Python Refresher 3

In this notebook, we cover:

- Applying functions to all elements in a NumPy array
- Masking
- Pandas

First, import numpy

```
import numpy as np
```

Applying functions to all elements in a NumPy array

Different methods with speed comparisons

Numpy's vectorize function lets you take operations you wrote as functions and apply them to your data set.

```
def square_f(x):
  return x*x
square_v = np.vectorize(square_f)
```

```
size = 1000000

# declaring array
array = np.arange(size)
len(array)
```

```
1000000
```

```
%%timeit
_ = square_v(array)
```

```
223 ms ± 13.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

However, list notation is sometimes even faster for simple functions.

```
%%timeit
_ = np.array([x*x for x in array])
```

```
193 ms ± 36.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

And **for the simplest functions** with numpy arrays such as **multiplication, division, and other simple math operators, you should just do those directly**. Notice how **multiplying an array by itself is the same as multiplying each item by itself but this is literally 100 times faster!** That's because libraries like numpy are built on languages like C and C++ which are much faster than python.

```
%%timeit
_ = array*array
```

```
1.66 ms ± 270 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Masking

First let's get some "real" data. sklearn or **sci-kit learn is a great library for simple statistics and machine learning tools**. In this library, they also provide the **fetch_openml tool** which **allows you to download data** from the website openml.org.

Let's download the MNIST dataset, which contains 70,000 28x28 pixel images of handwritten numerical digits.

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784')
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/datasets/_openml.py:110: UserWarning: A network error occurred while dow
  warn(
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-7-fd32cbaf5565> in <cell line: 0>()
      1 from sklearn.datasets import fetch_openml
----> 2 mnist = fetch_openml('mnist_784')

                          21 frames
/usr/lib/python3.11/socket.py in create_connection(address, timeout, source_address, all_errors)
    846             if source_address:
    847                 sock.bind(source_address)
--> 848             sock.connect(sa)
    849             # Break explicitly a reference cycle
    850             exceptions.clear()

KeyboardInterrupt:
```

← Run this cell block if the OpenML servers are down

```
import tensorflow as tf
from sklearn.utils import Bunch
import pandas as pd

def mnist_tf_keras():
    # Load the MNIST dataset using TensorFlow
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

    # Combine training and testing data
    x_data = tf.concat([x_train, x_test], axis=0)
    y_data = tf.concat([y_train, y_test], axis=0)

    # Flatten the images into vectors
    x_data_flat = x_data.numpy().reshape(len(x_data), -1)

    # Create a DataFrame for the feature data
    feature_names = [f"pixel{i}" for i in range(x_data_flat.shape[1])]
    x_df = pd.DataFrame(x_data_flat, columns=feature_names)

    # Create a DataFrame for the target data
    y_df = pd.DataFrame(y_data.numpy(), columns=["target"]).iloc[:,0].astype(np.uint8)

    # Combine features and target into a single DataFrame
    data_frame = pd.concat([x_df, y_df], axis=0)

    # Prepare a description for metadata
    description = "This dataset contains 70,000 images of handwritten digits (0-9) from the MNIST dataset. Each image is 28x

    # Create a Bunch object similar to the sklearn's fetch_openml output
    return Bunch(
        data=x_df,
        target=y_df,
        feature_names=feature_names,
        frame=data_frame,
        target_names=["digit_class"],
        DESCR=description,
    )


    return mnist

mnist = mnist_tf_keras()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Looking at the MNIST dataset

We can use .keys() to get the names of content within an sklearn openml file.

```
mnist.keys()
```

```
dict_keys(['data', 'target', 'feature_names', 'frame', 'target_names', 'DESCR'])
```

In this case, our openml dataset is a sklearn.utils.Bunch file.

```
type(mnist)
```

```
sklearn.utils._bunch.Bunch
def __init__(**kwargs)
```

```
>>> b.a = 3
>>> b['a']
3
>>> b.c = 6
>>> b['c']
6
```

Data and target are common labels for your input data and the output you would want to get from a model respectively.

```
X, y = mnist["data"], mnist["target"]
```

mnist["data"] is a pandas DataFrame. We'll get to that more later.

```
type(X)
```

```
pandas.core.frame.DataFrame
def __init__(data=None, index: Axes | None=None, columns: Axes | None=None, dtype: Dtype |
None=None, copy: bool | None=None) -> None
```

```
/usr/local/lib/python3.11/dist-packages/pandas/core/frame.py
Two-dimensional, size-mutable, potentially heterogeneous tabular data.

Data structure also contains labeled axes (rows and columns).
Arithmetic operations align on both row and column labels. Can be
thought of as a dict-like container for Series objects. The primary
```

Our pandas dataframe also has keys. Notice how here, all of the **keys are labeled as pixels 1-784**. So **each item** in that dataframe is probably a **784 pixel image**.

```
print(X.keys())
```

```
Index(['pixel0', 'pixel1', 'pixel2', 'pixel3', 'pixel4', 'pixel5', 'pixel6',
       'pixel7', 'pixel8', 'pixel9',
       ...
       'pixel774', 'pixel775', 'pixel776', 'pixel777', 'pixel778', 'pixel779',
       'pixel780', 'pixel781', 'pixel782', 'pixel783'],
      dtype='object', length=784)
```

We can **use .values to extract numeric or other data type values** from our dataframe **as a numpy array**.

```
X_mat = X.values
y_arr = y.values.astype(np.uint8)
```

```
type(X_mat)
```

```
numpy.ndarray
```

Our output/target data has a shape of 70,000 items and our input has a shape of 70,000 items by 784 pixels.

```
y_arr.shape
```

```
(70000,)
```

```
X_mat.shape
```

```
(70000, 784)
```

```
X_mat = X_mat.reshape(X_mat.shape[0], 28, 28)
```

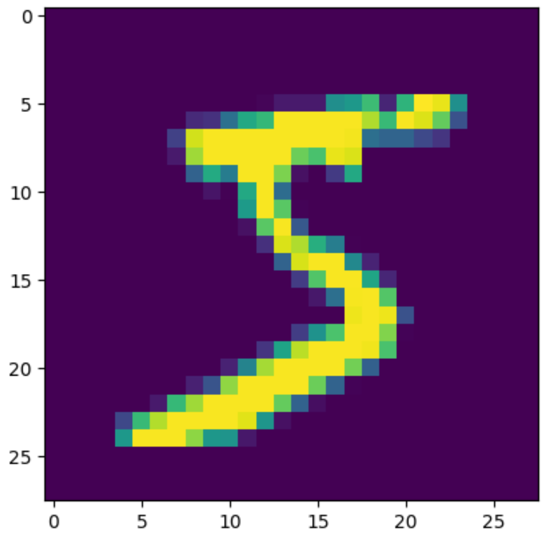
matplotlib is short for mathematical plotting library. It's **a collection of tools for visualising/plotting data**. **pyplot is a library within matplotlib** that has some tools **more specific to just creating plots**. The standard names we use to import these are mpl for matplotlib and plt for matplotlib.pyplot.

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Here we can use the pyplot plot type **imshow to view one of those 784 pixel images**.

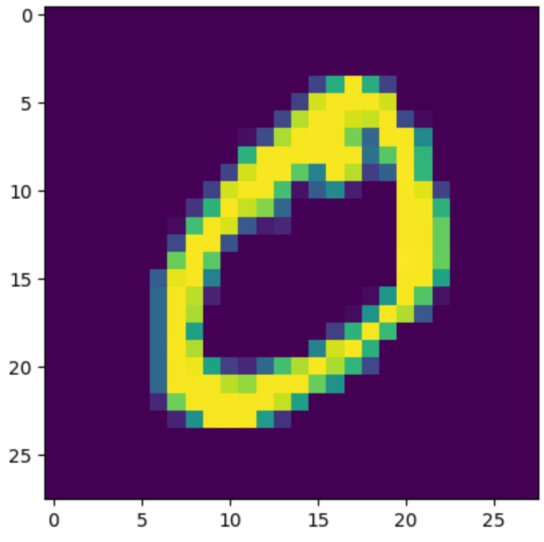
```
plt.imshow(X_mat[0])
```

```
<matplotlib.image.AxesImage at 0x7d5475be5ed0>
```



```
plt.imshow(X_mat[1])
```

```
<matplotlib.image.AxesImage at 0x7d54747f8390>
```



The first two images were a 5 and 0. And sure enough, our first 2 items in our output data are 5 and 0. So the output data is labels of the number image.

```
y_arr[0:2]
```

```
array([5, 0], dtype=uint8)
```

When we perform **logical operations on numpy arrays**, we can **create masks**. For example, writing **y_arr == 3 says: create a new array which is only True at the location of each 3 and False everywhere else**.

```
# this is a mask
y_arr == 3
```

```
array([False, False, False, ..., False, False, False])
```

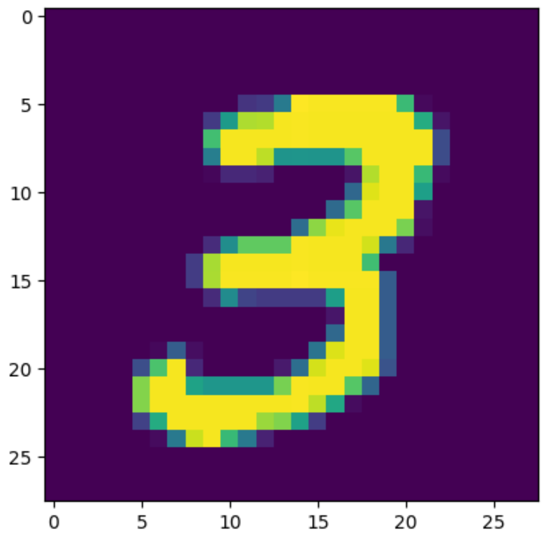
We can also **take that mask array** of True and False values **and pass it to another array like it's a collection of indices**. The result is that we'll
get values from that other array only at the indices that were labeled as True.

```
only_3 = X_mat[y_arr == 3]
```

Because the new array only_3 is the collection of images only at the location where our label was 3, we should only have images of 3's left over.

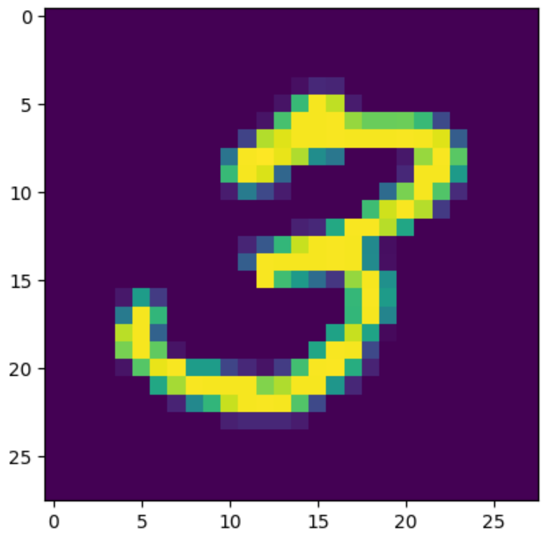
```
plt.imshow(only_3[0])
```

```
<matplotlib.image.AxesImage at 0x7d54748ace90>
```



```
plt.imshow(only_3[42])
```

```
<matplotlib.image.AxesImage at 0x7d54750a5a90>
```



Pandas

What is pandas?

Pandas is a library for data analysis, with a data storing system built
off of Numpy. It can handle data from a wide variety of formats and provides powerful
tools for viewing and manipulating that data.

Some additional resources:
Introduction to pandas - Interactive tutorial created by Google

Pandas documentation - Lots of good info here for both beginner and advanced users.

We import pandas with the shortened name `pd` by running `import pandas as pd`

```
import pandas as pd
```

Let's first create a NumPy array to work with.

We'll use the NumPy function `randint` to create an array of size (4,3)
filled with random integers ranging from 0 to 100

```
np.random.seed(42) # Set random seed for reproducible results
mydata = np.random.randint(low=0, high=101, size=(4,3))
print(mydata)
```

```
[[51 92 14]
 [71 60 20]
 [82 86 74]
 [74 87 99]]
```

We can now directly convert this array into a **pandas DataFrame**

DataFrames are the primary data storing objects used in pandas, and they
can be built from arrays using `pd.DataFrame()` . Let's try passing
our array in and store it in the variable `df`

```
df = pd.DataFrame(mydata)
```

Now let's see what `df` looks like

```
df
```

	0	1	2
0	51	92	14
1	71	60	20
2	82	86	74
3	74	87	99

Next steps: Generate code with df View recommended plots New interactive sheet

Ok, nothing too new, but we see an important difference from
NumPy arrays: **DataFrames include labeled indexes and columns**

In this case, we just have the default labels of 0, 1, 2, etc, but the
power of pandas starts to show when we specify the labels on our data.

Let's imagine that our data represents the scores of four students on
three different midterm exams. We'll call the students Bob, Ann, Steve, and Laura
and store them in a list. We'll also create the midterms:

```
students = ["Bob", "Ann", "Steve", "Laura"]
midterms = ["Midterm 1", "Midterm 2", "Midterm 3"]
```

We can now use these as arguments to `pd.DataFrame()` as follows:

```
df = pd.DataFrame(data=mydata, index=students, columns=midterms)
```

What does `df` look like now?

```
df
```

	Midterm 1	Midterm 2	Midterm 3
Bob	51	92	14
Ann	71	60	20
Steve	82	86	74
Laura	74	87	99

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

We have an informative table showing our data!

Another important property that differentiates DataFrames from NumPy arrays is their ability to store data of different types. For example, a single NumPy array cannot store both integers and strings, but a DataFrame can.

Here's an example to show this: let's create a DataFrame which stores the name, height (in meters), age (in years), and birth month of five people

```
name = ["Bob", "Ann", "Steve", "Laura", "Jack"]
height = [1.78, 1.70, 1.60, 1.83, 1.72]
age = [32, 24, 20, 74, 66]
birth_month = ["September", "April", "June", "September", "March"]
```

There are multiple ways to put multiple lists into a DataFrame. For now, we'll use the format: `pd.DataFrame({"label": data})`, separating our different categories with commas.

```
people_df = pd.DataFrame({"Name": name,
                          "Height (m)": height,
                          "Age (years)": age,
                          "Birth Month": birth_month})
```

```
people_df
```

	Name	Height (m)	Age (years)	Birth Month
0	Bob	1.78	32	September
1	Ann	1.70	24	April
2	Steve	1.60	20	June
3	Laura	1.83	74	September
4	Jack	1.72	66	March

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

Large Data Example

Let's now look at how a large machine learning dataset is stored in a pandas DataFrame and what methods we can use to view and understand the data.

We'll load in the MNIST dataset as before, storing the data samples in `X` and the labels in `y`

```
X, y = mnist["data"], mnist["target"]
y = y.astype(int)
```

Now, what type of object is `X`?

```
type(X)
```

```
pandas.core.frame.DataFrame
def __init__(data=None, index: Axes | None=None, columns: Axes | None=None, dtype: Dtype |
None=None, copy: bool | None=None) -> None
Constructing DataFrame from a dictionary.

>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
```

A pandas DataFrame!

We can check the shape of `X` using the DataFrame property `.shape`

```
X.shape
```

```
(70000, 784)
```

We interpret this as: `X` has 70,000 samples, each with 784 data points.
This is the MNIST dataset, so we have 70,000 different grayscale pictures,
each with 784 pixels of intensities ranging from 0 to 255.

With so many samples, we can't view them all at once! A common way to take a
quick look at the data is to use the `.head(n)` method, which shows the
first *n* samples of the DataFrame.

```
X.head() # shows first 5 rows by default
```

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775	pixel776	pixel7
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	

5 rows × 784 columns

```
X.head(8) # show first 8 rows
```

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775	pixel776	pixel7
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
5	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
6	0	0	0	0	0	0	0	0	0	0	...	0	0	0	
7	0	0	0	0	0	0	0	0	0	0	...	0	0	0	

8 rows × 784 columns

We can look at some of the statistical information of each feature, such as
mean and standard deviation, using `.describe()`

```
X.describe()
```

	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775	pixe
count	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	70000.0	...	70000.000000	70000.000000	70000.00
mean	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.197414	0.099543	0.04
std	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	5.991206	4.256304	2.78
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.00
25%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.00
50%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.00
75%	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.00
max	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	254.000000	254.000000	253.00

8 rows × 784 columns

We can index through DataFrames similarly to NumPy arrays using `.iloc`.

For example, `X.iloc[0]` returns the data points of the first sample.

```
X.iloc[0]
```

	0
pixel0	0
pixel1	0
pixel2	0
pixel3	0
pixel4	0
...	...
pixel779	0
pixel780	0
pixel781	0
pixel782	0
pixel783	0

784 rows × 1 columns

dtype: uint8

To get the actual numerical values in a NumPy array, we use `.values`

For example, `X.values[0]` returns a NumPy array containing the pixel intensity values of the first sample, which we'll store in `tmp`

```
tmp = X.values[0]
print(type(tmp))
print(tmp[210:230]) # Look at pixels 210 through 229
```

```
<class 'numpy.ndarray'>
[253 253 253 251  93  82  82  56  39   0   0   0   0   0   0   0   0   0
   0   0]
```

We can view the columns of a DataFrame using `.columns`

```
X.columns
```

```
Index(['pixel0', 'pixel1', 'pixel2', 'pixel3', 'pixel4', 'pixel5', 'pixel6',
       'pixel7', 'pixel8', 'pixel9',
       ...
       'pixel774', 'pixel775', 'pixel776', 'pixel777', 'pixel778', 'pixel779',
       'pixel780', 'pixel781', 'pixel782', 'pixel783'],
      dtype='object', length=784)
```

You can access the contents of a column simply using the column's name.

For example, we can look at the intensities of pixel1 across different samples by running `X['pixel1']` or `X.pixel1`

```
X['pixel1']
```

	pixel1
0	0
1	0
2	0
3	0
4	0
...	...
69995	0
69996	0
69997	0
69998	0
69999	0

70000 rows × 1 columns

dtype: uint8

```
X.pixel1
```

	pixel1
0	0
1	0
2	0
3	0
4	0
...	...
69995	0
69996	0
69997	0
69998	0
69999	0

70000 rows × 1 columns

dtype: uint8

Activity

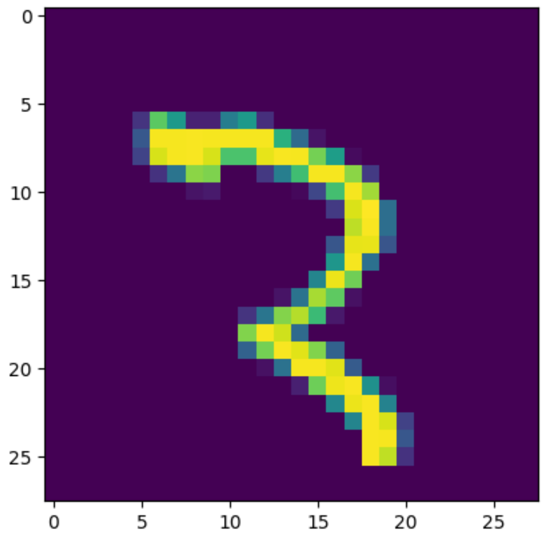
For this activity we will continue to use the mnist datasets we've been working with. This actvitivy will involve 6 steps:

1. Access the data and target values from the mnist data set and save it to new variables called `data` and `target`. **Do not reuse the names X, and y for those variables**.
2. Use the `iloc` function to get the item at index 500 from from `data` and `target` and save them to new variables called `data_500` and `target_500`
3. Use `.values` to convert `data_500` to a numpy array called `d_500_vals`.
4. Use `int(target_500)` to convert `target_500` from a string to an integer called `t_500_vals`.
5. Use `.reshape()` to reshape `d_500_vals` to have shape (28, 28).
6. Finally, use `plt.imshow()` and `print()` to display `d_500_vals` and `t_500_vals` in one cell.

```
#Generate the data here:
minist = mnist_tf_keras()
data, target = minist['data'], minist['target']
data_500, target_500 = data.iloc[500], target.iloc[500]
```

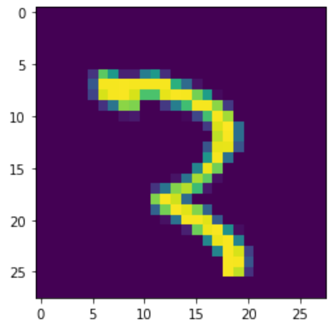
```
d_500_vals = data_500.values
t_500_vals = int(target_500)
d_500_vals = d_500_vals.reshape(28, 28)


#Display d_500_vals and t_500_vals here:
plt.imshow(d_500_vals)
print(t_500_vals)
```

```
3
```



```
#EXAMPLE OUTPUT
```

```
3
```



Exercise 1

File name convention: For group 42 and memebers Richard Stallman and Linus Torvalds it would be:
"Exercise1_Goup42_Stallman_Torvalds.pdf".

Submission via blackboard.

```
group_name = "Group 7"
group_members = ["Xiao Zou",
                 "Yang Yu",
                 "Handa Shi"]
```

Chapter 3 – Classification

Setup

First, let's **import a few common modules**, ensure MatplotLib plots figures inline, and prepare a function to save the figures. We also check that Python 3.5 or later is installed (although Python 2.x may work, it is deprecated so we strongly recommend you use Python 3 instead), as well as Scikit-Learn ≥0.20.

```
# Python ≥3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

MNIST

Next, let's **import the MNIST dataset** we saw in PythonRefreshers_3. As a reminder, this is a dataset of **70,000 handwritten characters stored in 28x28 pixel images**.

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
mnist.keys()
```

⟵ Run this cell block if the OpenML servers are down

```
import tensorflow as tf
from sklearn.utils import Bunch
import pandas as pd

def mnist_tf_keras():
    # Load the MNIST dataset using TensorFlow
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

    # Combine training and testing data
    # Combine training and testing data
    x_data = tf.concat([x_train, x_test], axis=0)
    y_data = tf.concat([y_train, y_test], axis=0)

    # Flatten the images into vectors
    x_data_flat = x_data.numpy().reshape(len(x_data), -1)

    # Create a DataFrame for the feature data
    feature_names = [f"pixel{i}" for i in range(x_data_flat.shape[1])]
    x_df = pd.DataFrame(x_data_flat, columns=feature_names)

    # Create a DataFrame for the target data
    y_df = pd.DataFrame(y_data.numpy(), columns=["target"]).iloc[:,0].astype(np.uint8)

    # Combine features and target into a single DataFrame
    data_frame = pd.concat([x_df, y_df], axis=0)

    # Prepare a description for metadata
    description = "This dataset contains 70,000 images of handwritten digits (0-9) from the MNIST dataset. Each image is 28x2

    # Create a Bunch object similar to the sklearn's fetch_openml output
    return Bunch(
        data=x_df,
        target=y_df,
        feature_names=feature_names,
        frame=data_frame,
        target_names=["digit_class"],
        DESCR=description,
    )


    return mnist

mnist = mnist_tf_keras()
```

If the OpenML servers worked then continue from here

We'll need the `data` of 28x28 pixel images and `target` of number labels 0-9 for this exercise.

```
X, y = mnist["data"], mnist["target"]
```

X is a pandas dataframe but we'll eventually want to work with a numpy array of the pixel values, which we can get later using `X.values`.

```
print(type(X))
print(X.shape)
```

```
<class 'pandas.core.frame.DataFrame'>
(70000, 784)
```

```
y.shape
```

```
(70000,)
```

```
# the images are 28x28 pixels
28 * 28
```

```
784
```

We can create a function that implements the `imshow` function from matplotlib so that we can visualize the handwritten characters.

```
import matplotlib as mpl
import matplotlib.pyplot as plt

def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = mpl.cm.binary,
               interpolation="nearest")
    plt.axis("off")
```

Here's an example of how we would use the function we just made. Note how it requires a numpy array which we get using `.values`. We can use `%matplotlib inline` to get our plots to show up below the current cell without having to repeatedly use `plt.show()`.

```
%matplotlib inline

# to get the values of the dataframe X, call X.values.
# X.values[0] gives the pixel values for the first image
plot_digit(X.values[0])
```

Don't worry about how the function `plot_digits` below works. This is just one way we can visualize a subset of our X values (handwritten characters).

```
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = mpl.cm.binary, **options)
    plt.axis("off")


plt.figure(figsize=(9,9))
example_images = X[:100].values
plot_digits(example_images, images_per_row=10)
plt.show()
```

Notice how **the type of our labels is a string by default**, but we can convert this to a collection of integers instead.

```
print(type(y[0]))
print(y[0])
```

```
<class 'numpy.uint8'>
5
```

```
y = y.astype(np.uint8)
type(y[0])
```

```
numpy.uint8
```

In chapter 1 you were introduced to the idea of a **train** and **test** set. Since **this dataset is already randomly shuffled**, we can get our train and test sets by just selecting the **first 60,000 characters and labels for training** and the **last 10,000 for testing**.

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Binary classifier

Let's train a classifier to predict whether a given digit is a 5 or not.

To start with, we'll create a sub-set of our data that is either True or False depending on whether the label value is 5. **Many functions can treat True and False as 0 or 1 values**.

```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

```
print(y_train_5[:10]) # Visualize first 10 samples of the training set
```

```
0     True
1    False
2    False
3    False
4    False
5    False
6    False
7    False
8    False
9    False
Name: target, dtype: bool
```

```
print(y_test_5[:10]) # Visualize first 10 samples of the test set
```

```
60000    False
60001    False
60002    False
60003    False
60004    False
60005    False
60006    False
60007    False
60008     True
60009    False
Name: target, dtype: bool
```

Task 1: SGD Classifier

SGD (Stochastic Gradient Descent) Classifier is a linear model which gradually attempts to decrease its loss by evaluating the way that changes made to the model weights affect the gradient of the loss value.

1) Create an SGD Classifier `sgd_clf` with the following hyperparameters:

- `max_iter=1000` : This sets the maximum number of times the model will see the whole dataset during training.
- `tol=1e-3` : This sets a stopping condition that, when your loss is greater than your best loss minus tol, it will stop training.
- `random_state=42` : This will ensure that you will get the same results every time you run the training algorithm.

2) Fit the classifier by calling its `fit` function with `X_train` and `y_train_5` as the parameters. This might take a while.

```
from sklearn.linear_model import SGDClassifier

# your code goes below
sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)

# fit the sgd_clf
sgd_clf.fit(X_train, y_train_5)
```

```
▾     SGDClassifier
SGDClassifier(random_state=42)
```

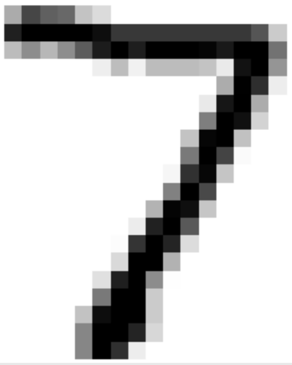
```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
sgd_clf.fit(X_train, y_train_5)
```

```
▾     SGDClassifier
SGDClassifier(random_state=42)
```

```
# let's test the model on an example. Feel free to play around and try more.

sgd_clf.predict(X_test[0:1])
```

```
array([False])
```

```
plot_digit(X_test[0:1].values)
```



Task 2: Cross Validation of SGD Classifier

1) Calculate the cross-validation score of the `sgd_clf` classifier with the train data. Use `cv=3` and `scoring="accuracy"`.

2) a) Explain what cross validation is. Hint: check chapter 2 of the textbook.

```
from sklearn.model_selection import cross_val_score


# 1)
# your code goes below
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.95035, 0.96035, 0.9604 ])
```

```
# cross val score



# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your code goes above this
```

Double click this:

Enter your answer to 2) a) below this:

Cross validation is a technique in machine learning to evaluate the model performance. It divides the dataset into multiple folds, using one of these folds as a validation set, and training the model on the remaining folds. This process is repeated multiple times, each time using a different fold as the validation set.

Here is some code for a stratified k-fold cross validation

2) b) Explain stratified cross validation. Hint: check chapter 2 of the textbook.

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train.values[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train.values[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))
```

```
0.9669
0.91625
0.96785
```

```
# your answer goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
```

Double click this:

Enter your answer to 2) b) below this:

Stratified cross validation ensures that each fold of the cross-validation process maintains the same class distribution as the entire dataset. The dataset is divided into k folds while maintaining the proportion of classes in each fold.

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your answer goes above this
```

Never 5 Classifier

This classifier always predicts 0 or False. Don't worry too much about the code below for now. The important things are that **this is a class or type of object in python that has an appropriate type to be used by sklearn functions**. What it's doing is just **creating a list of 0's of the same size of your input** or X values and returning that as its solutions.

```
from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

```
never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.91125, 0.90855, 0.90915])
```

Task 3

Explain why the "never 5 classifier" still has such "high" scores. Hint: think about True (1) and False (0) labels.

```
# your answer goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
```

Double click this:

Enter your answer to 3) below this:

Because dataset is highly imbalanced, with True(1) being a small fraction of the total labels, the "never 5 classifier" will correctly predict the majority of the labels (False labels) just by output 0.

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your answer goes above this
```

Back to the real classifier: `sgd_clf`

```
from sklearn.model_selection import cross_val_predict
```

```
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Task 4

Calculate the confusion matrix for

- predicted values (true y values and predicted y values)
- perfect predictions (Hint: use only true values)

Explain what the entries of the confusion matrix mean.

```
from sklearn.metrics import confusion_matrix


# your code goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓


# confusion matrix for predicted values
cm = confusion_matrix(y_train_5, y_train_pred)
print(cm)
```

```
[[53892   687]
 [ 1891  3530]]
```

```
# confusion matrix for perfect predictions
cm2 = confusion_matrix(y_train_5, y_train_5)
print(cm2)
```

```
[[54579     0]
 [    0  5421]]
```

Double click this:

Enter your exaplanation for 4) below this:

True negatives: 53892; False Positives: 687; False negatives: 1891; True positives: 3530

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your code goes above this
```

Task 5

Calculate precision and recall scores for training and predicted values.

Precision = TP / (TP + FP)

Recall = TP / (TP + FN)

```
from sklearn.metrics import precision_score, recall_score


# your code goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓


pre = precision_score(y_train_5, y_train_pred)
rec = recall_score(y_train_5, y_train_pred)
print(pre)
print(rec)
```

```
0.8370879772350012
0.6511713705958311
```

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your code goes above this
```

Task 6

Using the values from the confusion matrix, calculate

- the rate of false positives, true positives, false negatives, true negatives
- accuracy (TP + TN) / (TP + TN + FN + FP)
- precision

```
# your code goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓


tn = cm[0,0]
tp = cm[1,1]
fn = cm[0,1]
```

```
fp = cm[1,0]
fpr = fp / (fp + tn)
tpr = tp / (tp + fn)
fnr = fn / (tp + fn)
tnr = tn / (fp + tn)
accuracy = (tp + tn)/(tp +  tn + fn + fp)
precision = (tp) / (tp + fn)
print(fpr, tpr, fnr, tnr)
print(accuracy)
print(precision)
```

```
0.033899216607210084 0.8370879772350012 0.1629120227649988 0.9661007833927899
0.9570333333333333
0.8370879772350012
```

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your code goes above this
```

Task 7

Calculate the f1 score.
F1 = (2 * precision * recall) / (precision + recall)

```
from sklearn.metrics import f1_score


# your code goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓


f1 = f1_score(y_train_5, y_train_pred)
print(f1)
```

```
0.7325171197343847
```

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your code goes above this
```

Task 8

Explain what the scores in task 6 and 7 (accuracy, precision, f1) describe.
Do not just copy formulas; explain their meaning.

```
# your answer goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
```

Double click this:

Enter your answer to 8) below this:

1. Accuracy measures how often the classifier is correct overall. It's the proportion of all predictions (both positive and negative) that were correct.
2. Precision measures how many of the predictions for the positive class are actually correct.
3. Recall measures how many of the actual positive instances the classifier correctly identifies.
4. The F1 score is a metric that balances the trade-off between the precision and recall.

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your answer goes above this
```

Setting a Threshold on the Prediction

Models will often have some mathematical function which determines a so-called
confidence score. This score can be thought of as a likelihood of predicting
a particular value correctly. Sometimes these are percentages and other times
these are some continuous value.

```
some_digit = X.values[0]
```

```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

```
/usr/local/lib/python3.11/dist-packages/sklearn/utils/validation.py:2739: UserWarning: X does not have valid feature nam
  warnings.warn(
array([2164.22030239])
```

We can also set a **threshold for confidence** scores that says we **only consider a prediction if it's made with enough "confidence"**.

```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

```
array([ True])
```

```
threshold = 8000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

```
array([False])
```

These **confidence scores can also be generated using** the decision_function method of **cross_val_predict()**.

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

```
y_scores
```

```
array([  1200.93051237, -26883.79202424, -33072.03475406, ...,
        13272.12718981,  -7258.47203373, -16877.50840447])
```

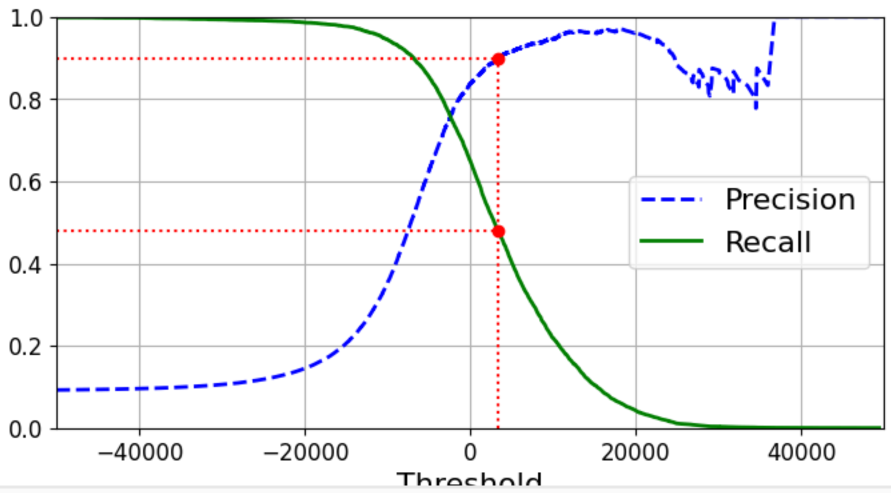
The **precision_recall_curve compares precision and recal values at certain confidence thresholds**. That is, if we require the model make a prediction with a certain degree of confidence, how does that affect our precision and recall.

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.legend(loc="center right", fontsize=16) # Not shown in the book
    plt.xlabel("Threshold", fontsize=16)        # Not shown
    plt.grid(True)                              # Not shown
    plt.axis([-50000, 50000, 0, 1])             # Not shown
```

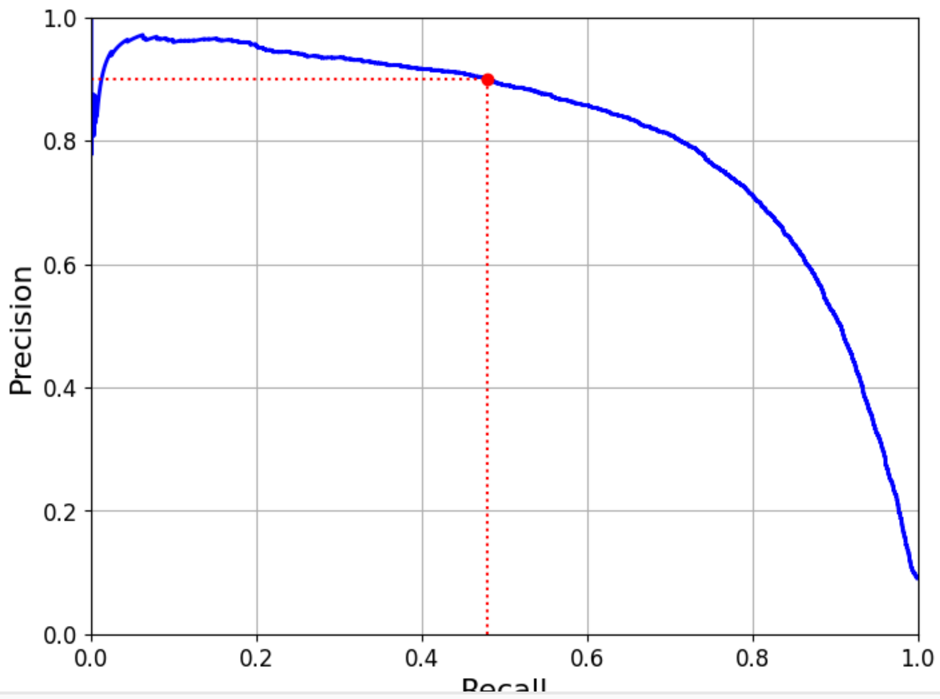
```
recall_90_precision = recalls[np.argmax(precisions >= 0.90)]
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
```

```
plt.figure(figsize=(8, 4))                                                                  # Not shown
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.plot([threshold_90_precision, threshold_90_precision], [0., 0.9], "r:")                 # Not shown
plt.plot([-50000, threshold_90_precision], [0.9, 0.9], "r:")                                # Not shown
plt.plot([-50000, threshold_90_precision], [recall_90_precision, recall_90_precision], "r:")# Not shown
plt.plot([threshold_90_precision], [0.9], "ro")                                             # Not shown
plt.plot([threshold_90_precision], [recall_90_precision], "ro")                             # Not shown
plt.show()
```



Another way to evaluate precision and recall is just plotting them against one another. **Notice the dashed red box. The right side shows what our recall performance is at a precision of 0.9 out of a maximum of 1**.

```
def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.plot([recall_90_precision, recall_90_precision], [0., 0.9], "r:")
plt.plot([0.0, recall_90_precision], [0.9, 0.9], "r:")
plt.plot([recall_90_precision], [0.9], "ro")
plt.show()
```



Let's check what our threshold value is at 0.9 precision.

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
```

```
threshold_90_precision
```

```
3370.0194991439557
```

We can also analyze our performance at the region above that threshold.

```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

```
precision_score(y_train_5, y_train_pred_90)
```

```
0.9000345901072293
```

```
recall_score(y_train_5, y_train_pred_90)
```

```
0.4799852425751706
```

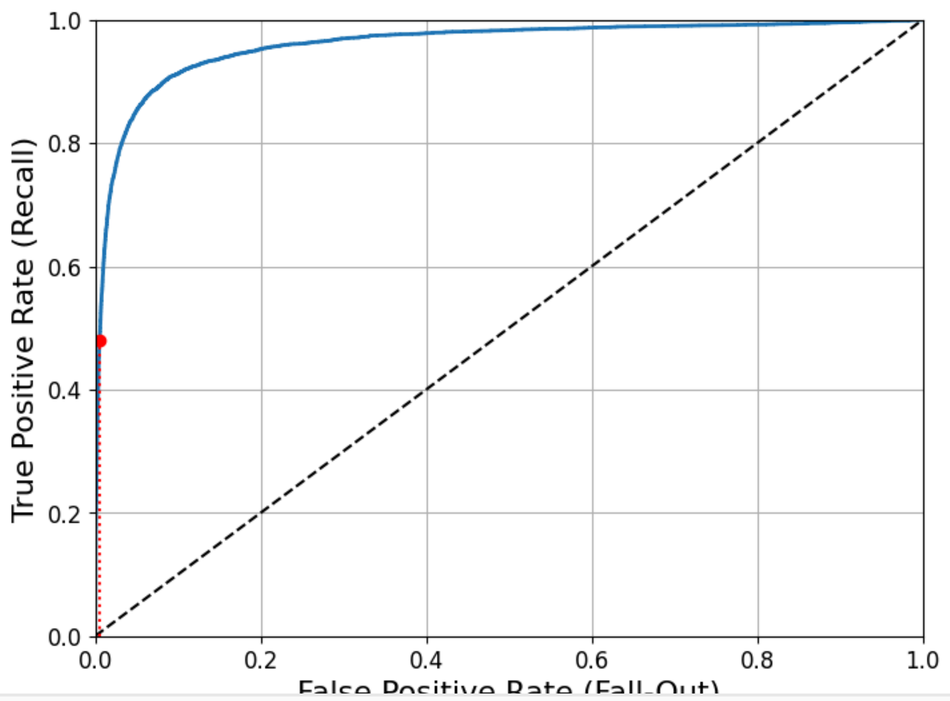
ROC curves

We can now go on to calculate the Receiver Operating Characteristic (ROC) curve.
For a good explanation see also here or the Statistics_1 notebook.

Basically instead of plotting measures like precision and recall as a function of the threshold, we plot the true positive rate (recall) vs the false positive rate (fall-out).

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (Fall-Out)', fontsize=16)
    plt.ylabel('True Positive Rate (Recall)', fontsize=16)
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)
fpr_90 = fpr[np.argmax(tpr >= recall_90_precision)]
plt.plot([fpr_90, fpr_90], [0., recall_90_precision], "r:")
plt.plot([0.0, fpr_90], [recall_90_precision, recall_90_precision], "r:")
plt.plot([fpr_90], [recall_90_precision], "ro")
plt.show()
```



The area under the curve (auc) is a measure of how good the model is.

```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_train_5, y_scores)
```

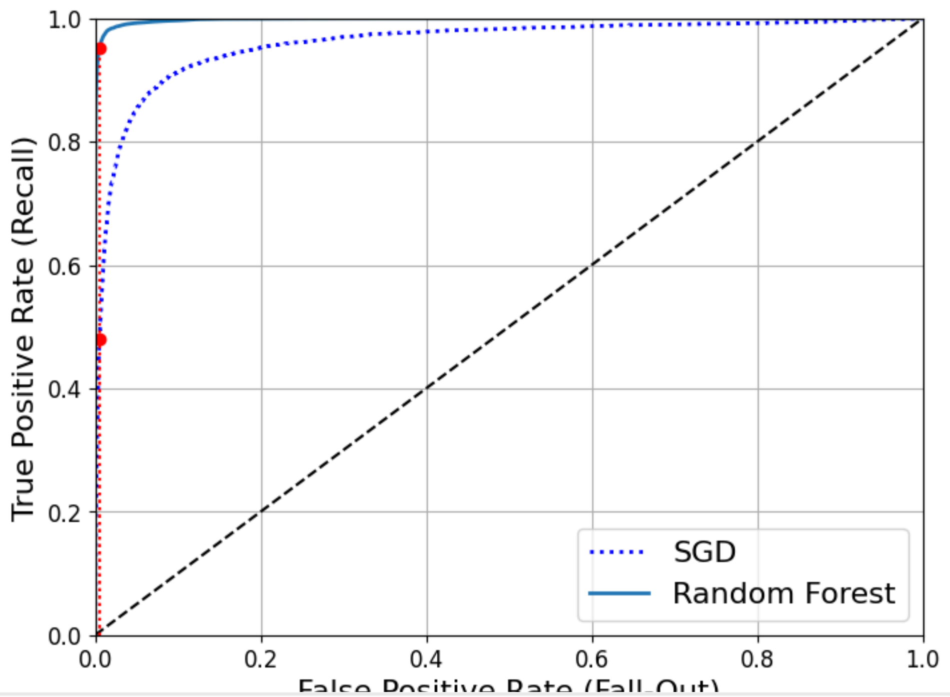
```
0.9604938554008616
```

Let's compare the SGD model with another model. This other model is called a "random forest" classifier. **We'll cover these more in depth in the coming weeks** but, **for now, just think of this as some other model of interest**.

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(n_estimators=100, random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

```
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

```
recall_for_forest = tpr_forest[np.argmax(fpr_forest >= fpr_90)]

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, "b:", linewidth=2, label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.plot([fpr_90, fpr_90], [0., recall_90_precision], "r:")
plt.plot([0.0, fpr_90], [recall_90_precision, recall_90_precision], "r:")
plt.plot([fpr_90], [recall_90_precision], "ro")
plt.plot([fpr_90, fpr_90], [0., recall_for_forest], "r:")
plt.plot([fpr_90], [recall_for_forest], "ro")
plt.grid(True)
plt.legend(loc="lower right", fontsize=16)
plt.show()
```



```
roc_auc_score(y_train_5, y_scores_forest)
```

```
0.9983436731328145
```

```
y_train_pred_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3)
precision_score(y_train_5, y_train_pred_forest)
```

```
0.9905083315756169
```

```
recall_score(y_train_5, y_train_pred_forest)
```

```
0.8662608374838591
```

Task 9

Interpret the results from above and answer the following questions:

- Which model performed better? SGD or Random Forests
- Explain what information above you used to reach this conclusion.

```
# your code goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓
```

Double click this:

Enter your answer to 9) below this:

Random Forests performed better. Both precision and recall are higher for the Random Forest, it means this model not only identifies positive instances better but also minimizes false alarms effectively. Additionally, a higher ROC-AUC (0.99 vs. 0.96) implies that the Random Forests model has a better trade-off between true positive and false positive rates across all thresholds.

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your code goes above this
```

Textbook exercises

Exercise 1

Feel free to play around with this question.
Try to build a different classifier.

Hint: the KNeighborsClassifier works quite well for this task. Good hyperparameters are `{'n_neighbors': 4, 'weights': 'distance'}`

```
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier

# your code goes below
# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

kn_clf = KNeighborsClassifier(n_neighbors=4, weights='distance')
kn_clf.fit(X_train, y_train_5)
y_pred_5 = kn_clf.predict(X_test)
acc_kn = accuracy_score(y_test_5, y_pred_5)
print(acc_kn)
```

```
0.9939
```

```
# ↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑
# your code goes above this
```

Exercise 2 (Optional)

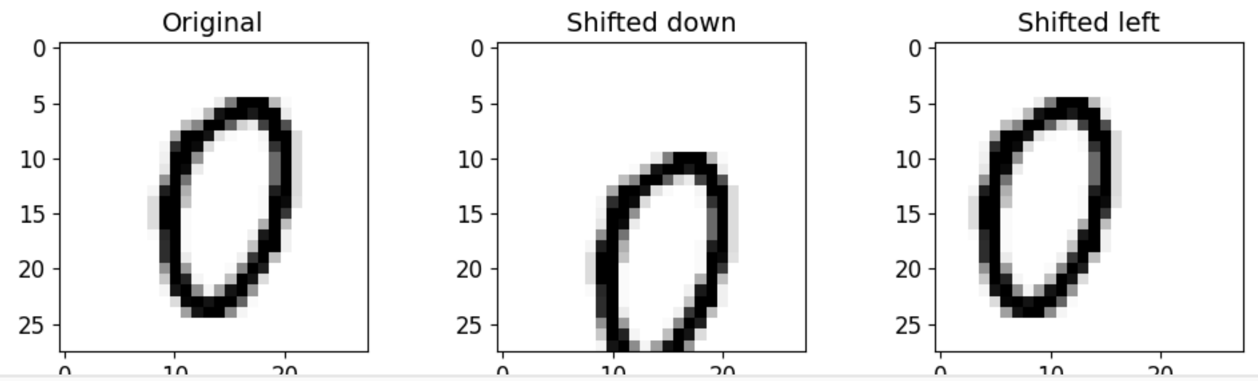
Write a function that can shift an MNIST image in any direction (left, right, up, or down) by five pixels. Then, for each image in the training set, create four shifted copies (one per direction) and add them to the training set. Finally, train your best model on this expanded training set and measure its accuracy on the test set. You should observe that your model performs even better now! This technique of artificially growing the training set is called data augmentation or training set expansion.

Exercise 2 Solution

```
from scipy.ndimage.interpolation import shift
```

```
<ipython-input-83-c50379b1544a>:1: DeprecationWarning: Please import `shift` from the `scipy.ndimage` namespace; the `sc
  from scipy.ndimage.interpolation import shift
```

```
def shift_image(image, dx, dy):
    image = image.reshape((28, 28))
    shifted_image = shift(image, [dy, dx], cval=0, mode="constant")
    return shifted_image.reshape([-1])


image = X_train.values[1000]
shifted_image_down = shift_image(image, 0, 5)
shifted_image_left = shift_image(image, -5, 0)

plt.figure(figsize=(12,3))
plt.subplot(131)
plt.title("Original", fontsize=14)
plt.imshow(image.reshape(28, 28), interpolation="nearest", cmap="Greys")
plt.subplot(132)
plt.title("Shifted down", fontsize=14)
plt.imshow(shifted_image_down.reshape(28, 28), interpolation="nearest", cmap="Greys")
plt.subplot(133)
plt.title("Shifted left", fontsize=14)
plt.imshow(shifted_image_left.reshape(28, 28), interpolation="nearest", cmap="Greys")
plt.show()
```



```
X_train_augmented = [image for image in X_train.values]
y_train_augmented = [label for label in y_train.astype(np.int8)]

for dx, dy in ((1, 0), (-1, 0), (0, 1), (0, -1)):
    for image, label in zip(X_train.values, y_train.astype(np.int8)):
        X_train_augmented.append(shift_image(image, dx, dy))
        y_train_augmented.append(label)

X_train_augmented = np.array(X_train_augmented)
y_train_augmented = np.array(y_train_augmented)


shuffle_idx = np.random.permutation(len(X_train_augmented))
X_train_augmented = X_train_augmented[shuffle_idx]
y_train_augmented = y_train_augmented[shuffle_idx]


from sklearn.model_selection import GridSearchCV


param_grid = [{
"n_neighbors": (3,4),
"weights": ["uniform", "distance"]
}]


knn_clf = KNeighborsClassifier()


grid_search = GridSearchCV(knn_clf, param_grid)
grid_search.fit(X_train.values, y_train.astype(np.int8))
```

```
---------------------------------------------------------------------------
```