Introduction to Scikit-Learn (sklearn)

This notebook demonstrates some of the most useful functions of the beautiful Scikit-Learn library.

What we're going to cover:

1. An end-to-end Scikit-Learn workflow
2. Getting the data ready
3. Choose the right estimator/algorithm for our problems
4. Fit the model/algorithm and use it to make predictions on our data
5. Evaluating a model
6. Improve a model
7. Save and load a trained model
8. Putting it all together!

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

0. An end-to-end Scikit-learn workflow

In [2]:
```
# 1. Get the data ready
heart_disease = pd.read_csv("heart-disease.csv")
heart_disease
```

Out[2]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3	0
299	45	1	3	110	264	0	1	132	0	1.2	1	0	3	0
300	68	1	0	144	193	1	1	141	0	3.4	1	2	3	0
301	57	1	0	130	131	0	1	115	1	1.2	1	1	3	0
302	57	0	1	130	236	0	0	174	0	0.0	1	1	2	0

303 rows × 14 columns

In [3]:
```
# Create the features matrix
X = heart_disease.drop("target", axis=1)

# Create labels
y = heart_disease["target"]
```

In [4]:
```
# 2. Choose the right model and hyperparameters
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier()

# we'll keep the default hyperparameters for now
clf.get_params()
```

Out[4]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

In [5]:
```
# 3. Fit the model to the training data
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2) # 80% training, 20% test
```

In [6]:
```
clf.fit(X_train, y_train);
```

In [7]:
```
# make a prediction
y_preds = clf.predict(X_test)
y_preds
```

Out[7]:
```
array([0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0,
       0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0], dtype=int64)
```

In [8]:
```
# 4. Evaluate the model on the training data...
clf.score(X_train, y_train)

# its 100% accurate because during training the model
# had the chance to see the actual label to try to
# predict the outcome
```

Out[8]: 1.0

In [9]:
```
# ... and test data
clf.score(X_test, y_test)
```

Out[9]: 0.8688524590163934

In [10]:
```
# use some other evaluation metrics
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
```

In [11]:
```
print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support

           0       0.92      0.79      0.85        29
           1       0.83      0.94      0.88        32

    accuracy                           0.87        61
   macro avg       0.88      0.87      0.87        61
weighted avg       0.87      0.87      0.87        61
```

In [12]:
```
confusion_matrix(y_test, y_preds)
```

Out[12]:
```
array([[23,  6],
       [ 2, 30]], dtype=int64)
```

In [13]:
```
accuracy_score(y_test, y_preds)
```

Out[13]: 0.8688524590163934

In [14]:
```
# 5. Improve the model
# Try different amount of n_estimators

np.random.seed(42)
for i in range(10, 100, 10):
    print(f"Trying model with {i} estimators")
    clf = RandomForestClassifier(i).fit(X_train, y_train)

    print(f"Model accuracy on test set: {clf.score(X_test, y_test) * 100:.2f}%")
    print("")
```

```
Trying model with 10 estimators
Model accuracy on test set: 85.25%

Trying model with 20 estimators
Model accuracy on test set: 80.33%

Trying model with 30 estimators
Model accuracy on test set: 85.25%

Trying model with 40 estimators
Model accuracy on test set: 85.25%

Trying model with 50 estimators
Model accuracy on test set: 85.25%

Trying model with 60 estimators
Model accuracy on test set: 88.52%

Trying model with 70 estimators
Model accuracy on test set: 86.89%

Trying model with 80 estimators
Model accuracy on test set: 85.25%

Trying model with 90 estimators
Model accuracy on test set: 85.25%
```

In [15]:
```
# 6. Save a model and load it
import pickle

pickle.dump(clf, open("random_forest_m1.pkl", "wb"))
```

In [16]:
```
loaded_model = pickle.load(open("random_forest_m1.pkl", 'rb'))
loaded_model.score(X_test, y_test)
```

Out[16]: 0.8524590163934426

1. Getting the data ready

Three main things we have to do:

1. Split the data into features and labels (usually `X` and `y`)
   - then split them into train set and test set
2. Converting non-numerical values to numerical values (`feature encoding`)
3. Filling (`imputing`) or disregarding missing values

In [17]:
```
heart_disease.head()
```

Out[17]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

1.1 Split the data into features and labels

In [18]:
```
X = heart_disease.drop("target", axis=1)  # features
y = heart_disease["target"]  # labels
```

Split features and labels into training and test sets

In [19]:
```
# Split the data into training and test sets
from sklearn.model_selection import train_test_split

test_size = 0.2
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)
```

In [20]:
```
X_train.shape, X_test.shape, y_train.shape, y_test.shape
# X_train
#    242: 80% of the dataset
#    13: number of lables (columns)
```

Out[20]: ((242, 13), (61, 13), (242,), (61,))

1.2 Converting non-numerical values

For this, we need another dataset

In [21]:
```
car_sales = pd.read_csv("car-sales-extended.csv")
car_sales.head()
```

Out[21]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Honda	White	35431	4	15323
1	BMW	Blue	192714	5	19943
2	Honda	White	84714	4	28343
3	Toyota	White	154365	4	13434
4	Nissan	Blue	181577	3	14043

In [22]:
```
car_sales.dtypes
```

Out[22]:
```
Make             object
Colour           object
Odometer (KM)     int64
Doors             int64
Price             int64
dtype: object
```

In [23]:
```
# Split into X/y
X = car_sales.drop("Price", axis=1)
y = car_sales["Price"]

# Split into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

In [24]:
```
# Build machine learning model
from sklearn.ensemble import RandomForestRegressor

try:
    model = RandomForestRegressor()
    model.fit(X_train, y_train)
    model.score(X_test, y_test)
except ValueError as e:
    print(f"ValueError: {e}")
```

```
ValueError: could not convert string to float: 'Toyota'
```

ValueError: could not convert string to float

In [25]:
```
# Turn the categories into numbers
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

categorical_features = ["Make", "Colour",
                        "Doors"] # doors too, because each car falls into
                                 # a "category" of doors (3, 4, 5)
one_hot = OneHotEncoder()
transformer = ColumnTransformer([("one_hot",
                                  one_hot,
                                  categorical_features)],
                                remainder="passthrough")
transformed_X = transformer.fit_transform(X)
transformed_X
```

Out[25]:
```
array([[0.00000e+00, 1.00000e+00, 0.00000e+00, ..., 1.00000e+00,
        0.00000e+00, 3.54310e+04],
       [1.00000e+00, 0.00000e+00, 0.00000e+00, ..., 0.00000e+00,
        1.00000e+00, 1.92714e+05],
       [0.00000e+00, 1.00000e+00, 0.00000e+00, ..., 1.00000e+00,
        0.00000e+00, 8.47140e+04],
       ...,
       [0.00000e+00, 0.00000e+00, 1.00000e+00, ..., 1.00000e+00,
        0.00000e+00, 6.66040e+04],
       [0.00000e+00, 1.00000e+00, 0.00000e+00, ..., 1.00000e+00,
        0.00000e+00, 2.15883e+05],
       [0.00000e+00, 0.00000e+00, 0.00000e+00, ..., 1.00000e+00,
        0.00000e+00, 2.48360e+05]])
```

In [26]:
```
X.head()
```

Out[26]:

	Make	Colour	Odometer (KM)	Doors
0	Honda	White	35431	4
1	BMW	Blue	192714	5
2	Honda	White	84714	4
3	Toyota	White	154365	4
4	Nissan	Blue	181577	3

In [27]:
```
pd.DataFrame(transformed_X).head()
```

Out[27]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	35431.0
1	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	192714.0
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	84714.0
3	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	154365.0
4	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	181577.0

What happened here?

Using the **OneHotEncoder**, we've transformed all the *categorical* features into *numerical* features so that our Machine Learning Model can work with them.

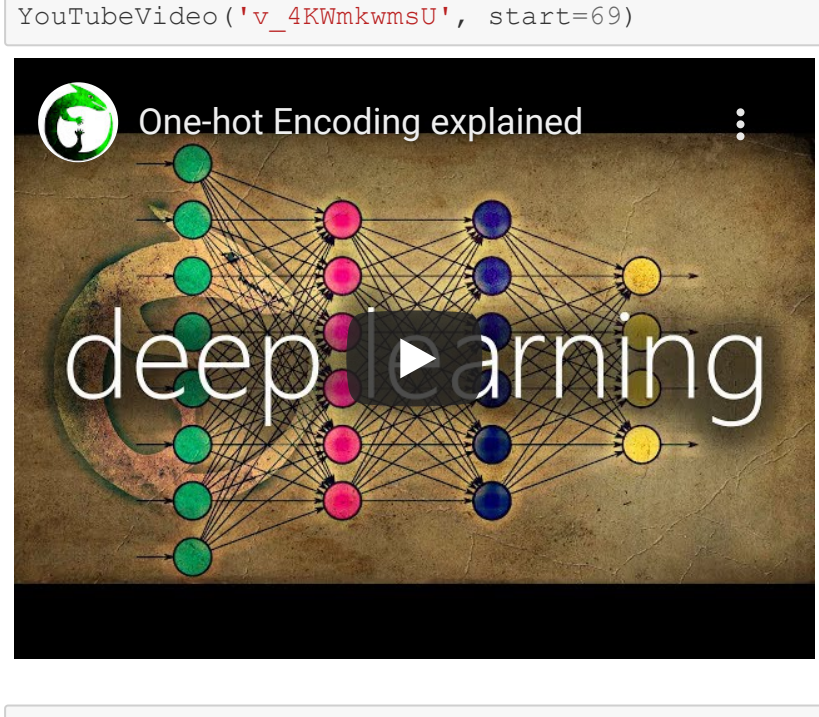
How did it do that?

Say that the feature `Make` has three possibilities: `Honda`, `Toyota` and `BMW`. The OneHotEncoder

1. transforms the feature `Make` into a vector of size 3 filled with zeros.
   - Say that the first zero represents `Honda`, the second `Toyota` and the third `BMW`
2. for each entry (row), it changes 0 into 1 in the slot that corresponds to the actual categorical feature.
   - Say that the entry was a `Toyota`, then the vector would be `[0,1,0]`

In [28]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('v_4KWmkwmsU', start=69)
```

Out[28]:



In [29]:
```
# a better view using a similar method from pandas
dummies = pd.get_dummies(X)
dummies.head()
```

Out[29]:

	Odometer (KM)	Doors	Make_BMW	Make_Honda	Make_Nissan	Make_Toyota	Colour_Black	Colour_Blue	Colour_Green	Colour_Red	Colo
0	35431	4	0	1	0	0	0	0	0	0	
1	192714	5	1	0	0	0	0	1	0	0	
2	84714	4	0	1	0	0	0	0	0	0	
3	154365	4	0	0	0	1	0	0	0	0	
4	181577	3	0	0	1	0	0	1	0	0	

In [30]:
```
# Let's refit the model
np.random.seed(42)  # needed if we want the same outcome every time we run this cell
X_train, X_test, y_train, y_test = train_test_split(transformed_X, y, test_size=0.2)

model.fit(X_train, y_train)
model.score(X_test, y_test)
```

Out[30]: 0.3235867221569877

1.3 Missing values

We have two options for handling missing data:

1. Fill the missing data (`imputation`)
2. Remove the samples with missing data

Let's import another dataset that has missing values.

In [31]:
```
car_sales_missing = pd.read_csv("car-sales-extended-missing-data.csv")
car_sales_missing.head()
```

Out[31]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Honda	White	35431.0	4.0	15323.0
1	BMW	Blue	192714.0	5.0	19943.0
2	Honda	White	84714.0	4.0	28343.0
3	Toyota	White	154365.0	4.0	13434.0
4	Nissan	Blue	181577.0	3.0	14043.0

From the head of the DataFrame, it doesn't seem like we have missing values.
We can calculate how many there are in this way.

In [32]:
```
car_sales_missing.isna().sum()
```

Out[32]:
```
Make             49
Colour           50
Odometer (KM)    50
Doors            50
Price            50
dtype: int64
```

Option 1: Fill missing data with Pandas

In [33]:
```
# Fill the "Make" column
car_sales_missing["Make"].fillna("missing", inplace=True)

# Fill the "Colour" column
car_sales_missing["Colour"].fillna("missing", inplace=True)

# Fill the "Odometer" column
car_sales_missing["Odometer (KM)"].fillna(car_sales_missing["Odometer (KM)"].mean(),
                                          inplace=True)

# Fill the "Doors" columns
car_sales_missing["Doors"].fillna(car_sales_missing["Doors"].mode().iat[0],
                                  inplace=True)
```

In [34]:
```
# Verify that our code worked
car_sales_missing.isna().sum()
```

Out[34]:
```
Make              0
Colour            0
Odometer (KM)     0
Doors             0
Price            50
dtype: int64
```

Option 2: Fill missing data with Scikit-learn

In [35]:
```
car_sales_missing2 = pd.read_csv("car-sales-extended-missing-data.csv")
car_sales_missing2.head()
```

Out[35]:

	Make	Colour	Odometer (KM)	Doors	Price
0	Honda	White	35431.0	4.0	15323.0
1	BMW	Blue	192714.0	5.0	19943.0
2	Honda	White	84714.0	4.0	28343.0
3	Toyota	White	154365.0	4.0	13434.0
4	Nissan	Blue	181577.0	3.0	14043.0

In [36]:
```
car_sales_missing2.isna().sum()
```

Out[36]:
```
Make             49
Colour           50
Odometer (KM)    50
Doors            50
Price            50
dtype: int64
```

In [37]:
```
# We are not able to train our model if some rows
# don't have the "Price" value (our label), so we
# just drop them
car_sales_missing2.dropna(subset=["Price"], inplace=True)
```

In [38]:
```
# Split into X & y
X = car_sales_missing2.drop("Price", axis=1)
y = car_sales_missing2["Price"]
```

In [39]:
```
# Fill missing values with Scikit-learn
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer

# Fill categorical values with "missing" and
# numerical values with mean
cat_imp = SimpleImputer(strategy="constant", fill_value="missing")
door_imp = SimpleImputer(strategy="constant", fill_value=4)
num_imp = SimpleImputer(strategy="mean")

# Define columns
cat_features = ["Make", "Colour"]
door_feature = ["Doors"]
num_features = ["Odometer (KM)"]

# Create an imputer (something that fills missing data)
imputer = ColumnTransformer([
    ("cat_imputer", cat_imp, cat_features),
    ("door_imputer", door_imp, door_feature),
    ("num_imputer", num_imp, num_features)
])

# Transform the data
filled_X = imputer.fit_transform(X)
filled_X
```

Out[39]:
```
array([['Honda', 'White', 4.0, 35431.0],
       ['BMW', 'Blue', 5.0, 192714.0],
       ['Honda', 'White', 4.0, 84714.0],
       ...,
       ['Nissan', 'Blue', 4.0, 66604.0],
       ['Honda', 'White', 4.0, 215883.0],
       ['Toyota', 'Blue', 4.0, 248360.0]], dtype=object)
```

In [40]:
```
pd.DataFrame(filled_X,
             columns=["Make", "Colour", "Odometer (KM)", "Doors"],
            ).isna().sum()
```

Out[40]:
```
Make             0
Colour           0
Odometer (KM)    0
Doors            0
dtype: int64
```

Option 3: Drop missing data

In [41]:
```
car_sales_missing.dropna(inplace=True)
```

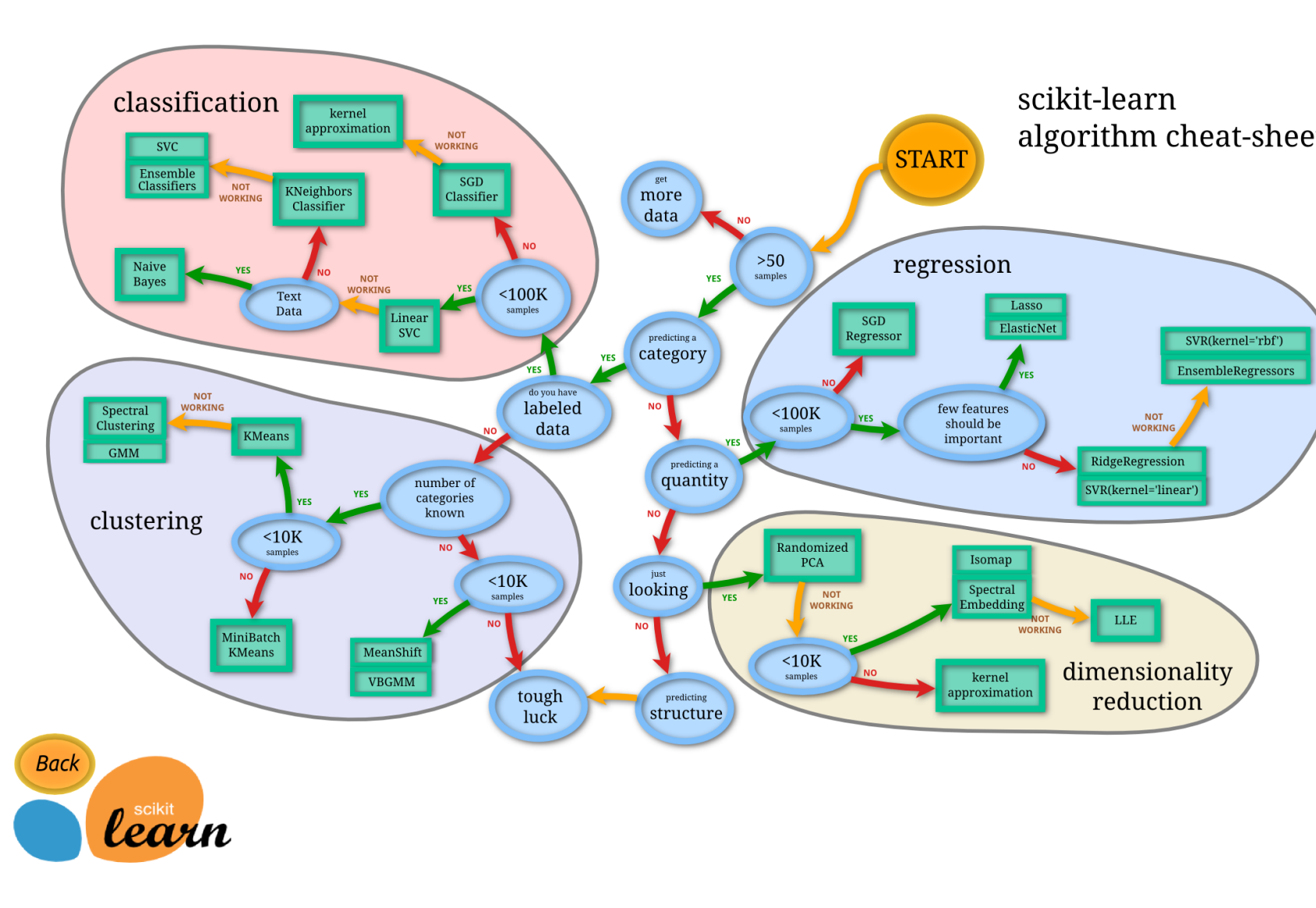
2. Choosing the right estimator/algorithm for our problem

Scikit-learn uses "estimator" as another temr for "machine learning model" or "algorithm".

There are at least two big kinds of algorithms:

- **Regression** - predicting a number
- **Classification** - predicting wether a sample is one thing or another (healthy, ill)

To help us choose the correct estimator, Scikit-learn has offered a flowchart to base upon.



In [42]:
```
# <img src="./sklearn-ml-map.png"></img>
```

2.1 Picking a machine learning model for a regression problem

In [43]:
```
# Import built-in Boston housing dataset
from sklearn.datasets import load_boston

boston_dict = load_boston()
```

In [44]:
```
boston_df = pd.DataFrame(boston_dict["data"],
                         columns=boston_dict["feature_names"])
boston_df["target"] = pd.Series(boston_dict["target"])
boston_df.head()
```

Out[44]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	target
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

In [45]:
```
# How many samples do we have?
len(boston_df)
```

Out[45]: 506

Following the Sklearn guidelines, we'll pick Ridge Regression model

In [46]:
```
from sklearn.linear_model import Ridge

# Setup numpy seed
seed = 46
np.random.seed(seed)

# Create the data
X = boston_df.drop("target", axis=1)
y = boston_df["target"]

# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Instantiate Ridge model
model = Ridge()
model.fit(X_train, y_train)

# Check the score on test data
model.score(X_test, y_test)
```

Out[46]: 0.794636752603225

This test worked quite fine. But what if Ridge wasn't working as expected?
And if we wanted to improve these results?
Scikit-learn suggests us another class of models: **EnsembleRegressor**.
The most famous and used is **RandomForest**

In [47]:
```
from sklearn.ensemble import RandomForestRegressor

# Setup numpy seed
seed = 46
np.random.seed(seed)

# Create the data
X = boston_df.drop("target", axis=1)
y = boston_df["target"]

# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Instantiate RandomForest
rf = RandomForestRegressor()
rf.fit(X_train, y_train)

# Check the score on test data
rf.score(X_test, y_test)
```

Out[47]: 0.8827430393207987

Using the same random seed, we've already improved our result by quite a margin!

2.2 Picking a machine learning model for a classification problem

In [48]:
```
heart_disease = pd.read_csv("heart-disease.csv")
heart_disease.head()
```

Out[48]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

In [49]:
```
len(heart_disease)
```

Out[49]: 303

Following the Sklearn guidelines, we'll pick LinearSVC

In [50]:
```
from sklearn.svm import LinearSVC

# Setup random seed
seed = 42
np.random.seed(seed)

# Make the data
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Istantiate LSVC
clf = LinearSVC()
clf.fit(X_train, y_train)

# Evaluate the LSVC
clf.score(X_test, y_test)
```

```
C:\Users\Utente\miniconda3\envs\ml\lib\site-packages\sklearn\svm\_base.py:976: ConvergenceWarning: Li
blinear failed to converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "
```

Out[50]: 0.8688524590163934

ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.

This warning tells us to increase the parameter `max_iter` (default 1000) when istantiating the LinearSVC class. We'll skip this for the moment.

Just as before, we're now interested in seeing another algorithm in action to see if we can improve our results.
Let's use **RandomForestClassifier**

In [51]:
```
from sklearn.ensemble import RandomForestClassifier

# Setup random seed
seed = 42
np.random.seed(seed)

# Make the data
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Istantiate RandomForestClassifier
clf = RandomForestClassifier()
clf.fit(X_train, y_train)

# Evaluate the RandomForestClassifier
clf.score(X_test, y_test)
```

Out[51]: 0.8524590163934426

In this case, they performed quite similar.

3. Fit the model on our data and use it to make predictions

3.1 Fit the model to the data

Different names for:

- X = features, feature variables, data
- y = labels, targets, target variables

In [52]:
```
from sklearn.ensemble import RandomForestClassifier

# Setup random seed
seed = 42
np.random.seed(seed)

# Make the data
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Istantiate RandomForestClassifier
clf = RandomForestClassifier()
```

In [53]:
```
clf.fit(X_train, y_train);
```

We've seen this line of code multiple times, but what does it *actually* do?
First, let's remind us how is our data structured.

In [54]:
```
X.head()
```
Out[54]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2

In [55]:
```
y.head()
```
Out[55]:
```
0    1
1    1
2    1
3    1
4    1
Name: target, dtype: int64
```

When we fit a model, it tries to deduce the patterns inside the features (data) that lead to the corresponding label (target).

3.2 Make predictions

Two ways to make predictions:

1. `predict()` function
2. `predict_proba()`

To predict a value, **the model needs an input with the exact same features and labels.**

3.2.1: `predict()`

In [56]:
```
try:
    clf.predict(np.array([1,4,2,6,2]))
except ValueError as e:
    print(f"ValueError: {e}")
```
```
ValueError: Expected 2D array, got 1D array instead:
array=[1. 4. 2. 6. 2.].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshap
e(1, -1) if it contains a single sample.
```

In [57]:
```
# since the model has never seen X_test
# let's use this as an example
clf.predict(X_test)
```
Out[57]:
```
array([0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0,
       1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0], dtype=int64)
```

Let's now compare the model predictions with the respective label

In [58]:
```
y_preds = clf.predict(X_test)
np.mean(y_preds == y_test)
```
Out[58]:
```
0.8524590163934426
```

This is the same as using

- `clf.score(X_test, y_test)`
- `sklearn.metrics.accuracy_score(y_test, y_preds)`

3.2.2: `predict_proba()`

Returns probabilities of a classification label.
... *What?*
Run the code.

In [59]:
```
clf.predict_proba(X_test[:5])
```
Out[59]:
```
array([[0.89, 0.11],
       [0.49, 0.51],
       [0.43, 0.57],
       [0.84, 0.16],
       [0.18, 0.82]])
```

In [60]:
```
# Let's predict on the same data
clf.predict(X_test[:5])
```
Out[60]:
```
array([0, 1, 1, 0, 1], dtype=int64)
```

`predict_proba()` gives us the probability of a classification label, meaning that the return value is expressed as the probability that the **n**th value is either 0 or 1, "black" or "white", "ill" or "healthy".

Let's take the first result: `[0.89, 0.11]`, meaning that it has been classified as

- **0** with 89% chance and
- **1** with 11% chance

4. Evaluating a model

Three ways to evaluate models/estimators:

1. Estimator `score()` method
2. Statistical estimators
3. The `scoring` parameter
4. Scikit-learn metric functions

In [61]:
```
from sklearn.ensemble import RandomForestClassifier

seed = 45
np.random.seed(seed)

X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

clf = RandomForestClassifier()
clf.fit(X_train, y_train);
```

4.1 Evaluating a model using the `score()` method

In [62]:
```
clf.score(X_test, y_test)
```
Out[62]:
```
0.8360655737704918
```

4.2 `-clf` Classification model evaluation metrics

1. Accuracy
2. Area under ROC curve
3. Confusion matrix
4. Classification report

In [63]:
```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

seed = 89
np.random.seed(seed)

X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf = RandomForestClassifier()
clf.fit(X_train, y_train);
```

4.2.1 Accuracy

In [64]:
```
np.mean(cross_val_score(clf, X, y, cv=5))
```
Out[64]:
```
0.8283060109289618
```

4.2.2 Area under ROC curve

ROC curves are a comparison of a model's **true positive rate (tpr)** versus a model's **false positive rate (fpr)**.

Because of this, we only truly care about what our model predicts to be positive (or 1).

In [65]:
```
from sklearn.metrics import roc_curve

# Make predictions with probabilities
y_probs = clf.predict_proba(X_test)

# Extract the probabilities of event "positive"
y_probs_positive = y_probs[:, 1]
y_probs_positive[10:]
```
Out[65]:
```
array([0.38, 0.26, 0.51, 0.22, 0.32, 0.98, 0.78, 0.2 , 0.67, 0.01, 0.76,
       0.07, 0.95, 0.67, 0.37, 0.93, 0.19, 0.17, 0.15, 0.92, 0.89, 0.37,
       0.56, 0.79, 0.93, 0.  , 0.65, 0.4 , 0.51, 0.52, 0.03, 0.85, 0.62,
       0.11, 0.72, 0.63, 0.14, 0.14, 0.44, 0.58, 1.  , 0.85, 0.08, 0.11,
       0.92, 0.81, 0.79, 0.48, 0.57, 0.51, 0.18])
```

In [66]:
```
# Calculate fpr, tpr and thresholds
fpr, tpr, thresholds = roc_curve(y_test, y_probs_positive)
```

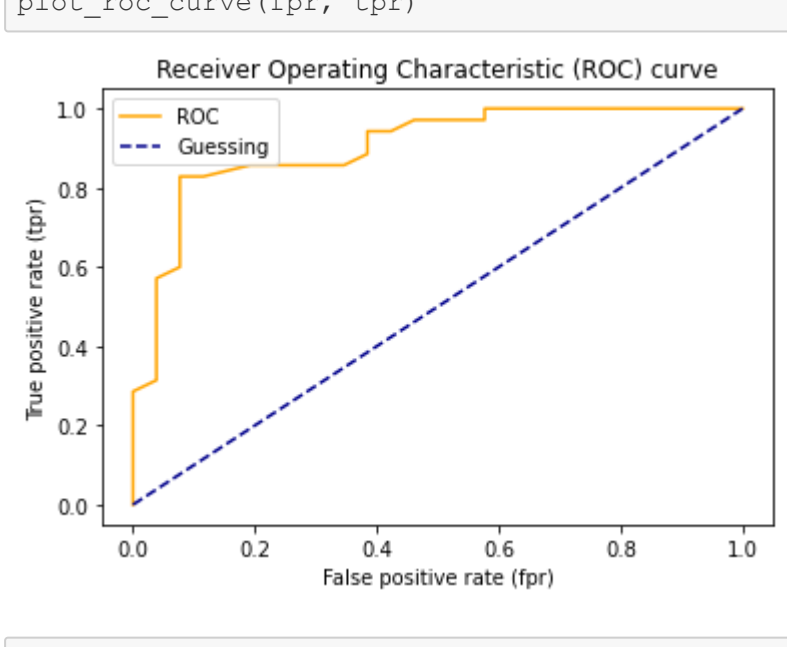
In [67]:
```
# Create a function to plot ROC curves
import matplotlib.pyplot as plt

def plot_roc_curve(fpr, tpr):
    # plot roc curve
    plt.plot(fpr, tpr, color="orange", label="ROC")
    # plot line with no predictive power (baseline)
    plt.plot([0,1], [0,1], color="darkblue", linestyle="--", label="Guessing")

    plt.xlabel("False positive rate (fpr)")
    plt.ylabel("True positive rate (tpr)")
    plt.title("Receiver Operating Characteristic (ROC) curve")
    plt.legend()

    plt.show()

plot_roc_curve(fpr, tpr)
```



In [68]:
```
# Evaluate the Area Under the Curve (AUC) score
from sklearn.metrics import roc_auc_score

roc_auc_score(y_test, y_probs_positive)
```
Out[68]:
```
0.9043956043956044
```

4.2.3 Confusion matrix

A confusion matrix is a quick way to compare the labels a model predicts and the actual labels it was supposed to predict.

In essence, giving you and idea of where the model is getting confused.

In [69]:
```
from sklearn.metrics import confusion_matrix

# predictions on the test data
y_preds = clf.predict(X_test)
# creating the confusion matrix
conf_mat = confusion_matrix(y_test, y_preds)
conf_mat
```
Out[69]:
```
array([[21,  5],
       [ 5, 30]], dtype=int64)
```

In [70]:
```
# having a better view at the matrix
pd.crosstab(y_test, y_preds,
           rownames=["Actual label"],
           colnames=["Predicted labels"])
```
Out[70]:

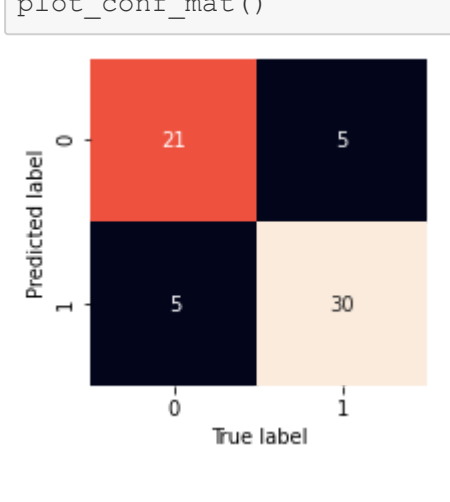
Predicted labels	0	1
Actual label		
0	21	5
1	5	30

In [71]:
```
# Having a colored view using Seaborn's heatmap()
import seaborn as sb

def plot_conf_mat():
    # Set the font scale
    # sb.set(font_scale=1.5)
    fig, ax = plt.subplots(figsize=(3,3))

    ax = sb.heatmap(conf_mat,
                    annot=True,  # adds annotations
                    cbar=False,  # removes the colorbar
                   )
    plt.xlabel("True label")
    plt.ylabel("Predicted label")

plot_conf_mat()
```



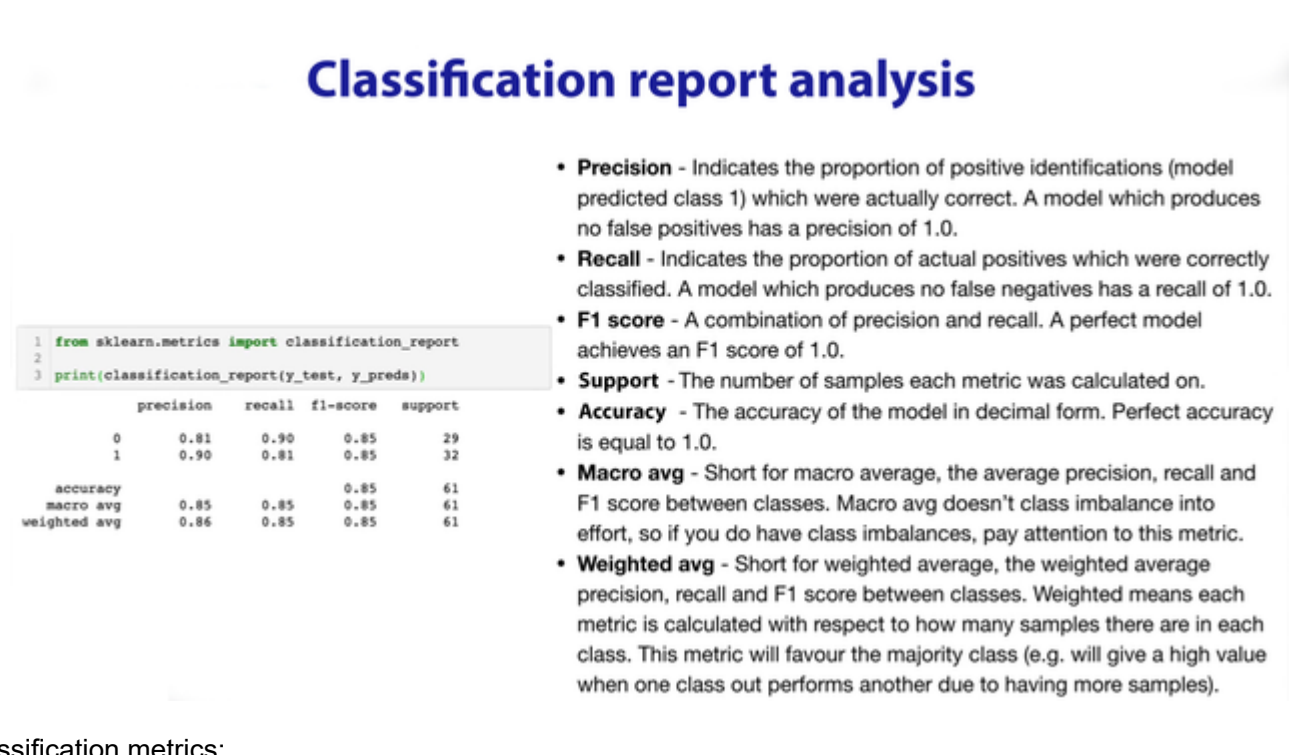
4.2.4 Classification report

In [72]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_preds))
```
```
              precision    recall  f1-score   support

           0       0.81      0.81      0.81        26
           1       0.86      0.86      0.86        35

    accuracy                           0.84        61
   macro avg       0.83      0.83      0.83        61
weighted avg       0.84      0.84      0.84        61
```



To summarize classification metrics:

- **Accuracy** is a good measure to start with if all classes are balanced (e.g. same amount of samples which are labelled with 0 or 1)
- **Precision** and **recall** become more important when classes are imbalanced
- If false positive predictions are worse than false negatives, aim for higher precision
- If false negative predictions are worse than false positives, aim for higher recall
- **F1-score** is a combination of precision and recall

4.2 `-reg` Regression model evaluation metrics

1. R^2 or coefficient of determination
2. Mean absolute error (MAE)
3. Mean squared error (MSE)

4.2.1 R^2

Compares your model predictions to the mean of the targets. Values from R^2 can range in $(-\infty, 1]$.
For example, if all your model does is predict the mean of the targets, its R^2 would be 0.
And if your model perfectly predicts a range of numbers, its R^2 value would be 1.

In [73]:
```
from sklearn.ensemble import RandomForestRegressor

seed = 562
np.random.seed(seed)

X = boston_df.drop("target", axis=1)
y = boston_df["target"]

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

model = RandomForestRegressor()
model.fit(X_train, y_train);
```

In [74]:
```
# This returns exactly that!
model.score(X_test, y_test)
```
Out[74]:
```
0.9125088209795402
```

Another way to calculate this is by using the `r2_score()` method

In [75]:
```
from sklearn.metrics import r2_score

# Fill an array with y_test mean
y_test_mean = np.full(len(y_test), y_test.mean())
```

In [76]:
```
r2_score(y_test, y_test_mean)
# score is 0 because it was tested on
# the mean of the test targets.
```
Out[76]:
```
0.0
```

In [77]:
```
r2_score(y_test, y_test)
# score is 1 because it perfectely
# predicted... itself.
```
Out[77]:
```
1.0
```

4.2.2 Mean absolute error (MAE)

Average of the absolute differences between predicted values and actual values.

In [78]:
```
from sklearn.metrics import mean_absolute_error

y_preds = model.predict(X_test)

mean_absolute_error(y_test, y_preds)
```
Out[78]:
```
2.0864117647058826
```

In [79]:
```
# Let's visualize the data first
mae_df = pd.DataFrame({
    "actual values": y_test,
    "predicted values": y_preds
})
mae_df.head(8)
```
Out[79]:

	actual values	predicted values
151	19.6	19.283
390	15.1	14.813
284	32.2	31.709
348	24.5	27.481
172	23.1	20.456
306	33.4	34.805
320	23.8	23.616
143	15.6	13.260

In [80]:
```
mae_df["differences"] = mae_df["predicted values"] - mae_df["actual values"]
mae_df.head(8)
```
Out[80]:

	actual values	predicted values	differences
151	19.6	19.283	-0.317
390	15.1	14.813	-0.287
284	32.2	31.709	-0.491
348	24.5	27.481	2.981
172	23.1	20.456	-2.644
306	33.4	34.805	1.405
320	23.8	23.616	-0.184
143	15.6	13.260	-2.340

In [81]:
```
# Mean Absolute Error manually calculated
np.mean(np.abs(mae_df["differences"]))
```
Out[81]:
```
2.0864117647058826
```

4.2.3 Mean squared error (MSE)

Average of the squared differences between predicted values and actual values.

In [82]:
```
from sklearn.metrics import mean_squared_error

mean_squared_error(y_test, y_preds)
```
Out[82]:
```
8.117253647058815
```

In [83]:
```
# Mean Squared Error manually calculated
np.mean(np.square(mae_df["differences"]))
```
Out[83]:
```
8.117253647058815
```

4.3 `-clf` Using the `scoring` parameter

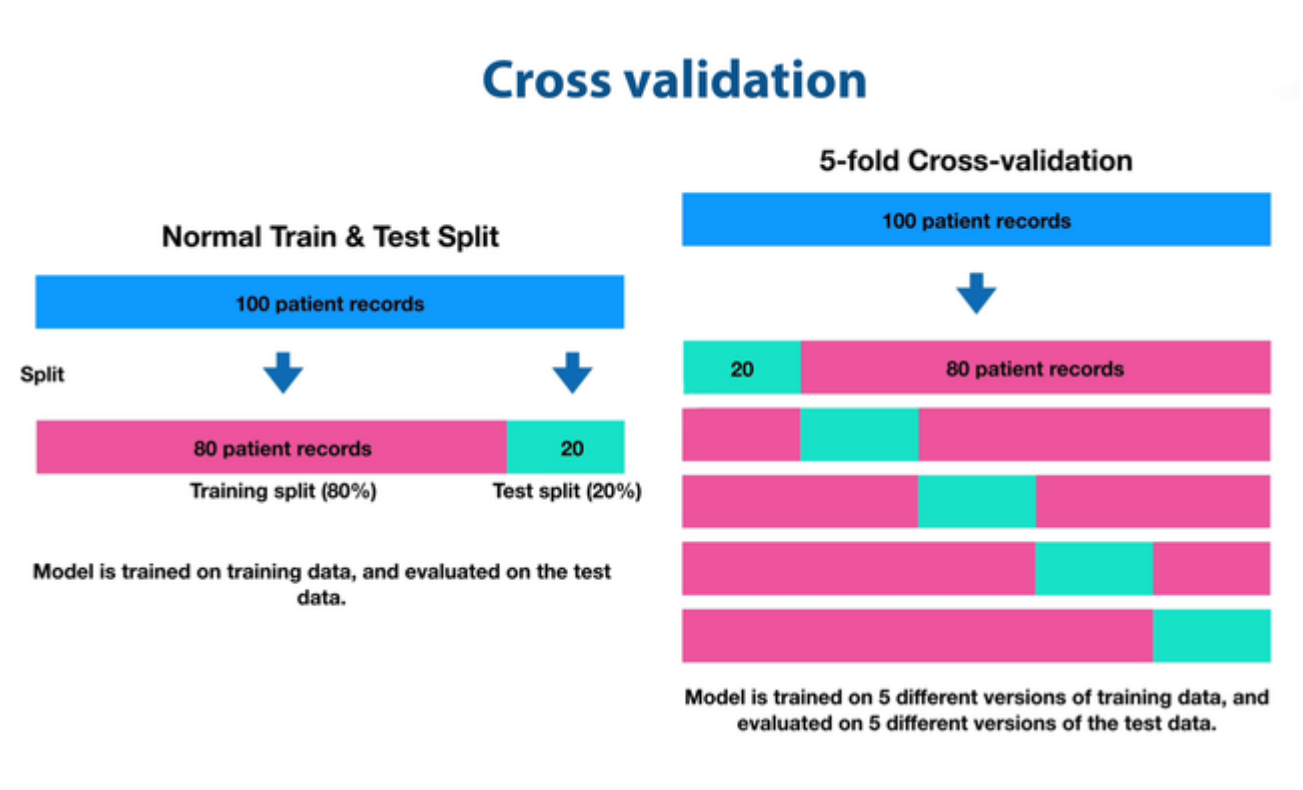
In [84]:
```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

seed = 89
np.random.seed(seed)

X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf = RandomForestClassifier()
clf.fit(X_train, y_train);

cross_val_score(clf,         # estimator
                X,           # features
                y,           # labels
                cv=5,        # cross-validation "iterations"
                scoring=None # scoring function. if None, default `clf.score()`
               )
```
Out[84]:
```
array([0.83606557, 0.8852459 , 0.78688525, 0.85      , 0.78333333])
```



k-fold cross-validation splits the data in training data and testing data in **k** possible combinations.

The result of `cross_val_score()` is an array of length `k` that represents the score of each different "version" of the data. This ensures that we didn't "luckily split" our data so that there were easy-to-find patterns in the training data.

In [85]:
```
cv_acc = cross_val_score(clf, X, y, cv=5, scoring="accuracy")
print(f"The cross-validated accuracy is: {np.mean(cv_acc)*100:.2f}%")
```
```
The cross-validated accuracy is: 80.17%
```

In [86]:
```
cv_prec = cross_val_score(clf, X, y, cv=5, scoring="precision")
print(f"The cross-validated precision is: {np.mean(cv_prec)*100:.2f}%")
```
```
The cross-validated precision is: 83.93%
```

In [87]:
```
cv_recall = cross_val_score(clf, X, y, cv=5, scoring="recall")
print(f"The cross-validated recall is: {np.mean(cv_recall)*100:.2f}%")
```
```
The cross-validated recall is: 83.64%
```

In [88]:
```
cv_f1 = cross_val_score(clf, X, y, cv=5, scoring="f1")
print(f"The cross-validated F1 is: {np.mean(cv_f1)*100:.2f}%")
```
```
The cross-validated F1 is: 83.27%
```

4.3 `-reg` Using the `scoring` parameter

In [89]:
```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor

seed = 62
np.random.seed(seed)

X = boston_df.drop("target", axis=1)
y = boston_df["target"]

model = RandomForestRegressor()
```

In [90]:
```
# R^2
np.random.seed(seed)
cv_r2 = cross_val_score(model, X, y, cv=5, scoring=None)
np.mean(cv_r2)
```
Out[90]:
```
0.6324075086312895
```

In [91]:
```
# Mean Absolute Error
np.random.seed(seed)
cv_mae = cross_val_score(model, X, y, cv=5, scoring="neg_mean_absolute_error")
np.mean(cv_mae)
```
Out[91]:
```
-2.9538927586876325
```

In [92]:
```
# Mean Squared Error
np.random.seed(seed)
cv_mse = cross_val_score(model, X, y, cv=5, scoring="neg_mean_squared_error")
np.mean(cv_mse)
```
Out[92]:
```
-20.989687216656954
```

4.4 `-clf` Using Scikit-learn functions

In [93]:
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

seed = 621
np.random.seed(seed)

X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

clf = RandomForestClassifier()
clf.fit(X_train, y_train);
```

In [94]:
```
# Make predictions using our model
y_preds = clf.predict(X_test)

# Evaluate the classifier using different functions
print("Classifier metrics on the test set\n")
print(f"Accuracy: \t{accuracy_score(y_test, y_preds)*100:.2f}%")
print(f"Precision: \t{precision_score(y_test, y_preds)*100:.2f}%")
print(f"Recall: \t{recall_score(y_test, y_preds)*100:.2f}%")
print(f"F1: \t\t{f1_score(y_test, y_preds)*100:.2f}%")
```
```
Classifier metrics on the test set

Accuracy:       83.61%
Precision:      83.33%
Recall:         88.24%
F1:             85.71%
```

4.4 `-reg` Using Scikit-learn functions

In [95]:
```
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

seed = 621
np.random.seed(seed)

X = boston_df.drop("target", axis=1)
y = boston_df["target"]

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

model = RandomForestRegressor()
model.fit(X_train, y_train);
```

In [96]:
```
# Make predictions using our model
y_preds = model.predict(X_test)

# Evaluate the regressor using different functions
print("Regression metrics on the test set\n")
print(f"R^2: \t{r2_score(y_test, y_preds)*100:.2f}%")
print(f"MAE: \t{mean_absolute_error(y_test, y_preds):.3f}")
print(f"MSE: \t{mean_squared_error(y_test, y_preds):.3f}")
```
```
Regression metrics on the test set

R^2:    82.97%
MAE:    2.405
MSE:    13.448
```

5. Improving a model

When improving a model, we call the first predictions **baseline predictions** and the first model **baseline model**.
On the subsequent iterations, we try to improve on these. There are two main ways:

1. From a data perspective
    - Could we collect more data? (more data is generally better)
    - Could we improve the data? (more info for each sample)

- From a model perspective
    - Could we use a more complex model?
    - Could we improve the current model by tuning the hyperparameters?

In [97]:
```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier()
```

In [98]:
```
# see the model's hyperparameters
clf.get_params()
```
Out[98]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_impurity_split': None,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

You can adjust the hyperparameters in three ways:

1. By hand
2. Randomly with **RandomSearchCV**
3. Exhaustively with **GridSearchCV**

5.1 Tuning hyperparameters by hand

Let's make three sets of data: training, validation and testing.

We're now going to try and adjust:

- `max_depth`
- `max_features`
- `min_samples_leaf`
- `min_samples_split`
- `n_estimators`

In [99]:
```
def evaluate_preds(y_true, y_preds):
    """
    Performs evaluation comparison on y_true labels vs y_preds labels
    """
    accuracy = accuracy_score(y_true, y_preds)
    precision = precision_score(y_true, y_preds)
    recall = recall_score(y_true, y_preds)
    f1 = f1_score(y_true, y_preds)

    metrics = {
        "accuracy": round(accuracy, 3),
        "precision": round(precision, 3),
        "recall": round(recall, 3),
        "f1": round(f1, 3)
    }

    print(metrics)
    return metrics
```

This time, since we need three sets of data, we have to manually split it.

In [100]:
```
seed = 724
np.random.seed(seed)

# Shuffle the data
heart_disease_shuffled = heart_disease.sample(frac=1)

# Set the split percentages
train_split = round(0.7 * len(heart_disease_shuffled))
valid_split = round(train_split + 0.15 * len(heart_disease_shuffled))

# Create X and y
X = heart_disease_shuffled.drop("target", axis=1)
y = heart_disease_shuffled["target"]

# Split the data
X_train, y_train = X[:train_split], y[:train_split]
X_val, y_val = X[train_split:valid_split], y[train_split:valid_split]
X_test, y_test = X[valid_split:], y[valid_split:]
```

In [101]:
```
clf.fit(X_train, y_train);
```

In [102]:
```
# Make baseline predictions on validation data
y_val_preds = clf.predict(X_val)

baseline_metrics = evaluate_preds(y_val, y_val_preds)
```
```
{'accuracy': 0.689, 'precision': 0.545, 'recall': 0.75, 'f1': 0.632}
```

Let's now create a second classifier and test it with different hyperparameters

In [103]:
```
seed = 724
np.random.seed(seed)

clf2 = RandomForestClassifier(n_estimators=600)
clf2.fit(X_train, y_train)

y_val_preds2 = clf2.predict(X_val)

baseline_metrics2 = evaluate_preds(y_val, y_val_preds2)

# doesn't change that much in this case, worse actually
```
```
{'accuracy': 0.644, 'precision': 0.5, 'recall': 0.688, 'f1': 0.579}
```

In [104]:
```
seed = 724
np.random.seed(seed)

clf3 = RandomForestClassifier(n_estimators=600, max_depth=15)
clf3.fit(X_train, y_train)

y_val_preds3 = clf3.predict(X_val)

baseline_metrics3 = evaluate_preds(y_val, y_val_preds3)
# nothing changed
```
```
{'accuracy': 0.644, 'precision': 0.5, 'recall': 0.688, 'f1': 0.579}
```

5.2 Tuning hyperparameters with RandomizedSearchCV

```python
In [105]: from sklearn.model_selection import RandomizedSearchCV

# create a dict having keys=hyperparam
# we want to tune and values=list of
# possible values we want to try
grid = {
    "n_estimators": [100,200,500,1000,1200],
    "max_depth": [None,5,10,20,30],
    "max_features": ["auto", "sqrt"],
    "min_samples_split": [2,4,6],
    "min_samples_leaf": [1,2,4]
}

seed = 724
np.random.seed(seed)

X = heart_disease_shuffled.drop("target", axis=1)
y = heart_disease_shuffled["target"]

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

clf = RandomForestClassifier(n_jobs=1)

# Setup RandomizedSearchCV
rs_clf = RandomizedSearchCV(
    estimator=clf,
    param_distributions=grid,
    n_iter=10,  # number of models to try
    cv=5,       # number of k cross-validations
    verbose=2,
)
```

`n_iter` is the most important parameter in this case: it represents how many random configurations will be taken into consideration. It will randomly choose one value per hyperparameter and test the model with that configuration.

```python
In [106]: # Fit the RandomizedSearchCV
rs_clf.fit(X_train, y_train);

# Total number of fits: n_iter * cv
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20, tot
al=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
```

```
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20, tot
al=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20, tot
al=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20, tot
al=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=1, max_features=auto, max_depth=20, tot
al=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None, t
otal=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None, t
otal=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None, t
otal=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None, t
otal=   0.1s
[CV] n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None
[CV]  n_estimators=100, min_samples_split=2, min_samples_leaf=2, max_features=sqrt, max_depth=None, t
otal=   0.1s
[CV] n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30
[CV]  n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30, to
tal=   1.1s
[CV] n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30
[CV]  n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30, to
tal=   1.1s
[CV] n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30
[CV]  n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30, to
tal=   1.1s
[CV] n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30
[CV]  n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30, to
tal=   1.1s
[CV] n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30
[CV]  n_estimators=1000, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=30, to
tal=   1.1s
[CV] n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10
[CV]  n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10, tot
al=   0.6s
[CV] n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10
[CV]  n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10, tot
al=   0.5s
[CV] n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10
[CV]  n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10, tot
al=   0.6s
[CV] n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10
[CV]  n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10, tot
al=   0.6s
[CV] n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10
[CV]  n_estimators=500, min_samples_split=2, min_samples_leaf=1, max_features=sqrt, max_depth=10, tot
al=   0.6s
[CV] n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10
[CV]  n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10, to
tal=   1.1s
[CV] n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10
[CV]  n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10, to
tal=   1.1s
[CV] n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10
[CV]  n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10, to
tal=   1.1s
[CV] n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10
[CV]  n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10, to
tal=   1.1s
[CV] n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10
[CV]  n_estimators=1000, min_samples_split=6, min_samples_leaf=4, max_features=auto, max_depth=10, to
tal=   1.1s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=2, max_features=auto, max_depth=10, to
tal=   1.3s
[CV] n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30
[CV]  n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30, tot
al=   0.5s
[CV] n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30
[CV]  n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30, tot
al=   0.5s
[CV] n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30
[CV]  n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30, tot
al=   0.6s
[CV] n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30
[CV]  n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30, tot
al=   0.6s
[CV] n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30
[CV]  n_estimators=500, min_samples_split=4, min_samples_leaf=2, max_features=sqrt, max_depth=30, tot
al=   0.6s
[CV] n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5
[CV]  n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5, tota
l=   0.7s
[CV] n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5
[CV]  n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5, tota
l=   0.6s
[CV] n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5
[CV]  n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5, tota
l=   0.6s
[CV] n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5
[CV]  n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5, tota
l=   0.7s
[CV] n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5
[CV]  n_estimators=500, min_samples_split=6, min_samples_leaf=1, max_features=sqrt, max_depth=5, tota
l=   0.6s
[CV] n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10, to
tal=   1.6s
[CV] n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10, to
tal=   1.5s
[CV] n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10, to
tal=   1.4s
[CV] n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10, to
tal=   1.4s
[CV] n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10
[CV]  n_estimators=1200, min_samples_split=6, min_samples_leaf=1, max_features=auto, max_depth=10, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30, to
tal=   1.3s
[CV] n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30
[CV]  n_estimators=1200, min_samples_split=2, min_samples_leaf=4, max_features=sqrt, max_depth=30, to
tal=   1.3s
```

```
[Parallel(n_jobs=1)]: Done  50 out of  50 | elapsed:   41.3s finished
```

```python
In [107]: # Show the combination that resulted
# in best cross-validation results
rs_clf.best_params_
```

```
Out[107]: {'n_estimators': 1200,
 'min_samples_split': 2,
 'min_samples_leaf': 2,
 'max_features': 'auto',
 'max_depth': 10}
```

From now on, when we call `predict()` or `score()`, **the best hyperparameters will be used**.

```python
In [108]: rs_y_preds = rs_clf.predict(X_test)

# Evaluate the predictions using our custom function
rs_metrics = evaluate_preds(y_test, rs_y_preds)
```

```
{'accuracy': 0.951, 'precision': 0.975, 'recall': 0.951, 'f1': 0.963}
```

5.3 Tuning hyperparameters with GridSearchCV

```python
In [109]: # Our current grid
grid
```

```
Out[109]: {'n_estimators': [100, 200, 500, 1000, 1200],
 'max_depth': [None, 5, 10, 20, 30],
 'max_features': ['auto', 'sqrt'],
 'min_samples_split': [2, 4, 6],
 'min_samples_leaf': [1, 2, 4]}
```

`GridSearchCV` is a brute-force algorithm. This means it will try every single possible combination. With large datasets and many hyperparameter options, this will take a really long time.
Thus, we'll reduce our hyperparameter options.

```python
In [110]: grid2 = {'n_estimators': [100, 200, 500],
    'max_depth': [None],
    'max_features': ['auto', 'sqrt'],
    'min_samples_split': [6],
    'min_samples_leaf': [1, 2]
}
```

```python
In [111]: from sklearn.model_selection import GridSearchCV

seed = 8915
np.random.seed(seed)

X = heart_disease_shuffled.drop("target", axis=1)
y = heart_disease_shuffled["target"]

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

clf = RandomForestClassifier(n_jobs=1)

# Setup GridSearchCV
gs_clf = GridSearchCV(
    estimator=clf,
    param_grid=grid2,
    cv=5,       # number of k cross-validations
    verbose=2,
)
```

```python
In [112]: gs_clf.fit(X_train, y_train);
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
```

```
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=auto, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.6s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.6s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.6s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=1, min_samples_split=6, n_estimators=500, t
otal=   0.6s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=100, t
otal=   0.1s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=200, t
otal=   0.2s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.6s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.6s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.5s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.6s
[CV] max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500
[CV]  max_depth=None, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=500, t
otal=   0.5s
```

```
[Parallel(n_jobs=1)]: Done  60 out of  60 | elapsed:   17.7s finished
```

```python
In [113]: gs_clf.best_params_
```

```
Out[113]: {'max_depth': None,
 'max_features': 'auto',
 'min_samples_leaf': 2,
 'min_samples_split': 6,
 'n_estimators': 100}
```

```python
In [114]: gs_y_preds = gs_clf.predict(X_test)

# evaluate the predictions
gs_metrics = evaluate_preds(y_test, gs_y_preds)
```

```
{'accuracy': 0.869, 'precision': 0.868, 'recall': 0.917, 'f1': 0.892}
```

Let's now compare our different models metrics.

```python
In [115]: compare_metrics_df = pd.DataFrame({
    "baseline": baseline_metrics,
    "clf2": baseline_metrics2,
    "random_search": rs_metrics,
    "grid search": gs_metrics
})
compare_metrics_df.plot.bar(figsize=(10,8));
```

6. Save and load a trained model

Two ways to save and load a trained model:

1. `pickle` module
2. `joblib` module

6.1 Saving and loading using `pickle`

```python
In [116]: import pickle

# Save a model
pickle.dump(gs_clf, open("gs_random_forest_classifier.pkl", 'wb'))
```

```python
In [117]: # Load a model
gs_clf_loaded = pickle.load(open("gs_random_forest_classifier.pkl", 'rb'))
gs_clf_loaded
```

```
Out[117]: GridSearchCV(cv=5, estimator=RandomForestClassifier(n_jobs=1),
             param_grid={'max_depth': [None], 'max_features': ['auto', 'sqrt'],
                         'min_samples_leaf': [1, 2], 'min_samples_split': [6],
                         'n_estimators': [100, 200, 500]},
             verbose=2)
```

```python
In [118]: # Test if it worked
pickled_y_preds = gs_clf_loaded.predict(X_test)
evaluate_preds(y_test, pickled_y_preds);
```

```
{'accuracy': 0.869, 'precision': 0.868, 'recall': 0.917, 'f1': 0.892}
```

6.2 Saving and loading using `joblib`

```python
In [119]: import joblib

# Save a model
joblib.dump(gs_clf, filename="gs_random_forest_classifier.jbl")
```

```
Out[119]: ['gs_random_forest_classifier.jbl']
```

```python
In [120]: # Load a model
gs_clf_loaded = joblib.load(filename="gs_random_forest_classifier.jbl")
gs_clf_loaded
```

```
Out[120]: GridSearchCV(cv=5, estimator=RandomForestClassifier(n_jobs=1),
             param_grid={'max_depth': [None], 'max_features': ['auto', 'sqrt'],
                         'min_samples_leaf': [1, 2], 'min_samples_split': [6],
                         'n_estimators': [100, 200, 500]},
             verbose=2)
```

```python
In [121]: # Test if it worked
joblib_y_preds = gs_clf_loaded.predict(X_test)
evaluate_preds(y_test, joblib_y_preds);
```

```
{'accuracy': 0.869, 'precision': 0.868, 'recall': 0.917, 'f1': 0.892}
```

6.3 Which one should you use?

According to the Scikit-learn model perstistence documentation:

> it may be better to use joblib's replacement of pickle (`dump` & `load`), which is more efficient on objects that carry large numpy arrays internally as is often the case for fitted scikit-learn estimators

7. Putting it all together

Steps we want to do (in one cell):

1. Fill missing data
2. Convert data to numbers
3. Build a model on the data

```python
In [122]: import pandas as pd
import numpy as np
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder

# Modelling
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split, GridSearchCV


seed = 689
np.random.seed(seed)

# Import data and drop rows with missing data
car_sales = pd.read_csv("./car-sales-extended-missing-data.csv")
car_sales.dropna(subset=["Price"], inplace=True)

# Define different features and transformer pipeline
categorical_features = ["Make", "Colour"]
categorical_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="constant", fill_value="missing")),
    ("onehot", OneHotEncoder(handle_unknown="ignore"))
])

door_feature = ["Doors"]
door_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="constant", fill_value=4)),
])

numeric_features = ["Odometer (KM)"]
numeric_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="mean")),
])

# Setup preprocessing steps (fill missing values, then convert to numbers)
preprocessor = ColumnTransformer(transformers=[
    ("cat", categorical_transformer, categorical_features),
    ("door", door_transformer, door_feature),
    ("num", numeric_transformer, numeric_features),
])

# Create a preprocessing and modelling Pipeline
model = Pipeline(steps=[
    ("preprocessor", preprocessor),
    ("model", RandomForestRegressor())
])

# Split data
X = car_sales.drop("Price", axis=1)
y = car_sales["Price"]

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

# Fit and score the model
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

```
Out[122]: 0.22006217452053511
```