Open in Colab (https://colab.research.google.com/github/mrdbourke/zero-to-mastery-ml/blob/master/section-2-data-science-and-ml-tools/introduction-to-scikit-learn.ipynb)
View source code (https://github.com/mrdbourke/zero-to-mastery-ml/blob/master/section-2-data-science-and-ml-tools/introduction-to-scikit-learn.ipynb) | Read notebook in online book format (https://dev.mrdbourke.com/zero-to-mastery-ml/introduction-to-scikit-learn/)

A Quick Machine Learning Modelling Tutorial with Python and Scikit-Learn

This notebook goes through a range of common and useful featues of the Scikit-Learn library.

There's a bunch here but I'm calling it quick because of how vast the Scikit-Learn library is.

Covering everything requires a full-blown documentation (https://scikit-learn.org/stable/user_guide.html), of which, if you ever get stuck, I'd highly recommend checking out.

In [5]:
```
import datetime
print(f"Last updated: {datetime.datetime.now()}")
```

```
Last updated: 2024-09-09 10:56:31.435756
```

What is Scikit-Learn (sklearn)?

Scikit-Learn (https://scikit-learn.org/stable/index.html), also referred to as `sklearn`, is an open-source Python machine learning library.

It's built on top on NumPy (Python library for numerical computing) and Matplotlib (Python library for data visualization).

a 6 step machine learning framework along will tools you can use for each step

Why Scikit-Learn?

Although the fields of data science and machine learning are vast, the main goal is finding patterns within data and then using those patterns to make predictions.

And there are certain categories which a majority of problems fall into.

If you're trying to create a machine learning model to predict whether an email is spam and or not spam, you're working on a classification problem (https://en.wikipedia.org/wiki/Statistical_classification#Binary_and_multiclass_classification) (whether something is one thing or another).

If you're trying to create a machine learning model to predict the price of houses given their characteristics, you're working on a regression problem (https://en.wikipedia.org/wiki/Regression_analysis) (predicting a number).

If you're trying to get a machine learning algorithm to group together similar samples (that you don't necessarily know which should go together), you're working on a clustering problem (https://developers.google.com/machine-learning/clustering/overview).

Once you know what kind of problem you're working on, there are also similar steps you'll take for each. Steps like splitting the data into different sets, one for your machine learning algorithms to learn on (the training set) and another to test them on (the testing set).

Choosing a machine learning model and then evaluating whether or not your model has learned anything.

Scikit-Learn offers Python implementations for doing all of these kinds of tasks (from preparing data to modelling data). Saving you from having to build them from scratch.

What does this notebook cover?

The Scikit-Learn library is very capable. However, learning everything off by heart isn't necessary. Instead, this notebook focuses some of the main use cases of the library.

More specifically, we'll cover:

0. An end-to-end Scikit-Learn worfklow
1. Getting the data ready
2. Choosing the right maching learning estimator/aglorithm/model for your problem
3. Fitting your chosen machine learning model to data and using it to make a prediction
4. Evaluting a machine learning model
5. Improving predictions through experimentation (hyperparameter tuning)
6. Saving and loading a pretrained model
7. Putting it all together in a pipeline

> **Note:** All of the steps in this notebook are focused on **supervised learning** (https://en.wikipedia.org/wiki/Supervised_learning) (having data and labels). The other side of supervised learning is **unsupervised learning** (https://en.wikipedia.org/wiki/Unsupervised_learning) (having data but no labels).

After going through it, you'll have the base knolwedge of Scikit-Learn you need to keep moving forward.

Where can I get help?

If you get stuck or think of something you'd like to do which this notebook doesn't cover, don't fear!

The recommended steps you take are:

1. **Try it** - Since Scikit-Learn has been designed with usability in mind, your first step should be to use what you know and try figure out the answer to your own question (getting it wrong is part of the process). If in doubt, run your code.
2. **Press SHIFT+TAB** - See you can the docstring of a function (information on what the function does) by pressing **SHIFT + TAB** inside it. Doing this is a good habit to develop. It'll improve your research skills and give you a better understanding of the library.
3. **Search for it** - If trying it on your own doesn't work, since someone else has probably tried to do something similar, try searching for your problem. You'll likely end up in 1 of 2 places:
   - Scikit-Learn documentation/user guide (https://scikit-learn.org/stable/user_guide.html) - the most extensive resource you'll find for Scikit-Learn information.
   - Stack Overflow (https://stackoverflow.com/) - this is the developers Q&A hub, it's full of questions and answers of different problems across a wide range of software development topics and chances are, there's one related to your problem.
   - ChatGPT (https://chat.openai.com/) - ChatGPT is very good at explaining code, however, it can make mistakes. Best to verify the code it writes first before using it. Try asking "Can you explain the following code for me? {your code here}" and then continue with follow up questions from there.

An example of searching for a Scikit-Learn solution might be:

> "how to tune the hyperparameters of a sklearn model"

Searching this on Google leads to the Scikit-Learn documentation for the `GridSearchCV` function: http://scikit-learn.org/stable/modules/grid_search.html (http://scikit-learn.org/stable/modules/grid_search.html)

The next steps here are to read through the documentation, check the examples and see if they line up to the problem you're trying to solve. If they do, **rewrite the code** to suit your needs, run it, and see what the outcomes are.

4. **Ask for help** - If you've been through the above 3 steps and you're still stuck, you might want to ask your question on Stack Overflow (https://www.stackoverflow.com) or in the ZTM Machine Learning and AI Discord channel. Be as specific as possible and provide details on what you've tried.

Remember, you don't have to learn all of the functions off by heart to begin with.

What's most important is continually asking yourself, "what am I trying to do with the data?".

Start by answering that question and then practicing finding the code which does it.

Let's get started.

First we'll import the libraries we've been using previously.

We'll also check the version of `sklearn` we've got.

In [6]:
```
# Standard imports
# %matplotlib inline # No longer required in newer versions of Jupyter (2022+)
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

import sklearn
print(f"Using Scikit-Learn version: {sklearn.__version__} (materials in this notebook require this version or n
```

```
Using Scikit-Learn version: 1.5.1 (materials in this notebook require this version or newer).
```

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

Before we get in-depth, let's quickly check out what an end-to-end Scikit-Learn workflow might look like.

Once we've seen an end-to-end workflow, we'll dive into each step a little deeper.

Specifically, we'll get hands-on with the following steps:

1. Getting data ready (split into features and labels, prepare train and test steps)
2. Choosing a model for our problem
3. Fit the model to the data and use it to make a prediction
4. Evaluate the model
5. Experiment to improve
6. Save a model for someone else to use

> **Note:** The following section is a bit information heavy but it is an end-to-end workflow. We'll go through it quite swiftly but we'll break it down more throughout the rest of the notebook. And since Scikit-Learn is such a vast library, capable of tackling many problems, the workflow we're using is only one example of how you can use it.

Random Forest Classifier Workflow for Classifying Heart Disease

1. Get the data ready

As an example dataset, we'll import `heart-disease.csv` .

This file contains anonymised patient medical records and whether or not they have heart disease or not (this is a classification problem since we're trying to predict whether something is one thing or another).

In [7]:
```
import pandas as pd
heart_disease = pd.read_csv('/home/smayan/Desktop/AI-ML-DS/sample_project/heart-disease.csv')
heart_disease.head()
```

Out[7]:

	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

Here, each row is a different patient and all columns except `target` are different patient characteristics.

The `target` column indicates whether the patient has heart disease (`target=1`) or not (`target=0`), this is our "label" columnm, the variable we're going to try and predict.

The rest of the columns (often called features) are what we'll be using to predict the `target` value.

> **Note:** It's a common custom to save features to a varialbe `X` and labels to a variable `y` . In practice, we'd like to use the `X` (features) to build a predictive algorithm to predict the `y` (labels).

In [8]:
```
# Create X (all the feature columns)
X = heart_disease.drop("target", axis=1)

# Create y (the target column)
y = heart_disease["target"]

# Check the head of the features DataFrame
X.head()
```

Out[8]:

	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 [9]:
```
# Check the head and the value counts of the labels
y.head(), y.value_counts()
```

Out[9]:
```
(0    1
 1    1
 2    1
 3    1
 4    1
 Name: target, dtype: int64,
 target
 1    165
 0    138
 Name: count, dtype: int64)
```

One of the most important practices in machine learning is to split datasets into training and test sets.

As in, a model will **train on the training set** to learn patterns and then those patterns can be **evaluated on the test set**.

Crucially, a model should **never** see testing data during training.

This is equivalent to a student studying course materials during the semester (training set) and then testing their abilities on the following exam (testing set).

Scikit-learn provides the `sklearn.model_selection.train_test_split` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html) method to split datasets in training and test sets.

> **Note:** A common practice to use an 80/20 or 70/30 or 75/25 split for training/testing data. There is also a third set, known as a validation set (e.g. 70/15/15 for training/validation/test) for hyperparamter tuning on but for now we'll focus on training and test sets.

In [10]:
```
# Split the data into training and test sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.25) # by default train_test_split uses 25% of t

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[10]:
```
((227, 13), (76, 13), (227,), (76,))
```

2. Choose the model and hyperparameters

Choosing a model often depends on the type of problem you're working on.

For example, there are different models that Scikit-Learn recommends whether you're working on a classification or regression problem.

You can see a map breaking down the different kinds of model options and recommendations in the Scikit-Learn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html).

Scikit-Learn refers to models as "estimators", however, they are often also referred to as `model` or `clf` (short for classifier).

A model's hyperparameters are settings you can change to adjust it for your problem, much like knobs on an oven you can tune to cook your favourite dish.

In [11]:
```
# Since we're working on a classification problem, we'll start with a RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier()
```

We can see the current hyperparameters of a model with the `get_params()` (https://scikit-learn.org/stable/developers/develop.html#get-params-and-set-params) method.

In [12]:
```
# View the current hyperparameters
clf.get_params()
```

Out[12]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'sqrt',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'monotonic_cst': None,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

We'll leave this as is for now, as Scikit-Learn models generally have good default settings.

3. Fit the model to the data and use it to make a prediction

Fitting a model a dataset involves passing it the data and asking it to figure out the patterns.

If there are labels (supervised learning), the model tries to work out the relationship between the data and the labels.

If there are no labels (unsupervised learning), the model tries to find patterns and group similar samples together.

Most Scikit-Learn models have the `fit(X, y)` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.fit) method built-in, where the `X` parameter is the features and the `y` parameter is the labels.

In our case, we start by fitting a model on the training split (`X_train` , `y_train`).

In [13]:
```
clf.fit(X=X_train, y=y_train)
```

Out[13]:
```
▾ RandomForestClassifier i ?
(https://
scikit-
learn.org/1.5/
modules/
generated/
RandomForestClassifier()
```

Use the model to make a prediction

The whole point of training a machine learning model is to use it to make some kind of prediction in the future.

Once your model instance is trained, you can use the `predict()` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.predict) method to predict a target value given a set of features.

In other words, use the model, along with some new, unseen and unlabelled data to predict the label.

> **Note:** Data you predict on should be in the same shape and format as data you trained on.

In [14]:
```
# This doesn't work... incorrect shapes
y_label = clf.predict(np.array([0, 2, 3, 4]))
```

```
/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/base.py:493: UserWarning: X does not have
valid feature names, but RandomForestClassifier was fitted with feature names
  warnings.warn(
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[14], line 2
      1 # This doesn't work... incorrect shapes
----> 2 y_label = clf.predict(np.array([0, 2, 3, 4]))

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/ensemble/_forest.py:904, in ForestCl
ssifier.predict(self, X)
    883 def predict(self, X):
    884     """
    885     Predict class for X.
    886 
   (...)
    902         The predicted classes.
    903     """
--> 904     proba = self.predict_proba(X)
    906     if self.n_outputs_ == 1:
    907         return self.classes_.take(np.argmax(proba, axis=1), axis=0)

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/ensemble/_forest.py:946, in ForestCl
ssifier.predict_proba(self, X)
    944 check_is_fitted(self)
    945 # Check data
--> 946 X = self._validate_X_predict(X)
    948 # Assign chunk of trees to jobs
    949 n_jobs, _, _ = _partition_estimators(self.n_estimators, self.n_jobs)

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/ensemble/_forest.py:641, in BaseFore
t._validate_X_predict(self, X)
    638 else:
    639     force_all_finite = True
--> 641 X = self._validate_data(
    642     X,
    643     dtype=DTYPE,
    644     accept_sparse="csr",
    645     reset=False,
    646     force_all_finite=force_all_finite,
    647 )
```

```
    648 if issparse(X) and (X.indices.dtype != np.intc or X.indptr.dtype != np.intc):
    649     raise ValueError("No support for np.int64 index based sparse matrices")

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/base.py:633, in BaseEstimator._valid
te_data(self, X, y, reset, validate_separately, cast_to_ndarray, **check_params)
    631         out = X, y
    632 elif not no_val_X and no_val_y:
--> 633     out = check_array(X, input_name="X", **check_params)
    634 elif no_val_X and not no_val_y:
    635     out = _check_y(y, **check_params)

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/validation.py:1050, in check_a
ray(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_writeable, force_all_finite, ensure_2d
allow_nd, ensure_min_samples, ensure_min_features, estimator, input_name)
   1043         else:
   1044             msg = (
   1045                 f"Expected 2D array, got 1D array instead:\narray={array}.\n"
   1046                 "Reshape your data either using array.reshape(-1, 1) if "
   1047                 "your data has a single feature or array.reshape(1, -1) "
   1048                 "if it contains a single sample."
   1049             )
-> 1050         raise ValueError(msg)
   1052 if dtype_numeric and hasattr(array.dtype, "kind") and array.dtype.kind in "USV":
   1053     raise ValueError(
   1054         "dtype='numeric' is not compatible with arrays of bytes/strings."
   1055         "Convert your data to numeric values explicitly instead."
   1056     )

ValueError: Expected 2D array, got 1D array instead:
array=[0. 2. 3. 4.].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape(1, -1) i
it contains a single sample.
```

Since our model was trained on data from `X_train` , predictions should be made on data in the same format and shape as `X_train` .

Our goal in many machine learning problems is to use patterns learned from the training data to make predictions on the test data (or future unseen data).

In [15]:

```
# In order to predict a label, data has to be in the same shape as X_train
X_test.head()
```

Out[15]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
288	57	1	0	110	335	0	1	143	1	3.0	1	1	3
188	50	1	2	140	233	0	1	163	0	0.6	1	1	3
47	47	1	2	138	257	0	0	156	0	0.0	2	0	2
80	41	1	2	112	250	0	1	179	0	0.0	2	0	2
141	43	1	0	115	303	0	1	181	0	1.2	1	0	2

In [16]:

```
# Use the model to make a prediction on the test data (further evaluation)
y_preds = clf.predict(X=X_test)
```

4. Evaluate the model

Now we've made some predictions, we can start to use some more Scikit-Learn methods to figure out how good our model is.

Each model or estimator has a built-in `score()` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier.score) method.

This method compares how well the model was able to learn the patterns between the features and labels.

The `score()` method for each model uses a standard evaluation metric to measure your model's results.

In the case of a classifier (our model), one of the most common evaluation metrics is accuracy (https://scikit-learn.org/stable/modules/model_evaluation.html#accuracy-score) (the fraction of correct predictions out of total predictions).

Let's check out our model's accuracy on the training set.

```
In [17]: # Evaluate the model on the training set
train_acc = clf.score(X=X_train, y=y_train)
print(f"The model's accuracy on the training dataset is: {train_acc*100}%")
```

```
The model's accuracy on the training dataset is: 100.0%
```

Woah! Looks like our model does pretty well on the training datset.

This is because it has a chance to see both data *and* labels.

How about the test dataset?

```
In [18]: # Evaluate the model on the test set
test_acc = clf.score(X=X_test, y=y_test)
print(f"The model's accuracy on the testing dataset is: {test_acc*100:.2f}%")
```

```
The model's accuracy on the testing dataset is: 86.84%
```

Hmm, looks like our model's accuracy is a bit less on the test dataset than the training dataset.

This is quite often the case, because remember, a model has never seen the testing examples before.

There are also a number of other evaluation methods we can use for our classification models.

All of the following classification metrics come from the `sklearn.metrics` (https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics) module:

- `classification_report(y_true, y_true)` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html#sklearn.metrics.classification_report) - Builds a text report showing various classification metrics such as precision, recall (https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html) and F1-score.
- `confusion_matrix(y_true, y_pred)` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html#sklearn.metrics.confusion_matrix) - Create a confusion matrix (https://en.wikipedia.org/wiki/Confusion_matrix) to compare predictions to truth labels.
- `accuracy_score(y_true, y_pred)` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html#sklearn.metrics.accuracy_score) - Find the accuracy score (the default metric) for a classifier.

All metrics have the following in common: they compare a model's predictions (`y_pred`) to truth labels (`y_true`).

```
In [19]: from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Create a classification report
print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support

           0       0.90      0.79      0.84        34
           1       0.85      0.93      0.89        42

    accuracy                           0.87        76
   macro avg       0.87      0.86      0.87        76
weighted avg       0.87      0.87      0.87        76
```

In [20]:
```
# Create a confusion matrix
conf_mat = confusion_matrix(y_test, y_preds)
conf_mat
```

Out[20]:
```
array([[27,  7],
       [ 3, 39]])
```

In [21]:
```
# Compute the accuracy score (same as the score() method for classifiers)
accuracy_score(y_test, y_preds)
```

Out[21]:
```
0.868421052631579
```

5. Experiment to improve

The first model you build is often referred to as a baseline (a baseline is often even simpler than the model we've used, a baseline could be "let's just by default predict the most common value and then try to improve").

Once you've got a baseline model, like we have here, it's important to remember, this is often not the final model you'll use.

The next step in the workflow is to try and improve upon your baseline model.

How?

With one of the most important mottos in machine learning...

Experiment, experiment, experiment!

Experiments can come in many different forms.

But let's break it into two.

1. From a model perspective.
2. From a data perspective.

From a model perspective may involve things such as using a more complex model or tuning your models hyperparameters.

From a data perspective may involve collecting more data or better quality data so your existing model has more of a chance to learn the patterns within.

If you're already working on an existing dataset, it's often easier try a series of model perspective experiments first and then turn to data perspective experiments if you aren't getting the results you're looking for.

One thing you should be aware of is if you're tuning a models hyperparameters in a series of experiments, your reuslts should always be cross-validated (we'll see this later on!).

Cross-validation (http://scikit-learn.org/stable/modules/cross_validation.html) is a way of making sure the results you're getting are consistent across your training and test datasets (because it uses multiple versions of training and test sets) rather than just luck because of the order the original training and test sets were created.

- Try different hyperparameters.
- All different parameters should be cross-validated.
  - **Note:** Beware of cross-validation for time series problems (as for time series, you don't want to mix samples from the future with samples from the past).

Different models you use will have different hyperparameters you can tune.

For the case of our model, the `RandomForestClassifier()` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html), we'll start trying different values for `n_estimators` (a measure for the number of trees in the random forest).

By default, `n_estimators=100`, so how about we try values from `100` to `200` and see what happens (generally more is better)?

In [22]:
```
# Try different numbers of estimators (trees)... (no cross-validation)
np.random.seed(42)
for i in range(100, 200, 10):
    print(f"Trying model with {i} estimators...")
    model = RandomForestClassifier(n_estimators=i).fit(X_train, y_train)
    print(f"Model accuracy on test set: {model.score(X_test, y_test) * 100:.2f}%")
    print("")
```

```
Trying model with 100 estimators...
Model accuracy on test set: 86.84%

Trying model with 110 estimators...
Model accuracy on test set: 86.84%

Trying model with 120 estimators...
Model accuracy on test set: 86.84%

Trying model with 130 estimators...
Model accuracy on test set: 84.21%

Trying model with 140 estimators...
Model accuracy on test set: 86.84%

Trying model with 150 estimators...
Model accuracy on test set: 86.84%

Trying model with 160 estimators...
Model accuracy on test set: 88.16%

Trying model with 170 estimators...
Model accuracy on test set: 86.84%

Trying model with 180 estimators...
Model accuracy on test set: 86.84%

Trying model with 190 estimators...
Model accuracy on test set: 86.84%
```

The metrics above were measured on a single train and test split.

Let's use `sklearn.model_selection.cross_val_score` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html) to measure the results across 5 different train and test sets.

We can achieve this by setting `cross_val_score(X, y, cv=5)` .

Where `X` is the *full* feature set and `y` is the *full* label set and `cv` is the number of train and test splits `cross_val_score` will automatically create from the data (in our case, `5` different splits, this is known as 5-fold cross-validation).

In [23]:
```
from sklearn.model_selection import cross_val_score

# With cross-validation
np.random.seed(42)
for i in range(100, 200, 10):
    print(f"Trying model with {i} estimators...")
    model = RandomForestClassifier(n_estimators=i).fit(X_train, y_train)

    # Measure the model score on a single train/test split
    model_score = model.score(X_test, y_test)
    print(f"Model accuracy on single test set split: {model_score * 100:.2f}%")

    # Measure the mean cross-validation score across 5 different train and test splits
    cross_val_mean = np.mean(cross_val_score(model, X, y, cv=5))
    print(f"5-fold cross-validation score: {cross_val_mean * 100:.2f}%")

    print("")
```

```
Trying model with 100 estimators...
Model accuracy on single test set split: 86.84%
5-fold cross-validation score: 82.15%

Trying model with 110 estimators...
Model accuracy on single test set split: 86.84%
5-fold cross-validation score: 81.17%

Trying model with 120 estimators...
Model accuracy on single test set split: 86.84%
5-fold cross-validation score: 83.16%

Trying model with 130 estimators...
Model accuracy on single test set split: 88.16%
5-fold cross-validation score: 83.14%

Trying model with 140 estimators...
Model accuracy on single test set split: 86.84%
5-fold cross-validation score: 82.48%

Trying model with 150 estimators...
Model accuracy on single test set split: 86.84%
5-fold cross-validation score: 80.17%

Trying model with 160 estimators...
Model accuracy on single test set split: 88.16%
5-fold cross-validation score: 80.83%

Trying model with 170 estimators...
Model accuracy on single test set split: 86.84%
5-fold cross-validation score: 81.83%

Trying model with 180 estimators...
Model accuracy on single test set split: 86.84%
5-fold cross-validation score: 81.50%

Trying model with 190 estimators...
Model accuracy on single test set split: 86.84%
```

```
5-fold cross-validation score: 81.83%
```

Which model had the best cross-validation score?

This is usually a better indicator of a quality model than a single split accuracy score.

Rather than set up and track the results of these experiments manually, we can get Scikit-Learn to do the exploration for us.

Scikit-Learn's `sklearn.model_selection.GridSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) is a way to search over a set of different hyperparameter values and automatically track which perform the best.

Let's test it!

In [24]:
```
# Another way to do it with GridSearchCV...
np.random.seed(42)
from sklearn.model_selection import GridSearchCV

# Define the parameters to search over in dictionary form
# (these can be any of your target model's hyperparameters)
param_grid = {'n_estimators': [i for i in range(100, 200, 10)]}

# Setup the grid search
grid = GridSearchCV(estimator=RandomForestClassifier(),
                    param_grid=param_grid,
                    cv=5,
                    verbose=1)

# Fit the grid search to the data
grid.fit(X, y)

# Find the best parameters
print(f"The best parameter values are: {grid.best_params_}")
print(f"With a score of: {grid.best_score_*100:.2f}%")
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
The best parameter values are: {'n_estimators': 120}
With a score of: 82.82%
```

We can extract the best model/estimator with the `best_estimator_` attribute.

In [25]:
```
# Set the model to be the best estimator
clf = grid.best_estimator_
clf
```

Out[25]:
```
▾ RandomForestClassifier    i ?
                              (https://
                              scikit-
                              learn.org/1.5/
                              modules/
                              generated/
RandomForestClassifier(n_estimators=120)
```

And now we've got the best cross-validated model, we can fit and score it on our original single train/test split of the data.

In [26]:
```
# Fit the best model
clf = clf.fit(X_train, y_train)

# Find the best model scores on our single test split
# (note: this may be lower than the cross-validation score since it's only on one split of the data)
print(f"Best model score on single split of the data: {clf.score(X_test, y_test)*100:.2f}%")
```

```
Best model score on single split of the data: 88.16%
```

6. Save a model for someone else to use

When you've done a few experiments and you're happy with how your model is doing, you'll likely want someone else to be able to use it.

This may come in the form of a teammate or colleague trying to replicate and validate your results or through a customer using your model as part of a service or application you offer.

Saving a model also allows you to reuse it later without having to go through retraining it. Which is helpful, especially when your training times start to increase.

You can save a Scikit-Learn model (https://scikit-learn.org/stable/model_persistence.html) using Python's in-built `pickle` module (https://docs.python.org/3/library/pickle.html).

In [27]:
```
import pickle

# Save an existing model to file
pickle.dump(model, open("random_forest_model_1.pkl", "wb"))
```

In [28]:
```
# Load a saved pickle model and evaluate it
loaded_pickle_model = pickle.load(open("random_forest_model_1.pkl", "rb"))
print(f"Loaded pickle model prediction score: {loaded_pickle_model.score(X_test, y_test) * 100:.2f}%")
```

```
Loaded pickle model prediction score: 86.84%
```

For larger models, it may be more efficient to use Joblib (https://joblib.readthedocs.io/en/stable/).

In [29]:
```
from joblib import dump, load

# Save a model using joblib
dump(model, "random_forest_model_1.joblib")
```

Out[29]:
```
['random_forest_model_1.joblib']
```

In [38]:
```
# Load a saved joblib model and evaluate it
loaded_joblib_model = load("random_forest_model_1.joblib")
print(f"Loaded joblib model prediction score: {loaded_joblib_model.score(X_test, y_test) * 100:.2f}%")
```

```
Loaded joblib model prediction score: 96.72%
```

Woah!

We've covered a lot of ground fast...

Let's break things down a bit more by revisting each section.

1. Getting the data ready

Data doesn't always come ready to use with a Scikit-Learn machine learning model.

Three of the main steps you'll often have to take are:

- Splitting the data into features (usually `X`) and labels (usually `y`).
- Splitting the data into training and testing sets (and possibly a validation set).
- Filling (also called imputing) or disregarding missing values.
- Converting non-numerical values to numerical values (also call feature encoding).

Let's see an example.

In [39]:
```
# Splitting the data into X & y
heart_disease.head()
```

Out[39]:

	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 [40]:

```python
# Splitting the data into features (X) and labels (y)
X = heart_disease.drop('target', axis=1)
X
```

Out[40]:

	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
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3	110	264	0	1	132	0	1.2	1	0	3
300	68	1	0	144	193	1	1	141	0	3.4	1	2	3
301	57	1	0	130	131	0	1	115	1	1.2	1	1	3
302	57	0	1	130	236	0	0	174	0	0.0	1	1	2

303 rows × 13 columns

Nice! Looks like our dataset has 303 samples with 13 features (13 columns).

Let's check out the labels.

In [41]:

```python
y = heart_disease['target']
y
```

Out[41]:
```
0      1
1      1
2      1
3      1
4      1
      ..
298    0
299    0
300    0
301    0
302    0
Name: target, Length: 303, dtype: int64
```

Beautiful, 303 labels with values of `0` (no heart disease) and `1` (heart disease).

Now let's split our data into training and test sets, we'll use an 80/20 split (80% of samples for training and 20% of samples for testing).

In [42]:
```
# Splitting the data into training and test sets
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) # you can change the test size

# Check the shapes of different data splits
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out[42]:
```
((242, 13), (61, 13), (242,), (61,))
```

In [43]:
```
# 80% of data is being used for the training set (the model will learn patterns on these samples)
X.shape[0] * 0.8
```

Out[43]:
```
242.4
```

In [44]:
```
# And 20% of the data is being used for the testing set (the model will be evaluated on these samples)
X.shape[0] * 0.2
```

Out[44]:
```
60.6
```

1.1 Make sure it's all numerical

Computers love numbers.

So one thing you'll often have to make sure of is that your datasets are in numerical form.

This even goes for datasets which contain non-numerical features that you may want to include in a model.

For example, if we were working with a car sales dataset, how might we turn features such as `Make` and `Colour` into numbers?

Let's figure it out.

First, we'll import the `car-sales-extended.csv` dataset.

In [45]:
```
# Import car-sales-extended.csv
car_sales = pd.read_csv("/home/smayan/Desktop/AI-ML-DS/ZTM/Scikit-Learn/car-sales-extended.csv")
car_sales
```

Out[45]:

	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
...	...	...	...	...	...
995	Toyota	Black	35820	4	32042
996	Nissan	White	155144	3	5716
997	Nissan	Blue	66604	4	31570
998	Honda	White	215883	4	4001
999	Toyota	Blue	248360	4	12732

1000 rows × 5 columns

We can check the dataset types with `.dtypes`.

In [46]:
```
car_sales.dtypes
```

Out[46]:
```
Make             object
Colour           object
Odometer (KM)     int64
Doors             int64
Price             int64
dtype: object
```

Notice the `Make` and `Colour` features are of `dtype=object` (they're strings) where as the rest of the columns are of `dtype=int64`.

If we want to use the `Make` and `Colour` features in our model, we'll need to figure out how to turn them into numerical form.

In [47]:
```
# Split into X & y and train/test
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)
```

Now let's try and build a model on our `car_sales` data.

In [48]:
```
# Try to predict with random forest on price column (doesn't work)
from sklearn.ensemble import RandomForestRegressor

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_15677/1044518071.py in ?()
      1 # Try to predict with random forest on price column (doesn't work)
      2 from sklearn.ensemble import RandomForestRegressor
      3 
      4 model = RandomForestRegressor()
----> 5 model.fit(X_train, y_train)
      6 model.score(X_test, y_test)

/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/base.py in ?(estimator, *args, **kwargs)
   1469                 skip_parameter_validation=(
   1470                     prefer_skip_nested_validation or global_skip_validation
   1471                 )
   1472             ):
-> 1473                 return fit_method(estimator, *args, **kwargs)

/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/ensemble/_forest.py in ?(self, X, y, samp
e_weight)
    359         # Validate or convert input data
    360         if issparse(y):
    361             raise ValueError("sparse multilabel-indicator for y is not supported.")
    362 
--> 363         X, y = self._validate_data(
    364             X,
    365             y,
    366             multi_output=True,

/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/base.py in ?(self, X, y, reset, validate_
eparately, cast_to_ndarray, **check_params)
    646                 if "estimator" not in check_y_params:
    647                     check_y_params = {**default_check_params, **check_y_params}
    648                 y = check_array(y, input_name="y", **check_y_params)
    649             else:
--> 650                 X, y = check_X_y(X, y, **check_params)
    651             out = X, y
    652 
    653         if not no_val_X and check_params.get("ensure_2d", True):

/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/validation.py in ?(X, y, accept_spa
se, accept_large_sparse, dtype, order, copy, force_writeable, force_all_finite, ensure_2d, allow_nd, multi_outp
t, ensure_min_samples, ensure_min_features, y_numeric, estimator)
   1297         raise ValueError(
   1298             f"{estimator_name} requires y to be passed, but the target y is None"
   1299         )
   1300 
-> 1301     X = check_array(
   1302         X,
   1303         accept_sparse=accept_sparse,
   1304         accept_large_sparse=accept_large_sparse,

/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/validation.py in ?(array, accept_sp
rse, accept_large_sparse, dtype, order, copy, force_writeable, force_all_finite, ensure_2d, allow_nd, ensure_mi
_samples, ensure_min_features, estimator, input_name)
   1009                         )
   1010                     array = xp.astype(array, dtype, copy=False)
   1011                 else:
   1012                     array = _asarray_with_order(array, order=order, dtype=dtype, xp=xp)
-> 1013             except ComplexWarning as complex_warning:
   1014                 raise ValueError(
   1015                     "Complex data not supported\n{}\n".format(array)
   1016                 ) from complex_warning

/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/_array_api.py in ?(array, dtype, or
er, copy, xp, device)
    747         # Use NumPy API to support order
    748         if copy is True:
    749             array = numpy.array(array, order=order, dtype=dtype)
    750         else:
--> 751             array = numpy.asarray(array, order=order, dtype=dtype)
    752 
    753         # At this point array is a NumPy ndarray. We convert it to an array
    754         # container that is consistent with the input's namespace.

/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/pandas/core/generic.py in ?(self, dtype, copy)
   2149     def __array__(
```

```
   2150         self, dtype: npt.DTypeLike | None = None, copy: bool_t | None = None
   2151     ) -> np.ndarray:
   2152         values = self._values
-> 2153         arr = np.asarray(values, dtype=dtype)
   2154         if (
   2155             astype_is_view(values.dtype, arr.dtype)
   2156             and using_copy_on_write()

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

Oops... this doesn't work, we'll have to convert the non-numerical features into numbers first.

The process of turning categorical features into numbers is often referred to as **encoding**.

Scikit-Learn has a fantastic in-depth guide on *Encoding categorical features* (https://scikit-learn.org/stable/modules/preprocessing.html#encoding-categorical-features).

But let's look at one of the most straightforward ways to turn categorical features into numbers, one-hot encoding (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html).

In machine learning, one-hot encoding (https://en.wikipedia.org/wiki/One-hot#Machine_learning_and_statistics) gives a value of `1` to the target value and a value of `0` to the other values.

For example, let's say we had five samples and three car make options, Honda, Toyota, BMW.

And our samples were:

1. Honda
2. BMW
3. BMW
4. Toyota
5. Toyota

If we were to one-hot encode these, it would look like:

Sample	Honda	Toyota	BMW
1	1	0	0
2	0	0	1
3	0	0	1
4	0	1	0
5	0	1	0

Notice how there's a 1 for each target value but a 0 for each other value.

We can use the following steps to one-hot encode our dataset:

1. Import `sklearn.preprocessing.OneHotEncoder` (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html) to one-hot encode our features and `sklearn.compose.ColumnTransformer` (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html) to target the specific columns of our DataFrame to transform.
2. Define the categorical features we'd like to transform.
3. Create an instance of the `OneHotEncoder`.
4. Create an instance of `ColumnTransformer` and feed it the transforms we'd like to make.
5. Fit the instance of the `ColumnTransformer` to our data and transform it with the `fit_transform(X)` (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html#sklearn.compose.ColumnTransformer.fit_transform) method.

> **Note:** In Scikit-Learn, the term "transformer" is often used to refer to something that *transforms* data.

In [54]:
```python
# 1. Import OneHotEncoder and ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

# 2. Define the categorical features to transform
categorical_features = ["Make", "Colour", "Doors"]

# 3. Create an instance of OneHotEncoder
one_hot = OneHotEncoder()

# 4. Create an instance of ColumnTransformer
transformer = ColumnTransformer([("one_hot", # name
                                  one_hot, # transformer
                                  categorical_features)], # columns to transform
                                  remainder="passthrough") # what to do with the rest of the columns? ("passthr

# 5. Turn the categorical features into numbers (this will return an array-like sparse matrix, not a DataFrame)
transformed_X = transformer.fit_transform(X)
```

```python
transformed_X
```

Out[54]:
```
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]])
```

> **Note:** You might be thinking why we considered `Doors` as a categorical variable. Which is a good question considering `Doors` is already numerical. Well, the answer is that `Doors` could be either numerical or categorical. However, I've decided to go with categorical, since where I'm from, number of doors is often a different *category* of car. For example, you can shop for 4-door cars or shop for 5-door cars (which always confused me since where's the 5th door?). However, you could experiment with treating this value as numerical or categorical, training a model on each, and then see how each model performs.

Woah! Looks like our samples are all numerical, what did our data look like previously?

In [55]:
```
X.head()
```

Out[55]:

	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

It seems `OneHotEncoder` and `ColumnTransformer` have turned all of our data samples into numbers.

Let's check out the first transformed sample.

In [56]:
```
# View first transformed sample
transformed_X[0]
```

Out[56]:
```
array([0.0000e+00, 1.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00,
       0.0000e+00, 0.0000e+00, 0.0000e+00, 1.0000e+00, 0.0000e+00,
       1.0000e+00, 0.0000e+00, 3.5431e+04])
```

And what were these values originally?

In [57]:
```
# View original first sample
X.iloc[0]
```

Out[57]:
```
Make             Honda
Colour           White
Odometer (KM)    35431
Doors                4
Name: 0, dtype: object
```

1.1.1 Nuemrically encoding data with pandas

Another way we can numerically encode data is directly with pandas.

We can use the `pandas.get_dummies()` (https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html) (or `pd.get_dummies()` for short) method and then pass it our target columns.

In return, we'll get a one-hot encoded version of our target columns.

Let's remind ourselves of what our DataFrame looks like.

In [58]:
```
# View head of original DataFrame
car_sales.head()
```

Out[58]:

	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

Wonderful, now let's use `pd.get_dummies()` to turn our categorical variables into one-hot encoded variables.

In [59]:
```
# One-hot encode categorical variables
categorical_variables = ["Make", "Colour", "Doors"]
dummies = pd.get_dummies(data=car_sales[categorical_variables])
dummies
```

Out[59]:

	Make_BMW	Make_Honda	Make_Nissan	Make_Toyota	Colour_Black	Colour_Blue	Colour_Green	Colour_Red	Colour_White	Doors_3
0	False	True	False	False	False	False	False	False	True	False
1	True	False	False	False	False	True	False	False	False	False
2	False	True	False	False	False	False	False	False	True	False
3	False	False	False	True	False	False	False	False	True	False
4	False	False	True	False	False	True	False	False	False	True
...	...	...	...	...	...	...	...	...	...	...
995	False	False	False	True	True	False	False	False	False	False
996	False	False	True	False	False	False	False	False	True	True
997	False	False	True	False	False	True	False	False	False	False
998	False	True	False	False	False	False	False	False	True	False
999	False	False	False	True	False	True	False	False	False	False

1000 rows × 12 columns

Nice!

Notice how there's a new column for each categorical option (e.g. `Make_BMW` , `Make_Honda` , etc).

But also notice how it also missed the `Doors` column?

This is because `Doors` is already numeric, so for `pd.get_dummies()` to work on it, we can change it to type `object` .

By default, `pd.get_dummies()` also turns all of the values to bools (`True` or `False`).

We can get the returned values as `0` or `1` by setting `dtype=float` .

In [60]:

```
# Have to convert doors to object for dummies to work on it...
car_sales["Doors"] = car_sales["Doors"].astype(object)
dummies = pd.get_dummies(data=car_sales[["Make", "Colour", "Doors"]],
                         dtype=float)
dummies
```

Out[60]:

	Make_BMW	Make_Honda	Make_Nissan	Make_Toyota	Colour_Black	Colour_Blue	Colour_Green	Colour_Red	Colour_White	Doors_3
0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
1	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
3	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0
4	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0
...	...	...	...	...	...	...	...	...	...	...
995	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
996	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0
997	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
998	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
999	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0

1000 rows × 12 columns

Woohoo!

We've now turned our data into fully numeric form using Scikit-Learn and pandas.

Now you might be wondering...

Should you use Scikit-Learn or pandas for turning data into numerical form?

And the answer is either.

But as a rule of thumb:

- If you're performing **quick data analysis and running small modelling experiments**, use `pandas` as it's generally quite fast to get up and running.
- If you're performing a **larger scale modelling experiment** or would like to put your **data processing steps into a production pipeline**, I'd recommend leaning towards Scikit-Learn, specifically a Scikit-Learn Pipeline (https://scikit-learn.org/stable/modules/compose.html#pipeline) (chaining together multiple estimator/modelling steps).

Since we've turned our data into numerical form, how about we try and fit our model again?

Let's recreate a train/test split except this time we'll use `transformed_X` instead of `X`.

In [61]:

```
np.random.seed(42)

# Create train and test splits with transformed_X
X_train, X_test, y_train, y_test = train_test_split(transformed_X,
                                                    y,
                                                    test_size=0.2)

# Create the model instance
model = RandomForestRegressor()

# Fit the model on the numerical data (this errored before since our data wasn't fully numeric)
model.fit(X_train, y_train)

# Score the model (returns r^2 metric by default, also called coefficient of determination, higher is better)
model.score(X_test, y_test)
```

Out[61]:

```
0.3235867221569877
```

1.2 What if there were missing values in the data?

Holes in the data means holes in the patterns your machine learning model can learn.

Many machine learning models don't work well or produce errors when they're used on datasets with missing values.

A missing value can appear as a blank, as a `NaN` or something similar.

There are two main options when dealing with missing values:

1. **Fill them with some given or calculated value (imputation)** - For example, you might fill missing values of a numerical column with the mean of all the other values. The practice of calculating or figuring out how to fill missing values in a dataset is called **imputing**. For a great resource on imputing missing values, I'd recommend refering to the Scikit-Learn user guide (https://scikit-learn.org/stable/modules/impute.html).
2. **Remove them** - If a row or sample has missing values, you may opt to remove them from your dataset completely. However, this potentially results in using less data to build your model.

> **Note:** Dealing with missing values differs from problem to problem, meaning there's no 100% best way to fill missing values across datasets and problem types. It will often take careful experimentation and practice to figure out the best way to deal with missing values in your own datasets.

To practice dealing with missing values, let's import a version of the `car_sales` dataset with several missing values.

In [62]:
```
# Import car sales dataframe with missing values
car_sales_missing = pd.read_csv("/home/smayan/Desktop/AI-ML-DS/ZTM/Scikit-Learn/car-sales-extended-missing-data
car_sales_missing
```

Out[62]:

	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
...	...	...	...	...	...
995	Toyota	Black	35820.0	4.0	32042.0
996	NaN	White	155144.0	3.0	5716.0
997	Nissan	Blue	66604.0	4.0	31570.0
998	Honda	White	215883.0	4.0	4001.0
999	Toyota	Blue	248360.0	4.0	12732.0

1000 rows × 5 columns

If you're dataset is large, it's likely you aren't going to go through it sample by sample to find the missing values.

Luckily, pandas has a method called `pd.DataFrame.isna()` (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.isna.html) which is able to detect missing values.

Let's try it on our DataFrame.

In [63]:
```
# Get the sum of all missing values
car_sales_missing.isna().sum()
```

Out[63]:
```
Make            49
Colour          50
Odometer (KM)   50
Doors           50
Price           50
dtype: int64
```

Hmm... seems there's about 50 or so missing values per column.

How about we try and split the data into features and labels, then convert the categorical data to numbers, then split the data into training and test and then try and fit a model on it (just like we did before)?

In [64]:
```
# Create features
X_missing = car_sales_missing.drop("Price", axis=1)
print(f"Number of missing X values:\n{X_missing.isna().sum()}")
```

```
Number of missing X values:
Make            49
Colour          50
Odometer (KM)   50
Doors           50
dtype: int64
```

In [65]:
```
# Create labels
y_missing = car_sales_missing["Price"]
print(f"Number of missing y values: {y_missing.isna().sum()}")
```

```
Number of missing y values: 50
```

Now we can convert the categorical columns into one-hot encodings (just as before).

In [66]:
```
# Let's convert the categorical columns to one hot encoded (code copied from above)
# Turn the categories (Make and Colour) into numbers
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

categorical_features = ["Make", "Colour", "Doors"]

one_hot = OneHotEncoder()

transformer = ColumnTransformer([("one_hot",
                                  one_hot,
                                  categorical_features)],
                                remainder="passthrough",
                                sparse_threshold=0) # return a sparse matrix or not

transformed_X_missing = transformer.fit_transform(X_missing)
transformed_X_missing
```

Out[66]:
```
array([[0.00000e+00, 1.00000e+00, 0.00000e+00, ..., 0.00000e+00,
        0.00000e+00, 3.54310e+04],
       [1.00000e+00, 0.00000e+00, 0.00000e+00, ..., 1.00000e+00,
        0.00000e+00, 1.92714e+05],
       [0.00000e+00, 1.00000e+00, 0.00000e+00, ..., 0.00000e+00,
        0.00000e+00, 8.47140e+04],
       ...,
       [0.00000e+00, 0.00000e+00, 1.00000e+00, ..., 0.00000e+00,
        0.00000e+00, 6.66040e+04],
       [0.00000e+00, 1.00000e+00, 0.00000e+00, ..., 0.00000e+00,
        0.00000e+00, 2.15883e+05],
       [0.00000e+00, 0.00000e+00, 0.00000e+00, ..., 0.00000e+00,
        0.00000e+00, 2.48360e+05]])
```

Finally, let's split the missing data samples into train and test sets and then try to fit and score a model on them.

In [67]:
```
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(transformed_X_missing,
                                                    y_missing,
                                                    test_size=0.2)

# Fit and score a model
model = RandomForestRegressor()
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[67], line 8
      6 # Fit and score a model
      7 model = RandomForestRegressor()
----> 8 model.fit(X_train, y_train)
      9 model.score(X_test, y_test)

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/base.py:1473, in _fit_context.<local
>.decorator.<locals>.wrapper(estimator, *args, **kwargs)
   1466     estimator._validate_params()
   1468 with config_context(
   1469     skip_parameter_validation=(
   1470         prefer_skip_nested_validation or global_skip_validation
   1471     )
   1472 ):
-> 1473     return fit_method(estimator, *args, **kwargs)

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/ensemble/_forest.py:363, in BaseFore
t.fit(self, X, y, sample_weight)
    360 if issparse(y):
    361     raise ValueError("sparse multilabel-indicator for y is not supported.")
--> 363 X, y = self._validate_data(
    364     X,
    365     y,
    366     multi_output=True,
    367     accept_sparse="csc",
    368     dtype=DTYPE,
    369     force_all_finite=False,
    370 )
    371 # _compute_missing_values_in_feature_mask checks if X has missing values and
    372 # will raise an error if the underlying tree base estimator can't handle missing
    373 # values. Only the criterion is required to determine if the tree supports
    374 # missing values.
    375 estimator = type(self.estimator)(criterion=self.criterion)

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/base.py:650, in BaseEstimator._valid
te_data(self, X, y, reset, validate_separately, cast_to_ndarray, **check_params)
```

```
    648         y = check_array(y, input_name="y", **check_y_params)
    649     else:
--> 650         X, y = check_X_y(X, y, **check_params)
    651     out = X, y
    653 if not no_val_X and check_params.get("ensure_2d", True):

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/validation.py:1318, in check_X
y(X, y, accept_sparse, accept_large_sparse, dtype, order, copy, force_writeable, force_all_finite, ensure_2d, a
low_nd, multi_output, ensure_min_samples, ensure_min_features, y_numeric, estimator)
   1297     raise ValueError(
   1298         f"{estimator_name} requires y to be passed, but the target y is None"
   1299     )
   1301 X = check_array(
   1302     X,
   1303     accept_sparse=accept_sparse,
   (...)
   1315     input_name="X",
   1316 )
-> 1318 y = _check_y(y, multi_output=multi_output, y_numeric=y_numeric, estimator=estimator)
   1320 check_consistent_length(X, y)
   1322 return X, y

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/validation.py:1328, in _check_
(y, multi_output, y_numeric, estimator)
   1326 """Isolated part of check_X_y dedicated to y validation"""
   1327 if multi_output:
-> 1328     y = check_array(
   1329         y,
   1330         accept_sparse="csr",
   1331         force_all_finite=True,
   1332         ensure_2d=False,
   1333         dtype=None,
   1334         input_name="y",
   1335         estimator=estimator,
   1336     )
   1337 else:
   1338     estimator_name = _check_estimator_name(estimator)
```

```
File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/validation.py:1064, in check_a
ray(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_writeable, force_all_finite, ensure_2d
allow_nd, ensure_min_samples, ensure_min_features, estimator, input_name)
   1058     raise ValueError(
   1059         "Found array with dim %d. %s expected <= 2."
   1060         % (array.ndim, estimator_name)
   1061     )
   1063 if force_all_finite:
-> 1064     _assert_all_finite(
   1065         array,
   1066         input_name=input_name,
   1067         estimator_name=estimator_name,
   1068         allow_nan=force_all_finite == "allow-nan",
   1069     )
   1071 if copy:
   1072     if _is_numpy_namespace(xp):
   1073         # only make a copy if `array` and `array_orig` may share memory`

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/validation.py:123, in _assert_
ll_finite(X, allow_nan, msg_dtype, estimator_name, input_name)
    120 if first_pass_isfinite:
    121     return
--> 123 _assert_all_finite_element_wise(
    124     X,
    125     xp=xp,
    126     allow_nan=allow_nan,
    127     msg_dtype=msg_dtype,
    128     estimator_name=estimator_name,
    129     input_name=input_name,
    130 )

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/sklearn/utils/validation.py:172, in _assert_
ll_finite_element_wise(X, xp, allow_nan, msg_dtype, estimator_name, input_name)
    155 if estimator_name and input_name == "X" and has_nan_error:
    156     # Improve the error message on how to handle missing values in
    157     # scikit-learn.
    158     msg_err += (
    159         f"\n{estimator_name} does not accept missing values"
```

```
    160         " encoded as NaN natively. For supervised learning, you might want"
   (...)
    170         "#estimators-that-handle-nan-values"
    171     )
--> 172 raise ValueError(msg_err)

ValueError: Input y contains NaN.
```

Ahh... dam! Looks like the model we're trying to use doesn't work with missing values.

When we try to fit it on a dataset with missing samples, Scikit-Learn produces the error:

`ValueError: Input X contains NaN. RandomForestRegressor does not accept missing values encoded as NaN natively...`

Looks like if we want to use `RandomForestRegressor`, we'll have to either fill or remove the missing values.

Note: Scikit-Learn does have a list of models which can handle NaNs or missing values directly (https://scikit-learn.org/stable/modules/impute.html#estimators-that-handle-nan-values).

Let's see what values are missing again.

In [68]:
```
car_sales_missing.isna().sum()
```

Out[68]:
```
Make             49
Colour           50
Odometer (KM)    50
Doors            50
Price            50
dtype: int64
```

How can fill (impute) or remove these?

1.2.1 Fill missing data with pandas

Let's see how we might fill missing values with pandas.

For categorical values, one of the simplest ways is to fill the missing fields with the string `"missing"`.

We could do this for the `Make` and `Colour` features.

As for the `Doors` feature, we could use `"missing"` or we could fill it with the most common option of `4`.

With the `Odometer (KM)` feature, we can use the mean value of all the other values in the column.

And finally, for those samples which are missing a `Price` value, we can remove them (since `Price` is the target value, removing probably causes less harm than imputing, however, you could design an experiment to test this).

In summary:

Column/Feature	Fill missing value with
`Make`	`"missing"`
`Colour`	`"missing"`
`Doors`	4 (most common value)
`Odometer (KM)`	mean of `Odometer (KM)`
`Price` (target)	NA, remove samples missing `Price`

> **Note:** The practice of filling missing data with given or calculated values is called **imputation** (https://scikit-learn.org/stable/modules/impute.html). And it's important to remember there's no perfect way to fill missing data (unless it's with data that should've actually been there in the first place). The methods we're using are only one of many. The techniques you use will depend heavily on your dataset. A good place to look would be searching for "data imputation techniques".

Let's start with the `Make` column.

We can use the pandas method `fillna(value="missing", inplace=True)` (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html) to fill all the missing values with the string `"missing"`.

In [69]:
```
# Fill the missing values in the Make column
car_sales_missing["Make"].fillna(value="missing", inplace=True)
```

```
/tmp/ipykernel_15677/3341163365.py:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Se
ies through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on
hich we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)
or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.


  car_sales_missing["Make"].fillna(value="missing", inplace=True)
```

And we can do the same with the `Colour` column.

In [70]:
```
# Fill the Colour column
car_sales_missing["Colour"].fillna(value="missing", inplace=True)
```

```
/tmp/ipykernel_15677/2718395805.py:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Se
ies through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on
hich we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)
or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.


  car_sales_missing["Colour"].fillna(value="missing", inplace=True)
```

How many missing values do we have now?

In [71]:
```
car_sales_missing.isna().sum()
```

Out[71]:
```
Make             0
Colour           0
Odometer (KM)   50
Doors           50
Price           50
dtype: int64
```

Wonderful! We're making some progress.

Now let's fill the `Doors` column with `4` (the most common value), this is the same as filling it with the median (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.median.html) or mode (https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.mode.html) of the `Doors` column.

In [72]:
```
# Find the most common value of the Doors column
car_sales_missing["Doors"].value_counts()
```

Out[72]:
```
Doors
4.0    811
5.0     75
3.0     64
Name: count, dtype: int64
```

In [73]:
```
# Fill the Doors column with the most common value
car_sales_missing["Doors"].fillna(value=4, inplace=True)
```

```
/tmp/ipykernel_15677/1671259165.py:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Se
ies through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on
hich we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)
or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.


  car_sales_missing["Doors"].fillna(value=4, inplace=True)
```

Next, we'll fill the `Odometer (KM)` column with the mean value of itself.

In [74]:
```python
# Fill the Odometer (KM) column
car_sales_missing["Odometer (KM)"].fillna(value=car_sales_missing["Odometer (KM)"].mean(), inplace=True)
```

```
/tmp/ipykernel_15677/2625164921.py:2: FutureWarning: A value is trying to be set on a copy of a DataFrame or Se
ies through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on
hich we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)
or df[col] = df[col].method(value) instead, to perform the operation inplace on the original object.


  car_sales_missing["Odometer (KM)"].fillna(value=car_sales_missing["Odometer (KM)"].mean(), inplace=True)
```

How many missing values do we have now?

In [75]:
```python
# Check the number of missing values
car_sales_missing.isna().sum()
```

Out[75]:
```
Make              0
Colour            0
Odometer (KM)     0
Doors             0
Price            50
dtype: int64
```

Woohoo! That's looking a lot better.

Finally, we can remove the rows which are missing the target value `Price` .

> **Note:** Another option would be to impute the `Price` value with the mean or median or some other calculated value (such as by using similar cars to estimate the price), however, to keep things simple and prevent introducing too many fake labels to the data, we'll remove the samples missing a `Price` value.

In [76]:
```python
# Remove rows with missing Price labels
car_sales_missing.dropna(inplace=True)
```

That should be no more missing values!

In [77]:
```python
# Check the number of missing values
car_sales_missing.isna().sum()
```

Out[77]:
```
Make             0
Colour           0
Odometer (KM)    0
Doors            0
Price            0
dtype: int64
```

Since we removed samples missing a `Price` value, there's now less overall samples but none of them have missing values.

In [78]:
```python
# Check the number of total samples (previously was 1000)
len(car_sales_missing)
```

Out[78]:
```
950
```

Can we fit a model now?

Let's try!

First we'll create the features and labels.

Then we'll convert categorical variables into numbers via one-hot encoding.

Then we'll split the data into training and test sets just like before.

Finally, we'll try to fit a `RandomForestRegressor()` model to the newly filled data.

In [79]:
```
# Create features
X_missing = car_sales_missing.drop("Price", axis=1)
print(f"Number of missing X values:\n{X_missing.isna().sum()}")

# Create labels
y_missing = car_sales_missing["Price"]
print(f"Number of missing y values: {y_missing.isna().sum()}")
```

```
Number of missing X values:
Make             0
Colour           0
Odometer (KM)    0
Doors            0
dtype: int64
Number of missing y values: 0
```

In [80]:
```
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

categorical_features = ["Make", "Colour", "Doors"]

one_hot = OneHotEncoder()

transformer = ColumnTransformer([("one_hot",
                                  one_hot,
                                  categorical_features)],
                                remainder="passthrough",
                                sparse_threshold=0) # return a sparse matrix or not

transformed_X_missing = transformer.fit_transform(X_missing)
transformed_X_missing
```

Out[80]:
```
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 [81]:
```
# Split data into training and test sets
np.random.seed(42)
X_train, X_test, y_train, y_test = train_test_split(transformed_X_missing,
                                                    y_missing,
                                                    test_size=0.2)

# Fit and score a model
model = RandomForestRegressor()
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

Out[81]:
```
0.22011714008302485
```

Fantastic!!!

Looks like filling the missing values with pandas worked!

Our model can be fit to the data without issues.

1.2.2 Filling missing data and transforming categorical data with Scikit-Learn

Now we've filled the missing columns using pandas functions, you might be thinking, "Why pandas? I thought this was a Scikit-Learn introduction?".

Not to worry, Scikit-Learn provides a class called `sklearn.impute.SimpleImputer()` (https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html) which allows us to do a similar thing.

`SimpleImputer()` transforms data by filling missing values with a given `strategy` parameter.

And we can use it to fill the missing values in our DataFrame as above.

At the moment, our dataframe has no mising values.

In [82]:
```
car_sales_missing.isna().sum()
```

Out[82]:
```
Make             0
Colour           0
Odometer (KM)    0
Doors            0
Price            0
dtype: int64
```

Let's reimport it so it has missing values and we can fill them with Scikit-Learn.

In [83]:
```
# Reimport the DataFrame
car_sales_missing = pd.read_csv("/home/smayan/Desktop/AI-ML-DS/ZTM/Scikit-Learn/car-sales-extended-missing-data
car_sales_missing.isna().sum()
```

Out[83]:
```
Make             49
Colour           50
Odometer (KM)    50
Doors            50
Price            50
dtype: int64
```

To begin, we'll remove the rows which are missing a `Price` value.

In [84]:
```
# Drop the rows with missing in the Price column
car_sales_missing.dropna(subset=["Price"], inplace=True)
```

Now there are no rows missing a `Price` value.

In [85]:
```
car_sales_missing.isna().sum()
```

Out[85]:
```
Make             47
Colour           46
Odometer (KM)    48
Doors            47
Price             0
dtype: int64
```

Since we don't have to fill any `Price` values, let's split our data into features (`X`) and labels (`y`).

We'll also split the data into training and test sets.

In [86]:
```
# Split into X and y
X = car_sales_missing.drop("Price", axis=1)
y = car_sales_missing["Price"]

# Split data into train and test
np.random.seed(42)
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.2)
```

> **Note:** We've split the data into train & test sets here first to perform filling missing values on them separately. This is best practice as the test set is supposed to emulate data the model has never seen before. For categorical variables, it's generally okay to fill values across the whole dataset. However, for numerical vairables, you should **only fill values on the test set that have been computed from the training set**.

Training and test sets created!

Let's now setup a few instances of `SimpleImputer()` to fill various missing values.

We'll use the following strategies and fill values:

- For categorical columns (`Make` , `Colour`), `strategy="constant"` , `fill_value="missing"` (fill the missing samples with a consistent value of `"missing"` .
- For the `Door` column, `strategy="constant"` , `fill_value=4` (fill the missing samples with a consistent value of `4`).
- For the numerical column (`Odometer (KM)`), `strategy="mean"` (fill the missing samples with the mean of the target column).
  - **Note:** There are more `strategy` and fill options in the `SimpleImputer()` documentation (https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html).

In [87]:
```
from sklearn.impute import SimpleImputer

# Create categorical variable imputer
cat_imputer = SimpleImputer(strategy="constant", fill_value="missing")

# Create Door column imputer
door_imputer = SimpleImputer(strategy="constant", fill_value=4)

# Create Odometer (KM) column imputer
num_imputer = SimpleImputer(strategy="mean")
```

Imputers created!

Now let's define which columns we'd like to impute on.

Why?

Because we'll need these shortly (I'll explain in the next text cell).

In [88]:
```
# Define different column features
categorical_features = ["Make", "Colour"]
door_feature = ["Doors"]
numerical_feature = ["Odometer (KM)"]
```

Columns defined!

Now how might we transform our columns?

Hint: we can use the `sklearn.compose.ColumnTransformer` (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html) class from Scikit-Learn, in a similar way to what we did before to get our data to all numeric values.

That's the reason we defined the columns we'd like to transform.

So we can use the `ColumnTransformer()` class.

`ColumnTransformer()` takes as input a list of tuples in the form `(name_of_transform, transformer_to_use, columns_to_transform)` specifying which columns to transform and how to transform them.

For example:

```
imputer = ColumnTransformer([
    ("cat_imputer", cat_imputer, categorical_features)
])
```

In this case, the variables in the tuple are:

- `name_of_transform = "cat_imputer"`
- `transformer_to_use = cat_imputer` (the instance of `SimpleImputer()` we defined above)
- `columns_to_transform = categorical_features` (the list of categorical features we defined above).

Let's exapnd upon this by extending the example.

In [89]:
```
from sklearn.compose import ColumnTransformer

# Create series of column transforms to perform
imputer = ColumnTransformer([
    ("cat_imputer", cat_imputer, categorical_features),
    ("door_imputer", door_imputer, door_feature),
    ("num_imputer", num_imputer, numerical_feature)])
```

Nice!

The next step is to fit our `ColumnTransformer()` instance (`imputer`) to the training data and transform the testing data.

In other words we want to:

1. Learn the imputation values from the training set.
2. Fill the missing values in the training set with the values learned in 1.
3. Fill the missing values in the testing set with the values learned in 1.

Why this way?

In our case, we're not calculating many variables (except the mean of the `Odometer (KM)` column), however, remember that the test set should always remain as unseen data.

So **when filling values in the test set, they should only be with values calculated or imputed from the training set**s.

We can achieve steps 1 & 2 simultaneously with the `ColumnTransformer.fit_transform()` (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html#sklearn.compose.ColumnTransformer.fit_transform) method (`fit` = find the values to fill, `transform` = fill them).

And then we can perform step 3 with the `ColumnTransformer.transform()` (https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html#sklearn.compose.ColumnTransformer.transform) method (we only want to transform the test set, not learn different values to fill).

In [90]:
```
# Find values to fill and transform training data
filled_X_train = imputer.fit_transform(X_train)

# Fill values in to the test set with values learned from the training set
filled_X_test = imputer.transform(X_test)

# Check filled X_train
filled_X_train
```

Out[90]:
```
array([['Honda', 'White', 4.0, 71934.0],
       ['Toyota', 'Red', 4.0, 162665.0],
       ['Honda', 'White', 4.0, 42844.0],
       ...,
       ['Toyota', 'White', 4.0, 196225.0],
       ['Honda', 'Blue', 4.0, 133117.0],
       ['Honda', 'missing', 4.0, 150582.0]], dtype=object)
```

Wonderful!

Let's now turn our `filled_X_train` and `filled_X_test` arrays into DataFrames to inspect their missing values.

In [91]:
```
# Get our transformed data array's back into DataFrame's
filled_X_train_df = pd.DataFrame(filled_X_train,
                                 columns=["Make", "Colour", "Doors", "Odometer (KM)"])

filled_X_test_df = pd.DataFrame(filled_X_test,
                                columns=["Make", "Colour", "Doors", "Odometer (KM)"])

# Check missing data in training set
filled_X_train_df.isna().sum()
```

Out[91]:
```
Make             0
Colour           0
Doors            0
Odometer (KM)    0
dtype: int64
```

And is there any missing data in the test set?

In [92]:
```
# Check missing data in the testing set
filled_X_test_df.isna().sum()
```

Out[92]:
```
Make             0
Colour           0
Doors            0
Odometer (KM)    0
dtype: int64
```

What about the original?

In [93]:
```
# Check to see the original... still missing values
car_sales_missing.isna().sum()
```

Out[93]:
```
Make             47
Colour           46
Odometer (KM)    48
Doors            47
Price             0
dtype: int64
```

Perfect!

No more missing values!

But wait...

Is our data all numerical?

In [94]:
```
filled_X_train_df.head()
```

Out[94]:

	Make	Colour	Doors	Odometer (KM)
0	Honda	White	4.0	71934.0
1	Toyota	Red	4.0	162665.0
2	Honda	White	4.0	42844.0
3	Honda	White	4.0	195829.0
4	Honda	Blue	4.0	219217.0

Ahh... looks like our `Make` and `Colour` columns are still strings.

Let's one-hot encode them along with the `Doors` column to make sure they're numerical, just as we did previously.

In [95]:
```
# Now let's one hot encode the features with the same code as before
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

categorical_features = ["Make", "Colour", "Doors"]

one_hot = OneHotEncoder()

transformer = ColumnTransformer([("one_hot",
                                  one_hot,
                                  categorical_features)],
                                remainder="passthrough",
                                sparse_threshold=0) # return a sparse matrix or not

# Fill train and test values separately
transformed_X_train = transformer.fit_transform(filled_X_train_df)
transformed_X_test = transformer.transform(filled_X_test_df)

# Check transformed and filled X_train
transformed_X_train
```

Out[95]:
```
array([[0.0, 1.0, 0.0, ..., 1.0, 0.0, 71934.0],
       [0.0, 0.0, 0.0, ..., 1.0, 0.0, 162665.0],
       [0.0, 1.0, 0.0, ..., 1.0, 0.0, 42844.0],
       ...,
       [0.0, 0.0, 0.0, ..., 1.0, 0.0, 196225.0],
       [0.0, 1.0, 0.0, ..., 1.0, 0.0, 133117.0],
       [0.0, 1.0, 0.0, ..., 1.0, 0.0, 150582.0]], dtype=object)
```

Nice!

Now our data is:

1. All numerical
2. No missing values

Let's try and fit a model!

In [96]:
```
# Now we've transformed X, let's see if we can fit a model
np.random.seed(42)
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor()

# Make sure to use the transformed data (filled and one-hot encoded X data)
model.fit(transformed_X_train, y_train)
model.score(transformed_X_test, y_test)
```

Out[96]:
```
0.21229043336119102
```

You might have noticed this result is slightly different to before.

Why do you think this is?

It's because we've created our training and testing sets differently.

We split the data into training and test sets *before* filling the missing values.

Previously, we did the reverse, filled missing values *before* splitting the data into training and test sets.

Doing this can lead to information from the training set leaking into the testing set.

Remember, one of the most important concepts in machine learning is making sure your model doesn't see *any* testing data before evaluation.

We'll keep practicing but for now, some of the main takeaways are:

- Keep your training and test sets separate.
- Most datasets you come across won't be in a form ready to immediately start using them with machine learning models. And some may take more preparation than others to get ready to use.
- For most machine learning models, your data has to be numerical. This will involve converting whatever you're working with into numbers. This process is often referred to as **feature engineering** or **feature encoding**.
- Some machine learning models aren't compatible with missing data. The process of filling missing data is referred to as **data imputation**.

2. Choosing the right estimator/algorithm for your problem

Once you've got your data ready, the next step is to choose an appropriate machine learning algorithm or model to find patterns in your data.

Some things to note:

- Scikit-Learn refers to machine learning models and algorithms as estimators.
- **Classification problem** - predicting a category (heart disease or not).
  - Sometimes you'll see `clf` (short for classifier) used as a classification estimator instance's variable name.
- **Regression problem** - predicting a number (selling price of a car).
- **Unsupervised problem (data with no labels)** - clustering (grouping unlabelled samples with other similar unlabelled samples).

If you know what kind of problem you're working with, one of the next places you should look at is the Scikit-Learn algorithm cheatsheet (https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html).

This cheatsheet gives you a bit of an insight into the algorithm you might want to use for the problem you're working on.

It's important to remember, you don't have to explicitly know what each algorithm is doing on the inside to start using them.

If you start to apply different algorithms but they don't seem to be working (not performing as well as you'd like), that's when you'd start to look deeper into each one.

Let's check out the cheatsheet and follow it for some of the problems we're working on.

You can see it's split into four main categories. Regression, classification, clustering and dimensionality reduction. Each has their own different purpose but the Scikit-Learn team has designed the library so the workflows for each are relatively similar.

2.1 Picking a machine learning model for a regression problem

Let's start with a regression problem (trying to predict a number). We'll use the California Housing dataset (https://scikit-learn.org/stable/datasets/real_world.html#california-housing-dataset) built into Scikit-Learn's `datasets` module.

The goal of the California Housing dataset is to predict a given district's median house value (in hundreds of thousands of dollars) on things like the age of the home, the number of rooms, the number of bedrooms, number of people living the home and more.

In [97]:
```
# Get California Housing dataset
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
housing; # gets downloaded as dictionary
```

Since it's in a dictionary, let's turn it into a DataFrame so we can inspect it better.

In [98]:
```
housing_df = pd.DataFrame(housing["data"], columns=housing["feature_names"])
housing_df["target"] = pd.Series(housing["target"])
housing_df.head()
```

Out[98]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	target
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23	4.526
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22	3.585
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24	3.521
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25	3.413
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25	3.422

In [99]:
```
# How many samples?
len(housing_df)
```

Out[99]: 20640

Beautiful, our goal here is to use the feature columns, such as:

- `MedInc` - median income in block group
- `HouseAge` - median house age in block group
- `AveRooms` - average number of rooms per household
- `AveBedrms` - average number of bedrooms per household

To predict the `target` column which expresses the median house value for specfici California districts in hundreds of thousands of dollars (e.g. 4.526 = $452,600).

In essence, each row is a different district in California (the data) and we're trying to build a model to predict the median house value in that distract (the target/label) given a series of attributes about the houses in that district.

Since we have data and labels, this is a supervised learning problem.

And since we're trying to predict a number, it's a regression problem.

Knowing these two things, how do they line up on the Scikit-Learn machine learning algorithm cheat-sheet?

Following the map through, knowing what we know, it suggests we try `RidgeRegression` (https://scikit-learn.org/stable/modules/linear_model.html#ridge-regression). Let's chek it out.

In [100…
```
# Import the Ridge model class from the linear_model module
from sklearn.linear_model import Ridge

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

# Split the data into features (X) and labels (y)
X = housing_df.drop("target", axis=1)
y = housing_df["target"]
```

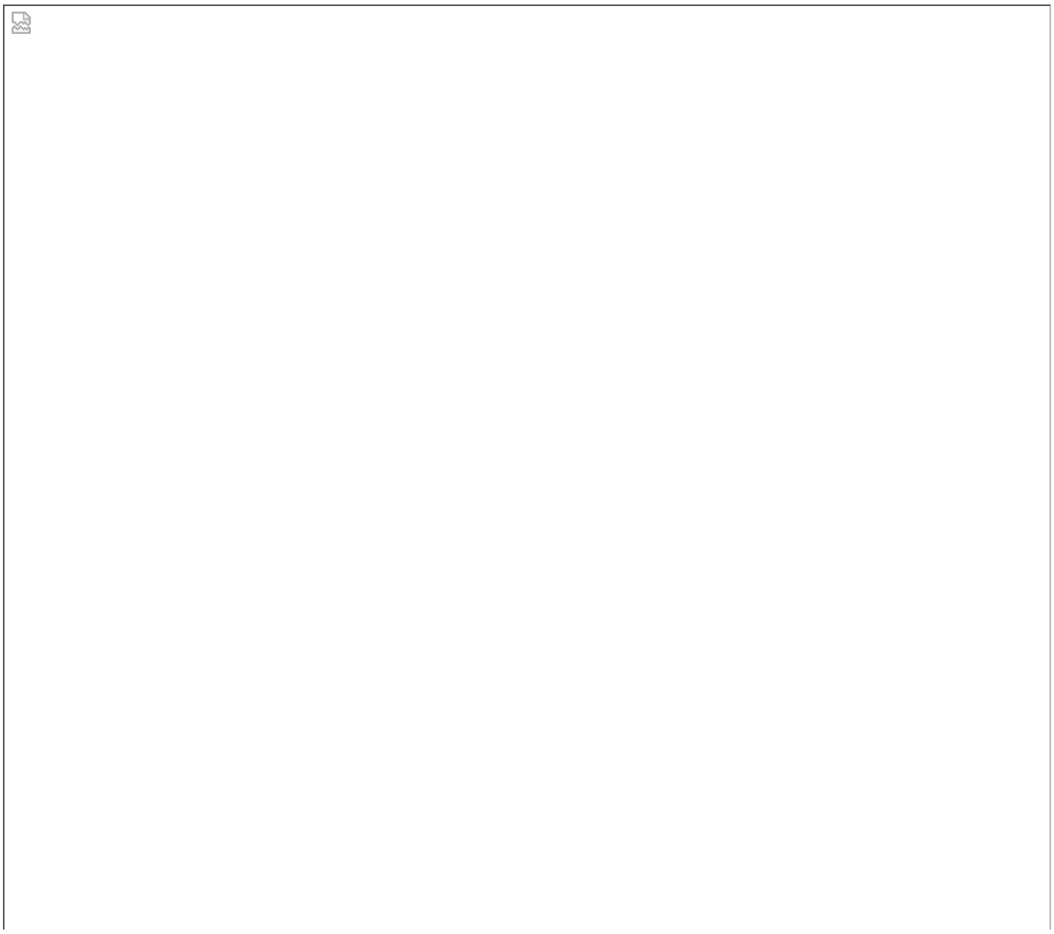
Out[100]: 0.5758549611440126

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

# Institate and fit the model (on the training set)
model = Ridge()
model.fit(X_train, y_train)

# Check the score of the model (on the test set)
# The default score() metirc of regression aglorithms is R^2
model.score(X_test, y_test)
```

What if `RidgeRegression` didn't work? Or what if we wanted to improve our results?



Following the diagram, the next step would be to try `EnsembleRegressors` (https://scikit-learn.org/stable/modules/ensemble.html).

Ensemble is another word for multiple models put together to make a decision.

One of the most common and useful ensemble methods is the Random Forest (https://scikit-learn.org/stable/modules/ensemble.html#forest). Known for its fast training and prediction times and adaptibility to different problems.

The basic premise of the Random Forest is to combine a number of different decision trees, each one random from the other and make a prediction on a sample by averaging the result of each decision tree.

An in-depth discussion of the Random Forest algorithm is beyond the scope of this notebook but if you're interested in learning more, An Implementation and Explanation of the Random Forest in Python (https://towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76) by Will Koehrsen is a great read.

Since we're working with regression, we'll use Scikit-Learn's `RandomForestRegressor` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html).

We can use the exact same workflow as above. Except for changing the model.

In [101…
```
# Import the RandomForestRegressor model class from the ensemble module
from sklearn.ensemble import RandomForestRegressor

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

# Split the data into features (X) and labels (y)
X = housing_df.drop("target", axis=1)
y = housing_df["target"]

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

# Institate and fit the model (on the training set)
model = RandomForestRegressor()
model.fit(X_train, y_train)

# Check the score of the model (on the test set)
# The default score metric of regression algorithms is R^2
model.score(X_test, y_test)
```

Out[101]:
```
0.8066196804802649
```

Woah!

We get a good boost in score on the test set by changing the model.

This is another incredibly important concept in machine learning, if at first something doesn't achieve what you'd like, *experiment, experiment, experiment!*

At first, the Scikit-Learn algorithm diagram can seem confusing.

But once you get a little practice applying different models to different problems, you'll start to pick up which sorts of algorithms do better with different types of data.

2.2 Picking a machine learning model for a classification problem

Now, let's check out the choosing process for a classification problem.

Say you were trying to predict whether or not a patient had heart disease based on their medical records.

The dataset in `../data/heart-disease.csv` contains data for just that problem.

In [102…
```
heart_disease = pd.read_csv("/home/smayan/Desktop/AI-ML-DS/sample_project/heart-disease.csv")
heart_disease.head()
```

Out[102]:

	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 [103…
```
# How many samples are there?
len(heart_disease)
```

Out[103]: 303

Similar to the California Housing dataset, here we want to use all of the available data to predict the target column (1 for if a patient has heart disease and 0 for if they don't).

So what do we know?

We've got 303 samples (1 row = 1 sample) and we're trying to predict whether or not a patient has heart disease.

Because we're trying to predict whether each sample is one thing or another, we've got a classification problem.

Let's see how it lines up with our Scikit-Learn algorithm cheat-sheet (https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html).

Following the cheat-sheet we end up at `LinearSVC` (https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) which stands for Linear Support Vector Classifier. Let's try it on our data.

In [104…

```
# Import LinearSVC from the svm module
from sklearn.svm import LinearSVC

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

# Split the data into X (features/data) and y (target/labels)
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

# Instantiate and fit the model (on the training set)
clf = LinearSVC(max_iter=1000, # iterations on the data, 1000 is the default
                dual="auto") # dual="auto" chooses best parameters for the model automatically
clf.fit(X_train, y_train)
```

```
# Check the score of the model (on the test set)
clf.score(X_test, y_test)
```

Out[104]:

```
0.8688524590163934
```

Straight out of the box (with no tuning or improvements) our model achieves over 85% accruacy!

Although this is a sensational result to begin with, let's check out the diagram and see what other models we might use.

Following the path (and skipping a few, don't worry, we'll get to this) we come up to `EnsembleMethods` (https://scikit-learn.org/stable/modules/ensemble.html) again.

Except this time, we'll be looking at ensemble classifiers instead of regressors.

Remember our `RandomForestRegressor` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html) from above?

We'll it has a dance partner, `RandomForestClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html) which is an ensemble based machine model learning model for classification.

You might be able to guess what we can use it for (hint: classification problems).

Let's try!

In [105…

```
# Import the RandomForestClassifier model class from the ensemble module
from sklearn.ensemble import RandomForestClassifier

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

# Split the data into X (features/data) and y (target/labels)
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

# Instantiate and fit the model (on the training set)
```

```python
# Instantiate and fit the model (on the training set)
clf = RandomForestClassifier(n_estimators=100) # 100 is the default, but you could try 1000 and see what happen
clf.fit(X_train, y_train)

# Check the score of the model (on the test set)
clf.score(X_test, y_test)
```

```
Out[105]: 0.8524590163934426
```

Hmmm, it looks like the default hyperparameters of `RandomForestClassifier` don't perform as well as `LinearSVC` .

Other than trying another classification model, we could start to run experiments to try and improve these models via hyperparameter tuning (http://scikit-learn.org/stable/modules/grid_search.html).

Hyperparameter tuning is fancy term for adjusting some settings on a model to try and make it better.

It usually happens once you've found a decent baseline model that you'd like to improve upon.

In this case, we could take either the `RandomForestClassifier` or the `LinearSVC` and try and improve it with hyperparameter tuning (which we'll see later on).

For example, you could try and take the `n_estimators` parameter (the number of trees in the forest) of `RandomForestClassifier` and change it from `100` (default) to `1000` and see what happens.

What about the other models?

Looking at the Scikit-Learn aglorithm cheat-sheet and the examples above, you may have noticed we've skipped a few models.

Why?

The first reason is time.

Covering every single one would take a fair bit longer than what we've done here. And the second one is the effectiveness of ensemble methods.

A little tidbit for modelling in machine learning is:

- If you have structured data (tables, spreadsheets or dataframes), use ensemble methods, such as, a Random Forest.
- If you have unstructured data (text, images, audio, things not in tables), use deep learning or transfer learning (see the ZTM TensorFlow (https://dbourke.link/ZTMTFcourse?ref=mrdbourke.com) and PyTorch (https://dbourke.link/ZTMPyTorch?ref=mrdbourke.com) courses for more on deep learning).

For this notebook, we're focused on structured data, which is why the Random Forest has been our model of choice.

If you'd like to learn more about the Random Forest and why it's the war horse of machine learning, check out these resources:

- Random Forest Wikipedia (https://en.wikipedia.org/wiki/Random_forest)
- An Implementation and Explanation of the Random Forest in Python (https://towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76) by Will Koehrsen

Experiment until something works

The beautiful thing is, the way the Scikit-Learn API is designed, once you know the way with one model, using another is much the same.

And since a big part of being a machine learning engineer or data scientist is experimenting, you might want to try out some of the other models on the cheat-sheet and see how you go. The more you can reduce the time between experiments, the better.

3. Fit the model to data and using it to make predictions

Now you've chosen a model, the next step is to have it learn from the data so it can be used for predictions in the future.

If you've followed through, you've seen a few examples of this already.

3.1 Fitting a model to data

In Scikit-Learn, the process of having a machine learning model learn patterns from a dataset involves calling the `fit()` method and passing it data, such as, `fit(X, y)`.

Where `X` is a feature array and `y` is a target array.

Other names for `X` include:

- Data
- Feature variables
- Features

Other names for `y` include:

- Labels
- Target variable

For supervised learning there is usually an `X` and `y`.

For unsupervised learning, there's no `y` (no labels).

Let's revisit the example of using patient data (`X`) to predict whether or not they have heart disease (`y`).

In [106…
```
# Import the RandomForestClassifier model class from the ensemble module
from sklearn.ensemble import RandomForestClassifier

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

# Split the data into X (features/data) and y (target/labels)
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

# Instantiate the model (on the training set)
clf = RandomForestClassifier(n_estimators=100)

# Call the fit method on the model and pass it training data
clf.fit(X_train, y_train)

# Check the score of the model (on the test set)
clf.score(X_test, y_test)
```

Out[106]:
```
0.8524590163934426
```

What's happening here?

Calling the `fit()` method will cause the machine learning algorithm to attempt to find patterns between `X` and `y`. Or if there's no `y`, it'll only find the patterns within `X`.

Let's see `X`.

In [107…
```
X.head()
```

Out[107]:

	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

And `y`.

In [108…
```
y.head()
```

Out[108]:
```
0    1
1    1
2    1
3    1
4    1
Name: target, dtype: int64
```

Passing X and y to fit() will cause the model to go through all of the examples in X (data) and see what their corresponding y (label) is.

How the model does this is different depending on the model you use.

Explaining the details of each would take an entire textbook.

For now, you could imagine it similar to how you would figure out patterns if you had enough time.

You'd look at the feature variables, X , the age , sex , chol (cholesterol) and see what different values led to the labels, y , 1 for heart disease, 0 for not heart disease.

This concept, regardless of the problem, is similar throughout all of machine learning.

During training (finding patterns in data):

A machine learning algorithm looks at a dataset, finds patterns, tries to use those patterns to predict something and corrects itself as best it can with the available data and labels. It stores these patterns for later use.

During testing or in production (using learned patterns):

A machine learning algorithm uses the patterns its previously learned in a dataset to make a prediction on some unseen data.

3.2 Making predictions using a machine learning model

Now we've got a trained model, one which has hoepfully learned patterns in the data, you'll want to use it to make predictions.

Scikit-Learn enables this in several ways.

Two of the most common and useful are predict() (https://github.com/scikit-learn/scikit-learn/blob/5f3c3f037/sklearn/multiclass.py#L299) and predict_proba() (https://github.com/scikit-learn/scikit-learn/blob/5f3c3f037/sklearn/linear_model/_logistic.py#L1617).

Let's see them in action.

In [109…
```
# Use a trained model to make predictions
clf.predict(X_test)
```

Out[109]:
```
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])
```

Given data in the form of X , the predict() function returns labels in the form of y .

> **Note:** For the predict() function to work, it must be passed X (data) in the same format the model was trained on. For example, if a model was trained on 10 features formatted in a certain way, predictions should be made on data with 10 features fortmatted in a certain way. Anything different and it will return an error.

It's standard practice to save these predictions to a variable named something like y_preds for later comparison to y_test or y_true (usually same as y_test just another name).

In [110…
```
# Compare predictions to truth
y_preds = clf.predict(X_test)
np.mean(y_preds == y_test)
```

Out[110]:
```
0.8524590163934426
```

Another way evaluating predictions (comparing them to the truth labels) is with Scikit-Learn's `sklearn.metrics` module (http://scikit-learn.org/stable/modules/model_evaluation.html).

Inside, you'll find method such as `accuracy_score()` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html), which is the default evaluation metric for classification problems.

In [111…
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_preds)
```

Out[111]:
```
0.8524590163934426
```

`predict_proba()` returns the probabilities (proba is short for probability) of a classification label.

In [112…
```
# Return probabilities rather than labels
clf.predict_proba(X_test[:5])
```

Out[112]:
```
array([[0.89, 0.11],
       [0.49, 0.51],
       [0.43, 0.57],
       [0.84, 0.16],
       [0.18, 0.82]])
```

Let's see the difference.

In [113…
```
# Return labels
clf.predict(X_test[:5])
```

Out[113]:
```
array([0, 1, 1, 0, 1])
```

`predict_proba()` returns an array of five arrays each containing two values.

Each number is the probability of a label given a sample.

In [114…
```
# Find prediction probabilities for 1 sample
clf.predict_proba(X_test[:1])
```

Out[114]:
```
array([[0.89, 0.11]])
```

This output means for the sample `X_test[:1]`, the model is predicting label 0 (index 0) with a probability score of 0.9.

Because the highest probability score is at index `0` (and it's over 0.5), when using `predict()`, a label of `0` is assigned.

In [115…
```
# Return the label for 1 sample
clf.predict(X_test[:1])
```

Out[115]:
```
array([0])
```

Where does 0.5 come from?

Because our problem is a binary classification task (heart disease or not heart disease), predicting a label with 0.5 probability every time would be the same as a coin toss (guessing 50/50 every time).

Therefore, once the prediction probability of a sample passes 0.5 for a certain label, it's assigned that label.

`predict()` can also be used for regression models.

In [116…
```python
# Import the RandomForestRegressor model class from the ensemble module
from sklearn.ensemble import RandomForestRegressor

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

# Split the data into features (X) and labels (y)
X = housing_df.drop("target", axis=1)
y = housing_df["target"]

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

# Institate and fit the model (on the training set)
model = RandomForestRegressor()
model.fit(X_train, y_train)

# Make predictions
y_preds = model.predict(X_test)
```

Now we can evaluate our regression model by using `sklearn.metrics.mean_absolute_error` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html) which returns the average error across all samples.

In [117…
```python
# Compare the predictions to the truth
from sklearn.metrics import mean_absolute_error
mean_absolute_error(y_test, y_preds)
```

Out[117]: `0.3265721842781009`

Now we've seen how to get a model how to find patterns in data using the `fit()` function and make predictions using what its learned using the `predict()` and `predict_proba()` functions, it's time to evaluate those predictions.

4. Evaluating a model

Once you've trained a model, you'll want a way to measure how trustworthy its predictions are.

Across the board, the main idea of evaluating a model is to **compare the model's predictions to what they should've ideally been** (the truth labels).

Scikit-Learn implements 3 different methods of evaluating models.

1. The `score()` method. Calling `score()` on a model instance will return a metric assosciated with the type of model you're using. The metric depends on which model you're using.
2. The `scoring` parameter. This parameter can be passed to methods such as `cross_val_score()` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html#sklearn.model_selection.cross_val_score) or `GridSearchCV()` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) to tell Scikit-Learn to use a specific type of scoring metric.
3. Problem-specific metric functions available in `sklearn.metrics` (https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics). Similar to how the `scoring` parameter can be passed different scoring functions, Scikit-Learn implements these as stand alone functions.

The scoring function you use will also depend on the problem you're working on.

Classification problems have different evaluation metrics and scoring functions to regression problems.

Let's look at some examples.

4.1 General model evaluation with `score()`

If we bring down the code from our previous classification problem (building a classifier to predict whether or not someone has heart disease based on their medical records).

We can see the `score()` method come into play.

In [118…
```
# Import the RandomForestClassifier model class from the ensemble module
from sklearn.ensemble import RandomForestClassifier

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

# Split the data into X (features/data) and y (target/labels)
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

# Instantiate the model (on the training set)
clf = RandomForestClassifier(n_estimators=100)

# Call the fit method on the model and pass it training data
clf.fit(X_train, y_train);
```

Once the model has been fit on the training data (`X_train` , `y_train`), we can call the `score()` method on it and evaluate our model on the test data, data the model has never seen before (`X_test` , `y_test`).

In [119…
```
# Check the score of the model (on the test set)
clf.score(X_test, y_test)
```

Out[119]:
```
0.8524590163934426
```

Each model in Scikit-Learn implements a default metric for `score()` which is suitable for the problem.

For example:

- Classifier models generally use `metrics.accuracy_score()` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html#sklearn.metrics.accuracy_score) as the default `score()` metric.
- Regression models generally use `metrics.r2_score` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html) as the default `score()` metric.
- There many more classification (https://scikit-learn.org/stable/modules/classes.html#classification-metrics) and regression (https://scikit-learn.org/stable/modules/classes.html#regression-metrics) specific metrics implemented in `sklearn.metrics` .

Because `clf` is an instance of `RandomForestClassifier` , the `score()` method uses mean accuracy as its score method.

You can find this by pressing **SHIFT + TAB** (inside a Jupyter Notebook, may be different elsewhere) within the brackets of `score()` when called on a model instance.

Behind the scenes, `score()` makes predictions on `X_test` using the trained model and then compares those predictions to the actual labels `y_test` .

A classification model which predicts everything 100% correct would receive an accuracy score of 1.0 (or 100%).

Our model doesn't get everything correct, but at ~85% accuracy (0.85 * 100), it's still far better than guessing.

Let's do the same but with the regression code from above.

In [120…
```python
# Import the RandomForestRegressor model class from the ensemble module
from sklearn.ensemble import RandomForestRegressor

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

# Split the data into features (X) and labels (y)
X = housing_df.drop("target", axis=1)
y = housing_df["target"]

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

# Institate and fit the model (on the training set)
model = RandomForestRegressor()
model.fit(X_train, y_train);
```

Due to the consistent design of the Scikit-Learn library, we can call the same `score()` method on `model`.

In [121…
```python
# Check the score of the model (on the test set)
model.score(X_test, y_test)
```

Out[121]: 0.8066196804802649

Here, `model` is an instance of `RandomForestRegressor` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html).

And since it's a regression model, the default metric built into `score()` is the coefficient of determination or R^2 (pronounced R-sqaured).

Remember, you can find this by pressing **SHIFT + TAB** within the brackets of `score()` when called on a model instance.

The best possible value here is 1.0, this means the model predicts the target regression values exactly.

Calling the `score()` method on any model instance and passing it test data is a good quick way to see how your model is going.

However, when you get further into a problem, it's likely you'll want to start using more powerful metrics to evaluate your models performance.

4.2 Evaluating your models using the `scoring` parameter

The next step up from using `score()` is to use a custom `scoring` parameter with `cross_val_score()` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html#sklearn.model_selection.cross_val_score) or `GridSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html).

As you may have guessed, the `scoring` parameter you set will be different depending on the problem you're working on.

We'll see some specific examples of different parameters in a moment but first let's check out `cross_val_score()`.

To do so, we'll copy the heart disease classification code from above and then add another line at the top.

In [122…
```python
# Import cross_val_score from the model_selection module
from sklearn.model_selection import cross_val_score

# Import the RandomForestClassifier model class from the ensemble module
from sklearn.ensemble import RandomForestClassifier

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

# Split the data into X (features/data) and y (target/labels)
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

# Instantiate the model (on the training set)
clf = RandomForestClassifier(n_estimators=100)

# Call the fit method on the model and pass it training data
clf.fit(X_train, y_train);
```

Using `cross_val_score()` is slightly different to `score()` .

Let's see a code example first and then we'll go through the details.

In [123…
```python
# Using score()
clf.score(X_test, y_test)
```

Out[123]:
```
0.8524590163934426
```

In [124…
```python
# Using cross_val_score()
cross_val_score(clf, X, y, cv=5) # cv = number of splits to test (5 by default)
```

Out[124]:
```
array([0.81967213, 0.86885246, 0.81967213, 0.78333333, 0.76666667])
```

What's happening here?

The first difference you might notice is `cross_val_score()` returns an array where as `score()` only returns a single number.

`cross_val_score()` returns an array because of a parameter called `cv` , which stands for cross-validation.

When `cv` isn't set, `cross_val_score()` will return an array of 5 numbers by default (`cv=None` is the same as setting `cv=5`).

Remember, you can see the parameters of a function using **SHIFT + TAB** (inside a Jupyter Notebook) from within the brackets.

But wait, you might be thinking, what even is cross-validation?

A visual might be able to help.

We've dealt with Figure 1.0 before using `score(X_test, y_test)`.

But looking deeper into this, if a model is trained using the training data or 80% of samples, this means 20% of samples aren't used for the model to learn anything.

This also means depending on what 80% is used to train on and what 20% is used to evaluate the model, it may achieve a score which doesn't reflect the entire dataset.

For example, if a lot of easy examples are in the 80% training data, when it comes to test on the 20%, your model may perform poorly.

The same goes for the reverse.

Figure 2.0 shows 5-fold cross-validation, a method which tries to provide a solution to:

1. Not training on all the data (always keeping training and test sets separate).
2. Avoiding getting lucky scores on single splits of the data.

Instead of training only on 1 training split and evaluating on 1 testing split, 5-fold cross-validation does it 5 times.

On a different split each time, returning a score for each.

Why 5-fold?

The actual name of this setup K-fold cross-validation. Where K is an abitrary number. We've used 5 because it looks nice visually, and it is the default value in `sklearn.model_selection.cross_val_score` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html).

Figure 2.0 is what happens when we run the following.

In [125…
```
# 5-fold cross-validation
cross_val_score(clf, X, y, cv=5) # cv is equivalent to K
```

Out[125]:
```
array([0.83606557, 0.8852459 , 0.7704918 , 0.8       , 0.8       ])
```

Since we set `cv=5` (5-fold cross-validation), we get back 5 different scores instead of 1.

Taking the mean of this array gives us a more in-depth idea of how our model is performing by converting the 5 scores into one.

In [126…
```
np.random.seed(42)

# Single training and test split score
clf_single_score = clf.score(X_test, y_test)

# Take mean of 5-fold cross-validation
clf_cross_val_score = np.mean(cross_val_score(clf, X, y, cv=5))

clf_single_score, clf_cross_val_score
```

Out[126]:
```
(0.8524590163934426, 0.8248087431693989)
```

Notice, the average `cross_val_score()` is slightly lower than single value returned by `score()` .

In this case, if you were asked to report the accuracy of your model, even though it's lower, you'd prefer the cross-validated metric over the non-cross-validated metric.

Wait?

We haven't used the `scoring` parameter at all.

By default, it's set to `None` .

In [127…
```
cross_val_score(clf, X, y, cv=5, scoring=None) # default scoring value, this can be set to other scoring metric
```

Out[127]:
```
array([0.78688525, 0.86885246, 0.80327869, 0.78333333, 0.76666667])
```

> **Note:** If you notice different scores each time you call `cross_val_score` , this is because each data split is random every time. So the model may achieve higher/lower scores on different splits of the data. To get reproducible scores, you can set the random seed.

When `scoring` is set to `None` (by default), it uses the same metric as `score()` for whatever model is passed to `cross_val_score()` .

In this case, our model is `clf` which is an instance of `RandomForestClassifier` which uses mean accuracy as the default `score()` metric.

You can change the evaluation score `cross_val_score()` uses by changing the `scoring` parameter.

And as you might have guessed, different problems call for different evaluation scores.

The Scikit-Learn documentation (https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter) outlines a vast range of evaluation metrics for different problems but let's have a look at a few.

4.2.1 Classification model evaluation metrics

Four of the main evaluation metrics/methods you'll come across for classification models are:

1. Accuracy (https://developers.google.com/machine-learning/crash-course/classification/accuracy)
2. Area under ROC curve (https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc) (receiver operating characteristic curve)
3. Confusion matrix (https://en.wikipedia.org/wiki/Confusion_matrix)
4. Classification report (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)

Let's have a look at each of these. We'll bring down the classification code from above to go through some examples.

In [128…
```
# Import cross_val_score from the model_selection module
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

np.random.seed(42)

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)
clf.score(X_test, y_test)
```

Out[128]:
```
0.8524590163934426
```

Accuracy

Accuracy is the default metric for the `score()` function within each of Scikit-Learn's classifier models. And it's probably the metric you'll see most often used for classification problems.

However, we'll see in a second how it may not always be the best metric to use.

Scikit-Learn returns accuracy as a decimal but you can easily convert it to a percentage.

In [129…
```
# Accuracy as percentage
print(f"Heart Disease Classifier Accuracy: {clf.score(X_test, y_test) * 100:.2f}%")
```

```
Heart Disease Classifier Accuracy: 85.25%
```

Area Under Receiver Operating Characteristic (ROC) Curve

If this one sounds like a mouthful, its because reading the full name is.

It's usually referred to as AUC for Area Under Curve and the curve they're talking about is the Receiver Operating Characteristic or ROC for short.

So if hear someone talking about AUC or ROC, they're probably talking about what follows.

ROC curves are a comparison of true postive rate (tpr) versus false positive rate (fpr).

For clarity:

- True positive = model predicts 1 when truth is 1
- False positive = model predicts 1 when truth is 0
- True negative = model predicts 0 when truth is 0
- False negative = model predicts 0 when truth is 1

Now we know this, let's see one. Scikit-Learn lets you calculate the information required for a ROC curve using the `roc_curve` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html#sklearn.metrics.roc_curve) function.

In [130…
```
from sklearn.metrics import roc_curve

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

# Keep the probabilites of the positive class only
y_probs = y_probs[:, 1]

# Calculate fpr, tpr and thresholds
fpr, tpr, thresholds = roc_curve(y_test, y_probs)

# Check the false positive rate
fpr
```

```
Out[130]: array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.03448276, 0.03448276, 0.03448276, 0.03448276, 0.06896552,
       0.06896552, 0.10344828, 0.13793103, 0.13793103, 0.17241379,
       0.17241379, 0.27586207, 0.4137931 , 0.48275862, 0.55172414,
       0.65517241, 0.72413793, 0.72413793, 0.82758621, 1.        ])
```

Looking at these on their own doesn't make much sense. It's much easier to see their value visually.

Let's create a helper function to make a ROC curve given the false positive rates (`fpr`) and true positive rates (`tpr`).

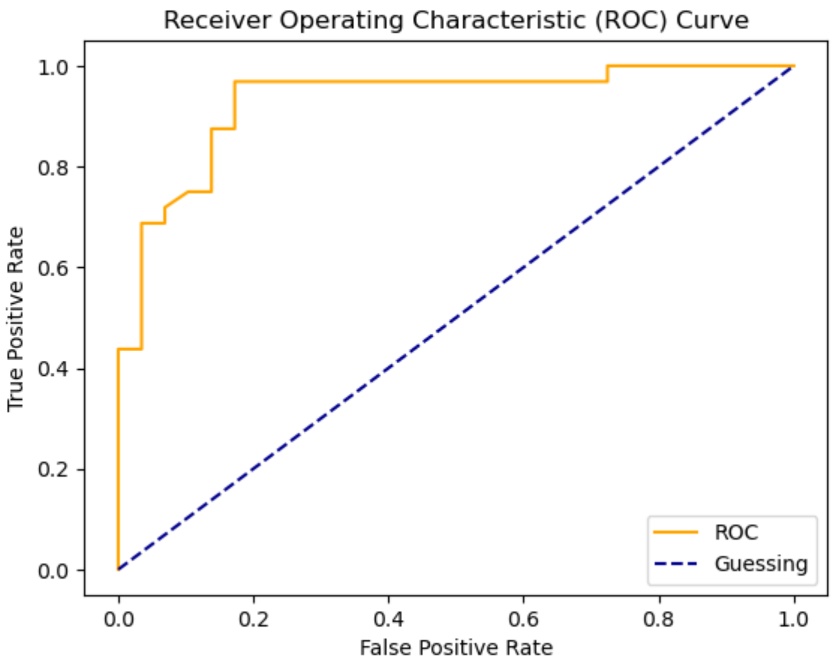
> **Note:** As of Scikit-Learn 1.2+, there is functionality of plotting a ROC curve. You can find this under `sklearn.metrics.RocCurveDisplay` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.RocCurveDisplay.html#sklearn-metrics-roccurvedisplay).

In [131…

```python
import matplotlib.pyplot as plt

def plot_roc_curve(fpr, tpr):
    """
    Plots a ROC curve given the false positve rate (fpr) and
    true postive rate (tpr) of a classifier.
    """
    # 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')
    # Customize the plot
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()

plot_roc_curve(fpr, tpr)
```



Looking at the plot for the first time, it might seem a bit confusing.

The main thing to take away here is our model is doing far better than guessing.

A metric you can use to quantify the ROC curve in a single number is AUC (Area Under Curve).

Scikit-Learn implements a function to caculate this called `sklearn.metrics.roc_auc_score` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html#sklearn.metrics.roc_auc_score).

The maximum ROC AUC score you can achieve is 1.0 and generally, the closer to 1.0, the better the model.

In [132…
```
from sklearn.metrics import roc_auc_score

roc_auc_score_value = roc_auc_score(y_test, y_probs)
roc_auc_score_value
```

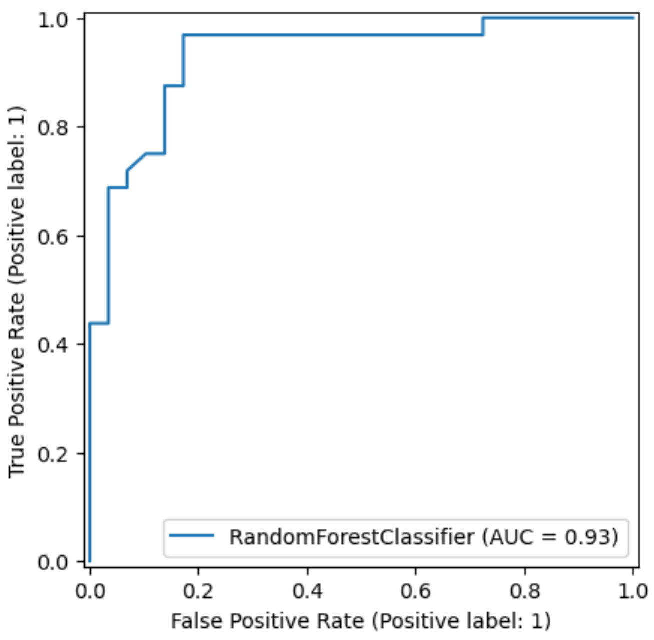
Out[132]: 0.9304956896551724

I'll let you in a secret...

Although it was good practice, we didn't actually need to create our own `plot_roc_curve` function.

Scikit-Learn allows us to plot a ROC curve directly from our estimator/model by using the class method `sklearn.metrics.RocCurveDisplay.from_estimator` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.RocCurveDisplay.html#sklearn.metrics.RocCurveDisplay.from_estimator) and passing it our `estimator`, `X_test` and `y_test`.

In [133…
```
from sklearn.metrics import RocCurveDisplay
roc_curve_display = RocCurveDisplay.from_estimator(estimator=clf,
                                                   X=X_test,
                                                   y=y_test)
```

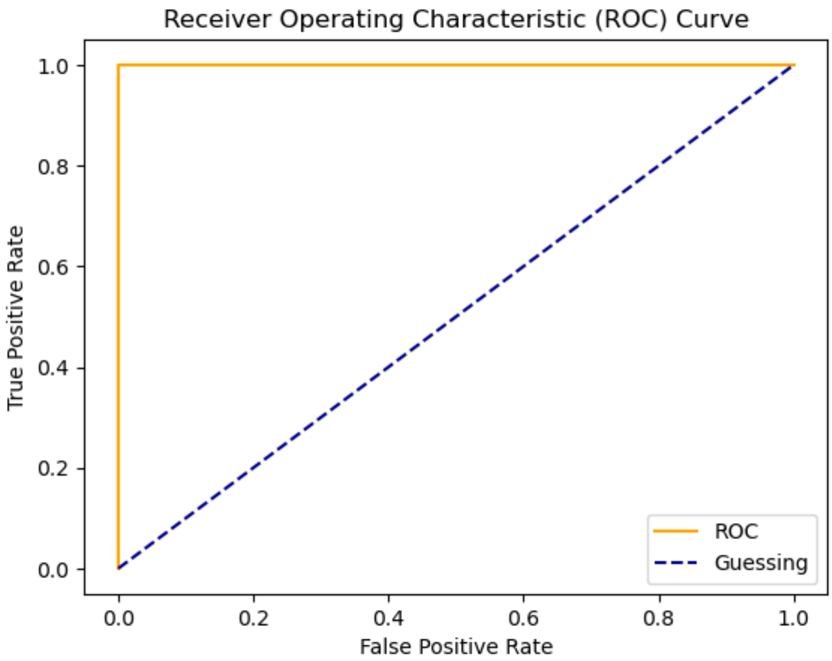


The most ideal position for a ROC curve to run along the top left corner of the plot.

This would mean the model predicts only true positives and no false positives. And would result in a ROC AUC score of 1.0.

You can see this by creating a ROC curve using only the `y_test` labels.

In [134…
```
# Plot perfect ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_test)
plot_roc_curve(fpr, tpr)
```



In [135…
```
# Perfect ROC AUC score
roc_auc_score(y_test, y_test)
```

Out[135]: 1.0

In reality, a perfect ROC curve is unlikely.

Confusion matrix

Another fantastic way to evaluate a classification model is by using a confusion matrix (https://en.wikipedia.org/wiki/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 an idea of where the model is getting confused.

In [136…
```
from sklearn.metrics import confusion_matrix

y_preds = clf.predict(X_test)

confusion_matrix(y_test, y_preds)
```

Out[136]:
```
array([[24,  5],
       [ 4, 28]])
```

Again, this is probably easier visualized.

One way to do it is with `pd.crosstab()`.

In [137…
```
pd.crosstab(y_test,
            y_preds,
            rownames=["Actual Label"],
            colnames=["Predicted Label"])
```

Out[137]:

Predicted Label	0	1
Actual Label		
0	24	5
1	4	28

Creating a confusion matrix using Scikit-Learn

Scikit-Learn has multiple different implementations of plotting confusion matrices:

1. `sklearn.metrics.ConfusionMatrixDisplay.from_estimator(estimator, X, y)` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ConfusionMatrixDisplay.html#sklearn.metrics.ConfusionMatrixDisplay.from_estimator) - this takes a fitted estimator (like our `clf` model), features (`X`) and labels (`y`), it then uses the trained estimator to make predictions on `X` and compares the predictions to `y` by displaying a confusion matrix.
2. `sklearn.metrics.ConfusionMatrixDisplay.from_predictions(y_true, y_pred)` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.ConfusionMatrixDisplay.html#sklearn.metrics.ConfusionMatrixDisplay.from_predictions) - this takes truth labels and predicted labels and compares them by displaying a confusion matrix.

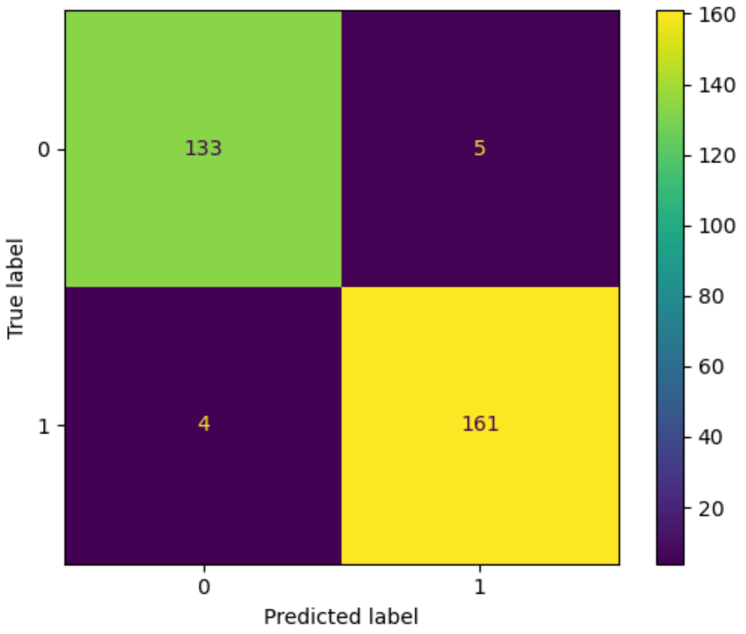
> **Note:** Both of these methods/classes require Scikit-Learn 1.0+. To check your version of Scikit-Learn run:

```
import sklearn
sklearn.__version__
```

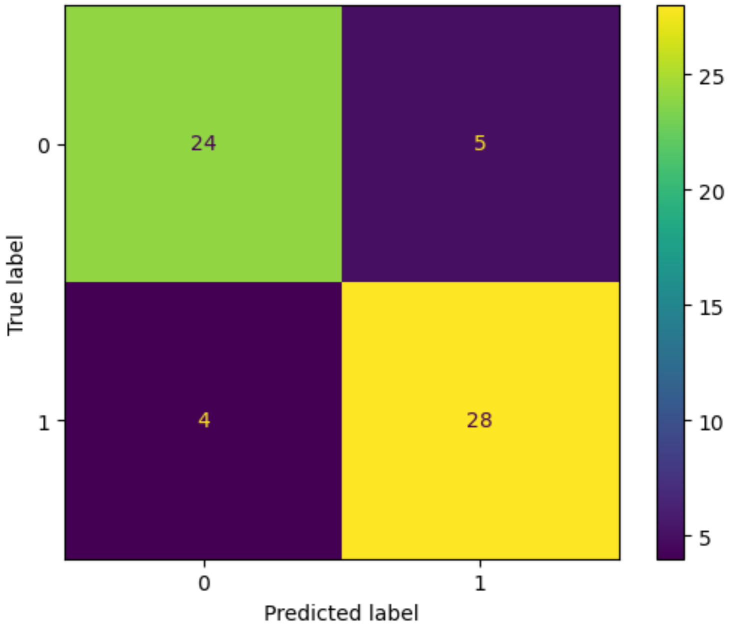
> If you don't have 1.0+, you can upgrade at: https://scikit-learn.org/stable/install.html (https://scikit-learn.org/stable/install.html)

In [138…
```
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_estimator(estimator=clf, X=X, y=y);
```



In [139…
```
# Plot confusion matrix from predictions
ConfusionMatrixDisplay.from_predictions(y_true=y_test,
                                        y_pred=y_preds);
```



Classification report

The final major metric you should consider when evaluating a classification model is a classification report.

A classification report is more so a collection of metrics rather than a single one.

You can create a classification report using Scikit-Learn's [sklearn.metrics.classification_report`](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)) method.

Let's see one.

In [140…
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support

           0       0.86      0.83      0.84        29
           1       0.85      0.88      0.86        32

    accuracy                           0.85        61
   macro avg       0.85      0.85      0.85        61
weighted avg       0.85      0.85      0.85        61
```

It returns four columns: precision, recall, f1-score and support.

The number of rows will depend on how many different classes there are. But there will always be three rows labell accuracy, macro avg and weighted avg.

Each term measures something slightly different:

- **Precision** - Indicates the proportion of positive identifications (model predicted class `1`) which were actually correct. A model which produces no false positives has a precision of 1.0.
- **Recall** - Indicates the proportion of actual positives which were correctly classified. A model which produces no false negatives has a recall of 1.0.
- **F1 score** - A combination of precision and recall. A perfect model achieves an F1 score of 1.0.
- **Support** - The number of samples each metric was calculated on.
- **Accuracy** - The accuracy of the model in decimal form. Perfect accuracy is equal to 1.0, in other words, getting the prediction right 100% of the time.
- **Macro avg** - Short for macro average, the average precision, recall and F1 score between classes. Macro avg doesn't take class imbalance into effect. So if you do have class imbalances (more examples of one class than another), you should pay attention to this.
- **Weighted avg** - Short for weighted average, the weighted average precision, recall and F1 score between classes. Weighted means each metric is calculated with respect to how many samples there are in each class. This metric will favour the majority class (e.g. it will give a high value when one class out performs another due to having more samples).

When should you use each?

It can be tempting to base your classification models perfomance only on accuracy. And accuracy is a good metric to report, except when you have very imbalanced classes.

For example, let's say there were 10,000 people. And 1 of them had a disease. You're asked to build a model to predict who has it.

You build the model and find your model to be 99.99% accurate. Which sounds great! ...until you realise, all its doing is predicting no one has the disease, in other words all 10,000 predictions are false.

In this case, you'd want to turn to metrics such as precision, recall and F1 score.

In [141…

```
# Where precision and recall become valuable
disease_true = np.zeros(10000)
disease_true[0] = 1 # only one case

disease_preds = np.zeros(10000) # every prediction is 0

pd.DataFrame(classification_report(disease_true,
                                   disease_preds,
                                   output_dict=True,
                                   zero_division=0))
```

Out[141]:

	0.0	1.0	accuracy	macro avg	weighted avg
precision	0.99990	0.0	0.9999	0.499950	0.99980
recall	1.00000	0.0	0.9999	0.500000	0.99990
f1-score	0.99995	0.0	0.9999	0.499975	0.99985
support	9999.00000	1.0	0.9999	10000.000000	10000.00000

You can see here, we've got an accuracy of 0.9999 (99.99%), great precision and recall on class 0.0 but nothing for class 1.0.

Ask yourself, although the model achieves 99.99% accuracy, is it useful?

To summarize:

- 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.

> **Resource:** For more on precision and recall and the tradeoffs between them, I'd suggest going through the Scikit-Learn Precision-Recall guide (https://scikit-learn.org/stable/auto_examples/model_selection/plot_precision_recall.html).

4.2.2 Regression model evaluation metrics

Similar to classification, there are several metrics you can use to evaluate your regression models (https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics).

We'll check out the following.

1. **R^2 (pronounced r-squared) or coefficient of determination** - Compares your models predictions to the mean of the targets. Values can range from negative infinity (a very poor model) to 1. For example, if all your model does is predict the mean of the targets, its R^2 value would be 0. And if your model perfectly predicts a range of numbers it's R^2 value would be 1. Higher is better.
2. **Mean absolute error (MAE)** - The average of the absolute differences between predictions and actual values. It gives you an idea of how wrong your predictions were. Lower is better.
3. **Mean squared error (MSE)** - The average squared differences between predictions and actual values. Squaring the errors removes negative errors. It also amplifies outliers (samples which have larger errors). Lower is better.

Let's see them in action. First, we'll bring down our regression model code again.

In [142…

```
# Import the RandomForestRegressor model class from the ensemble module
from sklearn.ensemble import RandomForestRegressor

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

# Split data into features (X) and labels (y)
X = housing_df.drop("target", axis=1)
y = housing_df["target"]

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

# Institate and fit the model (on the training set)
model = RandomForestRegressor()
model.fit(X_train, y_train);
```

R^2 Score (coefficient of determination)

Once you've got a trained regression model, the default evaluation metric in the `score()` function is R^2.

In [143…

```
# Calculate the models R^2 score
model.score(X_test, y_test)
```

Out[143]: 0.8066196804802649

Outside of the `score()` function, R^2 can be calculated using Scikit-Learn's `r2_score()` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html#sklearn.metrics.r2_score) function.

A model which only predicted the mean would get a score of 0.

In [144…

```
from sklearn.metrics import r2_score

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

r2_score(y_test, y_test_mean)
```

Out[144]: 0.0

And a perfect model would get a score of 1.

In [145…

```
r2_score(y_test, y_test)
```

Out[145]: 1.0

For your regression models, you'll want to maximise R^2, whilst minimising MAE and MSE.

Mean Absolute Error (MAE)

A model's mean absolute error can be calculated with Scikit-Learn's `sklearn.metrics.mean_absolute_error` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html) method.

In [146…
```
# Mean absolute error
from sklearn.metrics import mean_absolute_error

y_preds = model.predict(X_test)
mae = mean_absolute_error(y_test, y_preds)
mae
```

Out[146]: 0.3265721842781009

Our model achieves an MAE of 0.327.

This means, on average our models predictions are 0.327 units away from the actual value.

Let's make it a little more visual.

In [147…
```
df = pd.DataFrame(data={"actual values": y_test,
                        "predictions": y_preds})

df
```

Out[147]:

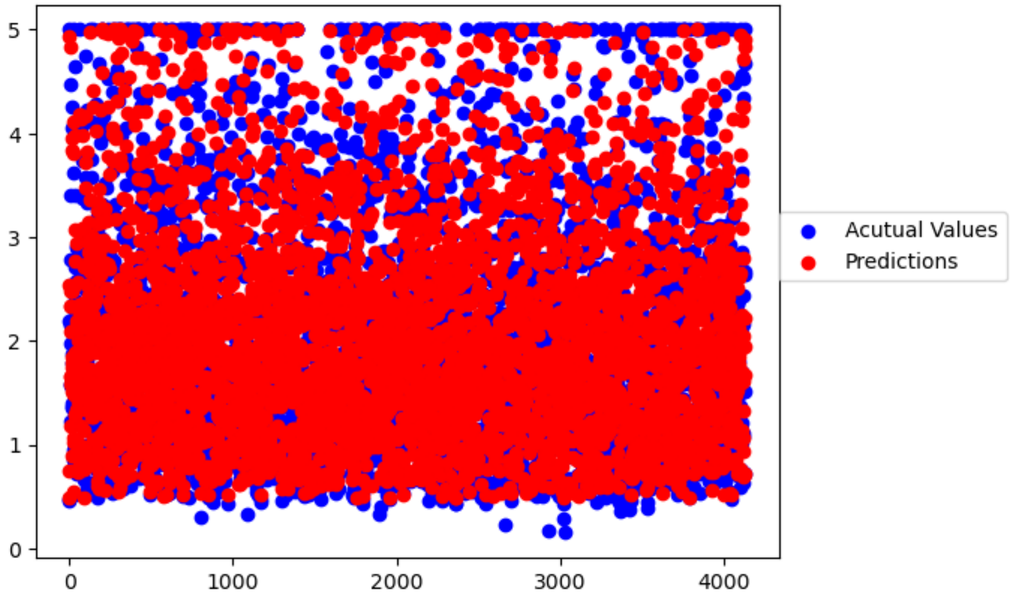
	actual values	predictions
20046	0.47700	0.493840
3024	0.45800	0.754940
15663	5.00001	4.928596
20484	2.18600	2.540290
9814	2.78000	2.331760
...	...	...
15362	2.63300	2.220380
16623	2.66800	1.947760
18086	5.00001	4.836378
2144	0.72300	0.717820
3665	1.51500	1.679010

4128 rows × 2 columns

You can see the predictions are slightly different to the actual values.

Depending what problem you're working on, having a difference like we do now, might be okay. On the flip side, it may also not be okay, meaning the predictions would have to be closer.

In [148…
```
fig, ax = plt.subplots()
x = np.arange(0, len(df), 1)
ax.scatter(x, df["actual values"], c='b', label="Acutual Values")
ax.scatter(x, df["predictions"], c='r', label="Predictions")
ax.legend(loc=(1, 0.5));
```



Mean Squared Error (MSE)

How about MSE?

We can calculate it with Scikit-Learn's `sklearn.metrics.mean_squared_error` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html).

In [149…

```
# Mean squared error
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_preds)
mse
```

Out[149]: 0.2534073069137548

MSE will often be higher than MAE because is squares the errors rather than only taking the absolute difference into account.

Now you might be thinking, which regression evaluation metric should you use?

- R^2 is similar to accuracy. It gives you a quick indication of how well your model might be doing. Generally, the closer your R^2 value is to 1.0, the better the model. But it doesn't really tell exactly how wrong your model is in terms of how far off each prediction is.
- MAE gives a better indication of how far off each of your model's predictions are on average.
- As for MAE or MSE, because of the way MSE is calculated, squaring the differences between predicted values and actual values, it amplifies larger differences. Let's say we're predicting the value of houses (which we are).
  - Pay more attention to MAE: When being $10,000 off is ***twice*** as bad as being $5,000 off.
  - Pay more attention to MSE: When being $10,000 off is ***more than twice*** as bad as being $5,000 off.

> **Note:** What we've covered here is only a handful of potential metrics you can use to evaluate your models. If you're after a complete list, check out the Scikit-Learn metrics and scoring documentation (https://scikit-learn.org/stable/modules/model_evaluation.html).

4.2.3 Evaluating a model using the `scoring` parameter

We've covered a bunch of ways to evaluate a model's predictions but haven't even touched the `scoring` parameter...

Not to worry, it's very similar to what we've been doing!

As a refresh, the `scoring` parameter can be used with a function like `cross_val_score()` to tell Scikit-Learn what evaluation metric to return using cross-validation.

Let's check it out with our classification model and the heart disease dataset.

In [150…
```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

np.random.seed(42)

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

clf = RandomForestClassifier(n_estimators=100)
```

First, we'll use the default, which is mean accuracy.

In [151…
```
np.random.seed(42)
cv_acc = cross_val_score(clf, X, y, cv=5)
cv_acc
```

Out[151]:
```
array([0.81967213, 0.90163934, 0.83606557, 0.78333333, 0.78333333])
```

We've seen this before, now we got 5 different accuracy scores on different test splits of the data.

Averaging this gives the cross-validated accuracy.

In [152…
```
# Cross-validated accuracy
print(f"The cross-validated accuracy is: {np.mean(cv_acc)*100:.2f}%")
```

```
The cross-validated accuracy is: 82.48%
```

We can find the same using the `scoring` parameter and passing it `"accuracy"`.

In [153…
```
np.random.seed(42)
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: 82.48%
```

The same goes for the other metrics we've been using for classification.

Let's try `"precision"`.

In [154…
```
np.random.seed(42)
cv_precision = cross_val_score(clf, X, y, cv=5, scoring="precision")
print(f"The cross-validated precision is: {np.mean(cv_precision):.2f}")
```

```
The cross-validated precision is: 0.83
```

How about `"recall"`?

In [155…
```
np.random.seed(42)
cv_recall = cross_val_score(clf, X, y, cv=5, scoring="recall")
print(f"The cross-validated recall is: {np.mean(cv_recall):.2f}")
```

```
The cross-validated recall is: 0.85
```

And `"f1"` (for F1 score)?

In [156…
```
np.random.seed(42)
cv_f1 = cross_val_score(clf, X, y, cv=5, scoring="f1")
print(f"The cross-validated F1 score is: {np.mean(cv_f1):.2f}")
```

```
The cross-validated F1 score is: 0.84
```

We can repeat this process with our regression metrics.

Let's revisit our regression model.

In [157…
```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestRegressor

np.random.seed(42)

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

model = RandomForestRegressor(n_estimators=100)
```

The default is `"r2"`.

In [158…
```
np.random.seed(42)
cv_r2 = cross_val_score(model, X, y, cv=5, scoring="r2")
print(f"The cross-validated R^2 score is: {np.mean(cv_r2):.2f}")
```

```
The cross-validated R^2 score is: 0.65
```

But we can use `"neg_mean_absolute_error"` for MAE (mean absolute error).

In [159…
```
np.random.seed(42)
cv_mae = cross_val_score(model, X, y, cv=5, scoring="neg_mean_absolute_error")
print(f"The cross-validated MAE score is: {np.mean(cv_mae):.2f}")
```

```
The cross-validated MAE score is: -0.47
```

Why the `"neg_"`?

Because Scikit-Learn documentation states:

> "All scorer objects follow the convention that higher return values are better than lower return values." (https://scikit-learn.org/stable/modules/model_evaluation.html#common-cases-predefined-values)

Which in this case, means a lower negative value (closer to 0) is better.

What about `"neg_mean_squared_error"` for MSE (mean squared error)?

In [160…
```
np.random.seed(42)
cv_mse = cross_val_score(model,
                         X,
                         y,
                         cv=5,
                         scoring="neg_mean_squared_error")
print(f"The cross-validated MSE score is: {np.mean(cv_mse):.2f}")
```

```
The cross-validated MSE score is: -0.43
```

4.3 Using different evaluation metrics with Scikit-Learn

Remember the third way of evaluating Scikit-Learn functions?

> 3. Problem-specific metric functions. Similar to how the `scoring` parameter can be passed different scoring functions, Scikit-Learn implements these as stand alone functions.

Well, we've kind of covered this third way of using evaulation metrics with Scikit-Learn.

In essence, all of the metrics we've seen previously have their own function in Scikit-Learn.

They all work by comparing an array of predictions, usually called `y_preds` to an array of actual labels, usually called `y_test` or `y_true`.

Classification functions

For:

- Accuracy we can use `sklearn.metrics.accuracy_score` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html)
- Precision we can use `sklearn.metrics.precision_score` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html)
- Recall we can use `sklearn.metrics.recall_score` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html)
- F1 we can use `sklearn.metrics.f1_score` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html)

In [161…

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

np.random.seed(42)

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(n_estimators=100)
clf.fit(X_train, y_train)

# Make predictions
y_preds = clf.predict(X_test)

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

```
Classifier metrics on the test set:
Accuracy: 85.25%
Precision: 0.85
Recall: 0.88
F1: 0.86
```

Regression metrics

We can use a similar setup for our regression problem, just with different methods.

For:

- R^2 we can use `sklearn.metrics.r2_score` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html)
- MAE (mean absolute error) we can use `sklearn.metrics.mean_absolute_error` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html)
- MSE (mean squared error) we can use `sklearn.metrics.mean_squared_error` (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html)

In [162…

```
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

np.random.seed(42)

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

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

model = RandomForestRegressor(n_estimators=100,
                              n_jobs=-1)
model.fit(X_train, y_train)

# Make predictions
y_preds = model.predict(X_test)

# Evaluate the model
print("Regression model metrics on the test set:")
print(f"R^2: {r2_score(y_test, y_preds):.2f}")
print(f"MAE: {mean_absolute_error(y_test, y_preds):.2f}")
print(f"MSE: {mean_squared_error(y_test, y_preds):.2f}")
```

```
Regression model metrics on the test set:
R^2: 0.81
MAE: 0.33
MSE: 0.25
```

Wow!

We've covered a lot!

But it's worth it.

Because evaluating a model's predictions is as important as training a model in any machine learning project.

There's nothing worse than training a machine learning model and optimizing for the wrong evaluation metric.

Keep the metrics and evaluation methods we've gone through when training your future models.

If you're after extra reading, I'd go through the Scikit-Learn guide for model evaluation (https://scikit-learn.org/stable/modules/model_evaluation.html).

Now we've seen some different metrics we can use to evaluate a model, let's see some ways we can improve those metrics.

5. Improving model predictions through experimentation (hyperparameter tuning)

The first predictions you make with a model are generally referred to as **baseline predictions**.

It's similar for the first evaluation metrics you get. These are generally referred to as **baseline metrics**.

Your next goal is to improve upon these baseline metrics.

How?

Experiment, experiment, experiment!

Two of the main methods to improve baseline metrics are:

1. From a data perspective.
2. From a model perspective.

From a data perspective asks:

- Could we collect more data? In machine learning, more data is generally better, as it gives a model more opportunities to learn patterns.
- Could we improve our data? This could mean filling in misisng values or finding a better encoding (turning data into numbers) strategy.

From a model perspective asks:

- Is there a better model we could use? If you've started out with a simple model, could you use a more complex one? (we saw an example of this when looking at the Scikit-Learn machine learning map (https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html), ensemble methods are generally considered more complex models)
- Could we improve the current model? If the model you're using performs well straight out of the box, can the hyperparameters be tuned to make it even better?

> **Note:** Patterns in data are also often referred to as data parameters. The difference between *parameters* and *hyperparameters* is a machine learning model seeks to find parameters in data on its own, where as, hyperparameters are settings on a model which a person (you) can adjust.

Since we have two existing datasets, we'll look at improving our results from a model perspective.

More specifically, we'll look at how we could improve our `RandomForestClassifier` and `RandomForestRegressor` models through hyperparameter tuning.

What even are hyperparameters?

Good question, let's check them out.

First, we'll instantiate a `RandomForestClassifier`.

In [163…
```
from sklearn.ensemble import RandomForestClassifier

clf = RandomForestClassifier()
```

When we instantiate a model like above, we're using the default hyperparameters.

These get printed out when you call the model instance and `get_params()`.

In [164…
```
clf.get_params()
```

```
Out[164]: {'bootstrap': True,
           'ccp_alpha': 0.0,
           'class_weight': None,
           'criterion': 'gini',
           'max_depth': None,
           'max_features': 'sqrt',
           'max_leaf_nodes': None,
           'max_samples': None,
           'min_impurity_decrease': 0.0,
           'min_samples_leaf': 1,
           'min_samples_split': 2,
           'min_weight_fraction_leaf': 0.0,
           'monotonic_cst': None,
           'n_estimators': 100,
           'n_jobs': None,
           'oob_score': False,
           'random_state': None,
           'verbose': 0,
           'warm_start': False}
```

You'll see things like `max_depth`, `min_samples_split`, `n_estimators`.

Each of these is a hyperparameter of the `RandomForestClassifier` you can adjust.

You can think of hyperparameters as being similar to dials on an oven.

On the default setting your oven might do an okay job cooking your favourite meal. But with a little experimentation, you find it does better when you adjust the settings.

The same goes for imporving a machine learning model by hyperparameter tuning.

The default hyperparameters on a machine learning model may find patterns in data well. But there's a chance a adjusting the hyperparameters may improve a models performance.

Every machine learning model will have different hyperparameters you can tune.

You might be thinking, "how the hell do I remember all of these?"

Another good question.

It's why we're focused on the Random Forest.

Instead of memorizing all of the hyperparameters for every model, we'll see how it's done with one.

And then knowing these principles, you can apply them to a different model if needed.

Reading the Scikit-Learn documentation for the Random Forest (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html), you'll find they suggest trying to change `n_estimators` (the number of trees in the forest) and `min_samples_split` (the minimum number of samples required to split an internal node).

We'll try tuning these as well as:

- `max_features` (the number of features to consider when looking for the best split)
- `max_depth` (the maximum depth of the tree)
- `min_samples_leaf` (the minimum number of samples required to be at a leaf node)

If this still sounds like a lot, the good news is, the process we're taking with the Random Forest and tuning its hyperparameters, can be used for other machine learning models in Scikit-Learn. The only difference is, with a different model, the hyperparameters you tune will be different.

Adjusting hyperparameters is usually an experimental process to figure out which are best. As there's no real way of knowing which hyperparameters will be best when starting out.

hyperparameters will be best when starting out.

To get familar with hyparameter tuning, we'll take our RandomForestClassifier and adjust its hyperparameters in 3 ways.

1. By hand
2. Randomly with `sklearn.model_selection.RandomizedSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html)
3. Exhaustively with `sklearn.model_selection.GridSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html)

5.1 Tuning hyperparameters by hand

So far we've worked with training and test datasets.

You train a model on a training set and evaluate it on a test dataset.

But hyperparameter tuning introduces a thrid set, a validation set.

Now the process becomes:

1. Train a model on the training data.
2. (Try to) improve the model's hyperparameters on the validation set.
3. Evaluate the model on the test set.

If our starting dataset contained 100 different patient records labels indicating who had heart disease and who didn't and we wanted to build a machine learning model to predict who had heart disease and who didn't, it might look like this:

Since we know we're using a `RandomForestClassifier` and we know the hyperparameters we want to adjust, let's see what it looks like.

First, let's remind ourselves of the base parameters.

In [165…
```
clf.get_params()
```

Out[165]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'sqrt',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 1,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'monotonic_cst': None,
 'n_estimators': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': None,
 'verbose': 0,
 'warm_start': False}
```

And we're going to adjust:

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

We'll use the same code as before, except this time we'll create a training, validation and test split.

With the training set containing 70% of the data and the validation and test sets each containing 15%.

Let's get some baseline results, then we'll tune the model.

And since we're going to be evaluating a few models, let's make an evaluation function.

In [166…
```
def evaluate_preds(y_true: np.array,
                   y_preds: np.array) -> dict:
    """
    Performs evaluation comparison on y_true labels vs. y_pred labels.

    Returns several metrics in the form of a dictionary.
    """
    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)
    metric_dict = {"accuracy": round(accuracy, 2),
                   "precision": round(precision, 2),
                   "recall": round(recall, 2),
                   "f1": round(f1, 2)}
    print(f"Acc: {accuracy * 100:.2f}%")
    print(f"Precision: {precision:.2f}")
    print(f"Recall: {recall:.2f}")
    print(f"F1 score: {f1:.2f}")

    return metric_dict
```

Wonderful!

Now let's recreate a previous workflow, except we'll add in the creation of a validation set.

In [167…
```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# Set the seed
np.random.seed(42)

# Read in the data
heart_disease = pd.read_csv("/home/smayan/Desktop/AI-ML-DS/sample_project/heart-disease.csv")

# Split into X (features) & y (labels)
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

# Training and test split (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

# Create validation and test split by spliting testing data in half (30% test -> 15% validation, 15% test)
X_valid, X_test, y_valid, y_test = train_test_split(X_test, y_test, test_size=0.5)

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

# Make predictions
y_preds = clf.predict(X_valid)

# Evaluate the classifier
baseline_metrics = evaluate_preds(y_valid, y_preds)
baseline_metrics
```

```
Acc: 80.00%
Precision: 0.78
Recall: 0.88
F1 score: 0.82
```

Out[167]:
```
{'accuracy': 0.8, 'precision': 0.78, 'recall': 0.88, 'f1': 0.82}
```

In [168…

```
# Check the sizes of the splits
print(f"Training data: {len(X_train)} samples, {len(y_train)} labels")
print(f"Validation data: {len(X_valid)} samples, {len(y_valid)} labels")
print(f"Testing data: {len(X_test)} samples, {len(y_test)} labels")
```

```
Training data: 212 samples, 212 labels
Validation data: 45 samples, 45 labels
Testing data: 46 samples, 46 labels
```

Beautiful, now let's try and improve the results.

We'll change 1 of the hyperparameters, `n_estimators=100` (default) to `n_estimators=200` and see if it improves on the validation set.

In [169…

```
np.random.seed(42)

# Create a second classifier
clf_2 = RandomForestClassifier(n_estimators=200)
clf_2.fit(X_train, y_train)

# Make predictions
y_preds_2 = clf_2.predict(X_valid)

# Evaluate the 2nd classifier
clf_2_metrics = evaluate_preds(y_valid, y_preds_2)
```

```
Acc: 77.78%
Precision: 0.77
Recall: 0.83
F1 score: 0.80
```

Hmm, it looks like doubling the `n_estimators` value performs *worse* than the default, perhaps there's a better value for `n_estimators` ?

And what other hyperparameters could we change?

Wait...

This could take a while if all we're doing is building new models with new hyperparameters each time.

Surely there's a better way?

There is.

5.2 Hyperparameter tuning with `RandomizedSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html)

Scikit-Learn's `sklearn.model_selection.RandomizedSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html) allows us to randomly search across different hyperparameters to see which work best.

It also stores details about the ones which work best!

Let's see it in action.

First, we create a dictionary of parameter distributions (collections of different values for specific hyperparamters) we'd like to search over.

This dictionary comes in the form:

`param_distributions = {"hyperparameter_name": [values_to_randomly_try]}`
Where `"hyperparameter_name"` is the value of a specific hyperparameter for a model and `[values_to_randomly_try]` is a list of values for that specific hyperparamter to randomly try.

In [170…
```
# Hyperparameter grid RandomizedSearchCV will search over
param_distributions = {"n_estimators": [10, 100, 200, 500, 1000, 1200],
                       "max_depth": [None, 5, 10, 20, 30],
                       "max_features": ["sqrt", "log2", None],
                       "min_samples_split": [2, 4, 6, 8],
                       "min_samples_leaf": [1, 2, 4, 8]}
```

Where did these values come from?

They're made up.

Made up?

Yes.

Not completely pulled out of the air but after reading the Scikit-Learn documentation on Random Forest's (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html) you'll see some of these values have certain values which usually perform well and certain hyperparameters take strings rather than integers.

Now we've got the parameter distribution dictionary setup, Scikit-Learn's `RandomizedSearchCV` will look at it, pick a random value from each, instantiate a model with those values and test each model.

How many models will it test?

As many as there are for each combination of hyperparameters to be tested. Let's add them up.

In [171…
```
# Count the total number of hyperparameter combinations to test
total_randomized_hyperparameter_combintions_to_test = np.prod([len(value) for value in param_distributions.valu
print(f"There are {total_randomized_hyperparameter_combintions_to_test} potential combinations of hyperparamete
```

```
There are 1440 potential combinations of hyperparameters to test.
```

Woah!

That's a lot of combinations!

Or...

We can set the `n_iter` parameter to limit the number of models `RandomizedSearchCV` tests (e.g. `n_iter=20` means to try `20` different random combintations of hyperparameters and will cross-validate each set, so if `cv=5` , 5x20 = 100 total fits).

The best thing?

The results we get will be cross-validated (hence the CV in `RandomizedSearchCV`) so we can use `train_test_split()` .

And since we're going over so many different models, we'll set `n_jobs=-1` in our `RandomForestClassifier` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html) so Scikit-Learn takes advantage of all the cores (processors) on our computers.

Let's see it in action.

> **Note:** Depending on `n_iter` (how many models you test), the different values in the hyperparameter grid, and the power of your computer, running the cell below may take a while (for reference, it took about ~1-minute on my M1 Pro MacBook Pro).

In [172…

```python
# Start the timer
import time
start_time = time.time()

from sklearn.model_selection import RandomizedSearchCV, train_test_split

np.random.seed(42)

# Split into X & y
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

# Set n_jobs to -1 to use all available cores on your machine (if this causes errors, try n_jobs=1)
clf = RandomForestClassifier(n_jobs=-1)

# Setup RandomizedSearchCV
n_iter = 30 # try 30 models total
rs_clf = RandomizedSearchCV(estimator=clf,
                            param_distributions=param_distributions,
                            n_iter=n_iter,
                            cv=5, # 5-fold cross-validation
                            verbose=2) # print out results

# Fit the RandomizedSearchCV version of clf (does cross-validation for us, so no need to use a validation set)
rs_clf.fit(X_train, y_train);

# Finish the timer
end_time = time.time()
print(f"[INFO] Total time taken for {n_iter} random combinations of hyperparameters: {end_time - start_time:.2f
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=10, max_features=None, min_samples_leaf=8, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=10, max_features=None, min_samples_leaf=8, min_samples_split=2, n_estimators=500; total time
0.6s
[CV] END max_depth=10, max_features=None, min_samples_leaf=8, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=10, max_features=None, min_samples_leaf=8, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=10, max_features=None, min_samples_leaf=8, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=2, min_samples_split=8, n_estimators=100; total time=
0.1s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=2, min_samples_split=8, n_estimators=100; total time=
0.1s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=2, min_samples_split=8, n_estimators=100; total time=
0.1s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=2, min_samples_split=8, n_estimators=100; total time=
```

```
0.1s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=2, min_samples_split=8, n_estimators=100; total time=
0.1s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=2, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=2, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=2, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=2, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=2, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=10, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=8, min_samples_split=4, n_estimators=1200; total tim
=   1.1s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=8, min_samples_split=4, n_estimators=1200; total tim
=   1.1s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=8, min_samples_split=4, n_estimators=1200; total tim
=   1.1s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=8, min_samples_split=4, n_estimators=1200; total tim
=   1.2s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=8, min_samples_split=4, n_estimators=1200; total tim
=   1.2s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
```

```
0.0s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=8, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=log2, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total time=
0.0s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=4, min_samples_split=4, n_estimators=1200; total t
me=   1.2s
[CV] END max_depth=None, max_features=None, min_samples_leaf=4, min_samples_split=4, n_estimators=1200; total t
```

```
me=   1.2s
[CV] END max_depth=None, max_features=None, min_samples_leaf=4, min_samples_split=4, n_estimators=1200; total t
me=   1.2s
[CV] END max_depth=None, max_features=None, min_samples_leaf=4, min_samples_split=4, n_estimators=1200; total t
me=   1.2s
[CV] END max_depth=None, max_features=None, min_samples_leaf=4, min_samples_split=4, n_estimators=1200; total t
me=   1.3s
[CV] END max_depth=5, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1000; total time
1.0s
[CV] END max_depth=5, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1000; total time
1.0s
[CV] END max_depth=5, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1000; total time
1.1s
[CV] END max_depth=5, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1000; total time
1.0s
[CV] END max_depth=5, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1000; total time
1.0s
[CV] END max_depth=5, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=5, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=20, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1200; total tim
=   1.2s
[CV] END max_depth=20, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1200; total tim
=   1.1s
[CV] END max_depth=20, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1200; total tim
=   1.3s
[CV] END max_depth=20, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1200; total tim
=   1.3s
[CV] END max_depth=20, max_features=None, min_samples_leaf=2, min_samples_split=8, n_estimators=1200; total tim
=   1.2s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=8, min_samples_split=2, n_estimators=1000; total tim
```

```
=   0.9s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=8, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=8, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=8, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=8, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=2, min_samples_split=6, n_estimators=10; total time=
0.0s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=4, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=4, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=4, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=4, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=4, min_samples_split=6, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=8, n_estimators=500; total time
0.5s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=8, n_estimators=500; total time
0.5s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=8, n_estimators=500; total time
0.5s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=8, n_estimators=500; total time
0.5s
[CV] END max_depth=20, max_features=None, min_samples_leaf=4, min_samples_split=8, n_estimators=500; total time
```

```
0.5s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=6, n_estimators=500; total time=
0.5s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=6, n_estimators=500; total time=
0.5s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=6, n_estimators=500; total time=
0.5s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=6, n_estimators=500; total time=
0.5s
[CV] END max_depth=5, max_features=None, min_samples_leaf=1, min_samples_split=6, n_estimators=500; total time=
0.5s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=1, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=1, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=1, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=1, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=sqrt, min_samples_leaf=1, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total time=
0.0s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total time=
0.0s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total time=
0.0s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total time=
```

```
0.0s
[CV] END max_depth=20, max_features=log2, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total time=
0.0s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=1000; total tim
=   1.1s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=20, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=2, min_samples_split=6, n_estimators=200; total ti
e=   0.2s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=2, min_samples_split=6, n_estimators=200; total ti
e=   0.2s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=2, min_samples_split=6, n_estimators=200; total ti
e=   0.2s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=2, min_samples_split=6, n_estimators=200; total ti
e=   0.2s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=2, min_samples_split=6, n_estimators=200; total ti
e=   0.2s
[CV] END max_depth=None, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=1, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=100; total time
0.1s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=100; total time
0.1s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=100; total time
```

```
0.1s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=100; total time
0.1s
[CV] END max_depth=10, max_features=sqrt, min_samples_leaf=1, min_samples_split=4, n_estimators=100; total time
0.1s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=2, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=2, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=2, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=2, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=20, max_features=sqrt, min_samples_leaf=2, min_samples_split=2, n_estimators=500; total time
0.5s
[CV] END max_depth=None, max_features=None, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=None, min_samples_leaf=8, min_samples_split=4, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[CV] END max_depth=None, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=10; total tim
=   0.0s
[INFO] Total time taken for 30 random combinations of hyperparameters: 56.91 seconds.
```

When `RandomizedSearchCV` goes through `n_iter` combinations of of hyperparameter search space, it stores the best ones in the attribute `best_params_` .

In [173…
```python
# Find the best hyperparameters found by RandomizedSearchCV
rs_clf.best_params_
```

Out[173]:
```
{'n_estimators': 200,
 'min_samples_split': 6,
 'min_samples_leaf': 4,
 'max_features': 'log2',
 'max_depth': 30}
```

Now when we call `predict()` on `rs_clf` (our `RandomizedSearchCV` version of our classifier), it'll use the best hyperparameters it found.

In [174…
```python
# Make predictions with the best hyperparameters
rs_y_preds = rs_clf.predict(X_test)

# Evaluate the predictions
rs_metrics = evaluate_preds(y_test, rs_y_preds)
```

```
Acc: 85.25%
Precision: 0.85
Recall: 0.88
F1 score: 0.86
```

Excellent!

Thanks to `RandomizedSearchCV` testing out a bunch of different hyperparameters, we get a nice boost to all of the evaluation metrics for our classification model.

5.3 Hyperparameter tuning with `GridSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html)

There's one more way we could try to improve our model's hyperparamters.

And it's with `sklearn.model_selection.GridSearchCV` (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html).

The main difference between `GridSearchCV` and `RandomizedSearchCV` is `GridSearchCV` searches across a grid of hyperparamters exhaustively (it will try every combination possible), where as, `RandomizedSearchCV` searches across a grid of hyperparameters randomly (stopping after `n_iter` combinations).

`GridSearchCV` also refers to a dictionary of parameter distributions as a parameter grid (via the parameter `param_grid`).

For example, let's see our dictionary of hyperparameters.

In [175…

```
param_distributions
```

Out[175]:

```
{'n_estimators': [10, 100, 200, 500, 1000, 1200],
 'max_depth': [None, 5, 10, 20, 30],
 'max_features': ['sqrt', 'log2', None],
 'min_samples_split': [2, 4, 6, 8],
 'min_samples_leaf': [1, 2, 4, 8]}
```

`RandomizedSearchCV` tries `n_iter` combinations of different values.

Where as, `GridSearchCV` will try every single possible combination.

And if you remember from before when we did the calculation: `max_depth` has 4 values, `max_features` has 2, `min_samples_leaf` has 3, `min_samples_split` has 3, `n_estimators` has 5.

That's 4x2x3x3x5 = 360 models!

This could take a long time depending on the power of the computer you're using, the amount of data you have and the complexity of the hyperparamters (usually higher values means a more complex model).

In our case, the data we're using is relatively small (only ~300 samples).

Since we've already tried to find some ideal hyperparameters using `RandomizedSearchCV` , we'll create another hyperparameter grid based on the `best_params_` of `rs_clf` with less options and then try to use `GridSearchCV` to find a more ideal set.

In essence, the workflow could be:

1. Tune hyperparameters by hand to get a feel of the data/model.
2. Create a large set of hyperparameter distributions and search across them randomly with `RandomizedSearchCV` .
3. Find the best hyperparameters from 2 and reduce the search space before searching across a smaller subset exhaustively with `GridSearchCV` .

> **Note:** Based on the `best_params_` of `rs_clf` implies the next set of hyperparameters we'll try are roughly in the same range of the best set found by `RandomizedSearchCV` .

In [176…

```
# Create hyperparameter grid similar to rs_clf.best_params_
param_grid = {"n_estimators": [200, 1000],
              "max_depth": [30, 40, 50],
              "max_features": ["log2"],
              "min_samples_split": [2, 4, 6, 8],
              "min_samples_leaf": [4]}
```

We've created another grid of hyperparameters to search over, this time with less total.

In [177…
```
# Count the total number of hyperparameter combinations to test
total_grid_search_hyperparameter_combinations_to_test = np.prod([len(value) for value in param_grid.values()])
print(f"There are {total_grid_search_hyperparameter_combinations_to_test} combinations of hyperparameters to te
print(f"This is {total_randomized_hyperparameter_combintions_to_test/total_grid_search_hyperparameter_combinati
 than before (previous: {total_randomized_hyperparameter_combintions_to_test}).")
```

```
There are 24 combinations of hyperparameters to test.
This is 60.0 times less than before (previous: 1440).
```

Now when we run `GridSearchCV` , passing it our classifier (`clf`), parameter grid (`param_grid`) and the number of cross-validation folds we'd like to use (`cv=5`), it'll create a model with every single combination of hyperparameters, and then cross-validate each 5 times (for example, 36 hyperparameter combinations * 5 = 135 fits in total) and check the results.

> **Note:** Depending on the compute power of the machine you're using, the following cell may take a few minutes to run (for reference, it took ~60 seconds on my M1 Pro MacBook Pro).

In [178…
```
# Start the timer
import time
start_time = time.time()

from sklearn.model_selection import GridSearchCV, train_test_split

np.random.seed(42)

# Split into X & y
X = heart_disease.drop("target", axis=1)
y = heart_disease["target"]

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

# Set n_jobs to -1 to use all available machine cores (if this produces errors, try n_jobs=1)
clf = RandomForestClassifier(n_jobs=-1)

# Setup GridSearchCV
gs_clf = GridSearchCV(estimator=clf,
                      param_grid=param_grid,
                      cv=5, # 5-fold cross-validation
                      verbose=2) # print out progress

# Fit the RandomizedSearchCV version of clf
gs_clf.fit(X_train, y_train);

# Find the running time
end_time = time.time()
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.3s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.3s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.3s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
```

```
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=30, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
```

```
=   0.9s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
```

```
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=40, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=2, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=200; total time
```

```
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   0.9s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=4, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=6, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
```

```
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=200; total time
0.2s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   1.0s
[CV] END max_depth=50, max_features=log2, min_samples_leaf=4, min_samples_split=8, n_estimators=1000; total tim
=   0.9s
/media/smayan/ohio/conda env/env/lib/python3.12/site-packages/numpy/ma/core.py:2820: RuntimeWarning: invalid va
ue encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
```

In [179…

```
# How long did it take?
total_time = end_time - start_time
print(f"[INFO] The total running time for running GridSearchCV was {total_time:.2f} seconds.")
```

```
[INFO] The total running time for running GridSearchCV was 71.21 seconds.
```

Once it completes, we can check the best hyperparameter combinations it found using the `best_params_` attribute.

In [180…

```
# Check the best hyperparameters found with GridSearchCV
gs_clf.best_params_
```

Out[180]:

```
{'max_depth': 30,
 'max_features': 'log2',
 'min_samples_leaf': 4,
 'min_samples_split': 2,
 'n_estimators': 200}
```

And by default when we call the `predict()` function on `gs_clf`, it'll use the best hyperparameters.

In [181…

```
# Max predictions with the GridSearchCV classifier
gs_y_preds = gs_clf.predict(X_test)

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

```
Acc: 88.52%
Precision: 0.88
Recall: 0.91
F1 score: 0.89
```

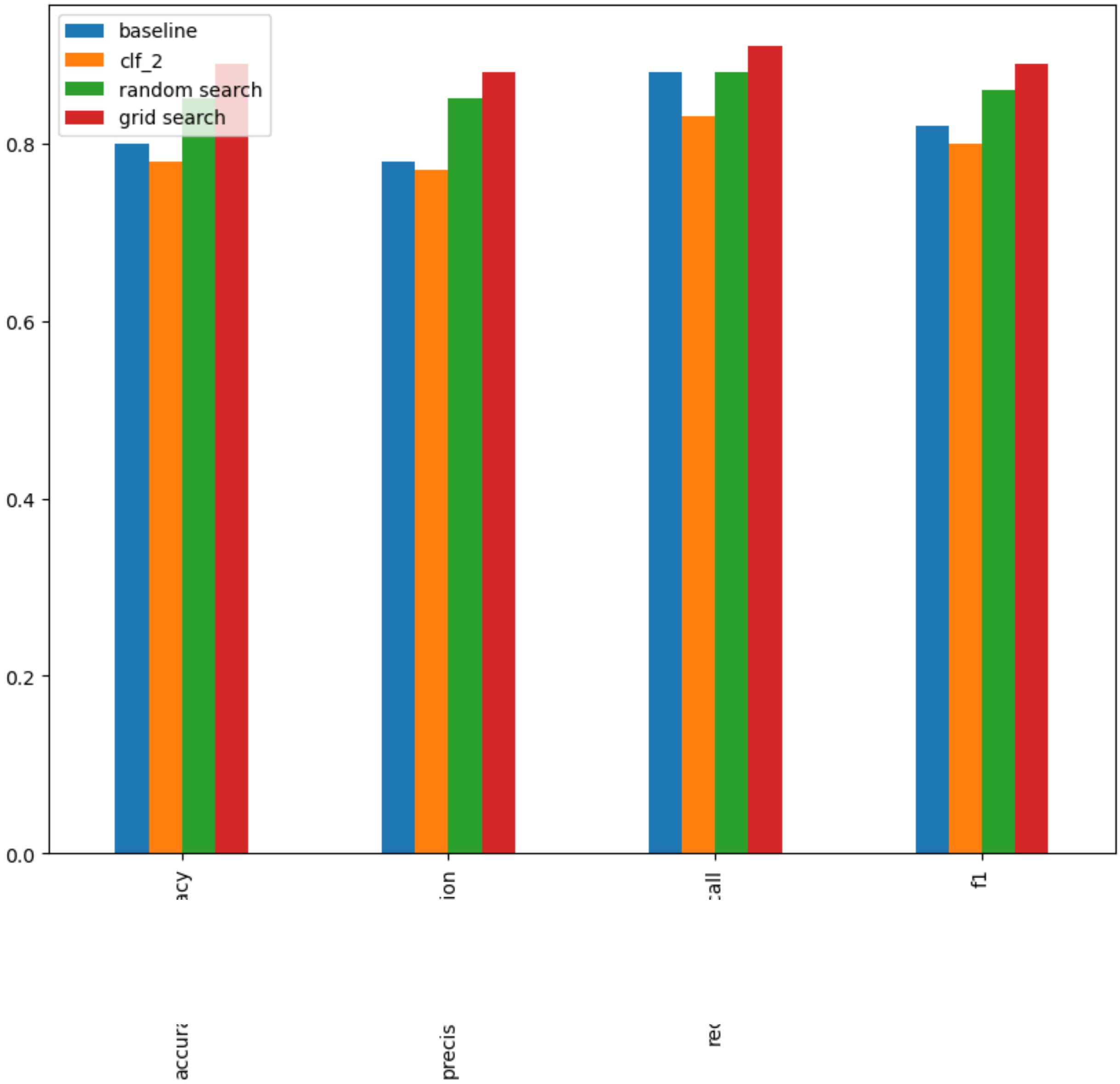
Out[181]:

```
{'accuracy': 0.89, 'precision': 0.88, 'recall': 0.91, 'f1': 0.89}
```

Let's create a DataFrame to compare the different metrics.

In [182…

```
compare_metrics = pd.DataFrame({"baseline": baseline_metrics,
                                "clf_2": clf_2_metrics,
                                "random search": rs_metrics,
                                "grid search": gs_metrics})
compare_metrics.plot.bar(figsize=(10, 8));
```



Nice!

After trying many different combinations of hyperparamters, we get a slight improvement in results.

However, sometimes you'll notice that your results don't change much.

These things might happen.

But it's important to remember, it's not over. There more things you can try.

In a hyperparameter tuning sense, there may be a better set we could find through more extensive searching with `RandomizedSearchCV` and `GridSearchCV`, this would require more experimentation.

Other techniques you could:

- **Collecting more data** - Based on the results our models are getting now, it seems like they're very capable of finding patterns. Collecting more data may improve a models ability to find patterns. However, your ability to do this will largely depend on the project you're working on.
- **Try a more advanced model** - Although our tuned Random Forest model is doing pretty well, a more advanced ensemble method such as XGBoost (https://xgboost.ai/) or CatBoost (https://catboost.ai/) might perform better. I'll leave these for extra-curriculum.

Since machine learning is part engineering, part science, these kind of experiments are common place in any machine learning project.

Now we've got a tuned Random Forest model, let's find out how we might save it and export it so we can share it with others or potentially use it in an external application.

6. Saving and loading trained machine learning models

Our `GridSearchCV` model (`gs_clf`) has the best results so far, we'll export it and save it to file.

6.1 Saving and loading a model with `pickle`

We saw right at the start, one way to save a model is using Python's `pickle` module (https://docs.python.org/3/library/pickle.html).

We'll use `pickle` 's `dump()` method and pass it our model, `gs_clf` , along with the `open()` function containing a string for the filename we want to save our model as, along with the `"wb"` string which stands for "write binary", which is the file type `open()` will write our model as.

In [183…
```python
import pickle

# Save an existing model to file
best_model_file_name_pickle = "gs_random_forest_model_1.pkl" # .pkl extension stands for "pickle"
pickle.dump(gs_clf, open(best_model_file_name_pickle, "wb"))
```

Once it's saved, we can import it using `pickle` 's `load()` function, passing it `open()` containing the filename as a string and `"rb"` standing for "read binary".

In [184…
```python
# Load a saved model
loaded_pickle_model = pickle.load(open(best_model_file_name_pickle, "rb"))
```

Once you've reimported your trained model using `pickle` , you can use it to make predictions as usual.

In [185…
```python
# Make predictions and evaluate the loaded model
pickle_y_preds = loaded_pickle_model.predict(X_test)
loaded_pickle_model_metrics = evaluate_preds(y_test, pickle_y_preds)
loaded_pickle_model_metrics
```

```
Acc: 88.52%
Precision: 0.88
Recall: 0.91
F1 score: 0.89
```

Out[185]:
```
{'accuracy': 0.89, 'precision': 0.88, 'recall': 0.91, 'f1': 0.89}
```

You'll notice the reimported model evaluation metrics are the same as the model before we exported it.

In [186…
```python
loaded_pickle_model_metrics == gs_metrics
```

Out[186]:
```
True
```

6.2 Saving and loading a model with `joblib` (https://joblib.readthedocs.io/en/latest/persistence.html)

The other way to load and save models is with `joblib` . Which works relatively the same as `pickle` .

To save a model, we can use `joblib` 's `dump()` function, passing it the model (`gs_clf`) and the desired filename.

In [187…
```python
from joblib import dump, load

# Save a model to file
best_model_file_name_joblib = "gs_random_forest_model_1.joblib"
dump(gs_clf, filename=best_model_file_name_joblib)
```

Out[187]:
```
['gs_random_forest_model_1.joblib']
```

Once you've saved a model using `dump()` , you can import it using `load()` and passing it the filename of the model.

In [188…
```python
# Import a saved joblib model
loaded_joblib_model = load(filename=best_model_file_name_joblib)
```

Again, once imported, we can make predictions with our model.

In [189…
```
# Make and evaluate joblib predictions
joblib_y_preds = loaded_joblib_model.predict(X_test)
loaded_joblib_model_metrics = evaluate_preds(y_test, joblib_y_preds)
loaded_joblib_model_metrics
```

```
Acc: 88.52%
Precision: 0.88
Recall: 0.91
F1 score: 0.89
```

Out[189]:
```
{'accuracy': 0.89, 'precision': 0.88, 'recall': 0.91, 'f1': 0.89}
```

And once again, you'll notice the evaluation metrics are the same as before.

In [190…
```
loaded_joblib_model_metrics == gs_metrics
```

Out[190]:
```
True
```

So which one should you use, `pickle` or `joblib` ?

According to Scikit-Learn's model persistence documentation (https://scikit-learn.org/stable/model_persistence.html), they suggest it may be more efficient to use `joblib` as it's more efficient with large numpy arrays (which is what may be contained in trained/fitted Scikit-Learn models).

7. Revisiting the entire pipeline

We've covered a lot. And so far, it seems to be all over the place, which it is.

But not to worry, machine learning projects often start out like this.

A whole bunch of experimenting and code all over the place at the start and then once you've found something which works, the refinement process begins.

What would this refinement process look like?

We'll use the car sales regression problem (predicting the sale price of cars) as an example.

To tidy things up, we'll be using Scikit-Learn's `sklearn.pipeline.Pipeline` (https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html) class.

You can imagine `Pipeline` as being a way to string a number of different Scikit-Learn processes together.

7.1 Creating a regression `Pipeline` (https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html)

You might recall when, way back in Section 2: Getting Data Ready, we dealt with the car sales data, to build a regression model on it, we had to encode the categorical features into numbers and fill the missing data.

The code we used worked, but it was a bit all over the place.

Good news is, `Pipeline` can help us clean it up.

Let's remind ourselves what the data looks like.

In [192…
```
data = pd.read_csv("/home/smayan/Desktop/AI-ML-DS/ZTM/Scikit-Learn/car-sales-extended.csv")
data.head()
```

Out[192]:

	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 [193…

```
data.dtypes
```

Out[193]:

```
Make              object
Colour            object
Odometer (KM)      int64
Doors              int64
Price              int64
dtype: object
```

In [194…

```
data.isna().sum()
```

Out[194]:

```
Make             0
Colour           0
Odometer (KM)    0
Doors            0
Price            0
dtype: int64
```

There's 1000 rows, three features are categorical (`Make` , `Colour` , `Doors`), the other two are numerical (`Odometer (KM)` , `Price`) and there's 249 missing values.

We're going to have to turn the categorical features into numbers and fill the missing values before we can fit a model.

We'll build a `Pipeline` to do so.

`Pipeline` 's main input parameter is `steps` which is a list of tuples (`[(step_name, action_to_take)]`) of the step name, plus the action you'd like it to perform.

In our case, you could think of the steps as:

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

Let's do it!

In [195…

```python
# Getting data ready
import pandas as pd
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

# Setup random seed
import numpy as np
np.random.seed(42)

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

# Define different features and transformer pipelines
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(n_jobs=-1))])

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

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[195], line 17
     14 np.random.seed(42)
     16 # Import data and drop the rows with missing labels
---> 17 data = pd.read_csv("../data/car-sales-extended-missing-data.csv")
     18 data.dropna(subset=["Price"], inplace=True)
     20 # Define different features and transformer pipelines

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/pandas/io/parsers/readers.py:1026, in read_c
v(filepath_or_buffer, sep, delimiter, header, names, index_col, usecols, dtype, engine, converters, true_values
false_values, skipinitialspace, skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, sk
p_blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser, date_format, dayfirst, cache_dat
s, iterator, chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting, doublequote, escap
char, comment, encoding, encoding_errors, dialect, on_bad_lines, delim_whitespace, low_memory, memory_map, floa
_precision, storage_options, dtype_backend)
   1013 kwds_defaults = _refine_defaults_read(
   1014     dialect,
   1015     delimiter,
   (...)
   1022     dtype_backend=dtype_backend,
   1023 )
   1024 kwds.update(kwds_defaults)
```

```
-> 1026 return _read(filepath_or_buffer, kwds)

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/pandas/io/parsers/readers.py:620, in _read(f
lepath_or_buffer, kwds)
    617 _validate_names(kwds.get("names", None))
    619 # Create the parser.
--> 620 parser = TextFileReader(filepath_or_buffer, **kwds)
    622 if chunksize or iterator:
    623     return parser

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/pandas/io/parsers/readers.py:1620, in TextFi
eReader.__init__(self, f, engine, **kwds)
   1617     self.options["has_index_names"] = kwds["has_index_names"]
   1619 self.handles: IOHandles | None = None
-> 1620 self._engine = self._make_engine(f, self.engine)

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/pandas/io/parsers/readers.py:1880, in TextFi
eReader._make_engine(self, f, engine)
   1878     if "b" not in mode:
   1879         mode += "b"
-> 1880 self.handles = get_handle(
   1881     f,
   1882     mode,
   1883     encoding=self.options.get("encoding", None),
   1884     compression=self.options.get("compression", None),
   1885     memory_map=self.options.get("memory_map", False),
   1886     is_text=is_text,
   1887     errors=self.options.get("encoding_errors", "strict"),
   1888     storage_options=self.options.get("storage_options", None),
   1889 )
   1890 assert self.handles is not None
   1891 f = self.handles.handle

File /media/smayan/ohio/conda env/env/lib/python3.12/site-packages/pandas/io/common.py:873, in get_handle(path_
r_buf, mode, encoding, compression, memory_map, is_text, errors, storage_options)
    868 elif isinstance(handle, str):
    869     # Check whether the filename is to be opened in binary mode.
    870     # Binary mode does not support 'encoding' and 'newline'.
    871     if ioargs.encoding and "b" not in ioargs.mode:
    872         # Encoding
--> 873         handle = open(
    874             handle,
    875             ioargs.mode,
    876             encoding=ioargs.encoding,
    877             errors=errors,
    878             newline="",
    879         )
    880     else:
    881         # Binary mode
    882         handle = open(handle, ioargs.mode)

FileNotFoundError: [Errno 2] No such file or directory: '../data/car-sales-extended-missing-data.csv'
```

What we've done is combine a series of data preprocessing steps (filling missing values, encoding numerical values) as well as a model into a `Pipeline`.

Doing so not only cleans up the code, it ensures the same steps are taken every time the code is run rather than having multiple different processing steps happening in different stages.

It's also possible to `GridSearchCV` or `RandomizedSearchCV` with a `Pipeline`.

The main difference is when creating a hyperparameter grid, you have to add a prefix to each hyperparameter (see the documentation for `RandomForestRegressor` (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html) for a full list of possible hyperparameters to tune).

The prefix is the name of the `Pipeline` step you'd like to alter, followed by two underscores.

For example, to adjust `n_estimators` of `"model"` in the `Pipeline`, you'd use: `"model__n_estimators"` (note the double underscore after `model__` at the start).

Let's see it!

> **Note:** Depending on your computer's processing power, the cell below may take a few minutes to run. For reference, it took about ~60 seconds on my M1 Pro MacBook Pro.

In [178…

```
# Using grid search with pipeline
pipe_grid = {
    "preprocessor__num__imputer__strategy": ["mean", "median"], # note the double underscore after each prefix
    "model__n_estimators": [100, 1000],
    "model__max_depth": [None, 5],
    "model__max_features": ["sqrt"],
    "model__min_samples_split": [2, 4]
}

gs_model = GridSearchCV(model, pipe_grid, cv=5, verbose=2)
gs_model.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, 
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, 
preprocessor__num__imputer__strategy=mean; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, 
preprocessor__num__imputer__strategy=mean; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, 
preprocessor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, 
preprocessor__num__imputer__strategy=mean; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, 
preprocessor__num__imputer__strategy=mean; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, 
preprocessor__num__imputer__strategy=median; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, 
preprocessor__num__imputer__strategy=median; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, 
preprocessor__num__imputer__strategy=median; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000,
```

```
preprocessor__num__imputer__strategy=median; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000,
preprocessor__num__imputer__strategy=median; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100,
reprocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=mean; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=median; total time=   0.8s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=median; total time=   0.8s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
```

```
preprocessor__num__imputer__strategy=median; total time=   0.9s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=median; total time=   0.8s
[CV] END model__max_depth=None, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000,
preprocessor__num__imputer__strategy=median; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=median; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
```

```
processor__num__imputer__strategy=median; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=median; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=median; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=2, model__n_estimators=1000, pr
processor__num__imputer__strategy=median; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=mean; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=100, pre
rocessor__num__imputer__strategy=median; total time=   0.1s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.8s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=mean; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
```

```
processor__num__imputer__strategy=median; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=median; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=median; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=median; total time=   0.7s
[CV] END model__max_depth=5, model__max_features=sqrt, model__min_samples_split=4, model__n_estimators=1000, pr
processor__num__imputer__strategy=median; total time=   0.7s
```

Out[178]:

```
▸ GridSearchCV
▸ estimator: Pipeline
  ▸ preprocessor: ColumnTransformer
    ▸ cat              ▸ door             ▸ num
    ▸ SimpleImputer    ▸ SimpleImputer    ▸ SimpleImputer
    ▸ OneHotEncoder
  ▸ RandomForestRegressor
```

Now let's find the score of our model (by default `GridSearchCV` saves the best model to the `gs_model` object).

In [179…

```python
# Score the best model
gs_model.score(X_test, y_test)
```

Out[179]: 0.2848784564026805

In [196…

```python
print(sklearn.__version__)
```

```
1.5.1
```

Beautiful!

Using `GridSearchCV` we see a nice boost in our models score.

And the best thing is, because it's all in a `Pipeline` , we could easily replicate these results.

Where to next?

If you've made it this far, congratulations! We've covered a lot of ground in the Scikit-Learn library.

As you might've guessed, there's a lot more to be discovered.

But for the time being, you should be equipped with some of the most useful features of the library to start trying to apply them to your own problems.

Somewhere you might like to look next is to apply what you've learned above to a Kaggle competition (https://www.kaggle.com).

Kaggle competitions are great places to practice your data science and machine learning skills and compare your results with others.

A great idea would be to try to combine the heart disease classification code, as well as the `Pipeline` code, to build a model for the Titanic dataset (https://www.kaggle.com/c/titanic).

Otherwise, if you'd like to figure out what else the Scikit-Learn library is capable of I'd highly recommend browsing through the Scikit-Learn User Guide (https://scikit-learn.org/stable/user_guide.html) and seeing what sparks your interest.

Finally, as an extra-curriculum extension, you might want to look into trying out the CatBoost library (https://catboost.ai) for dealing with non-numerical data automatically.

The CatBoost algorithm is advanced version of a decision tree (like a Random Forest with superpowers) and is used in production at several large technology companies (https://blog.cloudflare.com/how-cloudflare-runs-ml-inference-in-microseconds/), including Cloudflare.