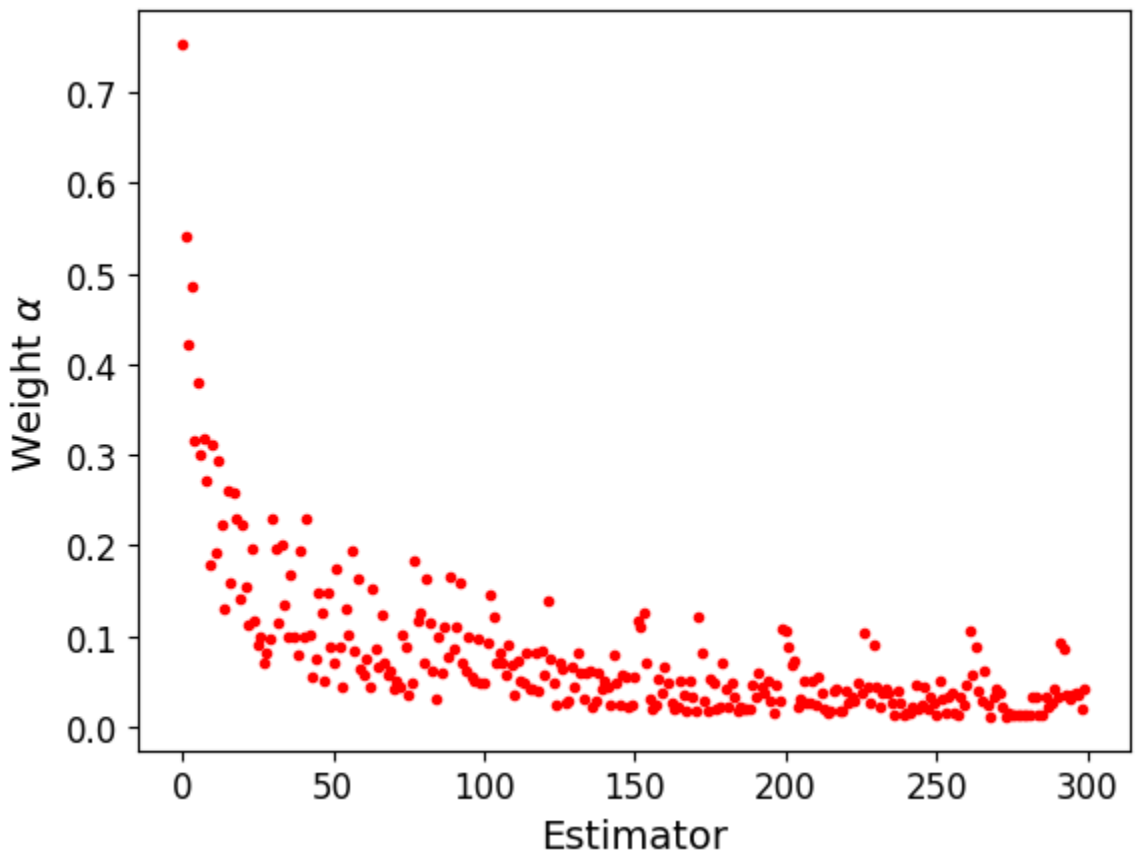
```
plot_tree(ada_clf.estimators_[1])
```

```
[Text(0.5, 0.75, 'x[1] <= -0.091\ngini = 0.497\nsamples = 375\nvalue = [0.541,
0.459]'),
 Text(0.25, 0.25, 'gini = 0.232\nsamples = 118\nvalue = [0.038, 0.244]'),
 Text(0.375, 0.5, 'True  '),
 Text(0.75, 0.25, 'gini = 0.42\nsamples = 257\nvalue = [0.503, 0.216]'),
 Text(0.625, 0.5, '  False')]
```



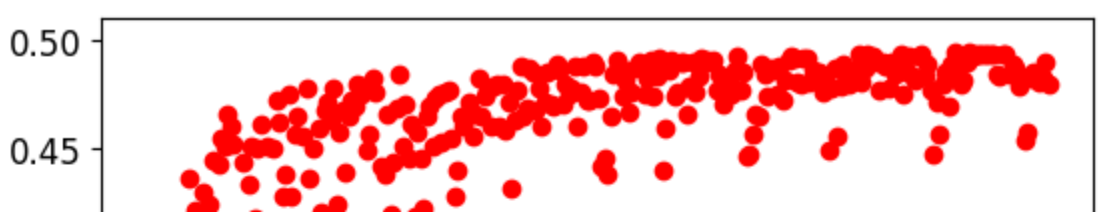
```
print(ada_clf.estimator_weights_.shape)
plt.plot(ada_clf.estimator_weights_, "r.")
plt.xlabel("Estimator")
plt.ylabel(r"Weight $\alpha$")
```

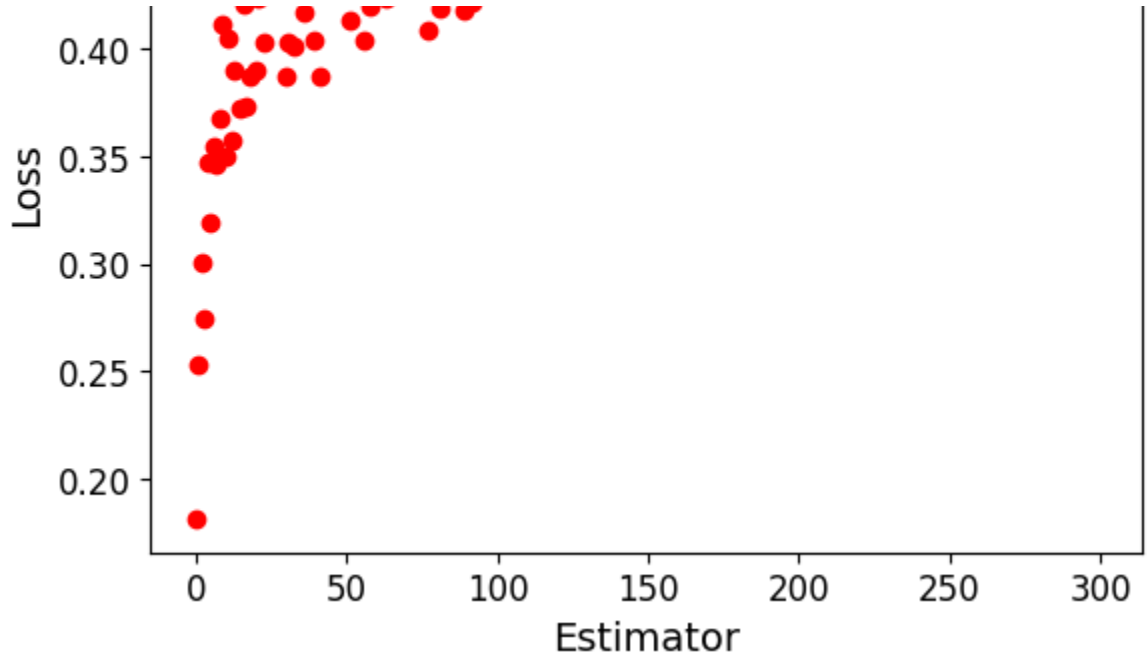
```
(300,)
Text(0, 0.5, 'Weight $\\alpha$')
```




```
print(ada_clf.estimator_errors_.shape)
plt.plot(ada_clf.estimator_errors_, "ro")
plt.xlabel("Estimator")
plt.ylabel("Loss")
```

```
(300,)
Text(0, 0.5, 'Loss')
```





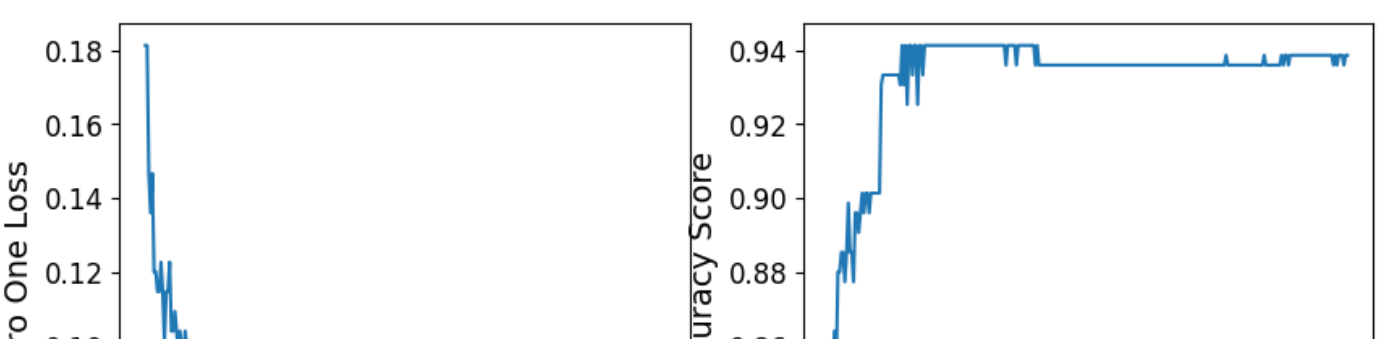
We can explore explicitly the evolution as we add estimators:

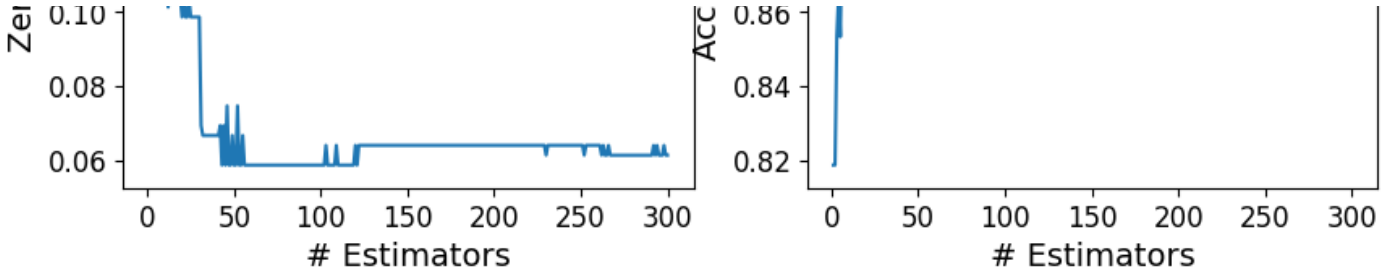
```
for nest, est_pred in enumerate(ada_clf.staged_predict(X_train[:2])):
    print(nest, est_pred[:2])
```

```
0 [1 1]
1 [1 1]
2 [1 0]
3 [1 1]
4 [1 0]
5 [1 1]
6 [1 1]
7 [1 1]
8 [1 1]
9 [1 0]
10 [1 1]
11 [1 1]
12 [1 1]
13 [1 1]
14 [1 0]
15 [1 1]
16 [1 1]
17 [1 0]
18 [1 1]
19 [1 0]
20 [1 1]
21 [1 0]
22 [1 0]
23 [1 1]
24 [1 0]
25 [1 0]
26 [1 0]
27 [1 0]
28 [1 0]
29 [1 0]
30 [1 0]
```

```
30 [1 0]
31 [1 0]
32 [1 0]
33 [1 0]
34 [1 0]
35 [1 0]
36 [1 0]
37 [1 0]
38 [1 0]
39 [1 0]
40 [1 0]
41 [1 0]
42 [1 0]
43 [1 0]
44 [1 0]
45 [1 1]
46 [1 0]
47 [1 0]
48 [1 0]
49 [1 0]
50 [1 0]
51 [1 1]
52 [1 0]
53 [1 0]
54 [1 0]
55 [1 0]
56 [1 0]
57 [1 0]
```

```
from sklearn.metrics import zero_one_loss  # counts the misclassified fraction

err_train = np.zeros((n_estimators, 2))
for i, y_pred in enumerate(ada_clf.staged_predict(X_train)):
    err_train[i, 0] = zero_one_loss(y_pred, y_train)
    err_train[i, 1] = accuracy_score(y_pred, y_train)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

ax[0].plot(np.arange(n_estimators) + 1, err_train[:, 0])
ax[1].plot(np.arange(n_estimators) + 1, err_train[:, 1])

ax[0].set_xlabel("# Estimators")
ax[1].set_xlabel("# Estimators")
ax[0].set_ylabel("Zero One Loss")
ax[1].set_ylabel("Accuracy Score")
plt.show()
```

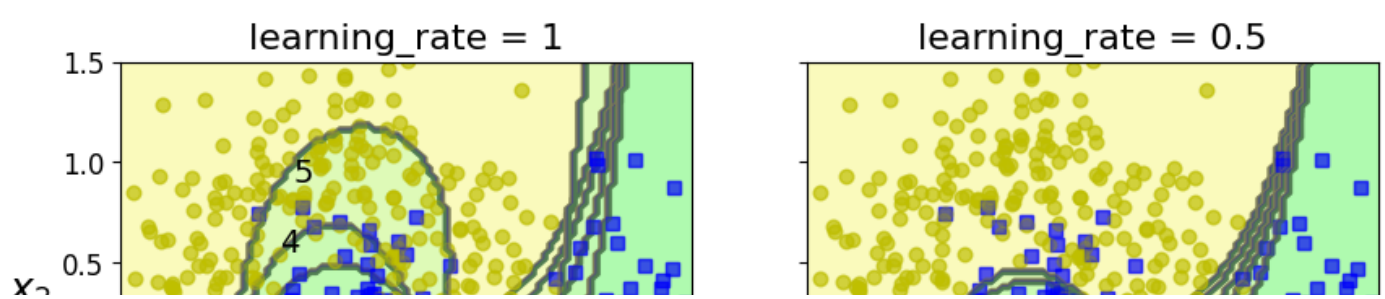


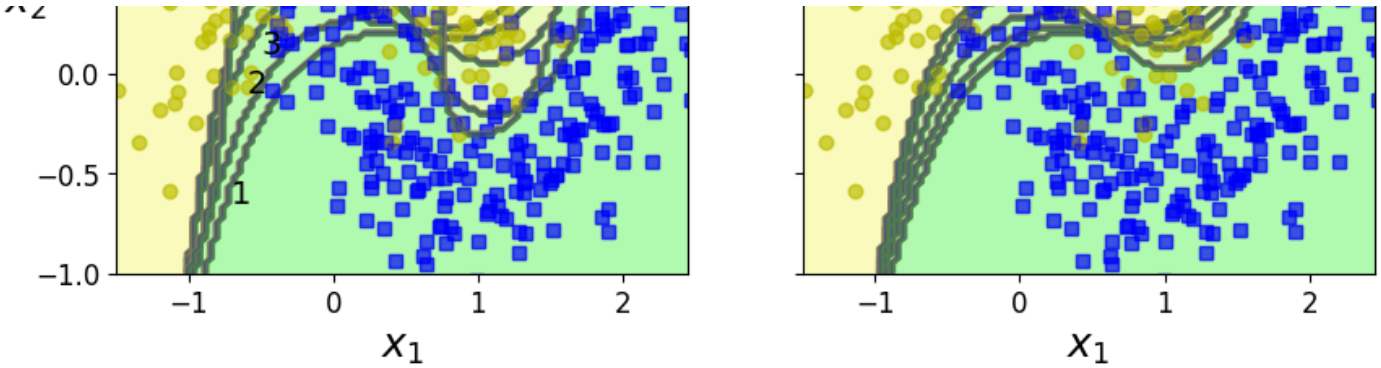


Learning rate effect in convergence

A nice example from Geron:

```
from sklearn.svm import SVC

m = len(X_train)

fix, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)
for subplot, learning_rate in ((0, 1), (1, 0.5)):
    sample_weights = np.ones(m)
    plt.sca(axes[subplot])
    for i in range(5):
        svm_clf = SVC(kernel="rbf", C=0.05, gamma="scale", random_state=42)
        svm_clf.fit(X_train, y_train, sample_weight=sample_weights)
        y_pred = svm_clf.predict(X_train)
        sample_weights[y_pred != y_train] *= 1 + learning_rate
        plot_decision_boundary(svm_clf, X, y, alpha=0.2)
        plt.title("learning_rate = {}".format(learning_rate), fontsize=16)
    if subplot == 0:
        plt.text(-0.7, -0.65, "1", fontsize=14)
        plt.text(-0.6, -0.10, "2", fontsize=14)
        plt.text(-0.5, 0.10, "3", fontsize=14)
        plt.text(-0.4, 0.55, "4", fontsize=14)
        plt.text(-0.3, 0.90, "5", fontsize=14)
    else:
        plt.ylabel("")

plt.show()
```



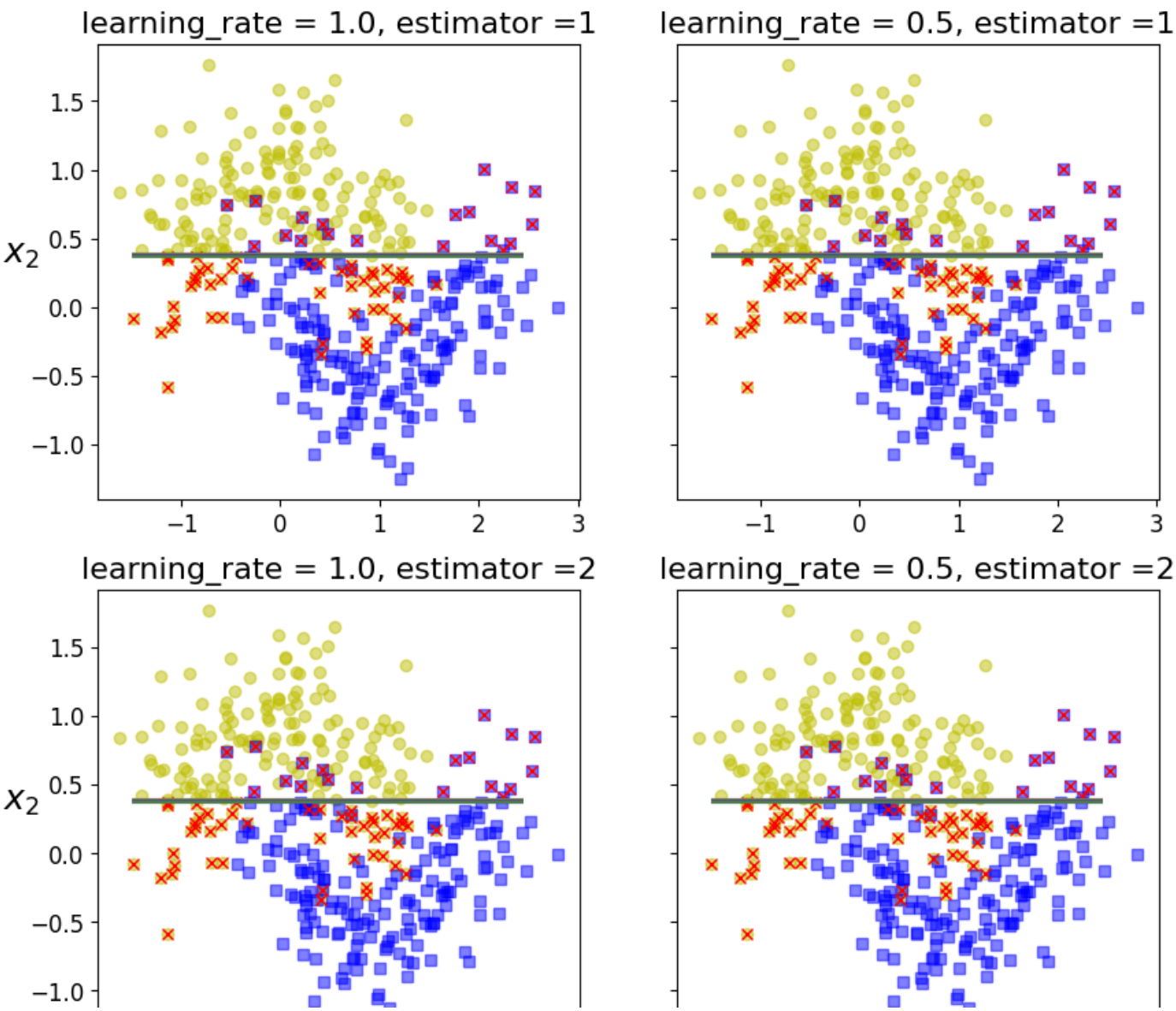


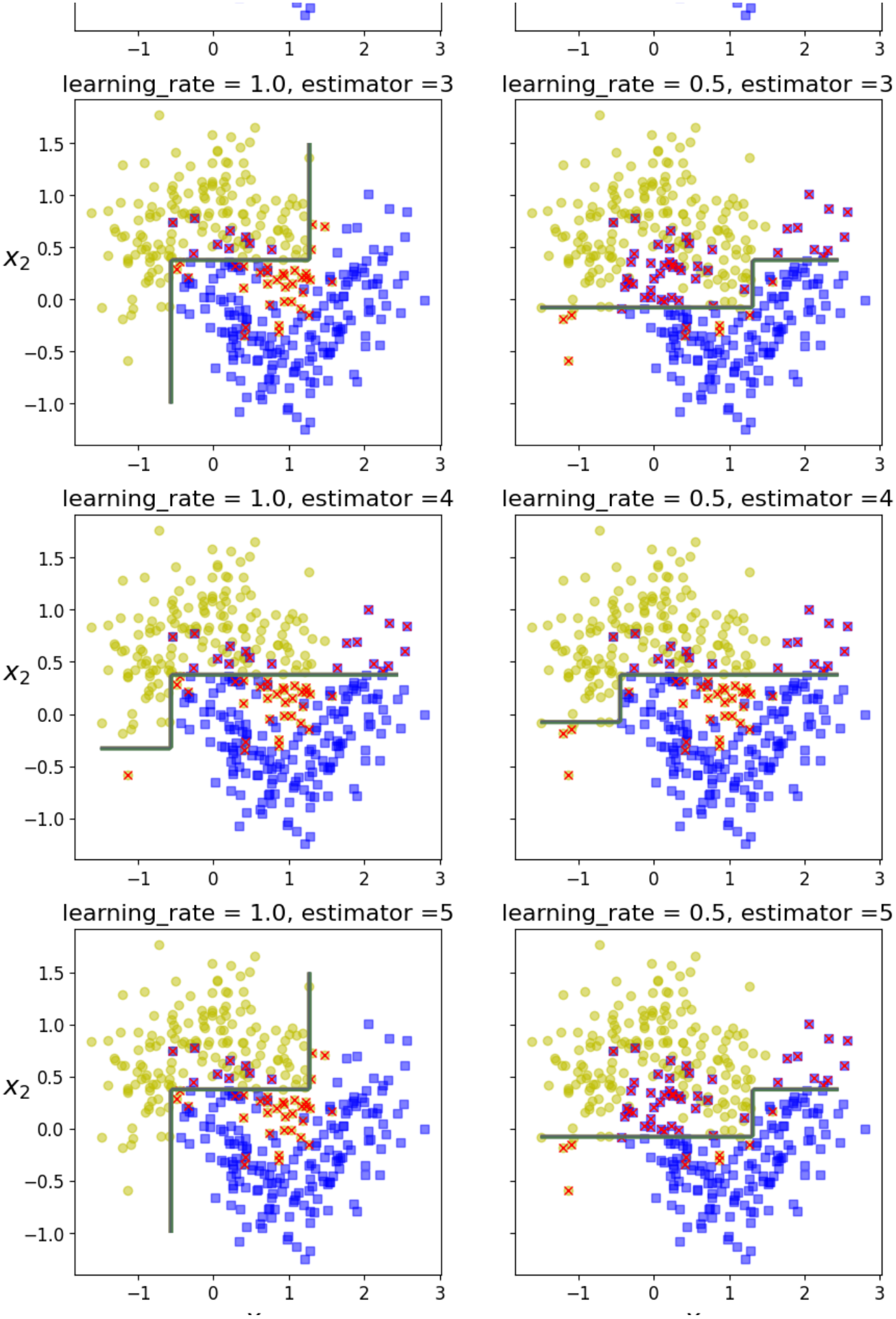
We can do this for Decision Trees and see the overall evolution

```
m = len(X_train)

learnings = [1.0, 0.5]
fix, axes = plt.subplots(
    nrows=5, ncols=len(learnings), figsize=(5 * len(learnings), 25), sharey=True
)
for subplot, learning_rate in enumerate(learnings):
    ada_clf = AdaBoostClassifier(
        DecisionTreeClassifier(max_depth=1),
        n_estimators=5,
        algorithm="SAMME",
        learning_rate=learning_rate,
        random_state=42,
    )
    ada_clf.fit(X_train, y_train)
    y_pred_train = np.zeros((5, X_train.shape[0]))
    for nest_train, est_dec_train in enumerate(ada_clf.staged_predict(X_train)):
        y_pred_train[nest_train] = est_dec_train
    # axes=[-1.5, 2.45, -1, 1.5]
    alpha = 0.5
    x1s = np.linspace(-1.5, 2.45, 100)
    x2s = np.linspace(-1, 1.5, 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    for nest, est_dec in enumerate(ada_clf.staged_predict(X_new)):
        y_pred_estimator_only = (
            ada_clf.estimators_[nest].predict(X_new).reshape(x1.shape)
        )
        y_pred = est_dec.reshape(x1.shape)
        custom_cmap2 = ListedColormap(["#7d7d58", "#4c4c7f", "#507d50"])
        axes[nest, subplot].plot(
            X_train[:, 0][y_train == 0], X_train[:, 1][y_train == 0], "yo", alpha=alpha
        )
        axes[nest, subplot].plot(
            X_train[:, 0][y_train == 1], X_train[:, 1][y_train == 1], "bs", alpha=alpha
```

```
            X_train[:, 0][y_train == 1], X_train[:, 1][y_train == 1], "bs", alpha=alpha
        )
        axes[nest, subplot].plot(
            X_train[:, 0][y_pred_train[nest] != y_train],
            X_train[:, 1][y_pred_train[nest] != y_train],
            "rx",
            alpha=1.0,
        )
        axes[nest, 0].set_ylabel(r"$x_2$", fontsize=18, rotation=0)
        axes[nest, subplot].contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
        # axes[nest,subplot].contour(x1, x2, y_pred_estimator_only, cmap='plasma', alph
        axes[nest, subplot].set_title(
            "learning_rate = {}, estimator ={}".format(learning_rate, nest + 1),
            fontsize=16,
        )
    #      plt.show()
    axes[-1, subplot].set_xlabel(r"$x_1$", fontsize=18)
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
```




learning_rate = 1.0, estimator =3
learning_rate = 0.5, estimator =3
learning_rate = 1.0, estimator =4
learning_rate = 0.5, estimator =4
learning_rate = 1.0, estimator =5
learning_rate = 0.5, estimator =5
x_2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3

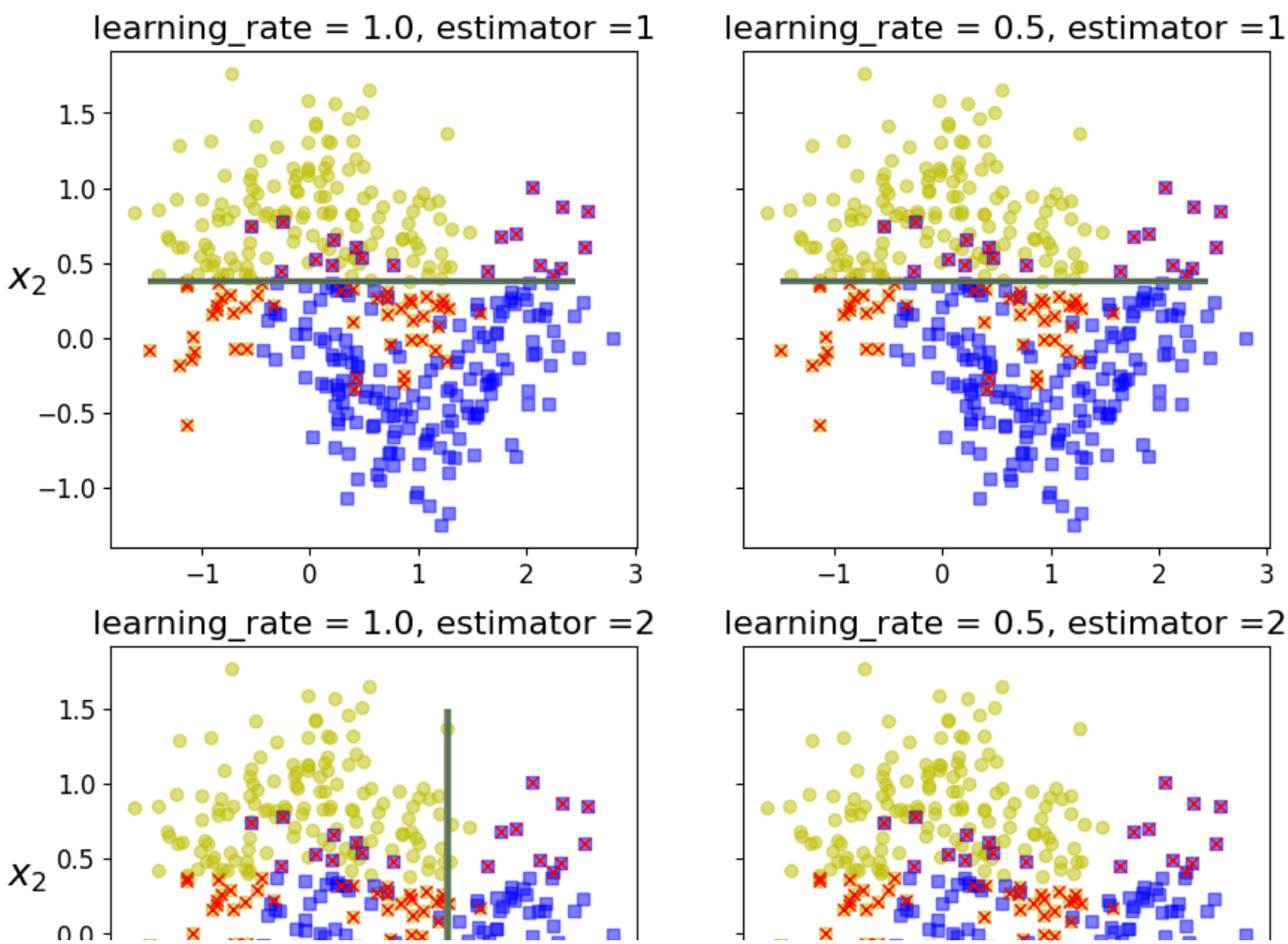
x_1 x_1

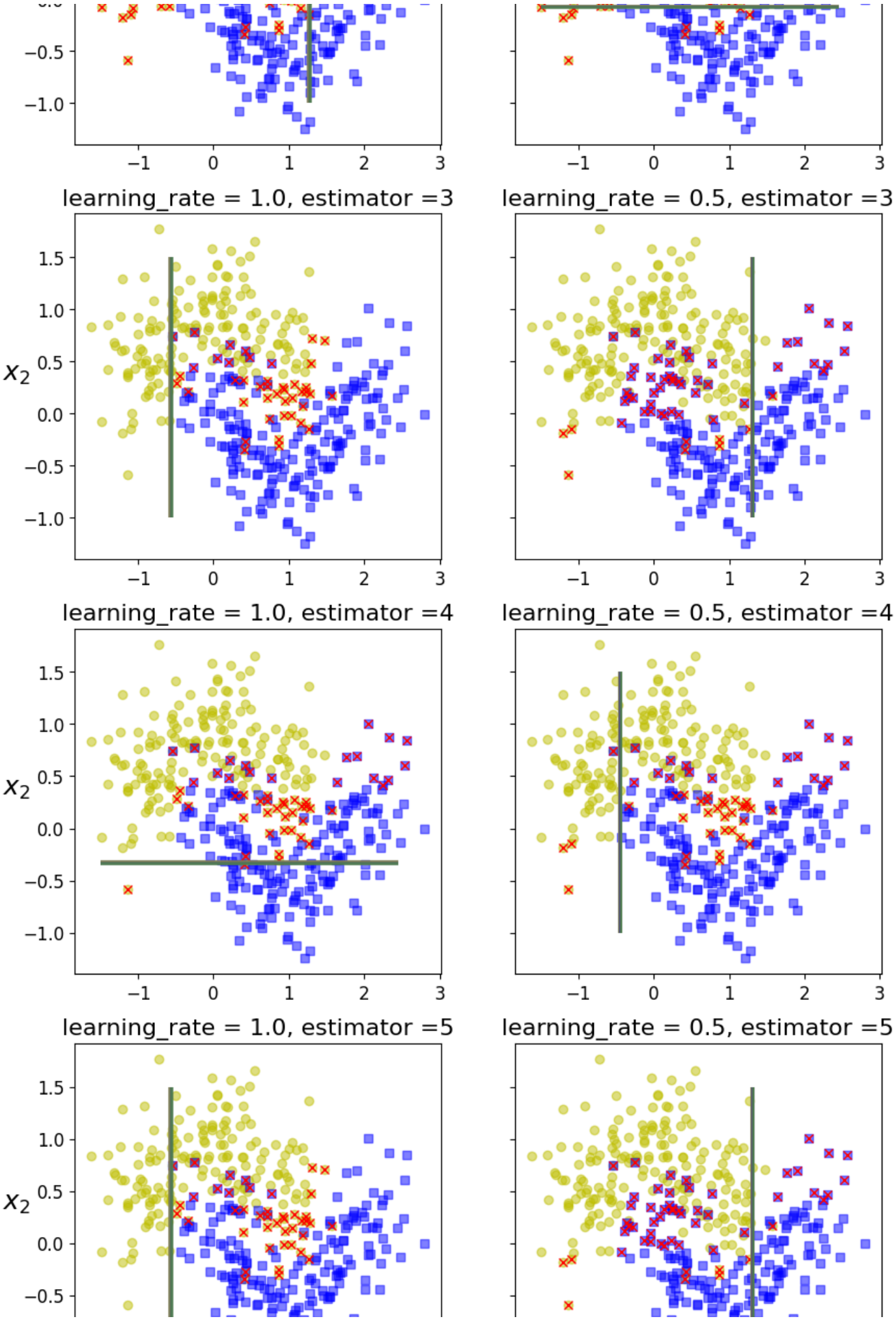
And the individual cuts

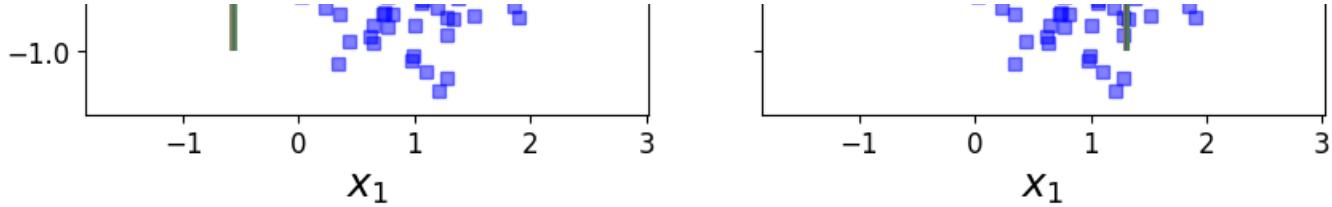
```
m = len(X_train)

learnings = [1.0, 0.5]
fix, axes = plt.subplots(
    nrows=5, ncols=len(learnings), figsize=(5 * len(learnings), 25), sharey=True
)
for subplot, learning_rate in enumerate(learnings):
    ada_clf = AdaBoostClassifier(
        DecisionTreeClassifier(max_depth=1),
        n_estimators=5,
        algorithm="SAMME",
        learning_rate=learning_rate,
        random_state=42,
    )
    ada_clf.fit(X_train, y_train)
    y_pred_train = np.zeros((5, X_train.shape[0]))
    for nest_train, est_dec_train in enumerate(ada_clf.staged_predict(X_train)):
        y_pred_train[nest_train] = est_dec_train
    # axes=[-1.5, 2.45, -1, 1.5]
    alpha = 0.5
    x1s = np.linspace(-1.5, 2.45, 100)
    x2s = np.linspace(-1, 1.5, 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    for nest, est_dec in enumerate(ada_clf.estimators_):
        y_pred = est_dec.predict(X_new).reshape(x1.shape)
```

```
        custom_cmap2 = ListedColormap(["#7d7d58", "#4c4c7f", "#507d50"])
        axes[nest, subplot].plot(
            X_train[:, 0][y_train == 0], X_train[:, 1][y_train == 0], "yo", alpha=alpha
        )
        axes[nest, subplot].plot(
            X_train[:, 0][y_train == 1], X_train[:, 1][y_train == 1], "bs", alpha=alpha
        )
        axes[nest, subplot].plot(
            X_train[:, 0][y_pred_train[nest] != y_train],
            X_train[:, 1][y_pred_train[nest] != y_train],
            "rx",
            alpha=1.0,
        )
        axes[nest, 0].set_ylabel(r"$x_2$", fontsize=18, rotation=0)
        axes[nest, subplot].contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
        axes[nest, subplot].set_title(
            "learning_rate = {}, estimator ={}".format(learning_rate, nest + 1),
            fontsize=16,
        )
    #      plt.show()
    axes[-1, subplot].set_xlabel(r"$x_1$", fontsize=18)
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/sklearn/ensemble/_weight_boosting.py:519: F
  warnings.warn(
```




learning_rate = 1.0, estimator =3
learning_rate = 0.5, estimator =3
learning_rate = 1.0, estimator =4
learning_rate = 0.5, estimator =4
learning_rate = 1.0, estimator =5
learning_rate = 0.5, estimator =5
x_2

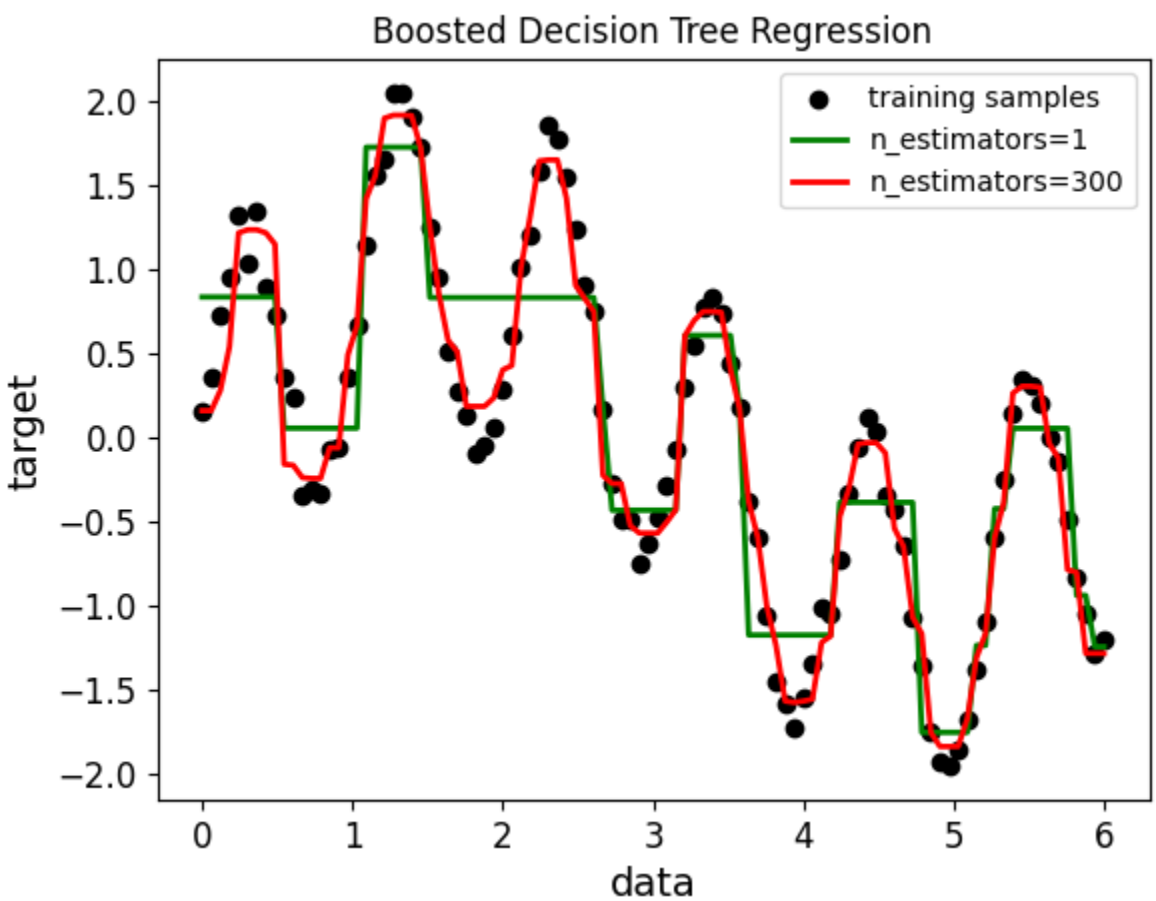


Regression example

This is an example of how to use `AdaBoostRegressor`. It's very similar, you only need to specify the `loss`.

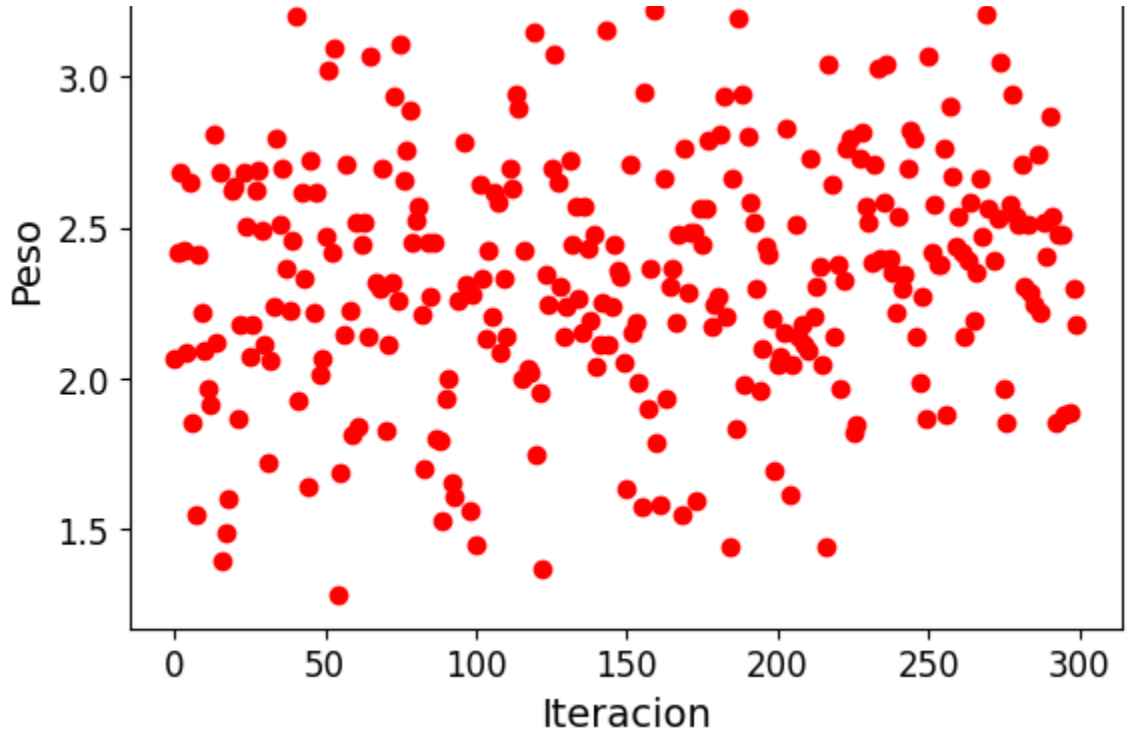
```
# Create the dataset
rng = np.random.RandomState(1)
X = np.linspace(0, 6, 100)[:, np.newaxis]
y = np.sin(X).ravel() + np.sin(6 * X).ravel() + rng.normal(0, 0.1, X.shape[0])

# Fit regression model
regr_1 = DecisionTreeRegressor(max_depth=4)

regr_2 = AdaBoostRegressor(
    DecisionTreeRegressor(max_depth=4),
    loss="square",
    n_estimators=300,
    random_state=rng,
)

regr_1.fit(X, y)
```

```
regr_2.fit(X, y)

# Predict
y_1 = regr_1.predict(X)
y_2 = regr_2.predict(X)

# Plot the results
plt.figure()
plt.scatter(X, y, c="k", label="training samples")
plt.plot(X, y_1, c="g", label="n_estimators=1", linewidth=2)
plt.plot(X, y_2, c="r", label="n_estimators=300", linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Boosted Decision Tree Regression")
plt.legend()
plt.show()
```



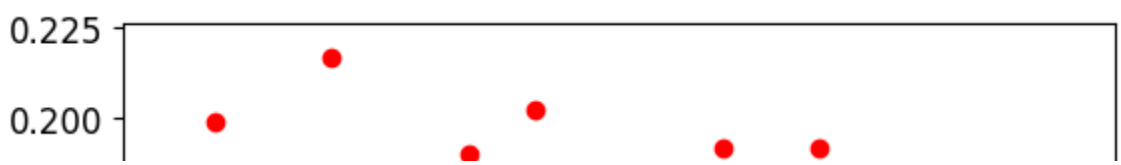

```
print(regr_2.estimator_weights_.shape)
plt.plot(regr_2.estimator_weights_, "ro")
plt.xlabel("Iteracion")
plt.ylabel("Peso")
```

```
(300,)
Text(0, 0.5, 'Peso')
```




```
print(regr_2.estimator_errors_.shape)
plt.plot(regr_2.estimator_errors_, "ro")
plt.xlabel("Iteracion")
plt.ylabel("Error")
```

```
(300,)
Text(0, 0.5, 'Error')
```



GradientBoosting

Gradient Boosting follows a different iterative procedure than AdaBoost.

Instead of correcting based on weights, Gradient Boosting improves by effectively training each estimator on the residuals of the previous estimators. For predictors h_m with $m = 1, \ldots, M$

$$\hat{y}_n^m = F_m(x_n)$$

with F built from the collection of all m estimators in an iterative way

$$F_m(x) = F_{m-1}(x) + h_m(x) = \sum_{m'=1}^{m} h_{m'}(x)$$

Thus, h_m is learned by optimizing

$$h_m = \arg\min \mathcal{L}_m(h) = \arg\min \sum_{} l(y_n, F_{m-1}(x_n) + h(x_n))$$

h h $\overline{n=1}$

which can be efficiently approximated via a linear expansion on h

$$l(y_n, F_{m-1}(x_n) + h(x_n)) \approx \mathcal{L}(y_n, F_{m-1}(x_n)) + h(x_n)\frac{\partial l(y_n, F(x_n))}{\partial F}|_{F=F_{m-1}}$$

and we have that

$$h_m = \arg\min_h \sum_{n=1}^{N} h(x_n)\frac{\partial l(y_n, F(x_n))}{\partial F}|_{F=F_{m-1}}$$

An example

From Geron, we can get a nice qualitative picture of how GradientBoosting works (it's not exactly this, but it's similar)

```
np.random.seed(42)
X = np.random.rand(100, 1) - 0.5
y = 3 * X[:, 0] ** 2 + 0.05 * np.random.randn(100)

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)

y2 = y - tree_reg1.predict(X)  # first estimator residuals
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg2.fit(X, y2)

y3 = y2 - tree_reg2.predict(X)  # second estimator residuals
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg3.fit(X, y3)
```

```
▾ DecisionTreeRegressor
DecisionTreeRegressor(max_depth=2, random_state=42)
```

And we can predict by aggregating the estimators

```
X_new = np.array([[0.8]])
y_pred = sum(
    tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3)
)  # sum of predictions from all trees
y_pred
```

```
array([0.75026781])
```

Let's plot this:

```
def plot_predictions(
    regressors, X, y, axes, label=None, style="r-", data_style="b.", data_label=None
):
    x1 = np.linspace(axes[0], axes[1], 500)
    y_pred = sum(regressor.predict(x1.reshape(-1, 1)) for regressor in regressors)
    plt.plot(X[:, 0], y, data_style, label=data_label)
    plt.plot(x1, y_pred, style, linewidth=2, label=label)
    if label or data_label:
        plt.legend(loc="upper center", fontsize=16)
    plt.axis(axes)


plt.figure(figsize=(11, 11))

plt.subplot(321)
plot_predictions(
    [tree_reg1],
    X,
    y,
    axes=[-0.5, 0.5, -0.1, 0.8],
    label="$h_1(x_1)$",
    style="g-",
    data_label="Training set",
)
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.title("Residuals and tree predictions", fontsize=16)

plt.subplot(322)
plot_predictions(
    [tree_reg1],
    X,
    y,
    axes=[-0.5, 0.5, -0.1, 0.8],
    label="$h(x_1) = h_1(x_1)$",
    data_label="Training set",
)
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.title("Ensemble predictions", fontsize=16)

plt.subplot(323)
plot_predictions(
    [tree_reg2],
    X,
    y2,
    axes=[-0.5, 0.5, -0.5, 0.5],
    label="$h_2(x_1)$",
    style="g-",
    data_style="k+",
    data_label="Residuals",
)
plt.ylabel("$y - h_1(x_1)$", fontsize=16)
```

```
plt.ylabel("$y - h_1(x_1)$", fontsize=16)

plt.subplot(324)
plot_predictions(
    [tree_reg1, tree_reg2],
    X,
    y,
    axes=[-0.5, 0.5, -0.1, 0.8],
    label="$h(x_1) = h_1(x_1) + h_2(x_1)$",
)
plt.ylabel("$y$", fontsize=16, rotation=0)

plt.subplot(325)
plot_predictions(
    [tree_reg3],
    X,
    y3,
    axes=[-0.5, 0.5, -0.5, 0.5],
    label="$h_3(x_1)$",
    style="g-",
    data_style="k+",
)
plt.ylabel("$y - h_1(x_1) - h_2(x_1)$", fontsize=16)
plt.xlabel("$x_1$", fontsize=16)

plt.subplot(326)
plot_predictions(
    [tree_reg1, tree_reg2, tree_reg3],
    X,
    y,
    axes=[-0.5, 0.5, -0.1, 0.8],
    label="$h(x_1) = h_1(x_1) + h_2(x_1) + h_3(x_1)$",
)
plt.xlabel("$x_1$", fontsize=16)
plt.ylabel("$y$", fontsize=16, rotation=0)

plt.show()
```

sklearn implementation

The two classes are GradientBoostingClassifier y GradientBoostingRegressor.

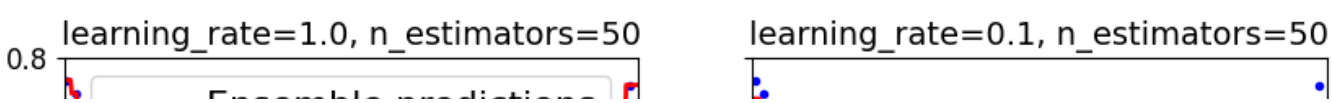
```
# GradientBoostingClassifier?
```

```
# GradientBoostingRegressor?
```

```
gbrt = GradientBoostingRegressor(
    max_depth=2, n_estimators=50, learning_rate=1.0, random_state=42
)
gbrt.fit(X, y)

gbrt_slow = GradientBoostingRegressor(
    max_depth=2, n_estimators=50, learning_rate=0.1, random_state=42
)
gbrt_slow.fit(X, y)
```

```
▾ GradientBoostingRegressor
GradientBoostingRegressor(max_depth=2, n_estimators=50, random_state=42)
```

```
fix, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)

plt.sca(axes[0])
plot_predictions(
    [gbrt], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="Ensemble predictions"
)
plt.title(
    "learning_rate={}, n_estimators={}".format(gbrt.learning_rate, gbrt.n_estimators),
    fontsize=14,
)
plt.xlabel("$x_1$", fontsize=16)
plt.ylabel("$y$", fontsize=16, rotation=0)

plt.sca(axes[1])
plot_predictions([gbrt_slow], X, y, axes=[-0.5, 0.5, -0.1, 0.8])
plt.title(
    "learning_rate={}, n_estimators={}".format(
        gbrt_slow.learning_rate, gbrt_slow.n_estimators
    ),
    fontsize=14,
)
plt.xlabel("$x_1$", fontsize=16)

plt.show()
```



Optimal number of trees

The number of estimators needs to be optimized. Too few, we underfit. Too many, we overfit. A nice way to find the optimal number of trees is by implementing an **early stopping** algorithm, which evaluates the predictor on a validation dataset to assess performance. Once the validation metric worsens, we can stop and get back to the best estimator.

Worsens, we can stop and get back to the best estimator.

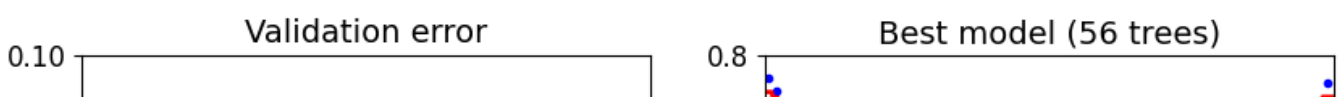
```
X_train, X_val, y_train, y_val = train_test_split(X, y, random_state=49)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120, random_state=42)
gbrt.fit(X_train, y_train)

errors = [
    np.sqrt(mean_squared_error(y_val, y_pred)) for y_pred in gbrt.staged_predict(X_val)
]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(
    max_depth=2, n_estimators=bst_n_estimators, random_state=42
)
gbrt_best.fit(X_train, y_train)
```

```
▾ GradientBoostingRegressor
GradientBoostingRegressor(max_depth=2, n_estimators=np.int64(56),
                          random_state=42)
```

```
print(bst_n_estimators)
```

```
56
```

```
min_error = np.min(errors)
plt.figure(figsize=(10, 4))

plt.subplot(121)
plt.plot(errors, "b.-")
plt.plot([bst_n_estimators, bst_n_estimators], [0, min_error], "k--")
plt.plot([0, 120], [min_error, min_error], "k--")
plt.plot(bst_n_estimators, min_error, "ko")
plt.text(bst_n_estimators, min_error * 1.2, "Minimum", ha="center", fontsize=14)
plt.axis([40, 120, 0, 0.1])
plt.xlabel("Number of trees")
plt.ylabel("Error", fontsize=16)
plt.title("Validation error", fontsize=14)

plt.subplot(122)
plot_predictions([gbrt_best], X, y, axes=[-0.5, 0.5, -0.1, 0.8])
plt.title("Best model (%d trees)" % bst_n_estimators, fontsize=14)
plt.ylabel("$y$", fontsize=16, rotation=0)
plt.xlabel("$x_1$", fontsize=16)

plt.show()
```



As shown, **early stopping** avoids overfitting (to a certain degree). However, in the code above we're still training all possible estimators. A realistic implementation usually stops once the metric worsens to avoid wasteful compute. We can do this through the `warm_start` option, which stores all trees trained during `fit`:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True, random_state=42)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break  # early stopping


print(gbrt.n_estimators)
print("Minimo MSE en el conjunto de validacion:", min_val_error)
```

```
61
Minimo MSE en el conjunto de validacion: 0.002712853325235463
```

```
gbrt = GradientBoostingRegressor(
    max_depth=2,
    n_estimators=120,
    warm_start=True,
    random_state=42,
    validation_fraction=0.2,
    n_iter_no_change=5,
)
gbrt.fit(X, y)
```

```
▾ GradientBoostingRegressor
GradientBoostingRegressor(max_depth=2, n_estimators=120, n_iter_no_change=5,
                          random_state=42, validation_fraction=0.2,
                          warm_start=True)
```

```
gbrt.n_estimators
```

```
120
```

Stochastic gradient boosting

There is an additional parameter called `subsample` which is very useful. It defines whether we use all possible data or if we consider a randomly chosen subset at each step, which usually accelerates training by lowering the variance of the estimator.

```
gbrt_all = GradientBoostingRegressor(
    max_depth=2, n_estimators=100, learning_rate=1.0, random_state=42
)
gbrt_all.fit(X, y)

gbrt_stochastic = GradientBoostingRegressor(
    max_depth=2, n_estimators=100, learning_rate=1.0, subsample=0.5, random_state=42
)
gbrt_stochastic.fit(X, y)

fix, axes = plt.subplots(ncols=2, figsize=(10, 4), sharey=True)

plt.sca(axes[0])
plot_predictions(
    [gbrt_all], X, y, axes=[-0.5, 0.5, -0.1, 0.8], label="Ensemble predictions"
)
plt.title(
    "subsample={}, n_estimators={}".format(gbrt_all.subsample, gbrt_all.n_estimators),
    fontsize=14,
)
plt.xlabel("$x_1$", fontsize=16)
plt.ylabel("$y$", fontsize=16, rotation=0)

plt.sca(axes[1])
plot_predictions([gbrt_stochastic], X, y, axes=[-0.5, 0.5, -0.1, 0.8])
plt.title(
    "subsample={}, n_estimators={}".format(
        gbrt_stochastic.subsample, gbrt_stochastic.n_estimators
    ),
    fontsize=14,
)
plt.xlabel("$x_1$", fontsize=16)

plt.show()
```

XGBoost

Although useful, the `sklearn` implementation is not the most powerful available.

One possible choice is to use **Extreme Gradient Boosting**, or `XGBoost`, which is an optimized implementation that prioritizes speed, scalability and portability. It is hugely popular (as can be seen in Kaggle) and can be used in a similar manner as `sklearn` (by design, they can be used together fairly easily). In particular, the `XGBRegressor` and `XGBClassifier` classes are built to be equivalent to `sklearn` models.

```
!pip install xgboost
```

```
Requirement already satisfied: xgboost in /usr/local/lib/python3.11/dist-packages (
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (fr
Requirement already satisfied: nvidia-nccl-cu12 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (fr
```

```
from xgboost import XGBRegressor, XGBClassifier
```

You can find the relevant document here.

The relevant hyperparameters for us are:

- `learning rate` (1 by default)
- `gamma` / `min_split_loss` (0 by default): the minimum loss reduction for the tree to continue splitting a leaf
- `max_depth` (6 by default)
- `min_child_weight` (1 by default): the minimum number of weighted measurements that must remain in a child when splitting a leaf node
- `subsample` (1 by default)
- `colsample_bytree`, `colsample_bylevel`, `colsample_bynode` (1 by default for all three): the fraction of features considered per tree, per level, and per node.
- `reg_lambda` (1 by default): L2 penalty factor in the weights
- `reg_alpha` (0 by default): L1 penalty factor in the weights
- `objective`: specifies the task to be performed. 'reg:squarederror' is the least squares loss. 'binary:logistic' or 'multi:softmax' are useful for classification with probabilistic outputs. There are several other options to play with.

```
# XGBRegressor?
```

```
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = pd.Series(iris.target)
```

```
print(X.head())
print(X.info())
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
X_train_2, X_val, y_train_2, y_val = train_test_split(X_train, y_train, test_size=0.2)
```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
dtypes: float64(4)
memory usage: 4.8 KB
None
```

```
regressor = XGBRegressor(
    n_estimators=200,
    learning_rate=0.5,
    reg_lambda=0.0,
    reg_alpha=0.0,
    gamma=0.0,
    eval_metric="rmse",
    early_stopping_rounds=5,
    objective="reg:squarederror",
    max_depth=3,
)
```

We can train using `fit`, with some hyperparameters set

```
regressor.fit(
    X_train_2,
    y_train_2,
    eval_set=[(X_train_2, y_train_2), (X_val, y_val)],
    verbose=True,
)
```

```
[0]	validation_0-rmse:0.41750	validation_1-rmse:0.48108
[1]	validation_0-rmse:0.23209	validation_1-rmse:0.28236
[2]	validation_0-rmse:0.14480	validation_1-rmse:0.22455
[3]	validation_0-rmse:0.11200	validation_1-rmse:0.21325
[4]	validation_0-rmse:0.09638	validation_1-rmse:0.21196
[5]	validation_0-rmse:0.05676	validation_1-rmse:0.20931
[6]	validation_0-rmse:0.05368	validation_1-rmse:0.21100
[7]	validation_0-rmse:0.04585	validation_1-rmse:0.21237
[8]	validation_0-rmse:0.04282	validation_1-rmse:0.21468
```

```
[8]	validation_0-rmse:0.04283	validation_1-rmse:0.21468
[9]	validation_0-rmse:0.03988	validation_1-rmse:0.21483
```

```
▾ XGBRegressor
XGBRegressor(base_score=None, booster=None, callbacks=None,
             colsample_bylevel=None, colsample_bynode=None,
             colsample_bytree=None, device=None, early_stopping_rounds=5,
             enable_categorical=False, eval_metric='rmse', feature_types=None,
             gamma=0.0, grow_policy=None, importance_type=None,
             interaction_constraints=None, learning_rate=0.5, max_bin=None,
             max_cat_threshold=None, max_cat_to_onehot=None,
             max_delta_step=None, max_depth=3, max_leaves=None,
             min_child_weight=None, missing=nan, monotone_constraints=None,
             multi_strategy=None, n_estimators=200, n_jobs=None,
             num_parallel_tree=None, random_state=None, ...)
```

```
np.sqrt(mean_squared_error(regressor.predict(X_val), y_val))
```

```
np.float64(0.20931475363275917)
```

```
regressor.evals_result()
```

```
{'validation_0': OrderedDict([('rmse',
                [0.4174991946545208,
                 0.23209394702056013,
                 0.14480418633898126,
                 0.11200466476246994,
                 0.09638490548979069,
                 0.0567608062634656,
                 0.0536785357829575,
                 0.04584860744834843,
                 0.04282747891721688,
                 0.03988297419510884,
                 0.03535937987079945])]),
 'validation_1': OrderedDict([('rmse',
                [0.48108314543177894,
                 0.28235983288233496,
                 0.22455271311342181,
                 0.21324631376570408,
                 0.21195560829026178,
                 0.20931474999626787,
                 0.21099609953838835,
                 0.2123663518622709,
                 0.21467686162262417,
                 0.21483295244600134,
                 0.2153588059669005])])}
```

We can explore feature importance

```
for i in range(len(iris.feature_names)):
    print((iris.feature_names[i], regressor.feature_importances_[i]))
```

```
('sepal length (cm)', np.float32(0.017386865))
```

```
('sepal length (cm)', np.float32(0.017386865))
('sepal width (cm)', np.float32(0.006139969))
('petal length (cm)', np.float32(0.89969236))
('petal width (cm)', np.float32(0.07678084))
```

Since it's so fast, we can even do `cross_val_score`.

```
# from sklearn.model_selection import cross_val_score

# scores = cross_val_score(
#    regressor, X_train, y_train, scoring="neg_root_mean_squared_error"
# )
# print(-scores.mean(), scores.std())
```

We can store the model

```
regressor.save_model("xbg_modelo_1.json")
```

```
params = regressor.get_xgb_params()
regressor_2 = XGBRegressor(**params)
regressor_2.get_xgb_params()
```

```
{'objective': 'reg:squarederror',
 'base_score': None,
 'booster': None,
 'colsample_bylevel': None,
 'colsample_bynode': None,
 'colsample_bytree': None,
 'device': None,
 'eval_metric': 'rmse',
 'gamma': 0.0,
 'grow_policy': None,
 'interaction_constraints': None,
 'learning_rate': 0.5,
 'max_bin': None,
 'max_cat_threshold': None,
 'max_cat_to_onehot': None,
 'max_delta_step': None,
 'max_depth': 3,
 'max_leaves': None,
 'min_child_weight': None,
 'monotone_constraints': None,
 'multi_strategy': None,
 'n_jobs': None,
 'num_parallel_tree': None,
 'random_state': None,
 'reg_alpha': 0.0,
 'reg_lambda': 0.0,
 'sampling_method': None,
 'scale_pos_weight': None,
 'subsample': None,
 'tree_method': None,
 'validate_parameters': None,
```

```
 'verbosity': None}
```

And load it

```
regressor_2.load_model("xbg_modelo_1.json")
regressor_2.get_xgb_params()
```

```
{'objective': 'reg:squarederror',
 'base_score': '1.0416666E0',
 'booster': 'gbtree',
 'colsample_bylevel': None,
 'colsample_bynode': None,
 'colsample_bytree': None,
 'device': None,
 'eval_metric': 'rmse',
 'gamma': 0.0,
 'grow_policy': None,
 'interaction_constraints': None,
 'learning_rate': 0.5,
 'max_bin': None,
 'max_cat_threshold': None,
 'max_cat_to_onehot': None,
 'max_delta_step': None,
 'max_depth': 3,
 'max_leaves': None,
 'min_child_weight': None,
 'monotone_constraints': None,
 'multi_strategy': None,
 'n_jobs': None,
 'num_parallel_tree': None,
 'random_state': None,
 'reg_alpha': 0.0,
 'reg_lambda': 0.0,
 'sampling_method': None,
 'scale_pos_weight': None,
 'subsample': None,
 'tree_method': None,
 'validate_parameters': None,
 'verbosity': None}
```

```
regressor.predict(X_train[:2])
```

```
array([1.9406105 , 0.02473785], dtype=float32)
```

```
regressor_2.predict(X_train[:2])
```

```
array([1.9406105 , 0.02473785], dtype=float32)
```

Exercise

At the LHC we can look for new particles. One possibility are W', which may decay to different final

At the LHC we can look for new particles. One possibility are W', which may decay to different final states. For example, a proton-proton collision may produce a very massive particle that decays to two jets, which we rank by transverse momentum P_T and call *leading* and *submladling* jets. Each of these jets is characterized by seven parameters: its invariant mass (M_j), its transverse momentum (P_T), its relativistic rapidity (Y), its azimuthal angle (ϕ), and three variables ($\tau_{21}, \tau_{31}, \tau_{32}$), which measure the substructure of each jet.

We have a dataset of 10000 simulated collisions where this new particle W', which we call *signal*, is actually produced, and another 10000 whose collisions does not result in the creation of this particle but instead originate from many SM model processes which constitute the irreducible *background* of the search.

The goal is to train a classifer based on the jets features to differentiate signal and background events. This classifier can be used as a tagger to select interesting events or even be used to build an optimal observable for statistical inference (based on the Neyman-Pearson lemma).

The following cells import the data and visualize them. Explore the dataset and train an optimized tagger using cross-validation. First train a classifier using either the leading or the sub-leading jets, then both. Get the feature importances and report all relevant metrics. Compare to a simpler classifier and decide whether BDT were worth it.

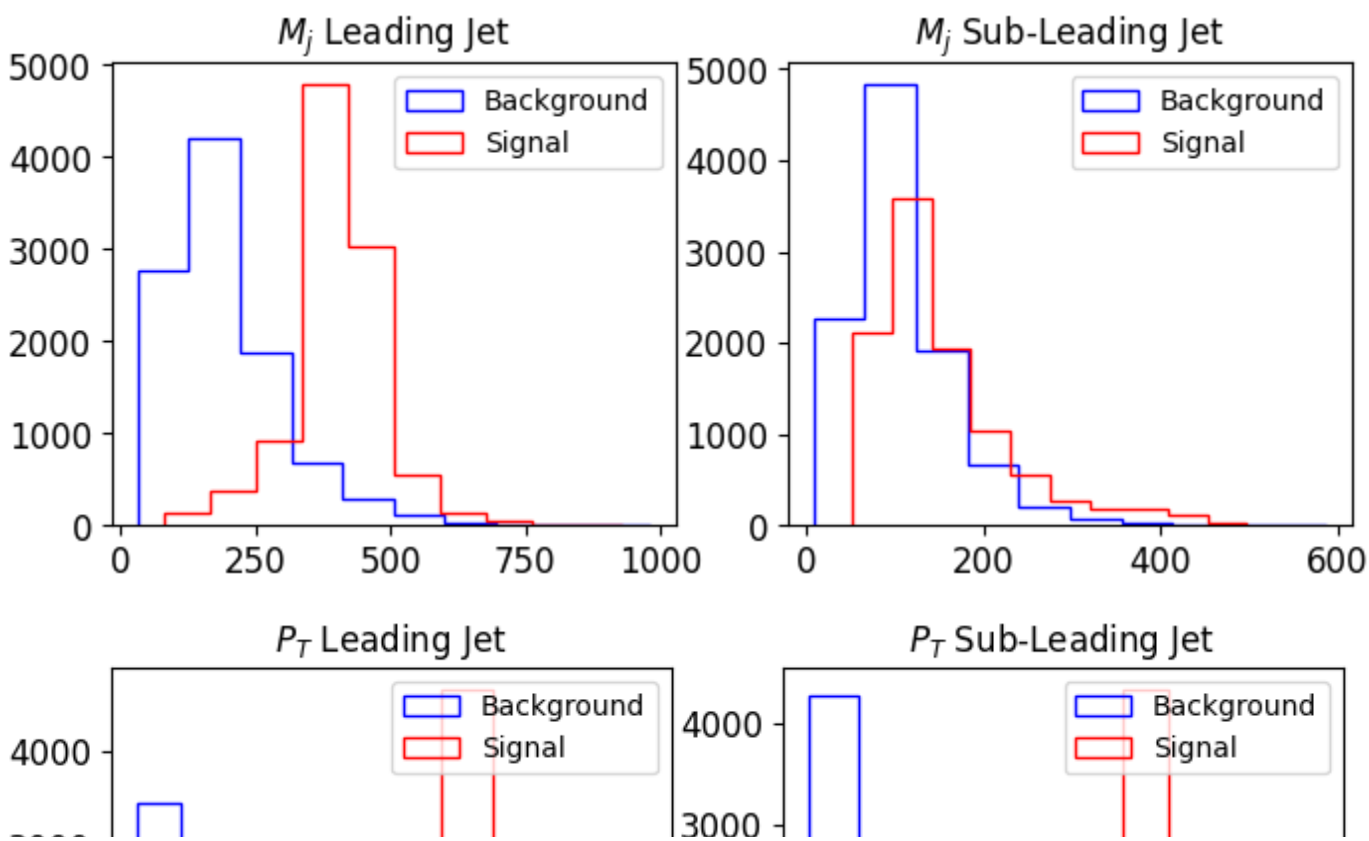
```
!wget -q -N https://gitlab.com/mcgen-ct/tutorials/-/raw/main/.full/ml/datasets/np_backg
!wget -q -N https://gitlab.com/mcgen-ct/tutorials/-/raw/main/.full/ml/datasets/np_signa
```

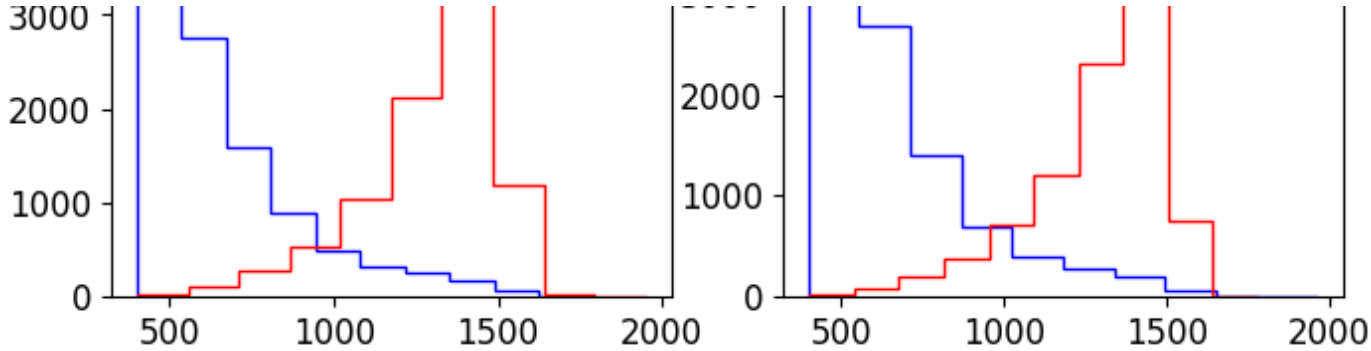
```
import numpy as np

background = []
# reads background events
with open("np_background.dat") as backgroundfile:
    for nline, line in enumerate(backgroundfile):
        if nline < 10000:
            Line = line.split(";")
            # separates the leading jet data from the sub-leading jet data, transforms
            # and constructs an array of dimensions [10000, 2, 7] (event, jet, feature)
            background_1 = list(map(lambda x: float(x), Line[0].split(",")))
            background_2 = list(map(lambda x: float(x), Line[1].split(",")))
            background.append([background_1, background_2])

background = np.asarray(background)

# Does the same for the signal data.
signal = []
with open("np_signal.dat") as signalfile:
    for nline, line in enumerate(signalfile):
        if nline < 10000:
            Line = line.split(";")
            signal_1 = list(map(lambda x: float(x), Line[0].split(",")))
            signal_2 = list(map(lambda x: float(x), Line[1].split(",")))
            signal.append([signal_1, signal_2])
```

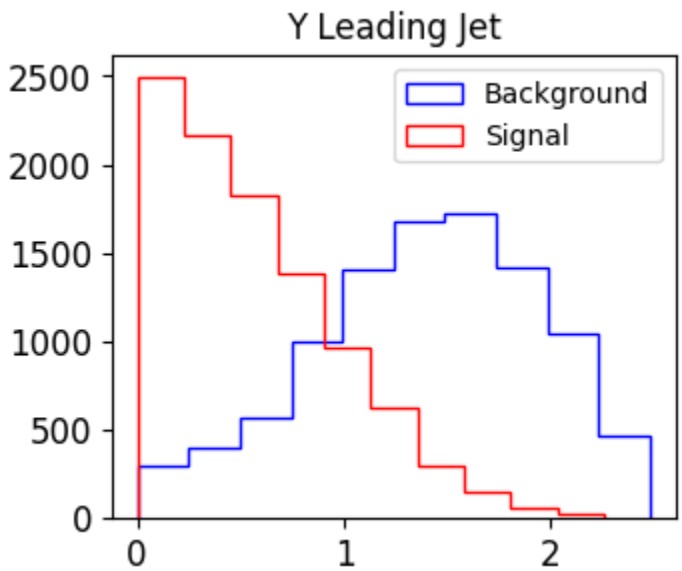
```
            signal.append([signal_1, signal_2])
signal = np.asarray(signal)

print("Shape of background and signal:", background.shape, signal.shape)

# group both datasets and assign labels, 0 for background and 1 for signal
X = np.vstack((background, signal))
Y = np.hstack((np.zeros(len(background)), np.ones(len(signal))))

print("Shapes of data and labels:", X.shape, Y.shape)
```

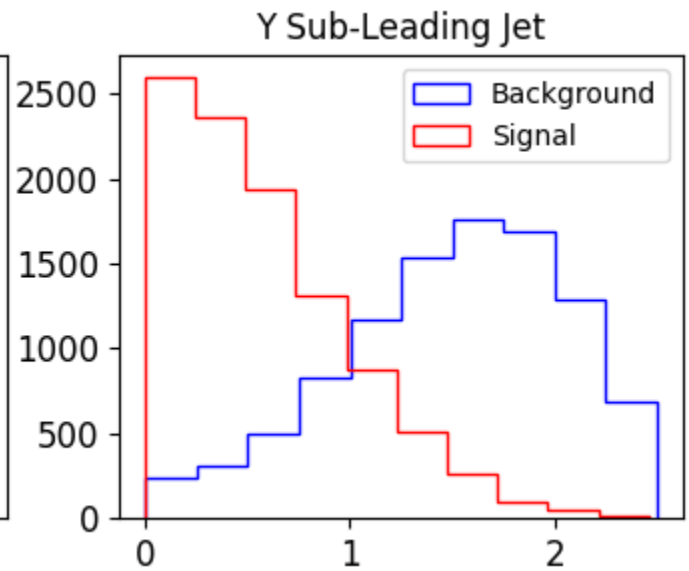
```
Shape of background and signal: (10000, 2, 7) (10000, 2, 7)
Shapes of data and labels: (20000, 2, 7) (20000,)
```

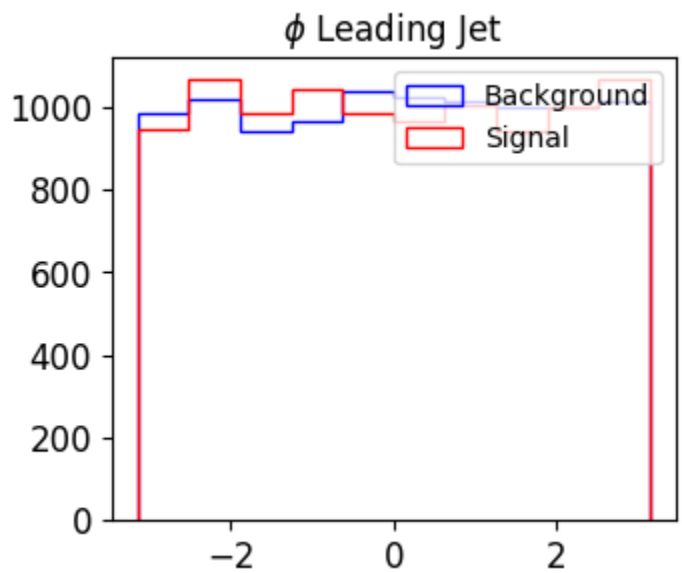
```
import matplotlib.pyplot as plt

vars = ["$M_j$", "$P_T$", "Y", "$\phi$", r"$\tau_{21}$", r"$\tau_{31}$", r"$\tau_{32}$"

# Let's plot the distributions of the variables for both leading and sub-leading jets a
for i in range(7):
    fig, axs = plt.subplots(1, 2, figsize=(8, 3))
    axs[0].hist(background[:, 0, i], histtype="step", color="blue", label="Background")
    axs[0].hist(signal[:, 0, i], histtype="step", color="red", label="Signal")
    axs[0].legend(loc="upper right")
    axs[0].set_title(vars[i] + " Leading Jet")
    axs[1].hist(background[:, 1, i], histtype="step", color="blue", label="Background")
    axs[1].hist(signal[:, 1, i], histtype="step", color="red", label="Signal")
    axs[1].legend(loc="upper right")
    axs[1].set_title(vars[i] + " Sub-Leading Jet")
    plt.show()
```

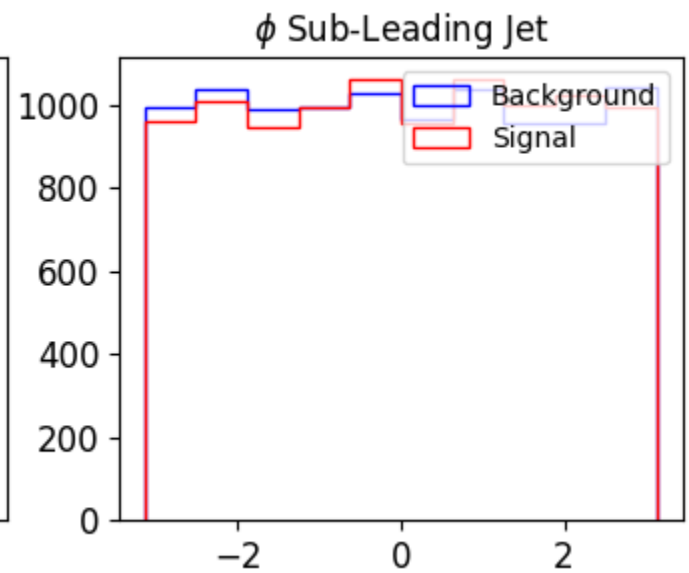


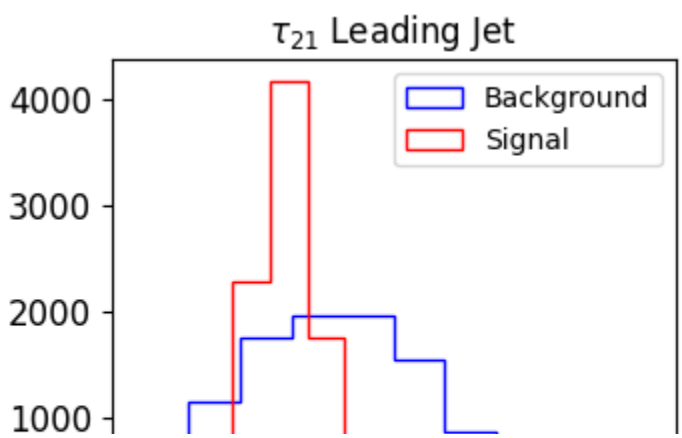

3000
2000
1000
0
500
1000
1500
2000
2000
1000
0
500
1000
1500
2000

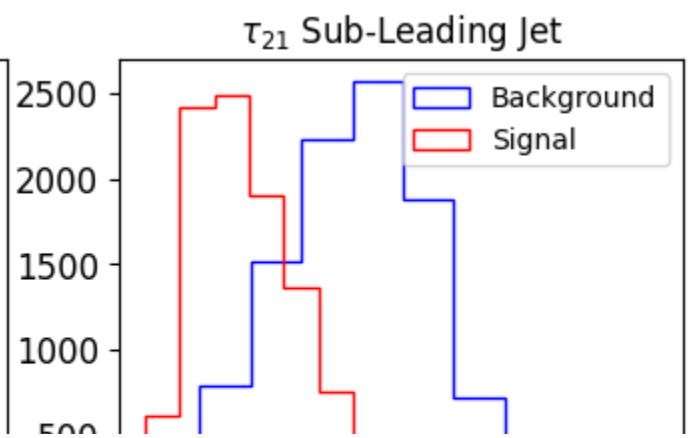

Y Leading Jet
Background
Signal
2500
2000
1500
1000
500
0
0
1
2

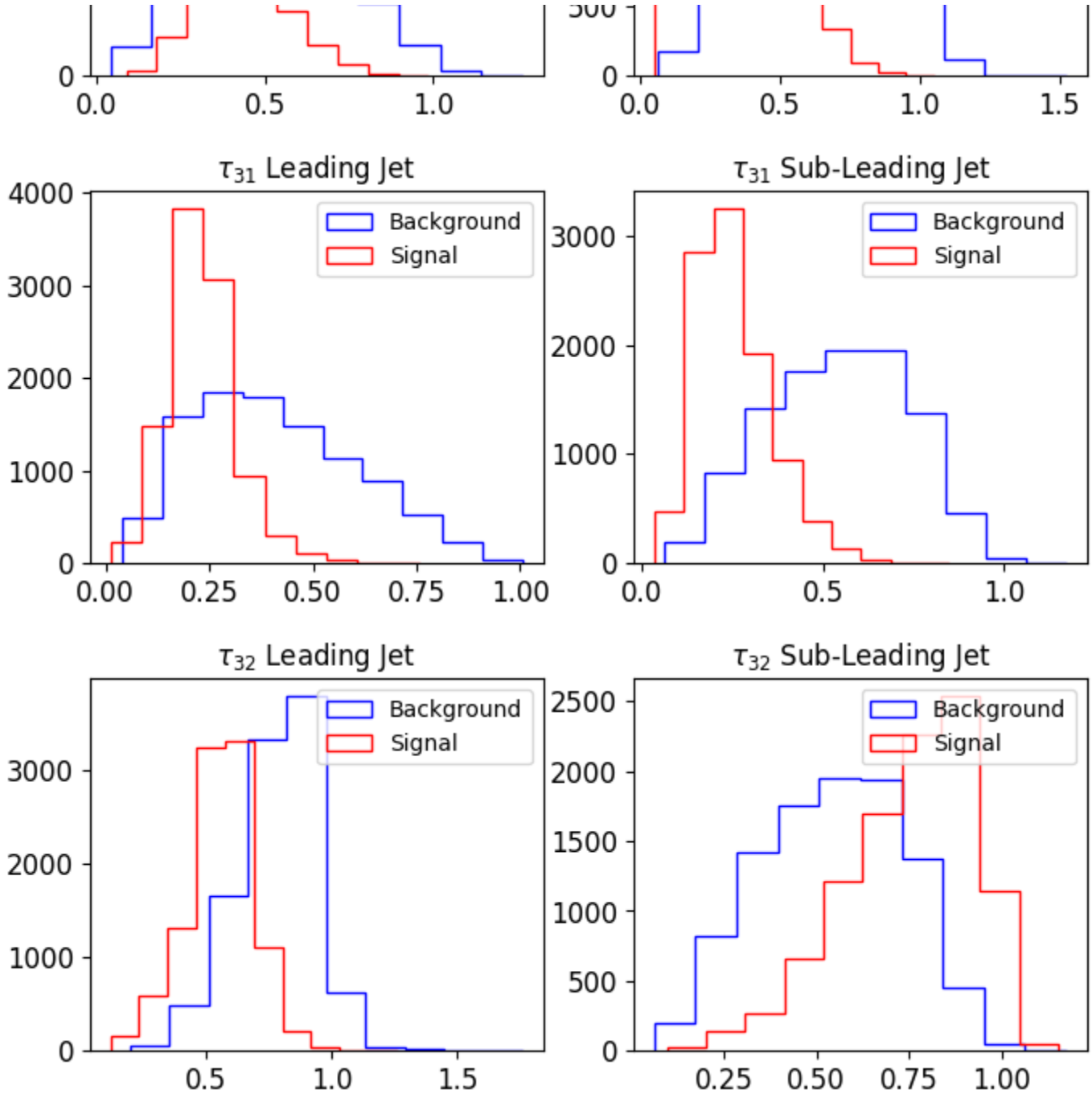

Y Sub-Leading Jet
Background
Signal
2500
2000
1500
1000
500
0
0
1
2


ϕ Leading Jet
Background
Signal
1000
800
600
400
200
0
−2
0
2


ϕ Sub-Leading Jet
Background
Signal
1000
800
600
400
200
0
−2
0
2


τ_{21} Leading Jet
Background
Signal
4000
3000
2000
1000


τ_{21} Sub-Leading Jet
Background
Signal
2500
2000
1500
1000




```
# Let's study the correlations between all other variables and the jet mass for both le
for i in range(6):
    fig, axs = plt.subplots(1, 4, figsize=(20, 3))
    f1 = axs[0].hist2d(background[:, 0, 0], background[:, 0, 1 + i], cmap="gist_heat_r"
    fig.colorbar(f1[3], ax=axs[0])
    axs[0].set_xlabel(vars[0])
    axs[0].set_ylabel(vars[1 + i])
    axs[0].set_title("Background Leading Jet")
    f2 = axs[1].hist2d(signal[:, 0, 0], signal[:, 0, 1 + i], cmap="gist_heat_r")
    fig.colorbar(f2[3], ax=axs[1])
    axs[1].set_xlabel(vars[0])
    # axs[1].set_ylabel(vars[1+i])
    axs[1].set_title("Signal Leading Jet")
    f3 = axs[2].hist2d(background[:, 1, 0], background[:, 1, 1 + i], cmap="gist_heat_r"
    fig.colorbar(f3[3], ax=axs[2])
```

```
axs[2].set_xlabel(vars[0])
axs[2].set_title("Background Sub-Leading Jet")
f4 = axs[3].hist2d(signal[:, 1, 0], signal[:, 1, 1 + i], cmap="gist_heat_r")
axs[3].set_xlabel(vars[0])
axs[3].set_title("Signal Sub-Leading Jet")
fig.colorbar(f4[3], ax=axs[3])
plt.show()
```