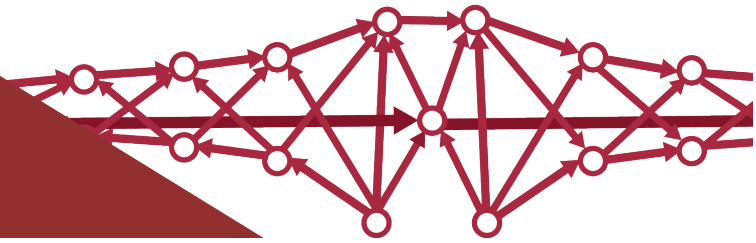
Machine Learning
& Data Analysis



Course notes

MLADA · 2026

Elliot J. Crowley

University of Edinburgh

elliotjcrowley.com/mlada.html

Course logo designed by Joe Mellor.

About these notes

This book collects the nine sets of notes for Machine Learning and Data Analysis at the University of Edinburgh. It combines explanations, equations, Python examples and the notebooks' saved outputs.

Chapter numbers follow the teaching weeks. Week 6, Machine learning in society, has lecture slides but no accompanying notebook, so there is no Chapter 6. All lecture slides and downloadable notebooks are available on the course page.

Working with the examples

Code cells within each chapter should be run in order in the corresponding notebook. Each chapter starts with its imports and setup. Download and extract the notebook ZIP before running it, keeping the accompanying data folder beside the notebook.

Some examples require online data or additional software. Where their output is not included, an **Execution note** explains the requirements: the penguin plots in Chapter 4 need online data; the Graphviz diagrams in Chapter 7 need Graphviz; and the ConvNet training example in Chapter 10 needs PyTorch and skorch. Requirements are listed beside each affected example.

Credits and licence

Written by Elliot J. Crowley. © The University of Edinburgh 2022–2026. Course materials are licensed under CC BY-NC-SA 4.0. Linked and third-party material retains its own attribution and licence. The downloadable MNIST data has a separate MIT licence notice.

Thanks to Joe Mellor, Amos Storkey, Justin Engelmann, Tom Lee and Henry Gouk for their help in developing the course. The course logo was designed by Joe Mellor.

Contents

About these notes i

1 Representing data 1
- 1.1 What are these notebooks for? . . . 1
- 1.2 Data modalities . . . 1
- 1.3 Preprocessing . . . 10

2 Linear models for regression 16
- 2.1 Simple linear regression . . . 17
- 2.2 Minimising squared error . . . 19
- 2.3 Multiple linear regression . . . 22
- 2.4 Evaluating regression models . . . 24
- 2.5 Polynomial regression . . . 25
- 2.6 Linear regression with regularisation . . . 29

3 Linear models for classification 32
- 3.1 Linear models for binary classification . . . 32
- 3.2 Logistic regression . . . 33
- 3.3 Logistic regression example . . . 35
- 3.4 Perceptrons [Optional] . . . 38
- 3.5 Support Vector Machines . . . 39
- 3.6 Multinomial Logistic Regression . . . 39

4 Visualising data 41
- 4.1 Plots . . . 41
- 4.2 Dimensionality reduction . . . 51

5 Model selection and evaluation 57
- 5.1 k-nearest neighbours (k-NN) . . . 57
- 5.2 Generative models [Optional] . . . 58
- 5.3 Model selection . . . 65
- 5.4 Cross-validation . . . 65
- 5.5 Grid search with k-fold cross validation . . . 66
- 5.6 Pipelines . . . 67
- 5.7 Evaluating binary classifiers . . . 67

7 Decision trees and ensemble methods 70
- 7.1 Non-parametric models . . . 70
- 7.2 Classification trees . . . 71
- 7.3 Classification tree learning - the CART algorithm . . . 75
- 7.4 Regression trees . . . 79
- 7.5 Ensembles . . . 81
- 7.6 Bagging . . . 81
- 7.7 Gradient Boosting . . . 83

8 Gaussian processes **86**
8.1 Continuous random variables and the Gaussian distribution 86
8.2 Joint pdfs and the Bivariate Gaussian 88
8.3 The Multivariate Gaussian 91
8.4 A Gaussian Process model 92
8.5 Gaussian process prior 93
8.6 Gaussian Process posterior 96
8.7 Dealing with noisy measurements 99
8.8 Optimising kernel hyperparameters 101

9 Deep neural networks **104**
9.1 Feature learning 104
9.2 Multilayer perceptrons (MLPs) 105
9.3 Learning the parameters of an MLP 107
9.4 MLPs in sklearn 107

10 Neural architectures and transfer learning **114**
10.1 Convolutional Neural Networks (ConvNets) 114
10.2 Feature extraction 117
10.3 Fine-tuning 117

1 Representing data

1.1 What are these notebooks for?

Most weeks of the course have a lecture and a workshop. In the lecture you are introduced to material, and in the workshop you use Python to solve problems using that material.

These notebooks act as a bridge between the lecture and the workshop. Each one summarises the lecture, and provides code examples as a starting point for writing your own code. In this notebook, I will import packages on-the-fly as needed so you can see when they are required. In the later notebooks, and in the workshop, imports will be done at the start.

The first lecture was slightly unusual, in that it was also an introduction to the course. I am not going to summarise the introduction (whatever that would entail) but I am going to look at different data modalities, and how you can process them in Python.

1.2 Data modalities

1.2.1 Time series

In time series data, we have some quantity we care about at different points in time. We are going to consider the value of the pound (GBP) vs. the dollar (USD). A lot of the time we import our data from spreadsheets (e.g. excel files, CSV files) and this is no exception!

I have already downloaded the spreadsheet containing GBP vs. USD values from 04/10/2021 to 03/10/2022 from Yahoo Finance which is in CSV format.

How do we get this into Python? There is a fantastic data analysis library called pandas which does all the hard work for us.

```
# Import pandas for dataframes
import pandas as pd

# Read CSV into dataframe using pandas
df = pd.read_csv("data/GBPUSD=X.csv")

# Show dataframe
df
```

	Date	Open	High	Low	Close	Adj Close	Volume
0	2021-10-04	1.357092	1.363977	1.353363	1.357092	1.357092	0
1	2021-10-05	1.360766	1.365001	1.358585	1.360748	1.360748	0
2	2021-10-06	1.362565	1.363141	1.354463	1.362769	1.362769	0
3	2021-10-07	1.359268	1.363736	1.357184	1.359213	1.359213	0
4	2021-10-08	1.361953	1.365374	1.358493	1.362027	1.362027	0
..	...	...	...	...	...	...	...
256	2022-09-27	1.077006	1.083541	1.073849	1.077284	1.077284	0
257	2022-09-28	1.072846	1.083964	1.054163	1.072754	1.072754	0
258	2022-09-29	1.083459	1.106562	1.076449	1.083600	1.083600	0

```
259  2022-09-30  1.116184  1.123394  1.102682  1.116944   1.116944       0
260  2022-10-03  1.118456  1.127485  1.108672  1.122914   1.122914       0

[261 rows x 7 columns]
```

Let's say we care about the closing value at the end of each day. This is in the `Close` column. We can plot this against `Date` using a few lines of code.

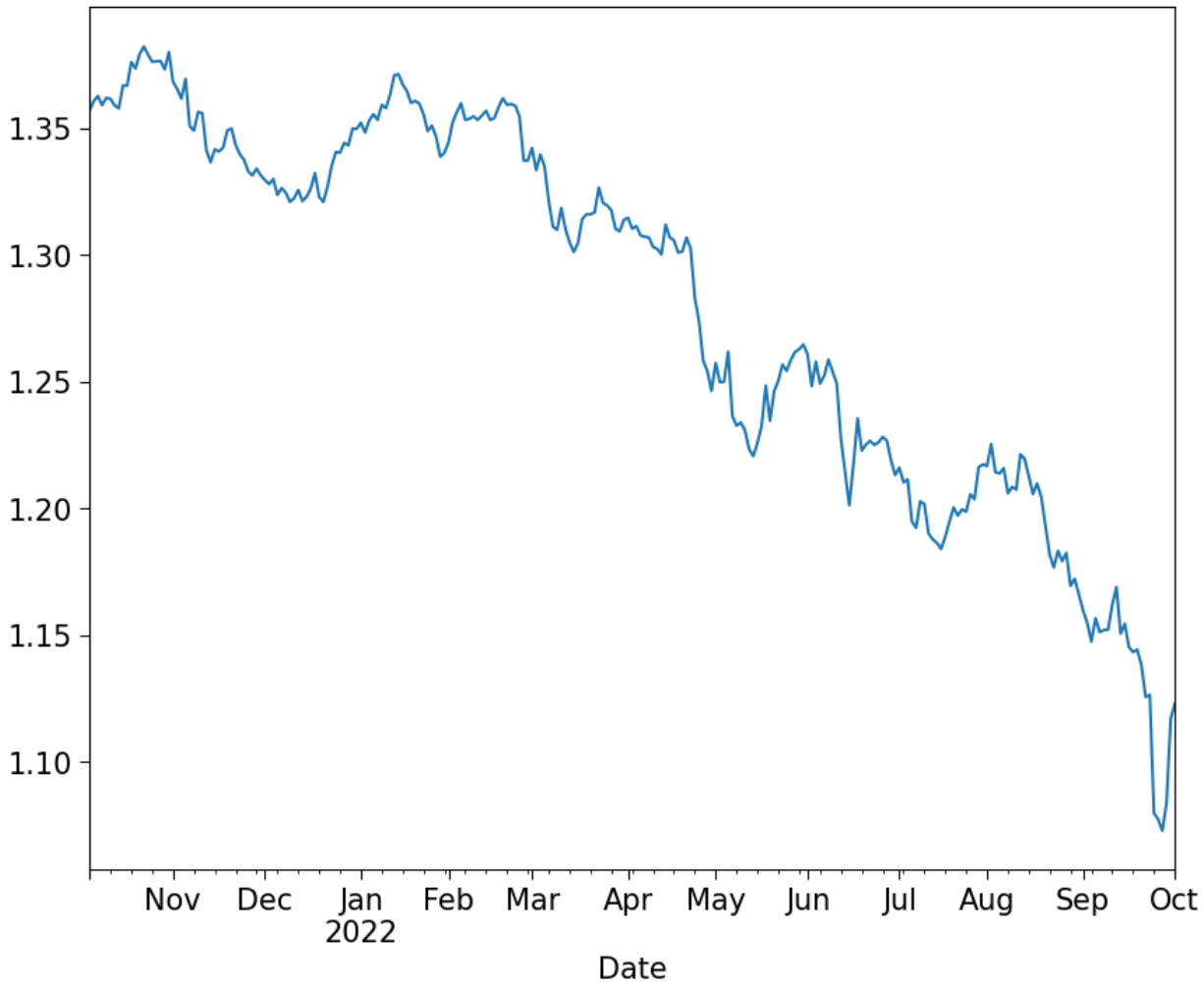
```
# matplotlib for plotting
import matplotlib.pyplot as plt

# This makes matplotlib output nice figures without much tweaking
plt.rcParams.update(
    {
        "lines.markersize": 10,  # Big points
        "font.size": 15,  # Larger font
        "xtick.major.size": 5.0,  # Bigger xticks
        "ytick.major.size": 5.0,  # Bigger yticks
    }
)

# Plotting code
fig, ax = plt.subplots(figsize=[8,6])
ax.plot(df["Date"], df["Close"])
ax.set_xlabel('Date')
ax.set_ylabel('GBP to USD rate')
ax.set_title('GBP vs USD from October 2021-2022')

# Add a grid
ax.grid()

# Formatting specifically for date strings
fig.autofmt_xdate()

# A hacky way of showing a subset of x ticks
dates = df["Date"].to_numpy()
ax.set_xticks([dates[i] for i in range(0, len(dates), 50)])
plt.show()
```

There is an alternative way to do this just in pandas. Notice that each row has an index (0,1,2,3...). We can turn the `Date` column into a datetime format, and then make that the index instead. We can then just call `.plot` on the dataframe to plot a column against the date index.

Let's do all this on a copy of the original dataframe:

```
df2 = df.copy()
df2['Date'] = pd.to_datetime(df2['Date'])
df2 = df2.set_index("Date")
df2["Close"].plot(figsize=[10,8])
```

```
<Axes: xlabel='Date'>
```



We can also take a column and convert it into a numpy array. Let's take the `Close` column, convert it into an array, and find its mean. (This can be done directly in pandas but is for illustration!)

```
# import numpy for arrays
import numpy as np

close_array = df["Close"].values
close_mean = np.mean(close_array)
print(f"The mean close value is {close_mean:3f}")
```

```
The mean close value is 1.280153
```

1.2.2 Tabular data

We actually did just extract our time series data from tabular data! Tabular data will make up the vast majority of data you look at in this course. It is very common in the real world.

Let's say we want the list of dates for which the opening value of GBP vs. USD was greater than 1.3.

First, we can see which rows this corresponds to:

```
valid_rows = df["Open"] > 1.3
print(valid_rows)
```

```
0       True
1       True
2       True
3       True
4       True
       ...
256    False
257    False
258    False
259    False
260    False
Name: Open, Length: 261, dtype: bool
```

We can then use this boolean array to filter out the appropriate dates.

```
df["Date"][valid_rows]
```

```
0      2021-10-04
1      2021-10-05
2      2021-10-06
3      2021-10-07
4      2021-10-08
          ...
140    2022-04-18
141    2022-04-19
142    2022-04-20
143    2022-04-21
144    2022-04-22
Name: Date, Length: 145, dtype: object
```

If we want to get the rows that correspond to dates in the year 2022, we can use the following to get a Boolean array, and filter with it:

```
valid_rows = df["Date"].str.contains("2022")
df[valid_rows]
```

```
           Date      Open      High       Low     Close  Adj Close  Volume
65   2022-01-03  1.352228  1.353180  1.343274  1.352228   1.352228       0
66   2022-01-04  1.348279  1.355675  1.346076  1.348327   1.348327       0
67   2022-01-05  1.353052  1.357331  1.352265  1.353143   1.353143       0
68   2022-01-06  1.355546  1.355748  1.349109  1.355565   1.355565       0
69   2022-01-07  1.353400  1.358345  1.352942  1.353363   1.353363       0
..          ...       ...       ...       ...       ...        ...     ...
256  2022-09-27  1.077006  1.083541  1.073849  1.077284   1.077284       0
257  2022-09-28  1.072846  1.083964  1.054163  1.072754   1.072754       0
258  2022-09-29  1.083459  1.106562  1.076449  1.083600   1.083600       0
259  2022-09-30  1.116184  1.123394  1.102682  1.116944   1.116944       0
260  2022-10-03  1.118456  1.127485  1.108672  1.122914   1.122914       0

[196 rows x 7 columns]
```

1.2.3 Images

PIL (the Python imaging library) lets us read in images, as well as perform high-level manipulations. We will load in a JPEG of a dog, and also print the image size in pixels.

```
# Image from PIL lets us manipulate images
from PIL import Image

# Read image
image = Image.open("data/dog.jpg")

# Use Juypter's inbuilt display function
display(image)
print(f"The image has a size of {image.size}")
```

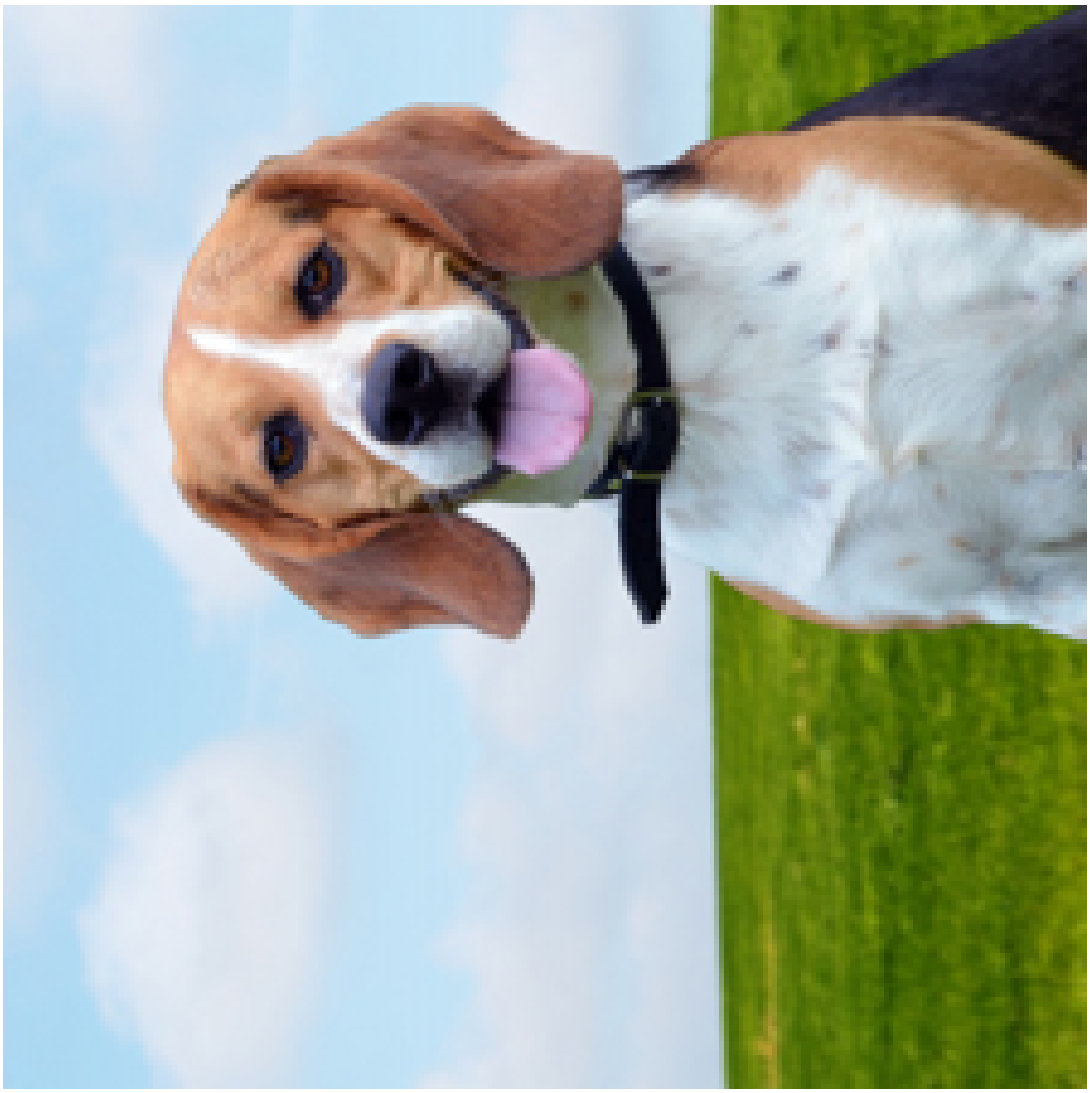
```
The image has a size of (615, 615)
```

We can downsize the image...

```
image = image.resize((224, 224))
display(image)
```

and rotate it!

```
image = image.rotate(90)
display(image)
```



Recall from the lecture that an image is actually stored as a 3D array (height by width by colour channel). We can see this if we convert our image into a numpy array.

```
# Convert image into a numpy array
data_im = np.array(image)

# See what the array looks like and print its shape
print(data_im)
print(f"the image in numpy has shape {data_im.shape}")
```

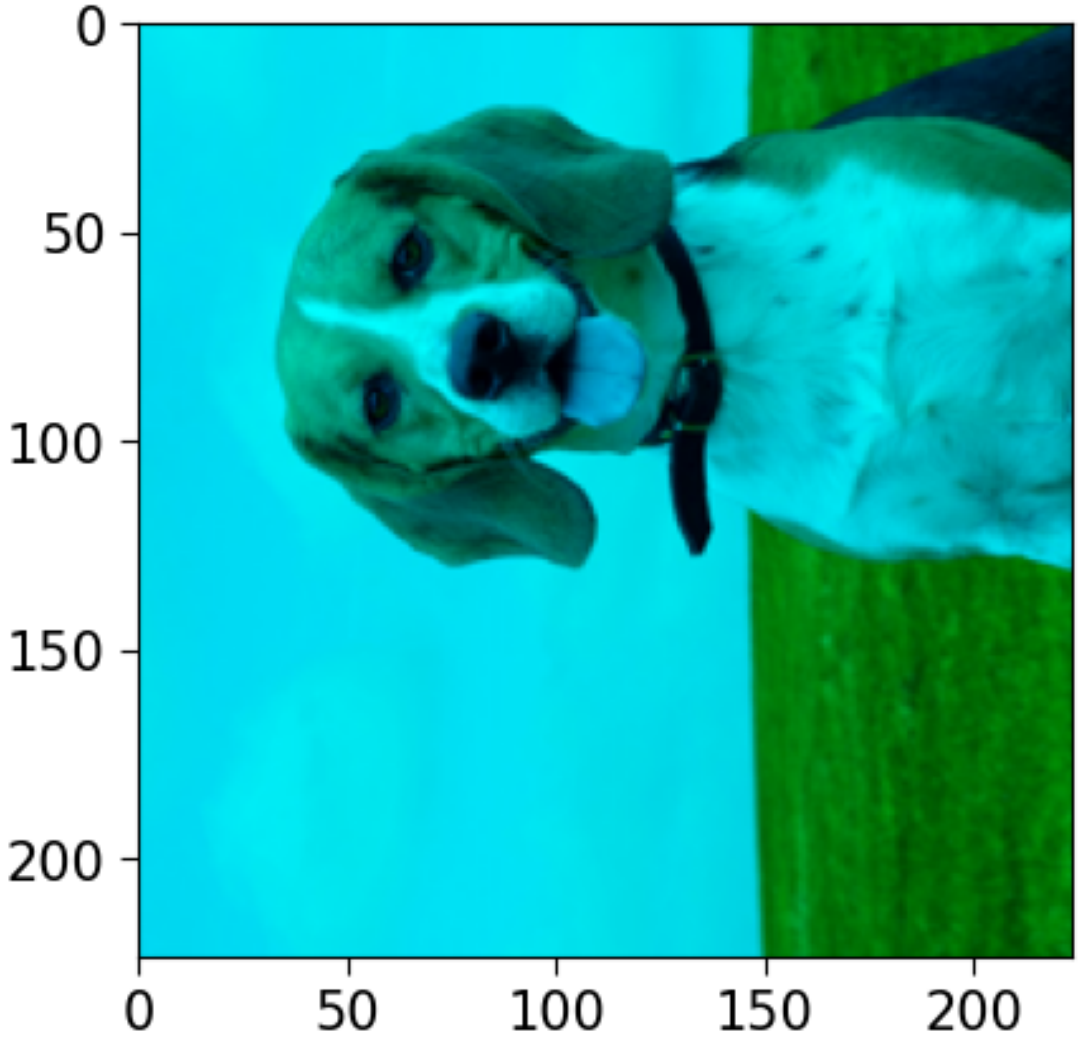
```
[[[217 229 243]
  [217 229 243]
  [218 229 242]
  ...
  [ 47  66  24]
  [ 50  57  52]
  [ 57  56  73]]

 [[216 230 243]
  [217 230 243]
  [217 229 242]
  ...
  [ 53  57  71]
  [ 60  60  83]
```

```
   [ 61  57  81]]

  [[216 230 243]
   [216 230 243]
   [216 229 243]
   ...
   [ 60  58  82]
   [ 55  53  78]
   [ 53  50  71]]

  ...

  [[171 214 239]
   [170 214 240]
   [170 216 242]
   ...
   [114 131  18]
   [110 128  15]
   [101 119   5]]

  [[170 214 239]
   [170 215 240]
   [170 217 242]
   ...
   [111 128  18]
   [113 128  15]
   [110 126  12]]

  [[170 215 240]
   [170 216 240]
   [171 217 241]
   ...
   [ 94 110   6]
   [112 123  12]
   [124 135  22]]]
the image in numpy has shape (224, 224, 3)
```

Let's manually set the red colour channel to zero for all pixels and see what happens.

```
# Create a copy of our image array
data_nored = data_im.copy()

# Set all values in the red channel to 0
data_nored[:, :, 0] = 0

# Now we have a numpy array we have to use matplotlib to display it
plt.imshow(data_nored)
```

```
<matplotlib.image.AxesImage at 0x12c9e9eb0>
```



Without any red, the blue and green have got more ... blue and green. This wasn't the most obvious manipulation! Try some others by changing the code above.

1.3 Preprocessing

1.3.1 Vectors and matrices

Many machine learning (ML) techniques require us to represent our data points as vectors of real values, each of the same dimensionality D. The elements of these vectors correspond to variables, or **features**. In machine learning it is convention to use *column* vectors, so we will stick to this.

We can refer to some arbitrary data point as $\mathbf{x} \in \mathbb{R}^D$ and if we want to refer to specific data points in our dataset of N points we can write $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \mathbf{x}^{(3)}, \ldots, \mathbf{x}^{(N)}$ where the ordering isn't important as long as it is consistent.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \qquad \mathbf{x}^{(n)} = \begin{bmatrix} x_1^{(n)} \\ x_2^{(n)} \\ \vdots \\ x_D^{(n)} \end{bmatrix}$$

The whole dataset is the collection of these vectors $\{\mathbf{x}^{(n)}\}_{n=1}^{N}$ and can be stored in a dataset matrix $\mathbf{X}$. Annoyingly, it is convention to have the *rows* of this matrix representing different data points so that $\mathbf{X} \in \mathbb{R}^{N \times D}$. I tend to refer to this as a dataset matrix, but it is often referred to as a design matrix in the literature.

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \\ \vdots \\ \mathbf{x}^{(N)\top} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_D^{(2)} \\ x_1^{(3)} & x_2^{(3)} & \dots & x_D^{(3)} \\ \dots & \dots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_D^{(N)} \end{bmatrix}$$

Given some tabular data, we need to convert it into this $\mathbf{X}$ format. If the features in our table are all continuous variables then this is very straightforward. In fact, with a table represented as a pandas dataframe it only requires a single line of code.

Let's look at the iris dataset from the lecture. We will get the dataframe from sklearn and then convert it into a dataset matrix.

> **sklearn** (or Scikit-learn) is a high-level ML library for Python. It integrates with NumPy for arrays, SciPy for optimisation, and Pandas for dataframes (tabular data). It is really, really good and we will use it heavily in this course. It is high level, which lets us spend less time hacking together algorithms ourselves, and more time seeing how they work on data.

```
# Get loader for the iris dataset and import pandas and numpy
from sklearn.datasets import load_iris
import pandas as pd
import numpy as np

# Convert into dataframe with the columns as the feature names
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)

df
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width
      ↪  (cm)
0                  5.1               3.5                1.4
 ↪  0.2
1                  4.9               3.0                1.4
 ↪  0.2
2                  4.7               3.2                1.3
 ↪  0.2
3                  4.6               3.1                1.5
 ↪  0.2
4                  5.0               3.6                1.4
 ↪  0.2
..                 ...               ...                ...
 ↪  ...
145                6.7               3.0                5.2
 ↪  2.3
146                6.3               2.5                5.0
 ↪  1.9
147                6.5               3.0                5.2
 ↪  2.0
148                6.2               3.4                5.4
 ↪  2.3
```

```
149                5.9               3.0                5.1
 ↪  1.8

[150 rows x 4 columns]
```

To convert this into $\mathbf{X} \in \mathbb{R}^{N \times D}$ we just have to run a line of code.

```python
# Convert dataframe into array
X = df.values

# Check the shape of X
print(f"X has shape {X.shape}")
print(X)
```

```
X has shape (150, 4)
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]
 [5.4 3.9 1.7 0.4]
 [4.6 3.4 1.4 0.3]
 [5.  3.4 1.5 0.2]
 [4.4 2.9 1.4 0.2]
 [4.9 3.1 1.5 0.1]
 [5.4 3.7 1.5 0.2]
 [4.8 3.4 1.6 0.2]
 [4.8 3.  1.4 0.1]
 [4.3 3.  1.1 0.1]
 [5.8 4.  1.2 0.2]
 [5.7 4.4 1.5 0.4]
 [5.4 3.9 1.3 0.4]
 [5.1 3.5 1.4 0.3]
 [5.7 3.8 1.7 0.3]
 [5.1 3.8 1.5 0.3]
 [5.4 3.4 1.7 0.2]
 [5.1 3.7 1.5 0.4]
 [4.6 3.6 1.  0.2]
 [5.1 3.3 1.7 0.5]
 [4.8 3.4 1.9 0.2]
 [5.  3.  1.6 0.2]
 [5.  3.4 1.6 0.4]
 [5.2 3.5 1.5 0.2]
 [5.2 3.4 1.4 0.2]
 [4.7 3.2 1.6 0.2]
 [4.8 3.1 1.6 0.2]
 [5.4 3.4 1.5 0.4]
 [5.2 4.1 1.5 0.1]
 [5.5 4.2 1.4 0.2]
 [4.9 3.1 1.5 0.2]
 [5.  3.2 1.2 0.2]
 [5.5 3.5 1.3 0.2]
 [4.9 3.6 1.4 0.1]
 [4.4 3.  1.3 0.2]
 [5.1 3.4 1.5 0.2]
 [5.  3.5 1.3 0.3]
 [4.5 2.3 1.3 0.3]
 [4.4 3.2 1.3 0.2]
 [5.  3.5 1.6 0.6]
 [5.1 3.8 1.9 0.4]
 [4.8 3.  1.4 0.3]
 [5.1 3.8 1.6 0.2]
 [4.6 3.2 1.4 0.2]
 [5.3 3.7 1.5 0.2]
```

```
[5.  3.3 1.4 0.2]
[7.  3.2 4.7 1.4]
[6.4 3.2 4.5 1.5]
[6.9 3.1 4.9 1.5]
[5.5 2.3 4.  1.3]
[6.5 2.8 4.6 1.5]
[5.7 2.8 4.5 1.3]
[6.3 3.3 4.7 1.6]
[4.9 2.4 3.3 1. ]
[6.6 2.9 4.6 1.3]
[5.2 2.7 3.9 1.4]
[5.  2.  3.5 1. ]
[5.9 3.  4.2 1.5]
[6.  2.2 4.  1. ]
[6.1 2.9 4.7 1.4]
[5.6 2.9 3.6 1.3]
[6.7 3.1 4.4 1.4]
[5.6 3.  4.5 1.5]
[5.8 2.7 4.1 1. ]
[6.2 2.2 4.5 1.5]
[5.6 2.5 3.9 1.1]
[5.9 3.2 4.8 1.8]
[6.1 2.8 4.  1.3]
[6.3 2.5 4.9 1.5]
[6.1 2.8 4.7 1.2]
[6.4 2.9 4.3 1.3]
[6.6 3.  4.4 1.4]
[6.8 2.8 4.8 1.4]
[6.7 3.  5.  1.7]
[6.  2.9 4.5 1.5]
[5.7 2.6 3.5 1. ]
[5.5 2.4 3.8 1.1]
[5.5 2.4 3.7 1. ]
[5.8 2.7 3.9 1.2]
[6.  2.7 5.1 1.6]
[5.4 3.  4.5 1.5]
[6.  3.4 4.5 1.6]
[6.7 3.1 4.7 1.5]
[6.3 2.3 4.4 1.3]
[5.6 3.  4.1 1.3]
[5.5 2.5 4.  1.3]
[5.5 2.6 4.4 1.2]
[6.1 3.  4.6 1.4]
[5.8 2.6 4.  1.2]
[5.  2.3 3.3 1. ]
[5.6 2.7 4.2 1.3]
[5.7 3.  4.2 1.2]
[5.7 2.9 4.2 1.3]
[6.2 2.9 4.3 1.3]
[5.1 2.5 3.  1.1]
[5.7 2.8 4.1 1.3]
[6.3 3.3 6.  2.5]
[5.8 2.7 5.1 1.9]
[7.1 3.  5.9 2.1]
[6.3 2.9 5.6 1.8]
[6.5 3.  5.8 2.2]
[7.6 3.  6.6 2.1]
[4.9 2.5 4.5 1.7]
[7.3 2.9 6.3 1.8]
[6.7 2.5 5.8 1.8]
[7.2 3.6 6.1 2.5]
[6.5 3.2 5.1 2. ]
[6.4 2.7 5.3 1.9]
[6.8 3.  5.5 2.1]
[5.7 2.5 5.  2. ]
[5.8 2.8 5.1 2.4]
```

```
[6.4 3.2 5.3 2.3]
[6.5 3.  5.5 1.8]
[7.7 3.8 6.7 2.2]
[7.7 2.6 6.9 2.3]
[6.  2.2 5.  1.5]
[6.9 3.2 5.7 2.3]
[5.6 2.8 4.9 2. ]
[7.7 2.8 6.7 2. ]
[6.3 2.7 4.9 1.8]
[6.7 3.3 5.7 2.1]
[7.2 3.2 6.  1.8]
[6.2 2.8 4.8 1.8]
[6.1 3.  4.9 1.8]
[6.4 2.8 5.6 2.1]
[7.2 3.  5.8 1.6]
[7.4 2.8 6.1 1.9]
[7.9 3.8 6.4 2. ]
[6.4 2.8 5.6 2.2]
[6.3 2.8 5.1 1.5]
[6.1 2.6 5.6 1.4]
[7.7 3.  6.1 2.3]
[6.3 3.4 5.6 2.4]
[6.4 3.1 5.5 1.8]
[6.  3.  4.8 1.8]
[6.9 3.1 5.4 2.1]
[6.7 3.1 5.6 2.4]
[6.9 3.1 5.1 2.3]
[5.8 2.7 5.1 1.9]
[6.8 3.2 5.9 2.3]
[6.7 3.3 5.7 2.5]
[6.7 3.  5.2 2.3]
[6.3 2.5 5.  1.9]
[6.5 3.  5.2 2. ]
[6.2 3.4 5.4 2.3]
[5.9 3.  5.1 1.8]]
```

1.3.2 Standardisation

Standardising your data is **extremely important for many ML algorithms.** It allows features to be compared on a like-for-like basis so that those with naturally large measurements don't dominate.

For some of the toy examples in the lectures and notebooks where I have generated synthetic data, I haven't bothered with standardisation. There are two reasons for this:

1. It becomes more fiddly to make the kind of data I want to use to illustrate a point
2. The features in these synthetic datasets are all at similar scales and things still works (they are *roughly* standardised)

This will rarely be the case with real data! When in doubt, standardise your data, seriously!

To do this, for each feature we compute the mean and standard deviation (SD) of the measurements across all data points. For each measurement of that feature, we subtract the mean and divide by the SD.

$$\mathbf{X}_{new} = \begin{bmatrix} \frac{x_1^{(1)}-\mu_1}{\sigma_1} & \frac{x_2^{(1)}-\mu_2}{\sigma_2} & \cdots & \frac{x_D^{(1)}-\mu_D}{\sigma_D} \\ \frac{x_1^{(2)}-\mu_1}{\sigma_1} & \frac{x_2^{(2)}-\mu_2}{\sigma_2} & \cdots & \frac{x_D^{(2)}-\mu_D}{\sigma_D} \\ \cdots & \cdots & \ddots & \vdots \\ \frac{x_1^{(N)}-\mu_1}{\sigma_1} & \frac{x_2^{(N)}-\mu_2}{\sigma_2} & \cdots & \frac{x_D^{(N)}-\mu_D}{\sigma_D} \end{bmatrix}$$

where $\mu_j = \frac{1}{N}\sum_n \mathbf{x}_j^{(n)}$ and $\sigma_j^2 = \frac{1}{N}\sum_n (\mathbf{x}_j^{(n)} - \mu_j)^2$.

We can do this very easily in sklearn using the StandardScaler class.

```
# Make a copy of our dataset to compare stats after
X_old = np.array(df)

# Import scaler
from sklearn.preprocessing import StandardScaler

# Make a scaler object
scaler = StandardScaler()

# Fit the scaler to the dataset. This means computing the means and
 ↪  STDs
scaler.fit(X)

# Finally apply the scaler to transform the data
X = scaler.transform(X)


# Compare stats
print(
    f"Before standardisation feature means were
     ↪  {np.round(X_old.mean(0),2)} and SDs were
     ↪  {np.round(X_old.std(0),2)}"
)
print(
    f"After standardisation feature means are {np.round(X.mean(0),2)}
     ↪  and SDs are {np.round(X.std(0),2)}"
)
```

```
Before standardisation feature means were [5.84 3.06 3.76 1.2 ] and SDs
 ↪  were [0.83 0.43 1.76 0.76]
After standardisation feature means are [-0. -0. -0. -0.] and SDs are [1.
 ↪  1. 1. 1.]
```

2 Linear models for regression

```
# RUN THIS CELL FIRST

# Imports
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# sklearn
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.metrics import mean_squared_error, r2_score

# Better plots
plt.rcParams.update(
    {
        "lines.markersize": 10,   # Big points
        "font.size": 15,   # Larger font
        "xtick.major.size": 5.0,   # Bigger xticks
        "ytick.major.size": 5.0,   # Bigger yticks
    }
)
```

The goal in regression is to learn a model that can predict what a real-valued target $y \in \mathbb{R}$ should be for some data point $\mathbf{x} \in \mathbb{R}^D$. It is often appealing to use **linear models** as these are simple, intepretable, and can be effective. A linear model takes the form

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} = \hat{y}$$

where $\hat{y}$ is the model's prediction of the target for $\mathbf{x}$. **When we write the maths for regression, we usually make the assumption that each of our data points has been prepended with a** 1 so that $\mathbf{x} = \begin{bmatrix} 1 & x_1 & x_2 & \dots \end{bmatrix}^\top$.

$\mathbf{w}$ is a learnable vector of parameters $\mathbf{w} = \begin{bmatrix} b & w_1 & w_2 & \dots \end{bmatrix}^\top$ where $w_1, w_2, \dots$ are weights and b is a bias. The prediction is just a weighted combination of the input features plus the bias:

$$\hat{y} = f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} = b + \sum_{d=1}^{D} w_d x_d$$

We use a **training dataset** $\mathfrak{D}_{train}$ to learn what the elements of $\mathbf{w}$ should be. $\mathfrak{D}_{train}$ is a set of N data point-target pairs $\{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$. It tells our model what the predictions *should be* for a discrete set of points.

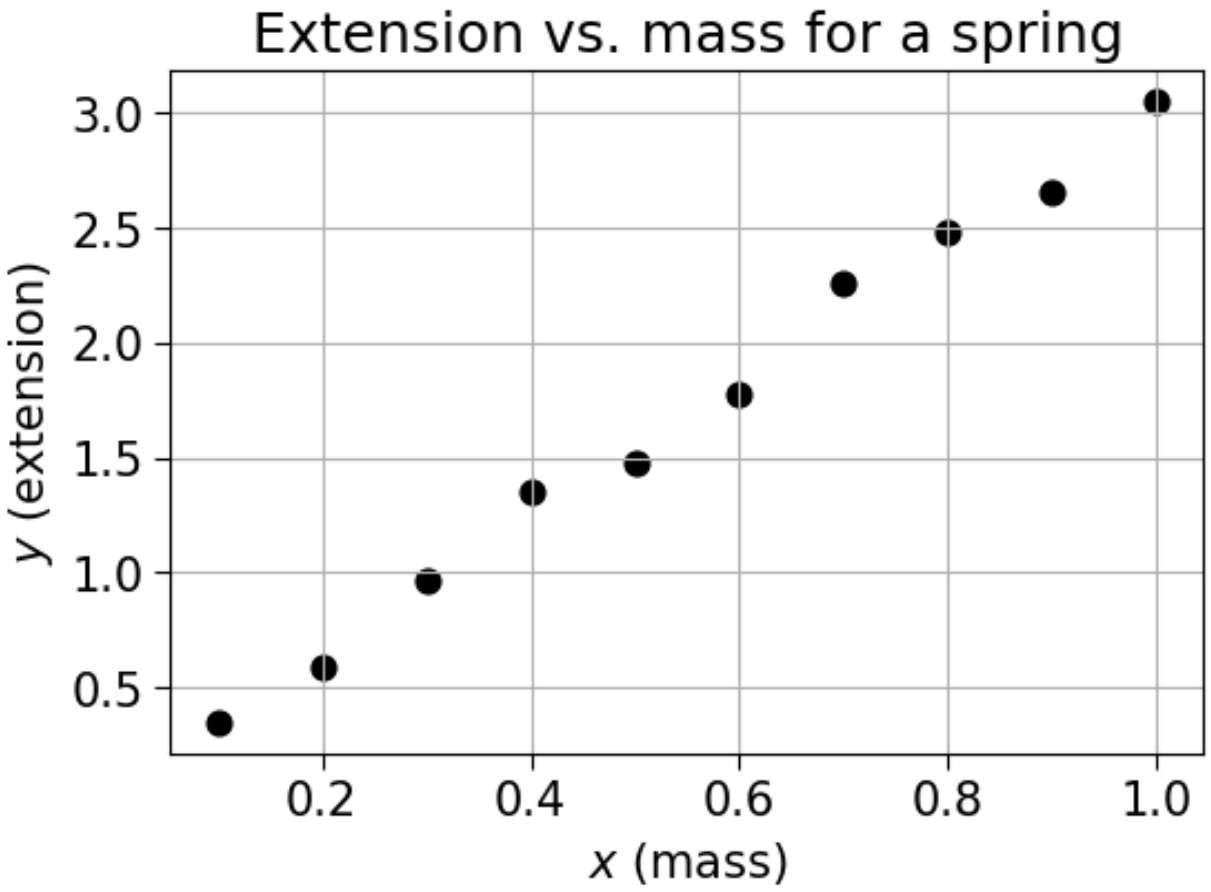
The standard machine learning approach is to devise a loss function and find (/learn/fit) the model parameters that minimise it. A loss function is a function of the training data and model parameters that is **low** when the model does what we want.

2.1 Simple linear regression

In simple linear regression the input data points are 1D i.e. $x \in \mathbb{R}$. We could for example, want to model the extension of a spring y given the mass x attached to it (ignoring that x is typically the symbol for extension in most literature :)).

Let's generate some synthetic training data $\mathfrak{D}_{train} = \{(x^{(n)}, y^{(n)})\}_{n=1}^{10}$ for this:

```
# Seed to keep things the same between runs
np.random.seed(42)

# Let's simulate some measurements, accepting that they will be noisy!
x = np.linspace(0.1, 1, 10)
y = np.linspace(0.1, 1, 10) * 3 + np.random.normal(size=10) * 0.1

# Plot y vs x as a scatter plot
fig, ax = plt.subplots(figsize=[6, 4])
ax.scatter(x, y, s=60, color="k")
ax.grid()
ax.set_xlabel("$x$ (mass)")
ax.set_ylabel("$y$ (extension)")
ax.set_title("Extension vs. mass for a spring")
plt.show()
```



You may have noticed that I added some noise to the target values. Noise in targets (or indeed, in the actual data) is an inevitability. Perhaps some of these measurements were taken with a

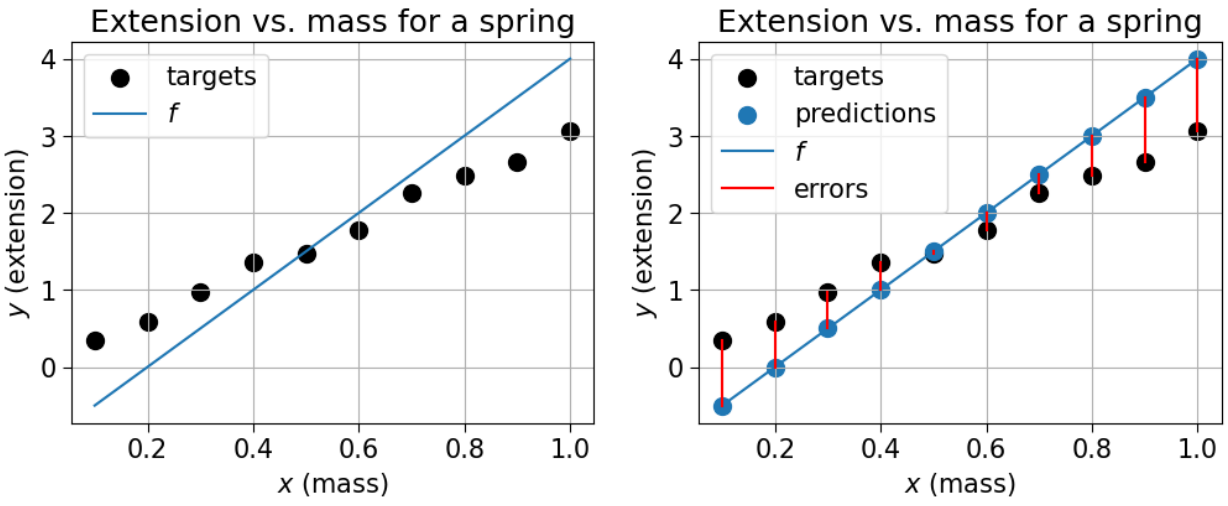
different ruler, or incorrectly, or the spring deformed slightly, or the masses were wrong. None of these things happened because this is synthetic data, but hopefully it helped to build the picture.

A linear model in this case is given by

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} = \begin{bmatrix} b & w \end{bmatrix} \begin{bmatrix} 1 \\ x \end{bmatrix} = wx + b$$

The neat thing about working in 1D is that we can draw models alongside our data. Let's consider a model with arbitrary parameters $w = 5$ and $b = -1$ and produce some plots.

```
# Do two plots side by side
fig, axes = plt.subplots(1, 2, figsize=[12, 4])

# Made up model
w = 5
b = -1
f = w * x + b

# Plot 0 shows model against points
axes[0].scatter(x, y, color="k")
axes[0].plot(x, f)
axes[0].grid()
axes[0].set_xlabel("$x$ (mass)")
axes[0].set_ylabel("$y$ (extension)")
axes[0].set_title("Extension vs. mass for a spring")
axes[0].legend(["targets", "$f$"])

# Plot 1 makes predictions of training points explicit
axes[1].scatter(x, y, color="k")
axes[1].scatter(x, f)
axes[1].plot(x, f)
axes[1].grid()
axes[1].set_xlabel("$x$ (mass)")
axes[1].set_ylabel("$y$ (extension)")
axes[1].set_title("Extension vs. mass for a spring")
for i in range(len(x)):
    axes[1].plot([x[i], x[i]], [y[i], f[i]], color="r")
axes[1].legend(["targets", "predictions", "$f$", "errors"])

plt.show()
```



We can see from the left plot that this clearly isn't the best model. In the right model we are explicitly showing training predictions as the blue points along the line. Ideally, we want the vertical distance between these and the targets (black points) to be as small as possible. These distances are depicted using red lines.

2.2 Minimising squared error

To achieve this, we can devise a loss function whose value is **low** when the distances between predictions and targets is low. Specifically, we will write out the total squared distance between predictions $\hat{y}^{(1)}, \hat{y}^{(2)}, \dots, \hat{y}^{(N)}$ and their respective targets $y^{(1)}, y^{(2)}, \dots, y^{(N)}$, using squared distances because they give a smooth loss and penalise large errors more heavily. Minimising squared error is generally different from minimising absolute error.

$$L_{SE} = \sum_n (y^{(n)} - \hat{y}^{(n)})^2$$

Recalling that $\hat{y} = f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ we can see that this is a function of the training data and the model parameters

$$L_{SE}(\mathbf{w}, \mathcal{D}_{train}) = \sum_n (y^{(n)} - \mathbf{w}^\top \mathbf{x}^{(n)})^2$$

> I tend to write loss functions without explicitly writing what they are a function of. This is likely because I'm not a mathematician nor a physicist. Note that I also write the model as $f(\mathbf{x})$ when it should technically be $f(\mathbf{x}, \mathbf{w})$

To learn our model, we want to solve $\underset{\mathbf{w}}{\text{minimise}}\ L_{SE}$. It is easiest to do this by converting the loss function into a vector norm and then using vector calculus, because this works for both simple linear regression and multiple linear regression (see later).

First let's write the targets out in one vector and consider the dataset in matrix form:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(N)} \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \\ \vdots \\ \mathbf{x}^{(N)\top} \end{bmatrix}$$

Now consider the matrix $\mathbf{y} - \mathbf{X}\mathbf{w}$, bearing in mind that the order of the two vectors in a dot product doesn't matter:

$$\mathbf{y} - \mathbf{X}\mathbf{w} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \\ \vdots \\ y^{(N)} \end{bmatrix} - \begin{bmatrix} \mathbf{x}^{(1)\top}\mathbf{w} \\ \mathbf{x}^{(2)\top}\mathbf{w} \\ \mathbf{x}^{(3)\top}\mathbf{w} \\ \vdots \\ \mathbf{x}^{(N)\top}\mathbf{w} \end{bmatrix} = \begin{bmatrix} y^{(1)} - \mathbf{w}^\top\mathbf{x}^{(1)} \\ y^{(2)} - \mathbf{w}^\top\mathbf{x}^{(2)} \\ y^{(3)} - \mathbf{w}^\top\mathbf{x}^{(3)} \\ \vdots \\ y^{(N)} - \mathbf{w}^\top\mathbf{x}^{(N)} \end{bmatrix}$$

If we take the norm of $\mathbf{y} - \mathbf{X}\mathbf{w}$ and square it, we can see that this is the same as L_{SE}.

$$\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w}) = \sum_n (y^{(n)} - \mathbf{w}^\top\mathbf{x}^{(n)})^2 = L_{SE}$$

If we assume fixed training data, L_{SE} is only a function of $\mathbf{w}$. It is a special kind of function known as a **convex** function. For a differentiable convex function, any point with zero gradient is a global minimum. The minimum need not be unique. This means that the optimal model parameters are those for which $\nabla_{\mathbf{w}} L_{SE} = 0$.

Using Equation 84 in the matrix cookbook (I don't expect you to do any vector calculus by hand) we can write

$$L_{SE} = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})$$

$$\nabla_{\mathbf{w}} L_{SE} = -2\mathbf{X}^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) = 0$$

If $\mathbf{X}$ has linearly independent columns, then $\mathbf{X}^\top\mathbf{X}$ is invertible and this can be rearranged to give the unique optimal parameters $\mathbf{w}^*$ as

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

> Recall that the gradient of some scalar field f (i.e. a function with a scalar output and a vector input) wrt. to its input $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_D \end{bmatrix}^\top$ can be denoted as $\nabla_{\mathbf{x}} f$ where $\nabla_{\mathbf{x}} f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \dots & \frac{\partial f}{\partial x_D} \end{bmatrix}^\top$

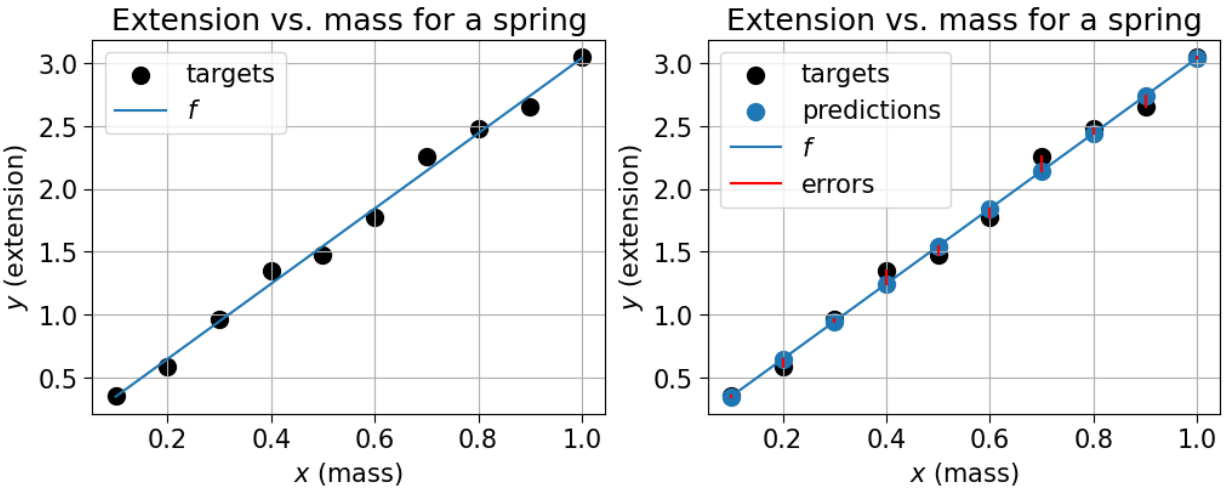
Let's get these for our spring model above, and repeat the previous plots.

```
# Explicitly write the data as a matrix
X = x.reshape(-1, 1)

# Prepend a column of ones to this matrix
X = np.hstack([np.ones_like(X), X])

# Compute w* using the above equation
w_star = np.linalg.inv(X.T @ X) @ X.T @ y

print(f"w_star is {w_star}")
print(f"The optimal b is {w_star[0]:.2f}")
print(f"The optimal w is {w_star[1]:.2f}")
```

```
w_star is [0.04860132 2.99309963]
The optimal b is 0.05
The optimal w is 2.99
```

```
# Do two plots side by side
fig, axes = plt.subplots(1,2,figsize=[12,4])

# Made up model
b, w = w_star
f = w*x+b

# Plot 0 shows model against points
axes[0].scatter(x,y,color='k')
axes[0].plot(x,f)
axes[0].grid()
axes[0].set_xlabel('$x$ (mass)')
axes[0].set_ylabel('$y$ (extension)')
axes[0].set_title('Extension vs. mass for a spring')
axes[0].legend(['targets','$f$'])
```

```
# Plot 1 makes predictions of training points explicit
axes[1].scatter(x,y,color='k')
axes[1].scatter(x,f)
axes[1].plot(x,f)
axes[1].grid()
axes[1].set_xlabel('$x$ (mass)')
axes[1].set_ylabel('$y$ (extension)')
axes[1].set_title('Extension vs. mass for a spring')
for i in range(len(x)):
    axes[1].plot([x[i],x[i]],[y[i],f[i]],color='r')
axes[1].legend(['targets','predictions','$f$','errors'])

plt.show()
```



This is clearly a much better fit.

Most of the time on this course, we will use sklearn to create and fit machine learning models. These abstract parameter learning away in a `.fit` method, and computing predictions in a `.predict` method. They also deal with weights and biases, so we don't need to modify our input data by hand.

```
# sklearn expects a dataset matrix input.
# We don't have to prepend a column of ones as sklearn takes care of
 ↪  the bias
X = x.reshape(-1, 1)

model = LinearRegression()

# fit the model to training data
model.fit(X, y)

# Print weights
print(f"The optimal b is {model.intercept_:.2f}")
print(f"The optimal w is {model.coef_[0]:.2f}")

# Apply the model to some new input
x_new = np.array([0.444]).reshape(-1, 1)
print(f"Model prediction for x={x_new} is {model.predict(x_new)}")
```

```
The optimal b is 0.05
The optimal w is 2.99
Model prediction for x=[[0.444]] is [1.37753755]
```

2.3 Multiple linear regression

Most of what we did for simple linear regression carries over to multiple linear regression (they aren't really any different).

Now we have $\mathbf{x} \in \mathbb{R}^D$ and our linear model is

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} = \begin{bmatrix} b & w_1 & w_2 & \dots & w_D \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} = b + \sum_{d=1}^{D} w_d x_d$$

We can again write the loss as

$$L_{SE} = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})$$

which has the same optimal parameters

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

As we did in the lecture, we're going to consider multiple regression with the iris dataset. Specifically, we will use petal width as the targets and the other three measurements as features for our data points.

```
# Load iris dataset as pandas dataframe
df, _ = load_iris(as_frame=True, return_X_y=True)
print(df)

# Extract dataset matrix X and targets y from dataframe
X = df[["sepal length (cm)", "sepal width (cm)", "petal length
 ↪  (cm)"]].values
y = df["petal width (cm)"].values
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width
      ↪  (cm)
0                  5.1               3.5                1.4
 ↪  0.2
1                  4.9               3.0                1.4
 ↪  0.2
2                  4.7               3.2                1.3
 ↪  0.2
3                  4.6               3.1                1.5
 ↪  0.2
4                  5.0               3.6                1.4
 ↪  0.2
..                 ...               ...                ...
 ↪  ...
145                6.7               3.0                5.2
 ↪  2.3
146                6.3               2.5                5.0
 ↪  1.9
147                6.5               3.0                5.2
 ↪  2.0
148                6.2               3.4                5.4
 ↪  2.3
```

```
149                5.9               3.0                5.1
 ↪  1.8

[150 rows x 4 columns]
```

The whole point of machine learning is to get models that *generalise* to new data. This means we should evaluate models on data that wasn't used to train the model. To achieve this we can split our dataset into a **training set** $\mathfrak{D}_{train}$ and a **test set** $\mathfrak{D}_{test}$. We use the training set to fit the model, and the test set to measure how well it generalises.

Sklearn provides a function for doing this split. Let's go with 80/20 train/test:

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

print(f"X_train has shape {X_train.shape}")
print(f"y_train has shape {y_train.shape}")
print(f"X_test has shape {X_test.shape}")
print(f"y_test has shape {y_test.shape}")
```

```
X_train has shape (120, 3)
y_train has shape (120,)
X_test has shape (30, 3)
y_test has shape (30,)
```

It is good practice to standardise our data. Although as our test set is meant to represent new data that we don't have access to, we can't use it to compute the statistics for standardisation. We must use the training set for this purpose, and then use those statistics to standardise the test set.

> We could also choose to scale the targets. This is fairly standard for neural network and Gaussian process models (which you will see later in the course) but less so for linear models. We will not scale the targets here.

```
# Create scaler object for X and y.
# This is inefficient as I could have kept X and y together, split, then
 ↪  scaled
scaler_X = StandardScaler()
scaler_y = StandardScaler()

# Fit the scaler to the training data and transform it in one go
X_train = scaler_X.fit_transform(X_train)

# Apply the fitted scaler to the test set
X_test = scaler_X.transform(X_test)
```

We can straightforwardly fit the model to the training set using sklearn:

```
model = LinearRegression().fit(X_train, y_train)

# Print weights
print(f"The optimal weights are {model.coef_}")
print(f"The optimal b is {model.intercept_}")
```

```
The optimal weights are [-0.17160163  0.07987933  0.89118069]
The optimal b is 1.1866666666666679
```

Because we standardised our data, each weight describes the change in the prediction for a one-standard-deviation increase in that feature, holding the other features fixed. Their magnitudes can help compare effects, but are not a reliable measure of feature importance when features are correlated.

2.4 Evaluating regression models

It's a lot harder to visualise models now we have 3D inputs (and above 4D is close to impossible) so we will instead summarise their performance using scores.

The most common scores for evaluating regression models are mean squared error (MSE) and the coefficient of determination (R^2).

$$L_{MSE} = \frac{1}{N}\sum_n (y^{(n)} - \hat{y}^{(n)})^2$$

L_{MSE} is just the squared error loss divided by the number of points. It can be evaluated on the training set and/or the test set. We want it to be low.

$$R^2 = 1 - \frac{\sum_n (y^{(n)} - \hat{y}^{(n)})^2}{\sum_n (y^{(n)} - \bar{y})^2}$$

R^2 is the proportional reduction in squared error when you use the model's predictions instead of the target mean $\bar{y}$ for the data being evaluated. It is zero for that constant baseline and can be negative if the model performs worse. We want it to be high. It is what we will use most of the time, and can be maximally 1.

Unlike MSE, it isn't sensitive to the scaling applied to your data.

```
print(f"Train MSE is
 ↪  {mean_squared_error(y_train,model.predict(X_train)):.3f}")
print(f"Test MSE is
 ↪  {mean_squared_error(y_test,model.predict(X_test)):.3f}")

print(f"Train R^2 is {r2_score(y_train,model.predict(X_train)):.3f}")
print(f"Test R^2 is {r2_score(y_test,model.predict(X_test)):.3f}")
```

```
Train MSE is 0.033
Test MSE is 0.049
Train R^2 is 0.938
Test R^2 is 0.933
```

This model has a high R^2 on the test set, so can generalise to new data (which is want we want!).

Note that R^2 is the default score for regression models in sklearn. This means we can compute it directly using the `.score` method as follows:

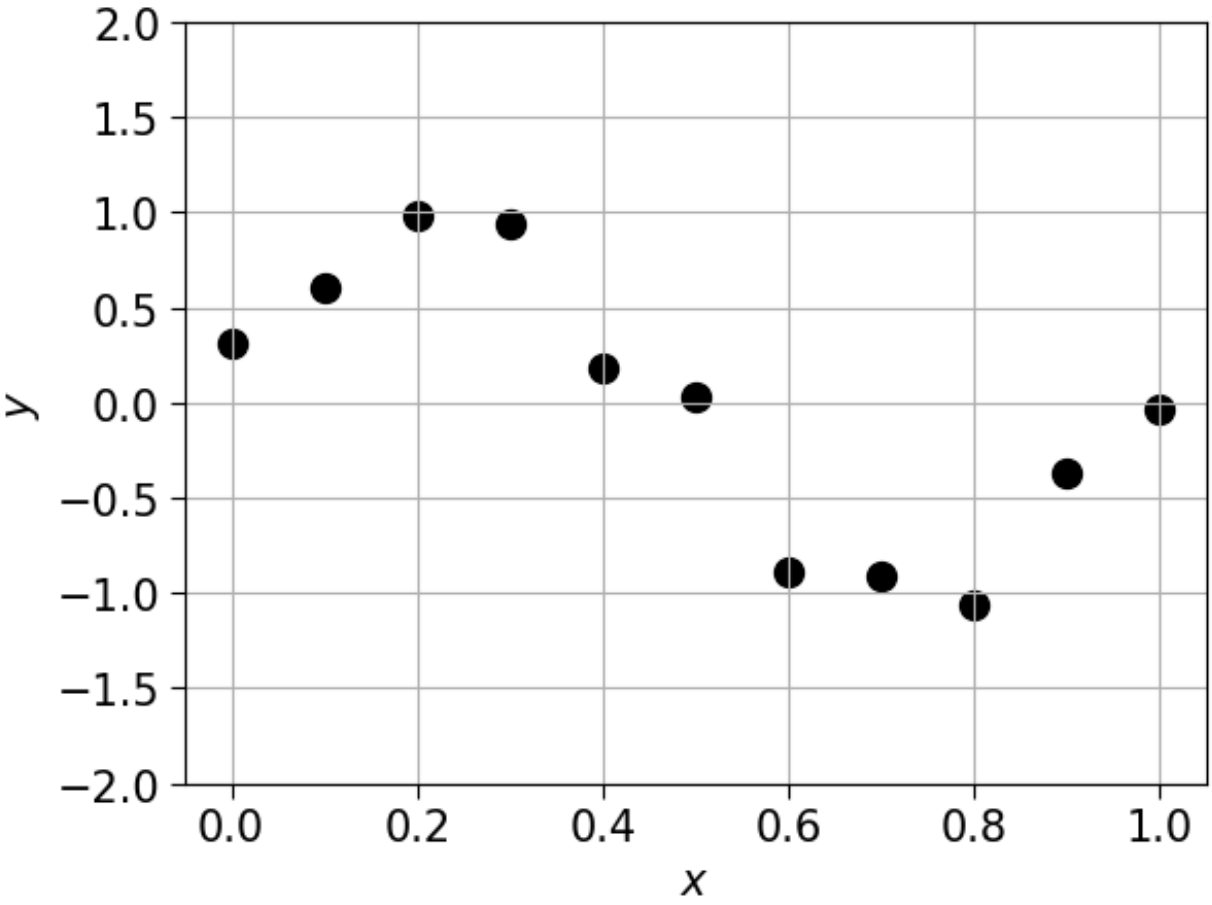
```
print(f"Train R^2 is {model.score(X_train, y_train):.3f}")
print(f"Test R^2 is {model.score(X_test, y_test):.3f}")
```

```
Train R^2 is 0.938
Test R^2 is 0.933
```

2.5 Polynomial regression

Let's consider a training set $\mathfrak{D}_{train} = \{(x^{(n)}, y^{(n)})\}_{n=1}^{10}$ where fitting a line won't work.

```
# Random seed to keep everything the same each time
np.random.seed(14)

# Synthetic data
X = np.linspace(0, 1, 11)
y = np.sin(2 * np.pi * X) + np.random.normal(0, 0.2, size=len(X))

# Plot y vs x as a scatter plot
fig, ax = plt.subplots()
ax.scatter(X, y, color="k")
ax.grid()
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")

# Setting the ylimit so high function values don't distort our plots
ax.set_ylim(-2, 2)

plt.show()
```



Let's instead fit a polynomial of degree M:

$$\hat{y} = f(x) = b + \sum_{m=1}^{M} w_m x^m$$

If we define $\phi(\mathbf{x}) = \begin{bmatrix} 1 & x & x^2 & \dots & x^M \end{bmatrix}^\top$ and write $\mathbf{w} = \begin{bmatrix} b & w_1 & w_2 & \dots & w^M \end{bmatrix}^\top$ then we have $f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x})$.

The model is still linear as its output is a weighted sum of the (new) inputs.

Our objective is once again $\min_{\mathbf{w}}$ise L_{SE}, and we can use most of the same maths as before to do it. If we define

$$\mathbf{\Phi} = \begin{bmatrix} \phi(\mathbf{x}^{(1)})^\top \\ \phi(\mathbf{x}^{(2)})^\top \\ \phi(\mathbf{x}^{(3)})^\top \\ \vdots \\ \phi(\mathbf{x}^{(N)})^\top) \end{bmatrix}$$

then we get

$$L_{SE} = \sum_n (y^{(n)} - \hat{y}^{(n)})^2 = \|\mathbf{y} - \mathbf{\Phi}\mathbf{w}\|^2$$

with solution

$$\mathbf{w}^* = (\mathbf{\Phi}^\top \mathbf{\Phi})^{-1} \mathbf{\Phi}^\top \mathbf{y}$$

.

Let's first look at a model for $M = 3$. Fortunately, sklearn provides an easy way to produce polynomial features through `PolynomialFeatures`.

(See the documentation here.)

```
# Write X as a matrix
X = X.reshape(-1, 1)

# Create an object that lets us transform, and then apply the
 ↪  transformation to our data for M=3
phi = PolynomialFeatures(3, include_bias=False)
P = phi.fit_transform(X)

# Have a look at P to make sure it makes sense.
# We do not have a column of 1s as this will be included in the model
 ↪  bias.
print(P)
```

```
[[0.    0.    0.   ]
 [0.1   0.01  0.001]
 [0.2   0.04  0.008]
 [0.3   0.09  0.027]
 [0.4   0.16  0.064]
 [0.5   0.25  0.125]
 [0.6   0.36  0.216]
 [0.7   0.49  0.343]
 [0.8   0.64  0.512]
 [0.9   0.81  0.729]
 [1.    1.    1.   ]]
```

We can fit the model just by using `LinearRegression` with these features. Note that sklearn makes no assumptions about the input data at all. It just asks for:

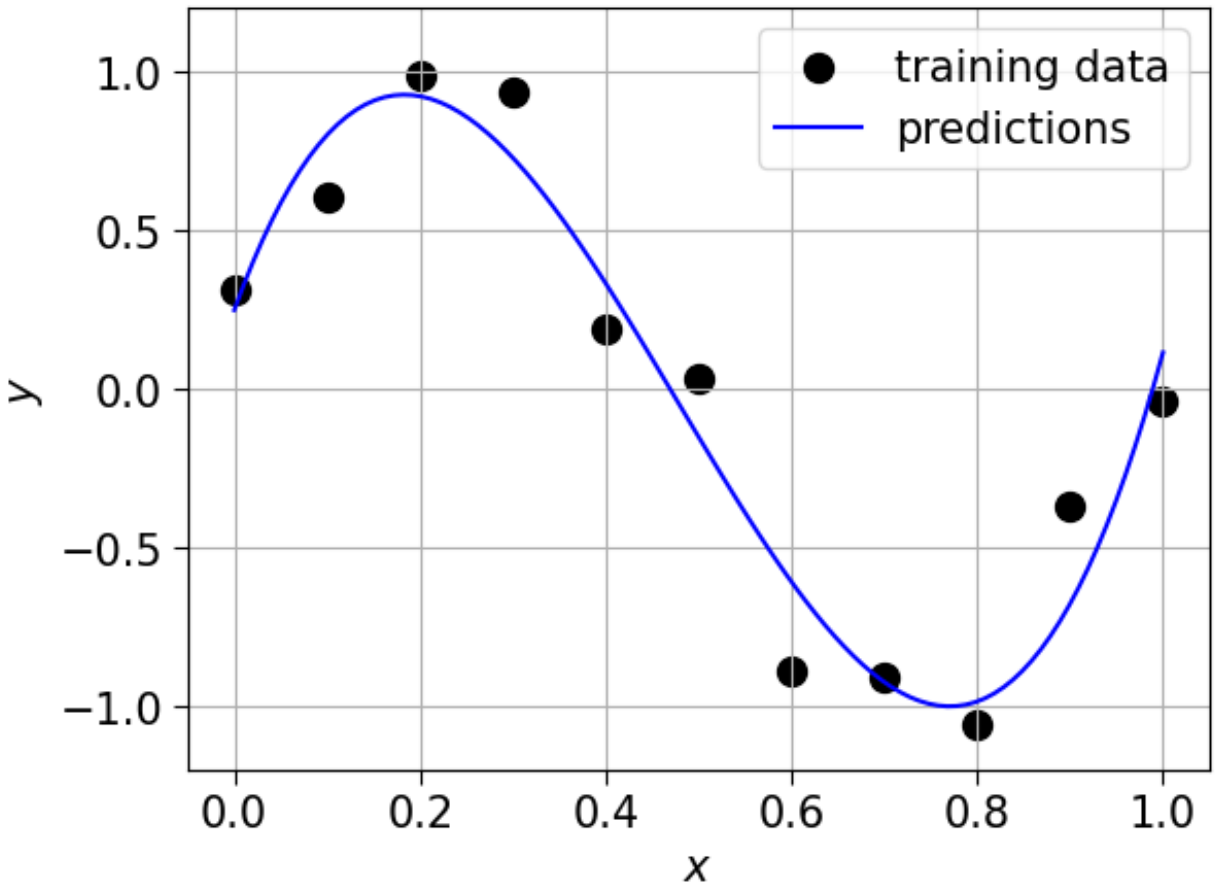
> X{array-like, sparse matrix} of shape (n_samples, n_features)

This can be the *raw* training data, or feature transformations thereof.

```
model = LinearRegression().fit(P, y)
```

Let's plot our model to see how it does.

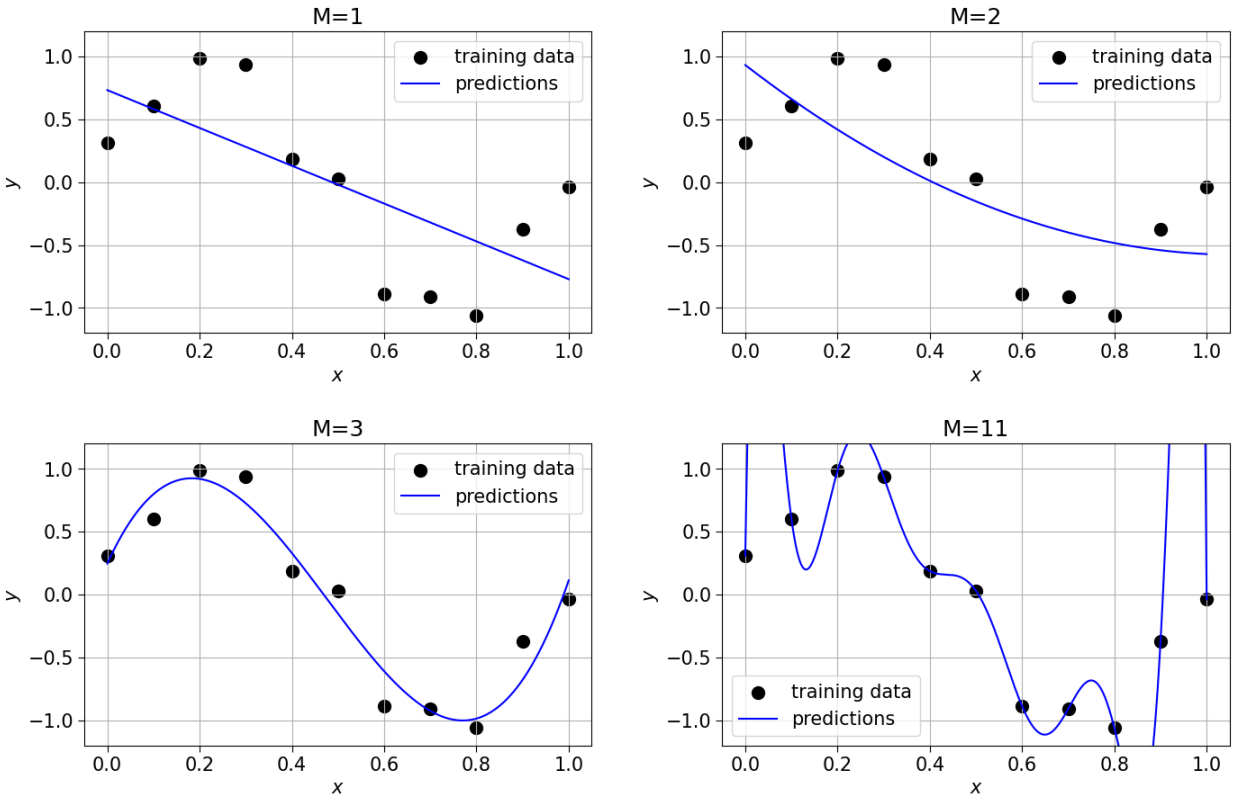
```
# Crate some dummy points for plotting and apply our model to them
X_dummy = np.linspace(0, 1, 100).reshape(-1, 1)
fx_dummy = model.predict(phi.fit_transform(X_dummy))

# Plot y vs x as a scatter plot
fig, ax = plt.subplots()
ax.scatter(X, y, color="k")

# Plot dummy points and predictions to show function
ax.plot(X_dummy, fx_dummy, color="b")

ax.grid()
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
plt.legend(["training data","predictions"])

# Setting the ylimit so high function values don't distort our plots
ax.set_ylim(-1.2, 1.2)

plt.show()
```



Let's see what the model would look like for other values of M. We'll produce a plot with multiple figures for $M = \{1, 2, 3, 11\}$.

```
# Values of M we are going to try out

# Create a list to store different models
models = []

# Create a list that contains the M values to try out
M_values = [1, 2, 3, 11]


# Plot for each model
fig, axes = plt.subplots(2, 2, figsize=[15, 10])

# Flattening the axes means we can index it like axes[0] axes[1] etc.
axes = axes.flatten()

# Padding to stop subplots clashing
fig.tight_layout(pad=4)

for i, m in enumerate(M_values):
    phi = PolynomialFeatures(m, include_bias=False)
    P = phi.fit_transform(X)
    model = LinearRegression().fit(P, y)

    # Make dummy points and fit model to them
    x_dummy = np.linspace(0, 1, 1000).reshape(-1, 1)
    fx_dummy = model.predict(phi.fit_transform(x_dummy))

    # Plot y vs x as a scatter plot
```

```
    axes[i].scatter(X, y, color="k")

    # Plot predictions as a curve
    axes[i].plot(x_dummy, fx_dummy, color="b")

    axes[i].grid()
    axes[i].set_xlabel("$x$")
    axes[i].set_ylabel("$y$")
    axes[i].legend(["training data", "predictions"])
    axes[i].set_title(f"M={M_values[i]}")
    axes[i].set_ylim(-1.2, 1.2)

plt.show()
```



The $M = 1$ and $M = 2$ models does not have the capacity to fit the function. They have *underfitted*. In contrast, the $M = 11$ model has overfit to the noisy data points.

In this particular example, the $M = 3$ model looks like the best fit; it does appear almost sinusoidal. However, it only has 4 parameters. There will exist more complicated functions where we will need more.

2.6 Linear regression with regularisation

We want our model to have sufficient capacity to represent the function we care about, however the capacity in question is rarely obvious.

What we could do is use a high capacity model to avoid underfitting, and then **regularise** this model to reduce the risk of overfitting.

This is done by adding a regularisation term to our loss function. If this penalises the L2 norm of our weights then we call the whole thing "ridge regression".

$$L_{ridge}(\mathbf{w}) = \underbrace{\|\mathbf{y} - \mathbf{\Phi}\mathbf{w}\|^2}_{SE} + \underbrace{\lambda \mathbf{w}^\top \mathbf{P} \mathbf{w}}_{regularisation}$$

The $\mathbf{w}$ that minimises this is

$$\mathbf{w} = (\mathbf{\Phi}^\top \mathbf{\Phi} + \lambda \mathbf{P})^{-1} \mathbf{\Phi}^\top \mathbf{y}$$

Remember that you can view $\mathbf{X}$ and $\mathbf{\Phi}$ as being interchangeable in these equations

Recall from the lecture that we do not regularise the bias parameter. With the bias as the first element of $\mathbf{w}$, use $\mathbf{P} = \text{diag}(0, 1, \ldots, 1)$. This leaves the bias unpenalised, as sklearn does by default.

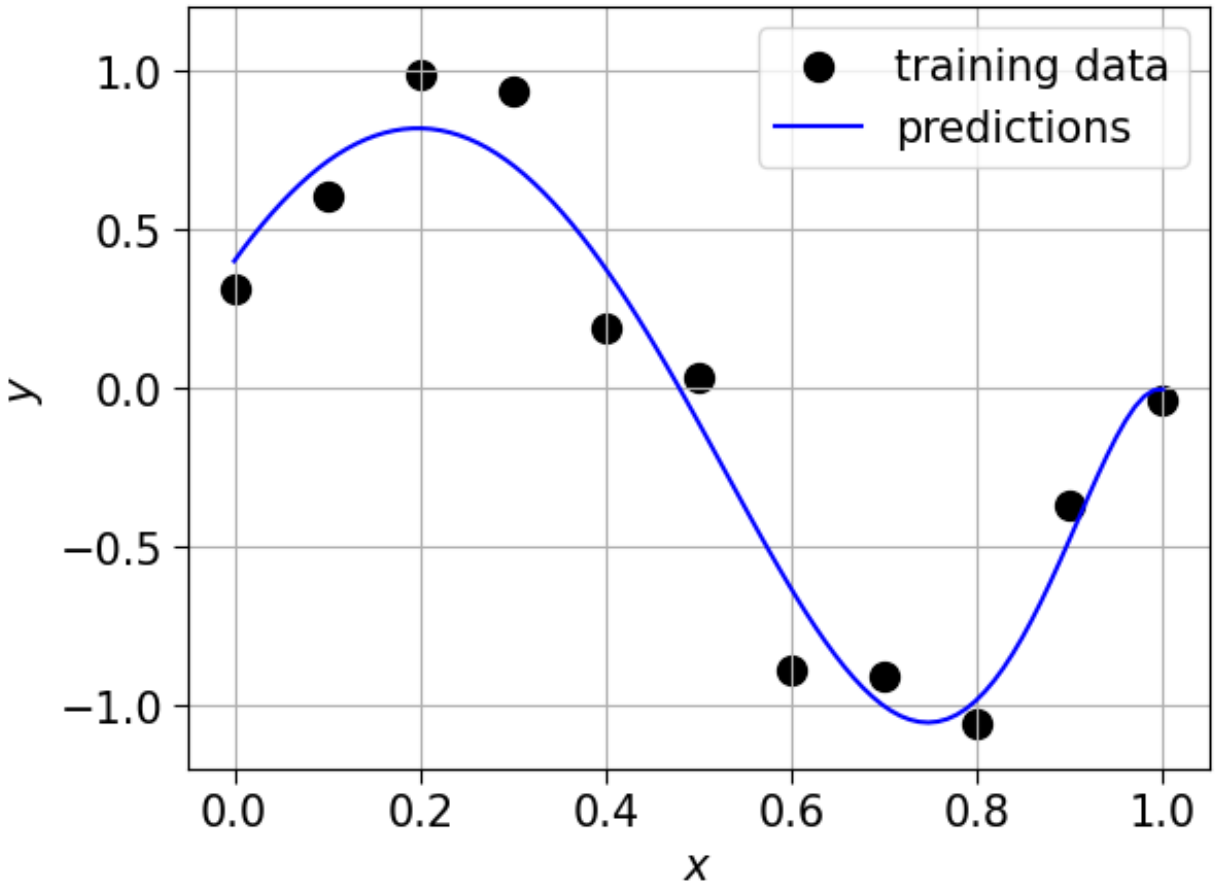
Let's perform ridge regression to regularise a model for $M = 11$. We can see from above that without regularisation it overfits badly.

We can again use sklearn to do this with its `Ridge` class (documentation here). The main difference here is that in sklearn the regularisation coefficent is refered to as `alpha` instead of λ.

```
# M=11 model
phi = PolynomialFeatures(11, include_bias=False)
P   = phi.fit_transform(X)

# Fit model
model = Ridge(alpha=0.001)
model.fit(P, y)
```

```
Ridge(alpha=0.001)
```

```
# Create some dummy points for plotting and apply our model to them
X_dummy = np.linspace(0, 1, 100).reshape(-1, 1)
fx_dummy = model.predict(phi.fit_transform(X_dummy))

# Plot y vs x as a scatter plot
fig, ax = plt.subplots()
ax.scatter(X, y, color="k")

# Plot dummy points and predictions to show function
ax.plot(X_dummy, fx_dummy, color="b")

ax.grid()
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
plt.legend(["training data", "predictions"])

# Setting the ylimit so high function values don't distort our plots
ax.set_ylim(-1.2, 1.2)

plt.show()
```



We can see that our regularised model closer resembles the original function.

Sklearn also provides functionality for `Lasso` regression (documentation here), which you will use in the lab.

You will also explore using a **validation set** for hyperparameter tuning.

3 Linear models for classification

```
# RUN THIS CELL FIRST

# Imports
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# sklearn
from sklearn.linear_model import LogisticRegression
from sklearn.inspection import DecisionBoundaryDisplay

# This makes matplotlib output nice figures without much tweaking
plt.rcParams.update(
    {
        "lines.markersize": 10,   # Big points
        "font.size": 15,   # Larger font
        "xtick.major.size": 5.0,   # Bigger xticks
        "ytick.major.size": 5.0,   # Bigger yticks
    }
)
```

Given a training set of N data point-target pairs $\{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$ where $\mathbf{x} \in \mathbb{R}^D$ and $y \in \mathbb{Z}^+_{<K} = \{0, 1, \dots, K-1\}$ the goal in classification is to obtain a model that can predict the correct class (i.e. correct value of y) for each training point. Although, what we actually want is for the model to generalise to unseen test data.

> **Note: I tend to use the terms linear classifier and linear model interchangeably for classification but if we're being rigorous, a linear classifier should refer to a linear model coupled with a choice of threshold function.**

3.1 Linear models for binary classification

In a binary classification problem, there are two possible labels. These are typically either denoted as $y \in \{0, 1\}$ or $y \in \{-1, 1\}$ depending on what is most mathematically convenient.

A linear model for binary classification looks very similar to a linear model for regression:

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$

> **When we write the maths for classification, we DON'T tend to prepend our data points with a 1 as we did for regression. The bias parameter is typically made explicit.**

$\mathbf{w} \in \mathbb{R}^D$ is a vector of learnable weights and there is a learnable bias $b \in \mathbb{R}$.

Once we have learnt the weights, $f(\mathbf{x}) \in \mathbb{R}$ can be used to make class predictions using some threshold function such as

$$\hat{y} = \begin{cases} 1 & \text{if} f(\mathbf{x}) \geq 0 \\ 0 & \text{if} f(\mathbf{x}) < 0 \end{cases}$$

This leads to the important concept of a decision boundary. For the threshold function above, $f(\mathbf{x}) = 0$ defines a hyperplane either side of which points will be put into different classes.

In 2D, this is a line. By writing $\mathbf{w} = \begin{bmatrix} w_1 & w_2 \end{bmatrix}^\top$ and $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}^\top$ we can rewrite $f(\mathbf{x})$ in terms of scalar quantities:

$$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + b$$

We can then set $f(\mathbf{x}) = 0$ and rearrange to get the equation of a line which defines the decision boundary.

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$

3.2 Logistic regression

The parameters $\mathbf{w}, b$ of a linear classifier need to be learnt using training data. This boils down to defining a loss function L and then solving $\underset{\mathbf{w},b}{\text{minimise}} L$. This process is usually referred to as fitting, learning, or training.

We are going to focus on logistic regression which involves minimising the so-called **Log Loss**.

The motivation is as follows: we don't just want $f(\mathbf{x})$ to just tell us how a certain point should be classified, we would prefer that it captured some notion of uncertainty. Probabilities are ideal quantities for this purpose. However, $f(\mathbf{x})$ cannot represent probabilities as it is unbounded; it can take values greater than 1 and less than 0 whereas probabilities must be between 0 and 1.

Instead, we allow $f(\mathbf{x})$ to represent some unbounded quantity that is *related* to a probability. Specifically, we define $f(\mathbf{x})$ as the log-odds that a point $\mathbf{x}$ is in class 1, which relates to $p(y = 1|\mathbf{x})$.

$$f(\mathbf{x}) = \log \frac{p(y = 1|\mathbf{x})}{1 - p(y = 1|\mathbf{x})}$$

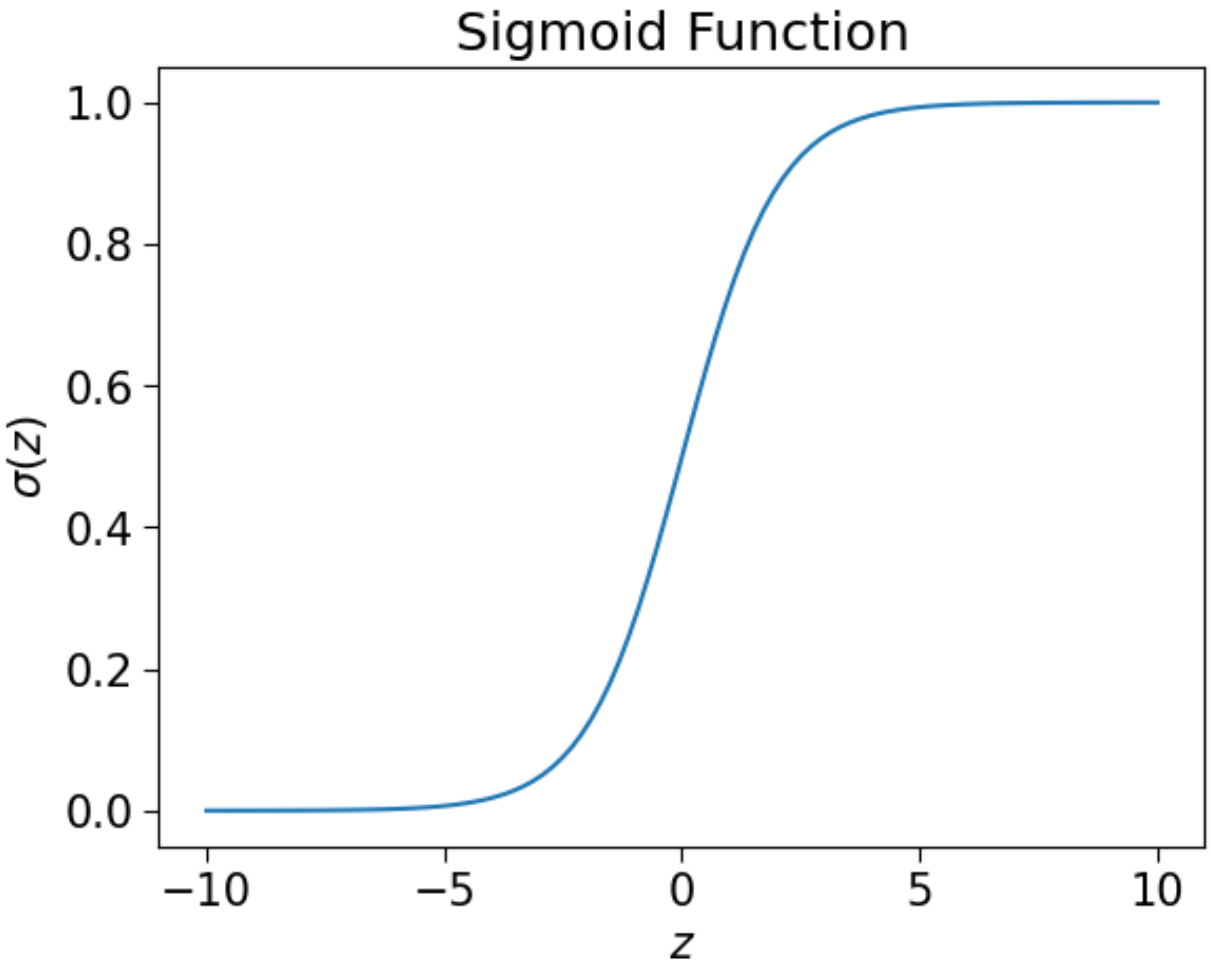
> **Note: If I don't specify the base for a logarithm, then I am referring to the natural logarithm.**

We can rearrange this expression to get $p(y = 1|\mathbf{x})$ in terms of log-odds:

$$p(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-f(\mathbf{x})}} = \sigma(f(\mathbf{x})) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

Here we have defined the sigmoid function as $\sigma(z) = \frac{1}{1+e^{-z}}$ which squashes unbounded quantities to be between 0 and 1.

```
z = np.linspace(-10, 10, 100)
sigma_z = 1 / (1 + np.exp(-z))

fig, ax = plt.subplots()
ax.plot(z, sigma_z)
ax.set_title("Sigmoid Function")
ax.set_ylabel(r"$\sigma (z)$")
ax.set_xlabel("$z$")
plt.show()
```



To make a prediction, it is sensible to classify a point $\mathbf{x}$ as class 1 if $p(y = 1|\mathbf{x}) > p(y = 0|\mathbf{x})$ and as class 0 otherwise.

As there are two classes, $p(y = 1|\mathbf{x}) + p(y = 0|\mathbf{x}) = 1$. This means that $p(y = 1|\mathbf{x}) > p(y = 0|\mathbf{x})$ occurs when $p(y = 1|\mathbf{x}) > 0.5$. Our threshold function should therefore be

$$\hat{y} = \begin{cases} 1 & \text{if } p(y = 1|\mathbf{x}) \geq 0.5 \\ 0 & \text{if } p(y = 1|\mathbf{x}) < 0.5 \end{cases}$$

Noting that $p(y = 1|\mathbf{x}) = 0.5$ when $f(\mathbf{x}) = 0$ lets us rewrite this in terms of the log-odds.

$$\hat{y} = \begin{cases} 1 & \text{if} f(\mathbf{x}) \geq 0 \\ 0 & \text{if} f(\mathbf{x}) < 0 \end{cases}$$

To derive the log loss itself, we start by noting that

$$p(y = 0|\mathbf{x}) = 1 - \sigma(f(\mathbf{x}))$$

We can combine this with our expression for $p(y = 1|\mathbf{x})$ to succinctly write

$$p(y|\mathbf{x}) = \sigma(f(\mathbf{x}))^y (1 - \sigma(f(\mathbf{x})))^{1-y}$$

We can now consider the likelihood of our training data $\{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^N$ according to the model. That is, the probability of the labels being $y^{(1)}, y^{(2)}, ...y^{(N)}$ given that the data points are $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(N)}$.

If we assume that data points are independent of each other, then we can write the likelihood as:

$$\prod_n p(y^{(n)}|\mathbf{x}^{(n)}) = \prod_n \sigma(f(\mathbf{x}^{(n)}))^{y^{(n)}} (1 - \sigma(f(\mathbf{x}^{(n)})))^{1-y^{(n)}}$$

We want this quantity to be high. This is equivalent to making the **negative log** of this quantity low. We can further divide this by the number of data points as it has no effect on the optimisation problem. This gives us the negative log-likelihood, or **log loss**.

$$\text{NLL}(\mathbf{w}, b) = -\frac{1}{N} \sum_n \left[y^{(n)} \log \sigma(f(\mathbf{x}^{(n)})) + (1 - y^{(n)}) \log(1 - \sigma(f(\mathbf{x}^{(n)}))) \right]$$

We can make this more succinct by writing $p^{(n)} = \sigma(f(\mathbf{x}^{(n)}))$:

$$L_{log} = -\frac{1}{N} \sum_n \left[y^{(n)} \log p^{(n)} + (1 - y^{(n)}) \log(1 - p^{(n)}) \right]$$

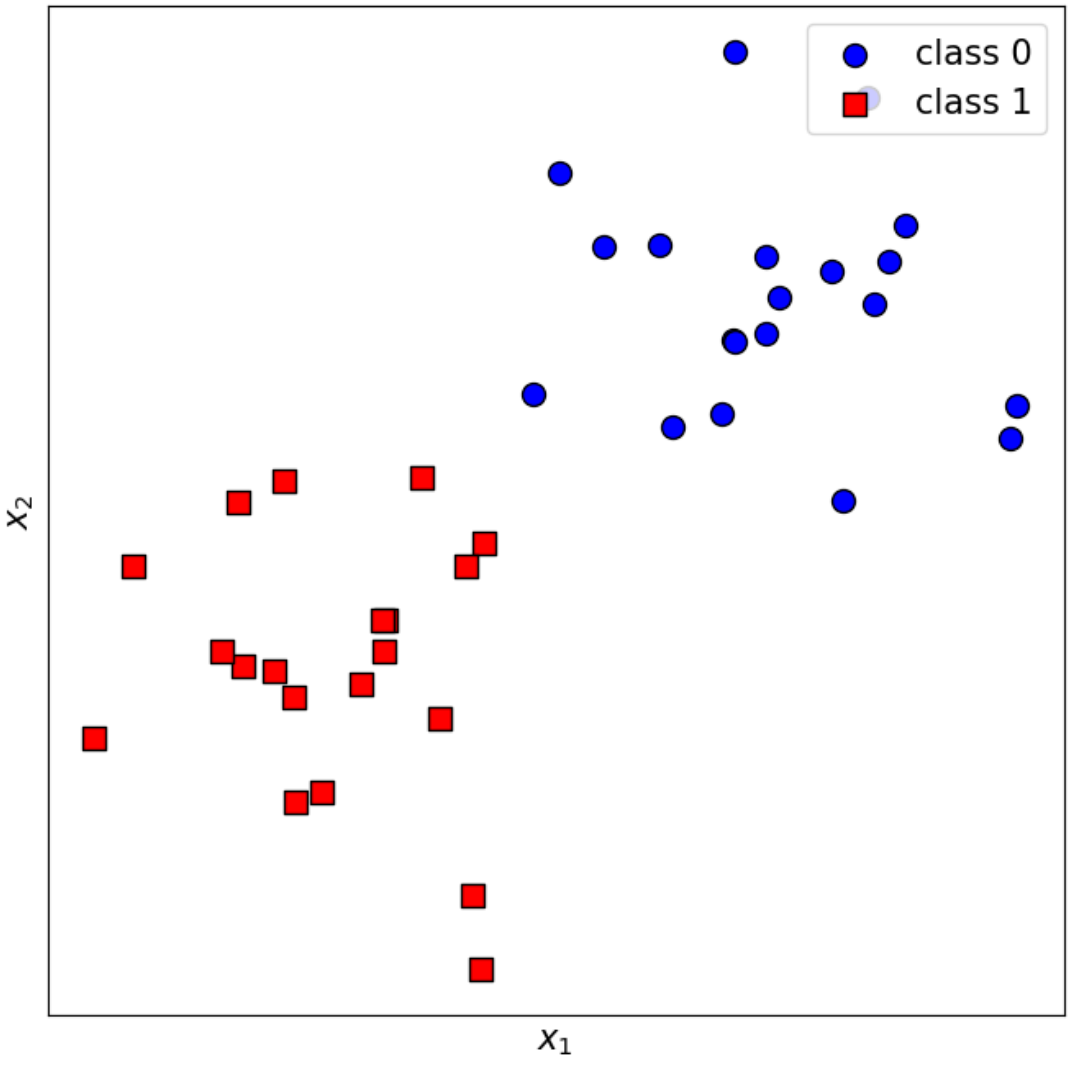
Solving $\underset{\mathbf{w},b}{\text{minimise}} L_{log}$ is equivalent to performing logistic regression.

3.3 Logistic regression example

After that mathematical pre-amble, let's look at a logistic regression model on some 2D dummy data.

First, let's create and plot the data:

```
np.random.seed(42)

x0 = np.vstack(
    (np.random.normal(1, 0.6, size=20), np.random.normal(1, 0.6,
     ↪ size=20))
).T

x1 = np.vstack(
    (np.random.normal(-1, 0.6, size=20), np.random.normal(-1, 0.6,
     ↪ size=20))
).T

# Make plot
fig, ax = plt.subplots(figsize=[8, 8])
ax.scatter(x0[:, 0], x0[:, 1], color="b", edgecolors="black")
ax.scatter(x1[:, 0], x1[:, 1], color="r", marker="s",
 ↪ edgecolors="black")
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1"], loc="upper right")
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```



Now we can use a LogisticRegression object from sklearn to do the grunt work. This includes L2 regularisation by default, so when we call `.fit` it is actually finding the parameters that minimise

$$L = \underbrace{L_{log}}_{classification} + \underbrace{\frac{\lambda}{2}\|\mathbf{w}\|^2}_{regularisation}$$

where, with the average log loss defined above and N equally weighted training examples, $\lambda = 1/(NC)$. Smaller values of the `C` argument therefore mean stronger regularisation.

```
X = np.vstack((x0, x1))
y = np.hstack((np.zeros(len(x0)), np.ones(len(x1))))
model = LogisticRegression(random_state=0, C=1).fit(X, y)
```

Given some dummy point, we can see the predicted class labels and probabilities for class 0 and 1:

```
x_dummy = np.array([0, 1]).reshape(1, -1)

print(model.predict(x_dummy))
print(model.predict_proba(x_dummy))
```

```
[0.]
[[0.85958037 0.14041963]]
```

We can also see its learnt parameters:

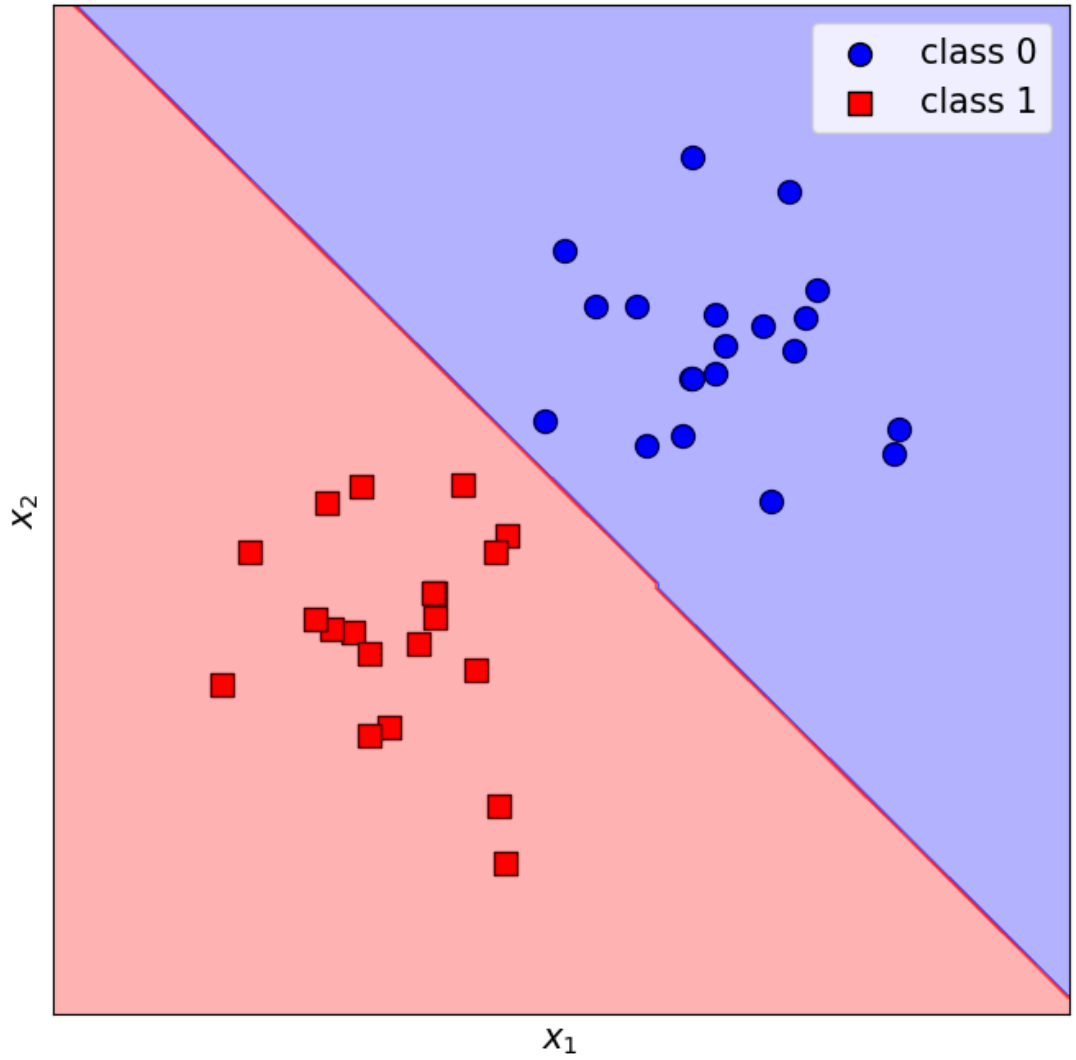
```
print(f"w:{model.coef_[0]}")
print(f"b:{model.intercept_[0]}")
```

```
w:[-1.75815726 -1.57035444]
b:-0.24145460644949604
```

We can use sklearn's DecisionBoundaryDisplay to show the decision boundary of the classifier alongside the training data.

```
# Make plot
fig, ax = plt.subplots(figsize=[8, 8])

ax.scatter(x0[:, 0], x0[:, 1], color="b", edgecolors="black")
ax.scatter(x1[:, 0], x1[:, 1], color="r", marker="s",
 ↪  edgecolors="black")
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1"], loc="upper right")
ax.set_xticks([])
ax.set_yticks([])
colours = ["b", "r"]

# Plot decision boundary
disp = DecisionBoundaryDisplay.from_estimator(
    model,
    X,
    response_method="predict",
    alpha=0.3,
    grid_resolution=200,
```

```
    cmap=ListedColormap(colours),
    ax=ax,
)

plt.show()
```



In the lab, you will consider lines that correspond to certain probabilities of the model.

3.4 Perceptrons [Optional]

A perceptron is an obsolete, but historically significant classifier. There is no reason why you should use it but it is available in sklearn.

It is a linear model $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ whose weights and biases have been learnt through the perceptron algorithm.

Assuming that the labels are given by $y \in \{-1, 1\}$, we will use the threshold function $\hat{y} = \begin{cases} 1 & \text{if } f(\mathbf{x}) \geq 0 \\ -1 & \text{if } f(\mathbf{x}) < 0 \end{cases}$

If $\hat{y}^{(n)}$ is our class prediction for some training point $\mathbf{x}^{(n)}$ then we want

$$f(\mathbf{x}^{(n)}) = \mathbf{w}^\top \mathbf{x}^{(n)} + b \quad \text{s.t.} \quad \hat{y}^{(n)} = \begin{cases} 1 & \text{if} f(\mathbf{x}^{(n)}) \geq 0 \\ -1 & \text{if} f(\mathbf{x}^{(n)}) < 0 \end{cases} \forall n$$

For the perceptron learning algorithm we iterate through shuffled training data and if some point $\mathbf{x}^{(n)}$ is misclassified using the threshold function we update

1. $\mathbf{w} \leftarrow \mathbf{w} + \alpha\, y^{(n)} \mathbf{x}^{(n)}$
2. $b \leftarrow b + \alpha\, y^{(n)}$

The perceptron algorithm is simply equivalent to minimising the following loss through stochastic gradient descent (SGD).

$$L_{hinge} = \frac{1}{N} \sum_n \max\Big(0, -y^{(n)} f(\mathbf{x}^{(n)})\Big)$$

3.5 Support Vector Machines

In the first year that this course ran, there was a whole lecture on SVMs! I've since decided to cut this out as they are rather outdated.

All you need to know for this course this year is that SVMs are linear models that are obtained by solving $\underset{\mathbf{w},b}{\text{minimise}} L_{SVM}$ where

$$L_{SVM} = \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_n \max\Big(0, 1 - y^{(n)} f(\mathbf{x}^{(n)})\Big)$$

> Note that C is a hyperparameter that controls the relative importance of the first and second terms.

This encourages the distance from the decision boundary to the nearest points from each class (the margin) to be large, so that small perturbations in data don't result in a completely different classification. A logistic regression model with sufficient regularisation also has a large margin.

SVMs are available in sklearn through the LinearSVC class.

3.6 Multinomial Logistic Regression

While there are several strategies for performing multi-class classification ($y \in \mathbb{Z}^+_{<K} = \{0, 1, \ldots, K-1\}$) involving combining binary classifiers, it is simplest to use a model that gives a vector output $f(\mathbf{x}) \in \mathbb{R}^K$ where elements correspond to classification scores for different classes.

This can be achieved by writing the linear model as

$$f(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

where $\mathbf{W} \in \mathbb{R}^{K \times D}$ and $\mathbf{b} \in \mathbb{R}^K$.

You can view this as being like K classifiers side-by-side as each row of the matrix $\mathbf{W}$ plus the corresponding element of b is used to compute the classifier score for a particular class.

In Multinomial logistic regression, we define each element of $f(\mathbf{x})$ as being the **logit** for a particular class. This is an unnormalised log-probability, and may be seen as a generalisation of the log-odds to this multiclass setting.

We relate these logits to class probabilities through a squashing function called a softmax.

$$\mathbf{p} = \begin{bmatrix} p(y=0|\mathbf{x}) \\ p(y=1|\mathbf{x}) \\ p(y=2|\mathbf{x}) \\ \vdots \\ p(y=K-1|\mathbf{x}) \end{bmatrix} = S(f(\mathbf{x}))$$

where

$$S(\mathbf{z}) = S(\begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_{K-1} \end{bmatrix}) = \begin{bmatrix} \frac{\exp z_0}{\sum_{k=0}^{K-1} \exp z_k} \\ \frac{\exp z_1}{\sum_{k=0}^{K-1} \exp z_k} \\ \vdots \\ \frac{\exp z_{K-1}}{\sum_{k=0}^{K-1} \exp z_k} \end{bmatrix}$$

In short, the softmax function S turns a vector of logits into a vector of probabilities that sum to 1.

The likelihood (again assuming independence between data points) is now given by

$$\prod_{n=1}^{N} \prod_{k=0}^{K-1} p(y^{(n)} = k|\mathbf{x}^{(n)})^{\mathbb{I}(y^{(n)}=k)}$$

$\mathbb{I}(y^{(n)} = k)$ is an indicator function that outputs 1 if $y^{(n)} = k$ and 0 otherwise.

From this, we may write the negative log-likelihood (divided by the number of data points) as

$$\text{NLL}(\mathbf{w}, b) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=0}^{K-1} \mathbb{I}(y^{(n)} = k) \log p(y^{(n)} = k|\mathbf{x}^{(n)})$$

This is a bit verbose. To succinctly write out the NLL, we will use a one-hot vector representation for each class label. This means each class label is now a vector $\mathbf{y}$ which is 1 for the element corresponding to its class, and zero elsewhere. For example, if we had 4 classes and $y^{(t)}$ was labelled as class 2 we would write $\mathbf{y}^{(t)} = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}^\top$ (recalling that we count from zero when it comes to classes). This gives us the NLL, or log loss in the succinct form of

$$L_{log} = \frac{1}{N} \sum_{n} -\mathbf{y}^{(n)\top} \log \mathbf{p}^{(n)}$$

Sklearn will automatically use multinomial logistic regression if you call `.fit` with data where there are more than 2 class labels.

4 Visualising data

```
# RUN THIS CELL FIRST

# Imports
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# sklearn
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# Better plots
plt.rcParams.update(
    {
        "lines.markersize": 10,  # Big points
        "font.size": 15,  # Larger font
        "xtick.major.size": 5.0,  # Bigger xticks
        "ytick.major.size": 5.0,  # Bigger yticks
    }
)
```

4.1 Plots

For plotting in Python there are two standard options. The first is to use matplotlib, and the second is to used seaborn which is based on matplotlib. Seaborn typically creates better-looking plots than matplotlib with less effort, but matplotlib is a lot more customisable.

I have used matplotlib for the vast majority of the plots in the lectures and labs, but you are more than welcome to use seaborn for the labs and tests if you prefer it.

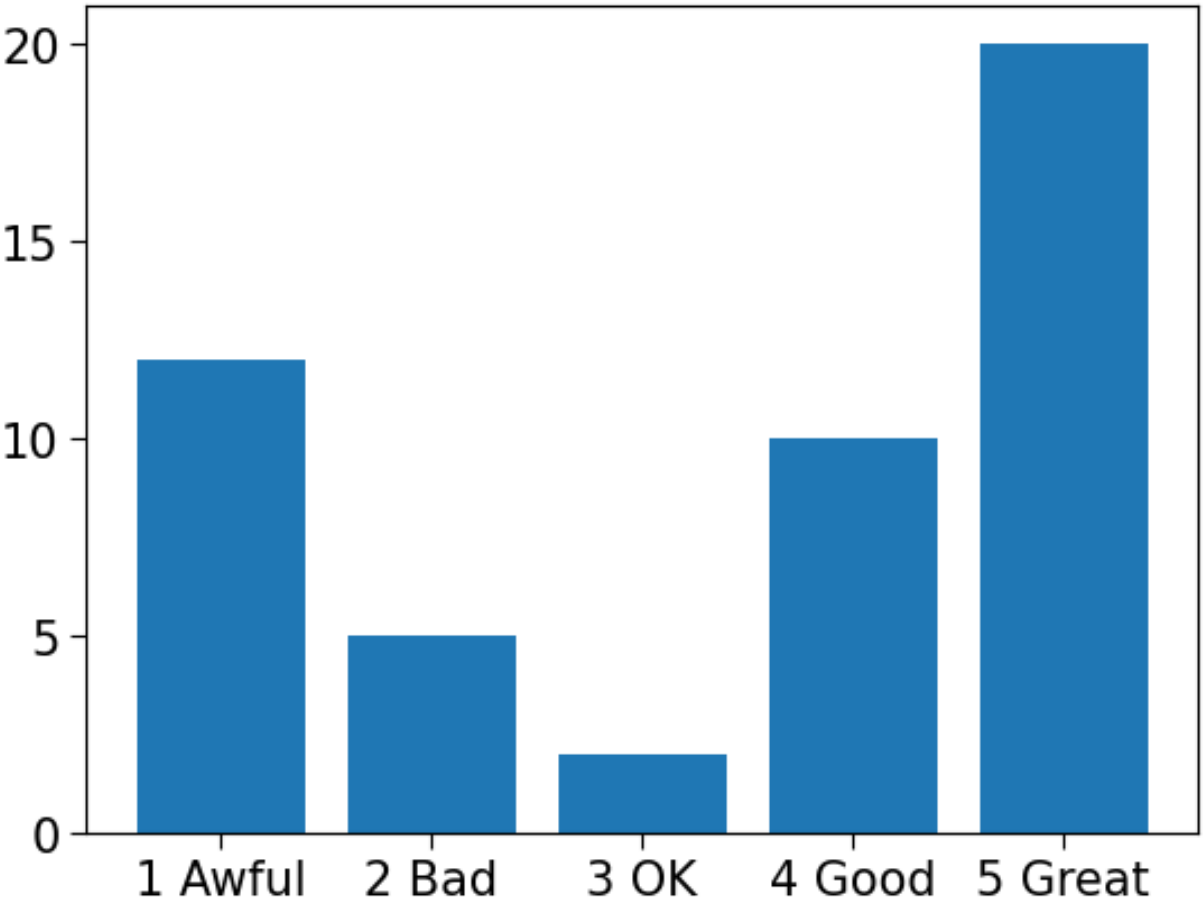
> It is also possible to plot through pandas itself if you are working with a dataframe (some examples are here) using matplotlib as a backend. This is quite neat but less flexible than using matplotlib directly.

I'll now provide code to recreate different plots from the lecture. The `fig, ax = plt.subplots()` that we use to create the figure and its axes looks a bit esoteric, but you can find a good explanation of why we're doing it here.

4.1.1 Bar plots

These are useful for visualising the distribution of a categorical variable.

```
### Bar plot of lecture ratings
fig, ax = plt.subplots()

# Synthesise data
opinions = ["1 Awful", "2 Bad", "3 OK", "4 Good", "5 Great"]
tallies  = [12, 5, 2, 10, 20]

# Plot
ax.bar(opinions, tallies)
plt.show()
```

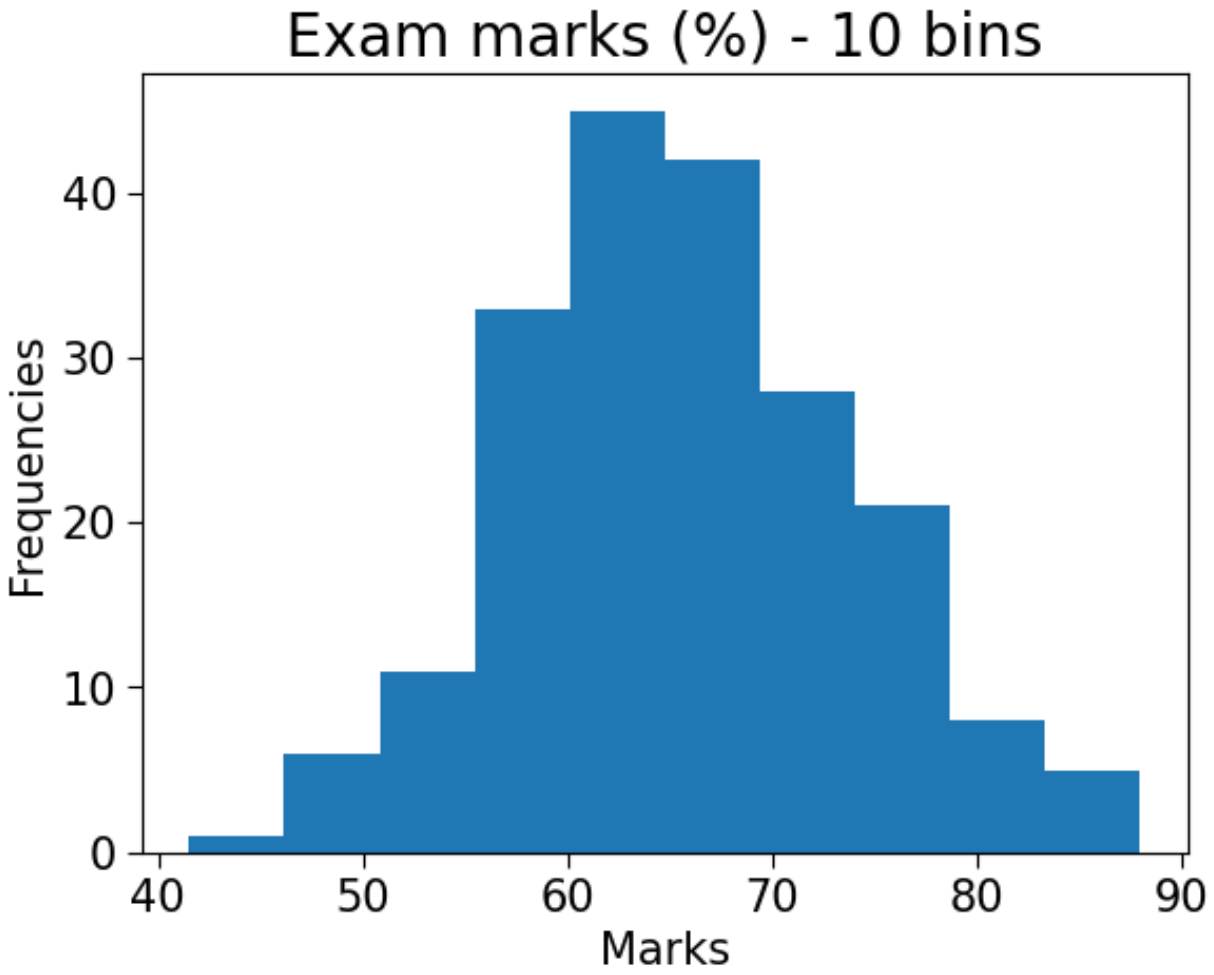


4.1.2 Histograms

These are useful for visualising the distribution of a continuous variable, although you may need to tweak the bin size.

```
### Histogram of exam results

# Package for creating skewed data
from scipy.stats import skewnorm

# The number of bins.
num_bins = 10

fig, ax = plt.subplots()
```

```
# Setting a seed so the plot looks the same each time
np.random.seed(4)

# Skewed data
x = skewnorm.rvs(1, loc=60, scale=10, size=200)

ax.hist(x, bins=num_bins)

ax.set_xlabel("Marks")
ax.set_ylabel("Frequencies")
ax.set_title(f"Exam marks (%) - {num_bins} bins", fontsize=20)

plt.show()
```

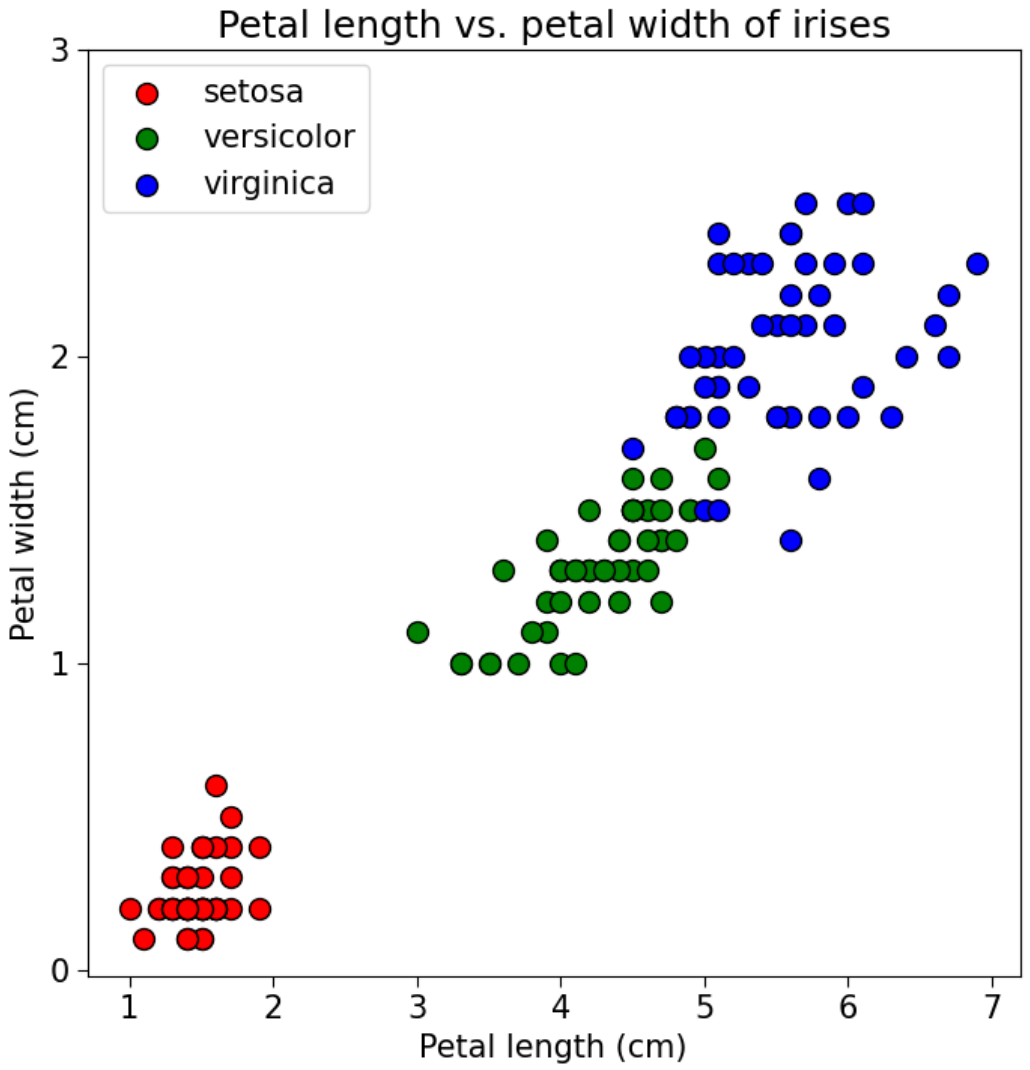


4.1.3 Scatter plots

I would argue these are the most important plot of all. They let you compare values for different features in either 2D or 3D.

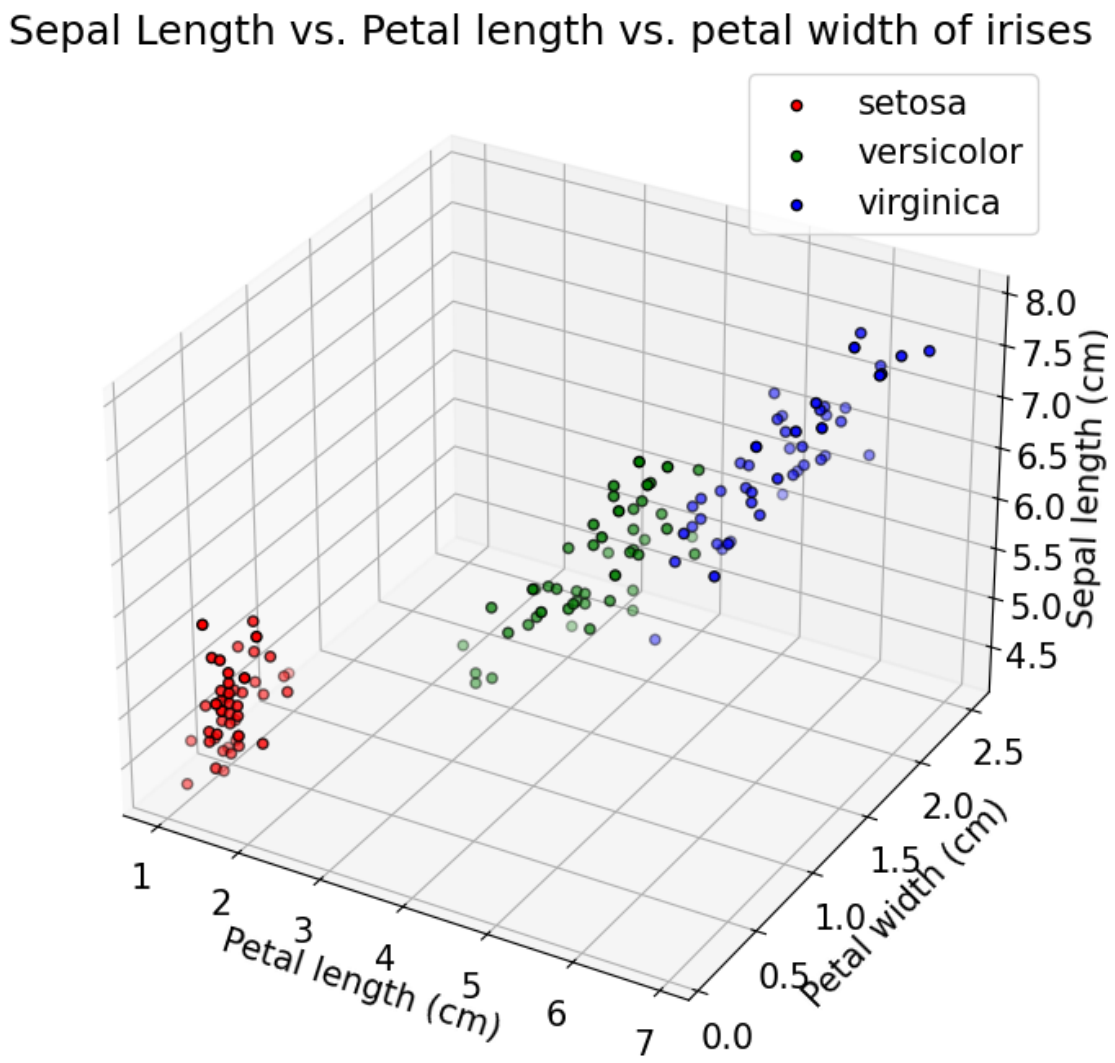
```
### Petal length vs. petal width for irises (2D)

data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
```

```
# Add a column with the species as an integer 0,1,2
# We can use different colours for different species
df.insert(4, "species", data.target)

fig, ax = plt.subplots(figsize=[8,8])
ax.scatter(
    df["petal length (cm)"][df["species"] == 0],
    df["petal width (cm)"][df["species"] == 0],
    color="r",
    edgecolor='k',
)
ax.scatter(
    df["petal length (cm)"][df["species"] == 1],
    df["petal width (cm)"][df["species"] == 1],
    color="g",
    edgecolor='k',
)
ax.scatter(
    df["petal length (cm)"][df["species"] == 2],
    df["petal width (cm)"][df["species"] == 2],
    color="b",
    edgecolor='k',
)

ax.legend(data.target_names)

# Tidy up ticks (optional)
ax.set_yticks([0, 1, 2, 3])

# Label
ax.set_title("Petal length vs. petal width of irises")
ax.set_xlabel("Petal length (cm)")
ax.set_ylabel("Petal width (cm)")

plt.show()
```




```
### Petal length vs. petal width vs. sepal length for irises (3D)

# We have to make fig, ax in a slightly different way for 3D
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(projection="3d")

ax.scatter(
    df["petal length (cm)"][df["species"] == 0],
    df["petal width (cm)"][df["species"] == 0],
    df["sepal length (cm)"][df["species"] == 0],
    color="r",
    edgecolor='k',
)

ax.scatter(
    df["petal length (cm)"][df["species"] == 1],
    df["petal width (cm)"][df["species"] == 1],
    df["sepal length (cm)"][df["species"] == 1],
    color="g",
    edgecolor='k',
)

ax.scatter(
    df["petal length (cm)"][df["species"] == 2],
    df["petal width (cm)"][df["species"] == 2],
    df["sepal length (cm)"][df["species"] == 2],
    color="b",
```

```
    edgecolor='k',
)
ax.legend(data.target_names)

# Label
ax.set_title("Sepal Length vs. Petal length vs. petal width of irises")
ax.set_xlabel("Petal length (cm)")
ax.set_ylabel("Petal width (cm)")
ax.set_zlabel("Sepal length (cm)")

plt.show()
```

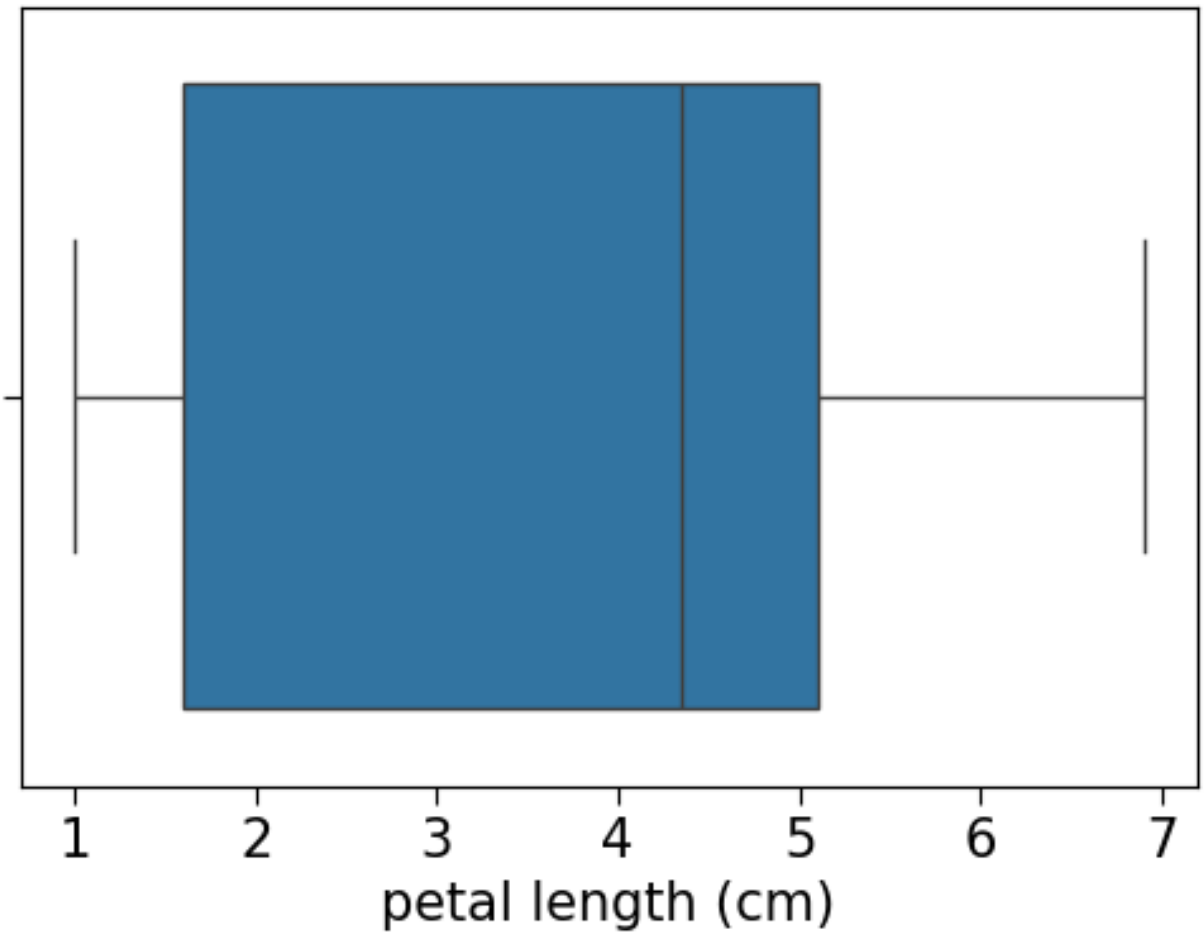


4.1.4 Box plots

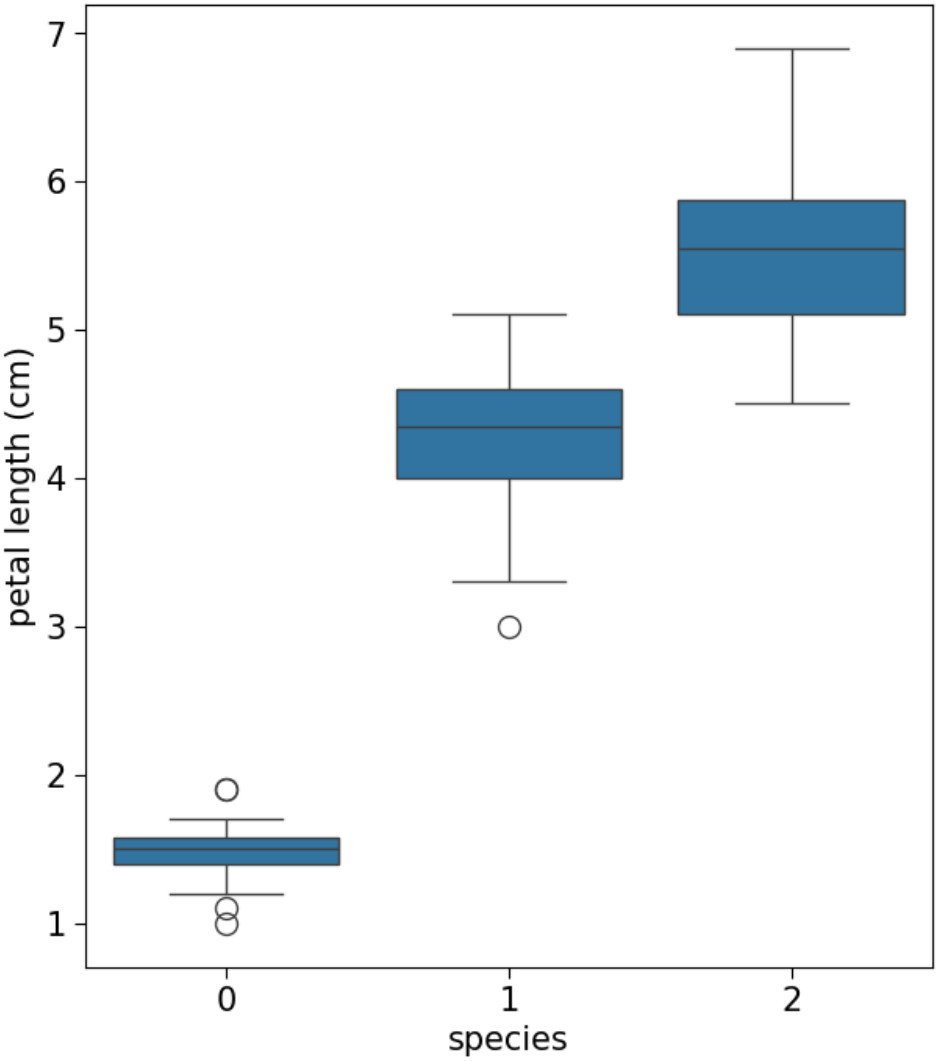
I do use seaborn for boxplots, as they look a lot better.

```
### Box plot of petal length for the iris dataset

# import seaborn
import seaborn as sns

fig = plt.figure(figsize=(6, 4))
```

```
sns.boxplot(x=df['petal length (cm)'])
plt.show()
```



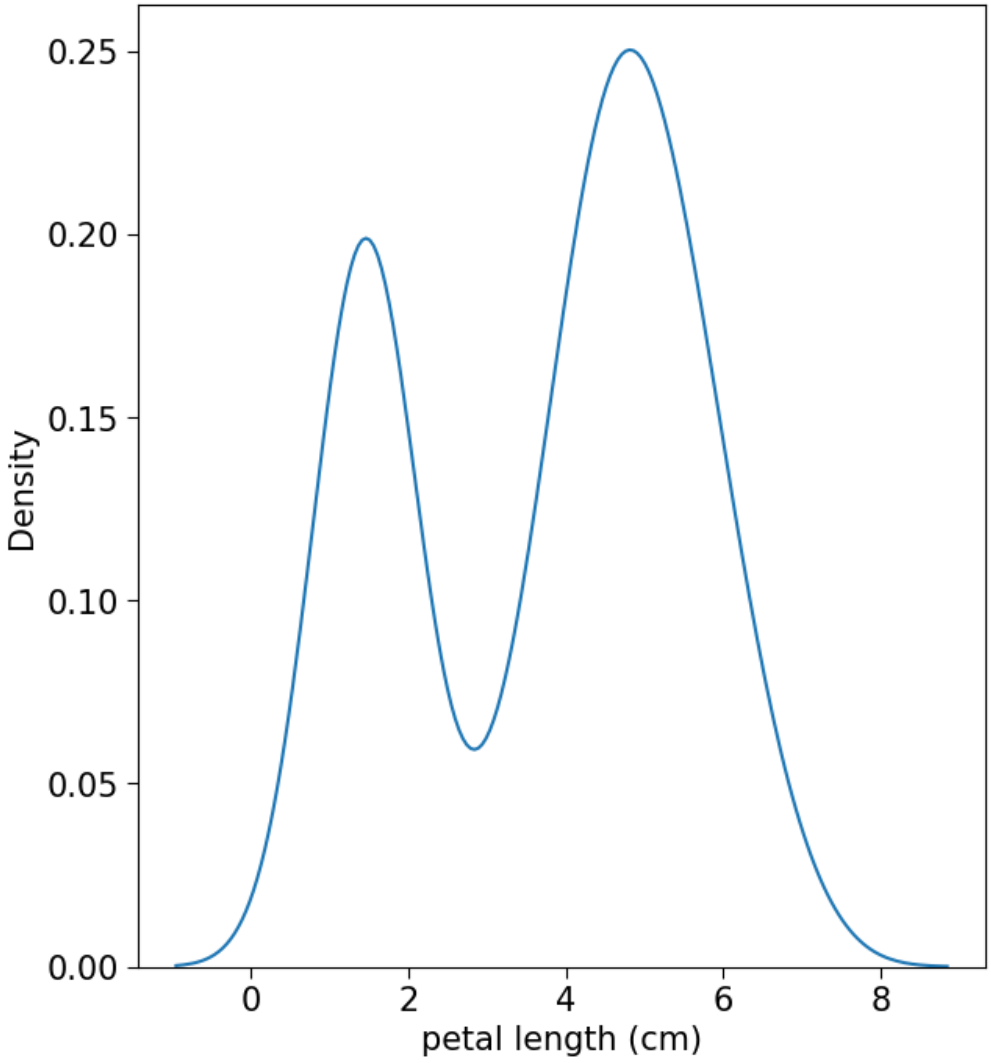

```
### Box plot of petal length for the iris dataset by species
fig = plt.figure(figsize=(7, 8))
sns.boxplot(y=df["petal length (cm)"], x=df["species"])
plt.show()
```



4.1.5 KDE plots

Kernel density estimation (KDE) plots are easiest to do in seaborn.

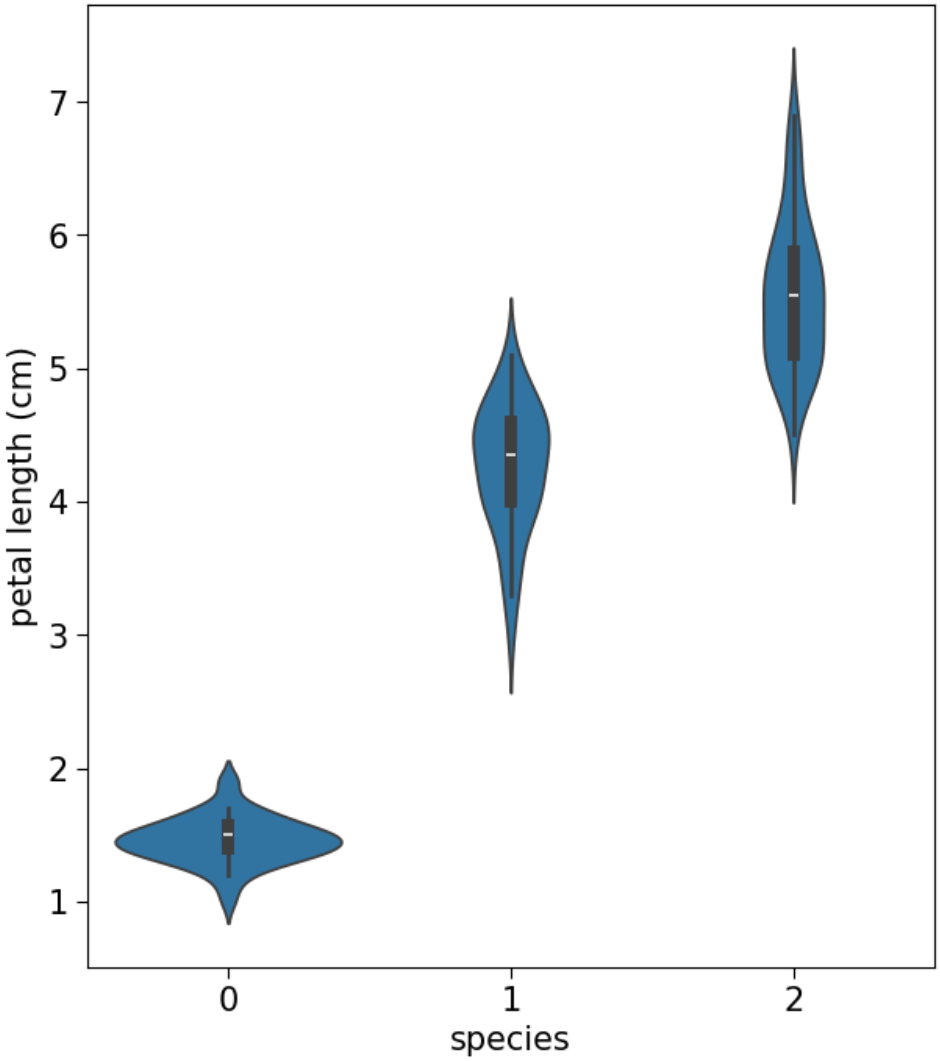
```
### KDE plot of petal length for the iris dataset
fig = plt.figure(figsize=(7, 8))
sns.kdeplot(x=df['petal length (cm)'])
plt.show()
```



4.1.6 Violin plots

Also seaborn!

```
### Violin plot of petal length for the iris dataset by species
fig = plt.figure(figsize=(7, 8))
sns.violinplot(y=df["petal length (cm)"], x=df["species"])
plt.show()
```



4.1.7 Heat maps

Heat maps are a nice way to visualise the magnitude of quantities in a matrix. Here we are going to use seaborn to see how different attributes correlate for a penguin dataset.

Execution note

Plot output is not included in this edition. This example downloads the penguins dataset from the seaborn data repository and requires internet access. Run the notebook to generate the heat map.

```
### Penguin correlation heatmap

# Read pandas dataset from seaborn package
df = sns.load_dataset("penguins", cache=False)

# Compute correlation matrix and plot using seaborn
# Notice have used .corr() from pandas

matrix = df.corr(numeric_only=True).round(2)
sns.heatmap(matrix, annot=True, vmax=1, vmin=-1, center=0,
 ↪  cmap="vlag").set(
    title="Correlations for penguin attributes"
```

```
)
plt.show()
```

4.1.8 Pair plots

These are great for a quick overview of a dataset you are considering.

Execution note

Plot output is not included in this edition. This example uses the penguins dataset loaded in the preceding cell; run both cells with internet access to generate the pair plot.

```
### Penguin pair plot

# The hue argument below does the class separation in the pair plot.
sns.pairplot(df, hue='species')
```

4.2 Dimensionality reduction

Unfortunately, a lot of datasets are high dimensional, and can't really be visualised using the plots we've looked at. One way to resolve this is to use dimensionality reduction techniques to bring a high dimensional dataset down to e.g. 2D where we can just view it on a scatter plot. In the lectures we looked at two dimensionality reduction techniques: PCA and t-SNE.

4.2.1 Principal Component Analysis (PCA)

PCA takes a **standardised** dataset matrix $\mathbf{X} \in \mathbb{R}^{N \times D}$ and returns an orthonormal matrix $\mathbf{W}_{PC} \in \mathbb{R}^{D \times D}$. Note that orthonormal just means that $\mathbf{W}_{PC}^\top \mathbf{W}_{PC} = \mathbf{I}$.

The columns of $\mathbf{W}_{PC}$, which we will denote as $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_D$ are the eigenvectors of the empirical covariance matrix $\mathbf{\Sigma} = \frac{1}{N}\mathbf{X}^\top \mathbf{X}$ sorted from highest to lowest eigenvalue $\lambda_1, \lambda_2, \dots, \lambda_D$. These eigenvectors are known as the **principal components** of the data. These are unit vectors (i.e. of length 1).

We are interested in PCA for the purpose of *dimensionality reduction*. We often have a dataset that is too high dimensional to visualise. We would like a simple linear transform $\mathbf{Z} = \mathbf{X}\mathbf{W}$ that turns our high dimensional data points $\{\mathbf{x}^{(n)}\}_{n=1}^N$ (the rows of $\mathbf{X}$) into low dimensional data points $\{\mathbf{z}^{(n)}\}_{n=1}^N$ (the rows of $\mathbf{Z}$). Here, $\mathbf{x} \in \mathbb{R}^D$ and $\mathbf{z} \in \mathbb{R}^d$ where $d < D$.

In order to quantify how good this transform is, we create a setup where its transpose can be used to (attempt to) reverse the transformation. This means we can compute $\hat{\mathbf{X}} = \mathbf{Z}\mathbf{W}^\top = \mathbf{X}\mathbf{W}\mathbf{W}^\top$ where the rows of $\hat{\mathbf{X}}$ are reconstructions of the original data points $\{\hat{\mathbf{x}}^{(n)}\}_{n=1}^N$.

We can now assess how good a transformation $\mathbf{W}$ is by computing the average (squared) distance between points and their reconstructions.

$$\frac{1}{N}\sum_n \|\mathbf{x}^{(n)} - \hat{\mathbf{x}}^{(n)}\|^2 = \frac{1}{N}\sum_n \|\mathbf{x}^{(n)} - \mathbf{W}\mathbf{W}^\top \mathbf{x}^{(n)}\|^2$$

We can write this as an optimisation problem. We want to solve

$$\underset{\mathbf{W}}{\text{minimise}} \ \frac{1}{N}\sum_n \|\mathbf{x}^{(n)} - \mathbf{W}\mathbf{W}^\top \mathbf{x}^{(n)}\|^2$$

As it happens, we actually want to solve the constrained optimisation problem:

$$\underset{\mathbf{W}}{\text{minimise}} \ \frac{1}{N}\sum_n \|\mathbf{x}^{(n)} - \mathbf{W}\mathbf{W}^\top \mathbf{x}^{(n)}\|^2 \ \text{ subject to } \ \mathbf{W}^\top\mathbf{W} = \mathbf{I}$$

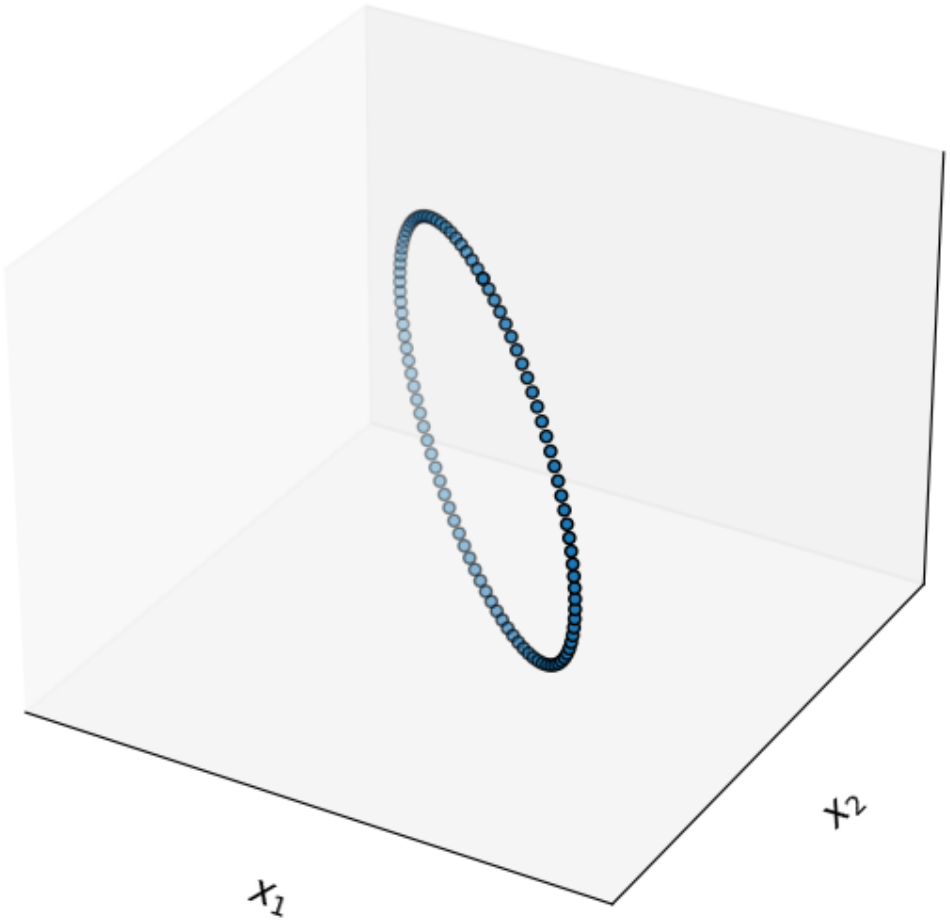
The constraint makes the columns of $\mathbf{W}$ orthonormal.

What has this got to do with PCA? Well, the solution to this optimisation problem when projecting down to d dimensions is the matrix whose d columns are the first d principal components of the data.

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1 & \mathbf{w}_2 & \dots & \mathbf{w}_d \end{bmatrix}$$

Let's visit the ellipse example in the lecture to see PCA in action.

```
# Generate points on an unrotated ellipse
a = 10
b = 1
theta = np.linspace(0,2*np.pi,100)

y = a*np.sin(theta)
z = b*np.cos(theta)
x = np.zeros(100)

# Stack into dataset matrix
X = np.vstack((x,y,z)).T

# Rotate
t = np.pi / 4
R = np.array([[np.cos(t), np.sin(t),0],[-np.sin(t),
 ↪  np.cos(t),0],[0,0,1]])
X = X @ R

# Standardise in a single line. This is an alternative to what we did
 ↪  in the last cell
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Finally, plot
fig = plt.figure(figsize=(12,6))
ax = fig.add_subplot(projection='3d')
ax.grid()
ax.set_xlim([-2,2])
ax.set_ylim([-2,2])
ax.set_zlim([-2,2])
ax.set_xticks([])
ax.set_yticks([])
ax.set_zticks([])
ax.scatter(X[:,0],X[:,1],X[:,2],edgecolor='k')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
# Annoyingly, this x_3 doesn't seem to show below. My apologies.
ax.set_zlabel('$x_3$')
fig.tight_layout()
```

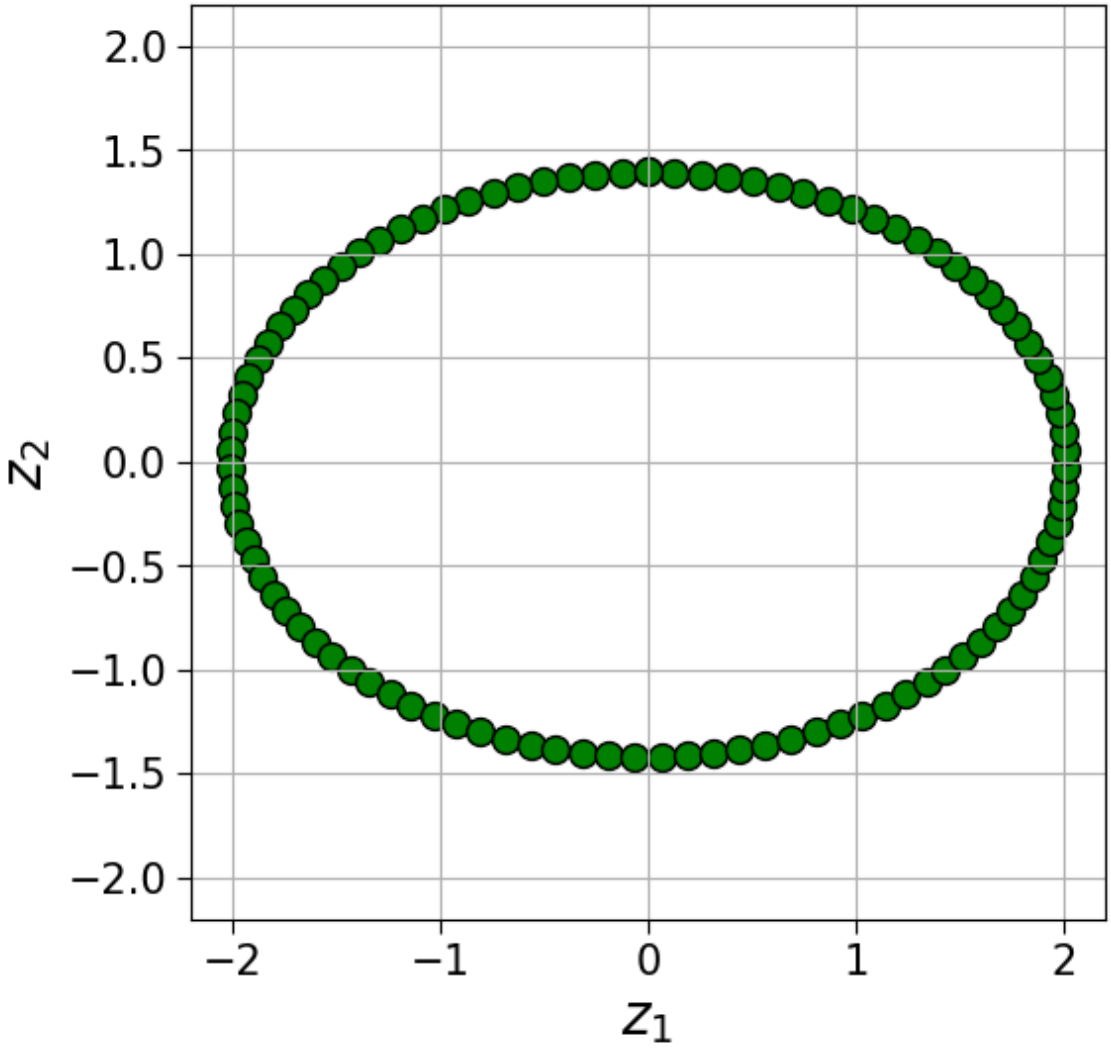


Here, we have a dataset of points that maps out an ellipse in 3D. We know that ellipses are only 2D so we would hope that if we projected this dataset down to 2D we would get something that looks an awful lot like an ellipse.

Let's use the PCA class from sklearn to compute $\mathbf{W}$ and then $\mathbf{Z} = \mathbf{XW}$. When creating an object from this class, it takes in an argument `n_components`. This refers to the number of principal components we want to keep. In this example, we want to go down to 2D so we need to keep 2 principal components.

```
# Import PCA package

# Create a pca object
pca = PCA(n_components=2)

# Calling pca.fit(X) computes W behind the scenes
pca.fit(X)

# Calling pca.transform(X) then computes X@W
Z = pca.transform(X)

# Let's now plot our transformed dataset to see if it makes sense

# Plot Z
fig, ax = plt.subplots(figsize=[6,6])
```

```
ax.set_aspect("equal", "box")
ax.grid()
ax.set_xlim([-2.2, 2.2])
ax.set_ylim([-2.2, 2.2])
ax.scatter(Z[:, 0], Z[:, 1],color='g',edgecolor='k')
ax.set_xlabel("$z_1$", size=20)
ax.set_ylabel("$z_2$", size=20)
plt.show()
```



Great! It's an ellipse! In addition to giving you a transformation that minimises reconstruction error, the principal components (i.e. the features in the transformed space) correspond to (orthogonal) directions of maximum variance. This is why z_1 above is the major axis of the ellipse, and z_2 is the minor axis because the variance along the first axis is greatest, followed by the variance along the second.

We can quantify how much of the variance has been captured in these dimensions by considering the ratio of the eigenvalues corresponding to the principal components we've used relative to the sum of all eigenvalues of $\mathbf{\Sigma}$. Sklearn computes these automatically:

```
# The amount of variance captured in the 1st and 2nd principal
 ↪  component
# It captures all of the variance because the third dimension is
 ↪  redundant in describing a 2D shape
pca.explained_variance_ratio_
```

```
array([0.66666667, 0.33333333])
```

4.2.2 t-distributed Stochastic neighbour embedding (t-SNE)

We saw in the lecture that there were some limitations to using PCA in that it was linear. t-SNE is a very popular non-linear technique that produces a low-dimensional mapping that preserves local structure. Instead of learning some transformation matrix, we **directly learn the values** of each data point in this mapping.

> This means that you can't use t-SNE to map a new point. You would have to run the algorithm again incorporating the new point.

To do this, we first devise an expression for $p_{i,j}$: the probability that two point $\mathbf{x}^{(i)}$ and $\mathbf{x}^{(j)}$ are paired in the original space. In t-SNE we use something Gaussian-like:

$$p_{i,j} = \frac{p_{i|j} + p_{j|i}}{2N}$$

where

$$p_{j|i} = \frac{\exp(-\frac{1}{2\sigma_i^2}\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2)}{\sum_{k \neq i} \exp(-\frac{1}{2\sigma_i^2}\|\mathbf{x}^{(i)} - \mathbf{x}^{(k)}\|^2)}$$

and each bandwidth σ_i is chosen to match a user-selected **perplexity**, which controls the effective neighbourhood size. We set $p_{i,i} = 0$. This gives high pair probabilities to nearby points.

The next step is to devise an expression for $q_{i,j}$: the probability that the low dimensional embeddings of these points $\mathbf{z}^{(i)}$ and $\mathbf{z}^{(j)}$ are paired. In t-SNE we use a Cauchy distribution which has long tails:

$$q_{i,j} = \frac{(1 + \|\mathbf{z}^{(i)} - \mathbf{z}^{(j)}\|^2)^{-1}}{\sum_{k \neq l}(1 + \|\mathbf{z}^{(k)} - \mathbf{z}^{(l)}\|^2)^{-1}}$$

We also set $q_{i,i} = 0$.

> If you think these expressions are a bit random and unprincipled, then you are absolutely right. Often empirical effectiveness trumps all else in ML.

Given some pair, we would like the probabilities of pairing to be preserved in both the original space and the embedded space. Given what we know about linear regression, it is tempting to write a loss function that is the sum (or mean) of the distances between probabilities. However, we can do better. Given that we are working with probabilities, we can use the KL divergence:

$$D_{KL}(P\|Q) = \sum_i \sum_j p_{i,j} \log \frac{p_{i,j}}{q_{i,j}}$$

which is a measure of how one probability distribution diverges from another. Specifically, we can find the exact embeddings $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots, \mathbf{z}^{(N)}$ that minimise this quantity by solving $\underset{\mathbf{z}^{(1)},\mathbf{z}^{(2)},\ldots,\mathbf{z}^{(N)}}{\text{minimise}} D_{KL}(P\|Q)$.

This can be done using good old Gradient descent or something similar. You will compare t-SNE to PCA in the lab.

5 Model selection and evaluation

```
# RUN THIS CELL FIRST

# Imports
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# Sklearn imports
from sklearn.datasets import make_moons, make_blobs, load_iris,
 ↪  make_classification
from sklearn.preprocessing import MaxAbsScaler, StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.discriminant_analysis import (
    QuadraticDiscriminantAnalysis,
    LinearDiscriminantAnalysis,
)
from sklearn.model_selection import train_test_split, cross_validate,
 ↪  GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import RocCurveDisplay
from sklearn.pipeline import Pipeline

# This makes matplotlib output nice figures without much tweaking
plt.rcParams.update(
    {
        "lines.markersize": 10,  # Big points
        "font.size": 15,  # Larger font
        "xtick.major.size": 5.0,  # Bigger xticks
        "ytick.major.size": 5.0,  # Bigger yticks
    }
)
```

Before we go onto the main theme of the week (model selection and evaluation, as it would happen), we're going to do a tour of some other models for classification. These are models that didn't quite make the cut on the course syllabus, but are relevant and worth being aware of.

5.1 k-nearest neighbours (k-NN)

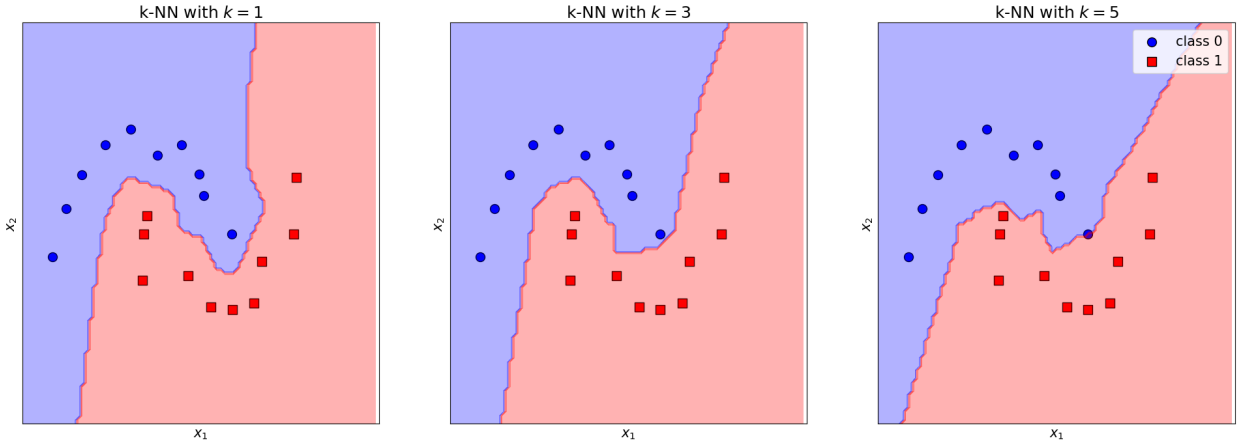
This is a very simple non-parametric model for classification.

k-NN has a single hyperparameter k. Given a training set of N data point-target pairs $\{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$, a new point can be classified by:

1. Computing the distance between the new point and every training point
2. Identifying the k training points with the lowest distance to the new point
3. Taking the mode class label of these k nearest points

The code below creates a two-class training set, and then displays the decision boundary of a k-NN classifier for different values of k.

```
X, y = make_moons(n_samples=20, noise=0.1, random_state=42)

fig, axes = plt.subplots(1,3,figsize=[24,8])

for i,k in enumerate([1,3,5]):
    axes[i].scatter(X[y==0,0],X[y==0,1],color='b',edgecolor='k')
    axes[i].scatter(X[y==1,0],X[y==1,1],color='r',marker='s',edgecolor
     ↪ ='k')


    axes[i].set_xlabel('$x_1$')
    axes[i].set_ylabel('$x_2$')
    plt.legend(['class 0','class 1'], loc='best')
    axes[i].set_xticks([])
    axes[i].set_yticks([])
    axes[i].set_xlim([-1.5,3])
    axes[i].set_ylim([-1.5,2])

    model = KNeighborsClassifier(n_neighbors=k).fit(X,y)
    colours = ['b','r']
    axes[i].set_title(f'k-NN with $k={k}$')
    disp = DecisionBoundaryDisplay.from_estimator(
        model,
        X,
        response_method="predict",
        alpha=0.3,
        grid_resolution=100,
        cmap=ListedColormap(colours),
        ax=axes[i],
    )

plt.show()
```



k-NN naturally extends to multiclass problems.

5.2 Generative models [Optional]

While you may be expected to understand these at a high level in Sklearn, you do not need to understand their inner-workings as part of the exam. You will however be expected to use Bayes' rule when required.

A logistic regression classifier is known as a **discriminative model**. It gives us $p(y = k|\mathbf{x})$ for some class k through the log-odds/logits, and only lets us discriminate between classes.

Generative models on the other hand give us $p(\mathbf{x}, y = k)$. This lets us perform classification by obtaining $p(y = k|\mathbf{x})$ through Bayes rule, as well as letting us generate new data points for each class by sampling from $p(\mathbf{x}|y = k)$.

Typically, discriminative classifiers are seen as being more accurate (see Murphy 9.4.1) but it is worth knowing about both.

We will consider Quadratic discriminant analysis (QDA), Linear discriminant analysis (LDA), and Gaussian Naive Bayes classifiers. These are all closely related.

We start by considering a classification problem where there are K classes ($y \in \mathbb{Z}^+_{<K} = \{0, 1, \dots, K-1\}$). Let's look at the probability that the correct class label is k given some data point $\mathbf{x} \in \mathbb{R}^D$: $p(y = k|\mathbf{x})$. By applying Bayes' rule we can write

$$p(y = k|\mathbf{x}) = \frac{p(\mathbf{x}|y = k)p(y = k)}{p(\mathbf{x})}$$

It is sensible to classify a data point according to the most probable class label $\underset{k}{\text{argmax}} p(y = k|\mathbf{x})$. As $p(\mathbf{x})$ is just some constant that is the same for all classes, we can ignore it.

$$p(y = k|\mathbf{x}) \propto p(\mathbf{x}|y = k)p(y = k)$$

We will therefore classify according to

$$\underset{k}{\text{argmax}} p(\mathbf{x}|y = k)p(y = k)$$

All the methods we are about to consider are different ways of modelling $p(\mathbf{x}|y = k)$ for each k. $p(y = k)$ is simply the fraction of data points within our training set that belong to class k. Note that these are generative models according to the definition above because $p(\mathbf{x}|y = k)p(y = k) = p(\mathbf{x}, y = k)$.

5.2.1 QDA

For Quadratic discriminant analysis (QDA) we model each $p(\mathbf{x}|y = k)$ with a Multivariate Gaussian distribution.

$$p(\mathbf{x}|y = k) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}_k|^{1/2}} \exp\Big[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1}(\mathbf{x} - \boldsymbol{\mu}_k) \Big]$$

where $\boldsymbol{\mu} \in \mathbb{R}^{\mathbb{D}}$ is the mean vector, and $\boldsymbol{\Sigma} \in \mathbb{R}^{\mathbb{D}\times\mathbb{D}}$ is the covariance matrix of the distribution.

$$\boldsymbol{\mu}_k = \begin{bmatrix} \mu_1 & \mu_2 & \dots & \mu_D \end{bmatrix}^\top$$

$$\boldsymbol{\Sigma}_k = \begin{bmatrix} \Sigma_{1,1} & \Sigma_{1,2} & \dots & \Sigma_{1,D} \\ \Sigma_{2,1} & \Sigma_{2,2} & \dots & \Sigma_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{D,1} & \Sigma_{D,2} & \dots & \Sigma_{D,D} \end{bmatrix}$$

Multivariate Gaussians will crop up a lot for Gaussian Processes later in the course!

These values can all be computed empirically from training data (which is equivalent to performing maximum likelihood estimation). Consider the subset of N_k training points belonging to class k: $\mathfrak{D}_k = \{(\mathbf{x}^{(n)}, y^{(n)}) \in N : y^{(n)} = k\}$.

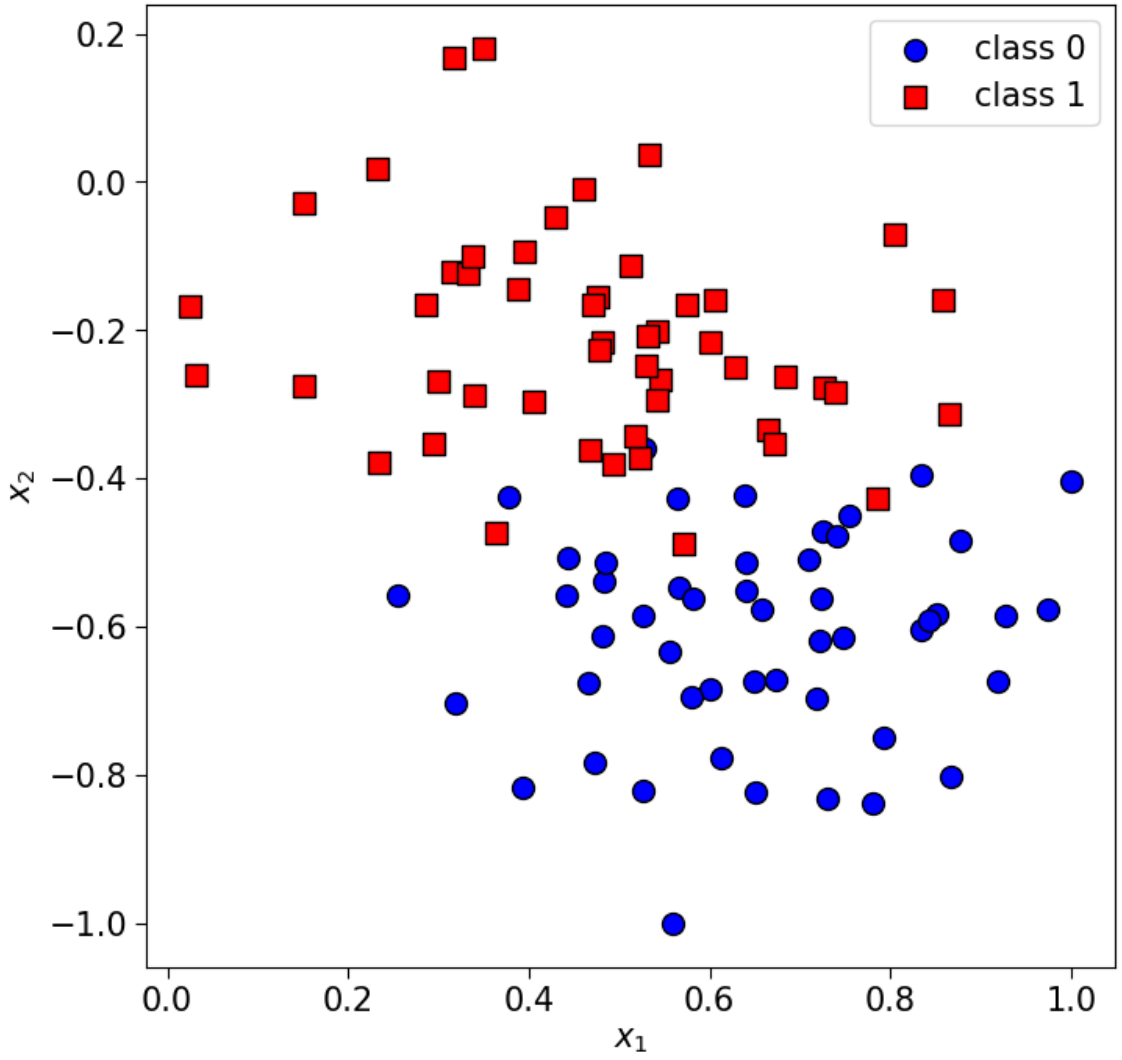
Then

$$\mu_d = \frac{1}{N_k} \sum_{n=1}^{N_k} x_d^{(n)}$$

$$\Sigma_{i,j} = \frac{1}{N_k} \sum_{n=1}^{N_k} (x_i^{(n)} - \mu_i)(x_j^{(n)} - \mu_j)$$

Let's make it clear what is going on by considering a binary classification problem.

```
X, y = make_blobs(centers=2, random_state=44, cluster_std=1.8)
X = MaxAbsScaler().fit_transform(X)

fig, ax = plt.subplots(figsize=[8, 8])
ax.scatter(X[y == 0, 0], X[y == 0, 1], color="b", edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color="r", marker="s",
 ↪  edgecolor="k")
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1"], loc="best")
plt.show()
```



When we fit a QDA model, we basically expect there to be μ_0, Σ_0 for class 0, and μ_1, Σ_1 for class 1.

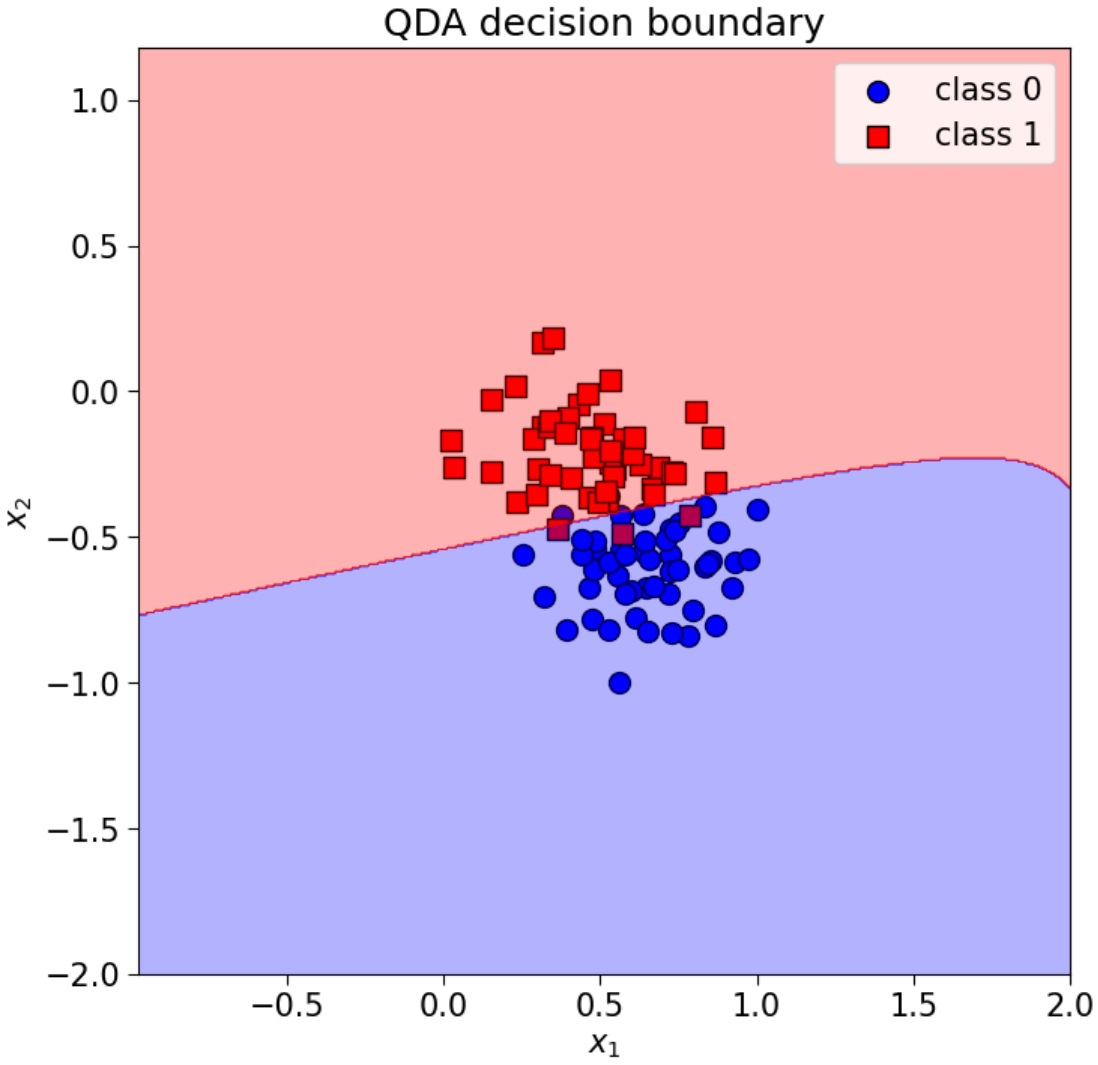
```
model = QuadraticDiscriminantAnalysis(store_covariance=True).fit(X, y)

# Print the array containing mu_0 and mu_1
print(model.means_)

# Print the array containing sigma_0 and sigma_1
print(model.covariance_)
```

```
[[ 0.64887613 -0.61097152]
 [ 0.47130899 -0.2100766 ]]
[array([[0.029616  , 0.00198287],
       [0.00198287, 0.0192939 ]]), array([[ 0.03762276, -0.00663298],
       [-0.00663298,  0.02108555]])]
```

Looking at the values in the mean array we can see that they correspond to the centres of the blue and red clusters in the figure above, which makes sense.

Plotting the decision boundary of the model, we can see it is quadratic (just about!), hence the name QDA.

```
fig, ax = plt.subplots(figsize=[8, 8])
ax.scatter(X[y == 0, 0], X[y == 0, 1], color="b", edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color="r", marker="s",
 ↪  edgecolor="k")
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1"], loc="best")
colours = ["b", "r"]
ax.set_title(f"QDA decision boundary")
disp = DecisionBoundaryDisplay.from_estimator(
    model,
    X,
    response_method="predict",
    alpha=0.3,
    grid_resolution=500,
    cmap=ListedColormap(colours),
    ax=ax,
)
plt.show()
```



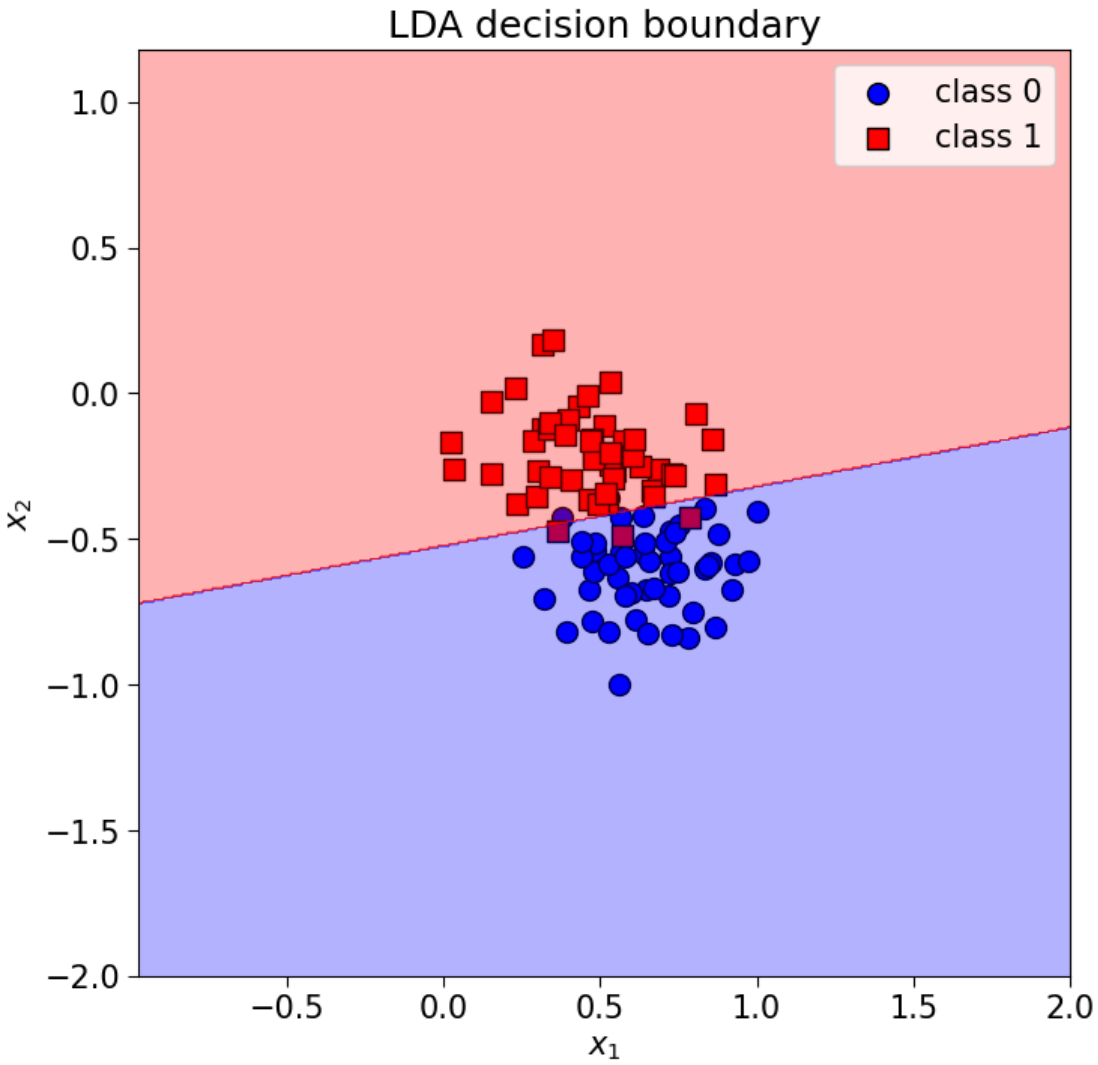
5.2.2 LDA

In Linear Discriminant Analysis, we again model each $p(\mathbf{x}|y = k)$ using a multivariate Gaussian, except we assume that the covariance matrices are tied between classes $\mathbf{\Sigma}_0 = \mathbf{\Sigma}_1 = \cdots = \mathbf{\Sigma}$. We estimate this shared covariance by pooling within-class covariances, measuring deviations from each class mean. This gives a linear decision boundary.

```
model = LinearDiscriminantAnalysis(store_covariance=True).fit(X, y)

# Print the array containing mu_0 and mu_1
print(model.means_)

# Print the array containing the shared sigma
print(model.covariance_)
```

```
[[ 0.64887613 -0.61097152]
 [ 0.47130899 -0.2100766 ]]
[[ 0.03294699 -0.00227855]
 [-0.00227855  0.01978593]]
```

```
fig, ax = plt.subplots(figsize=[8, 8])
ax.scatter(X[y == 0, 0], X[y == 0, 1], color="b", edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color="r", marker="s",
 ↪  edgecolor="k")
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1"], loc="best")
colours = ["b", "r"]
ax.set_title(f"LDA decision boundary")
disp = DecisionBoundaryDisplay.from_estimator(
    model,
    X,
    response_method="predict",
    alpha=0.3,
    grid_resolution=500,
    cmap=ListedColormap(colours),
    ax=ax,
)
plt.show()
```



5.2.3 Gaussian Naive Bayes

The starting point for Gaussian Naive Bayes is QDA. That is, for each class k we have:

$$p(\mathbf{x}|y = k) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

We however make the "naive" assumption that the features of our data are uncorrelated so that each $\boldsymbol{\Sigma}_k$ is a diagonal matrix. This lets us write each multivariate Gaussian as a product of univariate Gaussians.

$$p(\mathbf{x}|y = k) = \prod_d p(x_d|y = k) = \prod_d \mathcal{N}(x_d|\mu_d, \Sigma_{d,d})$$

Other types of Naive Bayes exist where another univariate distribution is used in the equation above. For instance, Multinomial Naive Bayes uses a ... multinomial distribution!

5.3 Model selection

Hopefully by now you are happy with the purpose of the train/validation/test split used in machine learning.

- The training set is for training different models
- The validation set is for evaluating trained models and selecting the best one
- The test set is for a final **unbiased** estimate of how well the chosen model will perform when deployed

On this course, the validation set was initially introduced in the context of hyperparameter tuning (specifically a regularisation hyperparameter). You can think of different types of models (e.g. logistic regression, k-NN, SVMs, QDA, etc. etc.) as a hyperparameter that can be tuned.

In this sense, the purpose of the validation set is to perform **model selection** i.e. finding the best type of model with the best choice of hyperparameters. Exhaustively searching over all models and hyperparameters isn't possible so pragmatism is required, along with some intuition about what will work.

I have actually been guiding you through the model selection process in the labs, whereby you evaluate multiple models on the validation set, and at the very end select and evaluate **one** on the test set.

5.4 Cross-validation

Using a dedicated validation set has two disadvantages:

1. You are sacrificing data that could be used for training.
2. Model selection will be sensitive to the way the dataset was partitioned.

An alternative strategy is to evaluate models through cross-validation. This only requires splitting the original dataset into training and test.

Then, to perform k-fold cross-validation on a model:

- Split the training set into k folds
- Use each fold as if it were a validation set: train a model on the other folds and evaluate on the given fold
- Average the performance across each fold

Let's examine cross-validation on the iris dataset (bearing in mind that the iris dataset is so easy we expect to see very high numbers).

```
# Load dataset
X, y = load_iris(return_X_y=True)

# Split into train/test
X_train, X_test, y_train, y_test = train_test_split(X, y,
 ↪  random_state=42)

# Put standardisation inside a pipeline so it is fitted only on the
# training portion of each fold, without seeing that fold's validation
 ↪  data.
pipe = Pipeline([("scaler", StandardScaler()), ("logreg",
 ↪  LogisticRegression())])

# Cross-validate the whole pipeline
scores = cross_validate(pipe, X_train, y_train)

# Look at results
print(scores)
```

```
{'fit_time': array([0.00193   , 0.00088   , 0.00078177, 0.000736  ,
 ↪  0.00077295]), 'score_time': array([0.00017881, 0.00015998, 0.0001421 ,
 ↪  0.00013804, 0.000144  ]), 'test_score': array([0.95652174, 1.        ,
 ↪  0.86363636, 1.        , 0.95454545])}
```

This did 5-fold cross-validation by default. What we are interested in here is what is unhelpfully called `test_score` (this isn't test data!). There are 5 numbers that correspond to the 5 accuracies obtained when performing cross-validation. We can score the model by the mean of these quantities when comparing it to other models.

5.5 Grid search with k-fold cross validation

If we are performing a grid search over hyperparameter values, we can score each element in the grid by the mean accuracy obtained through cross-validation.

Sklearn lets us do this quite simply. We will use a 1D grid of regularisation hyperparameters as an example. In a pipeline, `logreg__C` refers to the `C` parameter of the step named `logreg`.

```
hyperparameters = {"logreg__C": [1, 10, 100, 1000]}

clf = GridSearchCV(pipe, hyperparameters)
clf.fit(X_train, y_train)

print(clf.cv_results_)
```

```
{'mean_fit_time': array([0.00193181, 0.00112429, 0.00276151, 0.00196571]),
 ↪  'std_fit_time': array([0.00060527, 0.0002717 , 0.00078131,
 ↪  0.00064866]), 'mean_score_time': array([0.00068846, 0.00019302,
 ↪  0.00036511, 0.00027704]), 'std_score_time': array([7.61762644e-04,
 ↪  7.23231039e-05, 9.29525031e-05, 5.52343350e-05]), 'param_logreg__C':
 ↪  masked_array(data=[1, 10, 100, 1000],
             mask=[False, False, False, False],
```

```
fill_value=999999), 'params': [{'logreg__C': 1}, {'logreg__C': 10},
 ↪ {'logreg__C': 100}, {'logreg__C': 1000}], 'split0_test_score':
 ↪ array([0.95652174, 0.95652174, 0.95652174, 0.95652174]),
 ↪ 'split1_test_score': array([1.        , 1.        , 1.        ,
 ↪ 0.95652174]), 'split2_test_score': array([0.86363636,
 ↪ 0.81818182, 0.81818182, 0.81818182]), 'split3_test_score':
 ↪ array([1., 1., 1., 1.]), 'split4_test_score':
 ↪ array([0.95454545, 0.95454545, 1.        , 1.        ]),
 ↪ 'mean_test_score': array([0.95494071, 0.9458498 , 0.95494071,
 ↪ 0.94624506]), 'std_test_score': array([0.04979923, 0.0668627 ,
 ↪ 0.07042232, 0.06691876]), 'rank_test_score': array([1, 4, 1,
 ↪ 3], dtype=int32)}
```

Quite a bit to process there, but you can see `mean_test_score` has the mean accuracies for each value of `C` in the grid.

To make things easier, this `GridSearchCV` object will train a model on all the training data with the best hyperparameters so it can then be directly used just like any other sklearn model.

There are many techniques for hyperparameter tuning that are better than grid search, but these are beyond the scope of this course!

5.6 Pipelines

The iris examples above put `StandardScaler` inside a pipeline. This matters for cross-validation: each fold must fit its scaler using only its own training portion. Fitting the scaler on all of `X_train` before cross-validation would leak information from the validation folds. The held-out test set must also never be used to fit the scaler.

After model selection, we can fit the whole pipeline on all the training data:

```
pipe = Pipeline([("scaler", StandardScaler()), ("logreg",
 ↪ LogisticRegression())])
pipe.fit(X_train, y_train)
```

```
Pipeline(steps=[('scaler', StandardScaler()), ('logreg',
 ↪ LogisticRegression())])
```

This first fits the scaler and then fits the model on the transformed training data. The pipeline can be used just like any other classifier object; it automatically applies the fitted scaler to new inputs.

5.7 Evaluating binary classifiers

We typically default to accuracy (the proportion of correct classified examples) when it comes to evaluating classifiers (binary or otherwise), but this isn't that informative if there is a significant class imbalance. For instance, consider a scenario where there are 2 classes and 90% of the examples are in class 0 with the remaining 10% in class 1. A dummy classifier could just predict everything as being in class 0, and would get 90% accuracy which sounds vastly better than it is.

If we denote one class as being positive with P examples, and the other as *negative* with N examples then we can define the true positive rate:

$$TPR = \frac{TP}{P}$$

and the false positive rate:

$$FPR = \frac{FP}{N}$$

Where positives are e.g. some rare disease we would want high TPR and low FPR.

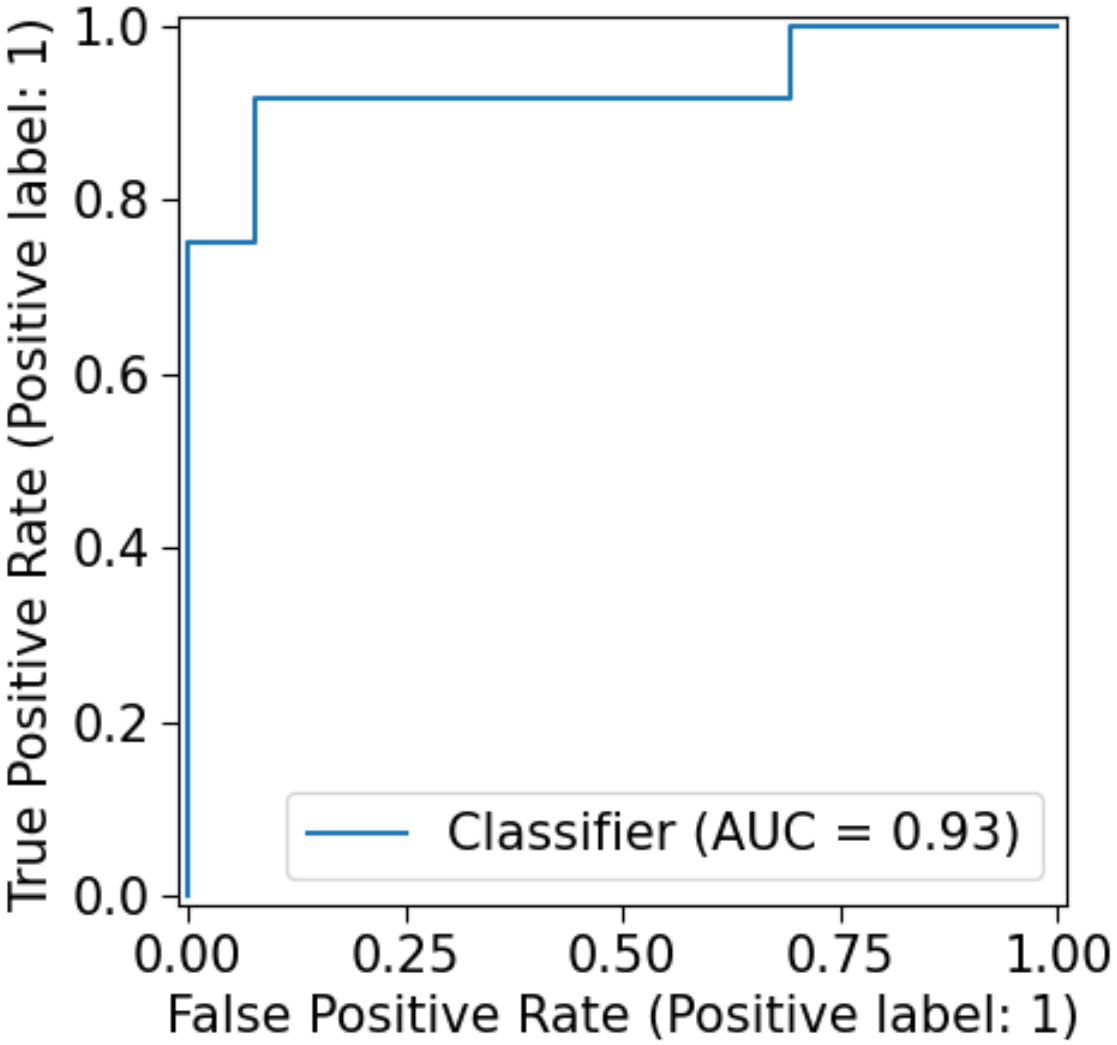
The predictions of a binary classifier are made according to some threshold function

$$\hat{y} = \begin{cases} 1 & \text{if} f(\mathbf{x}) \geq \tau \\ 0 & \text{if} f(\mathbf{x}) < \tau \end{cases}$$

and changing τ trades off TPR against FPR.

With this, we can summarise the performance of a binary classifier in the context of unbalanced classes by considering its Receiver Operating Characteristic (ROC) curve, which is a plot of TPR against FPR as we vary τ.

```
# Generate dummy data for a 2 class problem
X, y = make_classification(random_state=0)

# Split into train and val (let's ignore test)
X_train, X_val, y_train, y_val = train_test_split(X, y, random_state=0)

# Fit a Logistic regression classifier
clf = LogisticRegression(random_state=0).fit(X_train, y_train)

# Decision function gives f(x) (log-odds)
y_logodds = clf.decision_function(X_val)
RocCurveDisplay.from_predictions(y_val, y_logodds)
plt.show()
```



Ideally we would want the curve to hug the top-left corner. With this in mind, we can summarise the curve by the area underneath it (the AUC). If we wanted to get access to the TPR, FPR and τ values that make up the curve then we can use roc_curve.

In the lecture we also consider PR curves, which are closely related.

7 Decision trees and ensemble methods

```
# RUN THIS CELL FIRST

# imports
import numpy as np
import matplotlib.pyplot as plt
import graphviz
from matplotlib.colors import ListedColormap

# Sklearn imports
from sklearn.tree import (
    DecisionTreeClassifier,
    DecisionTreeRegressor,
    plot_tree,
    export_graphviz,
)
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from sklearn.ensemble import BaggingClassifier, GradientBoostingClassifier

# This makes matplotlib output nice figures without much tweaking
plt.rcParams.update(
    {
        "lines.markersize": 10,  # Big points
        "font.size": 15,  # Larger font
        "xtick.major.size": 5.0,  # Bigger xticks
        "ytick.major.size": 5.0,  # Bigger yticks
    }
)
```

7.1 Non-parametric models

The linear models we have considered thus far are parametric. They have a fixed set of parameters which effectively means they have a fixed capacity. The number of parameters doesn't scale with the number of data points. k-nearest neighbours which we saw earlier in the course is an example of a non-parametric model. Intuitively, its decision boundary gets more complicated the more data points there are; it is a model whose capacity scales with the number of data points.

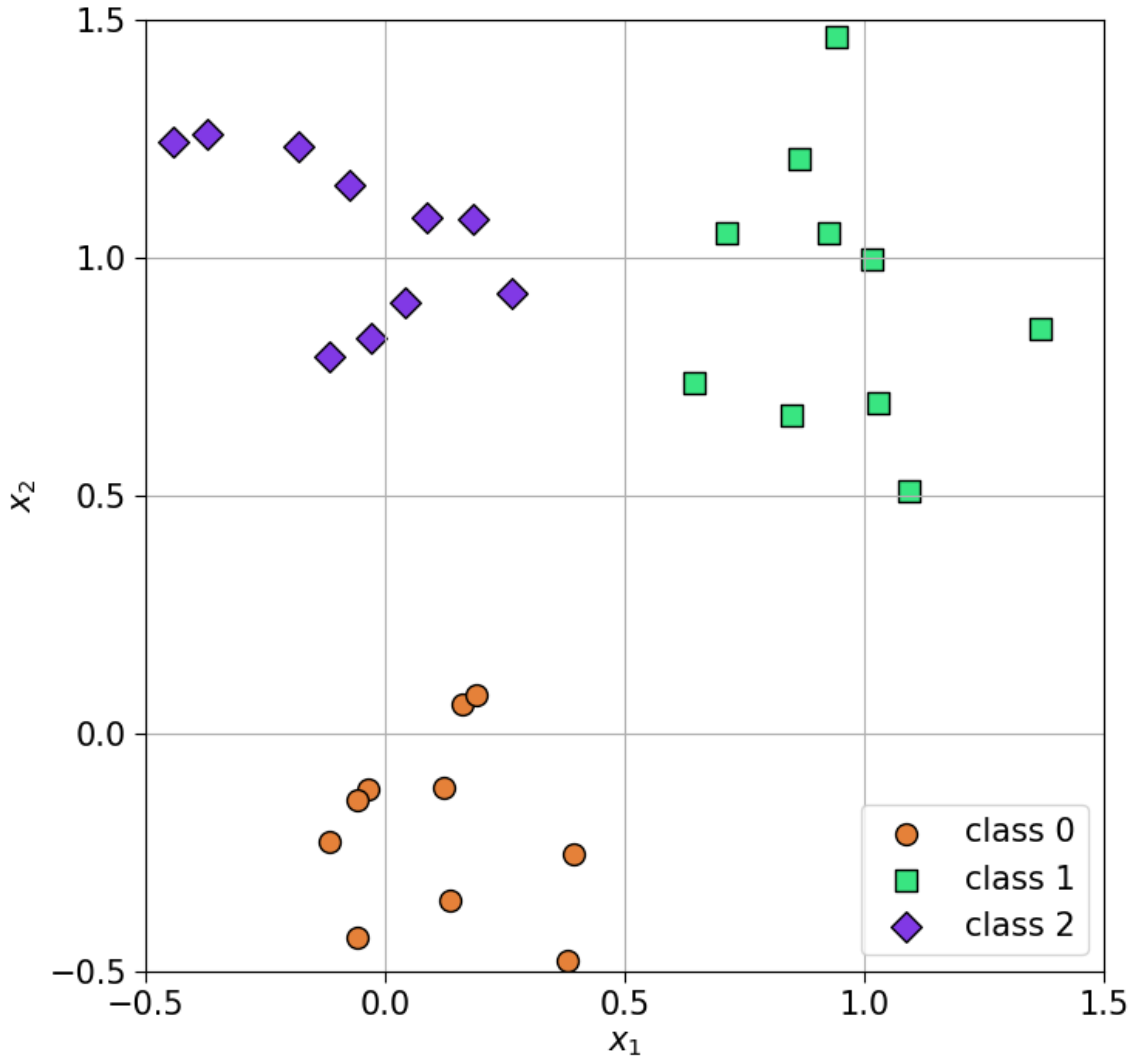
Another family of non-parametric models are classification and regression trees (CARTs) which are also known as **decision trees**. Ensembles of these trees are worth being aware of because they are strong baselines on tabular datasets and often outperform deep neural networks.

While tabular datasets are arguably less exciting than datasets of text or images, they do make up a lot of the world's data. Think how many Excel spreadsheets that you have come across in your life so far!

7.2 Classification trees

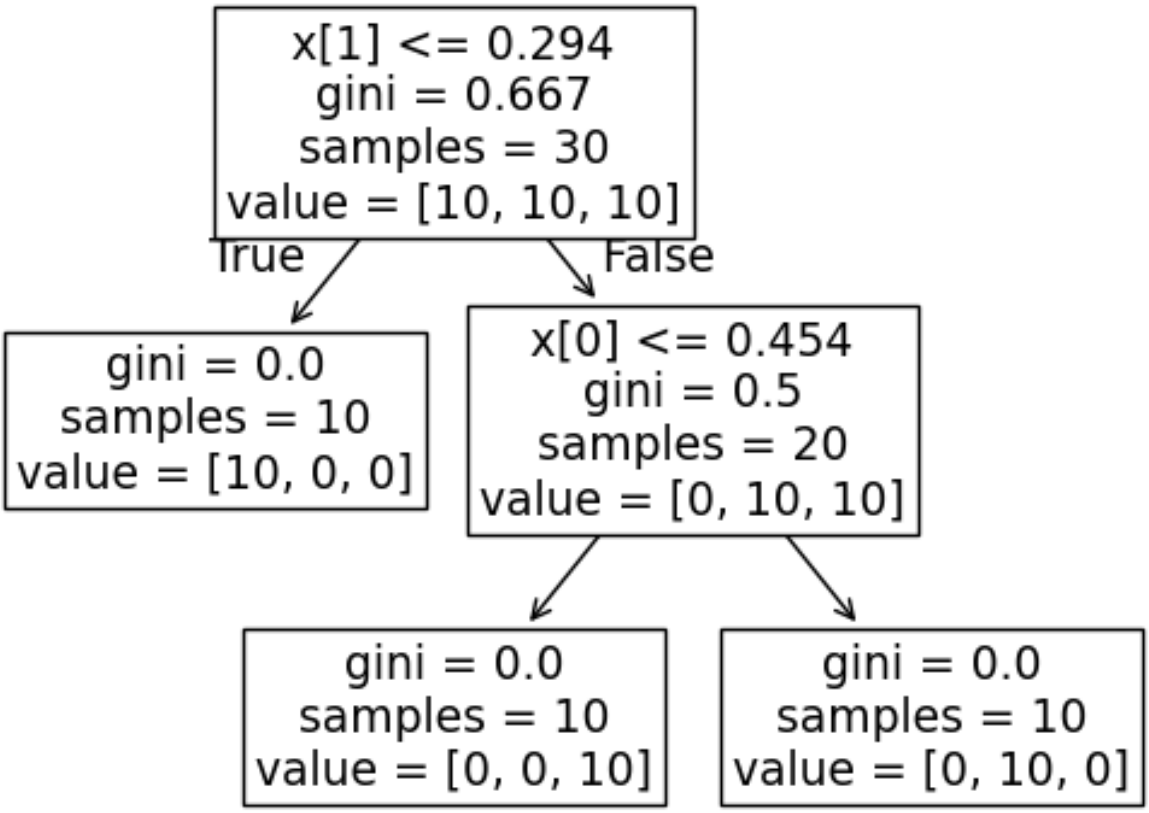
These look like upside down trees, and are best understood visually. First, let's generate and plot some training data for a multiclass problem.

```
np.random.seed(42)

x0 = np.vstack(
    (np.random.normal(0, 0.25, size=10), np.random.normal(0, 0.25,
     ↪  size=10))
).T

x1 = np.vstack(
    (np.random.normal(1, 0.25, size=10), np.random.normal(1, 0.25,
     ↪  size=10))
).T

x2 = np.vstack(
    (np.random.normal(0, 0.25, size=10), np.random.normal(1, 0.25,
     ↪  size=10))
).T

X = np.vstack((x0, x1, x2))
y = np.hstack((0 * np.ones(10), 1 * np.ones(10), 2 * np.ones(10)))

fig, ax = plt.subplots(figsize=[8, 8])

colours = ["#e58139", "#39e581", "#8139e5"]

ax.scatter(X[y == 0, 0], X[y == 0, 1], color=colours[0], edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color=colours[1], marker="s",
 ↪  edgecolor="k")
ax.scatter(X[y == 2, 0], X[y == 2, 1], color=colours[2], marker="D",
 ↪  edgecolor="k")
ax.grid()

ax.set_xticks([-0.5, 0, 0.5, 1, 1.5])
ax.set_yticks([-0.5, 0, 0.5, 1, 1.5])
ax.set_xlim([-0.5, 1.5])
ax.set_ylim([-0.5, 1.5])
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1", "class 2"], loc="lower right")
plt.show()
```



Now let's train and visualise a classification tree on this data.

```
model = DecisionTreeClassifier().fit(X, y)

plot_tree(model)
plt.show()
```



This is the barebones way to visualise the tree. We can do it in a snazzier manner using `graphviz`.

Execution note

Diagram output is not included in this edition. Rendering this tree requires the Python graphviz package and the Graphviz dot executable. The preceding matplotlib example provides an alternative tree visualisation.

```
dot_data = export_graphviz(
    model,
    out_file=None,
    feature_names=["x_1", "x_2"],
    class_names=["class 0", "class 1", "class 2"],
    filled=True,
    rounded=True,
    special_characters=True,
)
graph = graphviz.Source(dot_data)
display(graph)
```

Each box is a **node**. The training data comes into the top node (the root node). The `value` field tells us the distribution of class labels for the data coming into that node. It is $[10, 10, 10]$ for the root node as this is the distribution of the whole training set.

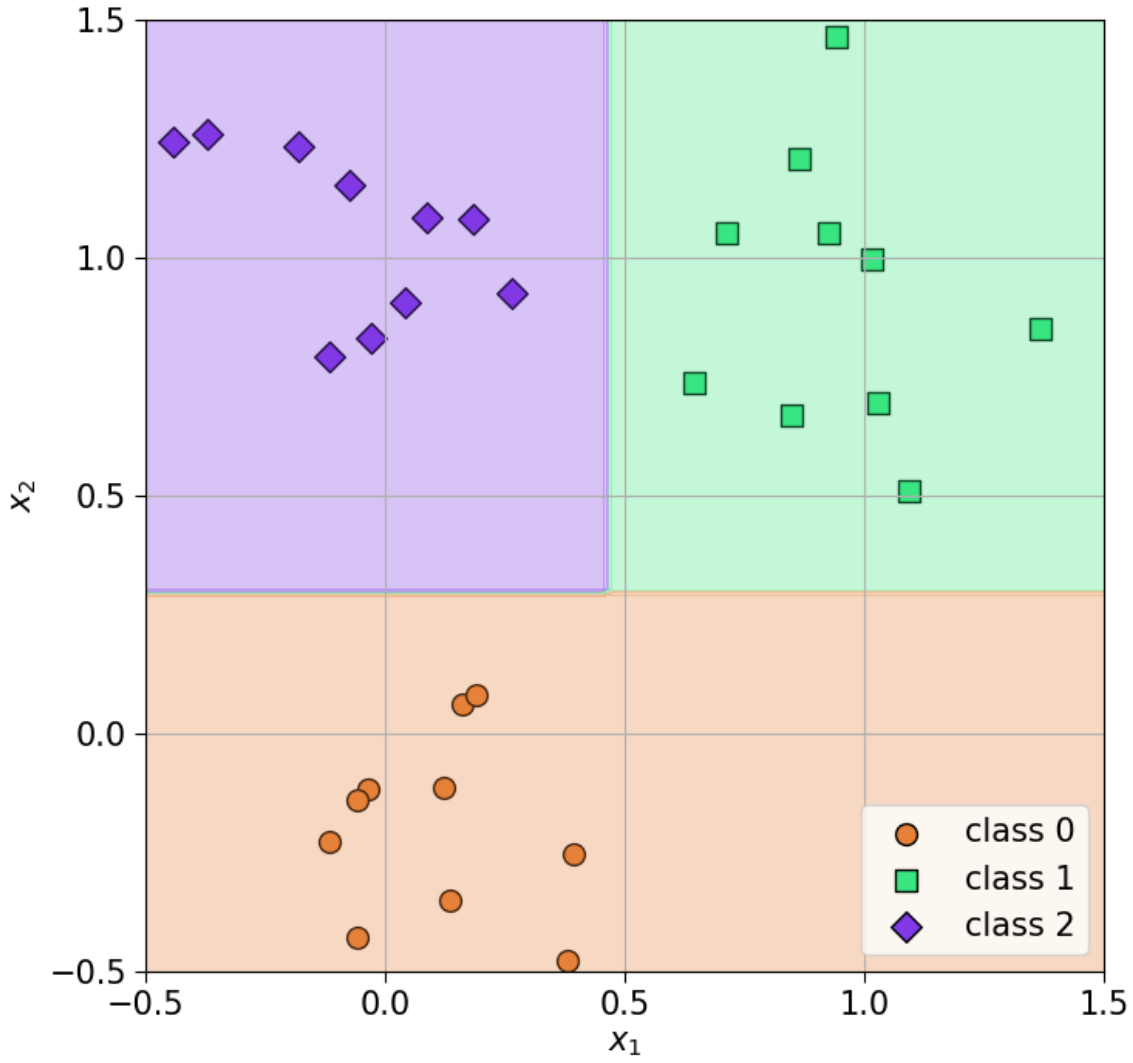
The training data is split according to some criterion. For the first node, this is whether x_2 i.e. the value of the second feature is less than or equal to 0.294. By looking at the two *child* nodes of this

root node, we can see that 10 examples, all of which are labelled as class 0, fall into the left child node and 20 examples (10 of which are class 1, 10 of which are class 2) fall into the right child node.

At test time a point proceeds down the tree and is then classified according to the majority class in the corresponding split of training data at that node.

We can look at the decision boundary for the tree to get a feel for what it does in feature space.

```
fig, ax = plt.subplots(figsize=[8, 8])

ax.scatter(X[y == 0, 0], X[y == 0, 1], color="#e58139", edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color="#39e581", marker="s",
 ↪  edgecolor="k")
ax.scatter(X[y == 2, 0], X[y == 2, 1], color="#8139e5", marker="D",
 ↪  edgecolor="k")
ax.grid()

ax.set_xticks([-0.5, 0, 0.5, 1, 1.5])
ax.set_yticks([-0.5, 0, 0.5, 1, 1.5])
ax.set_xlim([-0.5, 1.5])
ax.set_ylim([-0.5, 1.5])
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1", "class 2"], loc="lower right")

# Add the decision boundary
disp = DecisionBoundaryDisplay.from_estimator(
    model,
    X,
    response_method="predict",
    alpha=0.3,
    grid_resolution=200,
    ax=ax,
    cmap=ListedColormap(colours),
)
plt.show()
```



The decision boundary is non-linear, but consists of a bunch of horizontal and vertical lines! Each node separates data points according to some threshold for one particular feature, and corresponds to one of these lines.

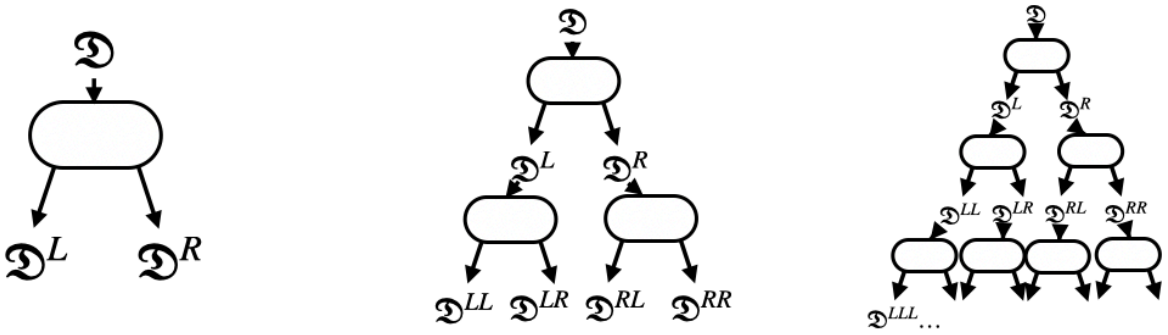
7.3 Classification tree learning - the CART algorithm

We have a training set $\mathfrak{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$ where $\mathbf{x} \in \mathbb{R}^D$ and $y \in \mathbb{Z}^{+}_{<K}$.

Classification tree learning consists of creating a node for splitting the dataset into two data splits, and then creating a node that splits each of these data splits in two, and so on.

This is done **greedily** in that we optimise for each split without giving any thought as to what will happen at later splits.

If we get to a node where the incoming data only has points from a single class, or we reach some kind of stopping criterion (e.g. we're only allowed to go to a certain depth) then the node is converted into a leaf node and no further splitting occurs.



Consider a subset of data (indexed by $\mathcal{N}_i$) arriving at some node i:

$$\mathfrak{D}_i = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n \in \mathcal{N}_i}$$

The node will split the subset into

$$\mathfrak{D}_i^L = \{(\mathbf{x}^{(n)}, y^{(n)}) \in \mathfrak{D}_i : x_j^{(n)} \leq t\}$$

and

$$\mathfrak{D}_i^R = \{(\mathbf{x}^{(n)}, y^{(n)}) \in \mathfrak{D}_i : x_j^{(n)} > t\}.$$

Remember we are being greedy. We don't care what happens later on, we just want to find the feature j and the threshold t that give us the best split here.

This is starting to sound like an optimisation problem! In fact, what we want to solve is

$$\underset{j,t}{\text{minimise}} \ \frac{|\mathfrak{D}_i^L(j,t)|}{|\mathfrak{D}_i|} c(D_i^L(j,t)) + \frac{|\mathfrak{D}_i^R(j,t)|}{|\mathfrak{D}_i|} c(D_i^R(j,t))$$

This looks intimidating, so let's break it down:

c is a function that takes a split and returns a scalar indicating how **bad** that split is. For instance, if $c(D_i^L(j,t))$ is low then our left split $D_i^L(j,t)$ is good. However, we care about how many data items are in the split. If there aren't many then we don't really care about how bad it is, but if there are lots then we really do. This is why the first term above is the (negative) quality of the left split $c(D_i^L(j,t))$ **weighted** by its relative size $\frac{|\mathfrak{D}_i^L(j,t)|}{|\mathfrak{D}_i|}$ to the right split.

The second term is the (negative) quality of the right split weighted by its relative size. We want both the left and right splits to be good, but we balance how much we care about the splits according to their size.

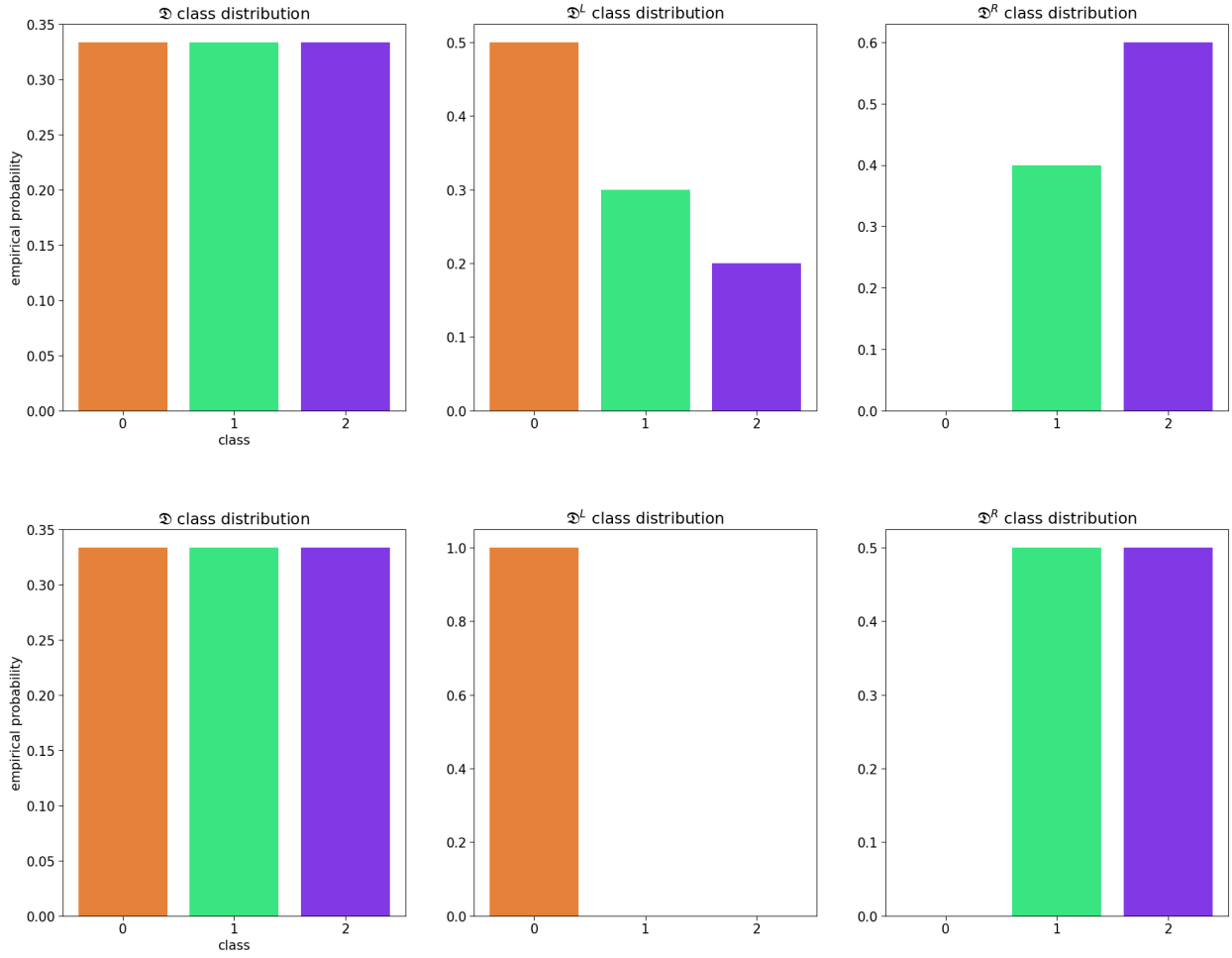
But what is a good split for classification? If a split was able to entirely isolate one class from the rest that would be great. In constrast, if a split wasn't able to separate classes out from each other that wouldn't be ideal.

We can quantify this by looking at the distribution of class labels after a split. If we normalise this distribution by the number of data points we get an empirical probability distribution (specifically, a categorical distribution).

An important point to be aware of, is that a test point is classified at a leaf node according to the highest probability in this distribution.

Let's consider our original training set, and look at some probability distributions for various splits.

```
def split(X, y, j, t):
    left  = X[:, j] <= t
    right = ~left

    X_L, y_L = X[left], y[left]
    X_R, y_R = X[right], y[right]

    return X_L, y_L, X_R, y_R


def plot_split_dists(X, y, j, t):

    _, y_L, _, y_R = split(X, y, j, t)

    fig, axes = plt.subplots(1, 3, figsize=[24, 8])

    axes[0].bar(
        [0, 1, 2],
        [np.sum(y == k) / len(y) for k in range(3)],
        color=["#e58139", "#39e581", "#8139e5"],
    )
    axes[0].set_xticks([0, 1, 2])
    axes[0].set_xlabel("class")
    axes[0].set_ylabel("empirical probability")
    axes[0].set_title(r"$\mathfrak{D}$ class distribution")

    axes[1].bar(
        [0, 1, 2],
        [np.sum(y_L == k) / len(y_L) for k in range(3)],
        color=["#e58139", "#39e581", "#8139e5"],
    )
    axes[1].set_xticks([0, 1, 2])
    axes[1].set_title(r"$\mathfrak{D}^L$ class distribution")

    axes[2].bar(
        [0, 1, 2],
        [np.sum(y_R == k) / len(y_R) for k in range(3)],
        color=["#e58139", "#39e581", "#8139e5"],
    )
    axes[2].set_xticks([0, 1, 2])
    axes[2].set_title(r"$\mathfrak{D}^R$ class distribution")


# Split at
plot_split_dists(X, y, 1, 1)
plot_split_dists(X, y, 1, 0.2)
```



The split corresponding to the bottom row is better as it completely isolates class 0 in the left split. The split correspond to the top row isn't great as the left split still has examples for all the classes.

With this in mind, we can look at the two most common choices for c when it comes to evaluating splits. The first is **entropy**. The entropy of a probability distribution basically tells us how uniform it is.

$$H = -\sum_k p_k \log_2 p_k$$

We assume that $0 \log_2 0 = 0$, should it arise.

The second is **Gini impurity**. This is the probability of incorrectly classifying a random point in a split according to the probability distribution of that split.

$$G = \sum_k p_k (1 - p_k)$$

These perform similarly, and Gini impurity is cheaper to compute so we will use that by default.

Let's edit our code above to compute the quantity we want to minimise for both the splits i.e. $\frac{|\mathfrak{D}_i^L(j,t)|}{|\mathfrak{D}_i|} c(D_i^L(j,t)) + \frac{|\mathfrak{D}_i^R(j,t)|}{|\mathfrak{D}_i|} c(D_i^R(j,t))$ where c is Gini Impurity.

```python
def gini(y):
    _, counts = np.unique(y, return_counts=True)
    p = counts / counts.sum()
    return p.T @ (1 - p)


def score_split(X, y, j, t):
    left = X[:, j] <= t
    right = ~left

    _, y_L = X[left], y[left]

    _, y_R = X[right], y[right]

    loss = (len(y_L) / len(y)) * gini(y_L) + (len(y_R) / len(y)) *
     ↪  gini(y_R)
    return loss


print(f"Loss for splitting with j=1 and t=1.0 is {score_split(X, y, 1,
 ↪  1):.5f}")
print(f"Loss for splitting with j=1 and t=0.2 is {score_split(X, y, 1,
 ↪  0.2):.5f}")
```

```
Loss for splitting with j=1 and t=1.0 is 0.57333
Loss for splitting with j=1 and t=0.2 is 0.33333
```

Clearly, the second split is better. Solving this optimisation problem can simply be done by considering each feature, and then using each of the unique values of that feature in the split as a possible threshold (but sklearn does all this for us).

7.4 Regression trees

We now have a training set $\mathfrak{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$ where $\mathbf{x} \in \mathbb{R}^D$ and $y \in \mathbb{R}$. y is now a target for regression instead of a class label.

Regression trees are a lot like classification trees except each node is associated with a scalar quantity (target) rather than a distribution over classes. Specifically, this quantity is the mean target of all the data points in the split $\bar{y}$.

Additionally, the c used in the objective for splitting is different. It is usually **mean squared error**. At each node we still solve

$$\underset{j,t}{\text{minimise}} \ \frac{|\mathfrak{D}_i^L(j,t)|}{|\mathfrak{D}_i|} c(D_i^L(j,t)) + \frac{|\mathfrak{D}_i^R(j,t)|}{|\mathfrak{D}_i|} c(D_i^R(j,t))$$

but with

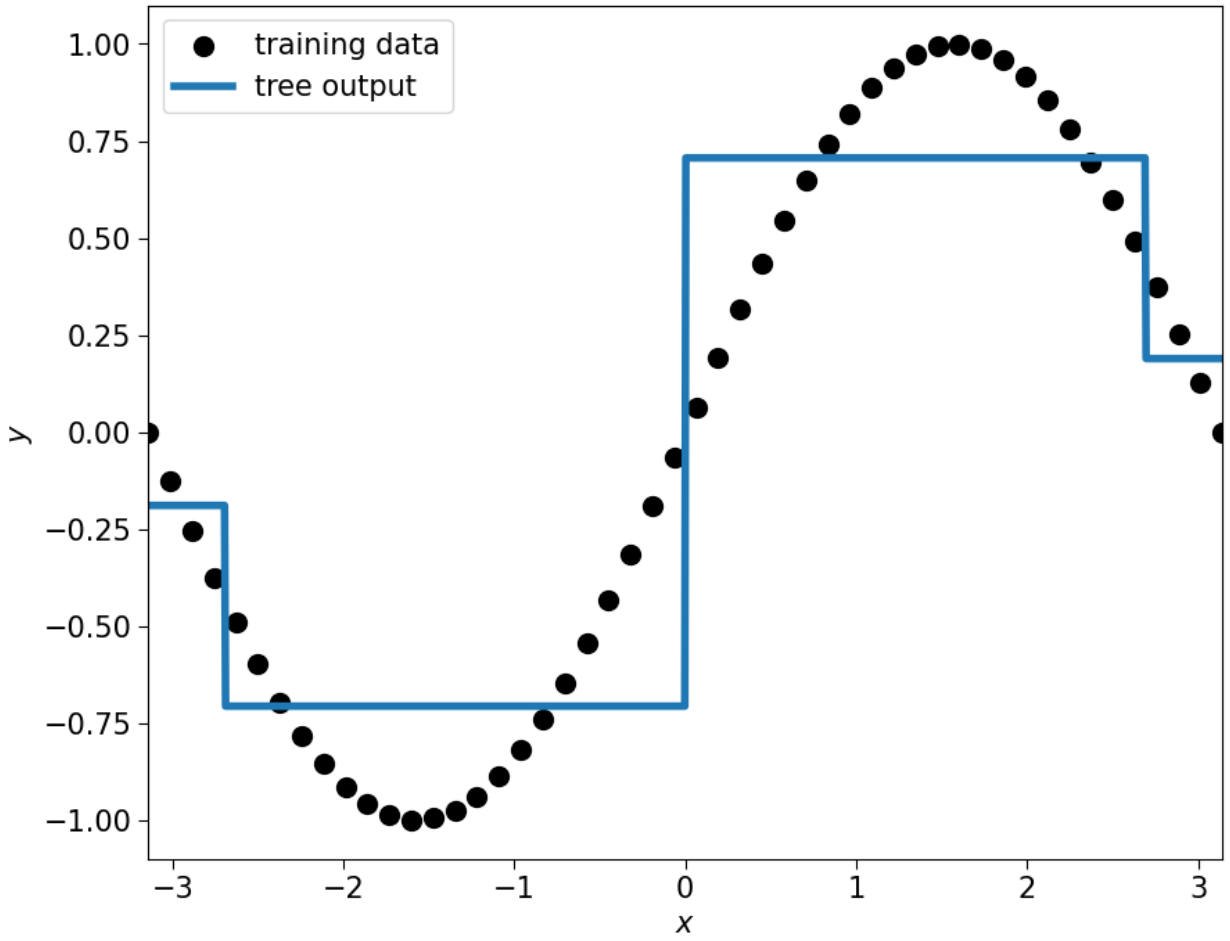
$$c(\mathfrak{D}_i) = \frac{1}{|\mathfrak{D}_i|} \sum_{n \in \mathfrak{D}_i} (y^{(n)} - \bar{y})^2$$

When a test point goes into the tree, we predict its target as being the value associated with the leaf node it ends up at.

Let's look at a simple 1D example. We will fit a regression tree (curtailed to maximum depth of 2 for simplicity), plot its predictions alongside the training data, and visualise the tree structure.

Bear in mind that as this is 1D, there is only 1 feature (so $j = 0$ for all splits).

```
fig, ax = plt.subplots(figsize=[10, 8])

x = np.linspace(-np.pi, np.pi, 50)
y = np.sin(x)
ax.set_xlim([-np.pi, np.pi])
ax.scatter(x, y, color="k")

X = x.reshape(-1, 1)
model = DecisionTreeRegressor(max_depth=2).fit(X, y)

x_dummy = np.linspace(-np.pi, np.pi, 1000)
y_hat= model.predict(x_dummy.reshape(-1, 1))

ax.plot(x_dummy, y_hat, linewidth=4)
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")

plt.legend(["training data", "tree output"])
plt.show()
```



Execution note

Diagram output is not included in this edition. Rendering this regression tree requires the Python graphviz package and the Graphviz dot executable.

```
dot_data = export_graphviz(
    model,
    out_file=None,
    feature_names=["x"],
    filled=True,
    rounded=True,
    special_characters=True,
)
graph = graphviz.Source(dot_data)
display(graph)
```

Make sure that you are happy that the four leaf nodes of this tree correspond to the four horizontal lines in the plot.

7.5 Ensembles

Decision trees are extremely susceptible to overfitting, and in practice aren't that useful on their own.

However, as part of an ensemble they can be very powerful.

Random forests and (ensembles of) gradient boosted decision trees both excel on tabular data. **We can have ensembles of other model types but trees tend to excel in ensembles!**

Ensembles are when we have multiple models $f_0(\mathbf{x}), f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots$ and we make predictions using some combination of their outputs. This is usually the mean:

$$f(\mathbf{x}) = \frac{1}{T}\sum_t f_t(\mathbf{x})$$

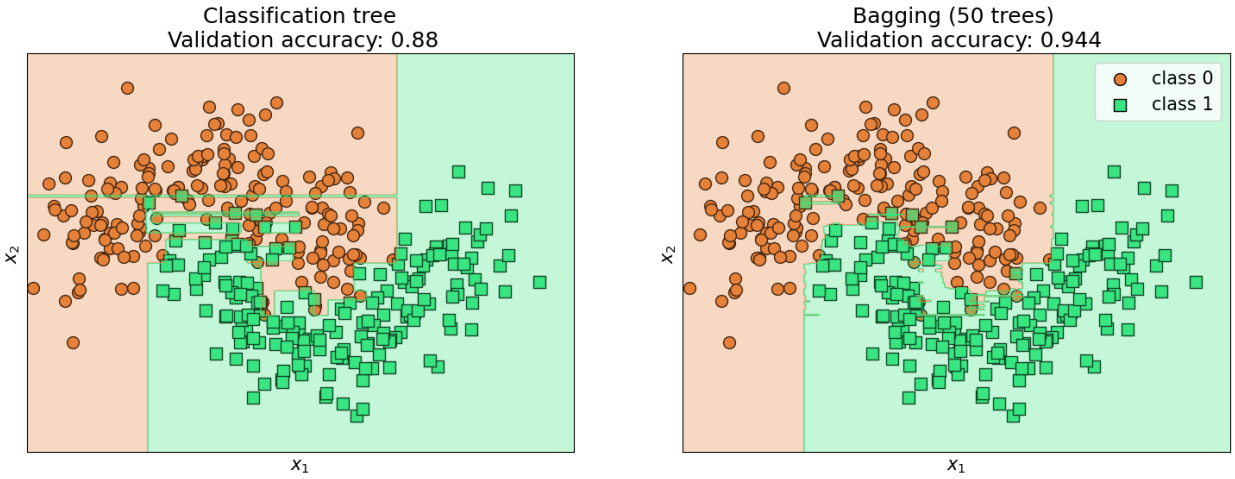
Intuitively, you can think of this as being the "wisdom of the crowd", or more informally, like asking the audience in "Who wants to be a millionaire?". However, tree learning (at least in how it's been presented here) is deterministic, so we would need to introduce some randomness.

7.6 Bagging

In boostrap aggregation, or bagging, the randomness comes from each tree in an ensemble being trained on different versions of the training data. Specifically, each tree is trained on a subset of the training data **sampled with replacement**.

Let's look at an example of a single tree versus bagging on a toy dataset.

```
X, y = make_moons(n_samples=500, noise=0.275, random_state=42)
X, X_val, y, y_val = train_test_split(X, y, random_state=42)

fig, axes = plt.subplots(1, 2, figsize=[18, 6])

for i in range(2):
    axes[i].scatter(X[y == 0, 0], X[y == 0, 1], color="#e58139",
     ↪  edgecolor="k")
    axes[i].scatter(
        X[y == 1, 0], X[y == 1, 1], color="#39e581", marker="s",
         ↪  edgecolor="k"
    )
    axes[i].set_xlabel("$x_1$")
    axes[i].set_ylabel("$x_2$")
    plt.legend(["class 0", "class 1"], loc="best")
    axes[i].set_xticks([])
    axes[i].set_yticks([])
    axes[i].set_xlim([-1.5, 3])
    axes[i].set_ylim([-1.5, 2])

colours = ["#e58139", "#39e581"]

# Traditional Tree
tree = DecisionTreeClassifier().fit(X, y)
axes[0].set_title(
    f"Classification tree\n Validation accuracy:
     ↪  {tree.score(X_val,y_val)}"
)
disp = DecisionBoundaryDisplay.from_estimator(
    tree,
    X,
    response_method="predict",
    alpha=0.3,
    grid_resolution=500,
    cmap=ListedColormap(colours),
    ax=axes[0],
)

# Bagging solution
bag = BaggingClassifier(n_estimators=50, random_state=63).fit(X, y)
axes[1].set_title(
    f"Bagging (50 trees) \n Validation accuracy:
     ↪  {bag.score(X_val,y_val)}"
)
disp = DecisionBoundaryDisplay.from_estimator(
    bag,
    X,
    response_method="predict",
    alpha=0.3,
    grid_resolution=500,
    cmap=ListedColormap(colours),
    ax=axes[1],
)

plt.show()
```



Notice that the bagging ensemble performs better on held-out validation data.

A **Random Forest** is a very popular ensemble model that is identical to a bagging ensemble of trees apart from one difference: at each node during tree learning (for every tree) only a subset of features are made available to find the optimal split. This prevents trees from relying on the same features for learning, and encourages variety.

7.7 Gradient Boosting

> Gradient boosting is conceptually quite tricky, but forms the basis of XGBoost which is an extremely popular algorithm famous for winning Kaggle competitions. I therefore felt it was worth including.

First, I'll describe the process of **boosting**. Assume we have a weak learner (read: a rubbish model) $F_0(\mathbf{x})$. This model isn't good and makes lots of mistakes.

Now let's introduce another weak learner $F_1(\mathbf{x})$ that tries to fix the mistakes of $F_0(\mathbf{x})$. These can be used in an ensemble $f_1(\mathbf{x}) = F_0(\mathbf{x}) + F_1(\mathbf{x})$.

Now let's introduce yet another weak learner $F_2(\mathbf{x})$ that tries to fix ths mistakes made by this ensemble and so on.

Gradient boosting is basically gradient descent in function space, and adding a weak learner corresponds to taking a negative gradient step.

"What?" would be the correct response to the last statement, so let's take it very slowly.

Let's briefly recap gradient descent with a parametric model $f_{\boldsymbol{\theta}}(\mathbf{x})$ where $\boldsymbol{\theta}$ is a vector of the model's parameters (for a linear model this would just be a concatenation of the weight vector and a bias). Our goal is to find the parameters that minimise some loss function L_{avg} (note the unusual subscript) which is an average of individual losses $L^{(n)}$ across training examples for those parameters.

$$L_{avg} = \frac{1}{N}\sum_n L(y^{(n)}, \mathbf{x}^{(n)}, \boldsymbol{\theta}) = \frac{1}{N}\sum_n L^{(n)}$$

e.g. for regression, we typically use a mean squared error loss per point $L = (y - f_{\boldsymbol{\theta}}(\mathbf{x}))^2$.

We first initialise the parameters at random; we will denote these initial parameters as $\boldsymbol{\theta}_0$. We then compute the gradient $\nabla_{\boldsymbol{\theta}} L_{avg}(\boldsymbol{\theta}_0)$ and take a step in the opposite direction in weight space:

$$\boldsymbol{\theta}_1 = \boldsymbol{\theta}_0 - \alpha \nabla_{\boldsymbol{\theta}} L_{avg}(\boldsymbol{\theta}_0)$$

This is done iteratively

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} L_{avg}(\boldsymbol{\theta}_t)$$

Now on to gradient boosting. We will start by representing our model as a vector according to its value at the training points.

$$\mathbf{f}_t = \begin{bmatrix} f_t(\mathbf{x}^{(1)}) & f_t(\mathbf{x}^{(2)}) & \dots & f_t(\mathbf{x}^{(N)}) \end{bmatrix}^\top$$

This isn't as wacky as it first sounds. If we had points everywhere, then we'd have a very (infinitely!) long vector that perfectly describes the function (model).

Notice the subscript t: this corresponds to a *time-step* and we will be updating the function in a similar way to how we update the weights in gradient descent.

We can compute the loss for each of our training points (this is a function of our model and class labels) and store these in a vector too:

$$\mathbf{L}(\mathbf{f}_t) = \begin{bmatrix} L(y^{(1)}, f_t(\mathbf{x}^{(1)})) & L(y^{(2)}, f_t(\mathbf{x}^{(2)})) & \dots & L(y^{(N)}, f_t(\mathbf{x}^{(N)})) \end{bmatrix}^\top$$

Our loss function L can (usually) be differentiated with respect to $f(\mathbf{x})$. For instance, if $L = (y - f_{\boldsymbol{\theta}}(\mathbf{x}))^2$ then $\nabla_{f(\mathbf{x})} L = 2(f_{\boldsymbol{\theta}}(\mathbf{x}) - y)$. We can therefore create a vector of **gradients** with respect to $f(\mathbf{x})$.

$$\mathbf{g}_t = \begin{bmatrix} \nabla_{f_t} L(y^{(1)}, f_t(\mathbf{x}^{(1)})) & \nabla_{f_t} L(y^{(2)}, f_t(\mathbf{x}^{(2)})) & \dots & \nabla_{f_t} L(y^{(N)}, f_t(\mathbf{x}^{(N)})) \end{bmatrix}^\top$$

or concisely,

$$\mathbf{g}_t = \begin{bmatrix} g_t^{(1)} & g_t^{(2)} & \dots & g_t^{(N)} \end{bmatrix}^\top$$

Gradient descent in function space is then iteratively performing

$$\mathbf{f}_{t+1} = \mathbf{f}_t - \alpha \mathbf{g}_t$$

If you followed all that, great! If not, reread it and write it out until you do!

Gradient tree boosting looks like the above where

1. The function at time step t is an ensemble of regression trees $f_t(\mathbf{x}) = \sum_{m=0}^{t} F_m(\mathbf{x})$ (even if the problem is classification)
2. The gradient update step is approximated by fitting a regression tree F_{t+1} to $\{(\mathbf{x}^{(n)}, -g_t^{(n)})\}_{n=1}^N$ and adding a scaled copy to the ensemble: $f_{t+1}(\mathbf{x}) = f_t(\mathbf{x}) + \alpha F_{t+1}(\mathbf{x})$. The learning rate α controls the size of this update; the sum above corresponds to $\alpha = 1$.

Gradient boosted decision trees can be used straightforwardly in sklearn, but you will implement them by hand in the lab!

```
gb = GradientBoostingClassifier().fit(X, y)
gb.score(X_val, y_val)
```

```
0.928
```

8 Gaussian processes

```
# RUN THIS CELL FIRST

# Imports
import matplotlib.pyplot as plt
import numpy as np

# sklearn
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import ConstantKernel, RBF

# Better plots
plt.rcParams.update(
    {
        "lines.markersize": 10,   # Big points
        "font.size": 15,   # Larger font
        "savefig.dpi": 300,   # Higher res output
        "savefig.format": "pdf",   # PDF outputs
        "savefig.bbox": "tight",   # remove whitespace around figure
        "savefig.transparent": True,   # transparent background
        "xtick.major.size": 5.0,   # Bigger xticks
        "ytick.major.size": 5.0,   # Bigger yticks
    }
)
```

8.1 Continuous random variables and the Gaussian distribution

Consider some real-valued quantity $X \in \mathbb{R}$ that we care about, but when we try to measure it there is noise and we get different values each time. We have an idea about what it *should be* (e.g. the mean of our measurements) but we want to be able to express our degree of uncertainty (e.g. through the standard deviation of our measurements).

We can achieve this by treating X as a continous random variable, and represent it by a probability density function $p(x)$, evaluated at a possible value x. A density is not the probability of one exact value: for a continuous variable, $\mathrm{P}(X = x) = 0$.

> Here X is a random variable, and x is a realisation of that random variable. It is fairly likely I will be a bit sloppy with my notation with respect to these (while protesting that I am no mathematician). The main purpose of the maths you see is to aid understanding, so if it achieves that then I will be happy enough.

Integrating $p(x)$ between two points gives the probability that a measurement of X will lie between those values:

$$\mathrm{P}(a < X \leq b) = \int_a^b p(x)\, dx$$

It follows that $\int_{-\infty}^{\infty} p(x)\,dx = 1$.

The mean and variance of X can be computed from this pdf:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} xp(x)dx = \mu$$

$$\mathbb{E}[(X-\mu)^2] = \mathbb{V}[X] = \int_{-\infty}^{\infty} (x-\mu)^2 p(x)dx = \sigma^2$$

It is very common to default to a Gaussian pdf for a continuous random variable

$$p(x) = \mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

We can write this more succinctly as $X \sim \mathcal{N}(\mu, \sigma^2)$.

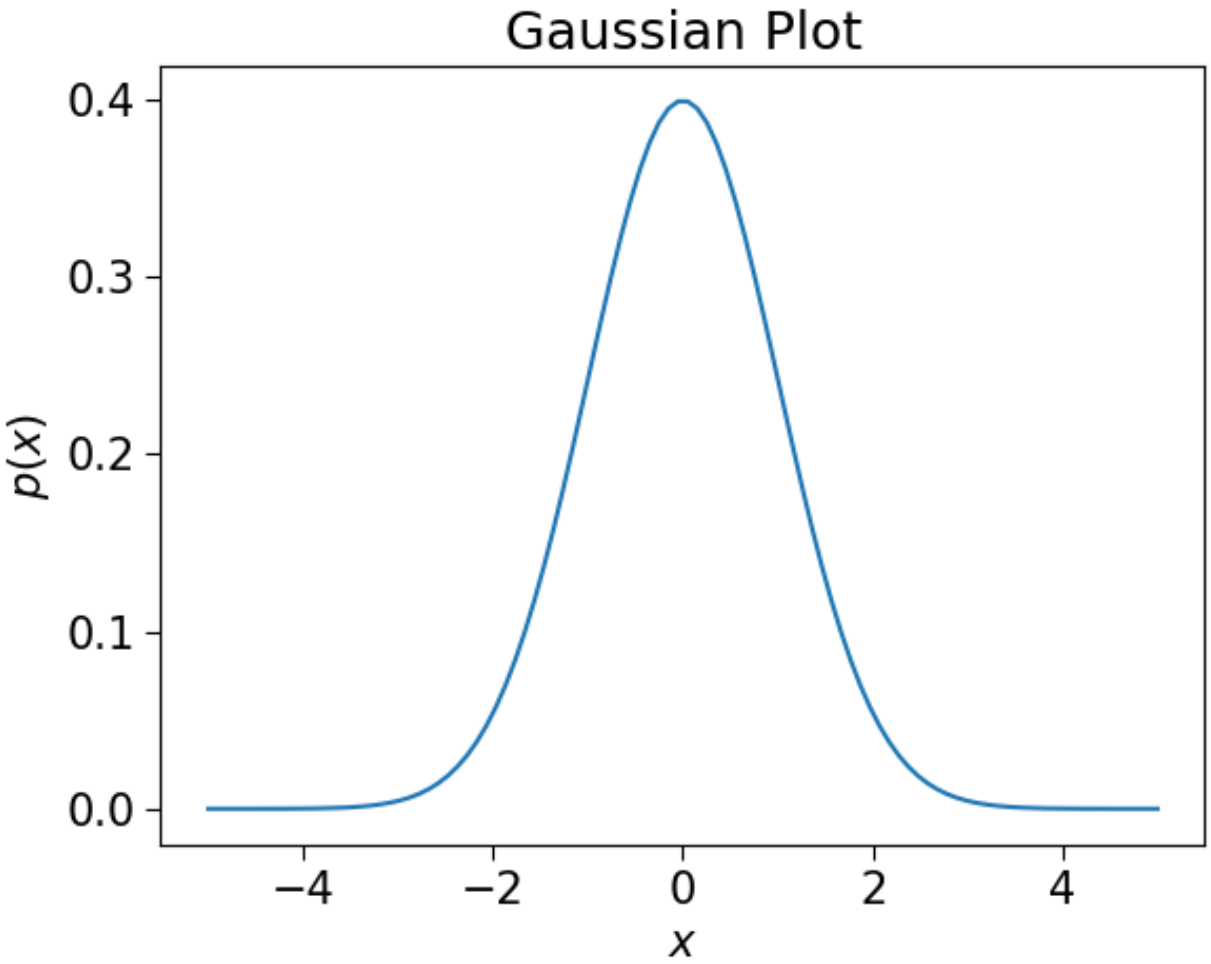
There are several reasons why Gaussians are commonly used:

- They are easy to interpret, being defined entirely by their mean (the peak) and their variance (the spread)
- They are mathematically convienient, being closed under marginalisation and conditioning (see later)
- Under suitable conditions (such as independent, identically distributed variables with finite variance), the central limit theorem says that a sum, after centring and scaling, approaches a Gaussian distribution

While we are not going to go into the details, it's important to appreciate that you can **sample** from a pdf.

The code below plots a Gaussian pdf. You can change μ and σ to see how it affects the plot.

```
def gauss(x, mu, sigma):
    return (
        1
        / (np.sqrt(2 * np.pi * sigma**2))
        * np.exp(-((1) / (2 * sigma**2)) * (x - mu) ** 2)
    )


x = np.linspace(-5, 5, 100)
sigma = 1.0
mu = 0.0

px = gauss(x, mu, sigma)

fig, ax = plt.subplots()
ax.plot(x, px)
ax.set_xlabel("$x$")
ax.set_ylabel("$p(x)$")
ax.set_title("Gaussian Plot")

plt.show()
```



8.2 Joint pdfs and the Bivariate Gaussian

Often we have random variables that are related to each other. If we have two such random variables X_1 and X_2, we can consider their joint pdf $p(x_1, x_2)$. From this, we can recover the individual distributions through **marginalisation**.

$$p(x_1) = \int_{-\infty}^{\infty} p(x_1, x_2)\, dx_2$$

$$p(x_2) = \int_{-\infty}^{\infty} p(x_1, x_2)\, dx_1$$

We can also do **conditioning**. For instance, if we find out the exact value of X_1 is x_1, we can look at the conditional distribution for X_2 given this information $p(x_2|X_1 = x_1)$. For brevity, I will write this as $p(x_2|x_1)$. This can be obtained from Bayes' rule:

$$p(x_2|x_1) = \frac{p(x_1, x_2)}{p(x_1)} = \frac{p(x_1, x_2)}{\int_{-\infty}^{\infty} p(x_1, x_2)\, dx_2}$$

This looks different to how Bayes' rule is usually represented, but recall that the numerator $p(x_1, x_2) = p(x_1|x_2)p(x_2)$.

The most common joint pdf (and the one we will default to) is the **Multivariate Gaussian**. For some vector of D random variables $\mathbf{x} = \left[X_1, X_2, \dots, X_D\right]^\top$, this is given by

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\Big[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}) \Big]$$

In the case of two random variables ($\mathbf{x} = \left[X_1, X_2\right]^\top$) this is referred to as a Bivariate Gaussian and

$$p(x_1, x_2) = \mathcal{N}(x_1, x_2; \boldsymbol{\mu}, \boldsymbol{\Sigma}) \quad \text{where} \quad \boldsymbol{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} \quad \text{and} \quad \boldsymbol{\Sigma} = \begin{bmatrix} \Sigma_{1,1} & \Sigma_{1,2} \\ \Sigma_{2,1} & \Sigma_{2,2} \end{bmatrix}$$

We can also write this as

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \sim \mathcal{N}\Big(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{1,1} & \Sigma_{1,2} \\ \Sigma_{2,1} & \Sigma_{2,2} \end{bmatrix} \Big)$$

Here, the covariance $\Sigma_{i,j} = \mathbb{E}\Big[(X_i - \mathbb{E}[X_i])(X_j - \mathbb{E}[X_j]) \Big]$. I am using this to represent both variances ($i = j$) and covariances ($i \neq j$).

> Writing the Bivariate Gaussian distribution without vectors and matrices is painful, which is why I started with the definition of a Multivariate Gaussian :)

I mentioned earlier that an advantage of working with Gaussians is that they are closed under marginalisation and conditioning. What this means is that the marginal distributions and conditional distributions are also Gaussian. This is mathematically very convenient.

To marginalise a Bivariate Gaussian, we just pick out the corresponding mean and variance:

$$p(x_1) = \mathcal{N}(x_1; \mu_1, \Sigma_{1,1})$$

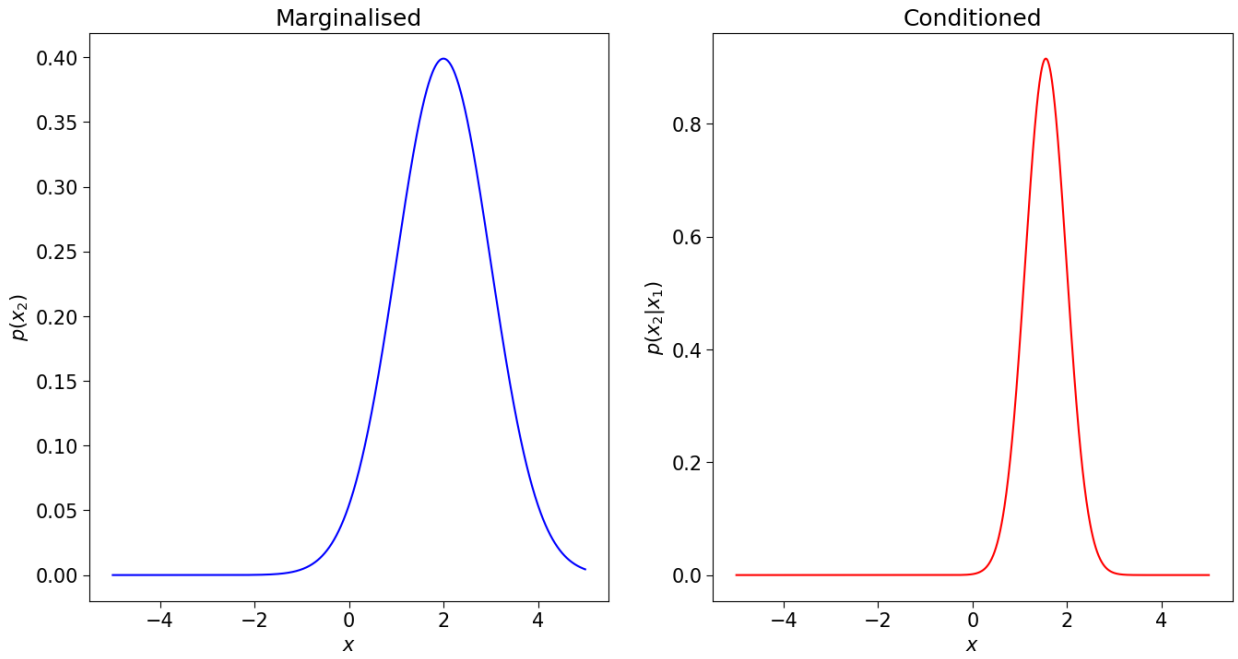
$$p(x_2) = \mathcal{N}(x_2; \mu_2, \Sigma_{2,2})$$

The conditional distribution $p(x_2|x_1)$ (given without proof) is:

$$p(x_2|x_1) = \mathcal{N}(x_2; \mu_2 + \frac{\Sigma_{1,2}}{\Sigma_{1,1}}(x_1 - \mu_1), \Sigma_{2,2} - \frac{\Sigma_{1,2}^2}{\Sigma_{1,1}})$$

Let's consider an example:

```
mus    = np.array([0,2])
sigmas = np.array(
    [
        [1,0.9],
        [0.9,1]
    ],
)

# We marginalise our bivariate
x    = np.linspace(-5, 5, 1000)
p_x2 = gauss(x=x, mu=mus[1],sigma=np.sqrt(sigmas[1,1]))

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=[16,8])
axes[0].plot(x, p_x2, c='b')
axes[0].set_title("Marginalised")
axes[0].set_xlabel("$x$")
axes[0].set_ylabel("$p(x_{2})$")


# We then find out that x1 is definitely -0.5
x1 = -0.5

# With this, we condition our bivariate
mu           = mus[1] + ((sigmas[0,1])/(sigmas[0,0]))*(x1 - mus[0])
sigma        = np.sqrt(sigmas[1,1] - (sigmas[0,1])**2/(sigmas[0,0]))
p_x2_with_x1 = gauss(x=x, mu=mu, sigma=sigma)

axes[1].plot(x,p_x2_with_x1, c='r')
axes[1].set_title("Conditioned")
axes[1].set_xlabel("$x$")
axes[1].set_ylabel("$p(x_{2}|x_{1})$")

plt.show()
```



Notice that both the marginalised *and* conditioned plots are themselves Gaussian functions.

8.3 The Multivariate Gaussian

Let's now consider the more general multivariate case. If we have a vector of random variables $\mathbf{x} = \begin{bmatrix} X_1, X_2, \dots, X_D \end{bmatrix}^\top$ that are jointly Gaussian then

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\Big[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}) \Big]$$

where $\boldsymbol{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_D \end{bmatrix}$ and $\boldsymbol{\Sigma} = \begin{bmatrix} \Sigma_{1,1} & \Sigma_{1,2} & \dots & \Sigma_{1,D} \\ \Sigma_{2,1} & \Sigma_{2,2} & \dots & \Sigma_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_{D,1} & \Sigma_{D,2} & \dots & \Sigma_{D,D} \end{bmatrix}$.

Marginalising is very straightforward. As with the Bivariate case, we pick out the corresponding mean and variance for a variable.

$$p(x_i) = \mathcal{N}(x_i; \mu_i, \Sigma_{i,i})$$

Conditioning is slightly more involved because we can condition the distribution for some random vector on another and both can be a variety of sizes. To write a general expression for conditioning, we partition $\mathbf{x}$ as $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1, \mathbf{x}_2 \end{bmatrix}^\top$ where $\mathbf{x}_1 \in \mathbb{R}^{D_1}$ and $\mathbf{x}_2 \in \mathbb{R}^{D_2}$ (therefore $D_1 + D_2$ must equal D). After making this partition, we can write

$$\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} \sim \mathcal{N}(\begin{bmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{1,1} & \boldsymbol{\Sigma}_{1,2} \\ \boldsymbol{\Sigma}_{2,1} & \boldsymbol{\Sigma}_{2,2} \end{bmatrix})$$

All we've done here is split the mean vector in two, and the covariance matrix into four.

Imagine that $\mathbf{x} \in \mathbb{R}^5$ and we had $D_1 = 2$ and $D_2 = 3$.

- $\boldsymbol{\mu}_1$ would be the top two elements of the mean vector, and $\boldsymbol{\mu}_2$ would be the bottom three elements.
- $\boldsymbol{\Sigma}_{1,1}$ would be the top-left 2×2 submatrix of $\boldsymbol{\Sigma}$.
- $\boldsymbol{\Sigma}_{1,2}$ would be the top-right 2×3 submatrix.
- $\boldsymbol{\Sigma}_{2,1}$ would be the bottom-left 3×2 submatrix.
- $\boldsymbol{\Sigma}_{2,2}$ would be the bottom-right 3×3 submatrix.

If we find out the exact values for the elements in $\mathbf{x}_1$ then we can compute the condition distribution $p(\mathbf{x}_2|\mathbf{x}_1)$. Without proof, this is given by:

$$p(\mathbf{x}_2|\mathbf{x}_1) = \mathcal{N}(\mathbf{x}_2; \boldsymbol{\mu}_{2|1}, \boldsymbol{\Sigma}_{2|1})$$

where

$$\boldsymbol{\mu}_{2|1} = \boldsymbol{\mu}_2 + \boldsymbol{\Sigma}_{2,1}\boldsymbol{\Sigma}_{1,1}^{-1}(\mathbf{x}_1 - \boldsymbol{\mu}_1)$$

$$\boldsymbol{\Sigma}_{2|1} = \boldsymbol{\Sigma}_{2,2} - \boldsymbol{\Sigma}_{2,1}\boldsymbol{\Sigma}_{1,1}^{-1}\boldsymbol{\Sigma}_{1,2}$$

We now have everything we need to look at Gaussian processes. They are basically just a great big Multivariate Gaussian!

8.4 A Gaussian Process model

For regression problems, we try and find a model $f(\mathbf{x})$ that best maps data points $\mathbf{x} \in \mathbb{R}^D$ to targets $y \in \mathbb{R}$.

Implicitly, $f(\mathbf{x})$ is our model of some underlying, unknown function. We are going to introduce Gaussian processes in the context of modelling an unknown function and tie it to regression later in.

We will denote a Gaussian process model using

$$f(\mathbf{x}) \sim GP(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

It is defined entirely in terms of its mean function $m(\mathbf{x})$ and kernel function $k(\mathbf{x}, \mathbf{x}')$, which we will return to shortly.

For **any** non-empty finite set of input points $\mathbf{X} = \left[\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(M)}\right]^\top$, the function values $\mathbf{f} = \left[f(\mathbf{x}^{(1)}), f(\mathbf{x}^{(2)}), \dots, f(\mathbf{x}^{(M)})\right]^\top$ are treated as **random variables** that have a **Multivariate Gaussian distribution**

$$\mathbf{f} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

$\boldsymbol{\mu}$ is determined by the mean function $\boldsymbol{\mu} = [m(\mathbf{x}^{(1)}), m(\mathbf{x}^{(2)}), \dots, m(\mathbf{x}^{(M)})]^\top$ and $\boldsymbol{\Sigma}$ is determined by the kernel function in that $\boldsymbol{\Sigma}_{i,j} = k(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$.

In full, we can write

$$\begin{bmatrix} f(\mathbf{x}^{(1)}) \\ f(\mathbf{x}^{(2)}) \\ \vdots \\ f(\mathbf{x}^{(M)}) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(\mathbf{x}^{(1)}) \\ m(\mathbf{x}^{(2)}) \\ \vdots \\ m(\mathbf{x}^{(M)}) \end{bmatrix}, \begin{bmatrix} k(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & k(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) & \dots & k(\mathbf{x}^{(1)}, \mathbf{x}^{(M)}) \\ k(\mathbf{x}^{(2)}, \mathbf{x}^{(1)}) & k(\mathbf{x}^{(2)}, \mathbf{x}^{(2)}) & \dots & k(\mathbf{x}^{(2)}, \mathbf{x}^{(M)}) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}^{(M)}, \mathbf{x}^{(1)}) & k(\mathbf{x}^{(M)}, \mathbf{x}^{(2)}) & \dots & k(\mathbf{x}^{(M)}, \mathbf{x}^{(M)}) \end{bmatrix} \right)$$

Please stare at this equation until you understand it!

Typically, we use a zero mean function ($m(\mathbf{x}) = 0$) and we will assume that this is always the case on this course. The generalisation properties of the GP are largely determined by its kernel function.

Given two input points $\mathbf{x}$ and $\mathbf{x}'$, the kernel tells us what the covariance between their function values $f(\mathbf{x})$ and $f(\mathbf{x}')$ is. Recall that for a GP model, all function values are treated as random variables.

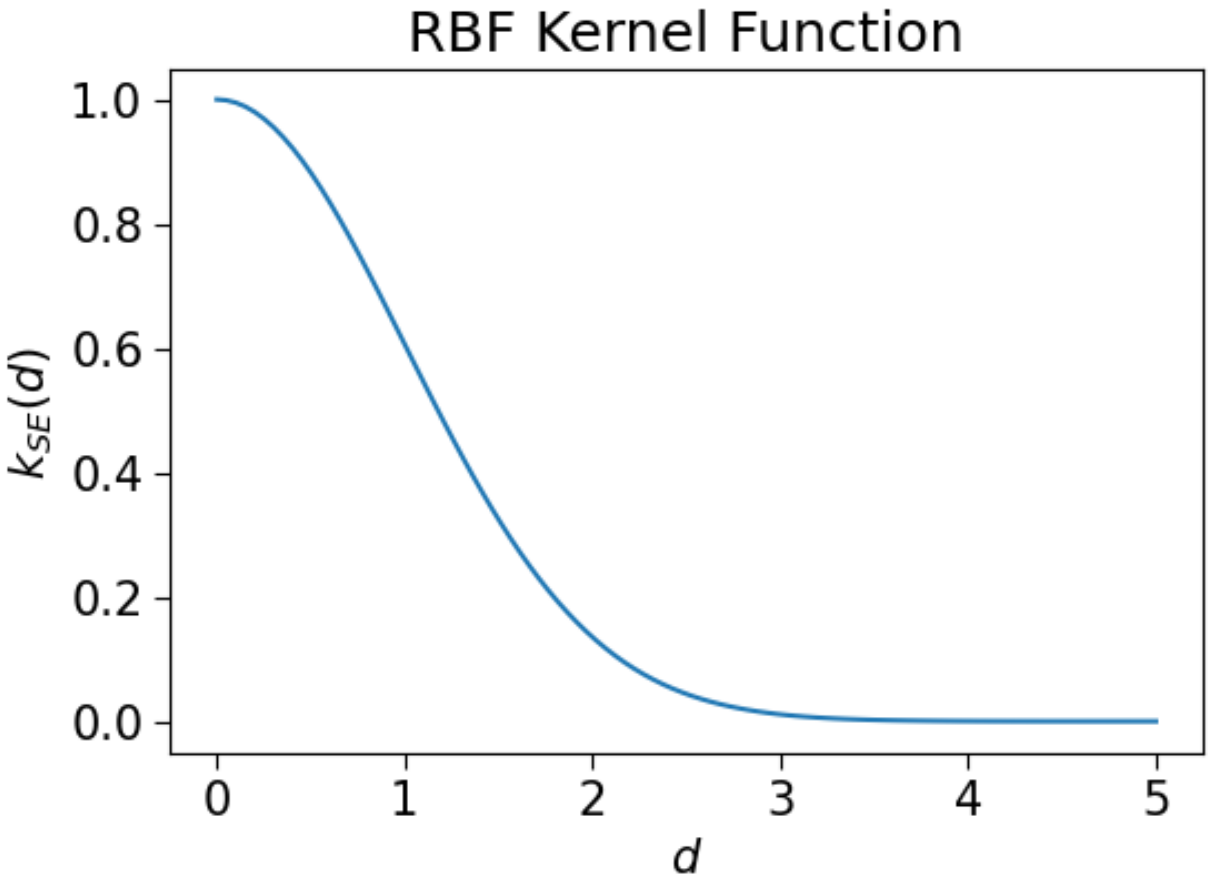
The default kernel (in sklearn and in general) is the radial basis function (RBF) (or squared exponential) kernel:

$$k_{SE}(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right)$$

Here, ℓ is a hyperparameter known as the lengthscale, which we will set to 1 for now.

Let's plot this kernel as a function of $d = \|\mathbf{x} - \mathbf{x}'\|$:

```
d = np.linspace(0, 5, 100)
k = np.exp(-(d**2) / 2)

fig, ax = plt.subplots(figsize=[6, 4])
ax.plot(d, k)
ax.set_xlabel("$d$")
ax.set_ylabel("$k_{SE}(d)$")
ax.set_title("RBF Kernel Function")

plt.show()
```



We can see that when the distance between two input points is low, the covariance (and therefore correlation) between their function values is high, and that when the distance between two input points is high, the correlation between their function values is low.

This is quite intuitive if you think about house prices (an example I heard in an online lecture years ago). Imagine that you are modelling the relationship between the location of a house (longitude, latitude) and its price. Houses that are close together will have correlated house prices, whereas houses far away from each other will not.

8.5 Gaussian process prior

A GP that has not been *fit* to training data is referred to as a GP prior. By providing a mean and kernel function we have baked in our apriori assumptions about what the function we are trying

to model should look like.

We are going to consider the 1D input case $x \in \mathbb{R}$ where we have a GP prior

$$f(x) \sim GP(0, \exp\left(-\frac{\|x - x'\|^2}{2}\right))$$

Note that $\boldsymbol{\mu} = \mathbf{0}$ because we are assuming that $\forall x, m(x) = 0$

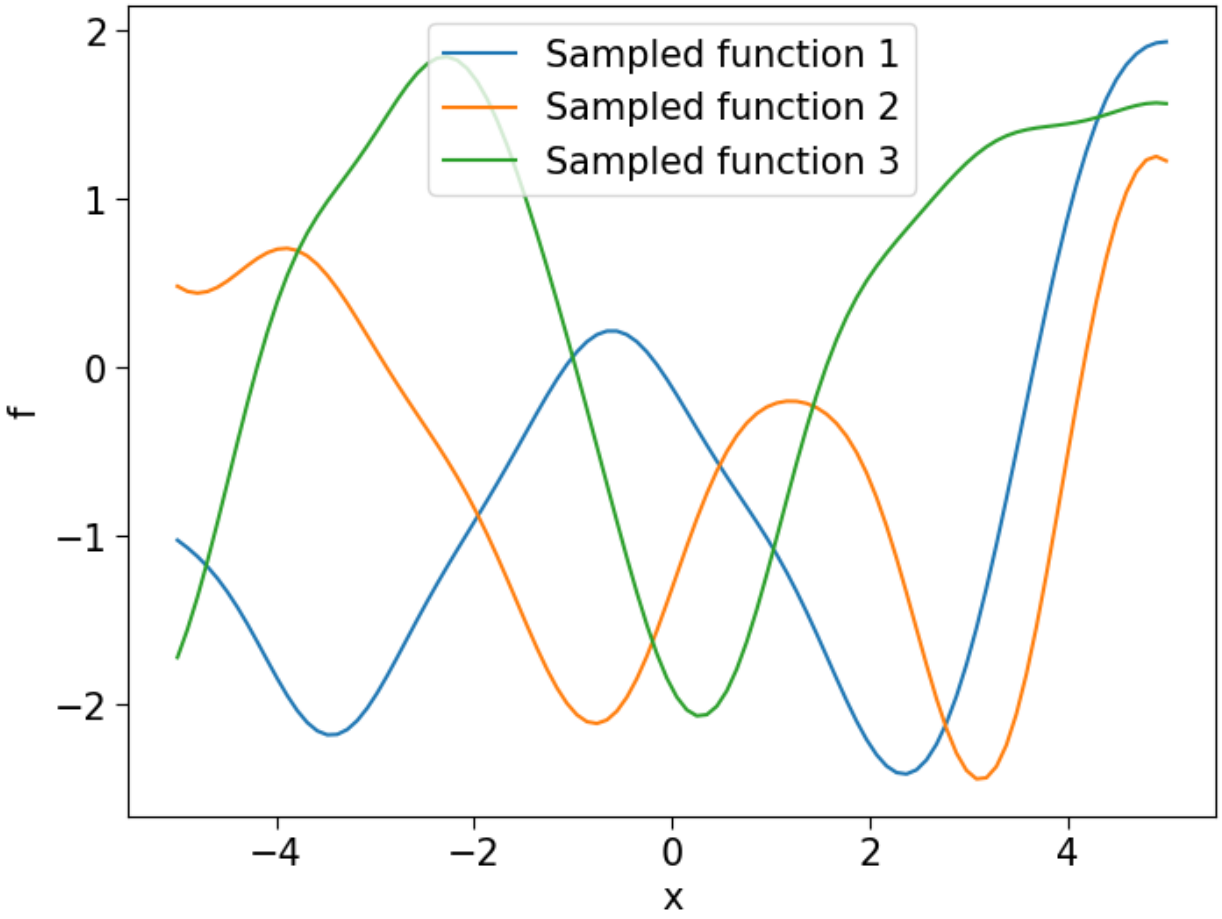
We can sample possible functions from the prior by creating some closely-spaced dummy points $\mathbf{X}_* = \left[x_*^{(1)}, x_*^{(2)}, \dots, x_*^{(M)}\right]^\top$ and then sampling realisations of $\mathbf{f}_* = \left[f(x_*{}^{(1)}), f(x_*{}^{(2)}), \dots, f(x_*{}^{(M)})\right]^\top$ from

$$\begin{bmatrix} f(x^{(1)}) \\ f(x^{(2)}) \\ \vdots \\ f(x^{(M)}) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} k(x_*^{(1)}, x_*^{(1)}) & k(x_*^{(1)}, x_*^{(2)}) & \dots & k(x_*^{(1)}, x_*^{(M)}) \\ k(x_*^{(2)}, x_*^{(1)}) & k(x_*^{(2)}, x_*^{(2)}) & \dots & k(x_*^{(2)}, x_*^{(M)}) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_*^{(M)}, x_*^{(1)}) & k(x_*^{(M)}, x_*^{(2)}) & \dots & k(x_*^{(M)}, x_*^{(M)}) \end{bmatrix} \right)$$

We can write the above concisely as $\mathbf{f}_* \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_{\mathbf{X}_*, \mathbf{X}_*})$.

Let's do this in sklearn:

```
# Create a Gaussian process that will default to zero mean and RBF
 ↪  kernel
# Ignore alpha for now
gp = GaussianProcessRegressor(alpha=0)

# Create an array of dummy points and reshape into a format sklearn
 ↪  will be happy with
X_star = np.linspace(-5, 5, 100).reshape(-1, 1)

# Sample 3 realisations of f_star (3 possible functions) from the GP
 ↪  prior with different seeds
f_star_0 = gp.sample_y(X_star, random_state=0)
f_star_1 = gp.sample_y(X_star, random_state=1)
f_star_2 = gp.sample_y(X_star, random_state=2)

# Plot these
fig, ax = plt.subplots(figsize=[8, 6])
ax.plot(X_star, f_star_0)
ax.plot(X_star, f_star_1)
ax.plot(X_star, f_star_2)

ax.set_xlabel("x")
ax.set_ylabel("f")
plt.legend(["Sampled function 1", "Sampled function 2", "Sampled
 ↪  function 3"])

plt.show()
```



The functions are wiggly with peaks and troughs. These are regions where points are close together and have function values that are highly correlated, as specified by our kernel.

We can also look at the individual distribution of function values at each of these dummy points. This can be achieved by marginalising the distribution above to get $f(x^{(1)}) \sim \mathcal{N}(0, k(x_*^{(1)}, x_*^{(1)}))$, $f(x^{(2)}) \sim \mathcal{N}(0, k(x_*^{(2)}, x_*^{(2)}))$ etc. Note that for every one of these marginal distributions, the output of the kernel is 1.

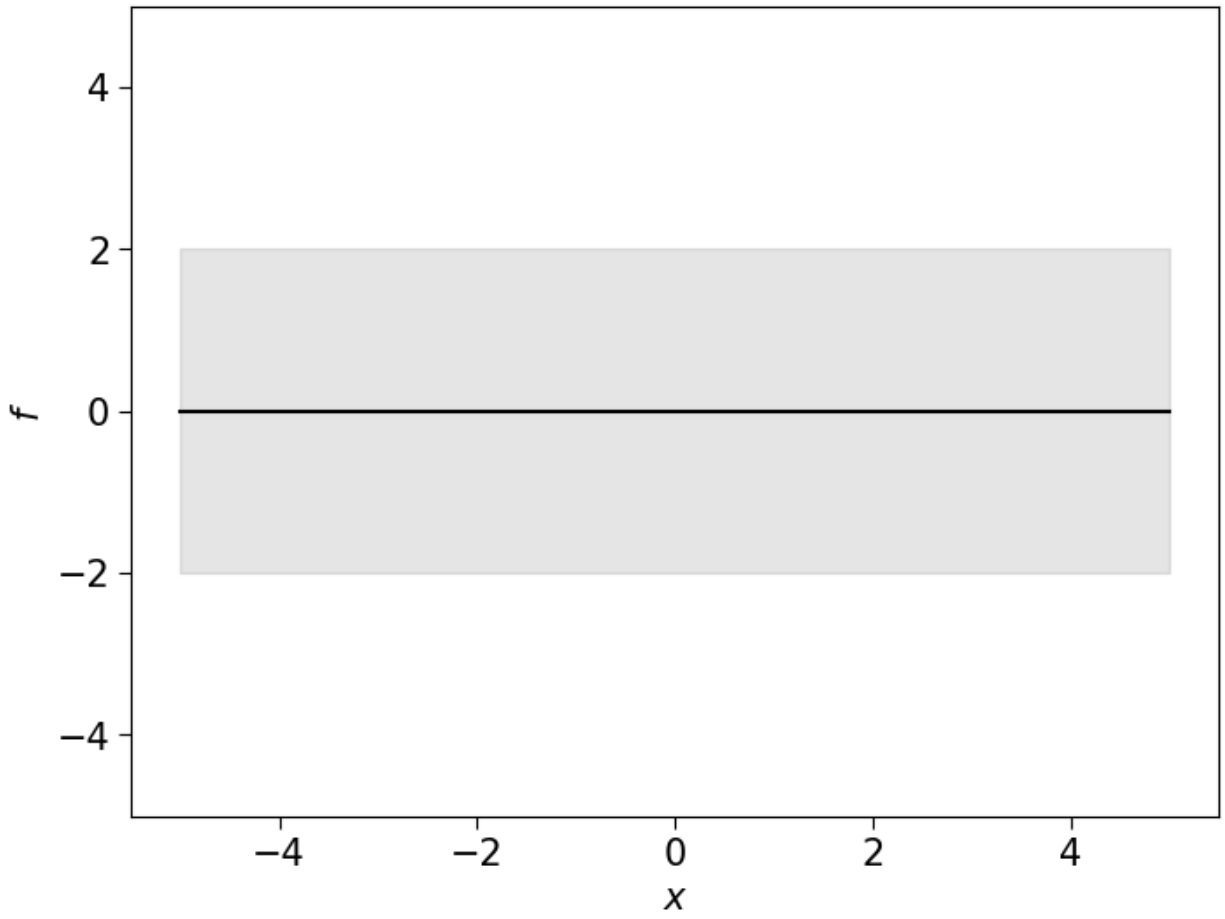
To visualise all these marginal distributions at once, we can produce a plot where we draw a curve that goes through the mean function value for each point and where we shade the area that is plus or minus two standard deviations from this mean. The significance of this is that 95% of the time, a sample drawn from a Gaussian distribution will lie within this region, so it represents our uncertainty.

The code below does this plot for the GP prior.

> Please note that it is very boring for this example, as every function value has mean 0 and variance 1!

```
def plot_gp_marginals(X, f_mean, f_std, fig=None, ax=None, show=False):

    if fig is None or ax is None:
        fig, ax = plt.subplots(figsize=[8, 6])

    ax.plot(X.flatten(), f_mean, color="black", label="Mean")
    ax.fill_between(
        X.flatten(),
```

```
            f_mean - 2 * f_std,
            f_mean + 2 * f_std,
            alpha=0.1,
            color="black",
        )
        ax.set_xlabel("$x$")
        ax.set_ylabel("$f$")
        ax.set_ylim([-5, 5])

        if show:
            plt.show()

        return


prior_means, prior_stds = gp.predict(X_star, return_std=True)
plot_gp_marginals(X_star, prior_means, prior_stds, show=True)
```



8.6 Gaussian Process posterior

Recall that when we had our Multivariate Gaussian, we partitioned our random vector into two. We can do the same thing with our GP. Let's split our points into $\mathbf{X}$ and $\mathbf{X}_*$ with corresponding function values $\mathbf{f}$ and $\mathbf{f}_*$ respectively. Continuing with our zero mean function assumption, we can write

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}(\mathbf{0}, \begin{bmatrix} \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}} & \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}_*} \\ \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}} & \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}_*} \end{bmatrix})$$

and condition as before:

$$p(\mathbf{f}_*|\mathbf{f}, \mathbf{X}, \mathbf{X}_*) = \mathcal{N}(\mathbf{f}_*; \boldsymbol{\mu}_{\mathbf{X}_*|\mathbf{X}}, \mathbf{\Sigma}_{\mathbf{X}_*|\mathbf{X}})$$

where

$$\boldsymbol{\mu}_{\mathbf{X}_*|\mathbf{X}} = \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}} \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}}^{-1} \mathbf{f}$$

$$\mathbf{\Sigma}_{\mathbf{X}_*|\mathbf{X}} = \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}_*} - \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}} \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}}^{-1} \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}_*}$$

These equations are slightly simpler than the conditioning equations from earlier because the prior means are both zero.

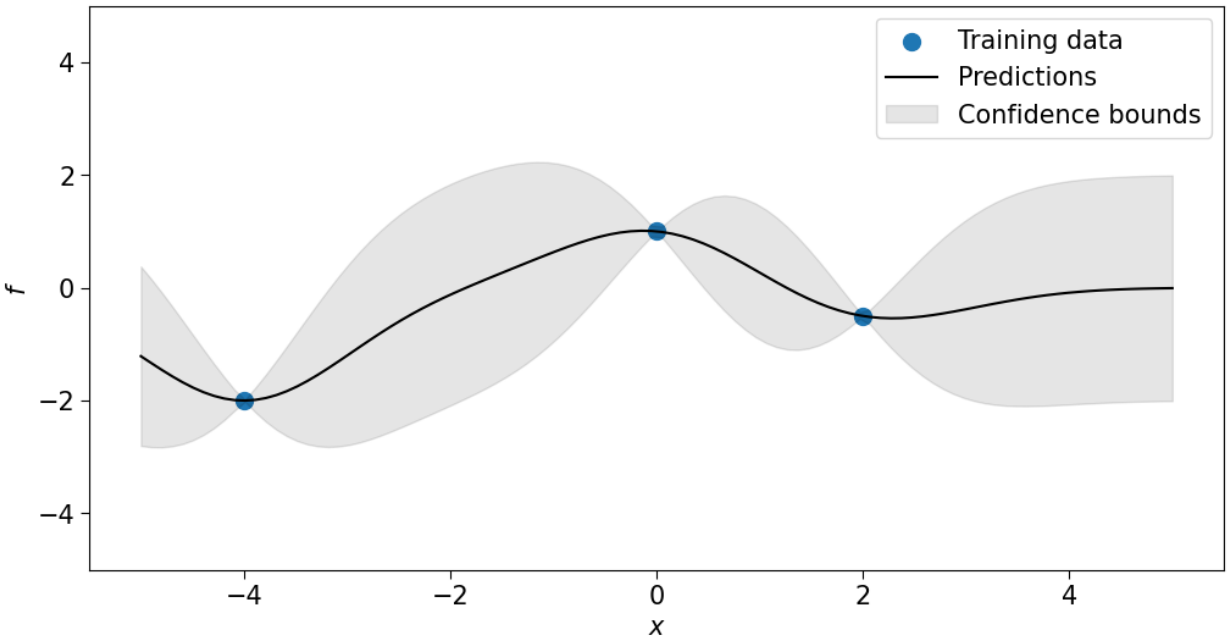
We now have everything we need to perform Gaussian process regression. If we have training inputs $\mathbf{X}$ with targets $\mathbf{f}$ and we want to make predictions at points $\mathbf{X}_*$ then we just compute $p(\mathbf{f}_*|\mathbf{f}, \mathbf{X}, \mathbf{X}_*)$.

We refer to $p(\mathbf{f}_*|\mathbf{f}, \mathbf{X}, \mathbf{X}_*)$ as our GP posterior. It is the distribution of function values $\mathbf{f}_*$ conditioned on our training data.

We can marginalise $p(\mathbf{f}_*|\mathbf{f}, \mathbf{X}, \mathbf{X}_*)$ for each individual function value to get a mean and variance (/standard deviation). We can use the mean as our prediction at that point and the standard deviation as a measure of uncertainty.

We will now fit a Gaussian process to training data (fitting a GP just means computing the posterior $p(\mathbf{f}_*|\mathbf{f}, \mathbf{X}, \mathbf{X}_*)$ for fixed hyperparameters) and look at the posterior's marginal distributions. We will use a simple 1D dataset with 3 data points.

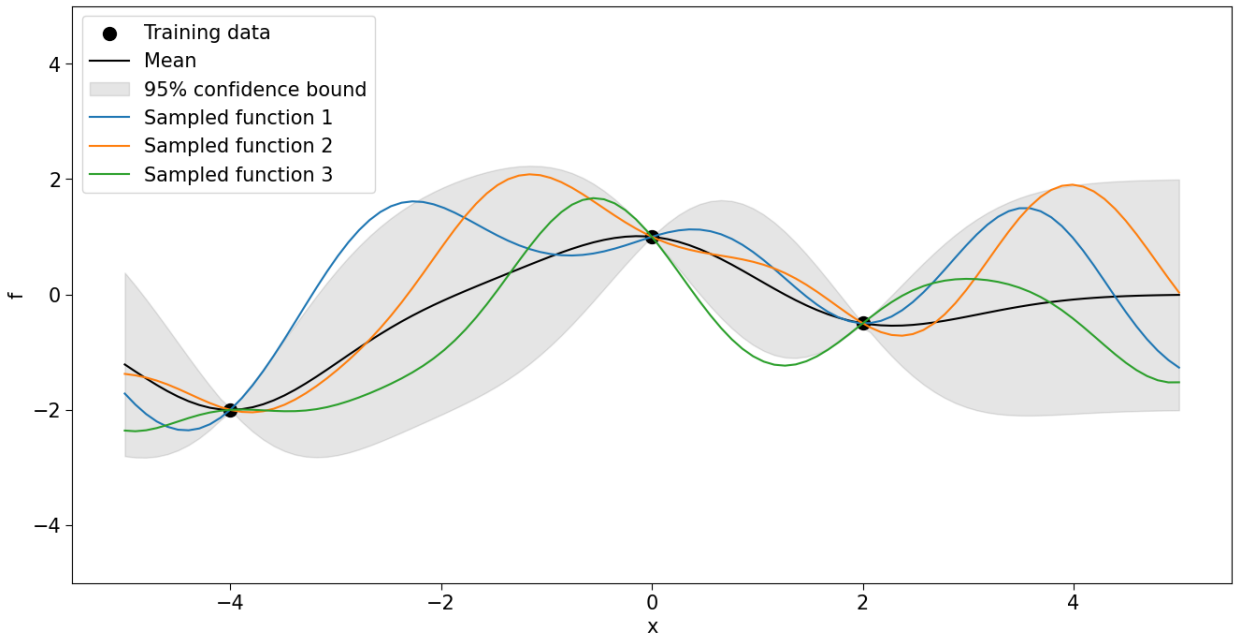
```
# Training set
X = np.array([-4, 0, 2]).reshape(-1, 1)
y = np.array([-2, 1, -0.5])

# Dummy points we want to evaluate at
X_star = np.linspace(-5, 5, 100).reshape(-1, 1)

# Fit the GP (ignore what alpha is for now)
gp.fit(X, y)

fig, ax = plt.subplots(figsize=[12, 6])

# Plot the training data
ax.scatter(X, y)

# Compute then plot the marginal distributions of the posterior
posterior_means, posterior_stds = gp.predict(X_star, return_std=True)
plot_gp_marginals(X_star, posterior_means, posterior_stds, fig=fig,
 ↪  ax=ax)

plt.legend(["Training data", "Predictions", "Confidence bounds"])

plt.show()
```



Points near training points have low uncertainty and points further away have higher uncertainty. Getting uncertainty measurements alongside predictions is really nice!

In addition to this "sausage plot", we can sample functions from the GP posterior.

```
# Plot training data
fig, ax = plt.subplots(figsize=[16, 8])
ax.scatter(X, y, color="k")

plot_gp_marginals(X_star, posterior_means, posterior_stds, fig=fig,
 ↪  ax=ax)

# Sample 3 realisations of f_star (3 possible functions) from the GP
 ↪  posterior with different seeds
f_star_0 = gp.sample_y(X_star, random_state=0)
f_star_1 = gp.sample_y(X_star, random_state=1)
f_star_2 = gp.sample_y(X_star, random_state=2)

# Plot these
ax.plot(X_star, f_star_0)
ax.plot(X_star, f_star_1)
ax.plot(X_star, f_star_2)

ax.set_xlabel("x")
ax.set_ylabel("f")
plt.legend(
    [
        "Training data",
        "Mean",
        "95% confidence bound",
        "Sampled function 1",
        "Sampled function 2",
        "Sampled function 3",
    ]
)

plt.show()
```



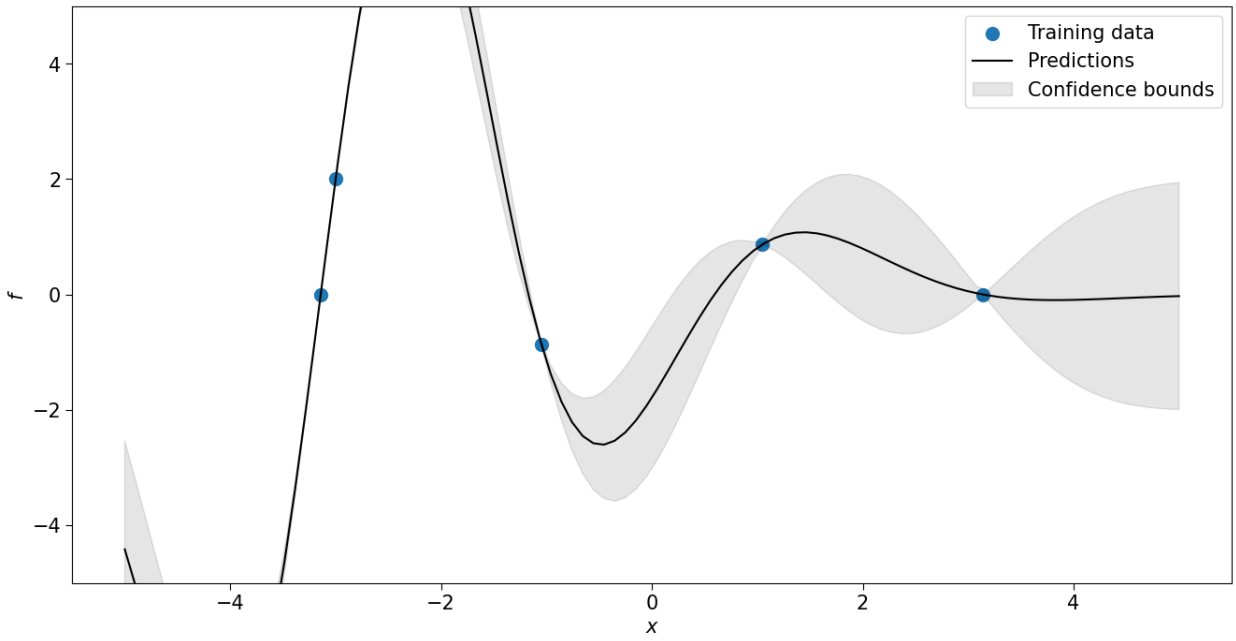
All functions sampled interpolate the training data. This is sensible, except it does not account for noise in the training targets.

8.7 Dealing with noisy measurements

Most of the time it is unrealistic to assume that our targets are free of any noise, and it isn't ideal if the functions generated by the GP posterior are forced to pass through a dubious training point.

In the example below, most of the data points are drawn from a sine curve, but I've added one data point that isn't. Rather than following the nice sine pattern, the GP predictions oscillate wildly to ensure that all the points are interpolated.

```
X = np.linspace(-np.pi, np.pi, 4)
y = np.sin(X)

# Add a daft point
X = np.append(X, -3)
y = np.append(y, 2)

# Fit a GP
gp = GaussianProcessRegressor(alpha=0).fit(X.reshape(-1, 1), y)

# Dummy points we want to evaluate at
X_star = np.linspace(-5, 5, 100).reshape(-1, 1)

# Plot
fig, ax = plt.subplots(figsize=[16, 8])
ax.scatter(X, y)
posterior_means, posterior_stds = gp.predict(X_star, return_std=True)
plot_gp_marginals(X_star, posterior_means, posterior_stds, fig=fig,
 ↪  ax=ax)
plt.legend(["Training data", "Predictions", "Confidence bounds"])
plt.show()
```



We will assume that our targets $\mathbf{y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(N)} \end{bmatrix}^\top$ suffer from additive Gaussian noise with standard deviation σ_y.

$$y = f(x) + \mathcal{N}(0, \sigma_y^2)$$

This means for some target y we have $y = f(x) + \mathcal{N}(0, \sigma_y^2)$. For multiple targets $\mathbf{y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(N)} \end{bmatrix}^\top$ we have $\mathbf{y} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}} + \sigma_y^2 \mathbf{I})$.

Instead of writing the joint distribution of $\mathbf{f}$ and $\mathbf{f}_*$ as we did in the noise-free case, we can write the joint distribution of $\mathbf{y}$ and $\mathbf{f}_*$ (which is very similar):

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}(\mathbf{0}, \begin{bmatrix} \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}} + \sigma_y^2 \mathbf{I} & \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}_*} \\ \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}} & \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}_*} \end{bmatrix})$$

We can condition to get

$$p(\mathbf{f}_* | \mathbf{y}, \mathbf{X}, \mathbf{X}_*) = \mathcal{N}(\mathbf{f}_*; \boldsymbol{\mu}_{\mathbf{X}_*|\mathbf{X}}, \mathbf{\Sigma}_{\mathbf{X}_*|\mathbf{X}})$$

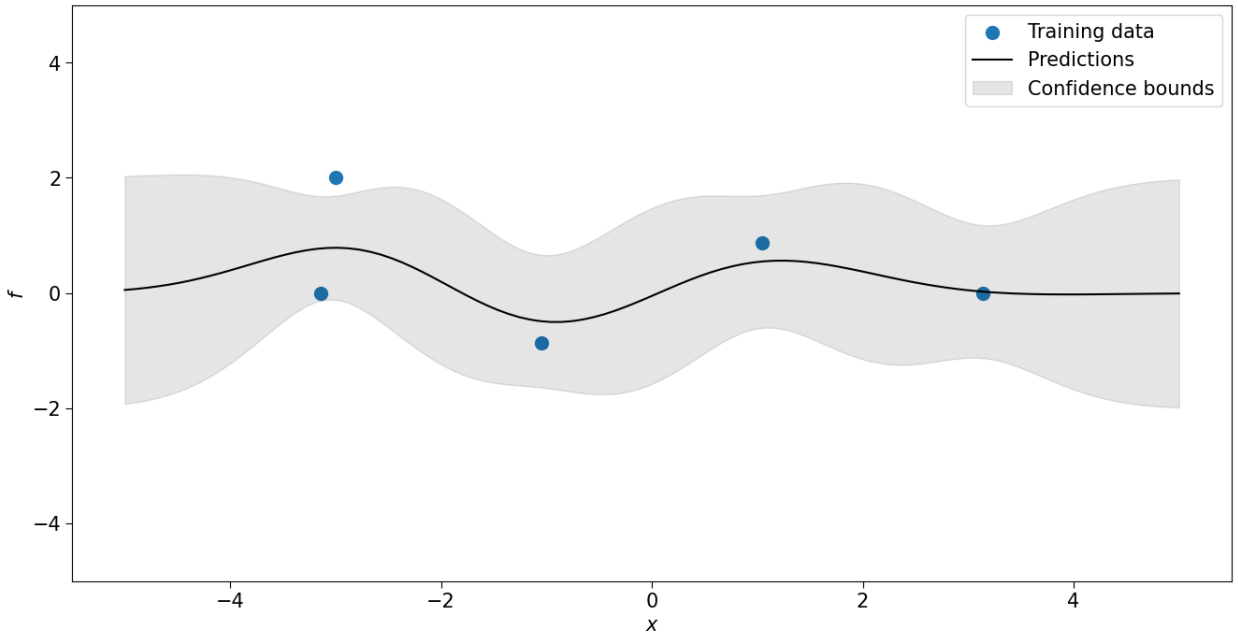
where

$$\boldsymbol{\mu}_{\mathbf{X}_*|\mathbf{X}} = \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}} (\mathbf{\Sigma}_{\mathbf{X},\mathbf{X}} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{y}$$

$$\mathbf{\Sigma}_{\mathbf{X}_*|\mathbf{X}} = \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}_*} - \mathbf{\Sigma}_{\mathbf{X}_*,\mathbf{X}} (\mathbf{\Sigma}_{\mathbf{X},\mathbf{X}} + \sigma_y^2 \mathbf{I})^{-1} \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}_*}$$

σ_y^2 (note the square) is given by the `alpha` argument to the GP in sklearn. Let's redo the plot above but with a moderate σ_y^2.

```
# Fit a GP, assuming there is additive label noise with var 0.5
gp = GaussianProcessRegressor(alpha=0.5).fit(X.reshape(-1, 1), y)

# Dummy points we want to evaluate at
X_star = np.linspace(-5, 5, 100).reshape(-1, 1)

# Plot
fig, ax = plt.subplots(figsize=[16, 8])
ax.scatter(X, y)
posterior_means, posterior_stds = gp.predict(X_star, return_std=True)
plot_gp_marginals(X_star, posterior_means, posterior_stds, fig=fig,
 ↪  ax=ax)
plt.legend(["Training data", "Predictions", "Confidence bounds"])
plt.show()
```



The prediction curve no longer interpolates the training points and there is now uncertainty even when it lines up with training data.

8.8 Optimising kernel hyperparameters

Consider a GP with zero mean function and kernel

$$k(\mathbf{x}, \mathbf{x}') = \sigma^2 \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\ell^2}\right)$$

This is the RBF kernel — which has hyperparameter ℓ — scaled by another hyperparameter σ^2. Changing these has a significant effect on the functions the GP can produce.

These hyperparameters can be optimised through e.g. a grid search but a fast alternative is to instead **find the hyperparameters that maximise the marginal likelihood of our targets** $p(\mathbf{y}|\mathbf{X})$. This is called a *marginal* likelihood because we have marginalised out $\mathbf{f}$

$$p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|\mathbf{X}, \mathbf{f})\, p(\mathbf{f}|\mathbf{X})\, d\mathbf{f}$$

We already know that $p(\mathbf{y}|\mathbf{X}) = \mathcal{N}(\mathbf{y}; \mathbf{0}, \mathbf{\Sigma}_{\mathbf{X},\mathbf{X}} + \sigma_y^2\mathbf{I})$ for noisy measurements. The exponential in the normal is annoying, but maximising $p(\mathbf{y}|\mathbf{X})$ is the same as minimising $-\log p(\mathbf{y}|\mathbf{X})$.

$$\log p(\mathbf{y}|\mathbf{X}) = -\frac{1}{2}\mathbf{y}^\top(\mathbf{\Sigma}_{\mathbf{X},\mathbf{X}} + \sigma_y^2\mathbf{I})^{-1}\mathbf{y} - \frac{1}{2}\log|\mathbf{\Sigma}_{\mathbf{X},\mathbf{X}} + \sigma_y^2\mathbf{I}| - \frac{N}{2}\log 2\pi$$

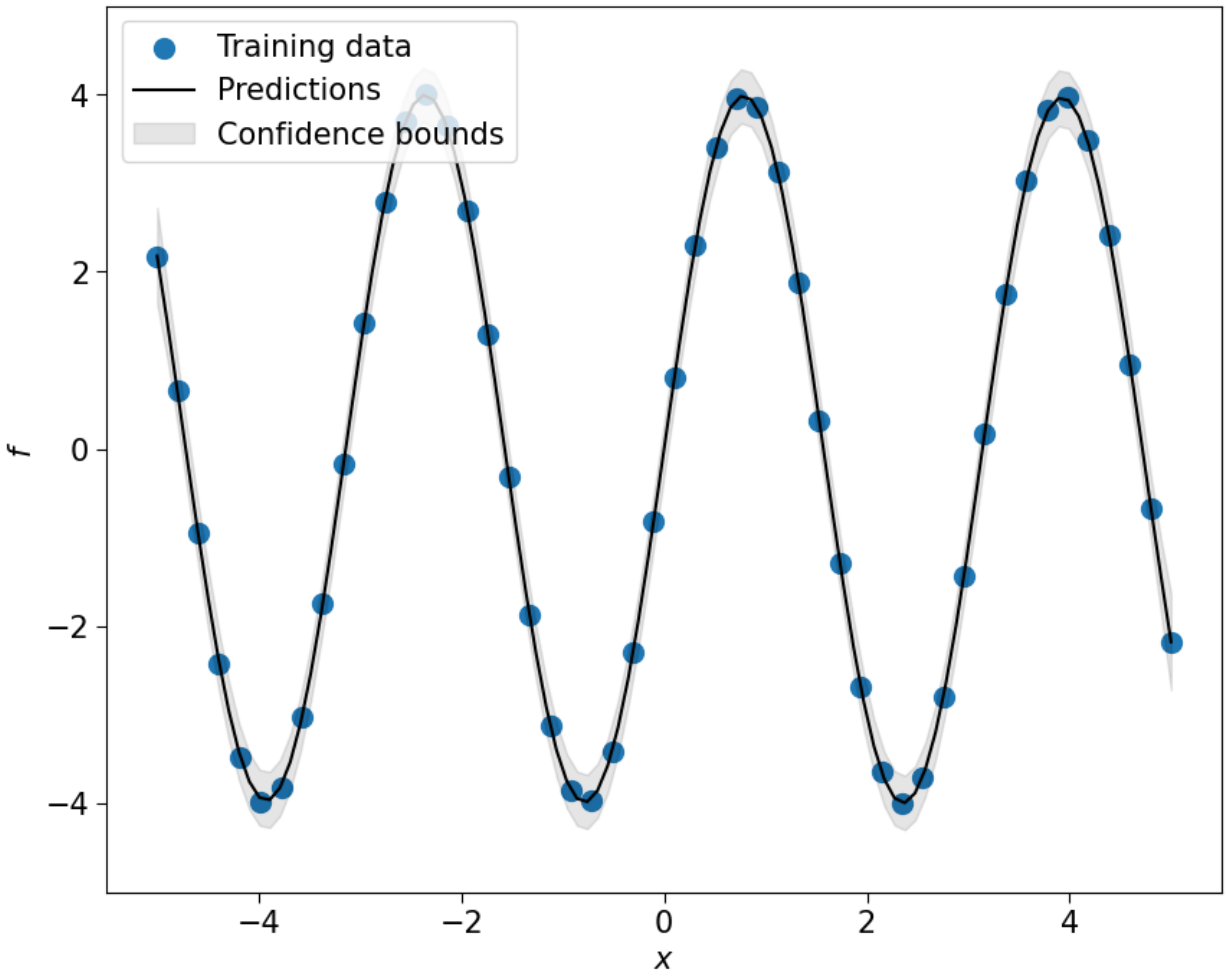
This can be differentiated with respect to hyperparameters as long as the kernel is differentiable (although I am not going to write out the expression!). This means a gradient based optimiser can be used to solve $\underset{\boldsymbol{\theta}}{\text{minimise}}\ -\log p(\mathbf{y}|\mathbf{X})$.

> When you call `.fit` for GPs in sklearn, this is done by default.
>
> This means that training data is used to optimise these hyperparameters instead of validation data, which is unusual. However, there is some work that claims there is an equivalence to cross-validation. There is also work claiming that maximising marginal likelihood can harm generalisation!
>
> I am not going to claim that maximising marginal likelihood is the correct thing to do, only note that it is what sklearn does and to be aware that alternative approaches may be better. As with a lot of things, it all depends on your data.

Let's see it in action using a GP with the kernel above.

```
# We will use a bigger dataset of points on a sine curve
X = np.linspace(-5, 5, 50)
y = 4 * np.sin(2 * X)

# kernel = constant kernel * RBF
kernel = ConstantKernel() * RBF()

# Create a GP with this kernel, reducing the measurement noise somewhat
gp = GaussianProcessRegressor(kernel=kernel, alpha=0.1)

# Print the initial kernel hyperparameters
print(f"Initial kernel:\t{gp.kernel}")

# Fit the GP
gp.fit(X.reshape(-1, 1), y)

# Print the fitted kernel hyperparameters
print(f"Fitted kernel:\t{gp.kernel_}")

# Plot
fig, ax = plt.subplots(figsize=[10, 8])
ax.scatter(X, y)
posterior_means, posterior_stds = gp.predict(X_star, return_std=True)
plot_gp_marginals(X_star, posterior_means, posterior_stds, fig=fig,
 ↪  ax=ax)
plt.legend(["Training data", "Predictions", "Confidence bounds"],
 ↪  loc='upper left')
plt.show()
```

```
Initial kernel:    1**2 * RBF(length_scale=1)
Fitted kernel:    5.75**2 * RBF(length_scale=1.14)
```


Training data
Predictions
Confidence bounds
4
2
0
−2
−4
f
−4
−2
0
2
4
x

9 Deep neural networks

```
# RUN THIS CELL FIRST

# Imports
import warnings
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# sklearn
from sklearn.exceptions import ConvergenceWarning
from sklearn.linear_model import LogisticRegression
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.neural_network import MLPClassifier

# Kill convergence warnings
warnings.filterwarnings(action="ignore", category=ConvergenceWarning)

# This makes matplotlib output nice figures without much tweaking
plt.rcParams.update(
    {
        "lines.markersize": 10,  # Big points
        "font.size": 15,  # Larger font
        "xtick.major.size": 5.0,  # Bigger xticks
        "ytick.major.size": 5.0,  # Bigger yticks
    }
)
```

We now move on to the topic that is responsible for most of the current hype in machine learning. When someone in the media refers to "AI", they usually mean an (artificial) neural network. Although you're about to find out that neural networks are pretty mundane in form!

> Neural networks are often called **deep** neural networks (**DNNs**). The *deep* refers to the fact that these comprise of many *layers* of processing (and because *deep* is a buzzword) and the *neural* is because of an extremely tenuous connection to how biological neurons work in the brain (and because *neural* is a buzzword).

9.1 Feature learning

For classification and regression we have predominantly considered linear models of the form

$$f(\mathbf{x}) = \mathbf{w}^\top \phi(\mathbf{x}) + b$$

where:

- $\mathbf{x} \in \mathbb{R}^D$ and $f(\mathbf{x}) \in \mathbb{R}$
- ϕ is a transformation that maps $\mathbf{x}$ to a feature vector $\phi(\mathbf{x}) \in \mathbb{R}^Z$

- $\mathbf{w} \in \mathbb{R}^Z$ and $b \in \mathbb{R}$

This setup is appropriate for regression to one dimensional outputs, or for binary classification. If, however, we want to perform regression to multidimensional outputs or perform multiway classification, then we can write a linear model more generally as:

$$f(\mathbf{x}) = \mathbf{W}\phi(\mathbf{x}) + \mathbf{b}$$

where:

- $\mathbf{x} \in \mathbb{R}^D$ and $f(\mathbf{x}) \in \mathbb{R}^K$
- ϕ is a transformation that maps $\mathbf{x}$ to a feature vector $\phi(\mathbf{x}) \in \mathbb{R}^Z$
- $\mathbf{W} \in \mathbb{R}^{K \times Z}$ and $b \in \mathbb{R}^K$

The features we use are very important! Before DNNs became mainstream there was a lot of hand-engineering of (terrible) features. DNNs give us a model framework where we can learn features directly from data. We effectively have a parameterised feature transformation:

$$f(\mathbf{x}) = \mathbf{W}\phi_{\boldsymbol{\theta}_f}(\mathbf{x}) + \mathbf{b}$$

For some loss L with a linear model we would solve $\underset{\mathbf{W},\mathbf{b}}{\text{minimise}}\, L$. With a DNN we instead solve $\underset{\boldsymbol{\theta}_f,\mathbf{W},\mathbf{b}}{\text{minimise}}\, L$. This means we learn a feature transformation jointly with learning the linear transformation that is applied to those features.

So what does $\phi_{\boldsymbol{\theta}}(\mathbf{x})$ look like? We will consider the simplest DNN - **the multilayer perceptron (MLP)** - where $\phi_{\boldsymbol{\theta}}(\mathbf{x})$ is a series of linear transformations, each followed by an *activation function*.

9.2 Multilayer perceptrons (MLPs)

A linear transformation takes in a vector $\mathbf{h}^{(l-1)} \in \mathbb{R}^{N_{l-1}}$, pre-multiplies it by a weight matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{N_l \times N_{l-1}}$ and adds a bias vector $\mathbf{b}^{(l)} \in \mathbb{R}^{N_l}$ to give us a vector $\mathbf{a}^{(l)} \in \mathbb{R}^{N_l}$

$$\mathbf{a}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$$

An activation function takes in a vector and applies a non-linear function to each element of that vector. The most common function used for this is a RELU (rectified linear unit) $\text{RELU}(a) = \max(a, 0)$. This simply sets negative elements of the incoming vector to zero, e.g.

$$\text{RELU}\left(\begin{bmatrix} 5 \\ -4 \\ -1 \\ 7 \end{bmatrix}\right) = \begin{bmatrix} 5 \\ 0 \\ 0 \\ 7 \end{bmatrix}$$

- Each linear transform + activation function pair is referred to as a **layer**.
- The output of a layer is referred to as an **activation**.

- The input to the activation function is referred to as a **pre-activation**.

When counting layers we typically include the final linear transformation too.

If we denote $\mathbf{x}$ as $\mathbf{h}^{(0)}$, and the activation function as g then we can write an $\mathcal{L}$ layer MLP as:

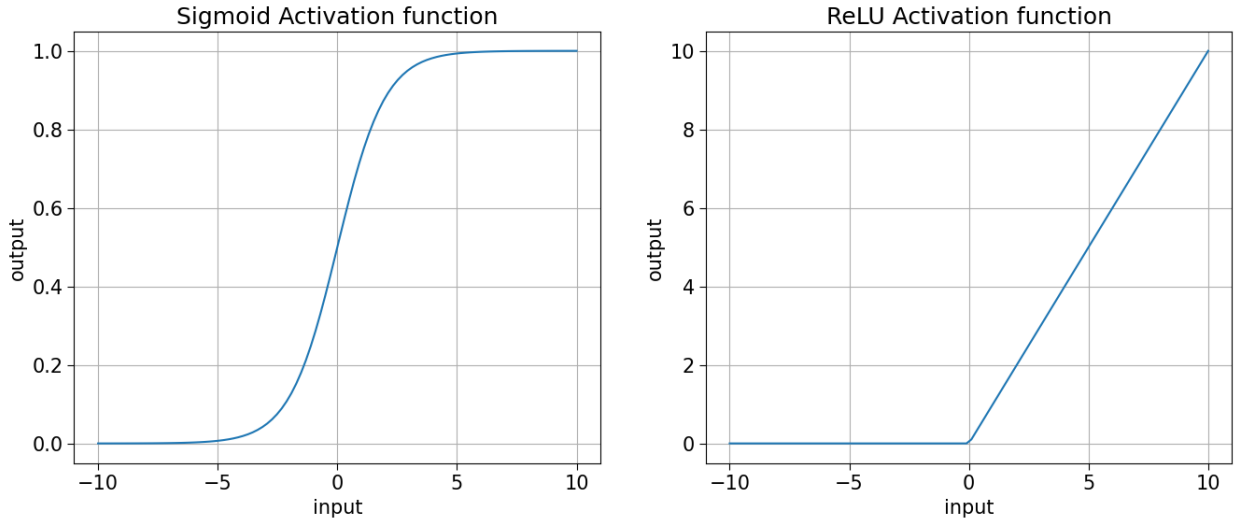
$$\mathbf{h}^{(l)} = g(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \text{ for } l = 1, 2, \ldots, \mathcal{L} - 1$$

$$f(\mathbf{x}) = \mathbf{W}^{(\mathcal{L})}\mathbf{h}^{(\mathcal{L}-1)} + \mathbf{b}^{(\mathcal{L})}$$

Note that the final layer needn't always be linear. For instance, if you were regressing to a value that you knew could only be between 0 and 1, you could pump the output through a sigmoid.

We can compare this second equation to the $f(\mathbf{x})$ above to see that our feature $\phi_{\boldsymbol{\theta}_f}(\mathbf{x})$ for an input $\mathbf{x}$ is the output of the penultimate layer ($\mathbf{h}^{(\mathcal{L}-1)}$) and that $\boldsymbol{\theta}_f$ consists of the weight matrices and bias vectors of all the MLP layers apart from the last. We can store all our parameters into one array $\boldsymbol{\theta} = \{\boldsymbol{\theta}_f, \mathbf{W}^{(\mathcal{L})}, \mathbf{b}^{(\mathcal{L})}\}$ for notational convenience.

```
# Some example activation functions
x = np.linspace(-10,10,100)

sigmoid = [1/(1+np.exp(-z)) for z in x]
ReLU    = [max(0,z) for z in x]

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=[16,6])

axes[0].plot(x, sigmoid)
axes[0].set_title("Sigmoid Activation function")
axes[1].plot(x, ReLU)
axes[1].set_title("ReLU Activation function")

for ax in axes:
    ax.grid()
    ax.set_xlabel("input")
    ax.set_ylabel("output")
    ax.set_xticks([-10,-5,0,5,10])

plt.show()
```



9.3 Learning the parameters of an MLP

Let's say we have a training set $\{(\mathbf{x}^{(n)}, \mathbf{y}^{(n)})\}_{n=1}^{N}$, where, for classification, each $\mathbf{y}$ is a one-hot class label and, for regression, is some vector of continuous values.

The problem of learning the parameters of an MLP is $\underset{\boldsymbol{\theta}}{\text{minimise}}\ L$ where $\boldsymbol{\theta} = \left\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\right\}_{l=1}^{\mathcal{L}}$

For classification L is typically the log loss

$$L_{log} = \frac{1}{N}\sum_{n} -\mathbf{y}^{(n)\top} \log S(f(\mathbf{x}^{(n)}))$$

where S is the softmax function. For regression, L is typically mean squared error (MSE).

$$L_{MSE} = \frac{1}{N}\sum_{n} \|\mathbf{y}^{(n)} - f(\mathbf{x}^{(n)})\|^2$$

Note that this is slightly different to the MSE you saw in the 1D output case.

We will solve this optimisation problem with a gradient-based optimiser. Specifically, we will iteratively use the backpropagation algorithm (which is really just the chain rule!) to collect the gradients of the loss wrt. all the weight matrices and bias vectors:

$$\nabla_{\boldsymbol{\theta}} L = \left\{\left(\frac{\partial L}{\partial \mathbf{W}^{(l)}}\right)^{\top}, \left(\frac{\partial L}{\partial \mathbf{b}^{(l)}}\right)^{\top}\right\}_{l=1}^{\mathcal{L}}$$

and use these to perform updates. While I did give some more detail about backpropagation in the lecture, this is all you need to know for this course.

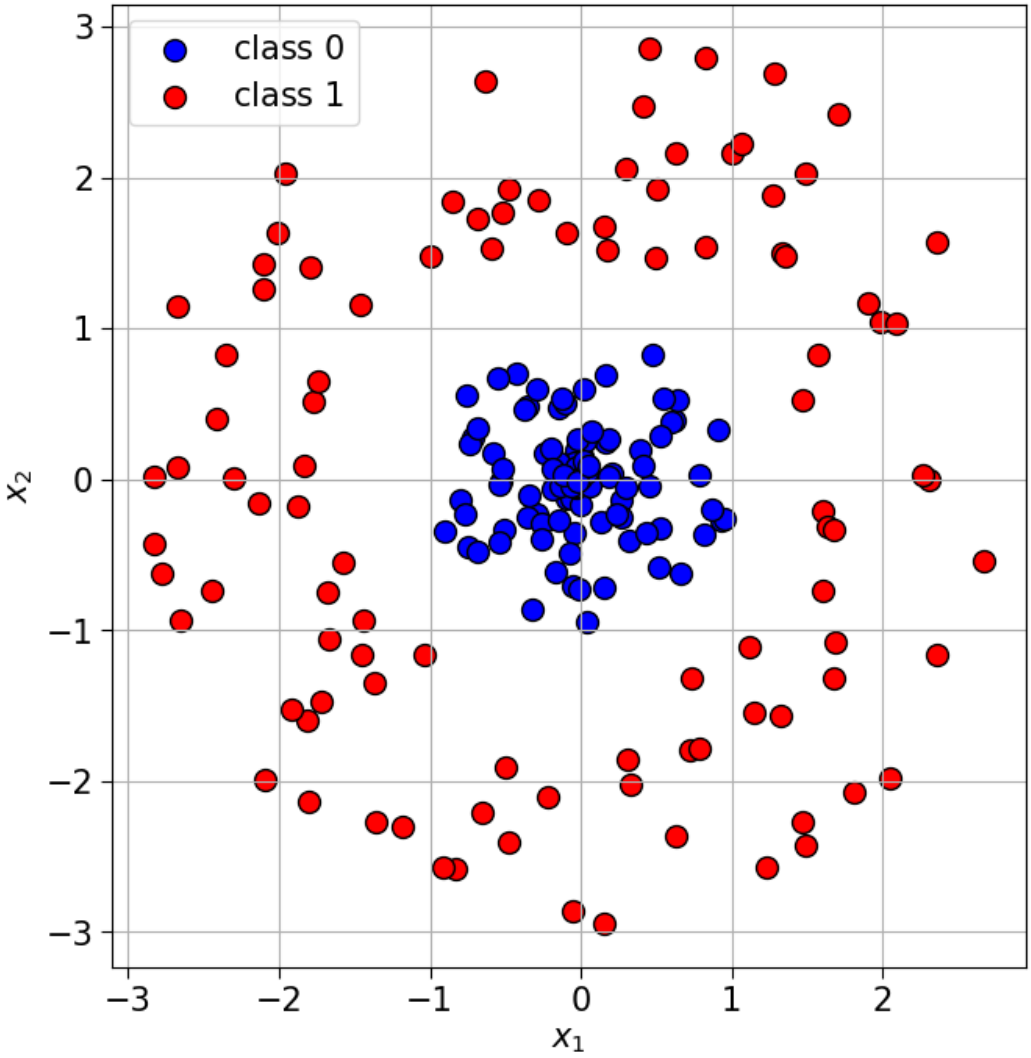
In practice, MLPs are optimised using stochastic gradient descent (or a variant of it called Adam) where at each step, a **mini-batch** of data points is used to approximate the gradients for the whole dataset. This is mainly because computing gradients requires storing activations, which is prohibitively expensive for a large dataset.

> It's worth being aware that neither of the above losses are convex when our model is an MLP. However, we often find that we end up in a local minimum which is good enough for the job.

9.4 MLPs in sklearn

We'll now investigate fitting MLPs in sklearn on a toy training set, which is created and plotted using the code below.

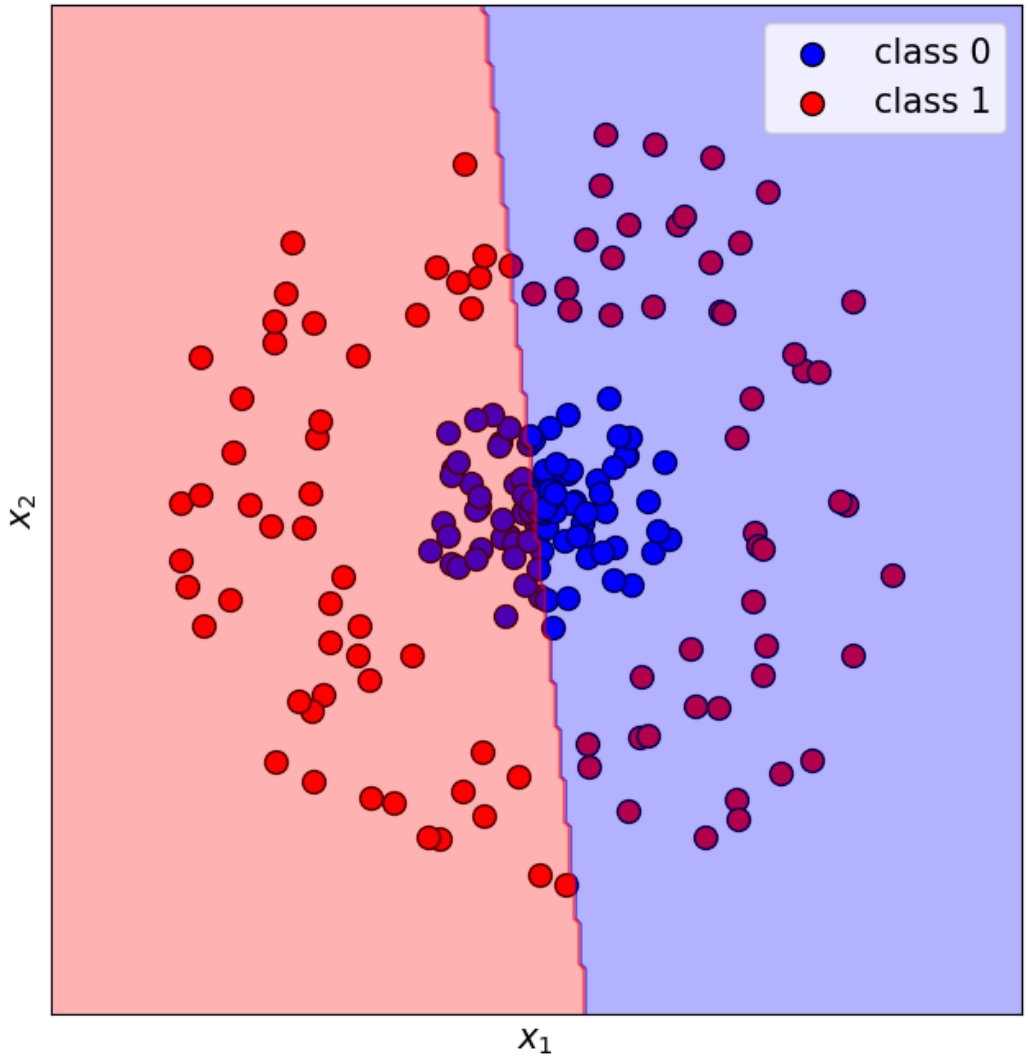
```
# Generate points from a circle
np.random.seed(42)

radii = np.hstack(
    [np.random.uniform(0, 1, size=100), np.random.uniform(1.5, 3,
     ↪  size=100)]
)
theta = np.random.uniform(0, 2 * np.pi, size=200)

X       = np.zeros((200, 2))
X[:, 0] = radii * np.sin(theta)
X[:, 1] = radii * np.cos(theta)

y = np.hstack([np.zeros(100), np.ones(100)])

# Now let's plot our data, with a different colour for each class
fig, ax = plt.subplots(figsize=[8, 8])
ax.set_aspect("equal", "box")
ax.scatter(X[y == 0, 0], X[y == 0, 1], color="b", edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color="r", edgecolor="k")
ax.grid()
ax.legend(["class $0$", "class $1$"])
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.show()
```



This is a binary classification problem and the training data is clearly not linearly separable. Let's briefly fit a linear model and draw its decision boundary to see how unsuitable it is.

```
# Fit logistic regression
model = LogisticRegression().fit(X, y)

# Make plot
fig, ax = plt.subplots(figsize=[8, 8])
ax.set_aspect("equal", "box")
ax.scatter(X[y == 0, 0], X[y == 0, 1], color="b", edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color="r", edgecolor="k")
ax.grid()
ax.legend(["class $0$", "class $1$"])

ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1"], loc="upper right")
ax.set_xticks([])
ax.set_yticks([])
colours = ["b", "r"]

# Plot decision boundary
disp = DecisionBoundaryDisplay.from_estimator(
    model,
    X,
```

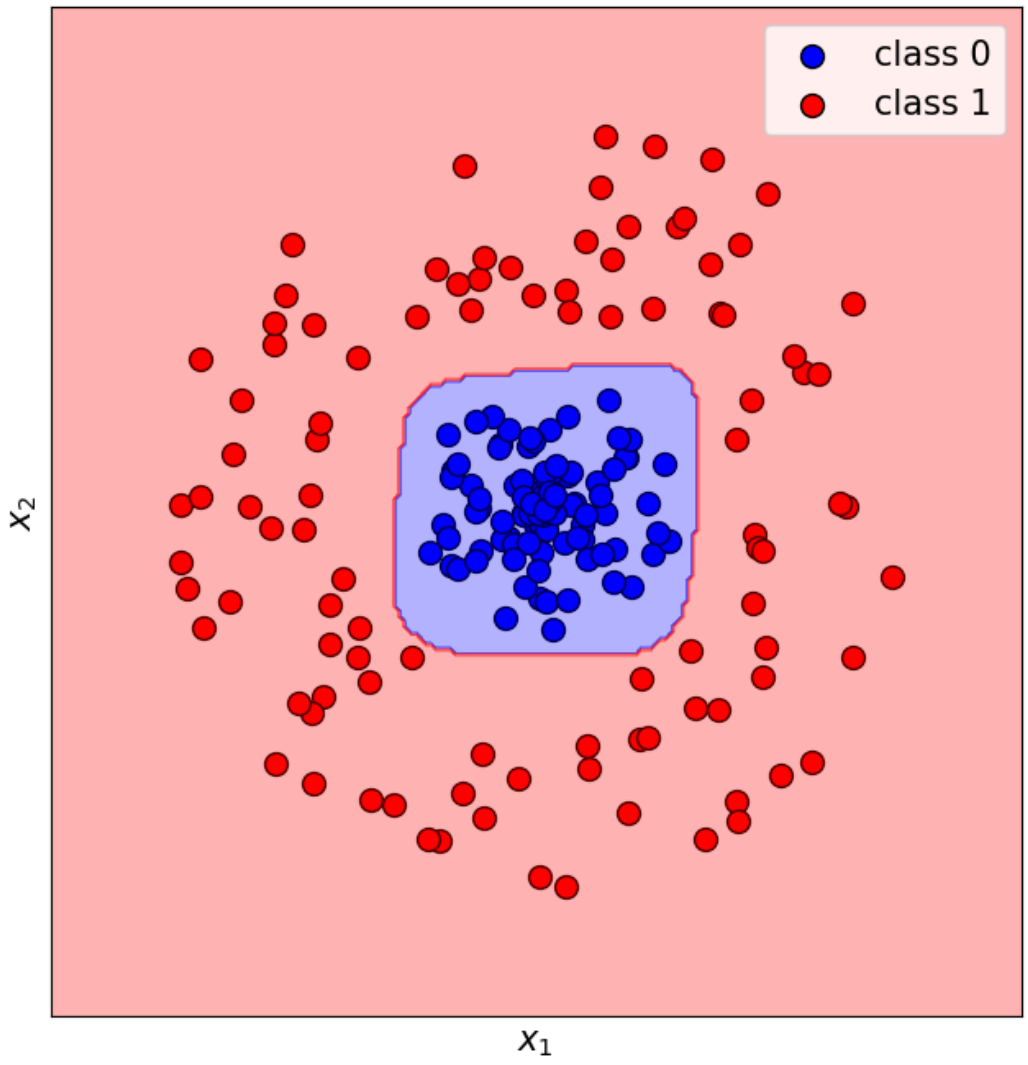
```
    response_method="predict",
    alpha=0.3,
    grid_resolution=200,
    cmap=ListedColormap(colours),
    ax=ax,
)

plt.show()
```



Ouch. Let's fit an MLP instead. We can use MLPClassifier for this. By default, the model will be a 2 layer MLP (i.e. an MLP with 1 hidden layer) with hidden width of 100 that uses ReLU activations:

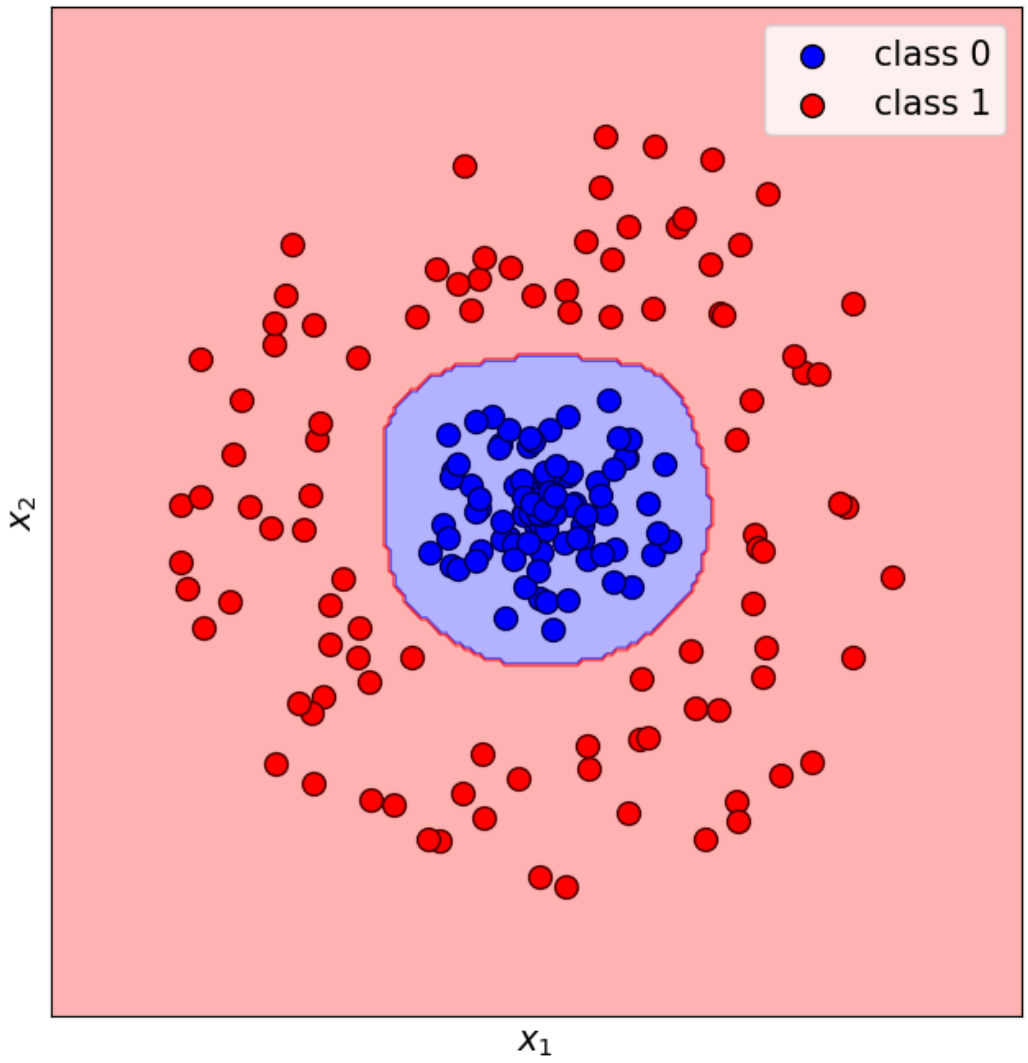
```
# Fit MLP
model = MLPClassifier(random_state=42).fit(X, y)

# Make plot
fig, ax = plt.subplots(figsize=[8, 8])
ax.set_aspect("equal", "box")
ax.scatter(X[y == 0, 0], X[y == 0, 1], color="b", edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color="r", edgecolor="k")
ax.grid()
ax.legend(["class $0$", "class $1$"])
```

```
ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1"], loc="upper right")
ax.set_xticks([])
ax.set_yticks([])
colours = ["b", "r"]

# Plot decision boundary
disp = DecisionBoundaryDisplay.from_estimator(
    model,
    X,
    response_method="predict",
    alpha=0.3,
    grid_resolution=200,
    cmap=ListedColormap(colours),
    ax=ax,
)

plt.show()
```



This is much better, but the decision boundary could be more circular. Let's create a 3 layer MLP. We can do this by setting `hidden_layer_sizes=[100, 100]`. This tells us the MLP has 2 hidden layers, each with a width of 100 (so 3 layers in total).

```
# Fit MLP
model = MLPClassifier(hidden_layer_sizes=[100, 100],
 ↪  random_state=42).fit(X, y)

# Make plot
fig, ax = plt.subplots(figsize=[8, 8])
ax.set_aspect("equal", "box")
ax.scatter(X[y == 0, 0], X[y == 0, 1], color="b", edgecolor="k")
ax.scatter(X[y == 1, 0], X[y == 1, 1], color="r", edgecolor="k")
ax.grid()
ax.legend(["class $0$", "class $1$"])

ax.set_xlabel("$x_1$")
ax.set_ylabel("$x_2$")
plt.legend(["class 0", "class 1"], loc="upper right")
ax.set_xticks([])
ax.set_yticks([])
colours = ["b", "r"]

# Plot decision boundary
disp = DecisionBoundaryDisplay.from_estimator(
    model,
    X,
    response_method="predict",
    alpha=0.3,
    grid_resolution=200,
    cmap=ListedColormap(colours),
    ax=ax,
)

plt.show()
```



That's better. You will explore the effect on changing the number of hidden layers and their widths on MNIST in the lab.

10 Neural architectures and transfer learning

Execution note

The PyTorch/skorch example in this chapter was not executed for this edition, so training output and test accuracy are not reported. It requires PyTorch and skorch; the installation command also requires internet access if skorch is not already installed. The downloadable notebook includes the required MNIST data.

```
# RUN THIS CELL FIRST.
import numpy as np
import matplotlib.pyplot as plt

# If you are running this on Notable, torch should be preinstalled
import torch
from torch import nn

# Install and import skorch, a sklearn-style interface to PyTorch
!pip install skorch
import skorch
from skorch import NeuralNetClassifier
```

sklearn is designed primarily for classical machine learning and does not provide end-to-end building blocks for neural network architectures such as convolutional networks (ConvNets) or Transformers. In practice, these models are implemented and trained using deep learning frameworks such as PyTorch or TensorFlow. To keep the tooling consistent, I won't expect you to learn a full deep learning framework in the final taught week of the course.

That said, I will include an example using skorch (an sklearn-style interface to PyTorch) to show what a practical implementation looks like. You are not expected to be able to write or modify skorch/PyTorch code for this course.

10.1 Convolutional Neural Networks (ConvNets)

Last week, we looked at MLPs. These take in a vector input $\mathbf{x} \in \mathbb{R}^D$ and pass it through multiple layers. Each layer takes in some vector $\mathbf{h}^{(l-1)} \in \mathbb{R}^{H_{l-1}}$, multiplies it by a matrix, adds a bias, and then applies an elementwise nonlinearity:

$$\mathbf{h}^{(l)} = g(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \text{ for } l = 1, 2, \ldots$$

An image is represented as a 3D tensor $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$. Vectorising an image for use in an MLP preserves the pixel values, but an MLP does not explicitly exploit their spatial arrangement. In the lecture, we looked at convolutional neural networks (ConvNets). These take in images as 3D tensors, and use a mixture of convolutional, pooling, and linear (fully connected) layers.

10.1.1 Convolutional layers

A convolutional layer is quite similar to a linear layer in an MLP, in that it consists of a linear transformation followed by a non-linearity. However, it takes in a 3D tensor $\mathbf{h}^{(l-1)} \in \mathbb{R}^{C_{l-1} \times H_{l-1} \times W_{l-1}}$ instead of a vector, and the linear transformation is a **2D convolution**:

$$\mathbf{h}^{(l)} = g(\mathbf{W}^{(l)} * \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \text{ for } l = 1, 2, \ldots$$

Each $\mathbf{W}^{(l)} \in \mathbb{R}^{C_l \times C_{l-1} \times k \times k}$ consists of C_l filters, each of dimensionality $C_{l-1} \times k \times k$ where k is the kernel size.

If we define $\mathbf{z}^{(l)} = \mathbf{W}^{(l)} * \mathbf{h}^{(l-1)} \in \mathbb{R}^{C_l \times H_l \times W_l}$ then each of its elements $\mathbf{z}^{(l)}_{c,i,j}$ is computed as

$$\mathbf{z}^{(l)}_{c,i,j} = \sum_{c'=1}^{C_{l-1}} \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} \mathbf{W}^{(l)}_{c,c',u,v} \, \mathbf{h}^{(l-1)}_{c',i+u,j+v}$$

The bias $\mathbf{b}^{(l)} \in \mathbb{R}^{C_l}$ has one element per output channel. Elements of the preactivation $\mathbf{a}^{(l)} = \mathbf{z}^{(l)} + \mathbf{b}^{(l)}$ (where I have been rather lax with my maths, as I am broadcasting) are computed as

$$\mathbf{a}^{(l)}_{c,i,j} = \mathbf{z}^{(l)}_{c,i,j} + b^{(l)}_c$$

10.1.2 Pooling layers

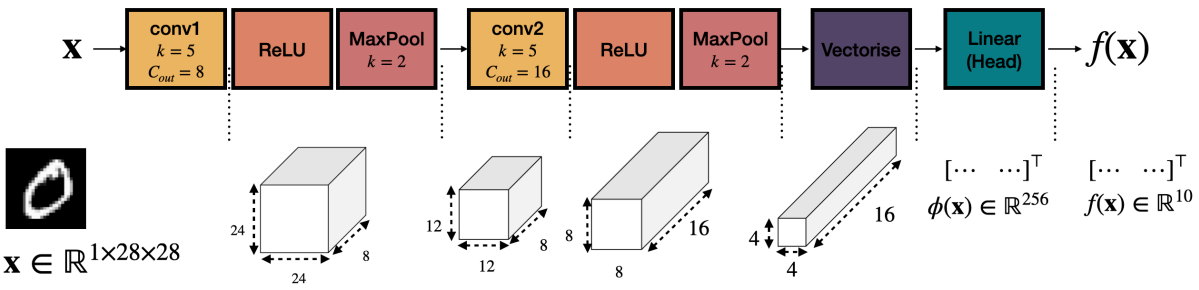
Pooling layers reduce the spatial resolution of the input. Coupled with convolution layers, they can make ConvNets less sensitive to small translations; they do not guarantee complete translation invariance. For max pooling with kernel size and stride k, assuming H and W are divisible by k, if the input is $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$ then the output is $\mathbf{y} \in \mathbb{R}^{C \times H/k \times W/k}$ where

$$\mathbf{y}_{c,i,j} = \max_{\substack{0 \le u < k \\ 0 \le v < k}} \mathbf{x}_{c,\, ik+u,\, jk+v}$$

For $k = 2$ this means that each 2×2 block in each channel is replaced by the maximum value across those 4 elements. It is also possible to perform **average** pooling. In this case each block is replaced by the mean value.

10.1.3 A small ConvNet for MNIST in skorch

Now that we have covered all the constituent parts of a ConvNet, I am going to provide code that trains the ConvNet below on MNIST using skorch and PyTorch. You will be implementing the same ConvNet in the lab, but using numpy instead.



First, let's load in MNIST, scale it to between 0 and 1, and reshape it so data points are 3D tensors (images) instead of vectors. Note that MNIST is greyscale so there is only 1 colour channel.

```
with open("data/mnist.npy", "rb") as f:
    X = np.load(f, allow_pickle=True)                    # (70000,
     ↪  784) uint8
    y = np.load(f, allow_pickle=True).astype(np.int64)   # (70000,)

# Scale to between 0 and 1 and change type for pytorch
X = (X/255.).astype(np.float32)

# Reshape to NCHW: (N, 1, 28, 28)
X = X.reshape(-1, 1, 28, 28)

X_train, y_train = X[:60000], y[:60000]
X_test, y_test = X[60000:], y[60000:]
```

Now let's create the ConvNet in PyTorch. There will be extra functionality for later.

```
# Attempt to fix RNG
torch.manual_seed(1)
np.random.seed(2)

class ConvNetModule(nn.Module):
    def __init__(self, num_classes=10, in_shape=(1, 28, 28)):
        super().__init__()

        # Feature extractor (everything except the final linear
         ↪  classifier)
        self.features = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=5, padding=0),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),

            nn.Conv2d(8, 16, kernel_size=5, padding=0),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
        )

        self.classifier = nn.Linear(256, num_classes)

    def extract_features(self, x):
        """Return flattened feature vector (N, D) for feature
 ↪  extraction."""
        x = self.features(x)
        return torch.flatten(x, 1)

    def forward(self, x):
        z = self.extract_features(x)
        return self.classifier(z)
```

Let's use skorch to train the ConvNet. Unfortunately, this will be slow on a CPU.

Execution note

Training output is not included in this edition. Run this cell in an environment with PyTorch and skorch to train the model. Training on a CPU may take some time.

```
convnet = NeuralNetClassifier(
    ConvNetModule,
    max_epochs=10,
    lr=0.002,
    optimizer=torch.optim.Adam,
    criterion=nn.CrossEntropyLoss,
)
convnet.fit(X_train, y_train)
```

Finally, we'll evaluate the accuracy of this ConvNet on the test set.

Execution note

No test accuracy is reported in this edition. This cell can be evaluated after the preceding training cell has completed.

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

ConvNets typically perform very well on MNIST because they exploit the spatial structure of images. Run the cell above to check the test accuracy of this model.

10.2 Feature extraction

A trained neural network can be used as a feature extractor. This is useful if we have a new, related task that can benefit from the information learnt in the original task, because the network has already learnt good features that can be reused instead of relearnt from scratch. For example, a vision model trained on a large image dataset will have learnt general visual features such as edges, textures, and shapes, which can be reused for a different image classification task. Similarly, a language model trained on large text corpora will have learnt useful linguistic features (e.g. word meaning and sentence structure) that can be reused for a new text classification task.

We can simply compute the features ($\phi(\mathbf{x})$) for the data in our new task, and then apply e.g. logistic regression. In the lab, you will extract features using a network trained on 1 million images to separate out 1000 classes and see that they work rather well.

10.3 Fine-tuning

Fine-tuning goes a step beyond feature extraction by updating (refitting) some or all of the weights of a pretrained neural network so it adapts to a new task. In practice, this is usually done with a smaller learning rate than would be used for training from scratch, and it may involve freezing certain layers (i.e. preventing their parameters from being updated).

The key idea is that pretraining (often on a large, generic dataset) provides a strong starting point: the model has already learned useful representations, so fine-tuning can adapt these representations to a new task with far less data and compute than full training.

This is especially appealing because:

- It's much cheaper (in time, compute, and energy) than training a new model from scratch

- It works well in low-data regimes, where you may not have enough labelled examples to train a deep network reliably
- It often improves performance compared with using the pretrained network as a fixed feature extractor, because the representations can shift toward what the new task needs

When fine-tuning, it is usually necessary to replace the ”head” (the final task-specific layer). For example, a network trained on ImageNet typically ends with a classification head that maps the learned feature vector to 1000 logits (one per ImageNet class). If your new task instead requires, say, 200 classes, you would replace that final linear layer with a new one that outputs 200 logits, and then train that new head (alongside some or all of the previous layers) on your new dataset.

That’s all folks!