



Open in Colab

[View source code](#) | [Read notebook in online book format](#)

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

This notebook goes through a range of common and useful features 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](#), of which, if you ever get stuck, I'd highly recommend checking out.

```
In [1]: import datetime
print(f"Last updated: {datetime.datetime.now()}")
```



Last updated: 2024-04-26 12:36:12.325496

What is Scikit-Learn (sklearn)?

[Scikit-Learn](#), also referred to as `sklearn`, is an open-source Python machine learning library.

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.

Table of contents

What is Scikit-Learn (sklearn)?

Why Scikit-Learn?

What does this notebook cover?

Where can I get help?

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

Random Forest Classifier

Workflow for Classifying Heart Disease

1. Get the data ready

2. Choose the model and hyperparameters

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

Use the model to make a prediction

4. Evaluate the model

5. Experiment to improve

6. Save a model for someone else to use

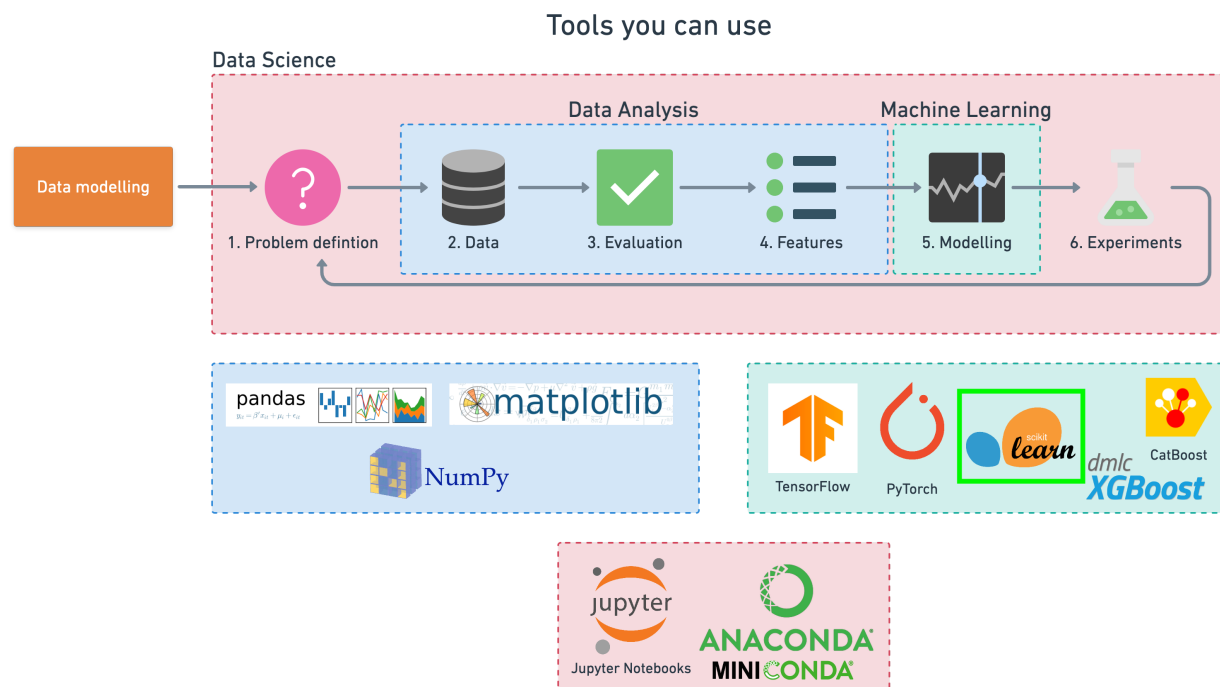
1. Getting the data ready

1.1 Make sure it's all numerical

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

1.1.1 Nuemrically encoding data with pandas

1.2 What if there were missing



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](#) (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](#) (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](#).

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).

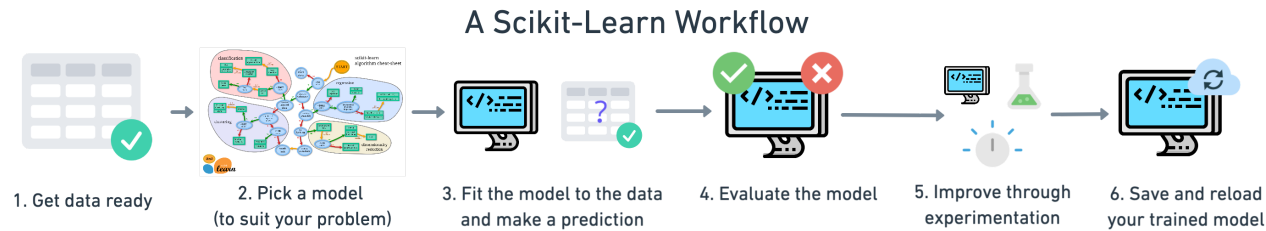
Choosing a machine learning model and then evaluating whether or not your model has learned anything.

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 workflow

1. Getting the data ready

2. Choosing the right machine learning estimator/algorithm/model for your problem

3. Fitting your chosen machine learning model to data and using it to make a prediction

4. Evaluating 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** (having data and labels).

The other side of supervised learning is **unsupervised learning** (having data but no labels).

After going through it, you'll have the base knowledge 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](#) - the most extensive resource you'll find for Scikit-Learn information.
 - [Stack Overflow](#) - 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](#) - 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

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](#) 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 [2]: # 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 note
```

Using Scikit-Learn version: 1.3.1 (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 [3]: import pandas as pd
heart_disease = pd.read_csv('../data/heart-disease.csv')
heart_disease.head()
```

Out[3]:

	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" column, 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 variable `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 [4]: # 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[4]:

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

```
Out[5]: (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` 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 hyperparameter tuning on but for now we'll focus on training and test sets.

```
In [6]: # 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

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

```
Out[6]: ((227, 13), (76, 13), (227,), (76,))
```

2. Choose the model and hyperparameters

PDFmyURL converts web pages and even full websites to PDF easily and quickly.

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](#).

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 [7]: # 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()` method.

```
In [8]: # View the current hyperparameters
        clf.get_params()
```

```
Out[8]: {'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_samples_leaf': 1,
         'min_samples_split': 2,
         'n_estimators': 100}
```

```
'min_impurity_decrease': 0.0,  
'min_samples_leaf': 1,  
'min_samples_split': 2,  
'min_weight_fraction_leaf': 0.0,  
'n_estimators': 100,  
'n_jobs': None,  
'oob_score': False,  
'random_state': None,  
'verbose': 0,  
'warm_start': False}
```

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



```
Out[9]: □ RandomForestClassifier
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()` 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 [10]: # This doesn't work... incorrect shapes
y_label = clf.predict(np.array([0, 2, 3, 4]))
```

```
/Users/daniel/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/bas
e.py:465: UserWarning: X does not have valid feature names, but RandomForestClassif
ier was fitted with feature names
  warnings.warn(
```

```
-----
ValueError                                Traceback (most recent call last)
/Users/daniel/code/zero-to-mastery-ml/section-2-data-science-and-ml-tools/introduct
ion-to-scikit-learn.ipynb Cell 24 line 2
      <a href='vscode-notebook-cell:/Users/daniel/code/zero-to-mastery-ml/section-2
-data-science-and-ml-tools/introduction-to-scikit-learn.ipynb#X32sZmlsZQ%3D%3D?line
=0'>1</a> # This doesn't work... incorrect shapes
----> <a href='vscode-notebook-cell:/Users/daniel/code/zero-to-mastery-ml/section-2
-data-science-and-ml-tools/introduction-to-scikit-learn.ipynb#X32sZmlsZQ%3D%3D?line
```

```

=> y_label = clf.predict(np.array([0, 2, 3, 4]))

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/ensemble/_forest.py:823, in ForestClassifier.predict(self, X)
    802 def predict(self, X):
    803     """
    804     Predict class for X.
    805     (...)
    821     The predicted classes.
    822     """
--> 823     proba = self.predict_proba(X)
    825     if self.n_outputs_ == 1:
    826         return self.classes_.take(np.argmax(proba, axis=1), axis=0)

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/ensemble/_forest.py:865, in ForestClassifier.predict_proba(self, X)
    863 check_is_fitted(self)
    864 # Check data
--> 865 X = self._validate_X_predict(X)
    867 # Assign chunk of trees to jobs
    868 n_jobs, _, _ = _partition_estimators(self.n_estimators, self.n_jobs)

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/ensemble/_forest.py:599, in BaseForest._validate_X_predict(self, X)
    596 """
    597 Validate X whenever one tries to predict, apply, predict_proba."""
    598 check_is_fitted(self)
--> 599 X = self._validate_data(X, dtype=DTYPE, accept_sparse="csr", reset=False)
    600 if issparse(X) and (X.indices.dtype != np.intc or X.indptr.dtype != np.intc):
    601     raise ValueError("No support for np.int64 index based sparse matrices")

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/base.py:605, in BaseEstimator._validate_data(self, X, y, reset, validate_separately, cast_to_ndarray, **check_params)
    603     out = X, y

```

```

604 elif not no_val_X and no_val_y:
--> 605     out = check_array(X, input_name="X", **check_params)
606 elif no_val_X and not no_val_y:
607     out = _check_y(y, **check_params)

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/utils/validation.py:938, in check_array(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, estimator, input_name)
    936     # If input is 1D raise error
    937     if array.ndim == 1:
--> 938         raise ValueError(
    939             "Expected 2D array, got 1D array instead:\narray={}\n".
    940             "Reshape your data either using array.reshape(-1, 1) if "
    941             "your data has a single feature or array.reshape(1, -1) "
    942             "if it contains a single sample.".format(array)
    943         )
    945 if dtype_numeric and hasattr(array.dtype, "kind") and array.dtype.kind in
"USV":
    946     raise ValueError(
    947         "dtype='numeric' is not compatible with arrays of bytes/strings."
    948         "Convert your data to numeric values explicitly instead."
    949     )

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) if 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 [11]: # In order to predict a label, data has to be in the same shape as X_train
X_test.head()
```

Out[11]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3
102	63	0	1	140	195	0	1	179	0	0.0	2	2	2
254	59	1	3	160	273	0	0	125	0	0.0	2	0	2
171	48	1	1	110	229	0	1	168	0	1.0	0	0	3
163	38	1	2	138	175	0	1	173	0	0.0	2	4	2

```
In [12]: # 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()` 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](#) (the fraction of correct predictions out of total predictions).

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

```
In [13]: # 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 dataset.

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

How about the test dataset?

```
In [14]: # 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: 75.00%

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` module:

- `classification_report(y_true, y_true)` - Builds a text report showing various classification metrics such as [precision](#), [recall](#) and [F1-score](#).
- `confusion_matrix(y_true, y_pred)` - Create a [confusion matrix](#) to compare predictions to truth labels.

- `accuracy_score(y_true, y_pred)` - 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 [15]: 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	0.81	0.60	0.69	35
1	0.72	0.88	0.79	41
accuracy			0.75	76
macro avg	0.76	0.74	0.74	76
weighted avg	0.76	0.75	0.74	76

```
In [16]: # Create a confusion matrix
conf_mat = confusion_matrix(y_test, y_preds)
conf_mat
```

```
Out[16]: array([[21, 14],
               [ 5, 36]])
```

```
In [17]: # Compute the accuracy score (same as the score() method for classifiers)
accuracy_score(y_test, y_preds)
```

```
Out[17]: 0.75
```

5. Experiment to improve

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.

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 results should always be cross-validated (we'll see this later on!).

Cross-validation 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()`, 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 [18]: # 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: 73.68%
```

Trying model with 110 estimators...
Model accuracy on test set: 73.68%

Trying model with 120 estimators...
Model accuracy on test set: 75.00%

Trying model with 130 estimators...
Model accuracy on test set: 72.37%

Trying model with 140 estimators...
Model accuracy on test set: 73.68%

Trying model with 150 estimators...
Model accuracy on test set: 73.68%

Trying model with 160 estimators...
Model accuracy on test set: 73.68%

Trying model with 170 estimators...
Model accuracy on test set: 75.00%

Trying model with 180 estimators...
Model accuracy on test set: 73.68%

Trying model with 190 estimators...
Model accuracy on test set: 75.00%

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

Let's use `sklearn.model_selection.cross_val_score` 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 [19]: 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: 73.68%
5-fold cross-validation score: 82.15%
```

```
Trying model with 110 estimators...
Model accuracy on single test set split: 73.68%
5-fold cross-validation score: 81.17%
```

```
Trying model with 120 estimators...
Model accuracy on single test set split: 75.00%
5-fold cross-validation score: 83.49%
```

```
Trying model with 130 estimators...
```

Model accuracy on single test set split: 72.37%
5-fold cross-validation score: 83.14%

Trying model with 140 estimators...
Model accuracy on single test set split: 73.68%
5-fold cross-validation score: 82.48%

Trying model with 150 estimators...
Model accuracy on single test set split: 73.68%
5-fold cross-validation score: 80.17%

Trying model with 160 estimators...
Model accuracy on single test set split: 71.05%
5-fold cross-validation score: 80.83%

Trying model with 170 estimators...
Model accuracy on single test set split: 73.68%
5-fold cross-validation score: 82.16%

Trying model with 180 estimators...
Model accuracy on single test set split: 72.37%
5-fold cross-validation score: 81.50%

Trying model with 190 estimators...
Model accuracy on single test set split: 72.37%
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` is a way to search over a set of different hyperparameter values and automatically track which perform the best.

Let's test it!

```
In [20]: # 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 [21]: # Set the model to be the best estimator
clf = grid.best_estimator_
```


clf

```
Out[21]:  RandomForestClassifier  
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 [22]: # 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)  
print(f"Best model score on single split of the data: {clf.score(X_test, y_test)}")
```

Best model score on single split of the data: 75.00%

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](#) using Python's in-built `pickle` module.

```
In [23]: import pickle

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

```
In [24]: # 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,
```

Loaded pickle model prediction score: 72.37%

For larger models, it may be more efficient to use [Joblib](#).

```
In [25]: from joblib import dump, load

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

```
Out[25]: ['random_forest_model_1.joblib']
```

```
In [26]: # 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,
```

Loaded joblib model prediction score: 72.37%

Woah!

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

Let's break things down a bit more by revisiting 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 [27]: # Splitting the data into X & y
heart_disease.head()
```

Out[27]:

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

```
X
```

```
Out [28]:
```

	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 [29]: y = heart_disease['target']  
y
```

```
Out [29]: 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 [30]: # 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 this

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

```
Out[30]: ((242, 13), (61, 13), (242,), (61,))
```

```
In [31]: # 80% of data is being used for the training set (the model will learn patterns from this)
X.shape[0] * 0.8
```

```
Out[31]: 242.4
```

```
In [32]: # And 20% of the data is being used for the testing set (the model will be evaluated on this)
X.shape[0] * 0.2
```

```
Out[32]: 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 [33]: # Import car-sales-extended.csv
car_sales = pd.read_csv("../data/car-sales-extended.csv")
car_sales
```

Out [33]:

	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 [34]: car_sales.dtypes
```

```
Out[34]: 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) whereas 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 [35]: # 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 [36]: `# 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)
/var/folders/c4/qj4gdk190td18bqvjjh0p3p00000gn/T/ipykernel_30502/1044518071.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)

~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/base.py in ?(estimator, *args, **kwargs)
    1148         skip_parameter_validation=(
    1149             prefer_skip_nested_validation or global_skip_validation
    1150         )
    1151     ):
-> 1152         return fit_method(estimator, *args, **kwargs)

~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/ensemble/_forest.py in ?(self, X, y, sample_weight)
    344         """
    345         # Validate or convert input data
    346         if issparse(y):
    347             raise ValueError("sparse multilabel-indicator for y is not supported.")
--> 348         X, y = self._validate_data(
    349             X, y, multi_output=True, accept_sparse="csc", dtype=DTYPE
    350         )
```



```

351         if sample_weight is not None:

~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/base.py in ?(self, X, y, reset, validate_separately, cast_to_ndarray, **check_params)
618             if "estimator" not in check_y_params:
619                 check_y_params = {**default_check_params, **check_y_params}
620             y = check_array(y, input_name="y", **check_y_params)
621         else:
--> 622             X, y = check_X_y(X, y, **check_params)
623             out = X, y
624
625             if not no_val_X and check_params.get("ensure_2d", True):

~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/utils/validation.py in ?(X, y, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, multi_output, ensure_min_samples, ensure_min_features, y_numeric, estimator)
1142         raise ValueError(
1143             f"{estimator_name} requires y to be passed, but the target y is None"
1144         )
1145
-> 1146     X = check_array(
1147         X,
1148         accept_sparse=accept_sparse,
1149         accept_large_sparse=accept_large_sparse,

~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/utils/validation.py in ?(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, estimator, input_name)
912         )
913         array = xp.astype(array, dtype, copy=False)
914     else:
915         array = _asarray_with_order(array, order=order, dtype=dtype, xp=xp)

```

```

--> 916         except ComplexWarning as complex_warning:
917             raise ValueError(
918                 "Complex data not supported\n{}\n".format(array)
919             ) from complex_warning

~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/utils/_array_ap
i.py in ?(array, dtype, order, copy, xp)
376         # Use NumPy API to support order
377         if copy is True:
378             array = numpy.array(array, order=order, dtype=dtype)
379         else:
--> 380             array = numpy.asarray(array, order=order, dtype=dtype)
381
382         # At this point array is a NumPy ndarray. We convert it to an array
383         # container that is consistent with the input's namespace.

~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/pandas/core/generic.py i
n ?(self, dtype)
2082     def __array__(self, dtype: npt.DTypeLike | None = None) -> np.ndarray:
2083         values = self._values
-> 2084         arr = np.asarray(values, dtype=dtype)
2085         if (
2086             astype_is_view(values.dtype, arr.dtype)
2087             and using_copy_on_write()
ValueError: could not convert string to float: 'Honda'

```

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](#).

But let's look at one of the most straightforward ways to turn categorical features into numbers, [one-hot encoding](#).

In machine learning, [one-hot encoding](#) 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` to one-hot encode our features and `sklearn.compose.ColumnTransformer` 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)` method.

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

```
In [37]: # 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

# 5. Turn the categorical features into numbers (this will return an array-like spa
```

```
transformed_X = transformer.fit_transform(X)
transformed_X
```

```
Out[37]: 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 [38]: X.head()
```

```
Out[38]:
```

	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 [39]: # View first transformed sample
transformed_X[0]
```

```
Out[39]: 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 [40]: # View original first sample
X.iloc[0]
```

```
Out[40]: Make          Honda
Colour         White
Odometer (KM)   35431
Doors           4
Name: 0, dtype: object
```

1.1.1 Numerically encoding data with pandas

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

We can use the `pandas.get_dummies()` (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 [41]: # View head of original DataFrame
car_sales.head()
```

Out [41]:

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

Out [42]:

	Doors	Make_BMW	Make_Honda	Make_Nissan	Make_Toyota	Colour_Black	Colour_Blue	Colour_Green
0	4	False	True	False	False	False	False	False
1	5	True	False	False	False	False	True	False
2	4	False	True	False	False	False	False	False

3	4	False	False	False	True	False	False	False
4	3	False	False	True	False	False	True	False
...
995	4	False	False	False	True	True	False	False
996	3	False	False	True	False	False	False	False
997	4	False	False	True	False	False	True	False
998	4	False	True	False	False	False	False	False
999	4	False	False	False	True	False	True	False

1000 rows × 10 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 [43]: # 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 [43]:
```

	Make_BMW	Make_Honda	Make_Nissan	Make_Toyota	Colour_Black	Colour_Blue	Colour_Green	Colour
--	----------	------------	-------------	-------------	--------------	-------------	--------------	--------

0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
1	1.0	0.0	0.0	0.0	0.0	1.0	0.0
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	1.0	0.0	0.0	0.0
4	0.0	0.0	1.0	0.0	0.0	1.0	0.0
...
995	0.0	0.0	0.0	1.0	1.0	0.0	0.0
996	0.0	0.0	1.0	0.0	0.0	0.0	0.0
997	0.0	0.0	1.0	0.0	0.0	1.0	0.0
998	0.0	1.0	0.0	0.0	0.0	0.0	0.0
999	0.0	0.0	0.0	1.0	0.0	1.0	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](#) (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 [44]: 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 fu
model.fit(X_train, y_train)

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

```
Out [44]: 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.

[PDFmyURL](#) converts web pages and even full websites to PDF easily and quickly.

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 referring to the [Scikit-Learn user guide](#).
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 [45]: # Import car sales dataframe with missing values
car_sales_missing = pd.read_csv("../data/car-sales-extended-missing-data.csv")
car_sales_missing
```

Out [45]:

	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 your 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()` which is able to detect missing values.

Let's try it on our DataFrame.

```
In [46]: # Get the sum of all missing values
car_sales_missing.isna().sum()
```

```
Out[46]: 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 [47]: # 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 [48]: # 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 [49]: # 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[49]: 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 [50]: # 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)
/Users/daniel/code/zero-to-mastery-ml/section-2-data-science-and-ml-tools/introduct
ion-to-scikit-learn.ipynb Cell 96 line 8
      <a href='vscode-notebook-cell:/Users/daniel/code/zero-to-mastery-ml/section-2
```

```

-data-science-and-ml-tools/introduction-to-scikit-learn.ipynb#Y164sZmlsZQ%3D%3D?line=5>6</a> # Fit and score a model
    <a href='vscode-notebook-cell:/Users/daniel/code/zero-to-mastery-ml/section-2-data-science-and-ml-tools/introduction-to-scikit-learn.ipynb#Y164sZmlsZQ%3D%3D?line=6'>7</a> model = RandomForestRegressor()
----> <a href='vscode-notebook-cell:/Users/daniel/code/zero-to-mastery-ml/section-2-data-science-and-ml-tools/introduction-to-scikit-learn.ipynb#Y164sZmlsZQ%3D%3D?line=7'>8</a> model.fit(X_train, y_train)
    <a href='vscode-notebook-cell:/Users/daniel/code/zero-to-mastery-ml/section-2-data-science-and-ml-tools/introduction-to-scikit-learn.ipynb#Y164sZmlsZQ%3D%3D?line=8'>9</a> model.score(X_test, y_test)

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/base.py:115
2, in _fit_context.<locals>.decorator.<locals>.wrapper(estimator, *args, **kwargs)
    1145     estimator._validate_params()
    1147     with config_context(
    1148         skip_parameter_validation=(
    1149             prefer_skip_nested_validation or global_skip_validation
    1150         )
    1151     ):
-> 1152     return fit_method(estimator, *args, **kwargs)

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/ensemble/_forest.py:348, in BaseForest.fit(self, X, y, sample_weight)
    346 if issparse(y):
    347     raise ValueError("sparse multilabel-indicator for y is not supported.")
--> 348 X, y = self._validate_data(
    349     X, y, multi_output=True, accept_sparse="csc", dtype=DTYPE
    350 )
    351 if sample_weight is not None:
    352     sample_weight = _check_sample_weight(sample_weight, X)

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/base.py:62
2, in BaseEstimator._validate_data(self, X, y, reset, validate_separately, cast_to_ndarray, **check_params)
    620     y = check_array(y, input_name="y", **check_y_params)
    621     else:

```

```

--> 622         X, y = check_X_y(X, y, **check_params)
      623         out = X, y
      625 if not no_val_X and check_params.get("ensure_2d", True):

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/utils/validation.py:1146, in check_X_y(X, y, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, multi_output, ensure_min_samples, ensure_min_features, y_numeric, estimator)
      1141         estimator_name = _check_estimator_name(estimator)
      1142         raise ValueError(
      1143             f"{estimator_name} requires y to be passed, but the target y is None"
      1144         )
-> 1146 X = check_array(
      1147     X,
      1148     accept_sparse=accept_sparse,
      1149     accept_large_sparse=accept_large_sparse,
      1150     dtype=dtype,
      1151     order=order,
      1152     copy=copy,
      1153     force_all_finite=force_all_finite,
      1154     ensure_2d=ensure_2d,
      1155     allow_nd=allow_nd,
      1156     ensure_min_samples=ensure_min_samples,
      1157     ensure_min_features=ensure_min_features,
      1158     estimator=estimator,
      1159     input_name="X",
      1160 )
      1162 y = _check_y(y, multi_output=multi_output, y_numeric=y_numeric, estimator=estimator)
      1164 check_consistent_length(X, y)

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/utils/validation.py:957, in check_array(array, accept_sparse, accept_large_sparse, dtype, order, copy, force_all_finite, ensure_2d, allow_nd, ensure_min_samples, ensure_min_features, estimator, input_name)
      951         raise ValueError(

```



```

952         "Found array with dim %d. %s expected <= 2."
953         % (array.ndim, estimator_name)
954     )
955     if force_all_finite:
--> 956         _assert_all_finite(
957             array,
958             input_name=input_name,
959             estimator_name=estimator_name,
960             allow_nan=force_all_finite == "allow-nan",
961         )
962     if ensure_min_samples > 0:
963         n_samples = _num_samples(array)

```

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/utils/validation.py:122, in _assert_all_finite(X, allow_nan, msg_dtype, estimator_name, input_name)

```

119     if first_pass_isfinite:
120         return
--> 121     _assert_all_finite_element_wise(
122         X,
123         xp=xp,
124         allow_nan=allow_nan,
125         msg_dtype=msg_dtype,
126         estimator_name=estimator_name,
127         input_name=input_name,
128     )

```

File ~/code/zero-to-mastery-ml/env/lib/python3.10/site-packages/sklearn/utils/validation.py:171, in _assert_all_finite_element_wise(X, xp, allow_nan, msg_dtype, estimator_name, input_name)

```

154     if estimator_name and input_name == "X" and has_nan_error:
155         # Improve the error message on how to handle missing values in
156         # scikit-learn.
157         msg_err += (
158             f"\n{estimator_name} does not accept missing values"
159             " encoded as NaN natively. For supervised learning, you might want"
160             (...)

```

```
169         "#estimators-that-handle-nan-values"
170     )
--> 171 raise ValueError(msg_err)

ValueError: Input X contains NaN.
RandomForestRegressor does not accept missing values encoded as NaN natively. For supervised learning, you might want to consider sklearn.ensemble.HistGradientBoostingClassifier and Regressor which accept missing values encoded as NaNs natively. Alternatively, it is possible to preprocess the data, for instance by using an imputer transformer in a pipeline or drop samples with missing values. See https://scikit-learn.org/stable/modules/impute.html You can find a list of all estimators that handle NaN values at the following page: https://scikit-learn.org/stable/modules/impute.html#estimators-that-handle-nan-values
```

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.