

0-Introduction

May 7, 2024

1 Advanced Python and Machine Learning Course

With this course we provide an overview of important Machine Learning concepts and introduce more advanced **Python 3** programming topics. In the following notebooks, we will cover the areas of **regression**, **classification**, and **knowledge discovery** as well as some of the most relevant Python libraries for Machine Learning. Of course, we won't be able to provide an exhaustive presentation of all topics and their mathematical depth. The goal of this course is to give you a practical introduction with some real-world examples. There might be some additional topics, objects or commands you later find useful but have not seen in this course. However, we believe that you really become "fluent" in a programming language and the different Machine Learning tools by trying to implement your own projects and learn from difficulties you encounter along the way.

1.0.1 Small overview of working with Jupyter Lab

In this course, we make use of Jupyter Notebooks which allow us to combine executable code with text in one document, increasing code-readibility and documentation.

In Jupyter Notebooks, you have different kinds of **cells** you can make use of: **Code**, **Markdown**, and **Raw**. In order to select a cell for execution you can just click on the relevant part of the notebook and a blue bar on the left will highlight the currently selected cell like in the image below.



Try to run the following code cell with `Shift+Enter`!

```
[1]: print("It works!")
```

It works!

You can stop the execution of a cell with `Esc` in the panel!

1.0.2 Markdown mode

Try to add a cell below, change to **Markdown** mode and write a note!

1. Add a cell with + in the panel above or *Esc+B*
2. Change to **Markdown** in the panel above (drop-down list) or *Esc+M*
3. As before, run the cell with ↵ in the panel above or *Shift+Enter*!

You can make notes inside the notebooks if you like! In order to edit a Markdown cell double-click somewhere in the cell!

In Markdown mode you can use the following special characters:

2 Title

2.1 Subtitle

- bullet point 1
- bullet point 2

italic font bold

highlighted code

*** (that's a separator line)

You can also use L^AT_EX if you want, like $\sum_{k=0}^{\infty} q^k = \frac{1}{1-q}$ (double-click here to see how it is used).

2.1.1 Code mode

Creating a **Code** cell works similarly. You can make use of it later on!

1. Add a cell with + in the panel above or *Esc+B*
 2. Change to **Code** in the panel above (drop-down list) or *Esc+Y*
 3. Again, run the cell with ↵ in the panel above or *Shift+Enter*!
-

2.1.2 Some Remarks

- Delete a cell with right-click on the cell and the “Delete Cells” option
 - **Be careful which cells you delete** because it might not be possible to retrieve them!
 - In case, you accidentally deleted something try *Esc+Z* to retrieve a cell or *Ctrl+Z* to retrieve changes within a cell
 - Save your notebooks with the *save icon* in the panel above or with *Ctrl+S!* **Try to save after every bigger change!** Depending on your machine, Jupyter Lab might crash from time to time.
 - Try not to open too many windows in Jupyter Lab. This might lead to a crash sometimes.
-

2.1.3 Final Test

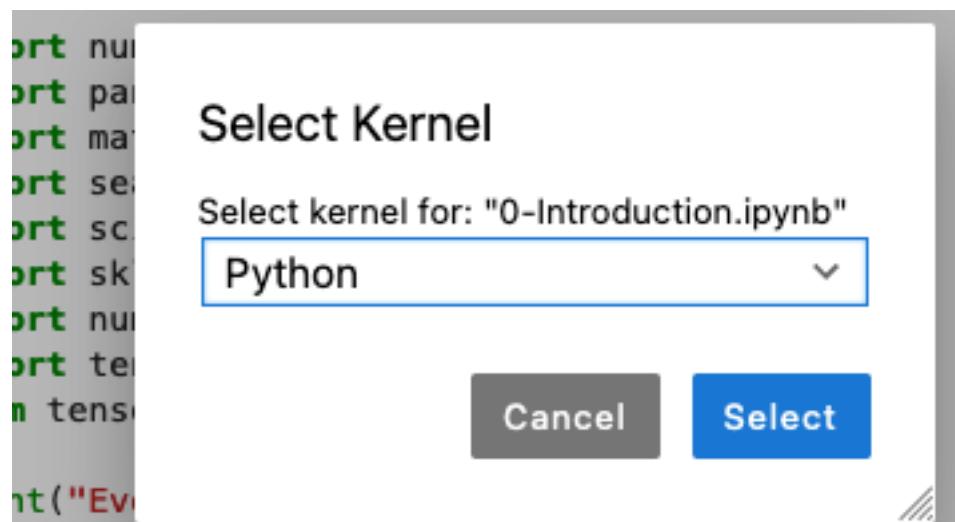
Let's see whether your setup is ready for the course. Depending on whether you (1) *use Noto* or (2) have *your own installation*, please note the following:

(1) If you are a Noto user you will need to make sure that you have the right *kernel* selected. You can do this by executing the following steps.

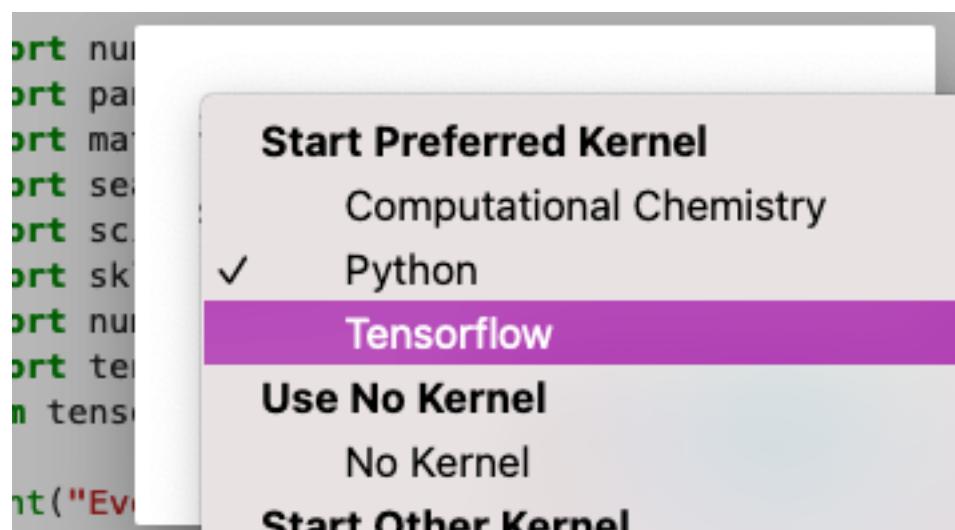
Step 1: Click in the top-right corner on the kernel name.



Step 2: A window pops up displaying the currently selected kernel. Click on the drop-down menu.



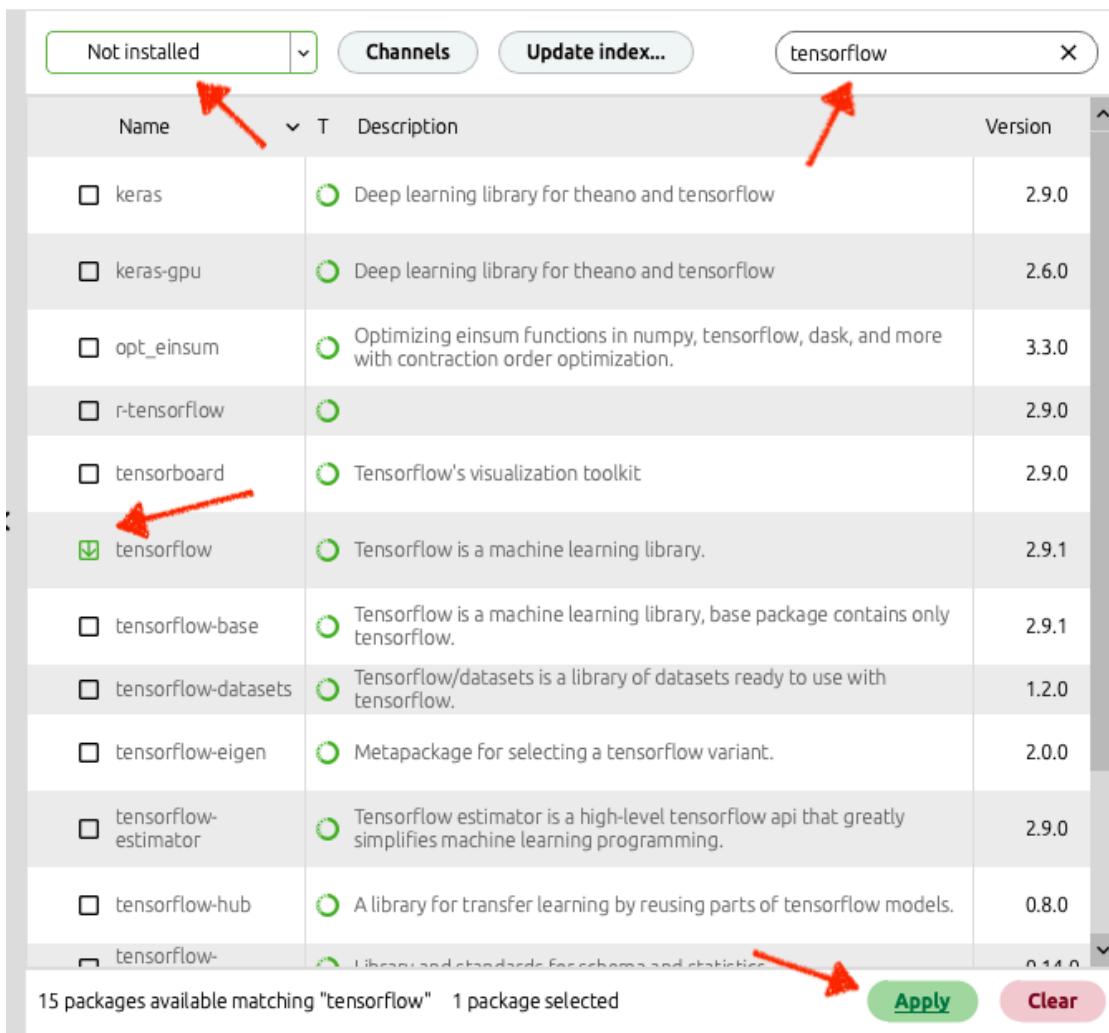
Step 3: Select the kernel called Tensorflow and click Select.



Step 4: Now, the top-right corner kernel name should read **Tensorflow**. Whenever something is not running properly, check whether the **Tensorflow** kernel is selected!



(2) If you have your own local installation you will need to install TensorFlow yourself. For this, you can follow the instructions provided in the YouTube video! In particular, you can install TensorFlow (as well as any other potentially missing library) in the Anaconda Navigator:



Note that fetching specifications and downloading the package might take some time (in some cases up to an hour)!

Please execute the cell at the bottom to check whether all libraries are available which we will use later on.

```
[2]: import numpy
import pandas
import matplotlib
import seaborn
import scipy
import sklearn
import numba
import deepsmiles
import rdkit
import xarray
import tensorflow
from tensorflow import keras

print("Everything is working!")
```

2024-04-29 10:28:29.672527: I tensorflow/core/platform/cpu_feature_guard.cc:182]

This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: SSE4.1 SSE4.2, in other operations, rebuild TensorFlow with the appropriate compiler flags.

Everything is working!

1-Python_Concepts

May 7, 2024

1 1. Python Concepts

We start by reviewing some more advanced Python concepts which might prove useful in implementing your Machine Learning projects. In this notebook we focus on the following topics which, in particular, review some concepts related to functions in Python:

- f-strings
- subplots with Matplotlib
- type annotations and type hints
- eval function
- lambda functions
- generators
- built-in functions map and filter
- *args and **kwargs
- decorators
- speeding up function executions with Numba

Keywords: `f"String"`, `find`, `plt.subplots`, `plt.figure.add_subplot`, `np.ravel`, `eval`, `lambda`,
`x: ...`, `yield`, `map`, `filter`, `*args`, `**kwargs`, `@decorator`, `@jit`, `%timeit`

1.1 Important Python Libraries

There are several useful Python libraries for scientific computing and Machine Learning in Python.

- **NumPy** extends basic Python data structures (like lists) and provides efficient numerical functions for computations with large data arrays.
- **SciPy** builds on NumPy and provides extended functionality for numerical and statistical methods.
- **Pandas** incorporates data frames (similar to programming language R) and allows for more statistical analyses.
- **Matplotlib** is the standard plotting library in Python.
- **Seaborn** builds on Matplotlib and, in particular, extends Pandas functionality to create appealing plots.
- **Scikit-learn** is a powerful machine learning library that provides a wide range of learning algorithms. It builds upon NumPy, SciPy, and Matplotlib.

- TensorFlow is a library for efficient Machine Learning implementation and a standard library for Deep Learning.
- Keras is a high-level Deep Learning library built on top of TensorFlow, which simplifies creating, training, and evaluating neural networks.
- PyTorch is another standard library for Deep Learning adhering more closely to basic Python principles.

In this course, we will mostly work with NumPy, Matplotlib, scikit-learn, and TensorFlow / Keras. You will be able to load these libraries like this

```
[1]: import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model
```

1.2 f-Strings

Recommended string formatting since Python 3.6:

```
[2]: float_var = 3.141592

print(f'String formatting allows including variables like {float_var:.2f}.')
```

String formatting allows including variables like 3.14.

And while we are talking about strings: When you want to find a substring in a string, you can use

```
[5]: example_string = f'String formatting allows including variables like {float_var:
    .2f}.'

search_string = 'variable'

index_of_search = example_string.find(search_string)

search_string_len = len(search_string)

print(f"Starting from index {index_of_search}: '{example_string[index_of_search:
    index_of_search+search_string_len]}")'
```

Starting from index 35: 'variable'

```
[6]: example_string.find('missing')
```

```
[6]: -1
```

1.3 Subplots with Matplotlib

Let us create some dummy images:

```
[7]: import numpy as np
import matplotlib.pyplot as plt

dummy_images = [[[0,0,0,0,0],
                 [0,1,1,1,0],
                 [0,0,1,0,0],
                 [0,0,1,0,0],
                 [0,0,0,0,0]],

                [[0,0,0,0,0],
                 [0,0,1,0,0],
                 [0,1,1,1,0],
                 [0,0,1,0,0],
                 [0,0,0,0,0]],

                [[0,0,0,0,0],
                 [0,1,1,0,0],
                 [0,0,1,0,0],
                 [0,0,1,0,0],
                 [0,0,0,0,0]]]
```

We usually use `plt.subplots` to create a figure with several sub-figures.

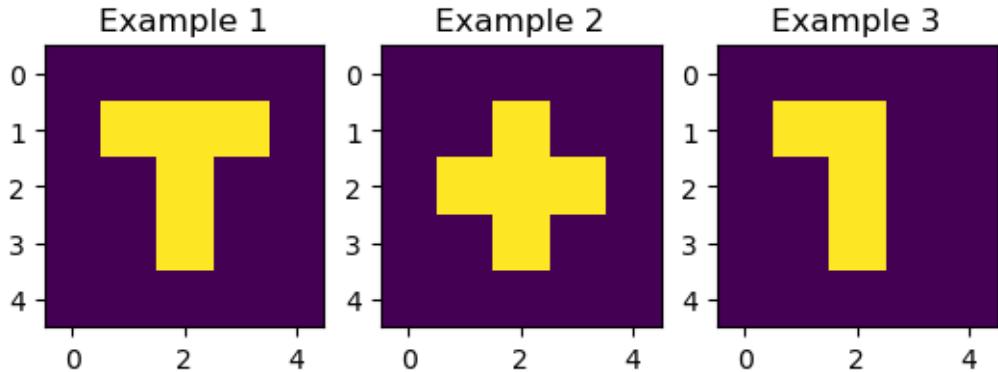
```
[11]: fig, axs = plt.subplots(1,3)

axs[0].imshow(dummy_images[0])
axs[0].set_title('Example 1')

axs[1].imshow(dummy_images[1])
axs[1].set_title('Example 2')

axs[2].imshow(dummy_images[2])
axs[2].set_title('Example 3')

plt.show()
```

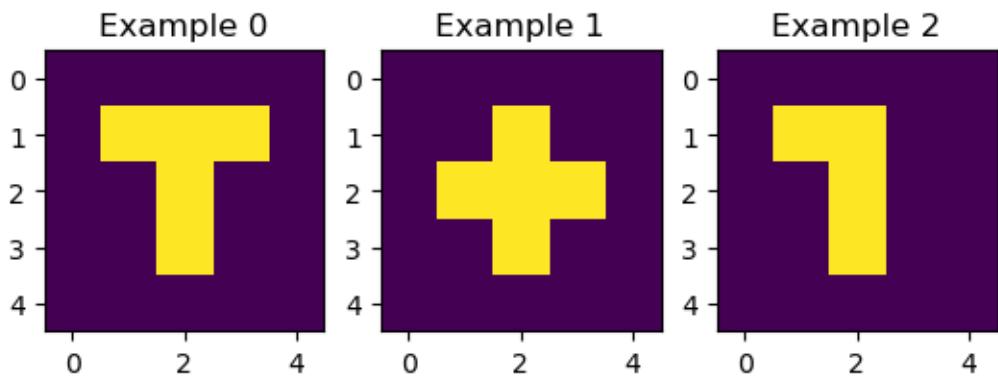


Another way to plot this is:

```
[15]: fig, axs = plt.subplots(1,3)

for i, ax in enumerate(axs.ravel()):
    ax.imshow(dummy_images[i])
    ax.set_title(f"Example {i}")

plt.show()
```



Note that you can iterate over subplots stored in object `axs`

```
In: print(axs)
Out: [<Axes: > <Axes: > <Axes: >]
```

with method `ravel` which creates a list of subplots.

There is a second way to create subplots which makes use of the `plt.figure.add_subplot` method. This is particularly useful when combining 3d and 2d sub-figures like in the following example.

```
[16]: data_3d = np.random.multivariate_normal(mean=[1,2,4], cov=np.eye(3), size=100)

fig = plt.figure(figsize=(10,2))

ax = fig.add_subplot(1,4,1, projection='3d')
ax.scatter(data_3d[:,0], data_3d[:,1], data_3d[:,2], marker='.')

ax.set_title('3D Example')

ax = fig.add_subplot(1,4,2)

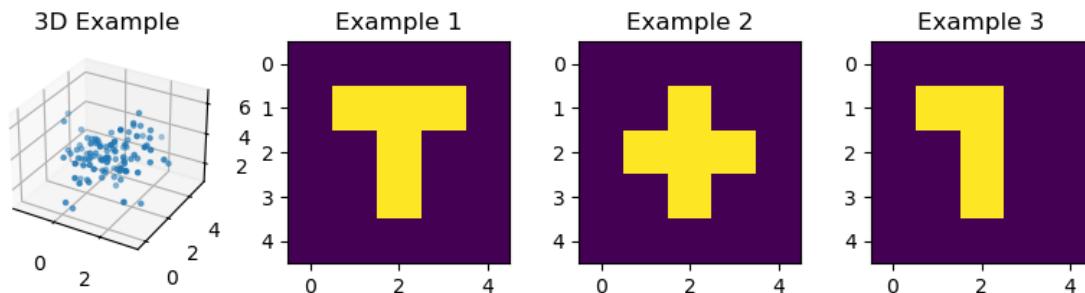
ax.imshow(dummy_images[0])
ax.set_title("Example 1")

ax = fig.add_subplot(1,4,3)

ax.imshow(dummy_images[1])
ax.set_title("Example 2")

ax = fig.add_subplot(1,4,4)
ax.imshow(dummy_images[2])
ax.set_title("Example 3")

plt.show()
```



1.4 Type Annotations and Type Hints

You might have appreciated Python for its *dynamic typing*, i.e. that Python allows variables to change their data type during runtime. However, this kind of flexibility can make your code more prone to errors such that proficient coding in Python often resorts back to a flavour of *static typing* by using **type annotations**.

Consider the following example:

```
[17]: counter = 1

text = 'This is a test'

country_codes = {'Switzerland': 41, 'France': 33, 'Italy': 39,
                 'UK': 44, 'Germany': 49}

text = 100
```

The following might remind you of variable declarations you've seen in C, C++ or other statically typed programming languages:

```
[18]: counter: int = 1

text: str = 'This is a test'

country_codes: dict[str, int] = {'Switzerland': 41, 'France': 33, 'Italy': 39,
                                 'UK': 44, 'Germany': 49}

text = 100
```

Advantages of type annotations include that typical IDEs support checking whether variables are assigned the expected type and automated unit tests can be used more efficiently. In PyCharm, the above example would provide the following

This can be particularly useful when your code becomes increasingly abstract, e.g. with functions

```
[19]: def divide_10_by(some_numbers: list[float]) -> None:

    for num in some_numbers:
        print(f'10/{num} = {10/num}')

my_numbers: list[float] = [0.5, 20.0, 100.0]

divide_10_by(some_numbers = my_numbers)
```

10/0.5 = 20.0
10/20.0 = 0.5
10/100.0 = 0.1

or classes

```
[ ]: class Person:
    ...

alice: Person = Person()
```

1.5 Eval Function

Did you ever wonder whether you can pass a string which gets evaluate as a Python expression within your code? Indeed, the `eval` provides this possibility:

```
[20]: my_data = np.array([10, 20, 30])

my_input = 'print(my_data**2)'

eval(my_input)

[100 400 900]
```

You require a valid Python expression in `my_input` for this to work!

```
[23]: my_input = input()

eval(my_input)
```

```
-----
KeyboardInterrupt                                     Traceback (most recent call last)
Cell In[23], line 1
----> 1 my_input = input()
      3 eval(my_input)

File ~/anaconda3/APML/lib/python3.11/site-packages/ipykernel/kernelbase.py:1262
  ↪in Kernel.raw_input(self, prompt)
  1260     msg = "raw_input was called, but this frontend does not support"
  ↪input requests."
  1261     raise StdinNotImplementedError(msg)
-> 1262 return self._input_request(
  1263     str(prompt),
  1264     self._parent_ident["shell"],
  1265     self.get_parent("shell"),
  1266     password=False,
  1267 )

File ~/anaconda3/APML/lib/python3.11/site-packages/ipykernel/kernelbase.py:1305
  ↪in Kernel._input_request(self, prompt, ident, parent, password)
  1302 except KeyboardInterrupt:
  1303     # re-raise KeyboardInterrupt, to truncate traceback
  1304     msg = "Interrupted by user"
-> 1305     raise KeyboardInterrupt(msg) from None
  1306 except Exception:
  1307     self.log.warning("Invalid Message:", exc_info=True)

KeyboardInterrupt: Interrupted by user
```

1.6 Lambda Functions

Typically, we declare a function like this in Python:

```
[25]: def fahrenheit_to_celsius(deg_F):
        return (deg_F - 32) * 5/9
```

```
[26]: fahrenheit_to_celsius(98.6)
```

```
[26]: 37.0
```

For such simple **single expression functions** a lambda function might be a convenient choice.

```
[27]: fahrenheit_to_celsius_lambda = lambda x: (x - 32) * 5/9
```

```
[28]: fahrenheit_to_celsius_lambda(98.6)
```

```
[28]: 37.0
```

They are sometimes referred to as **anonymous functions**, as they are not required to be bound to a name, e.g.

```
[29]: (lambda x: (x - 32) * 5/9)(98.6)
```

```
[29]: 37.0
```

We will see a more interesting example below!

1.7 Generators

Generator functions are special functions particularly suited for generating sequences of various kind.

```
[30]: def squared_sequence(limit):
        num = 0

        while num < limit:
            yield num**2
            num += 1
```

where `num += 1` is shorthand for `num = num + 1`.

```
[31]: squared_sequence(10)
```

```
[31]: <generator object squared_sequence at 0x175a305f0>
```

```
[32]: gen = squared_sequence(10)
```

```
[37]: next(gen)
```

[37]: 16

```
[38]: for i in squared_sequence(10):
    print(i)
```

```
0
1
4
9
16
25
36
49
64
81
```

Note that the above is equivalent to

```
for i in range(10):
    print(i**2)
```

Note the differences to standard functions:

- Generators *yield* values, suspend the function and maintain the local state
- The values are generated when they are required (called with `next()`), i.e. we do not store a whole list but generate one element at a time
- Once all elements are generated, the iteration stops

However, you can still have all elements in a list if need be:

```
[39]: list(squared_sequence(10))
```

```
[39]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

1.8 Map and Filter

The built-in functions `map` and `filter` are convenient ways to apply a function to all elements of an iterable and avoid writing a loop for that.

With `map` we apply the function to each element of the iterable(s) and return the results. We use it in the following way:

```
map( function, iterable(s) )
```

Let's see some examples.

```
[40]: from matplotlib.colors import to_rgb
colours = ['green', 'red', 'blue', 'yellow']
```

```
# for c in colours:
#     print(f"{c} in RGB is: {to_rgb(c)}")

rgb_colours = map( to_rgb, colours )

print(f"Try to print rgb_colours: {rgb_colours}\n")

print(list(rgb_colours))
```

Try to print rgb_colours: <map object at 0x1759bf760>

[(0.0, 0.5019607843137255, 0.0), (1.0, 0.0, 0.0), (0.0, 0.0, 1.0), (1.0, 1.0, 0.0)]

[41]: decimals = [3.14159, 2.71828, 1.61803]

```
rounded = list( map( round, decimals, range(1,4) ) )

print(rounded)
```

[3.1, 2.72, 1.618]

With `filter` we apply a boolean function to an iterable and provide only the elements which returned True. We use it like this:

`filter(function, iterable)`

where `function` returns boolean values `True` or `False`.

[43]: def passed(grade):
 return grade > 4.0

[44]: grades = [5.5, 6.0, 2.0, 3.5, 4.5]

list(filter(passed, grades))

[44]: [5.5, 6.0, 4.5]

A typical use case of lambda functions is actually in connection with `map` or `filter`.

[45]: list(filter(lambda grade: grade > 4.0, grades))

[45]: [5.5, 6.0, 4.5]

[46]: list(map(lambda grade: grade > 4.0, grades))

[46]: [True, True, False, False, True]

1.9 *args and **kwargs

Sometimes you see these `*args` and `**kwargs` arguments in functions and classes. We use them to write functions with variable number of arguments of positional and keyword arguments!

Here's an example:

```
[64]: def print_arguments(first, *args, **kwargs):
    print(first)

    if args:
        print(args)

    if kwargs:
        print(kwargs)
```

```
[65]: print_arguments()
```

```
-----  
TypeError                                         Traceback (most recent call last)  
Cell In[65], line 1  
----> 1 print_arguments()  
  
TypeError: print_arguments() missing 1 required positional argument: 'first'
```

```
[66]: print_arguments(1)
```

```
1
```

```
[67]: print_arguments(1, 2, 3)
```

```
1  
(2, 3)
```

```
[68]: print_arguments(1, 2, 3, 4, color='red', marker='x')
```

```
1  
(2, 3, 4)  
{'color': 'red', 'marker': 'x'}
```

Note that with `*args` we collect additional positional arguments in a tuple and with `**kwargs` the additional keyword arguments in a dictionary.

1.10 Decorators

Decorators are a great way to modify the behaviour of a function or class without changing the function or class itself.

```
[89]: def fahrenheit_to_celsius(deg_F):
    deg_C = (deg_F - 32) * 5/9
    return deg_C
```

```
[90]: fahrenheit_to_celsius(80)
```

[90]: 26.666666666666668

Let's define a decorator and "decorate" the previous function!

```
[91]: def just_int(func):

    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        print(int(res))
        return res

    return wrapper
```

```
[92]: @just_int
def fahrenheit_to_celsius(deg_F):
    deg_C = (deg_F - 32) * 5/9
    return deg_C
```

```
[93]: fahrenheit_to_celsius(deg_F=80)
```

26

[93]: 26.666666666666668

```
[94]: just_int(fahrenheit_to_celsius)(80)
```

26

26

[94]: 26.666666666666668

```
[74]: def just_int(print_out=None):

    def just_int_inner(func):

        def wrapper(*args, **kwargs):
            if print_out:
                print(print_out)
```

```

    res = func(*args, **kwargs)
    return int(res)

    return wrapper

    return just_int_inner

```

[77]: `@just_int(print_out='Get rid of decimals!')`

```

def fahrenheit_to_celsius(deg_F):
    deg_C = (deg_F - 32) * 5/9
    return deg_C

```

[78]: `fahrenheit_to_celsius(deg_F=80)`

[78]: 26

1.11 Numba

Numba is a just-in-time compiler for Python which can speed up your code substantially if it is based on loops, NumPy arrays and NumPy functions. Its main feature is the `jit` decorator. Find more on Numba in the [official documentation](#).

Let's see what it does on a small example.

[81]: `from numba import jit`
`import numpy as np`

[83]: `def calc_pi(total_num_points):`
 `np.random.seed(1234)`

 `num_circle_points = 0`

 `for i in range(total_num_points):`
 `x = np.random.random()`
 `y = np.random.random()`
 `radius_squared = np.power(x, 2) + np.power(y, 2)`

 `if radius_squared < 1:`
 `num_circle_points += 1`

 `pi_estimate = 4 * num_circle_points / total_num_points`
 `return pi_estimate`

[84]: `@jit`
`def calc_pi_faster(total_num_points):`
 `np.random.seed(1234)`

```

num_circle_points = 0

for i in range(total_num_points):
    x = np.random.random()
    y = np.random.random()
    radius_squared = np.power(x,2) + np.power(y,2)

    if radius_squared < 1:
        num_circle_points += 1

pi_estimate = 4 * num_circle_points / total_num_points
return pi_estimate

```

The function `calc_pi_faster` is compiled to machine code when called for the first time.

[85]: `total_num_points = 100000`

```

print(f"calc_pi: {calc_pi(total_num_points)}")
print(f"calc_pi_faster: {calc_pi_faster(total_num_points)}")

```

```

calc_pi: 3.14856
calc_pi_faster: 3.14856

```

[86]: `%timeit calc_pi(total_num_points)`

```
275 ms ± 2.89 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[87]: `%timeit calc_pi_faster(total_num_points)`

```
653 µs ± 4.36 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Note that Jupyter (IPython) provides some built-in Python functionality through *magic commands* `%command`.

1.12 Exercise Section

Generators are very useful when loading datasets, for instance. Suppose your dataset is too large to load into memory (RAM) (e.g. several tens of GB). Typically, a possible approach is to load parts of the dataset during training as needed.

In this exercise, we implement a generator to load *batches* of images from the **MNIST dataset**. MNIST is a standard Deep Learning benchmark dataset comprising 70 000 images (usually 28 x 28 pixels) of hand-written digits with the corresponding label as the classification target.

You can load the dataset (reduced to 5'000 samples) in the next cell.

[95]: `import numpy as np`

```
mnist_data = np.load('data/mnist_data_5k.npy', mmap_mode='r', allow_pickle=True)
```

```
mnist_targets = np.load('data/mnist_labels_5k.npy', mmap_mode='r',  
allow_pickle=True)
```

Note that `mmap_mode='r'` specifies that we do not load data into memory but instead a memory-mapped array is constructed. This means, the data is read from the disk whenever accessed. Note that

```
In[1]: mnist_data = np.load('data/mnist_data_5k.npy', mmap_mode='r')  
In[2]: type(mnist_data)  
Out[2]: numpy.memmap
```

provides a `memmap` object, while

```
In[1]: mnist_data = np.load('data/mnist_data_5k.npy')  
In[2]: type(mnist_data)  
Out[2]: numpy.ndarray
```

provides the standard NumPy array. These are two different types of objects. `memmap` lives still on the disk, while the NumPy array is loaded into RAM. Note that in a `memmap` many but not all NumPy operations will work.

(1.) Define a generator which allows passing the data, targets, and the size of the batch in the following cell. For e.g. `batch_size=10`, the generator shall slice both arrays into the first 10 samples and provide them with the `next` call. With every following call, the next 10 samples are provided. So you need to think about how to access a slice of data in NumPy arrays and how consecutively get the next slice with the next call by adjusting the indices.

[]:

If you were successful, you can create the generator with

[]: `mnist_gen = batch_data(data=mnist_data, targets=mnist_targets, batch_size=10)`

and retrieve a (new) batch of data with

[]: `new_data, new_targets = next(mnist_gen)`

```
print(f"Target labels in this batch: {new_targets}")
```

1.13 Proposed Solutions

Generators are very useful when loading datasets, for instance. Suppose your dataset is too large to load into memory (RAM) (e.g. several tens of GB). Typically, a possible approach is to load parts of the dataset during training as needed.

In this exercise, we implement a generator to load *batches* of images from the **MNIST dataset**. MNIST is a standard Deep Learning benchmark dataset comprising 70 000 images (usually 28 x 28 pixels) of hand-written digits with the corresponding label as the classification target.

You can load the dataset (reduced to 5 000 samples) in the next cell.

```
[96]: import numpy as np

mnist_data = np.load('data/mnist_data_5k.npy', mmap_mode='r', allow_pickle=True)
mnist_targets = np.load('data/mnist_labels_5k.npy', mmap_mode='r', allow_pickle=True)
```

Note that `mmap_mode='r'` specifies that we do not load data into memory but instead a memory-mapped array is constructed. This means, the data is read from the disk whenever accessed. Note that

```
In[1]: mnist_data = np.load('data/mnist_data_5k.npy', mmap_mode='r')
In[2]: type(mnist_data)
Out[2]: numpy.memmap
```

provides a `memmap` object, while

```
In[1]: mnist_data = np.load('data/mnist_data_5k.npy')
In[2]: type(mnist_data)
Out[2]: numpy.ndarray
```

provides the standard NumPy array. These are two different types of objects. `memmap` lives still on the disk, while the NumPy array is loaded into RAM. Note that in a `memmap` many but not all NumPy operations will work.

(1.) Define a generator which allows passing the data, targets, and the size of the batch in the following cell. For e.g. `batch_size=10`, the generator shall slice both arrays into the first 10 samples and provide them with the `next` call. With every following call, the next 10 samples are provided. So you need to think about how to access a slice of data in NumPy arrays and how consecutively get the next slice with the next call by adjusting the indices.

```
[100]: def batch_data(data, targets, batch_size):
    batch_num = 0
    tot_num_batches = int(data.shape[0] / batch_size)

    while batch_num < tot_num_batches:

        data_batch = data[batch_num*batch_size : (batch_num+1)*batch_size]
        targets_batch = targets[batch_num*batch_size:(batch_num+1)*batch_size]

        yield np.array(data_batch), np.array(targets_batch)

        batch_num = batch_num + 1
```

If you were successful, you can create the generator with

```
[101]: mnist_gen = batch_data(data=mnist_data, targets=mnist_targets, batch_size=10)
```

and retrieve a (new) batch of data with

```
[105]: new_data, new_targets = next(mnist_gen)

print(f"Target labels in this batch: {new_targets}")
```

Target labels in this batch: [3 8 6 9 0 5 6 0 7 6]

2-Regression_and_Classification

May 7, 2024

1 2. Regression and Classification

Regression and classification are two fundamental tasks of supervised Machine Learning where labels allows us to guide the learning. This notebook reviews more standard approaches to regression and classification.

We provide the fundamental ideas behind

- linear regression and ridge regression
- logistic regression for classification
- decision trees and Random Forests

and apply these techniques to simulated data and a dataset example (wine quality assessment).

Keywords: OLS, MSE, Overfitting, Regularisation, np.linalg.inv, sklearn.linear_model.LinearRegression, sklearn.linear_model.Ridge, sklearn.tree.DecisionTreeClassifier, sklearn.linear_model.LogisticRegression, sklearn.ensemble.RandomForestClassifier, sklearn.model_selection.train_test_split

We start by fixing a random seed which controls the generation of (pseudo) random number sequences.

```
[2]: import numpy as np
import matplotlib.pyplot as plt

np.random.seed(123)
```

Note that this enables reproducibility of our results even in the presence of “randomness”. For example, the first three runs of the following cell

```
[3]: np.random.random(size=4)
```

```
[3]: array([0.69646919, 0.28613933, 0.22685145, 0.55131477])
```

will always produce

1. array([0.69646919, 0.28613933, 0.22685145, 0.55131477])
2. array([0.71946897, 0.42310646, 0.9807642 , 0.68482974])

```
3. array([0.4809319 , 0.39211752, 0.34317802, 0.72904971])
```

1.1 Standard Linear Regression and Ordinary Least Squares

We assume a linear relationship between true **response** $Y_{\text{groundtruth}} = Xw$ and **predictors / covariates** X . However, usually **our observation is not perfect**, i.e. there is some noise additional ϵ .

A simple linear regression model is then

$$Y = Xw + \epsilon. \quad (1)$$

Let us simulate $N = 10$ datapoints with Gaussian noise $\epsilon \sim \mathcal{N}(\mu = 0, \sigma = 20)$.

```
[10]: np.random.seed(123)

num_datapoints = 10

X = np.random.random(size=num_datapoints)*50
X = np.sort(X).reshape(-1,1)

print(X.shape)
print(X)

w_groundtruth = 10
Y_groundtruth = w_groundtruth*X

epsilon_noise = np.random.normal(loc=0, scale=20.0, size=(num_datapoints,1))

Y = Y_groundtruth + epsilon_noise
```

```
(10, 1)
[[11.34257268]
 [14.30696675]
 [19.60587591]
 [21.15532301]
 [24.04659507]
 [27.56573845]
 [34.24148693]
 [34.82345928]
 [35.97344849]
 [49.03820992]]
```

A typical objective function is given by the **mean squared error** (MSE) loss

$$\text{Loss}(w) = \frac{1}{2N} \|\epsilon\|_2^2 = \frac{1}{2N} \|Y - Xw\|_2^2 = \frac{1}{2N} \sum_{n=1}^N (y_n - x_n \cdot w)^2. \quad (2)$$

Under certain conditions, the solution to this equation will provide the **best linear (unbiased) estimator** for w ! We identify the **minimum** through

$$\nabla \text{Loss}(w) = X^\top Y - X^\top X w \equiv 0, \quad (3)$$

which provides the **ordinary least squares (OLS)** solution

$$w_{\text{OLS}} = (X^\top X)^{-1} X^\top Y. \quad (4)$$

```
[5]: XT = np.transpose(X)
XTX_inverse = np.linalg.inv(XT @ X)

w_OLS = XTX_inverse @ XT @ Y

print(f"The solution is given by w_OLS = {w_OLS.squeeze()}.")
```

The solution is given by $w_{\text{OLS}} = 10.375338325958648$.

Note

- Operator `@` provides the matrix multiplication of two numpy array (similar to `np.matmul`)
- Function `numpy.array.squeeze()` removes all dimensions of size 1 from a NumPy array, i.e.

In: `print(w_OLS.shape)`
Out: `(1, 1)`

In: `print(w_OLS)`
Out: `[[10.04673211]]`

In: `print(w_OLS.squeeze())`
Out: `10.046732109803889`

Let's provide some predictions Y_{OLS} based on this solution.

```
[6]: Y_OLS = w_OLS * X
```

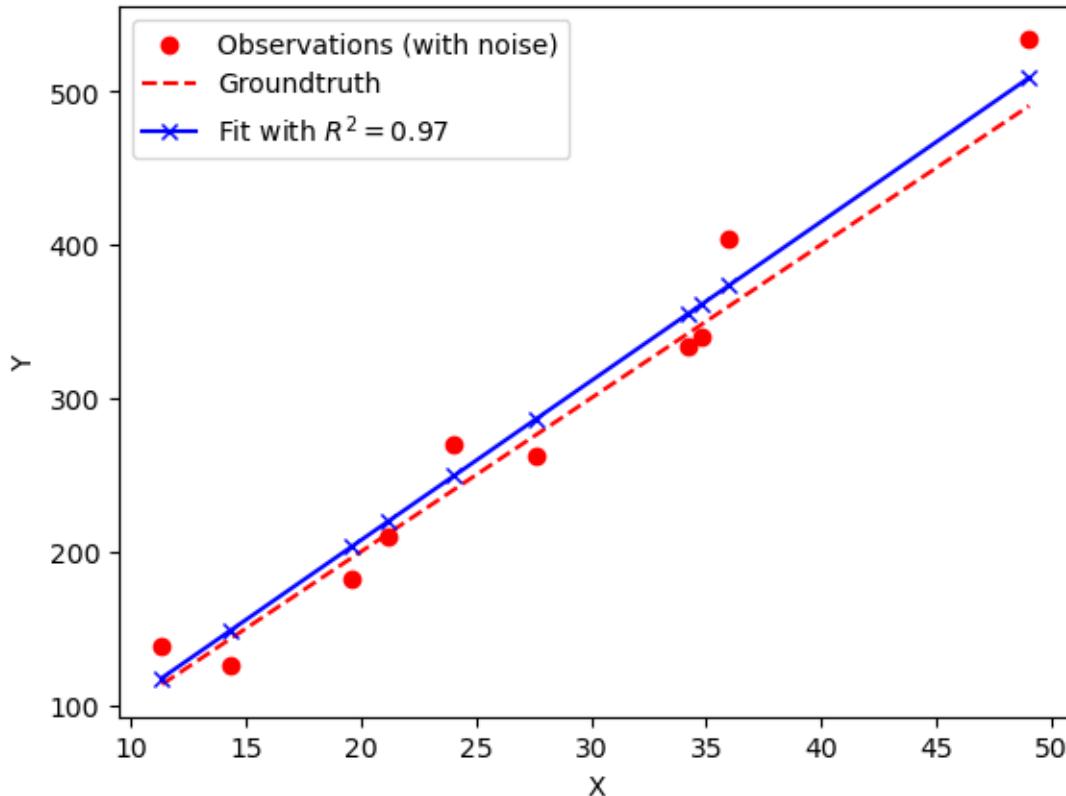
With this, we can calculate the **coefficient of determination R^2** ([Wikipedia](#)) through

```
[7]: Rsquared = 1 - np.sum((Y - Y_OLS)**2) / np.sum((Y - np.mean(Y))**2)
print(Rsquared)
```

0.965593882960831

```
[8]: plt.scatter(X, Y, color='red', label='Observations (with noise)')
plt.plot(X, Y_groundtruth, color='red', linestyle='dashed', label='Groundtruth')
plt.plot(X, Y_OLS, marker='x', color='blue', label=f'Fit with $R^2={Rsquared:.2f}$')
```

```
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```



What happens if you have more “flexible models” at your disposal? In the following, we try to fit our observed responses Y with a polynomial model of the form

$$Y = X_{\text{poly}}w + \epsilon \quad (5)$$

$$= x \cdot w_1 + x^2 \cdot w_2 + x^3 \cdot w_3 + \dots + x^7 \cdot w_7 + x^8 \cdot w_8 + \epsilon \quad (6)$$

and create a hundred test datapoints.

```
[9]: X_poly = np.hstack((X, X ** 2, X ** 3, X ** 4, X ** 5, X ** 6, X ** 7, X ** 8))
print(X_poly.shape)

X_test = np.linspace(10,50,100).reshape(-1, 1)
X_test = np.hstack((X_test, X_test ** 2, X_test ** 3, X_test ** 4, X_test ** 5, X_test ** 6, X_test ** 7, X_test ** 8))
```

(10, 8)

Let's see the OLS solution for that case:

```
[11]: XT = np.transpose(X_poly)
XTX_inverse = np.linalg.inv(XT @ X_poly)

w_OLS = XTX_inverse @ XT @ Y

print(f"Solution w_OLS = {w_OLS.squeeze()}")

Y_OLS = X_poly @ w_OLS
Y_OLS_test = X_test @ w_OLS

Rsquared = 1 - np.sum((Y - Y_OLS)**2) / np.sum((Y - np.mean(Y))**2)

fig, axs = plt.subplots(1,2, figsize=(12,4))

axs[0].scatter(X, Y, marker='o', color='red', label='Observations (with noise)')
axs[0].plot(X, Y_groundtruth, color='red', linestyle='dashed', label=f'Groundtruth')
axs[0].scatter(X, Y_OLS, color='blue', marker='x', label=f'Train; fit with R^2={Rsquared:.2f}')

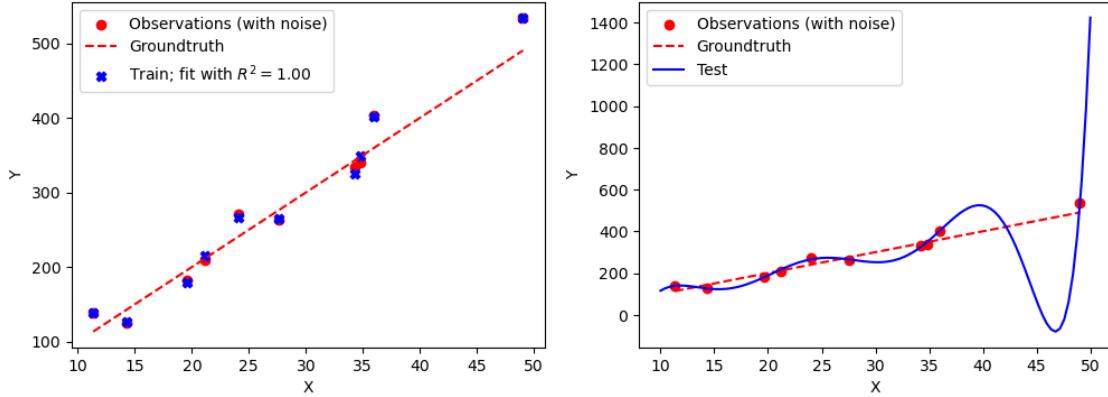
axs[1].scatter(X, Y, marker='o', color='red', label='Observations (with noise)')
axs[1].plot(X, Y_groundtruth, color='red', linestyle='dashed', label=f'Groundtruth')
axs[1].plot(X_test[:,0], Y_OLS_test, color='blue', label='Test')

axs[0].set_xlabel('X')
axs[0].set_ylabel('Y')
axs[0].legend()

axs[1].set_xlabel('X')
axs[1].set_ylabel('Y')
axs[1].legend()

plt.show()

Solution w_OLS = [-7.51130753e+02  2.57966644e+02 -3.52218133e+01
2.52026936e+00
-1.02565199e-01  2.38549629e-03 -2.94929359e-05  1.50136544e-07]
```



1.2 Ridge Regression

Ridge regression adds an additional assumption on (the distribution of) weights w . We rewrite the loss as

$$\text{Loss}(w) = \frac{1}{2N} \|Y - Xw\|_2^2 + \frac{1}{2} \lambda' \|w\|_2^2 = \frac{1}{2N} \sum_{n=1}^N (y_n - x_n \cdot w)^2 + \frac{1}{2} \lambda' \sum_{i=1}^{d_x} w_i^2 \quad (7)$$

where the minimisation also takes into account the value (norm) of the weights w_i . Here, d_x indicates the dimensions of our input X , e.g. $d_x = 8$ in the case of the polynomial model above. The solution to this minimisation is given by

$$w_{\text{ridge}} = (\lambda \mathbb{1}_{d_x} + X^\top X)^{-1} X^\top Y. \quad (8)$$

Let's see what happens with this additional term:

```
[12]: lambda_ridge = 100000

XT = np.transpose(X_poly)
XTX_ridge_inverse = np.linalg.inv(XT @ X_poly + lambda_ridge * np.
    ↪identity(X_poly.shape[1]))

w_ridge = XTX_ridge_inverse @ XT @ Y

print(f"Solutions: \n w_OLS = {w_OLS.squeeze()}, \n w_ridge = {w_ridge.
    ↪squeeze()}")
```

Solutions:

```
w_OLS =
[-7.51130753e+02  2.57966644e+02 -3.52218133e+01
 2.52026936e+00]
```

```
-1.02565199e-01 2.38549629e-03 -2.94929359e-05 1.50136544e-07] ,
w_ridge = [ 2.47873294e-03 1.58957362e-02 6.22240854e-02
4.74477828e-03
-7.57974781e-04 3.01976405e-05 -4.90619858e-07 2.84986667e-09].
```

```
[13]: Y_ridge = X_poly @ w_ridge
Y_ridge_test = X_test @ w_ridge

Rsquared_ridge = 1 - np.sum((Y - Y_ridge)**2) / np.sum((Y - np.mean(Y))**2)

print(Rsquared_ridge)

fig, axs = plt.subplots(1,2, figsize=(12,4))

axs[0].scatter(X, Y, marker='o', color='red', label='Observations (with noise)')
axs[0].plot(X, Y_groundtruth, color='red', linestyle='dashed', label='Groundtruth')
axs[0].scatter(X, Y_OLS, color='blue', marker='x', label=f'Train; OLS with R^2={Rsquared:.2f}')
axs[0].scatter(X, Y_ridge, color='green', marker='+', label=f'Train; ridge with R^2={Rsquared_ridge:.2f}')

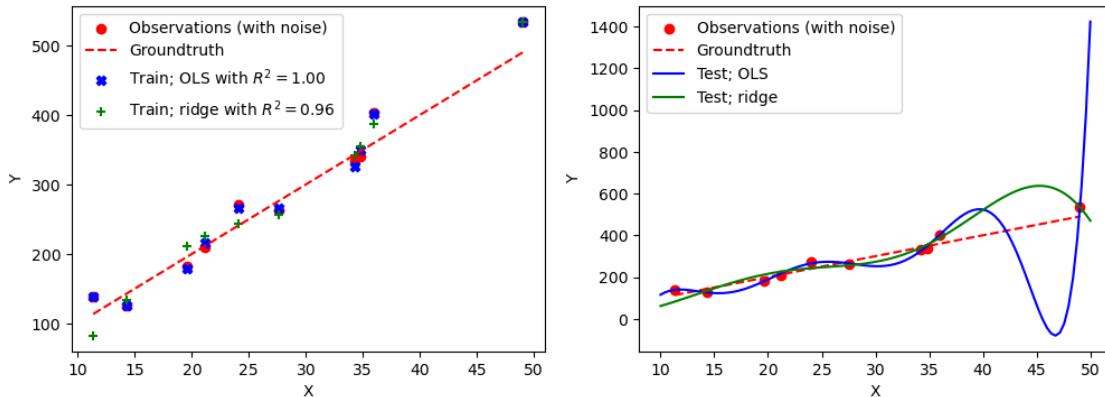
axs[1].scatter(X, Y, marker='o', color='red', label='Observations (with noise)')
axs[1].plot(X, Y_groundtruth, color='red', linestyle='dashed', label='Groundtruth')
axs[1].plot(X_test[:,0], Y_OLS_test, color='blue', label='Test; OLS')
axs[1].plot(X_test[:,0], Y_ridge_test, color='green', label='Test; ridge')

axs[0].set_xlabel('X')
axs[0].set_ylabel('Y')
axs[0].legend()

axs[1].set_xlabel('X')
axs[1].set_ylabel('Y')
axs[1].legend()

plt.show()
```

0.9604147033799222



Note that the additional term $\lambda \|w\|_2^2 = \lambda \sum_{i=1}^{d_x=8} w_i^2$ in the loss function enables deviation from the observed noisy datapoints. That is models (i.e. weights w_i) which too closely *overfit* to the noisy observations are penalised. Parameter λ serves as a **penalty factor** biasing towards *simpler* models which **reduce overfitting**.

Generally, having terms in your loss function favouring simpler models is referred to as **regularisation** ([Wikipedia](#)). Highly-flexible models (like the blue curve in the next figure) allow fitting observations (red dots) arbitrary well. Typically, we prefer to choose simpler models (like the green curve) which usually provide better predictions to new, unseen data (**generalisation**).

Image source: Wikipedia

1.3 Using scikit-learn for Regression

You don't need to do all the work yourself! `scikit-learn` offers you the required functionality with the functions `LinearRegression` and `Ridge`!

```
[14]: from sklearn.linear_model import LinearRegression, Ridge
```

```
linreg = LinearRegression(fit_intercept=False)
linreg.fit(X, Y)
```

```
[14]: LinearRegression(fit_intercept=False)
```

Note that we used `fit_intercept=False` because we chose to neglect on off-set / intercept b on the y-axis. That is, we consider $Y = Xw + b + \epsilon$ with $b = 0$, which simplifies writing the equations out.

`linreg` is the linear fit which provides you the same OLS solution as seen above.

```
[15]: w_OLS_sk = linreg.coef_
Y_OLS_sk = linreg.predict(X)
Rsquared_sk = linreg.score(X, Y)
```

```
print(f"w_OLS_sk = {w_OLS_sk.squeeze()}, Rsquared_sk = {Rsquared_sk}")
```

w_OLS_sk = 10.375338325958642, Rsquared_sk = 0.965593882960831

We can do the same for the ridge regression case.

```
[17]: ridgereg = Ridge(alpha=10, fit_intercept=False)
ridgereg.fit(X, Y)
```

```
w_ridge_sk = ridgereg.coef_
Y_ridge_sk = ridgereg.predict(X)
Rsquared_ridge_sk = ridgereg.score(X, Y)
```

```
print(f"w_ridge_sk = {w_ridge_sk.squeeze()}, Rsquared_ridge_sk = {Rsquared_ridge_sk}")
```

w_ridge_sk = 9.293151038782826, Rsquared_ridge_sk = 0.965593882960831

1.4 Logistic Regression for Classification

Next, we consider binary classification where our data falls into one of two classes which we will denote with class labels $Y = 0$ (class 1) and $Y = 1$ (class 2). Let's assume that observations of class 2 occur with an unknown probability of p_2 . As we only have two classes, the probability for class 1 is $p_1 = 1 - p_2$. A natural choice in such a setting is the **Bernoulli distribution** ([Wikipedia](#)) with probability (mass) function

$$p(Y) = p_2^Y p_1^{1-Y} = p_2^Y (1 - p_2)^{1-Y}. \quad (9)$$

The idea of logistic regression is to approximate the ratio of the two class probabilities with an exponential function of the form

$$\frac{p_2}{p_1} = \exp(x_1 \cdot w_1 + x_2 \cdot w_1 \cdot w_2 + \dots + x_{d_x} \cdot w_{d_x}) = \exp(Xw), \quad (10)$$

so again a linear model! Note that we can write this as

$$\exp(Xw) = \frac{p_2}{1 - p_2} \Rightarrow p_2 = \frac{1}{1 + \exp(-Xw)}. \quad (11)$$

This is the **logistic function**. In order to identify the weights w , i.e. a matching model, we use the **logistic loss** (also known as **negative log-likelihood** or **cross entropy loss**, depending on the context)

$$\text{Loss}(w) = -Y \log p_2 - (1 - Y) \log(1 - p_2) = -\sum_{n=1}^N y_n \log \frac{1}{1 + \exp(-x_n \cdot w)} + (1 - y_n) \log \left(1 - \frac{1}{1 + \exp(-x_n \cdot w)}\right) \quad (12)$$

for N datapoints.

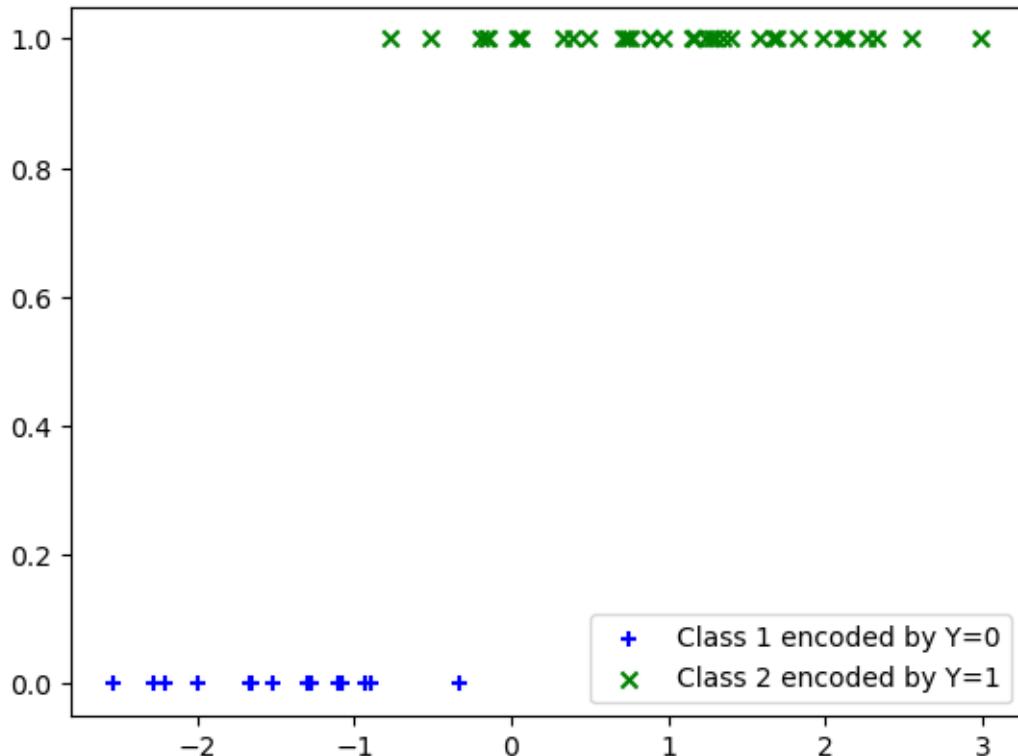
Let's create a simple binary classification problem with 50 datapoints.

```
[18]: from sklearn.datasets import make_classification

num_datapoints = 50

X, Y = make_classification(n_samples=num_datapoints,
                           n_features=1,
                           n_informative=1,
                           n_redundant=0,
                           n_classes=2,
                           n_clusters_per_class=1,
                           weights=[0.3, 0.7],
                           random_state=0)

plt.scatter(X[Y == 0], Y[Y == 0], color='blue', marker='+', label='Class 1\u2014  
encoded by Y=0')
plt.scatter(X[Y == 1], Y[Y == 1], color='green', marker='x', label='Class 2\u2014  
encoded by Y=1')
plt.legend()
plt.show()
```



Here we directly use `scikit-learn` function `LogisticRegression`.

```
[19]: from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression()
logreg.fit(X, Y)
```

```
[19]: LogisticRegression()
```

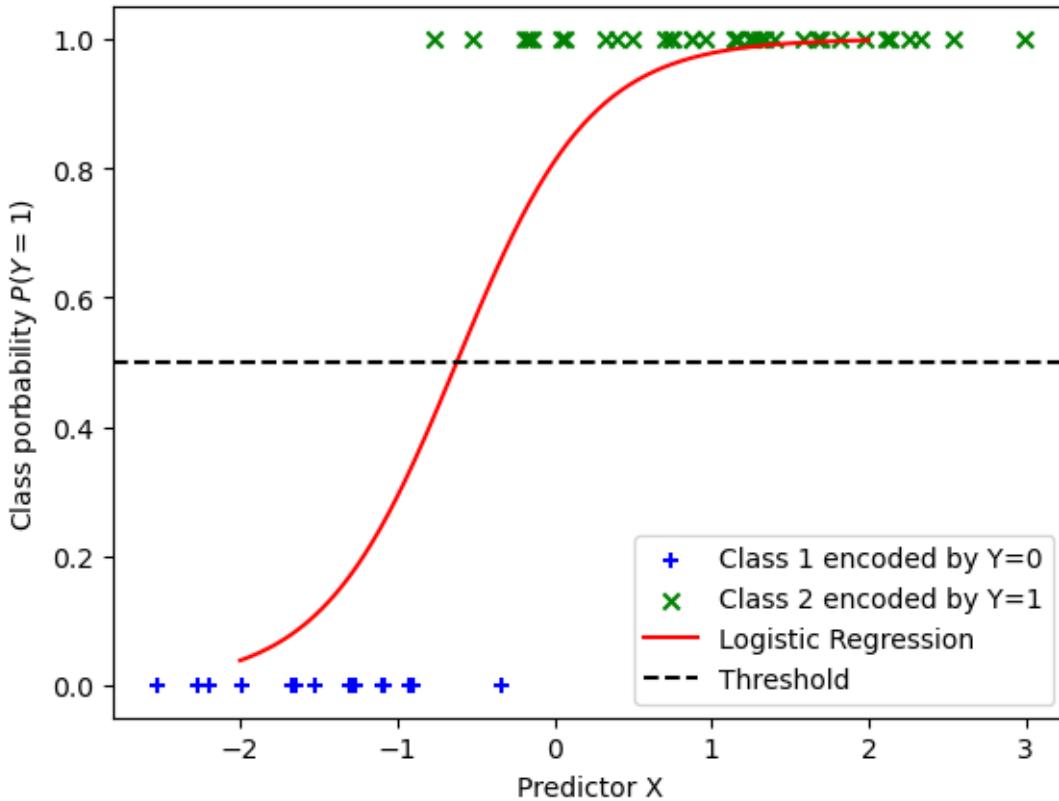
Using the method `predict_proba` allows predicting the probability $p(Y = 1)$ for class 2 of our logistic regression model `logreg`.

```
[20]: X_logreg = np.linspace(-2, 2, 1000).reshape(-1, 1)
Y_logreg = logreg.predict_proba(X_logreg)
```

Let's plot the result:

```
[21]: plt.scatter(X[Y == 0], Y[Y == 0], color='blue', marker='+', label='Class 1\u2020
    \u2020encoded by Y=0')
plt.scatter(X[Y == 1], Y[Y == 1], color='green', marker='x', label='Class 2\u2020
    \u2020encoded by Y=1')
plt.plot(X_logreg[:, 0], Y_logreg[:, 1], color='red', label='Logistic\u2020
    \u2020Regression')
plt.axhline(0.5, color='black', linestyle='--', label='Threshold')

plt.xlabel('Predictor X')
plt.ylabel(r'Class porbability $P(Y=1)$')
plt.legend()
plt.show()
```



1.5 Decision Trees and Random Forests

An more classical approach to classification problems are decision trees. In the next example, we will try to classify premium red wine from their measured properties / features which serve as the predictors / covariates.

```
[43]: import pandas as pd
import matplotlib.pyplot as plt

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')

# quality_threshold = 6
quality_threshold = 5

wine_data['premium'] = wine_data['quality'] > quality_threshold

wine_data
```

```
[43]:    fixed acidity  volatile acidity  citric acid  residual sugar  chlorides \
0            7.4          0.700        0.00           1.9       0.076
1            7.8          0.880        0.00           2.6       0.098
2            7.8          0.760        0.04           2.3       0.092
3           11.2          0.280        0.56           1.9       0.075
4            7.4          0.700        0.00           1.9       0.076
...
1594         ...          ...          ...          ...
1595         6.2          0.600        0.08           2.0       0.090
1595         5.9          0.550        0.10           2.2       0.062
1596         6.3          0.510        0.13           2.3       0.076
1597         5.9          0.645        0.12           2.0       0.075
1598         6.0          0.310        0.47           3.6       0.067

    free sulfur dioxide  total sulfur dioxide  density  pH  sulphates \
0                  11.0          34.0  0.99780  3.51      0.56
1                  25.0          67.0  0.99680  3.20      0.68
2                  15.0          54.0  0.99700  3.26      0.65
3                  17.0          60.0  0.99800  3.16      0.58
4                  11.0          34.0  0.99780  3.51      0.56
...
1594         ...          ...          ...          ...
1595         32.0          44.0  0.99490  3.45      0.58
1595         39.0          51.0  0.99512  3.52      0.76
1596         29.0          40.0  0.99574  3.42      0.75
1597         32.0          44.0  0.99547  3.57      0.71
1598         18.0          42.0  0.99549  3.39      0.66

    alcohol  quality  premium
0        9.4      5   False
1        9.8      5   False
2        9.8      5   False
3        9.8      6   True
4        9.4      5   False
...
1594     10.5      5   False
1595     11.2      6   True
1596     11.0      6   True
1597     10.2      5   False
1598     11.0      6   True

[1599 rows x 13 columns]
```

We focus on the quality assessment.

```
[44]: quality_counts = wine_data['quality'].value_counts()

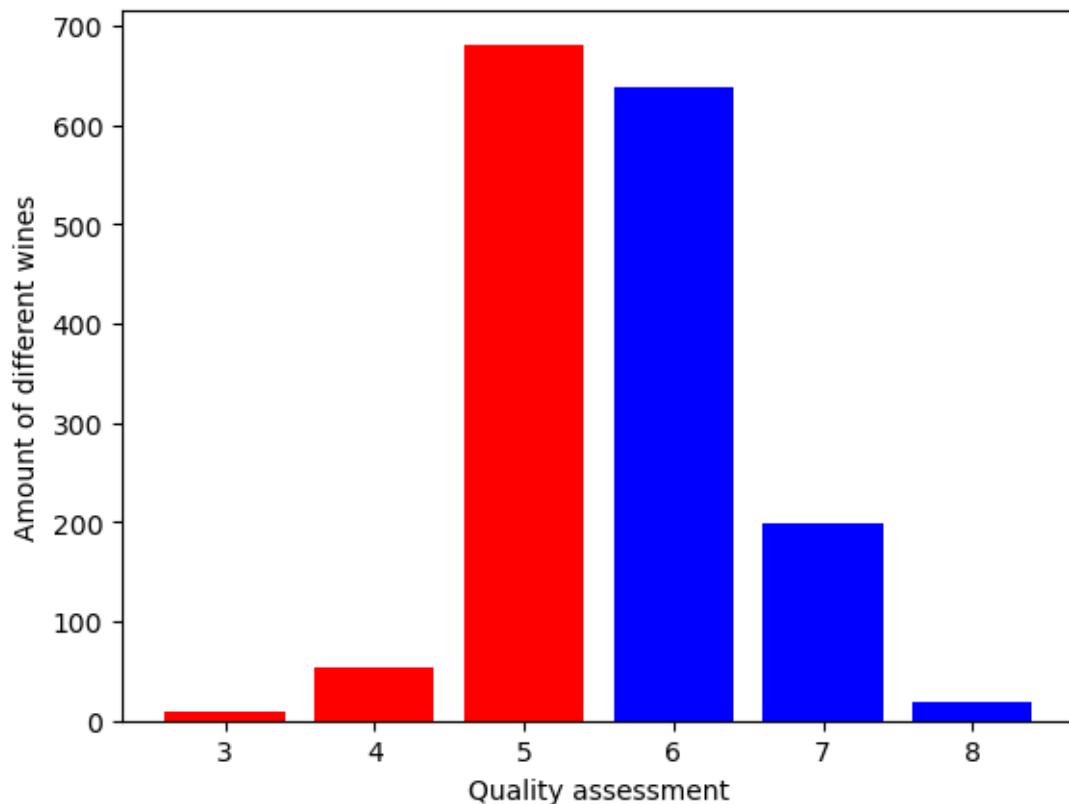
if quality_threshold == 5:
    colours = ['red', 'blue', 'blue', 'red', 'blue', 'red']
elif quality_threshold == 6:
```

```
colours = ['red', 'red', 'blue', 'red', 'blue', 'red']

print(quality_counts)

plt.bar(quality_counts.index, quality_counts, color=colours)
plt.xlabel('Quality assessment')
plt.ylabel('Amount of different wines')
plt.show()
```

```
quality
5      681
6      638
7      199
4      53
8      18
3      10
Name: count, dtype: int64
```



Perpare the features and target:

```
[45]: target = wine_data['premium'].astype(int)
wine_features = wine_data.drop( ['quality','premium'], axis=1)

print("Shape of wine_features:\t{}\nShape of target:\t{}\n"
      .format(wine_features.shape, target.shape)
      )

target.head(5)
```

Shape of wine_features: (1599, 11)
 Shape of target: (1599,)

```
[45]: 0    0
1    0
2    0
3    1
4    0
Name: premium, dtype: int64
```

The Goal is to learn a classifier which predicts whether a wine is a premium / non-premium wine on the basis of the measured wine features.

Select (randomly) a set on which the classifier is calibrated / trained on and a test set on which the performance is assessed. We consider a test set size of 30% of the original data set.

```
[46]: from sklearn.model_selection import train_test_split

feat_train, feat_test, target_train, target_test = train_test_split(
    wine_features, target, test_size = 0.3, random_state=123)

print("After splitting into train and test sets:\n\n"
      f"Shape of feat_train:{feat_train.shape}\nShape of target_train:
      \t{target_train.shape}\n"
      f"Shape of feat_test:{feat_test.shape}\nShape of target_test:
      \t{target_test.shape}"
      )
```

After splitting into train and test sets:

```
Shape of feat_train: (1119, 11)
Shape of target_train: (1119,)
Shape of feat_test: (480, 11)
Shape of target_test: (480,)
```

We have a first look on a decision tree and just consider the features providing the alcohol and sulphates content. We use the [DecisionTreeClassifier](#).

```
[47]: from sklearn.tree import DecisionTreeClassifier, plot_tree

feat_train_subset = feat_train[ ['alcohol','sulphates'] ]

wine_tree = DecisionTreeClassifier(max_depth=2)
wine_tree.fit(feat_train_subset, target_train)
```

[47]: DecisionTreeClassifier(max_depth=2)

```
[48]: fig, axs = plt.subplots(1,2, figsize=(10,5))

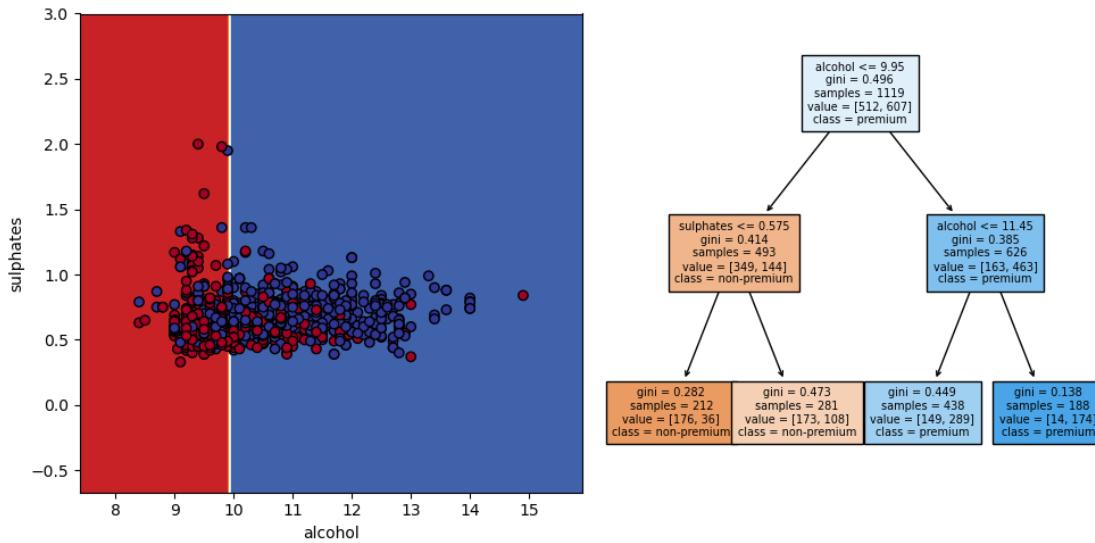
try:
    from sklearn.inspection import DecisionBoundaryDisplay

    display = DecisionBoundaryDisplay.from_estimator(
        wine_tree,
        feat_train_subset,
        cmap='RdYlBu',
        response_method="predict",
        ax=axs[0],
        xlabel='alcohol',
        ylabel='sulphates',
    )
except:
    print("Sorry, the library for plotting the decision "
          "boundary is currently missing")

plot_tree(wine_tree, max_depth=2,
          feature_names=['alcohol','sulphates'],
          class_names=['non-premium','premium'],
          ax=axs[1], fontsize=7, filled = True,
          )

axs[0].scatter(feat_train['alcohol'], feat_train['sulphates'],
               c=target_train, cmap='RdYlBu', edgecolor='black'
               )

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



Note that decision trees partition the input space into different regions with **decision boundaries** separating different classes, like premium wines (blue data / region) and non-premium wines (red data / region)!

Now let's consider the full dataset!

```
[49]: tree_height = 8

wine_tree = DecisionTreeClassifier(max_depth=tree_height)
wine_tree.fit(feat_train, target_train)

correct_pred = wine_tree.predict(feat_test) == target_test

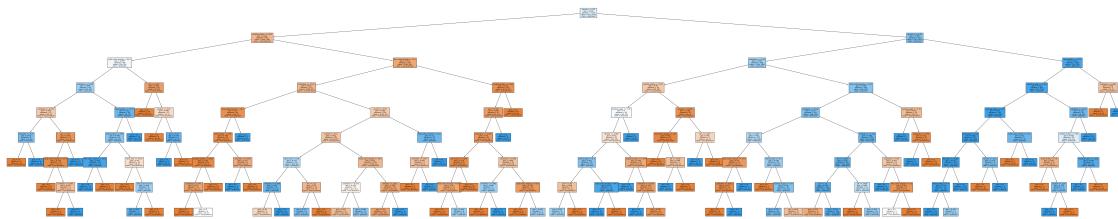
correct = correct_pred.value_counts()

accuracy = (correct[True] / (correct[True] + correct[False]))*100
print(f"The random forest identified premium / non-premium wines with "
      f"{accuracy}% accuracy!")

fig, ax = plt.subplots(1,1, figsize=(100,20))

plot_tree(wine_tree,
           feature_names=wine_features.columns.to_list(),
           class_names=['non-premium','premium'],
           filled = True,
           ax=ax,
           fontsize=8
          )
plt.show()
```

The random forest identified premium / non-premium wines with 70.83333333333334% accuracy!



You can learn more on decision trees from [scikit-learn](#) and the mathematical criteria for the decision splits.

Random Forests are now very powerful machine learning methods which build on decision trees. Random Forests construct a multitude of decision trees like the one above. The individual decision trees provide a classification result and by majority voting the final classification is obtained. We use the [RandomForestClassifier](#).

```
[50]: from sklearn.ensemble import RandomForestClassifier

num_trees = 100
tree_height = 8

wine_forest = RandomForestClassifier(n_estimators=num_trees,
                                      max_depth=tree_height,
                                      random_state=123)
wine_forest.fit(feat_train, target_train)
```

```
[50]: RandomForestClassifier(max_depth=8, random_state=123)
```

```
[51]: correct_pred = wine_forest.predict(feat_test) == target_test

correct = correct_pred.value_counts()

accuracy = (correct[True] / (correct[True] + correct[False]))*100
print(f"The random forest identified premium / non-premium wines with {accuracy}% accuracy!")
```

The random forest identified premium / non-premium wines with 78.125% accuracy!

That's a great accuracy! But let's check what is actually predicted wrongly:

```
[52]: rel_incorrect_pred = target_test[correct_pred == False].value_counts() / target_test.value_counts()
```

```

print(f"Incorrect predictions by premium quality:\n{rel_incorrect_pred}")
print(f"\nRatio of premium / non-premium wines in test set:\n{target_test.
    value_counts(normalize=True)}")
print(f"\nRatio of premium / non-premium wines in train set:\n{target_train.
    value_counts(normalize=True)}")

```

Incorrect predictions by premium quality:

```

premium
0      0.275862
1      0.165323
Name: count, dtype: float64

```

Ratio of premium / non-premium wines in test set:

```

premium
1      0.516667
0      0.483333
Name: proportion, dtype: float64

```

Ratio of premium / non-premium wines in train set:

```

premium
1      0.542449
0      0.457551
Name: proportion, dtype: float64

```

You can still plot one of the trees!

```

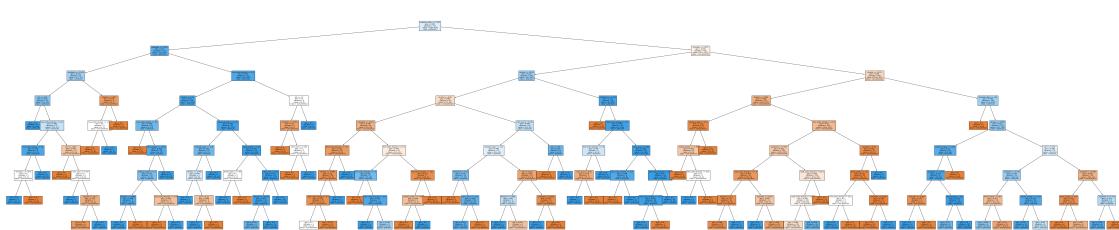
[53]: single_tree = wine_forest.estimators_[5]

fig, ax = plt.subplots(1,1, figsize=(100,20))

plot_tree(single_tree,
           feature_names=wine_features.columns.to_list(),
           class_names=['non-premium','premium'],
           filled = True,
           ax=ax,
           fontsize=8
          )

plt.show()

```

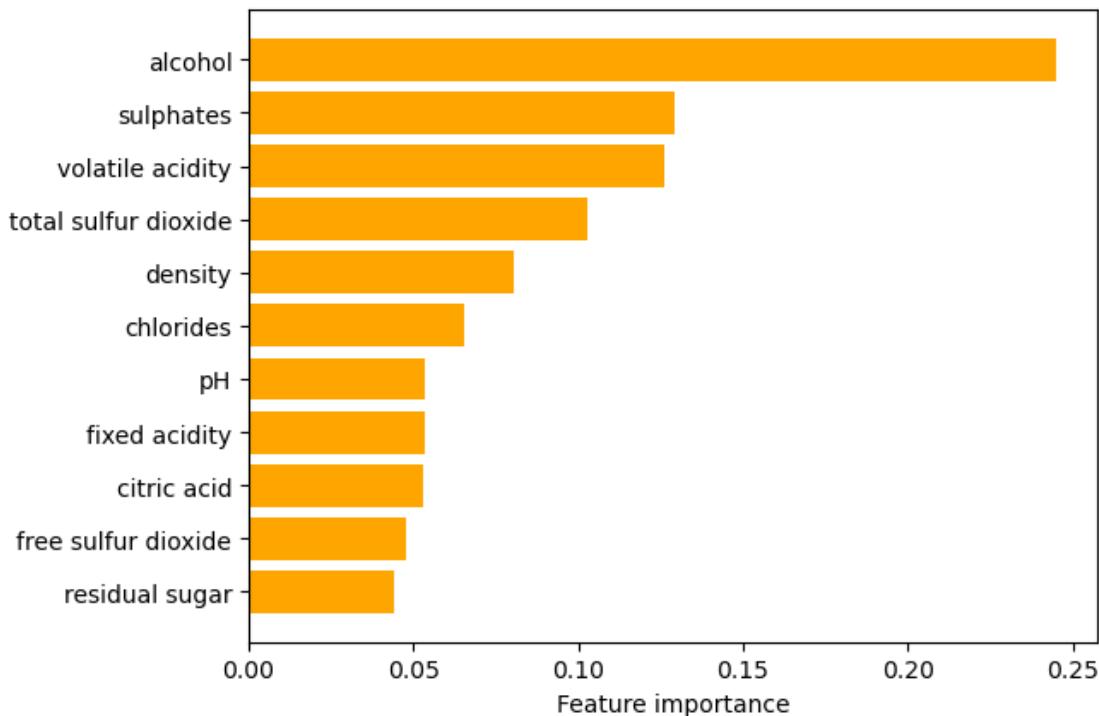


However, this is less insightful, as the decision is made by vote of many (different) trees! Still, we can study the **feature importance score**:

```
[54]: feature_scores = wine_forest.feature_importances_
feature_names = list(wine_features.columns)

important_features = pd.Series(feature_scores, index=feature_names).
    sort_values()

plt.barh(important_features.index, important_features, color='Orange')
plt.xlabel('Feature importance')
plt.show()
```



1.5.1 How to address imbalanced datasets?

As we have seen with Random Forests, looking only at the accuracy in imbalanced datasets can deceive. There are different approaches to rectify this problem; find the following link different [resampling strategies for imbalanced datasets](#).

1.5.2 How to tune hyperparameters?

A particular challenge with more advanced machine learning techniques is the choice of hyperparameters like `max_depth` / `tree_height` for decision trees and random forests. Typically, one attempts to define a reasonable scope for the range of different hyperparameters and identify more suitable values based on the **validation set** performance.

There exist tools which allow you an automated way to perform hyperparameter search. Check out for example this scikit-learn documentation on [Tuning the hyper-parameters of an estimator](#) and an [example for a random forest](#).

1.6 Exercise Section

(1.) In this exercise, we train a `Ridge` regressor for predicting the `quality` values on the test set `feat_test`. First, load the following cell:

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

from sklearn.model_selection import train_test_split

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')

ex_target = wine_data['quality']
ex_features = wine_data.drop(['quality'], axis=1)

feat_train, feat_test, target_train, target_test = train_test_split(
    ex_features, ex_target, test_size = 0.3, random_state=123)
```

```
[56]: feat_train.head(5)
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	\
374	14.0	0.410	0.63	3.8	0.089	
800	7.2	0.610	0.08	4.0	0.082	
1441	7.4	0.785	0.19	5.2	0.094	
1269	5.5	0.490	0.03	1.8	0.044	
691	9.2	0.920	0.24	2.6	0.087	
	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	\
374	6.0	47.0	1.00140	3.01	0.81	
800	26.0	108.0	0.99641	3.25	0.51	
1441	19.0	98.0	0.99713	3.16	0.52	
1269	28.0	87.0	0.99080	3.50	0.82	
691	12.0	93.0	0.99980	3.48	0.54	
	alcohol					

```
374    10.800000
800     9.400000
1441    9.566667
1269   14.000000
691     9.800000
```

[57]: `target_train.head(5)`

[57]:

	quality
374	6
800	5
1441	6
1269	8
691	5

Name: quality, dtype: int64

Put your result in the next cell and use `ex_pred_ridge` for the predicted quality values.

[]:

```
from sklearn.linear_model import Ridge

# Fill in

ex_pred_ridge = # Fill in
```

Execute the next cell to save the results in a summary data frame.

[]:

```
target_test = target_test.to_frame()
target_test['Ridge_predicted_quality'] = np.around(ex_pred_ridge, decimals=2)
target_test['Ridge_absolut_deviation'] = abs(target_test['quality'] -
                                             target_test['Ridge_predicted_quality'])
target_test
```

(2.) You can create a Random Forest not only for classification, but also regression. Make use of the `scikit-learn` method

[]:

```
from sklearn.ensemble import RandomForestRegressor
```

to make predictions on the wine quality based on the other features / predictors. Load the previous cell and train a the `RandomForestRegressor` for predicting the quality values on the test set `feat_test`. Put your result in the next cell and use `ex_pred_RF` for the predicted quality values.

Hint: You might want to revisit the steps for RF classification we saw above.

[]:

```
num_trees = 100
tree_height = 8

# Fill in

ex_pred_RF = # Fill in
```

```
[ ]: target_test['RF_predicted_quality'] = np.around(ex_pred_RF, decimals=2)
target_test['RF_absolut_deviation'] = abs(target_test['quality'] -
    ↪target_test['RF_predicted_quality'])
target_test
```

Finally, compare how the Random Forest and ridge regressor performed in comparison. For this, just execute the next cell.

```
[ ]: RF_pred_MSE = (target_test['RF_absolut_deviation']**2).mean()
Ridge_pred_MSE = (target_test['Ridge_absolut_deviation']**2).mean()

print(f"Mean squared error of RandomForestRegressor: {RF_pred_MSE:.2f}")
print(f"Mean squared error of Ridge: {Ridge_pred_MSE:.2f}")
```

1.7 Proposed Solutions

(1.) In this exercise, we train a Ridge regressor for predicting the `quality` values on the test set `feat_test`. First, load the following cell:

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

from sklearn.model_selection import train_test_split

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')

ex_target = wine_data['quality']
ex_features = wine_data.drop(['quality'], axis=1)

feat_train, feat_test, target_train, target_test = train_test_split(
    ex_features, ex_target, test_size = 0.3, random_state=123)
```

Put your result in the next cell and use `ex_pred_ridge` for the predicted quality values.

```
[68]: from sklearn.linear_model import Ridge

ridgereg = Ridge(alpha=1.0)
ridgereg.fit(feat_train, target_train)

ex_pred_ridge = ridgereg.predict(feat_test)

print(f"Rsquared_ridgereg: {ridgereg.score(feat_train, target_train)}")
```

`Rsquared_ridgereg: 0.3627326339932849`

Execute the next cell to save the results in a summary data frame.

```
[69]: target_test = target_test.to_frame()
target_test['Ridge_predicted_quality'] = np.around(ex_pred_ridge, decimals=2)
target_test['Ridge_absolut_deviation'] = abs(target_test['quality'] - target_test['Ridge_predicted_quality'])
target_test
```

	quality	Ridge_predicted_quality	Ridge_absolut_deviation
912	6	6.32	0.32
772	5	4.97	0.03
1037	5	4.77	0.23
1106	6	6.57	0.57
263	5	5.51	0.51
...
1466	7	5.57	1.43
580	5	5.32	0.32
1082	6	5.40	0.60
1279	7	6.31	0.69
1155	5	5.32	0.32

[480 rows x 3 columns]

(2.) You can create a Random Forest not only for classification, but also regression. Make use of the `scikit-learn` method

```
[70]: from sklearn.ensemble import RandomForestRegressor
```

to make predictions on the wine quality based on the other features / predictors. Load the previous cell and train a the `RandomForestRegressor` for predicting the quality values on the test set `feat_test`. Put your result in the next cell and use `ex_pred_RF` for the predicted quality values.

Hint: You might want to revisit the steps for RF classification we saw above.

```
[71]: num_trees = 100
tree_height = 8

ex_forest = RandomForestRegressor(n_estimators=num_trees,
                                   max_depth=tree_height, random_state=123)
ex_forest.fit(feat_train, target_train)
ex_pred_RF = ex_forest.predict(feat_test)

print(f"Rsquared_RF: {ex_forest.score(feat_train, target_train)}")
```

Rsquared_RF: 0.7545065944574373

```
[72]: target_test['RF_predicted_quality'] = np.around(ex_pred_RF, decimals=2)
target_test['RF_absolut_deviation'] = abs(target_test['quality'] - target_test['RF_predicted_quality'])
target_test
```

```
[72]:      quality  Ridge_predicted_quality  Ridge_absolut_deviation \
912          6                  6.32                 0.32
772          5                  4.97                 0.03
1037         5                  4.77                 0.23
1106         6                  6.57                 0.57
263          5                  5.51                 0.51
...
1466         7                  5.57                 1.43
580          5                  5.32                 0.32
1082         6                  5.40                 0.60
1279         7                  6.31                 0.69
1155         5                  5.32                 0.32

      RF_predicted_quality  RF_absolut_deviation
912                  6.41                 0.41
772                  5.10                 0.10
1037                 4.98                 0.02
1106                 6.44                 0.44
263                  5.73                 0.73
...
1466                  5.84                 1.16
580                  4.98                 0.02
1082                 5.49                 0.51
1279                 6.41                 0.59
1155                 5.18                 0.18
```

[480 rows x 5 columns]

Finally, compare how the Random Forest and ridge regressor performed in comparison. For this, just execute the next cell.

```
[73]: RF_pred_MSE = (target_test['RF_absolut_deviation']**2).mean()
Ridge_pred_MSE = (target_test['Ridge_absolut_deviation']**2).mean()

print(f"Mean squared error of RandomForestRegressor: {RF_pred_MSE:.2f}")
print(f"Mean squared error of Ridge: {Ridge_pred_MSE:.2f}")
```

Mean squared error of RandomForestRegressor: 0.34
 Mean squared error of Ridge: 0.42

3-Dim_Reduction_Clustering

May 7, 2024

1 3. Dimensionality Reduction and Clustering

In the previous notebook, we could rely on labels which supervised the learning algorithm. Next, we consider the unsupervised setting and knowledge discovery. We focus on the idea of identifying a subset of relevant dimensions which capture the most variation in the data. This is closely related to the idea of compression.

In this notebook, we cover

- Principal Component Analysis (PCA) for dimensionality reduction
- k-means clustering
- Gaussian Mixture Models (GMMs)

and apply these techniques to simulated data and two dataset examples (digit recognition).

Keywords: `MNIST`, `np.linalg.eig`, `sklearn.decomposition.PCA`, `sklearn.cluster.KMeans`, `sklearn.mixture.GaussianMixture`

1.1 Dimensionality Reduction

The topic of dimensionality reduction plays a vital role in Machine Learning. Typically, our datasets collect a great number of predictors or covariates or consist of images with hundreds of pixels. Getting an idea of the input space and relevant features is challenging in these high-dimensional problems. Three main use cases include:

1. **Identifying relevant features / dimensions** in the input space which as much as possible preserves the variance in the data. In the following simplified example, the black arrows indicate the dimensions of highest variance:

The longer arrow can be viewed as the **main direction / dimension of variance** which accounts for about 90% of total variance and thus as the “more relevant” feature.

2. **Data compression**: In many applications we can reduce the number of dimensions to capture the relevant information. In the previous image, we might choose the longer arrow as a compressed version of our data. Another compression example is shown in the next image:

Here, we encoded an image of Maxim’s cat (with an approach called *Fast Fourier Transformation*) into a much lower dimensional compressed image which only requires 9% of pixels compared to the original image. Decoding this compressed image reconstructs the orginial image almost perfectly.

3. **Visualising** high-dimensional data is generally difficult for dimensions larger than 3. Thus, dimensionality reduction is an important tool for visualisation and identifying patterns.
-

1.2 Principal Component Analysis

Principal Component Analysis (PCA) ([Wikipedia](#)) is such a standard technique used for the above purposes and is of great relevance in a great variety of different disciplines.

Let us generate simple 2-dimensional dataset to identify the main dimensions of variation.

```
[35]: import numpy as np
import matplotlib.pyplot as plt

np.random.seed(123)

num_datapoints = 1000

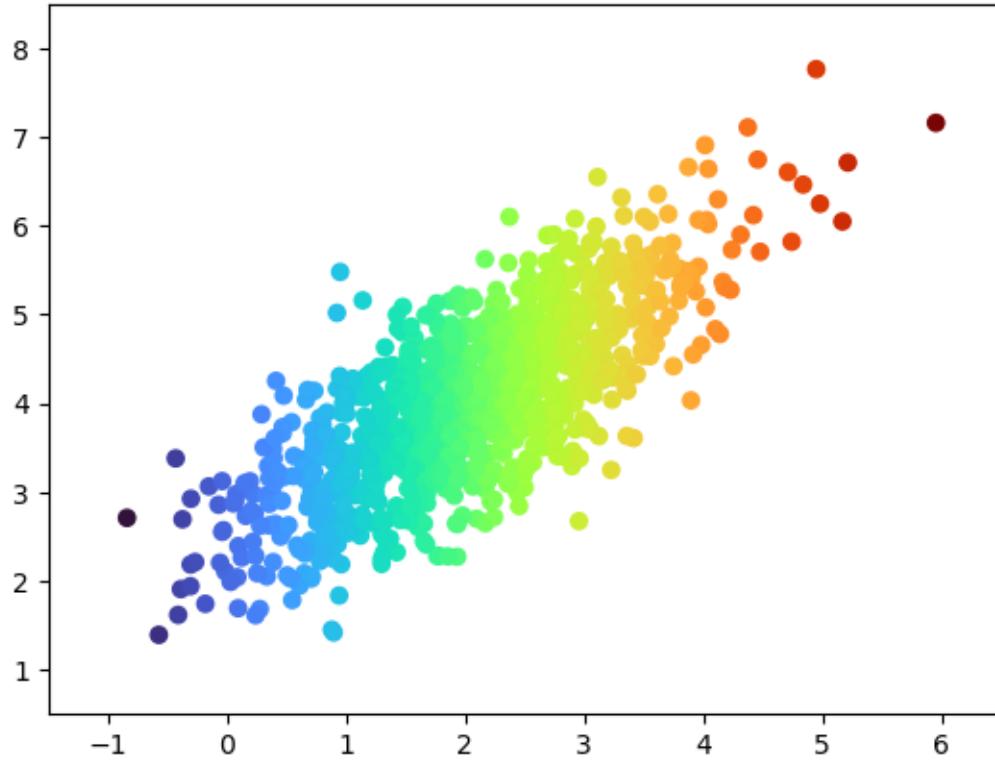
mu_generation = [2,4]
covariance_matrix_generation = np.array([[1,      0.75],
                                         [0.75,  1,    ],
                                         []])

gauss_2d_rotated = np.random.multivariate_normal(mean=mu_generation,
                                                    cov=covariance_matrix_generation,
                                                    size=num_datapoints)

plt.scatter(gauss_2d_rotated[:,0], gauss_2d_rotated[:,1], c=gauss_2d_rotated[:,0], cmap='turbo')

plt.ylim(0.5,8.5)
plt.xlim(-1.5,6.5)

plt.show()
```



The fundamental idea of PCA is to perform an eigendecomposition on the data covariance matrix $\text{cov}(X, X)$. If our dataset X is **centered**, i.e. the mean of all datapoints N is 0, we can write the (sample) covariance matrix as

$$\text{cov}(X, X) = \frac{1}{N-1} XX^\top. \quad (1)$$

The corresponding eigenvectors to the largest eigenvalues of this matrix provide the main axes of variation (principle components) in our dataset X .

Let's calculate these for our example and start by centering the dataset.

```
[36]: data_mean = np.mean(gauss_2d_rotated, axis=0)
data_centered = gauss_2d_rotated - data_mean
```

Next, calculate the eigenvalues and eigenvectors using `np.linalg.eig` and sort them in descending order.

```
[37]: cov_matrix = np.cov(data_centered, rowvar=False)

eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

idx = eigenvalues.argsort()[:-1]
eigenvalues = eigenvalues[idx]
```

```
eigenvectors = eigenvectors[:, idx]
```

Let's compare the calculated covariance and the covariance matrix we used to generate the data:

```
[38]: print(f"Calculated covariance:\n{cov_matrix}\n\nCovariance used for generation:\n{covariance_matrix_generation}")
```

Calculated covariance:

```
[[0.98354551 0.74236276]
 [0.74236276 0.97100139]]
```

Covariance used for generation:

```
[[1. 0.75]
 [0.75 1. ]]
```

Choose the two eigenvectors with the highest eigenvalues and project the data onto these vectors. These are our “new” x- and y-axes.

```
[39]: data_pca_transform = np.dot(data_centered, eigenvectors[:, :2])

fig, axs = plt.subplots(1,2, figsize=(10,3))

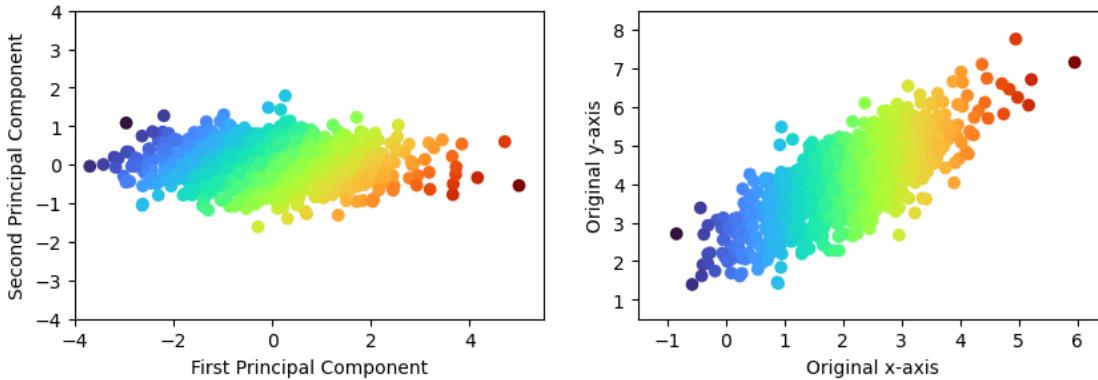
axs[0].scatter(data_pca_transform[:, 0], data_pca_transform[:, 1], c=gauss_2d_rotated[:,0], cmap='turbo')

axs[0].set_xlabel('First Principal Component')
axs[0].set_ylabel('Second Principal Component')
axs[0].set_xlim(-4,5.5)
axs[0].set_ylim(-4,4)

axs[1].scatter(gauss_2d_rotated[:,0], gauss_2d_rotated[:,1], c=gauss_2d_rotated[:,0], cmap='turbo')

axs[1].set_xlabel('Original x-axis')
axs[1].set_ylabel('Original y-axis')
axs[1].set_xlim(-1.5,6.5)
axs[1].set_ylim(0.5,8.5)

plt.show()
```



Note that in this 2d example, the **projection on the two eigenvectors corresponds to a rotation** of our dataset. This will be different, if the original dataset has more dimensions.

1.2.1 As before, we can just use scikit-learn

and its functionality with the function **PCA**!

```
[40]: from sklearn.decomposition import PCA

num_eigvectors = 2

data_pca = PCA(n_components=num_eigvectors)
data_pca.fit(gauss_2d_rotated)

data_pca_transform = data_pca.transform(gauss_2d_rotated)
```

We can even check the explained variance ratio of the first two principal components:

```
[41]: print(data_pca.explained_variance_ratio_)

[0.87982678 0.12017322]
```

1.2.2 MNIST Example

As a first, more interesting example, we examine the MNIST dataset!

```
[42]: mnist_data = np.load('data/mnist_data_5k.npy', allow_pickle=True)
mnist_targets = np.load('data/mnist_labels_5k.npy', allow_pickle=True)

fig, axs = plt.subplots(1,5)

for i, ax in enumerate(axs.ravel()):
    ax.imshow(mnist_data[i].reshape(28,28))

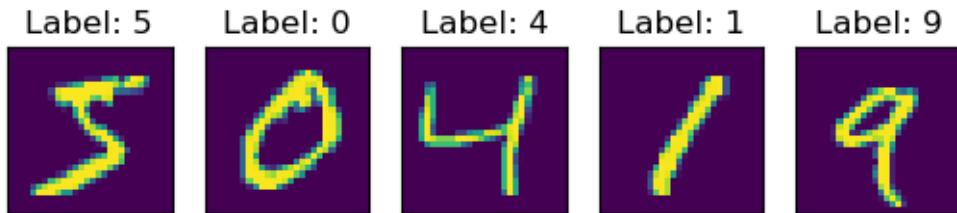
    ax.set_title(f"Label: {mnist_targets[i]}")
```

```

ax.set_xticks([])
ax.set_yticks([])

plt.show()

```



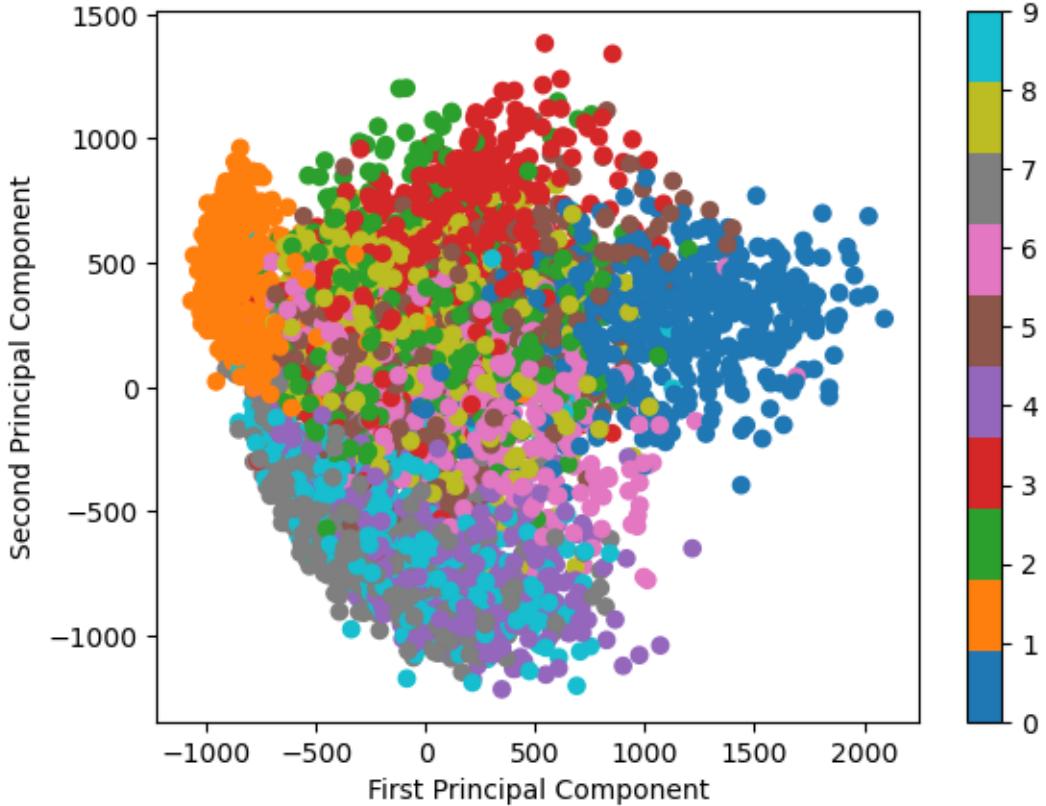
Let's see what the projection on the first two eigenvectors looks like in MNIST:

```

[67]: data_pca = PCA(n_components=2)
data_pca.fit(mnist_data)
data_pca_transform_mnist = data_pca.transform(mnist_data)

plt.scatter(data_pca_transform_mnist[:, 0], data_pca_transform_mnist[:, 1], c=mnist_targets, cmap='tab10')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.colorbar()
plt.show()

```



The result shows that **similar numbers are clustered** and thus that samples from the **same class share similar features**.

Let's examine the explained variance ratios:

```
[51]: print(data_pca.explained_variance_ratio_)
```

```
[0.09867566 0.07404546]
```

These two dimensions account for **less than 20% of the variance**. In other words, we might miss about 80% of relevant variation in our data and should not base our computation solely on these first two principal components!

1.3 Clustering

Another important approach to identifying patterns in an unsupervised fashion is clustering. In the previous examples, we have already seen instances where datapoints might form clusters indicating similar features.

Two popular approaches to clustering are **k-means** and **Gaussian Mixture Models** which we consider in the following.

k-means The underlying idea of k-means ([Wikipedia](#)) is assign datapoints to k cluster means / centroids in the dataset. The following GIF illustrates this for $k = 3$ centroids:

GIF source: Wikipedia

More formally, we consider a set of N datapoints $X = (x_1, \dots, x_N)$ which we try to group into k groups forming sets S_1, S_2, \dots, S_k . The objective function is given by

$$\operatorname{argmin}_S \sum_{i=1}^k \sum_{x \in S_i} \|x - \mu_i\|^2 \quad (2)$$

where $\mu_i = \frac{1}{|S_i|} \sum_{x \in S_i} x$ is the mean of the datapoints in cluster S_i .

The algorithm **starts** with placing the cluster means / centroids μ_i in a random position (see GIF above at Iteration #0). Datapoints are assigned to the closest mean μ_i , i.e. to cluster S_i , and a **new cluster mean** is calculated. The algorithm **iterates** this step several times until the cluster assignments remain unchanged.

Note that in an unsupervised setting, **we generally do not know the number of clusters k !** Thus, k is a *hyper-parameter* we need to choose!

Let's create a clustered dataset with two clusters of 100 datapoints each, i.e. $N = 200$.

```
[52]: import numpy as np
import matplotlib.pyplot as plt

np.random.seed(123)

num_datapoints = 100

cluster_1 = np.random.normal( 2, 1, size=(num_datapoints, 2))
cluster_2 = np.random.normal(-2, 1, size=(num_datapoints, 2))

cluster_data = np.concatenate([cluster_1, cluster_2])
```

We make use of the `scikit-learn` implementation of `KMeans` and choose $k = 2$.

```
[53]: from sklearn.cluster import KMeans

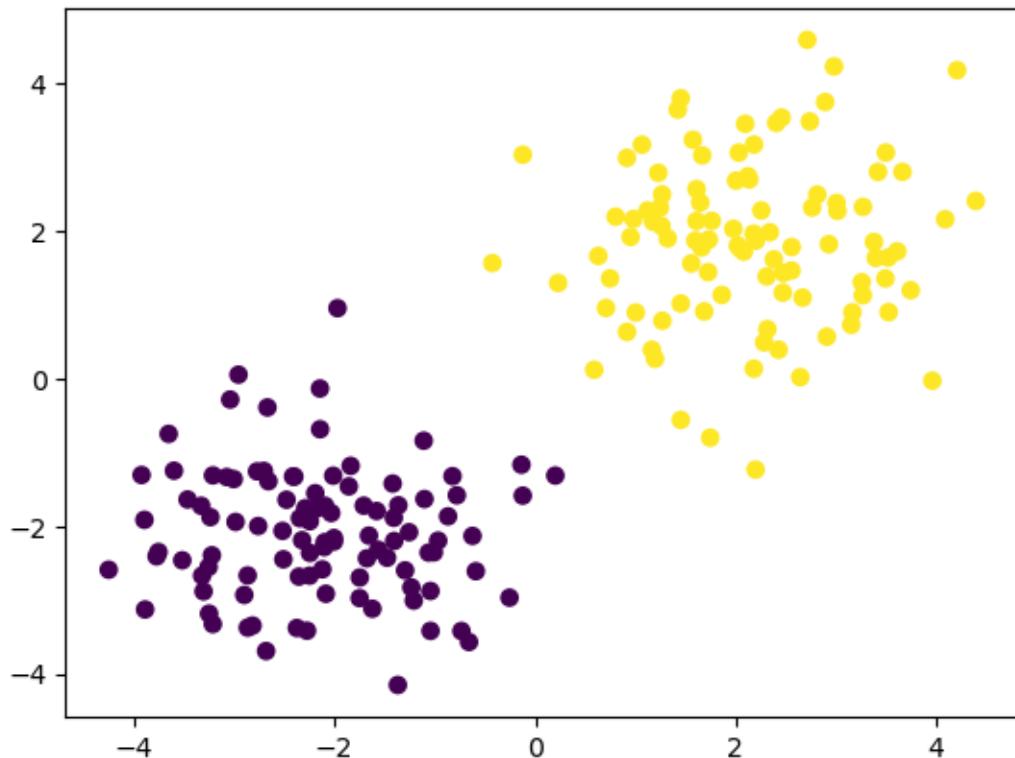
k = 2

kmeans = KMeans(n_clusters=k, random_state=123, n_init=10)
kmeans.fit(cluster_data)

cluster_prediction = kmeans.predict(cluster_data)

plt.scatter(cluster_data[:,0], cluster_data[:,1], c=cluster_prediction)
plt.show()
```

```
print(cluster_prediction)
```



1.3.1 Gaussian Mixture Model

A Gaussian Mixture Model (GMM) ([Wikipedia](#)) can be viewed as a (probabilistic) relaxation of k-means. Instead of “hard” cluster assignments, we consider “soft” assignments by means of probabilities.

An GMM assumes a mixture of several Gaussian distributions as the underlying distribution for the dataset and observed clustering (which can be a strong assumption, in practice). Each Gaussian distribution $\mathcal{N}(\mu_i, \Sigma_i)$ forms a cluster and is characterized by mean μ_i and covariance matrix Σ_i . The GMM (prior) distribution is then given by the mixture model

$$p(\mu, \Sigma) = \sum_{i=1}^k w_i \mathcal{N}(\mu_i, \Sigma_i) \quad (3)$$

where $\mu = (\mu_1, \dots, \mu_k)$, $\Sigma = (\Sigma_1, \dots, \Sigma_k)$, and $w = (w_1, \dots, w_k)$ is the weighting of the different mixture components.

As before, we start by choosing a number of k clusters and setting the parameters μ, Σ to some initial values. The cluster assignment and parameters are then updated iteratively with the **expectation-maximisation (EM) algorithm** until cluster assignments, i.e. the parameters μ, Σ , do not change any longer.

Let's consider the same dataset as for k-means and make use of the `scikit-learn GaussianMixture` method.

```
[54]: from sklearn.mixture import GaussianMixture

k = 2

GMM = GaussianMixture(n_components=k)
GMM.fit(cluster_data)
```

```
[54]: GaussianMixture(n_components=2)
```

In order to provide the probabilities, we need to obtain a grid of datapoints which can do with `numpy.meshgrid`.

```
[55]: x_min = cluster_data[:, 0].min() - 1
x_max = cluster_data[:, 0].max() + 1

y_min = cluster_data[:, 1].min() - 1
y_max = cluster_data[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                      np.linspace(y_min, y_max, 100))
```

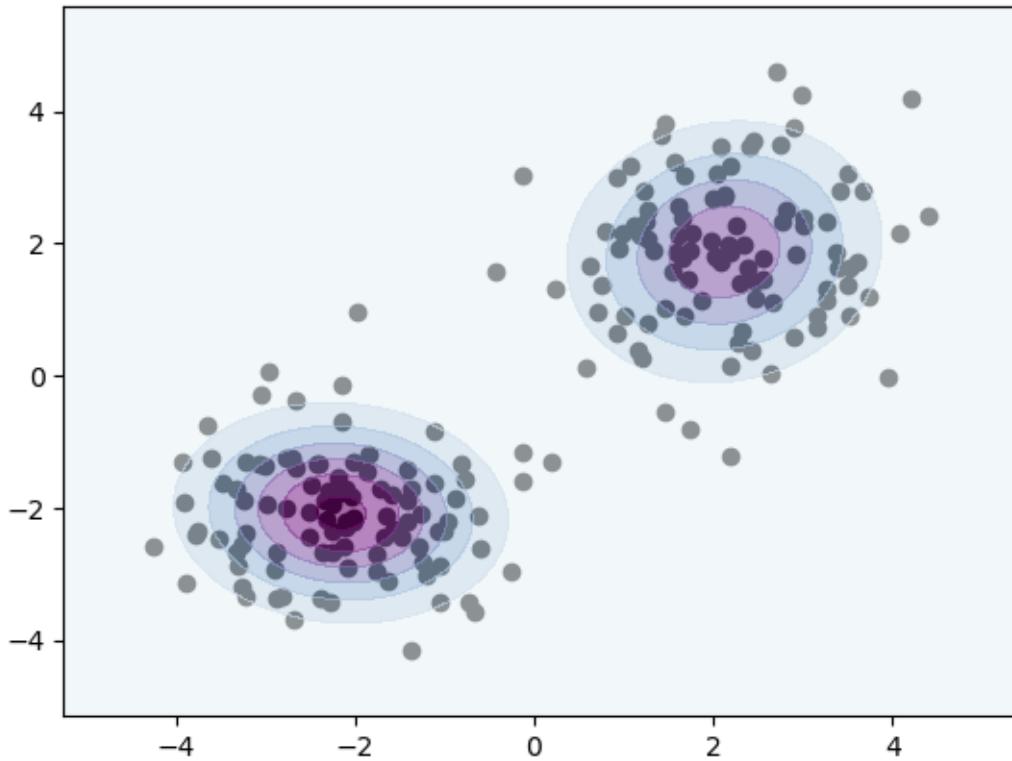
Finally, we can obtain the probability predictions with `score_samples` and plot the results. Note that this method provides the log-likelihood, i.e. we need to exponentiate these outputs!

```
[56]: probab = GMM.score_samples(np.c_[xx.ravel(), yy.ravel()])
probab = np.exp(probab)
probab = probab.reshape(xx.shape)

plt.scatter(cluster_data[:, 0], cluster_data[:, 1], c='black')
plt.contourf(xx, yy, probab, alpha=0.6, cmap='BuPu')

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)

plt.show()
```



Note that `np.c_[A,B]` concatenates NumPy arrays `A` and `B` along the second axis.

1.4 Exercise Section

In the next two exercises, we use k-means and a Gaussian Mixture Model on the projection of the MNIST data on its two first principal components.

(1.) Firstly, to simplify the clustering task, we will only consider digits 0, 3, 6, and 9. For this make us of `%3` to identify all of these labels in `mnist_targets` and select only the respective elements in `data_pca_transform_mnist` and `mnist_targets` (defined above). Provide the filtered datasets `ex_data` and the corresponding labels in `ex_targets` in the following cell.

```
[ ]: ex_data = # fill in
      ex_targets = # fill in
```

If you were successful, you can plot the filtered dataset below, which should look like the following plot:

```
[ ]: plt.scatter(ex_data[:, 0], ex_data[:, 1], c=ex_targets, cmap='tab10')
      plt.xlabel('First Principal Component')
      plt.ylabel('Second Principal Component')
      plt.colorbar()
```

```
plt.show()
```

(2.) Implement a k-means approach for clustering `ex_data` and provide the k-means object as `kmeans` and cluster predictions in `cluster_prediction`.

```
[ ]: # fill in
```

Use `cluster_prediction` and the above defined `kmeans` to plot the clusters with the following cell.

```
[ ]: plt.scatter(ex_data[:,0], ex_data[:,1], c=cluster_prediction)
plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1], marker='X', c='red')
plt.show()
```

(3.) Implement an Gaussian Mixture Model approach for clustering `ex_data`. Provide the Gaussian Mixture Model as object `GMM`, the grid coordinates `xx` and `yy` (as seen before) and the predictions in `probab`.

```
[ ]: # fill in
```

If you have defined `GMM`, `xx`, `yy`, and `probab` in the cell, the next cell will allow you to plot your result.

```
[ ]: plt.scatter(ex_data[:, 0], ex_data[:, 1], c='black')
plt.contourf(xx, yy, probab, alpha=0.6, cmap='BuPu')
plt.scatter(GMM.means_[:,0],GMM.means_[:,1], marker='X', c='orange')

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)

plt.show()
```

1.5 Proposed Solutions

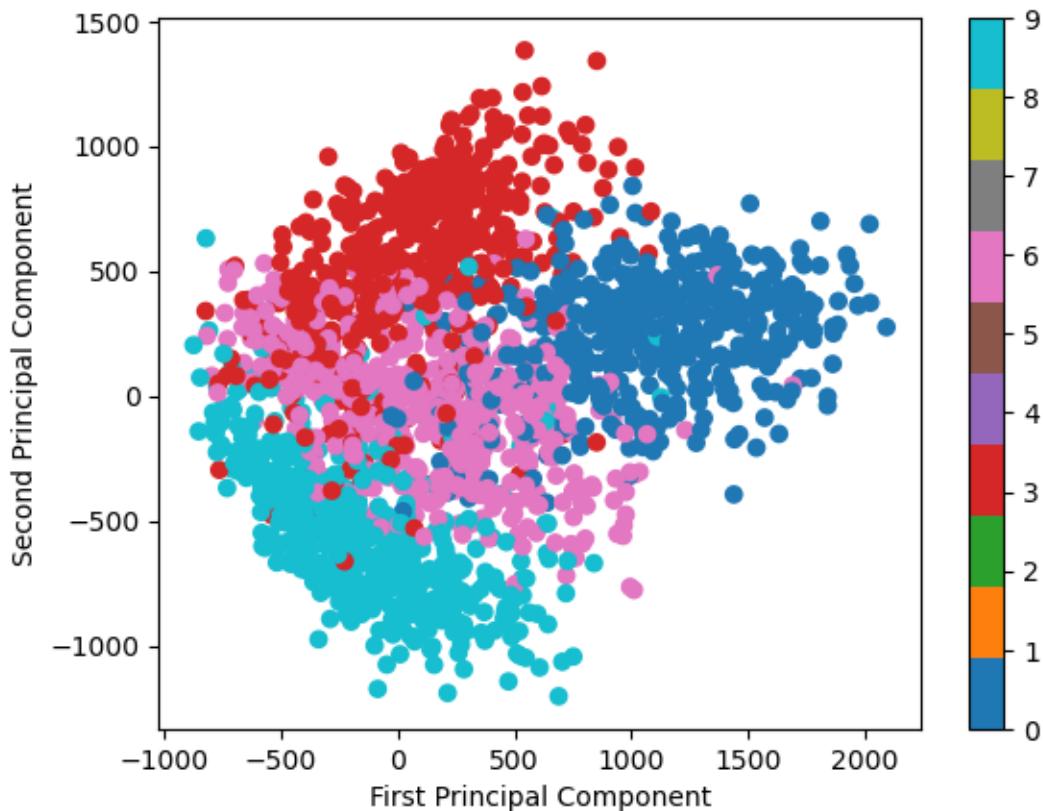
In the next two exercises, we use k-means and a Gaussian Mixture Model on the projection of the MNIST data on its two first principal components.

(1.) Firstly, to simplify the clustering task, we will only consider digits 0, 3, 6, and 9. For this make us of `%3` to identify all of these labels in `mnist_targets` and select only the respective elements in `data_pca_transform_mnist` and `mnist_targets` (defined above). Provide the filtered datasets `ex_data` and the corresponding labels in `ex_targets` in the following cell

```
[61]: ex_data = data_pca_transform_mnist[mnist_targets%3==0]
ex_targets = mnist_targets[mnist_targets%3==0]
```

If you were successful, you can plot the filtered dataset below, which should look like the following plot:

```
[62]: plt.scatter(ex_data[:, 0], ex_data[:, 1], c=ex_targets, cmap='tab10')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.colorbar()
plt.show()
```



(2.) Implement an k-means approach for clustering `ex_data` and provide the k-means object as `kmeans` and cluster predictions in `cluster_prediction`.

```
[63]: from sklearn.cluster import KMeans

k = 4

kmeans = KMeans(n_clusters=k, random_state=123)
kmeans.fit(ex_data)

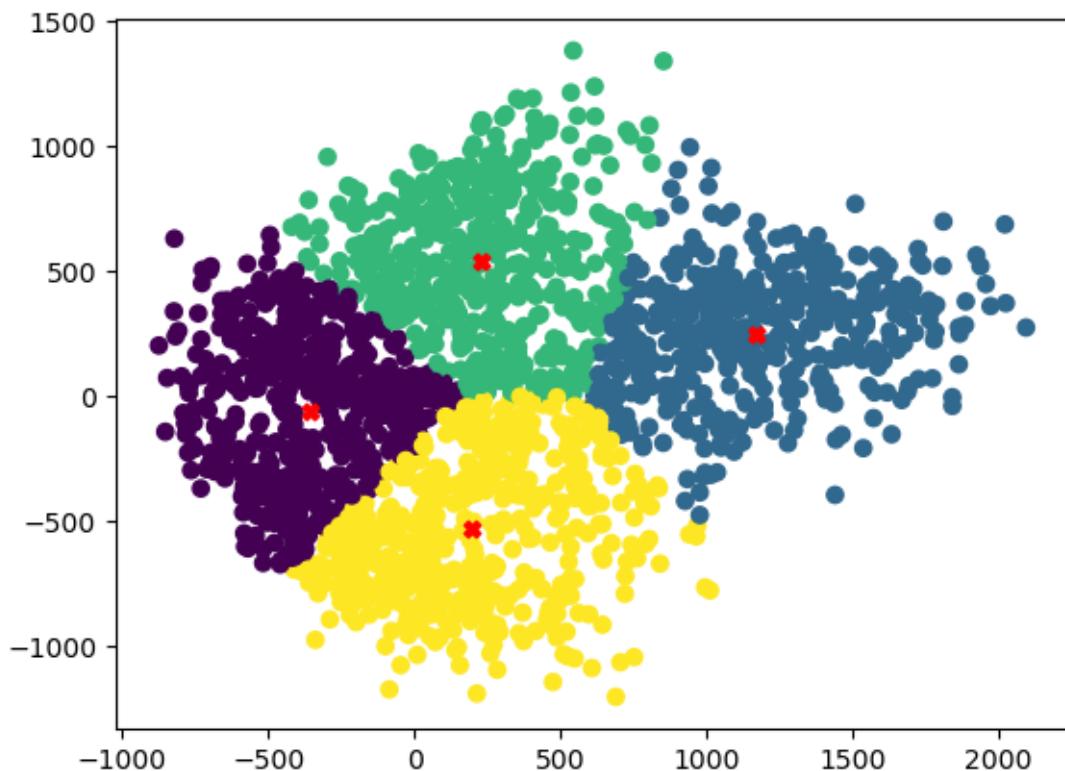
cluster_prediction = kmeans.predict(ex_data)
```

/Users/samarinm/anaconda3/APML/lib/python3.11/site-packages/sklearn/cluster/_kmeans.py:1412: FutureWarning: The default value of `n_init` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning

```
super().__check_params_vs_input(X, default_n_init=10)
```

Use `cluster_prediction` and the above defined `kmeans` to plot the clusters with the following cell.

```
[64]: plt.scatter(ex_data[:,0], ex_data[:,1], c=cluster_prediction)
plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1], marker='X', c='red')
plt.show()
```



(3.) Implement an Gaussian Mixture Model approach for clustering `ex_data`. Provide the Gaussian Mixture Model as object `GMM`, the grid coordinates `xx` and `yy` (as seen before) and the predictions in `probab`.

```
[65]: from sklearn.mixture import GaussianMixture
k = 4
GMM = GaussianMixture(n_components=k)
GMM.fit(ex_data)
x_min = ex_data[:, 0].min() - 1
x_max = ex_data[:, 0].max() + 1
```

```

y_min = ex_data[:, 1].min() - 1
y_max = ex_data[:, 1].max() + 1

xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                      np.linspace(y_min, y_max, 100))

probab = GMM.score_samples(np.c_[xx.ravel(), yy.ravel()])
probab = np.exp(probab)
probab = probab.reshape(xx.shape)

```

If you have defined GMM, xx, yy, and probab in the cell, the next cell will allow you to plot your result.

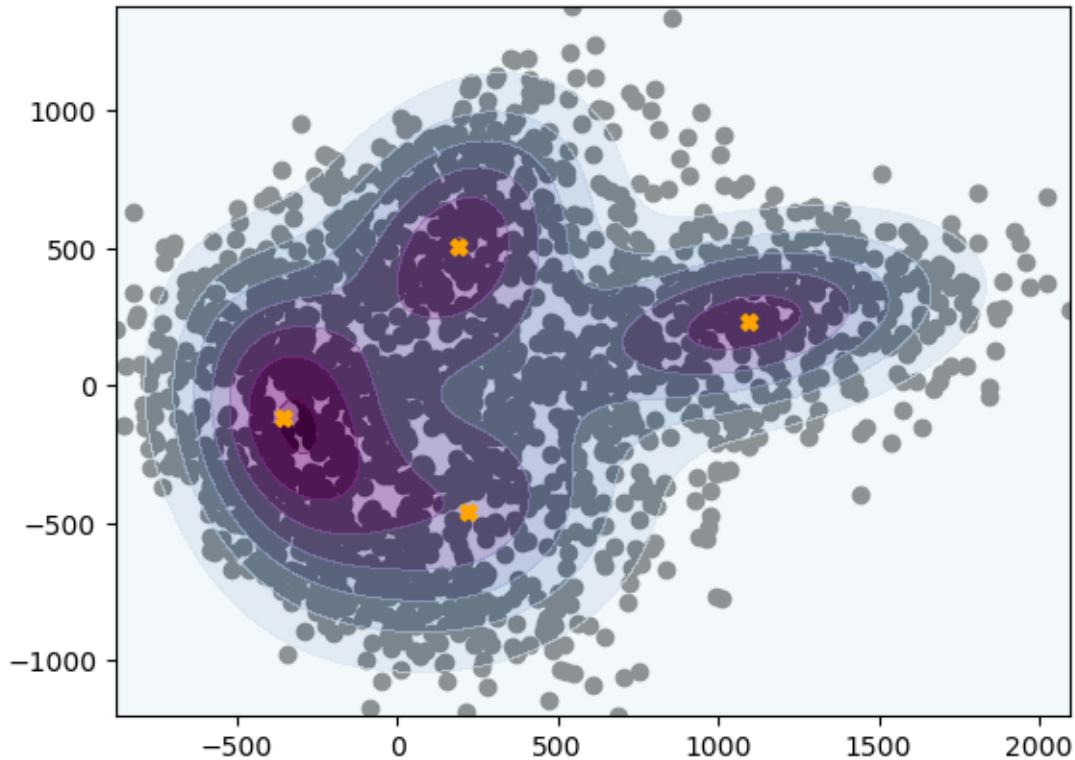
```

[66]: plt.scatter(ex_data[:, 0], ex_data[:, 1], c='black')
plt.contourf(xx, yy, probab, alpha=0.6, cmap='BuPu')
plt.scatter(GMM.means_[:,0],GMM.means_[:,1], marker='X', c='orange')

plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)

plt.show()

```



```
[68]: print(f"GMM cluster means = \n{GMM.means_}\n")
print(f"k-means cluster means = \n{kmeans.cluster_centers_}")
```

```
GMM cluster means =
[[ 189.23099224  509.27570034]
 [-353.88608806 -119.44188012]
 [1096.73107558  233.27452475]
 [ 221.6598133 -461.5002517 ]]
```

```
k-means cluster means =
[[-358.65060915  -62.02974473]
 [1172.16601688  245.21903822]
 [ 231.10451936  539.90273125]
 [ 197.0679518  -531.87250621]]
```

4-Kernel_Methods

May 7, 2024

1 4. Kernel Methods

In contrast to the idea of dimensionality reduction and compression, another popular direction in Machine Learning considers the opposite approach of solving problems in a higher-dimensional space. In this notebook we present the general idea of **kernel methods** on the example of

- Kernel ridge regression and
- Gaussian Processes

mostly focusing on simulated data to highlight their main benefits.

Keywords: `feature mapping`, `RBF kernel`, `sklearn.kernel_ridge.KernelRidge`, `sklearn.gaussian_process.GaussianProcessRegressor`, `sklearn.gaussian_process.kernels.RBF`, `sklearn.pipeline.make_pipeline`, `sklearn.preprocessing.StandardScaler`

1.1 Lifting Data into higher Dimensions

All models considered in the previous notebooks can be regarded as essentially linear models. Although linear models have their advantages (in particular with respect to mathematical derivations and guarantees), they severly limit the kind of functions we can represent. A common approach therefore is to **map** the problem from the input space into a **higher-dimensional feature space** as illustrated in the next figure for a classification setting.

The feature mapping ϕ allows us to map data from a lower-dimensional input space into a higher-dimensional feature space. The key idea is that identifying the complicated decision boundary in input space might be challenging, but **in feature space a linear model suffices** to identify the planar decision boundary.

More importantly, we can show that we do not need to ever evaluate $\phi(x)$. We can rely on the **kernel trick!** This implies that we only need to evalute scalar products of feature mappings – called kernels – $\kappa(x, x') = \langle \phi(x), \phi(x') \rangle$ for pairs of datapoints x, x' in our dataset X .

One such wildly used kernel is the **radial basis function (RBF) kernel**

$$\kappa(x, x') = \exp\left(-\frac{1}{2\sigma^2}\|x - x'\|_2^2\right) \quad (1)$$

which corresponds to an infinite-dimensional feature space. In general, we can design task-dependent kernels which might extract features we deem relevant to our problem. Furthermore, we can “kernelise” standard linear approaches to become more powerful!

1.2 Kernel Ridge Regression

We revisit our standard regression techniques and see how the kernelised version extends what we have seen before. We formulated the linear regression model before as

$$Y = Xw + \epsilon. \quad (2)$$

Let's simulate a sinusoidal dataset (with noise) of $N = 200$ datapoints.

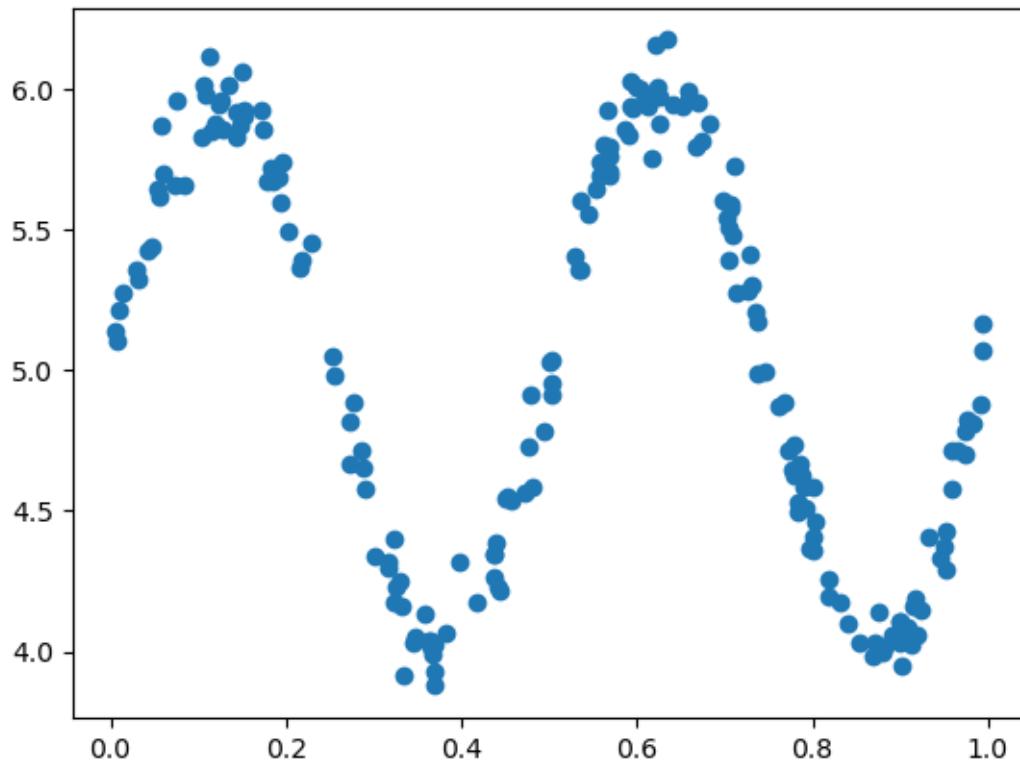
```
[1]: import numpy as np
import matplotlib.pyplot as plt

np.random.seed(1234)

num_datapoints = 200
X = np.random.rand(num_datapoints)
X = np.sort(X).reshape(-1,1)

Y = np.sin(4 * np.pi * X) + np.random.randn(num_datapoints, 1) * 0.1 + 5

plt.scatter(X,Y)
plt.show()
```



As we have seen before, we will try to fit these observations with a polynomial with linear and ridge regression.

In the following, we make use of an advanced `scikit-learn` tool. We build a `pipeline` (see [documentation](#)) which allows data processing steps to be executed with an estimator like `LinearRegression` or `Ridge`. In particular, we use the `StandardScalar` method (see [documentation](#)) which is a method to standardise our dataset, i.e. subtract the sample mean and divide by the sample standard deviation. With such a pipeline we can do some “data curation” before training!

```
[2]: from sklearn.linear_model import LinearRegression, Ridge
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

X_poly = np.hstack((X, X ** 2, X ** 3, X ** 4, X ** 5))

linreg = make_pipeline(StandardScaler(), LinearRegression())
linreg.fit(X_poly,Y)
Y_OLS = linreg.predict(X_poly)
Rsquared_lin = linreg.score(X_poly,Y)

ridgereg = make_pipeline(StandardScaler(), Ridge(alpha=0.001))
ridgereg.fit(X_poly,Y)
Y_ridge = ridgereg.predict(X_poly)
Rsquared_ridge = ridgereg.score(X_poly,Y)

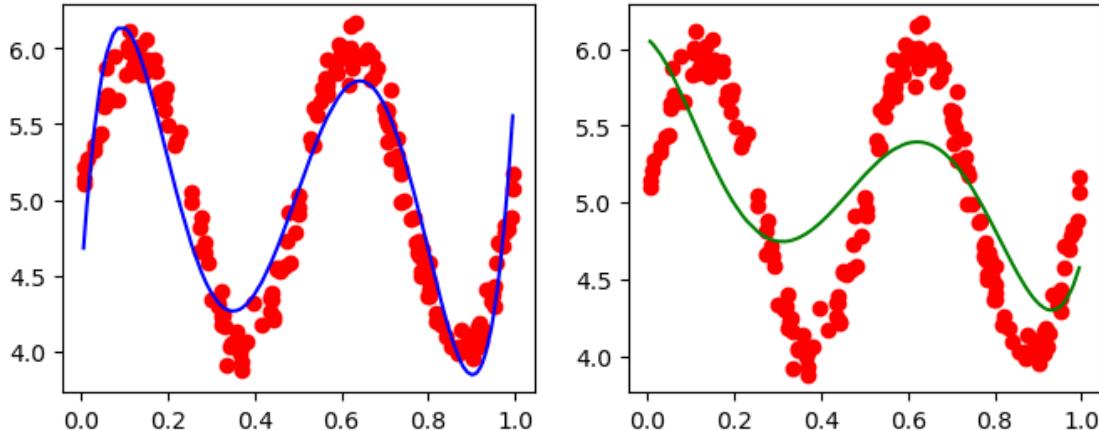
print(f"Rsquared_lin = {Rsquared_lin:.2f}, Rsquared_ridge = {Rsquared_ridge:.2f}")

fig, axs = plt.subplots(1,2, figsize=(8,3))

axs[0].scatter(X[:, 0], Y, color='red')
axs[0].plot(X[:, 0], Y_OLS, color='blue')

axs[1].scatter(X[:, 0], Y, color='red')
axs[1].plot(X[:, 0], Y_ridge, color='green')

plt.show()
```



1.2.1 Kernel Ridge Regression (KRR)

incorporates the idea of employing a feature mapping ϕ and we rewrite the linear regression model as

$$Y = \phi(X)w + \epsilon = f(x) + \epsilon. \quad (3)$$

Similar to the ridge regression result, the solution can be shown to be

$$w_{\text{KRR}} = \phi(X)^\top (\phi(X)^\top \phi(X) + \lambda \mathbb{1}_N)^{-1} Y. \quad (4)$$

and the learned function (without going into the mathematical details) is then given by

$$f(x) = K(x, X) (K(X, X) + \lambda \mathbb{1}_{d_N})^{-1} Y \quad (5)$$

with K being the kernel matrix $K_{nm} = \kappa(x_n, x_m) = \langle \phi(x_n), \phi(x_m) \rangle$.

Let's see how we can make use of for the above dataset and define the RBF kernel.

```
[3]: def rbf_kernel(x1, x2, sigma=1.0):
    return np.exp(-np.linalg.norm(x1 - x2) ** 2 / (2 * sigma ** 2))
```

As usually, we use a scikit-learn estimator: `KernelRidge`.

```
[4]: from sklearn.kernel_ridge import KernelRidge

KRR = make_pipeline(StandardScaler(), KernelRidge(kernel=rbf_kernel, alpha=0.
    ↪01, gamma=0.1))
KRR.fit(X, Y)
Y_KRR = KRR.predict(X)

Rsquared_KRR = KRR.score(X, Y)
```

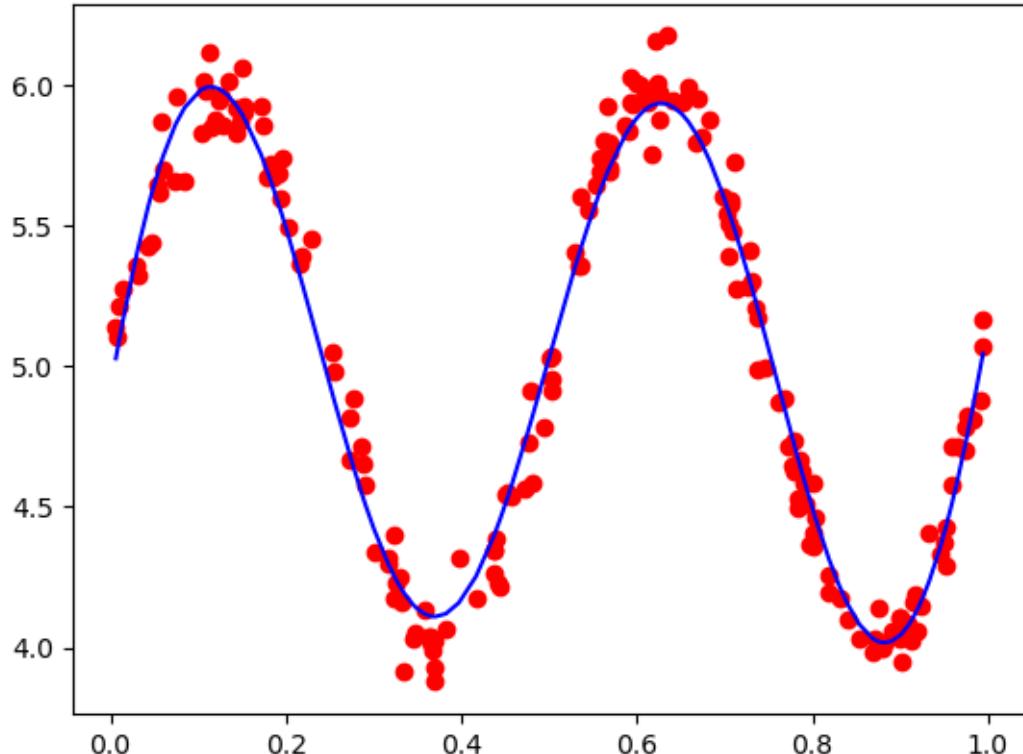
```

print(f"Rsquared_lin = {Rsquared_lin:.2f}, Rsquared_ridge = {Rsquared_ridge:.2f}, Rsquared_KRR = {Rsquared_KRR:.2f}")

plt.scatter(X[:, 0], Y, color='red')
plt.plot(X[:, 0], Y_KRR, color='blue')
plt.show()

```

Rsquared_lin = 0.89, Rsquared_ridge = 0.55, Rsquared_KRR = 0.98



1.3 Gaussian Processes

Kernel Ridge Regression has a close connection to **Gaussian Processes (GPs)** ([Wikipedia](#) or [scikit-learn](#)). Informally, a GP extends the notion of random variables to random functions. In a regression setting with input matrix X , the GP prior distribution $p(f | X) = \mathcal{N}(0, K)$ allows us to model the regression function $f(X) = [f(x_1), \dots, f(x_N)]^\top$ as

$$(f(x_1), \dots, f(x_N)) \sim \mathcal{N}(0, K) \quad (6)$$

with sample covariance matrix $K = \text{cov}(X, X)$. Let us consider a GP regression model with noisy responses Y as before, i.e.

$$Y = f(x) + \epsilon. \quad (7)$$

Leaving the mathematical derivation aside, in order to make a prediction for function $f(x)$ with new (test) data x , we can show that the posterior predictive distribution ([Wikipedia](#)) has the form

$$p(f | x, X, Y) = \mathcal{N}(\mu(x), \Sigma(x, X)) \quad (8)$$

$$\mu(x) = K(x, X)(K(X, X) + \lambda \mathbb{1}_N)^{-1} Y \quad (9)$$

$$\Sigma(x, X) = K(x, x) - K(x, X)^\top (K(X, X) + \lambda \mathbb{1}_N)^{-1} K(x, X) \quad (10)$$

Note that what we have written earlier for $f(x)$ in KRR **corresponds exactly** to $\mu(x)$ in GP regression, i.e. there is a close connection between KRR and GP regression (see e.g. [this comparison](#)). This motivates that the **covariance matrix K is a kernel**, too! Compared to KRR, GPs can be more flexible (as we can sample functions from the posterior) and we can make use of the GP formalism for training. Also, GPs enable us to quantify the uncertainty of a prediction due to $\Sigma(x, X)$.

Let's study this in a simplified example with $N = 4$ datapoints. As always, we can use `scikit-learn` methods for the `GaussianProcessRegressor` and different kernels. We employ the `RBF` kernel as well as a `ConstantKernel`, i.e. $\kappa(x, x') = \text{const}$.

```
[5]: from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, ConstantKernel

np.random.seed(1234)

num_datapoints = 4

X = np.random.rand(num_datapoints, 1)*2
Y = np.sin(2 * np.pi * X) + np.random.randn(num_datapoints, 1) * 0.1 + 5

kernel = ConstantKernel(constant_value=1.0, constant_value_bounds=(0.001, 1000))
kernel = kernel * RBF(length_scale=0.5, length_scale_bounds=(0.01, 100))

GPreg = GaussianProcessRegressor(kernel=kernel, alpha=0.1, n_restarts_optimizer=10)
GPreg.fit(X, Y)

X_pred = np.linspace(-2.0, 2.0, 100).reshape(-1, 1)
Y_GPreg, stddev_GPreg = GPreg.predict(X_pred, return_std=True)

X_pred = X_pred.reshape(-1)
Y_GPreg = Y_GPreg.reshape(-1)

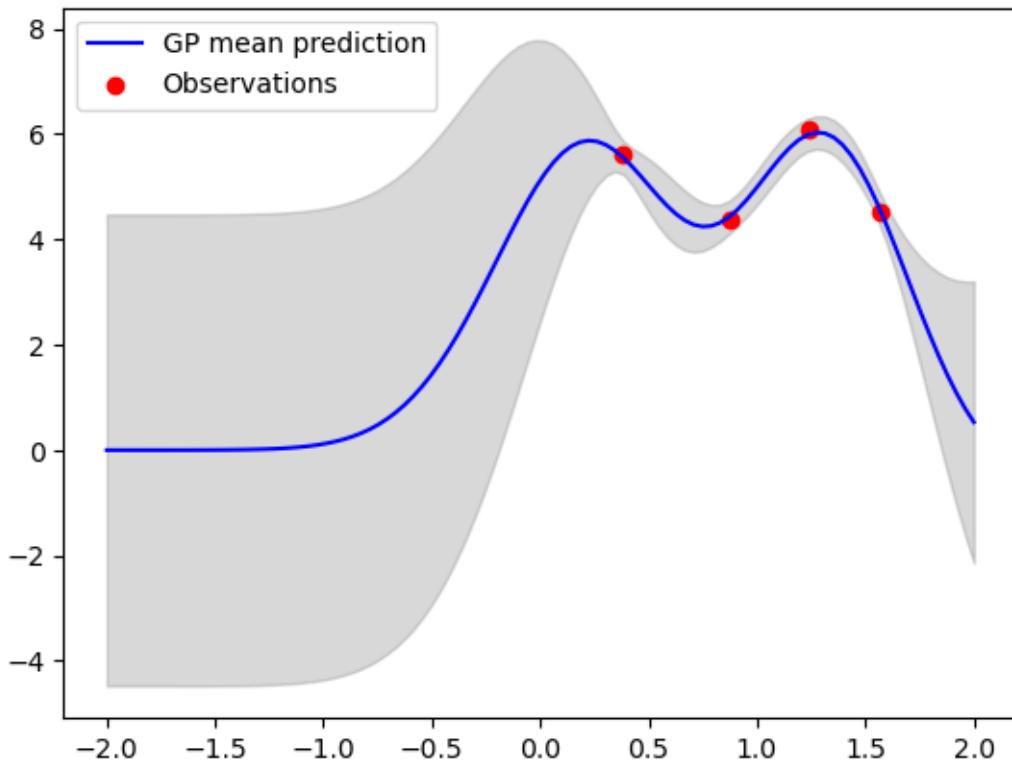
plt.fill_between(X_pred, Y_GPreg - stddev_GPreg, Y_GPreg + stddev_GPreg, alpha=0.3, color='grey')
```

```

plt.plot(X_pred, Y_GPreg, color='blue', label='GP mean prediction')
plt.scatter(X, Y, color='red', label='Observations')

plt.legend(loc='upper left')
plt.show()

```



1.4 Exercise Section

For the exercises, we consider again the red wine dataset but this time train a (1.) kernel ridge regressor and (2.) Gaussian Process regressor. As in notebook 2, we want to predict the quality values on the test set `feat_test` and compare the mean squared errors (MSEs) of KRR and GP regression to the previous results for ridge regression (0.42) and Random Forest regression (0.34).

In order, to prepare the data and kernel, load the following cell:

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

from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
```

```

from sklearn.preprocessing import StandardScaler
from sklearn.gaussian_process.kernels import RBF, ConstantKernel

np.random.seed(1234)

ex_kernel = RBF(length_scale=5, length_scale_bounds=(0.01, 100))

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')

ex_target = wine_data['quality']
ex_features = wine_data.drop(['quality'], axis=1)

feat_train, feat_test, target_train, target_test = train_test_split(
    ex_features, ex_target, test_size = 0.3, random_state=123)

target_test = target_test.to_frame()

```

[]: `feat_train.head(5)`

[]: `target_train.head(5)`

(1.) Implement kernel ridge regression as we have seen it in this notebook. Please provide the predictions for the test set `feat_test` in `ex_pred_KRR`.

[]: `# Fill in`

We use `ex_pred_KRR` in the following cell to provide the results. You can just execute the next cell.

[]: `target_test['KRR_predicted_quality'] = np.around(ex_pred_KRR, decimals=2)
target_test['KRR_absolut_deviation'] = abs(target_test['quality'] - target_test['KRR_predicted_quality'])
print(target_test)

KRR_pred_MSE = (target_test['KRR_absolut_deviation']**2).mean()
print(f"\nMean squared error of KernelRidge: {KRR_pred_MSE:.2f}")`

(2.) Implement GP regression as we have seen it in this notebook. Please provide the predictions for the test set `feat_test` in `ex_pred_GPreg`.

For Noto users: You might want to set `n_restarts_optimizer=1`(i.e. only one), as otherwise the computation might take too long.

[]: `# Fill in`

We use `ex_pred_GPreg` in the following cell to provide the results. You can just execute the next cell.

[]: `target_test['GPreg_predicted_quality'] = np.around(ex_pred_GPreg, decimals=2)
target_test['GPreg_absolut_deviation'] = abs(target_test['quality'] - target_test['GPreg_predicted_quality'])`

```

print(target_test)

GPreg_pred_MSE = (target_test['GPreg_absolut_deviations']**2).mean()

print(f"\nMean squared error of GaussianProcessRegressor: {GPreg_pred_MSE:.2f}")
print(f"Mean squared error of KernelRidge: {KRR_pred_MSE:.2f}")

```

1.5 Proposed Solutions

For the exercises, we consider again the red wine dataset but this time train a (1.) kernel ridge regressor and (2.) Gaussian Process regressor. As in notebook 2, we want to predict the quality values on the test set `feat_test` and compare the mean squared errors (MSEs) of KRR and GP regression to the previous results for ridge regression (0.42) and Random Forest regression (0.34).

In order, to prepare the data and kernel, load the following cell:

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

from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.gaussian_process.kernels import RBF, ConstantKernel

np.random.seed(1234)

ex_kernel = RBF(length_scale=5, length_scale_bounds=(0.01, 100))

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')

ex_target = wine_data['quality']
ex_features = wine_data.drop(['quality'], axis=1)

feat_train, feat_test, target_train, target_test = train_test_split(
    ex_features, ex_target, test_size = 0.3, random_state=123)

target_test = target_test.to_frame()
```

(1.) Implement kernel ridge regression as we have seen it in this notebook. Please provide the predictions for the test set `feat_test` in `ex_pred_KRR`.

```
[7]: from sklearn.kernel_ridge import KernelRidge

KRR_ex = make_pipeline(StandardScaler(), KernelRidge(kernel=ex_kernel, alpha=0.
    ↵1))
KRR_ex.fit(feat_train, target_train)
```

```
ex_pred_KRR = KRR_ex.predict(feat_test)

Rsquared_KRR_ex = KRR_ex.score(feat_train, target_train)
```

We use `ex_pred_KRR` in the following cell to provide the results. You can just execute the next cell.

```
[8]: target_test['KRR_predicted_quality'] = np.around(ex_pred_KRR, decimals=2)
target_test['KRR_absolut_deviation'] = abs(target_test['quality'] -
                                         target_test['KRR_predicted_quality'])
print(target_test)

KRR_pred_MSE = (target_test['KRR_absolut_deviation']**2).mean()
print(f"\nMean squared error of KernelRidge: {KRR_pred_MSE:.2f}")
```

	quality	KRR_predicted_quality	KRR_absolut_deviation
912	6	6.32	0.32
772	5	4.87	0.13
1037	5	5.04	0.04
1106	6	6.45	0.45
263	5	5.58	0.58
...
1466	7	5.60	1.40
580	5	4.71	0.29
1082	6	5.37	0.63
1279	7	6.25	0.75
1155	5	5.37	0.37

[480 rows x 3 columns]

Mean squared error of KernelRidge: 0.38

(2.) Implement GP regression as we have seen it in this notebook. Please provide the predictions for the test set `feat_test` in `ex_pred_GPreg`.

For Note users: You might want to set `n_restarts_optimizer=1`(i.e. only one), as otherwise the computation might take too long.

```
[9]: from sklearn.gaussian_process import GaussianProcessRegressor

GPreg_ex = make_pipeline(StandardScaler(), GaussianProcessRegressor(kernel=ex_kernel, alpha=0.15,
                                                                     n_restarts_optimizer=5))
GPreg_ex.fit(feat_train, target_train)
ex_pred_GPreg = GPreg_ex.predict(feat_test)

Rsquared_GPreg_ex = GPreg_ex.score(feat_train, target_train)
```

We use `ex_pred_GPreg` in the following cell to provide the results. You can just execute the next cell.

```
[10]: target_test['GPreg_predicted_quality'] = np.around(ex_pred_GPreg, decimals=2)
target_test['GPreg_absolut_deviation'] = abs(target_test['quality'] -_
    ↪target_test['GPreg_predicted_quality'])
print(target_test)

GPreg_pred_MSE = (target_test['GPreg_absolut_deviation']**2).mean()

print(f"\nMean squared error of GaussianProcessRegressor: {GPreg_pred_MSE:.2f}")
print(f"Mean squared error of KernelRidge: {KRR_pred_MSE:.2f}")
```

	quality	KRR_predicted_quality	KRR_absolut_deviation	\
912	6	6.32	0.32	
772	5	4.87	0.13	
1037	5	5.04	0.04	
1106	6	6.45	0.45	
263	5	5.58	0.58	
...	
1466	7	5.60	1.40	
580	5	4.71	0.29	
1082	6	5.37	0.63	
1279	7	6.25	0.75	
1155	5	5.37	0.37	

	GPreg_predicted_quality	GPreg_absolut_deviation
912	6.31	0.31
772	4.86	0.14
1037	5.04	0.04
1106	6.44	0.44
263	5.58	0.58
...
1466	5.59	1.41
580	4.74	0.26
1082	5.36	0.64
1279	6.23	0.77
1155	5.38	0.38

[480 rows x 5 columns]

Mean squared error of GaussianProcessRegressor: 0.38
 Mean squared error of KernelRidge: 0.38

5-Neural_Networks

May 7, 2024

1 5. Neural Networks

In the previous notebook, we have seen that we can choose feature mappings ϕ or kernels κ to design powerful non-linear models. Informally, deep learning can be characterised as trying to learn the feature mapping ϕ from data.

In this notebook we will discuss and implement our first

- feed-forward neural network / multi-layer perceptron and
- convolutional neural network

for the MNIST and wine quality classification task.

Keywords: Gradient descent, Cross-entropy loss, ReLU, One-hot encoding, Max pooling, Convolution, keras.layers, model.compile, model.fit, model.evaluate, model.save, model.summary, models.load_model, keras.optimizers.SGD

1.1 Neural Network Basics

We start with the classical **Feed-Forward Neural Networks (FFNNs)** which are also referred to as multi-layer perceptrons (MLPs). FFNNs were originally proposed as a computational model simulating mechanisms of the human brain. The general idea is that inputs are processed in a sequential manner by **artificial neurons**. The common architecture consists of an input layer, one to several hidden layers, and an output layer as illustrated in the following figure.

As before, we try to learn the function $f(X)$. The displayed three-layer FFNN **parameterises** this function in the following way

$$f(X, W) = \sigma(\sigma(XW_1)W_2)W_3 = \phi(X, W_1, W_2)W_3, \quad (1)$$

where weights $W = (W_1, W_2, W_3)$ are the matrices which we multiply layer-wise to the output of the previous layer. The outputs are called **activations**. An important part of why neural networks are so powerful is the use of **non-linear activation functions**. The **rectified linear unit (ReLU)** $\sigma(z) = \max\{0, z\}$ is a particularly popular choice, e.g. $\sigma(z = -2) = 0$ and $\sigma(z = 2) = 2$, with z being the preactivation.

We can make a connection between neural networks and what we have seen before. Let's consider a deeper L -layer FFNN. In the regression setting, our goal is again to find a model for

$$Y = f(X, W) + \epsilon = \phi(X, W_1, \dots, W_{L-1})W_L + \epsilon \quad (2)$$

where this time ϕ is learned! We calibrate our model to solve the task by using (**Stochastic Gradient Descent**) for updating the weights $W = (W_1, \dots, W_L)$:

$$W^{(t+1)} = W^{(t)} - \eta \nabla \text{Loss}(W^{(t)}). \quad (3)$$

As illustrated in the figure below, the idea is to start at some position of our loss landscape described by initially random weights $W^{(t=0)}$ drawn from a Gaussian distribution. With every update step, we approach a (local) minimum until the weights no longer change. Note that this requires an appropriate choice of the step size or **learning rate** η (which again is a hyperparameter like λ and k which we've seen before).

As before, we can use the mean-squared error loss for regression tasks

$$\text{Loss}(W) = \frac{1}{2N} \|Y - f(X, W)\|_2^2 = \frac{1}{2N} \sum_{n=1}^N (y_n - f(x_n, W))^2. \quad (4)$$

In the classification setting, we use the multi-class version of the logistic loss encountered earlier. In deep learning context, we usually call it the **cross-entropy loss**

$$\text{Loss}(W) = - \sum_{n=1}^N \sum_{c=1}^C y_{n,c} \log p_{n,c} = - \sum_{n=1}^N \sum_{c=1}^C y_{n,c} \log f_c(x_n, W), \quad (5)$$

where we consider C different classes our N datapoints belong to. The neural network provides in the output layer the probabilities for the different classes $p_{n,c} = f_c(x_n, W)$. For this, we use another kind of activation function known as the **softmax activation function** which maps the activation to the values between 0 and 1. The activation $\sigma(z)_i$ at output neuron i is given by

$$\sigma(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)}, \quad (6)$$

where C is the number of neurons in the output layer, i.e. the number of classes, and $i, j \in 1, \dots, C$. Typically, we change the class labels into a **one-hot encoding** which provides something like a class probability associated with the label. Suppose our datapoint x_n shows the digit 9 and the corresponding target is $y_n = 9$. In the one-hot encoding we replace y_n with the vector

$$y_n = \begin{pmatrix} y_{n,0} \\ y_{n,1} \\ \vdots \\ y_{n,9} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}, \quad (7)$$

where we have 0s at every index but the class index, i.e. 9 in this case.

Let's use all this to build an FFNN for classifying MNIST digits!

```
[1]: import numpy as np
from tensorflow import keras

mnist_data = np.load('data/mnist_data_5k.npy', allow_pickle=True)
mnist_targets = np.load('data/mnist_labels_5k.npy', allow_pickle=True)
```

2024-05-06 11:07:38.448809: I tensorflow/core/platform/cpu_feature_guard.cc:182]

This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.

To enable the following instructions: SSE4.1 SSE4.2, in other operations, rebuild TensorFlow with the appropriate compiler flags.

In the following, we normalise the vector of pixel values to the $[0, 1]$ interval. It is typically advisable to scale all your predictors in X (i.e. the columns) such that values range between 0 and 1. Furthermore, we split the data into training, validation, and test datasets.

```
[2]: mnist_data_flat = mnist_data.reshape(-1, 784, 1) / 255

num_train = 3000
num_val = 1000
num_test = 1000

x_train = mnist_data_flat[:num_train]
y_train = mnist_targets[:num_train]

x_val = mnist_data_flat[num_train:num_train+num_val]
y_val = mnist_targets[num_train:num_train+num_val]

x_test = mnist_data_flat[-num_test:]
y_test = mnist_targets[-num_test:]
```

Now, we define a three-layer FFNN with the following layers sizes:

- 784 neurons in the input layer
- 1024 (hidden) neurons in the first hidden layer with ReLU activation
- 512 (hidden) neurons in the second hidden layer with ReLU activation
- 10 output neurons for our classes in the output layer with a softmax activation

```
[3]: ffnn = keras.Sequential([
    keras.layers.InputLayer(input_shape=(784,)),
    keras.layers.Dense(units=1024, activation='relu'),
    keras.layers.Dense(units=512, activation='relu'),
    keras.layers.Dense(units=10, activation='softmax')
], name='FFNN_example')
```

Next, we specify hyperparameters. As outlined before, the **learning rate** η is an important hyperparameter. In addition, we typically cannot (but also do not want to) use all datapoints for training at the same time. We choose small batches of the dataset and thus need to specify the **batch size**.

```
[4]: learning_rate = 0.01
batch_size = 16
num_epochs = 10

print(f"Shape of training set: {x_train.shape}")
print(f"Number of batches: {x_train.shape[0] / batch_size}")
```

Shape of training set: (3000, 784, 1)
 Number of batches: 187.5

A training **epoch** is concluded after we have seen the total number of batches. We set multiple epochs to iteratively refine the neural network weights, i.e. training samples are seen several times during training.

Now, let's define our optimisation method where we use [Stochastic Gradient Descent](#).

```
[5]: keras.utils.set_random_seed(123)

ffnn.compile(optimizer=keras.optimizers.SGD(learning_rate=learning_rate),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

We use as the loss the `sparse_categorical_crossentropy` where

- *categorical* means that the neural network outputs are expected to be softmax values between 0 and 1
- *sparse* indicates that we have did not one-hot encode our y-targets; the loss will take care of that

```
[6]: print(y_train)
```

[5 0 4 ... 9 1 5]

We can now use the methods `fit` and `evaluate` to train our FFNN and evaluate it on the test set.

```
[7]: ffnn_history = ffnn.fit(x=x_train, y=y_train, batch_size=batch_size,
                           validation_data=(x_val, y_val), epochs=num_epochs)

test_loss, test_acc = ffnn.evaluate(x_test, y_test, verbose=0)

print(f'\nTest accuracy: {test_acc:.4f}')

ffnn.save('output/MNIST_FFNN.h5')
```

Epoch 1/10
 188/188 [=====] - 1s 4ms/step - loss: 1.5407 -
 accuracy: 0.6550 - val_loss: 0.9616 - val_accuracy: 0.8200
 Epoch 2/10
 188/188 [=====] - 0s 3ms/step - loss: 0.6911 -
 accuracy: 0.8517 - val_loss: 0.6014 - val_accuracy: 0.8680
 Epoch 3/10

```

188/188 [=====] - 0s 2ms/step - loss: 0.4792 -
accuracy: 0.8840 - val_loss: 0.4681 - val_accuracy: 0.8890
Epoch 4/10
188/188 [=====] - 1s 3ms/step - loss: 0.3894 -
accuracy: 0.8980 - val_loss: 0.4130 - val_accuracy: 0.8950
Epoch 5/10
188/188 [=====] - 1s 3ms/step - loss: 0.3391 -
accuracy: 0.9117 - val_loss: 0.3648 - val_accuracy: 0.9170
Epoch 6/10
188/188 [=====] - 1s 3ms/step - loss: 0.3028 -
accuracy: 0.9213 - val_loss: 0.3514 - val_accuracy: 0.9110
Epoch 7/10
188/188 [=====] - 1s 3ms/step - loss: 0.2741 -
accuracy: 0.9287 - val_loss: 0.3385 - val_accuracy: 0.9070
Epoch 8/10
188/188 [=====] - 1s 3ms/step - loss: 0.2505 -
accuracy: 0.9363 - val_loss: 0.3077 - val_accuracy: 0.9140
Epoch 9/10
188/188 [=====] - 0s 3ms/step - loss: 0.2318 -
accuracy: 0.9390 - val_loss: 0.3056 - val_accuracy: 0.9090
Epoch 10/10
188/188 [=====] - 0s 2ms/step - loss: 0.2139 -
accuracy: 0.9463 - val_loss: 0.2919 - val_accuracy: 0.9210

```

Test accuracy: 0.9070

We can summarise the network parameters:

[8]: `ffnn.summary()`

Model: "FFNN_example"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 1024)	803840
dense_1 (Dense)	(None, 512)	524800
dense_2 (Dense)	(None, 10)	5130
<hr/>		
Total params: 1,333,770		
Trainable params: 1,333,770		
Non-trainable params: 0		
<hr/>		

Let's see how the accuracy and loss developed over training making use of the `ffnn_history` object we assigned after training.

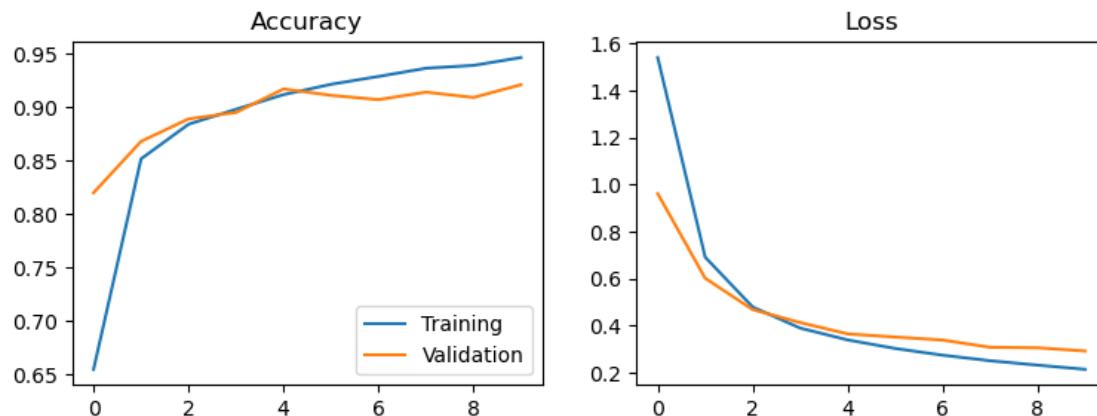
```
[9]: import matplotlib.pyplot as plt

fig, axs = plt.subplots(1,2, figsize=(9,3))

axs[0].plot(ffnn_history.history['accuracy'], label='Training')
axs[0].plot(ffnn_history.history['val_accuracy'], label='Validation')
axs[0].set_title('Accuracy')
axs[0].legend()

axs[1].plot(ffnn_history.history['loss'])
axs[1].plot(ffnn_history.history['val_loss'])
axs[1].set_title('Loss')

plt.show()
```



We use the `predict` method to provide predictions to new data. You can use the model you just trained or you can load a previously **pretrained** model.

```
[10]: # Our pretrained model (100 epochs):
ffnn = keras.models.load_model('output/MNIST_FFNN_pretrained.h5')

# ... or load the model you trained yourself:
# ffnn = keras.models.load_model('output/MNIST_FFNN.h5')

test_loss, test_acc = ffnn.evaluate(x_test, y_test, verbose=0)

print(f'Test accuracy: {test_acc:.4f}\n')

num_examples = 10

predictions_1hot = ffnn.predict(x_test, verbose=0)
```

```
# print(predictions_1hot[0])

nicer_predictions_1hot_0 = list(map(np.format_float_positional,
    ↪predictions_1hot[0], [5]*10))

print(f"Shape of the predictions: {predictions_1hot.shape}\n")
print(f"First prediction: {nicer_predictions_1hot_0}\n")

predictions = np.argmax(predictions_1hot, axis=1)

print(f"First class prediction: {predictions[0]}")
```

Test accuracy: 0.9260

Shape of the predictions: (1000, 10)

First prediction: ['0.00000', '0.00000', '0.00004', '0.00043', '0.00000',
 '0.00000', '0.00000', '0.99954', '0.00000', '0.00000']

First class prediction: 7

Note that the neural network outputs vectors with class probabilities. We choose the class with the highest probability.

Let's plot some of the predicted labels.

```
[11]: fig, axs = plt.subplots(1,num_examples, figsize=(num_examples,5))

start_index = 900

for i, ax in enumerate(axs.ravel()):
    ax.imshow(x_test[start_index+i].reshape(28,28))

    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title(f"Target: {y_test[start_index+i]}\nPred: {predictions[start_index+i]}",
        ↪fontsize=10)

plt.show()
```



1.2 Convolutional Neural Networks

An important class of neural network architectures are **convolutional neural networks (CNNs)**. CNNs are inspired by biological models for the primary visual cortex and their resemblance to the mathematical convolution operation. CNNs can be viewed as a particular kind of FFNNs where certain connections are missing and some neurons share the same weights. This gives rise to architectures as illustrated in the following figure.

CNNs are particularly suited to work with images and generally data **where some kind of local relationship / correlation** are present, e.g. adjacent pixels in an image or adjacent time steps in time series data. Two key components of CNNs are pooling and convolutional layers illustrated in the following.

Convolutional layers use a **convolutional filter** (sometimes called *kernel*, too) which processes the input images with respect to particular features (like edges) producing **feature maps**. Sequentially applying convolutional layers allow for **identifying increasingly complex features**. A typical size of a convolutional filter is 3×3 which can be thought of as a matrix of weights (in red above) which are adjusted during training to identify features in a data-driven fashion.

Pooling layers reduce the resolution of feature maps and don't have any associated weights. In particular, max pooling layers of size 2×2 are used to identify the main activation in adjacent pixels. This allows objects to be in different positions in the image but still be recognised as the same object.

Let's implement such a CNN and reshape the image vectors to actual images again.

```
[12]: mnist_data_image = mnist_data.reshape(-1, 28, 28, 1) / 255

x_train = mnist_data_image[:num_train]
x_val = mnist_data_image[num_train:num_train+num_val]
x_test = mnist_data_image[-num_test:]
```

We define a CNN with the following layers and train as before:

- 28 \times 28 \times 1 neurons in the input layer for the greyscale image
- 6 conv. filters of size 3×3 in the first hidden layer with ReLU activation followed by a 2×2 max pooling
- 16 conv. filters of size 3×3 in the second hidden layer with ReLU activation followed by a 2×2 max pooling
- flatten these feature maps in a single vector
- 1024 (hidden) neurons in the third hidden layer with ReLU activation
- 512 (hidden) neurons in the fourth hidden layer with ReLU activation
- 10 output neurons for our classes in the output layer with a softmax activation

```
[13]: keras.utils.set_random_seed(123)

cnn = keras.Sequential([
    keras.layers.InputLayer(input_shape=(28, 28, 1)),
    keras.layers.Conv2D(filters=6, kernel_size=(3, 3), activation='relu'),
```

```

keras.layers.MaxPooling2D(pool_size=(2, 2)),
keras.layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu'),
keras.layers.MaxPooling2D(pool_size=(2, 2)),
keras.layers.Flatten(),
keras.layers.Dense(units=1024, activation='relu'),
keras.layers.Dense(units=512, activation='relu'),
keras.layers.Dense(units=10, activation='softmax')
], name='CNN_example')

cnn.compile(optimizer=keras.optimizers.SGD(learning_rate=learning_rate),
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

cnn_history = cnn.fit(x=x_train, y=y_train, batch_size=batch_size,
                      validation_data=(x_val, y_val), epochs=num_epochs)

test_loss, test_acc = cnn.evaluate(x_test, y_test, verbose=0)

print(f'\nTest accuracy: {test_acc:.4f}\n')

cnn.save('output/MNIST_CNN.h5')

cnn.summary()

```

```

Epoch 1/10
188/188 [=====] - 1s 5ms/step - loss: 2.0536 -
accuracy: 0.4477 - val_loss: 1.3182 - val_accuracy: 0.6740
Epoch 2/10
188/188 [=====] - 1s 5ms/step - loss: 0.7038 -
accuracy: 0.8103 - val_loss: 0.6041 - val_accuracy: 0.8130
Epoch 3/10
188/188 [=====] - 1s 4ms/step - loss: 0.4115 -
accuracy: 0.8753 - val_loss: 0.3994 - val_accuracy: 0.8690
Epoch 4/10
188/188 [=====] - 1s 4ms/step - loss: 0.3023 -
accuracy: 0.9117 - val_loss: 0.3031 - val_accuracy: 0.8970
Epoch 5/10
188/188 [=====] - 1s 4ms/step - loss: 0.2517 -
accuracy: 0.9270 - val_loss: 0.2094 - val_accuracy: 0.9470
Epoch 6/10
188/188 [=====] - 1s 5ms/step - loss: 0.2132 -
accuracy: 0.9387 - val_loss: 0.2146 - val_accuracy: 0.9420
Epoch 7/10
188/188 [=====] - 1s 5ms/step - loss: 0.1818 -
accuracy: 0.9453 - val_loss: 0.2419 - val_accuracy: 0.9280
Epoch 8/10
188/188 [=====] - 1s 5ms/step - loss: 0.1586 -
accuracy: 0.9543 - val_loss: 0.1588 - val_accuracy: 0.9550

```

```

Epoch 9/10
188/188 [=====] - 1s 4ms/step - loss: 0.1421 -
accuracy: 0.9587 - val_loss: 0.1701 - val_accuracy: 0.9500
Epoch 10/10
188/188 [=====] - 1s 5ms/step - loss: 0.1206 -
accuracy: 0.9673 - val_loss: 0.1448 - val_accuracy: 0.9580

```

Test accuracy: 0.9360

Model: "CNN_example"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 6)	60
max_pooling2d (MaxPooling2D)	(None, 13, 13, 6)	0
conv2d_1 (Conv2D)	(None, 11, 11, 16)	880
max_pooling2d_1 (MaxPooling 2D)	(None, 5, 5, 16)	0
flatten (Flatten)	(None, 400)	0
dense_3 (Dense)	(None, 1024)	410624
dense_4 (Dense)	(None, 512)	524800
dense_5 (Dense)	(None, 10)	5130

Total params: 941,494
Trainable params: 941,494
Non-trainable params: 0

Let's study the performance of the CNN.

```
[14]: import matplotlib.pyplot as plt

fig, axs = plt.subplots(1,2, figsize=(9,3))

axs[0].plot(cnn_history.history['accuracy'], label='Training')
axs[0].plot(cnn_history.history['val_accuracy'], label='Validation')
axs[0].set_title('Accuracy')
axs[0].legend()
```

```

axs[1].plot(cnn_history.history['loss'])
axs[1].plot(cnn_history.history['val_loss'])
axs[1].set_title('Loss')

plt.show()

# Our pretrained model (100 epochs):
# cnn = keras.models.load_model('output/MNIST_CNN_pretrained.h5')

# ... or load your model:
cnn = keras.models.load_model('output/MNIST_CNN.h5')

test_loss, test_acc = cnn.evaluate(x_test, y_test, verbose=0)

print(f'Test accuracy: {test_acc:.4f}\n')

num_examples = 10

predictions_1hot = cnn.predict(x_test, verbose=0)
predictions = np.argmax(predictions_1hot, axis=1)

fig, axs = plt.subplots(1, num_examples, figsize=(num_examples,5))

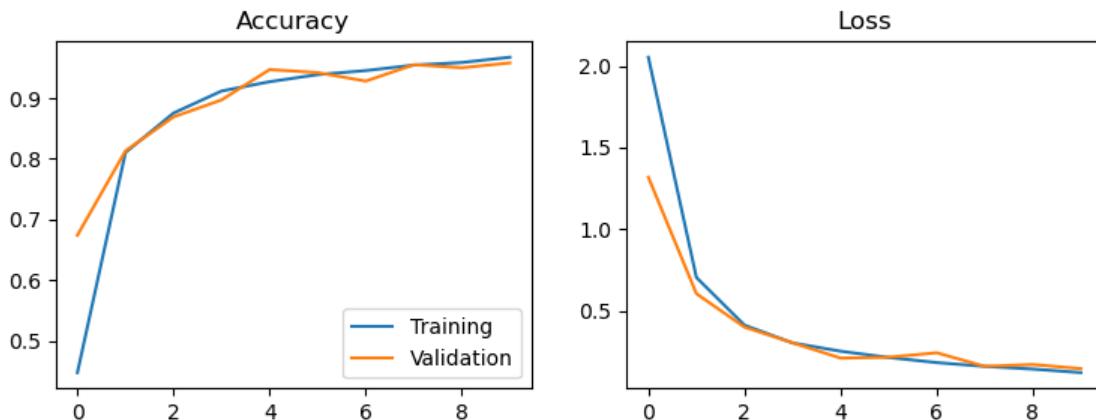
start_index = 900

for i, ax in enumerate(axs.ravel()):
    ax.imshow(x_test[start_index+i])

    ax.set_xticks([])
    ax.set_yticks([])
    ax.set_title(f"Target: {y_test[start_index+i]}\nPred:{predictions[i]}",
                fontsize=10)

plt.show()

```



Test accuracy: 0.9360



1.2.1 How to tune hyperparameters?

As mentioned in [Notebook 2](#), neural network performance is sensitive to the hyperparameter choice. Again, there are tools to perform hyperparameter search in a more automated manner like:

1. TensorFlow / Keras: [Keras Tuner](#)
 2. PyTorch: [Optuna](#)
-

1.3 Exercise Section

For a last time, we consider the red wine dataset with our new regression method. As before, we want to predict the quality values on the test set `feat_test` and compare it to the previous results:

- ridge regression MSE 0.42
- Random Forest regression MSE 0.34
- kernel ridge regression MSE 0.38
- Gaussian Process regression MSE 0.38

Prepare the data by load the following cell:

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

from sklearn.model_selection import train_test_split

from tensorflow import keras

np.random.seed(123)
keras.utils.set_random_seed(123)

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')

ex_target = wine_data['quality']
ex_features = wine_data.drop(['quality'], axis=1)

ex_features = (ex_features - ex_features.min()) / (ex_features.max() - ex_features.min())

feat_train, feat_test, target_train, target_test = train_test_split(
    ex_features, ex_target, test_size = 0.3, random_state=123)

target_test = target_test.to_frame()
```

(1.) Implement a feed-forward neural network with the following number of artificial neurons in the hidden layers: 1024, 512, 256, 128, 64. Use ReLU activations and the following hyperparameters:

```
[ ]: learning_rate = 0.005
batch_size = 64
num_epochs = 50
```

Put your result in the following cell and note:

- Provide the model as `ffnn_reg`, the history of the training in `ffnn_reg_history`, and the predicted quality values in `ex_pred_ffnn`.
- The output in the regression setting is just 1-dimensional and no activation is required.
- Use as the MSE loss `keras.losses.MeanSquaredError()` and do not use the `metrics` argument.
- You can use the test set for `validation_data`.

```
[ ]: # fill in
```

Try to check the accuracy and loss during training with the following plot.

```
[ ]: plt.plot(ffnn_reg_history.history['loss'][1:], label='Training')
plt.plot(ffnn_reg_history.history['val_loss'][1:], label='Test')
plt.title('Loss')
plt.legend()
```

```
plt.show()
```

As we have done earlier, the following cell provides a data frame with an overview of the predictions.

```
[27]: target_test['NN_predicted_quality'] = np.around(ex_pred_ffnn, decimals=2)
target_test['NN_absolut_deviation'] = np.around(abs(target_test['quality'] - □
    ↪target_test['NN_predicted_quality']), decimals=2)
print(target_test)

NN_pred_MSE = (target_test['NN_absolut_deviation']**2).mean()
print(f"\nMean squared error of Feedforward Neural Network regression: □
    ↪{NN_pred_MSE:.2f}")
```

	quality	NN_predicted_quality	NN_absolut_deviation
912	6	6.49	0.49
772	5	4.94	0.06
1037	5	4.74	0.26
1106	6	6.59	0.59
263	5	5.40	0.40
...
1466	7	5.59	1.41
580	5	4.69	0.31
1082	6	5.28	0.72
1279	7	6.29	0.71
1155	5	5.19	0.19

[480 rows x 3 columns]

Mean squared error of Feedforward Neural Network regression: 0.37

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

from sklearn.model_selection import train_test_split

from tensorflow import keras

np.random.seed(123)
keras.utils.set_random_seed(123)

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')

ex_target = wine_data['quality']
ex_features = wine_data.drop(['quality'], axis=1)

ex_features = (ex_features - ex_features.min()) / (ex_features.max() - □
    ↪ex_features.min())
```

```
feat_train, feat_test, target_train, target_test = train_test_split(
    ex_features, ex_target, test_size = 0.3, random_state=123)

target_test = target_test.to_frame()
```

(1.) Implement a feed-forward neural network with the following number of artificial neurons in the hidden layers: 1024, 512, 256, 128, 64. Use ReLU activations and the following hyperparameters:

[34]:

```
learning_rate = 0.005
batch_size = 64
num_epochs = 50
```

Put your result in the following cell and note:

- Provide the model as `ffnn_reg`, the history of the training in `ffnn_reg_history`, and the predicted quality values in `ex_pred_ffnn`.
- The output in the regression setting is just 1-dimensional and no activation is required.
- Use as the MSE loss `keras.losses.MeanSquaredError()` and do not use the `metrics` argument.
- You can use the test set for `validation_data`.

[35]:

```
ffnn_reg = keras.Sequential(
    [
        keras.layers.Dense(1024, activation='relu', input_shape=(11,)),
        keras.layers.Dense(512, activation='relu'),
        keras.layers.Dense(256, activation='relu'),
        keras.layers.Dense(128, activation='relu'),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(1)
    ],
    name='FFNN_regression')

ffnn_reg.compile(optimizer=keras.optimizers.SGD(learning_rate=learning_rate),
                  loss=keras.losses.MeanSquaredError())

ffnn_reg_history = ffnn_reg.fit(x=feat_train, y=target_train,
                                 validation_data=(feat_test, target_test),
                                 batch_size=batch_size, epochs=num_epochs)

ex_pred_ffnn = ffnn_reg.predict(feat_test, verbose=0)

ffnn_reg.save(f'output/MNIST_FFNN_reg.h5')
```

```
Epoch 1/50
18/18 [=====] - 0s 9ms/step - loss: 17.3084 - val_loss: 0.6507
Epoch 2/50
18/18 [=====] - 0s 5ms/step - loss: 0.6783 - val_loss:
```

```
0.6384
Epoch 3/50
18/18 [=====] - 0s 5ms/step - loss: 0.6341 - val_loss:
0.5898
Epoch 4/50
18/18 [=====] - 0s 5ms/step - loss: 0.6149 - val_loss:
0.5527
Epoch 5/50
18/18 [=====] - 0s 6ms/step - loss: 0.5946 - val_loss:
0.6049
Epoch 6/50
18/18 [=====] - 0s 5ms/step - loss: 0.5648 - val_loss:
0.5109
Epoch 7/50
18/18 [=====] - 0s 5ms/step - loss: 0.5551 - val_loss:
0.5547
Epoch 8/50
18/18 [=====] - 0s 5ms/step - loss: 0.5414 - val_loss:
0.4864
Epoch 9/50
18/18 [=====] - 0s 5ms/step - loss: 0.5218 - val_loss:
0.4878
Epoch 10/50
18/18 [=====] - 0s 5ms/step - loss: 0.4996 - val_loss:
0.4796
Epoch 11/50
18/18 [=====] - 0s 5ms/step - loss: 0.4966 - val_loss:
0.4754
Epoch 12/50
18/18 [=====] - 0s 6ms/step - loss: 0.4806 - val_loss:
0.4774
Epoch 13/50
18/18 [=====] - 0s 5ms/step - loss: 0.4737 - val_loss:
0.4592
Epoch 14/50
18/18 [=====] - 0s 5ms/step - loss: 0.4678 - val_loss:
0.4589
Epoch 15/50
18/18 [=====] - 0s 5ms/step - loss: 0.4623 - val_loss:
0.4516
Epoch 16/50
18/18 [=====] - 0s 5ms/step - loss: 0.4580 - val_loss:
0.4459
Epoch 17/50
18/18 [=====] - 0s 5ms/step - loss: 0.4597 - val_loss:
0.4725
Epoch 18/50
18/18 [=====] - 0s 5ms/step - loss: 0.4566 - val_loss:
```

```

0.4398
Epoch 19/50
18/18 [=====] - 0s 5ms/step - loss: 0.4521 - val_loss:
0.4499
Epoch 20/50
18/18 [=====] - 0s 5ms/step - loss: 0.4472 - val_loss:
0.4411
Epoch 21/50
18/18 [=====] - 0s 5ms/step - loss: 0.4493 - val_loss:
0.4379
Epoch 22/50
18/18 [=====] - 0s 5ms/step - loss: 0.4452 - val_loss:
0.4435
Epoch 23/50
18/18 [=====] - 0s 5ms/step - loss: 0.4521 - val_loss:
0.4311
Epoch 24/50
18/18 [=====] - 0s 5ms/step - loss: 0.4452 - val_loss:
0.4323
Epoch 25/50
18/18 [=====] - 0s 5ms/step - loss: 0.4374 - val_loss:
0.4297
Epoch 26/50
18/18 [=====] - 0s 5ms/step - loss: 0.4407 - val_loss:
0.4349
Epoch 27/50
18/18 [=====] - 0s 5ms/step - loss: 0.4442 - val_loss:
0.4442
Epoch 28/50
18/18 [=====] - 0s 5ms/step - loss: 0.4392 - val_loss:
0.4277
Epoch 29/50
18/18 [=====] - 0s 5ms/step - loss: 0.4327 - val_loss:
0.4261
Epoch 30/50
18/18 [=====] - 0s 6ms/step - loss: 0.4319 - val_loss:
0.4344
Epoch 31/50
18/18 [=====] - 0s 6ms/step - loss: 0.4332 - val_loss:
0.4253
Epoch 32/50
18/18 [=====] - 0s 6ms/step - loss: 0.4410 - val_loss:
0.4347
Epoch 33/50
18/18 [=====] - 0s 5ms/step - loss: 0.4311 - val_loss:
0.4235
Epoch 34/50
18/18 [=====] - 0s 5ms/step - loss: 0.4352 - val_loss:

```

```

0.4255
Epoch 35/50
18/18 [=====] - 0s 5ms/step - loss: 0.4308 - val_loss:
0.4209
Epoch 36/50
18/18 [=====] - 0s 5ms/step - loss: 0.4306 - val_loss:
0.4194
Epoch 37/50
18/18 [=====] - 0s 5ms/step - loss: 0.4286 - val_loss:
0.4356
Epoch 38/50
18/18 [=====] - 0s 5ms/step - loss: 0.4275 - val_loss:
0.4181
Epoch 39/50
18/18 [=====] - 0s 5ms/step - loss: 0.4250 - val_loss:
0.4380
Epoch 40/50
18/18 [=====] - 0s 5ms/step - loss: 0.4274 - val_loss:
0.4373
Epoch 41/50
18/18 [=====] - 0s 5ms/step - loss: 0.4255 - val_loss:
0.4215
Epoch 42/50
18/18 [=====] - 0s 5ms/step - loss: 0.4221 - val_loss:
0.4426
Epoch 43/50
18/18 [=====] - 0s 5ms/step - loss: 0.4222 - val_loss:
0.4576
Epoch 44/50
18/18 [=====] - 0s 5ms/step - loss: 0.4263 - val_loss:
0.4337
Epoch 45/50
18/18 [=====] - 0s 5ms/step - loss: 0.4211 - val_loss:
0.4273
Epoch 46/50
18/18 [=====] - 0s 5ms/step - loss: 0.4182 - val_loss:
0.4171
Epoch 47/50
18/18 [=====] - 0s 5ms/step - loss: 0.4186 - val_loss:
0.4359
Epoch 48/50
18/18 [=====] - 0s 5ms/step - loss: 0.4184 - val_loss:
0.4232
Epoch 49/50
18/18 [=====] - 0s 5ms/step - loss: 0.4229 - val_loss:
0.4288
Epoch 50/50
18/18 [=====] - 0s 5ms/step - loss: 0.4174 - val_loss:

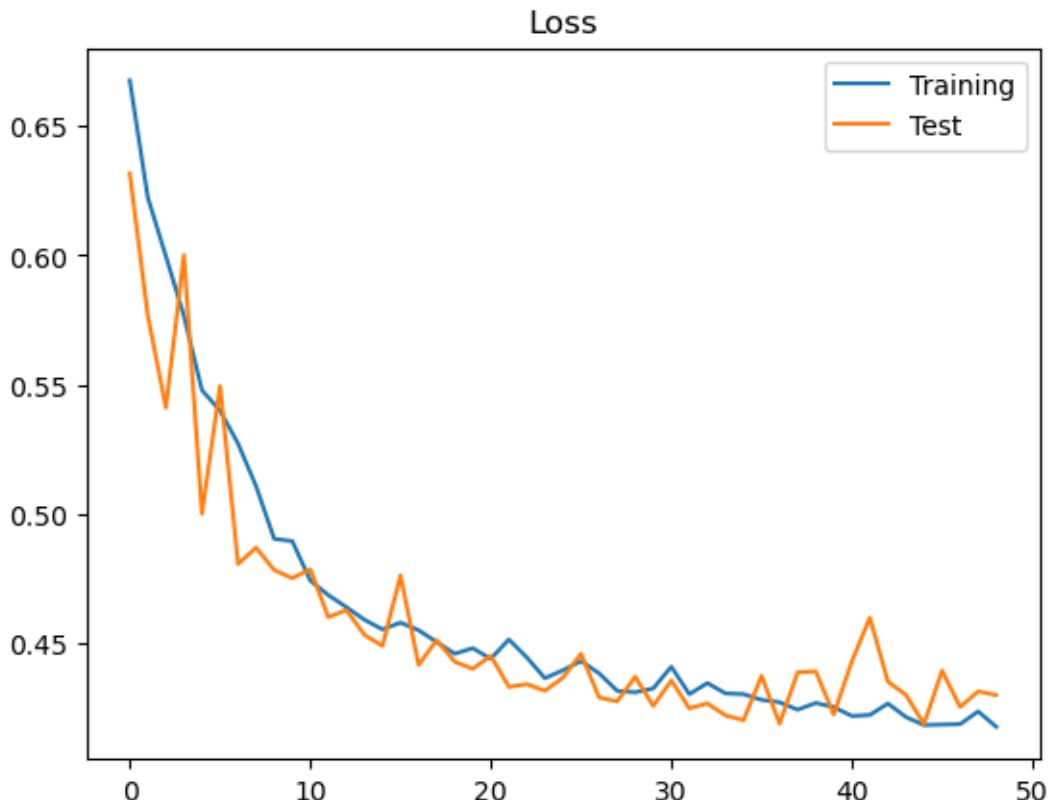
```

0.4273

Try to check the accuracy and loss during training with the following plot.

```
[32]: plt.plot(ffnn_reg_history.history['loss'][1:], label='Training')
plt.plot(ffnn_reg_history.history['val_loss'][1:], label='Test')
plt.title('Loss')
plt.legend()

plt.show()
```



As we have done earlier, the following cell provides a data frame with an overview of the predictions.

```
[ ]: # Pretrained for 1000 epochs
#ffnn_reg = keras.models.load_model('output/MNIST_FFNN_reg_pretrained.h5')
#ex_pred_ffnn = ffnn_reg.predict(feat_test, verbose=0)
```

```
[36]: target_test['NN_predicted_quality'] = np.around(ex_pred_ffnn, decimals=2)
target_test['NN_absolut_deviation'] = np.around(abs(target_test['quality'] - target_test['NN_predicted_quality']), decimals=2)
print(target_test)
```

```
NN_pred_MSE = (target_test['NN_absolut_deviation']**2).mean()
print(f"\nMean squared error of Feedforward Neural Network regression: {NN_pred_MSE:.2f}")
```

	quality	NN_predicted_quality	NN_absolut_deviation
912	6	6.52	0.52
772	5	5.09	0.09
1037	5	4.91	0.09
1106	6	6.60	0.60
263	5	5.51	0.51
...
1466	7	5.61	1.39
580	5	5.44	0.44
1082	6	5.47	0.53
1279	7	6.35	0.65
1155	5	5.34	0.34

[480 rows x 3 columns]

Mean squared error of Feedforward Neural Network regression: 0.43

1.4 Proposed Solutions

For a final time, we consider the red wine dataset with our new regression method. As before, we want to predict the `quality` values on the test set `feat_test` and compare it to the previous results:

- ridge regression MSE 0.42
- Random Forest regression MSE 0.34
- kernel ridge regression MSE 0.38
- Gaussian Process regression MSE 0.38

Prepare the data by load the following cell:

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

from sklearn.model_selection import train_test_split

from tensorflow import keras

np.random.seed(123)
keras.utils.set_random_seed(123)

wine_data = pd.read_csv('data/winequality-red.csv', sep=';')
```

```

ex_target = wine_data['quality']
ex_features = wine_data.drop(['quality'], axis=1)

ex_features = (ex_features - ex_features.min()) / (ex_features.max() - ex_features.min())

feat_train, feat_test, target_train, target_test = train_test_split(
    ex_features, ex_target, test_size = 0.3, random_state=123)

target_test = target_test.to_frame()

```

(1.) Implement a feed-forward neural network with the following number of artificial neurons in the hidden layers: 1024, 512, 256, 128, 64. Use ReLU activations and the following hyperparameters:

[16]:

```

learning_rate = 0.005
batch_size = 64
num_epochs = 50

```

Put your result in the following cell and note:

- Provide the model as `ffnn_reg`, the history of the training in `ffnn_reg_history`, and the predicted quality values in `ex_pred_ffnn`.
- The output in the regression setting is just 1-dimensional and no activation is required.
- Use as the MSE loss `keras.losses.MeanSquaredError()` and do not use the `metrics` argument.
- You can use the test set for `validation_data`.

[17]:

```

ffnn_reg = keras.Sequential(
    [
        keras.layers.Dense(1024, activation='relu', input_shape=(11,)),
        keras.layers.Dense(512, activation='relu'),
        keras.layers.Dense(256, activation='relu'),
        keras.layers.Dense(128, activation='relu'),
        keras.layers.Dense(64, activation='relu'),
        keras.layers.Dense(1)
    ],
    name='FFNN_regression')

ffnn_reg.compile(optimizer=keras.optimizers.SGD(learning_rate=learning_rate),
                  loss=keras.losses.MeanSquaredError())

ffnn_reg_history = ffnn_reg.fit(x=feat_train, y=target_train,
                                 validation_data=(feat_test, target_test),
                                 batch_size=batch_size, epochs=num_epochs)

ex_pred_ffnn = ffnn_reg.predict(feat_test, verbose=0)

ffnn_reg.save('output/MNIST_FFNN_reg.h5')

```

```

Epoch 1/50
18/18 [=====] - 0s 8ms/step - loss: 17.3084 - val_loss:
0.6507
Epoch 2/50
18/18 [=====] - 0s 5ms/step - loss: 0.6783 - val_loss:
0.6384
Epoch 3/50
18/18 [=====] - 0s 5ms/step - loss: 0.6341 - val_loss:
0.5898
Epoch 4/50
18/18 [=====] - 0s 5ms/step - loss: 0.6149 - val_loss:
0.5527
Epoch 5/50
18/18 [=====] - 0s 5ms/step - loss: 0.5946 - val_loss:
0.6049
Epoch 6/50
18/18 [=====] - 0s 4ms/step - loss: 0.5648 - val_loss:
0.5109
Epoch 7/50
18/18 [=====] - 0s 5ms/step - loss: 0.5551 - val_loss:
0.5547
Epoch 8/50
18/18 [=====] - 0s 5ms/step - loss: 0.5414 - val_loss:
0.4864
Epoch 9/50
18/18 [=====] - 0s 5ms/step - loss: 0.5218 - val_loss:
0.4878
Epoch 10/50
18/18 [=====] - 0s 5ms/step - loss: 0.4996 - val_loss:
0.4796
Epoch 11/50
18/18 [=====] - 0s 5ms/step - loss: 0.4966 - val_loss:
0.4754
Epoch 12/50
18/18 [=====] - 0s 5ms/step - loss: 0.4806 - val_loss:
0.4774
Epoch 13/50
18/18 [=====] - 0s 4ms/step - loss: 0.4737 - val_loss:
0.4592
Epoch 14/50
18/18 [=====] - 0s 4ms/step - loss: 0.4678 - val_loss:
0.4589
Epoch 15/50
18/18 [=====] - 0s 5ms/step - loss: 0.4623 - val_loss:
0.4516
Epoch 16/50
18/18 [=====] - 0s 5ms/step - loss: 0.4580 - val_loss:
0.4459

```

Epoch 17/50
 18/18 [=====] - 0s 5ms/step - loss: 0.4597 - val_loss:
 0.4725
 Epoch 18/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4566 - val_loss:
 0.4398
 Epoch 19/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4521 - val_loss:
 0.4499
 Epoch 20/50
 18/18 [=====] - 0s 5ms/step - loss: 0.4472 - val_loss:
 0.4411
 Epoch 21/50
 18/18 [=====] - 0s 5ms/step - loss: 0.4493 - val_loss:
 0.4379
 Epoch 22/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4452 - val_loss:
 0.4435
 Epoch 23/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4521 - val_loss:
 0.4311
 Epoch 24/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4452 - val_loss:
 0.4323
 Epoch 25/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4374 - val_loss:
 0.4297
 Epoch 26/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4407 - val_loss:
 0.4349
 Epoch 27/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4442 - val_loss:
 0.4442
 Epoch 28/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4392 - val_loss:
 0.4277
 Epoch 29/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4327 - val_loss:
 0.4261
 Epoch 30/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4319 - val_loss:
 0.4344
 Epoch 31/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4332 - val_loss:
 0.4253
 Epoch 32/50
 18/18 [=====] - 0s 4ms/step - loss: 0.4410 - val_loss:
 0.4347

```

Epoch 33/50
18/18 [=====] - 0s 4ms/step - loss: 0.4311 - val_loss:
0.4235
Epoch 34/50
18/18 [=====] - 0s 4ms/step - loss: 0.4352 - val_loss:
0.4255
Epoch 35/50
18/18 [=====] - 0s 4ms/step - loss: 0.4308 - val_loss:
0.4209
Epoch 36/50
18/18 [=====] - 0s 5ms/step - loss: 0.4306 - val_loss:
0.4194
Epoch 37/50
18/18 [=====] - 0s 5ms/step - loss: 0.4286 - val_loss:
0.4356
Epoch 38/50
18/18 [=====] - 0s 5ms/step - loss: 0.4275 - val_loss:
0.4181
Epoch 39/50
18/18 [=====] - 0s 4ms/step - loss: 0.4250 - val_loss:
0.4380
Epoch 40/50
18/18 [=====] - 0s 4ms/step - loss: 0.4274 - val_loss:
0.4373
Epoch 41/50
18/18 [=====] - 0s 5ms/step - loss: 0.4255 - val_loss:
0.4215
Epoch 42/50
18/18 [=====] - 0s 5ms/step - loss: 0.4221 - val_loss:
0.4426
Epoch 43/50
18/18 [=====] - 0s 5ms/step - loss: 0.4222 - val_loss:
0.4576
Epoch 44/50
18/18 [=====] - 0s 5ms/step - loss: 0.4263 - val_loss:
0.4337
Epoch 45/50
18/18 [=====] - 0s 4ms/step - loss: 0.4211 - val_loss:
0.4273
Epoch 46/50
18/18 [=====] - 0s 4ms/step - loss: 0.4182 - val_loss:
0.4171
Epoch 47/50
18/18 [=====] - 0s 5ms/step - loss: 0.4186 - val_loss:
0.4359
Epoch 48/50
18/18 [=====] - 0s 5ms/step - loss: 0.4184 - val_loss:
0.4232

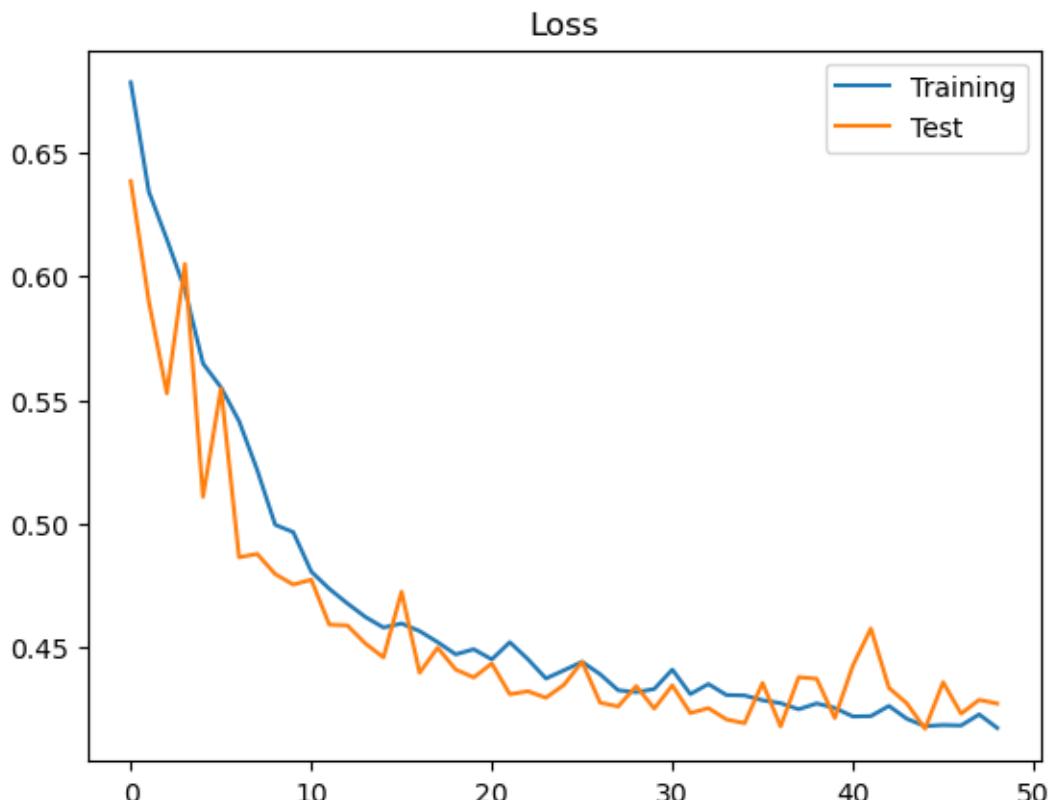
```

```
Epoch 49/50
18/18 [=====] - 0s 5ms/step - loss: 0.4229 - val_loss:
0.4288
Epoch 50/50
18/18 [=====] - 0s 4ms/step - loss: 0.4174 - val_loss:
0.4273
```

Try to check the accuracy and loss during training with the following plot.

```
[18]: plt.plot(ffnn_reg_history.history['loss'][1:], label='Training')
plt.plot(ffnn_reg_history.history['val_loss'][1:], label='Test')
plt.title('Loss')
plt.legend()

plt.show()
```



As we have done earlier, the following cell provides a data frame with an overview of the predictions.

```
[37]: # Pretrained for 1000 epochs
ffnn_reg = keras.models.load_model('output/MNIST_FFNN_reg_pretrained.h5')
ex_pred_ffnn = ffnn_reg.predict(feat_test, verbose=0)
```

```
[38]: target_test['NN_predicted_quality'] = np.around(ex_pred_ffnn, decimals=2)
target_test['NN_absolut_deviation'] = np.around(abs(target_test['quality'] - target_test['NN_predicted_quality']), decimals=2)
print(target_test)

NN_pred_MSE = (target_test['NN_absolut_deviation']**2).mean()
print(f"\nMean squared error of Feedforward Neural Network regression: {NN_pred_MSE:.2f}")
```

	quality	NN_predicted_quality	NN_absolut_deviation
912	6	6.49	0.49
772	5	4.94	0.06
1037	5	4.74	0.26
1106	6	6.59	0.59
263	5	5.40	0.40
...
1466	7	5.59	1.41
580	5	4.69	0.31
1082	6	5.28	0.72
1279	7	6.29	0.71
1155	5	5.19	0.19

[480 rows x 3 columns]

Mean squared error of Feedforward Neural Network regression: 0.37

[]:

6-VAE

May 7, 2024

1 6. Variational Autoencoder

We have seen approaches for compression previously. In this notebook, we cover a related approach based on neural networks which also enables us to generate novel (artificial) data.

In particular, we discuss

- latent variable models and
- the Variational Autoencoder

and provide a more advanced CNN implementation for MNIST and a molecule dataset.

Keywords: VAE, ELBO, reparameterisation trick, keras.Model, keras.layers.Conv2DTranspose, tensorflow.GradientTape, tf.sigmoid, model.save_weights, model.load_weights, keras.optimizers.Adam

1.1 Latent Variable Models

Latent variable models (LVMs) are statistical models which consider random variables. Suppose that our observations X and Y are random variables following a probability distribution $p(X, Y)$. LVMs assume that there are **latent, i.e. unobserved or hidden, variables** Z which have a relationship to our observed data X and Y . A typical assumption, for instance, is that our predictors X are generated by **lower-dimensional latent variables** Z , e.g. $p(X | Z)$.

Note that we have seen models such as these already. For example, in a **Gaussian Mixture Model** datapoints X_i belonging to cluster i with mean μ_i and covariance matrix Σ_i are generated by sampling from $\mathcal{N}(\mu_i, \Sigma_i)$. In other words, the low-dimensional latent variables are $Z = (\mu, \Sigma)$. **PCA** is another example where few principal components can allow describing the majority of variation in a dataset. However, these **linear LVMs** assume linear relationships between latent variable Z and the observations.

Due to the highly non-linear nature of neural networks, we can use neural networks to extend models such as PCA to **non-linear LVMs**. Here, we pick up the previously encountered idea of compression:

The **Variational Autoencoder (VAE)** is an unsupervised neural network approach which attempts to identify a meaningful **latent representation** Z capable of reconstructing the original input as perfectly as possible. The VAE can be viewed as a probabilistic non-linear PCA, with the general idea being illustrated in the following figure.

The VAE has three important components:

- 1. Encoder:** The encoder neural network attempts to compress the relevant information about X in latent representation Z with dimensions $d_X > d_Z$. The goal of the encoder is to parameterise a distribution $q_\phi(Z | X)$ which provides us a (variational) approximation for the distribution of the (compressed) latent variables Z given an input X . Here, ϕ denotes the weights of the encoder network. We typically assume that this approximation is given by a Gaussian distribution $\mathcal{N}(\mu_Z, \Sigma_Z)$ where $\mu_Z = (\mu_1, \dots, \mu_{d_Z})$ and $\Sigma_Z = (\sigma_1, \dots, \sigma_{d_Z})$ containing only the diagonal entries of covariance matrix Σ_Z (all other entries are 0). For technical reasons (details omitted), we require the **reparameterisation trick** to sample latent variables Z :

$$z_n(\varepsilon) = \mu_Z(x_n) + \Sigma_Z^2(x_n)\varepsilon$$

with $\varepsilon \sim \mathcal{N}(0, 1)$.

- 2. Latent representation / space Z :** Datapoints from the input space are mapped into the lower-dimensional latent space Z . To start with, we typically assume that the latent variables Z are normally distributed, i.e. a prior distribution $p(Z) = \mathcal{N}(0, \mathbb{1}_{d_Z})$ with d_Z being the number of latent dimensions. We train the encoder such that the (posterior) distribution $q_\phi(Z | X)$ matches the prior $p(Z)$ as much as possible. The deviations from the prior distribution will capture important features of our dataset. We measure the difference in these distributions with the (non-negative) [Kullback-Leibler divergence](#)

$$D_{\text{KL}}(q_\phi(Z | X) \| p(Z)) = -\frac{1}{2} \sum_{i=1}^{d_Z} (1 + \log \sigma_i^2 - \mu_i^2 - \sigma_i^2),$$

where the last expression follows from the Gaussian distribution assumptions outlined before.

- 3. Decoder:** The decoder neural network attempts to reconstruct the original input X from the compressed representation Z . The goal of the decoder is to parameterise the likelihood $p_\theta(X | Z)$ with decoder network weights θ . The **reconstruction error** is measured by the log-likelihood $\log(p_\theta(X | Z))$ which in the case of regression tasks is typically given by the mean-squared error loss and in classification tasks by the cross-entropy loss.

Combining these ingredients, we obtain the VAE loss function which is usually referred to as negative **evidence lower bound (ELBO)**

$$\text{Loss}(\phi, \theta) = \frac{1}{s} \sum_{i=1}^s \underbrace{\mathbb{E}_{q_\phi(z|x_i)}(-\log(p_\theta(x_i|z)))}_{\text{reconstruction loss}} + \underbrace{D_{\text{KL}}(q_\phi(z|x_i) \| p(z))}_{\text{latent loss}} \quad (1)$$

where s is the batch size (denoting that we sum the results for different batches). Note that the VAE objective has two parts which we try to balance:

1. The **reconstruction loss** measures how well we can reconstruct the input X from the compression Z . In other words, this term is an incentive to provide as much information of X in Z as possible, i.e. less compression.
2. The **latent loss** measures how well the posterior $q_\phi(Z | X)$ agrees with the prior $p(Z)$. This term favours stronger compressions.

With this probabilistic framework, we can now explore the latent space Z and, in particular, pick latent points in Z to generate novel artificial objects by decoding the picked latent point!

Let's implement and apply a VAE on MNIST! For this, we first define a class for the model which provides the individual parts as outlined above.

```
[1]: import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras

class VAE(keras.Model):

    def __init__(self, latent_dim, batch_size):
        super(VAE, self).__init__()
        self.latent_dim = latent_dim
        self.batch_size = batch_size

        # Encoder network:
        self.encoder = keras.Sequential(
            [
                keras.layers.InputLayer(input_shape=(28, 28, 1)),
                keras.layers.Conv2D(filters=32, kernel_size=(3,3), strides=(2, 2),
                    activation='relu'),
                keras.layers.Conv2D(filters=64, kernel_size=3, strides=(2, 2),
                    activation='relu'),
                keras.layers.Flatten(),
                keras.layers.Dense(latent_dim + latent_dim),
            ]
        )

        # Decoder network:
        self.decoder = keras.Sequential(
            [
                keras.layers.InputLayer(input_shape=(latent_dim,)),
                keras.layers.Dense(units=7 * 7 * 32, activation='relu'),
                keras.layers.Reshape(target_shape=(7, 7, 32)),
                keras.layers.Conv2DTranspose(filters=32, kernel_size=3,
                    strides=2, padding='same', activation='relu'),
                keras.layers.Conv2DTranspose(filters=64, kernel_size=3,
                    strides=2, padding='same', activation='relu'),
                keras.layers.Conv2DTranspose(filters=1, kernel_size=3,
                    strides=1, padding='same'),
            ]
        )

    def encode(self, x):
        mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
        return mean, logvar
```

```

def decode(self, z, apply_sigmoid=False):
    logits = self.decoder(z)
    if apply_sigmoid:
        probs = tf.sigmoid(logits)
        return probs
    return logits

@tf.function
def sample(self, eps=None):
    if eps is None:
        eps = tf.random.normal(shape=(self.batch_size, self.latent_dim))
    return self.decode(eps, apply_sigmoid=True)

# Reparameterisation trick:
def reparameterize(self, mean, logvar):
    eps = tf.random.normal(shape=(self.batch_size, self.latent_dim))
    return eps * tf.exp(logvar * .5) + mean

def call(self, x):
    mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
    z = self.reparameterize(mean, logvar)
    return self.decoder(z), mean, logvar

```

2024-05-07 09:59:07.630466: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations. To enable the following instructions: SSE4.1 SSE4.2, in other operations, rebuild TensorFlow with the appropriate compiler flags.

Note the effect of `padding` in the figure below (left: `padding='valid'`; right: `padding='same'`):

Image source

Furthermore, we define some functions we will use for computing the loss and the training steps. Note that in the previous notebook we used some built-in `keras` functions for that.

```
[2]: def log_normal_pdf(sample, mean, logvar, raxis=1):
    log2pi = tf.math.log(2.0 * np.pi)
    return tf.reduce_sum(-0.5 * ((sample - mean) ** 2.0 * tf.exp(-logvar) + logvar + log2pi), axis=raxis)

def compute_loss(model, x):
```

```

mean, logvar = model.encode(x)
z = model.reparameterize(mean, logvar)
x_logit = model.decode(z)
cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, □
↪labels=x)
logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
logpz = log_normal_pdf(z, 0., 0.)
logqz_x = log_normal_pdf(z, mean, logvar)

return -tf.reduce_mean(logpx_z + logpz - logqz_x)

@tf.function
def train_step(model, x, optimizer):

    with tf.GradientTape() as tape:
        loss = compute_loss(model, x)

    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

```

Now, we can load MNIST as before.

```
[3]: mnist_data = np.load('data/mnist_data_5k.npy', allow_pickle=True).
↪astype('float32')
mnist_targets = np.load('data/mnist_labels_5k.npy', allow_pickle=True).
↪astype('float32')

mnist_data = mnist_data.reshape(-1, 28, 28, 1) / 255

num_train = 3000
num_val = 1000
num_test = 1000

x_train = mnist_data[:num_train]
y_train = mnist_targets[:num_train]

x_val = mnist_data[num_train:num_train+num_val]
y_val = mnist_targets[num_train:num_train+num_val]

x_test = mnist_data[-num_test:]
y_test = mnist_targets[-num_test:]
```

Define the hyperparameters, the model, and the optimiser, where we choose the Adam optimiser this time.

```
[4]: learning_rate = 0.0001
batch_size = 64
```

```

num_epochs = 20
latent_dim = 2

vae_model = VAE(latent_dim, batch_size)
optimizer = keras.optimizers.Adam(learning_rate=learning_rate)

```

In the following, we write our own loop for training and validation.

For Noto users: Please note that the training can take several minutes on Noto!

```
[5]: for epoch in range(1, num_epochs + 1):

    # Training steps
    for i in range(num_train // batch_size):
        train_step(vae_model, x_train[i:i + batch_size], optimizer)

    # Validation steps
    r_val = num_val // batch_size
    loss = keras.metrics.Mean()
    elbo = 0

    for i in range(r_val):
        loss(compute_loss(vae_model, x_val[i:i + batch_size]))
        elbo += -loss.result()

    elbo /= r_val

    print(f'Epoch: {epoch}/{num_epochs}, validation loss (ELBO): {elbo}')
```

```

Epoch: 1/20, validation loss (ELBO): -521.6192016601562
Epoch: 2/20, validation loss (ELBO): -394.4845275878906
Epoch: 3/20, validation loss (ELBO): -307.2398376464844
Epoch: 4/20, validation loss (ELBO): -267.0062561035156
Epoch: 5/20, validation loss (ELBO): -243.44163513183594
Epoch: 6/20, validation loss (ELBO): -231.35269165039062
Epoch: 7/20, validation loss (ELBO): -224.27911376953125
Epoch: 8/20, validation loss (ELBO): -219.93687438964844
Epoch: 9/20, validation loss (ELBO): -217.21531677246094
Epoch: 10/20, validation loss (ELBO): -215.21432495117188
Epoch: 11/20, validation loss (ELBO): -213.58523559570312
Epoch: 12/20, validation loss (ELBO): -212.22254943847656
Epoch: 13/20, validation loss (ELBO): -210.66500854492188
Epoch: 14/20, validation loss (ELBO): -209.10780334472656
Epoch: 15/20, validation loss (ELBO): -207.26524353027344
Epoch: 16/20, validation loss (ELBO): -204.93399047851562
Epoch: 17/20, validation loss (ELBO): -202.47634887695312
Epoch: 18/20, validation loss (ELBO): -200.40728759765625
Epoch: 19/20, validation loss (ELBO): -198.74905395507812
Epoch: 20/20, validation loss (ELBO): -198.24456787109375

```

For our custom model, we need to store the trained weights in the following way:

```
[ ]: vae_model.save_weights('output/MNIST_VAE.h5', save_format='h5')
```

You can load a pretrained model like this:

```
[6]: vae_model = VAE(latent_dim, num_test)
vae_model(x_test) # For input dimension specification

# Our pretrained model (300 epochs):
vae_model.load_weights('output/MNIST_VAE_pretrained.h5')

# ... or load your model:
# vae_model.load_weights('output/MNIST_VAE.h5')
```

Let's check out the results for the test data.

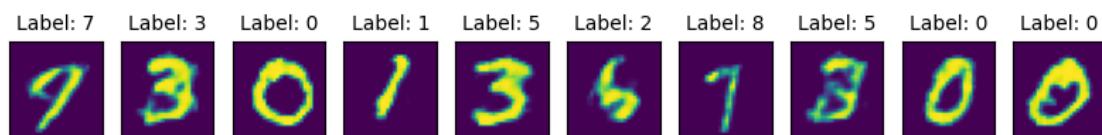
```
[7]: mean_z, logvar_z = vae_model.encode(x_test)
latent_z = vae_model.reparameterize(mean_z, logvar_z)
predictions = vae_model.sample(latent_z)

fig, axs = plt.subplots(1,10, figsize=(10,5))

for i, ax in enumerate(axs.ravel()):
    ax.imshow(predictions[i])

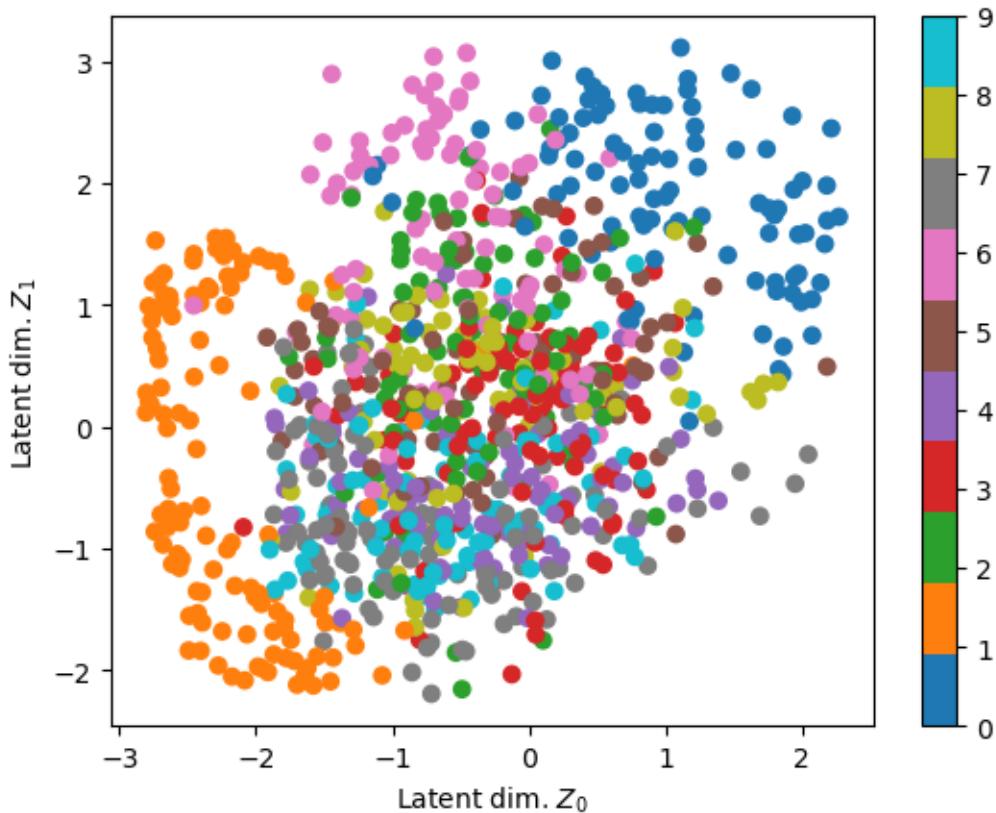
    ax.set_title(f"Label: {int(y_test[i])}", fontsize=10)
    ax.set_xticks([])
    ax.set_yticks([])

plt.show()
```



We can also investigate where the test datapoints lie in the latent space.

```
[8]: plt.scatter(latent_z[:, 0], latent_z[:, 1], c=y_test, cmap='tab10')
plt.colorbar()
plt.xlabel(r"Latent dim. $Z_0$")
plt.ylabel(r"Latent dim. $Z_1$")
plt.show()
```



1.2 Variational Autoencoder for Molecules

In the first part, we considered unsupervised learning. In the following we apply the VAE-idea on a molecular dataset and train a **Chemical VAE**. In addition to the standard, unsupervised VAE, we also predict a chemical property Y . This supervised setting allows us to do a **guided search of novel molecules** which match a property of our interest.

As illustrated above, we predict a chemical property with an **additional decoder network** directly from the latent sample. This does not only provide additional information about the molecules but also structures the latent representation Z with respect to property Y . This will have the effect the chemical property will gradually change within the latent space.

So far, we have worked with tabular data and images as inputs. Molecules can be represented in various ways. A common representation is a graph. Here, the nodes of a molecular graph are the atoms and the edges are molecular bonds. A popular and simple depiction of a molecular graph is a so-called **SMILES representation**. It is a text string that encloses all the information required to convert it to a graph.

For our experiments, we will use a variant called the **Deep-SMILES representation**, which makes generation of valid molecular structures easier.

Let's have a closer look on the dataset and how these SMILES and Deep-SMILES look like.

```
[9]: import pandas as pd
import numpy as np

molecule_data = pd.read_pickle('data/molecule_data.pkl')
molecule_data
```

	id	smiles	deep_smiles	band_gap
0	2	N	N	0.3399
1	3	O	O	0.3615
2	4	C#C	C#C	0.3351
3	7	CC	CC	0.4426
4	8	CO	CO	0.3437
...
31667	41529	C1CC2=CCCOCCC12	CCC=CCC0CC97	0.2548
31668	41530	C1CC2=CCOCCC12	CCC=CCOCCC97	0.2528
31669	41531	C1CC2=CCOCOC12	CCC=CCOCOC97	0.2504
31670	41533	C1OC2=NCCCCC12	COC=NCCCCC97	0.2701
31671	41535	C1OC2=NCCCOOC12	COC=NCCCOOC97	0.2648

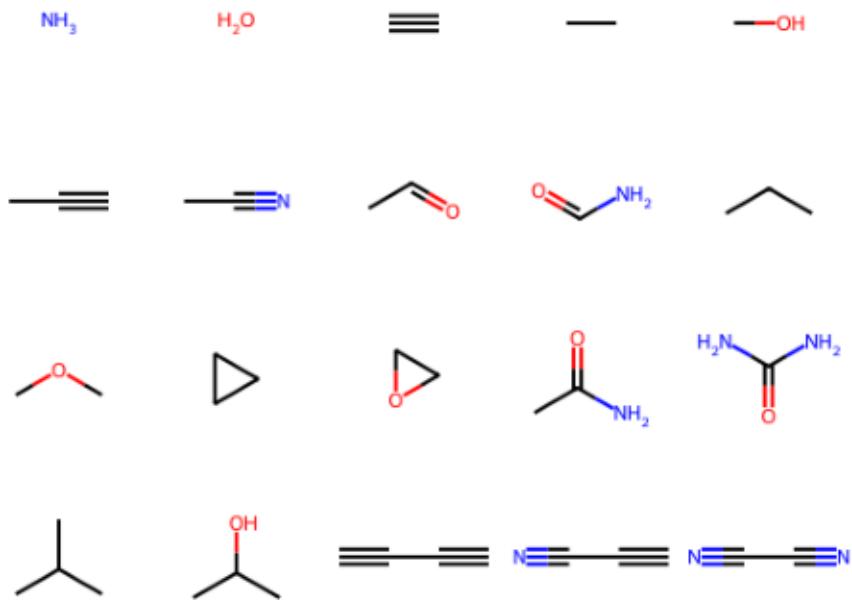
[31672 rows x 4 columns]

We can now plot the first and the last molecules from the dataset as chemical graphs.

```
[10]: from rdkit import Chem
from rdkit.Chem import Draw
import matplotlib.pyplot as plt

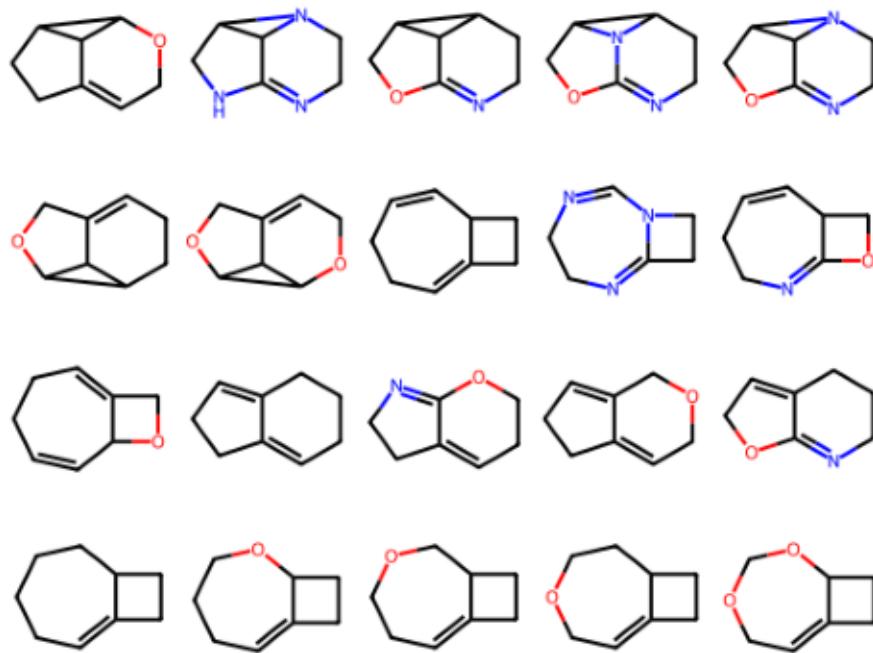
smiles_list = molecule_data['smiles'].values[0:20]
molecule_list = [Chem.MolFromSmiles(smiles) for smiles in smiles_list]
figure = Draw.MolsToGridImage(molecule_list, molsPerRow=5, subImgSize=(100, 100), returnPNG=False)

plt.imshow(figure)
plt.axis('off')
plt.show()
```



```
[11]: smiles_list = molecule_data['smiles'].values[31650:31670]
molecule_list = [Chem.MolFromSmiles(smiles) for smiles in smiles_list]
figure = Draw.MolsToGridImage(molecule_list, molsPerRow=5, subImgSize=(100, 100), returnPNG=False)

plt.imshow(figure)
plt.axis('off')
plt.show()
```



Let's define some functions which help us to convert deep-smiles into a one-hot representation and normalise the property values.

```
[12]: def smiles_to_onehot(deep_smiles):

    max_len = max([len(x) for x in deep_smiles])
    data = []

    for item in deep_smiles:
        ch_lst = list(item) + (max_len - len(item)) * [stop_character]

        res = []

        for ch in ch_lst:
            r = [0] * len(chars)
            r[chars.index(ch)] = 1
            res.append(r)

        data.append(res)

    return data
```

```

def one_hot_to_smiles(data, chars, stop_character):

    res = []

    for item in data:
        ch_lst = []

        for ch in item:
            ch_lst.append(chars[np.argmax(ch)])

        res.append(''.join(ch_lst).split(stop_character)[0])

    return res


def sparse_to_smiles(data):

    res = []

    for item in data:
        ch_lst = []

        for ch in item:
            ch_lst.append(chars[ch[0]])

        res.append(''.join(ch_lst))

    return res

```

First, we shuffle the data.

```
[13]: molecule_data = molecule_data.sample(frac=1, random_state=123)
```

Then, we use our function to convert deep smiles to a one-hot representation

```

[14]: stop_character = 'x'

chars = sorted(list(set(''.join(molecule_data['deep_smiles'].values))))
chars = chars + [stop_character]

print('Unique characters:', chars)

data = smiles_to_onehot(molecule_data['deep_smiles'].values)
data = np.array(data).reshape((len(data), 17, 15, 1)).astype(np.float32)

```

Unique characters: ['#', ')', '3', '4', '5', '6', '7', '8', '9', '=', 'C', 'F', 'N', 'O', 'x']

and normalise the property values

```
[15]: band_gap_mean = molecule_data['band_gap'].mean()
band_gap_std = molecule_data['band_gap'].std()
target = (molecule_data['band_gap'] - band_gap_mean) / band_gap_std
target = target.values.reshape(len(data), 1).astype(np.float32)
```

As usual, we split the dataset into training, validation, and test sets.

```
[16]: num_train = 28000
num_val = 1000
num_test = 1000

x_train = data[:num_train]
y_train = target[:num_train]

x_val = data[num_train:num_train + num_val]
y_val = target[num_train:num_train + num_val]

x_test = data[num_train + num_val:num_train + num_val + num_test]
y_test = target[num_train + num_val:num_train + num_val + num_test]
```

We can now create our new model CVAE which provides property predictions.

```
[17]: import tensorflow as tf
from tensorflow import keras

class CVAE(keras.Model):

    def __init__(self, latent_dim, batch_size):
        super(CVAE, self).__init__()
        self.latent_dim = latent_dim
        self.batch_size = batch_size

        self.encoder = keras.Sequential(
            [
                keras.layers.InputLayer(input_shape=(17, 15, 1)),
                keras.layers.Flatten(),
                keras.layers.Dense(units=1120, activation='relu'),
                keras.layers.Dense(units=1120, activation='relu'),
                keras.layers.Dense(latent_dim + latent_dim),
            ]
        )

        self.decoder = keras.Sequential(
            [
                keras.layers.InputLayer(input_shape=(latent_dim,)),
                keras.layers.Dense(units=1120, activation='relu'),
                keras.layers.Dense(units=1120, activation='relu'),
                keras.layers.Dense(units=255),
```

```

        keras.layers.Reshape(target_shape=(17, 15, 1)),
    ]
)

# Our new decoder:
self.decoder_property = keras.Sequential(
[
    keras.layers.InputLayer(input_shape=(latent_dim,)),
    keras.layers.Dense(units=1120, activation='relu'),
    keras.layers.Dense(units=1120, activation='relu'),
    keras.layers.Dense(units=1),
]
)

def call(self, x):
    mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
    z = self.reparameterize(mean, logvar)
    return self.decoder(z), mean, logvar

@tf.function
def sample(self, eps=None):
    if eps is None:
        eps = tf.random.normal(shape=(self.batch_size, self.latent_dim))
    return self.decode(eps, apply_sigmoid=True)

def encode(self, x):
    mean, logvar = tf.split(self.encoder(x), num_or_size_splits=2, axis=1)
    return mean, logvar

def reparameterize(self, mean, logvar):
    eps = tf.random.normal(shape=(self.batch_size, self.latent_dim))
    return eps * tf.exp(logvar * .5) + mean

def decode(self, z, apply_sigmoid=False):
    logits = self.decoder(z)
    if apply_sigmoid:
        probs = tf.sigmoid(logits)
        return probs
    return logits

def log_normal_pdf(sample, mean, logvar, raxis=1):
    log2pi = tf.math.log(2.0 * np.pi)

    return tf.reduce_sum(-0.5 * ((sample - mean) ** 2.0 * tf.exp(-logvar) + logvar + log2pi), axis=raxis)

```

```

def compute_loss(model, x, y):
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    x_logit = model.decode(z)
    cross_ent = tf.nn.sigmoid_cross_entropy_with_logits(logits=x_logit, labels=x)
    logpx_z = -tf.reduce_sum(cross_ent, axis=[1, 2, 3])
    logpz = log_normal_pdf(z, 0., 0.)
    logqz_x = log_normal_pdf(z, mean, logvar)

    # Property
    y_pred = model.decoder_property(z)
    y_mse = -tf.reduce_mean(tf.square(y - y_pred), axis = 0)

    return -tf.reduce_mean(100 * y_mse + logpx_z + logpz - logqz_x)

def compute_y_mae(model, x, y):
    mean, logvar = model.encode(x)
    z = model.reparameterize(mean, logvar)
    y_pred = model.decoder_property(z)
    y_mae = tf.reduce_mean(tf.abs(y - y_pred), axis = 0)
    return y_mae

@tf.function
def train_step(model, x, y, optimizer):
    with tf.GradientTape() as tape:
        loss = compute_loss(model, x, y)
    gradients = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(gradients, model.trainable_variables))

```

With this, we can now set the hyperparameters and create our model and optimiser.

```
[18]: learning_rate = 0.0001
batch_size = 100
num_epochs = 31
latent_dim = 14

optimizer = keras.optimizers.Adam(learning_rate)
model = CVAE(latent_dim, batch_size)
```

You can try to train this model:

```
[19]: for epoch in range(1, num_epochs + 1):

    # Training steps
```

```

for i in range(num_train // batch_size):
    train_step(model, x_train[i:i + batch_size], y_train[i:i + batch_size], optimizer)

# Validation steps
r_val = num_val // batch_size
loss = keras.metrics.Mean()
elbo = 0
y_loss = 0

for i in range(r_val):
    loss(compute_loss(model, x_val[i:i + batch_size], y_val[i:i + batch_size]))
    y_loss += compute_y_mae(model, x_val[i:i + batch_size], y_val[i:i + batch_size]).numpy()[0]
    elbo += -loss.result()

elbo /= r_val
y_loss = y_loss / r_val * band_gap_std * 627

# Print validation results
print(f'Epoch: {epoch}/{num_epochs}, validation loss (ELBO): {elbo}, Y MAE: {y_loss}')

if epoch % 10:
    path = 'output/Molecules_VAE.h5'
    model.save_weights(path, save_format='h5')

```

Epoch: 1/31, validation loss (ELBO): -70.43801879882812, Y MAE:

13.406085800693342

Epoch: 2/31, validation loss (ELBO): -63.81208419799805, Y MAE:

12.534704037632286

KeyboardInterrupt Cell In[19], line 5 <pre> 1 for epoch in range(1, num_epochs + 1): 2 3 # Training steps 4 for i in range(num_train // batch_size): 5 train_step(model, x_train[i:i + batch_size], y_train[i:i + batch_size], optimizer) 6 7 # Validation steps 8 r_val = num_val // batch_size </pre>	Traceback (most recent call last)
--	-----------------------------------

```

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/util/
    traceback_utils.py:150, in filter_traceback.<locals>.error_handler(*args, **kwargs)

```

```

148 filtered_tb = None
149 try:
--> 150     return fn(*args, **kwargs)
151 except Exception as e:
152     filtered_tb = _process_traceback_frames(e.__traceback__)

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳ polymorphic_function/polymorphic_function.py:894, in Function.__call__(self,*
→ *args, **kwds)
    891 compiler = "xla" if self._jit_compile else "nonXla"
    893 with OptionalXlaContext(self._jit_compile):
--> 894     result = self._call(*args, **kwds)
    896 new_tracing_count = self.experimental_get_tracing_count()
    897 without_tracing = (tracing_count == new_tracing_count)

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳ polymorphic_function/polymorphic_function.py:926, in Function._call(self,*
→ *args, **kwds)
    923     self._lock.release()
    924     # In this case we have created variables on the first call, so we run
→ the
    925     # defunned version which is guaranteed to never create variables.
--> 926     return self._no_variable_creation_fn(*args, **kwds) # pylint:u
→ disable=not-callable
    927 elif self._variable_creation_fn is not None:
    928     # Release the lock early so that multiple threads can perform the call
    929     # in parallel.
    930     self._lock.release()

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳ polymorphic_function/tracing_compiler.py:143, in TracingCompiler.
→ __call__(self, *args, **kwargs)
    140 with self._lock:
    141     (concrete_function,
    142      filtered_flat_args) = self._maybe_define_function(args, kwargs)
--> 143 return concrete_function._call_flat(
    144     filtered_flat_args, captured_inputs=concrete_function.
→ captured_inputs)

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳ polymorphic_function/monomorphic_function.py:1757, in ConcreteFunction.
→ __call_flat(self, args, captured_inputs, cancellation_manager)
    1753 possible_gradient_type = gradients_util.PossibleTapeGradientTypes(args)
    1754 if (possible_gradient_type == gradients_util.POSSIBLE_GRADIENT_TYPES_NONE
    1755     and executing_eagerly):
    1756     # No tape is watching; skip to running the function.
-> 1757     return self._build_call_outputs(self._inference_function.call(
    1758         ctx, args, cancellation_manager=cancellation_manager))
    1759 forward_backward = self._select_forward_and_backward_functions(

```

```

1760     args,
1761     possible_gradient_type,
1762     executing_eagerly)
1763 forward_function, args_with_tangents = forward_backward.forward()

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳polymorphic_function/monomorphic_function.py:381, in _EagerDefinedFunction.
↳call(self, ctx, args, cancellation_manager)
 379 with _InterpolateFunctionError(self):
 380     if cancellation_manager is None:
--> 381         outputs = execute.execute(
 382             str(self.signature.name),
 383             num_outputs=self._num_outputs,
 384             inputs=args,
 385             attrs=attrs,
 386             ctx=ctx)
 387     else:
 388         outputs = execute.execute_with_cancellation(
 389             str(self.signature.name),
 390             num_outputs=self._num_outputs,
(...)
```

`(...)`

```

 393             ctx=ctx,
 394             cancellation_manager=cancellation_manager)

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳execute.py:52, in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
 50 try:
 51     ctx.ensure_initialized()
--> 52     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name, op_name
 53                                     inputs, attrs, num_outputs)
 54 except core._NotOkStatusException as e:
 55     if name is not None:
```

`KeyboardInterrupt`:

Alternatively, you can just load the pretrained model here:

```
[20]: model = CVAE(latent_dim, num_test)
model(x_test)

# Our pretrained model (31 epochs):
model.load_weights('output/Molecules_VAE_pretrained.h5')

# ... or load your model:
# model.load_weights('output/Molecules_VAE.h5')
```

We can now make predictions with our model and use PCA for dimensionality reduction in order to visualise the latent space.

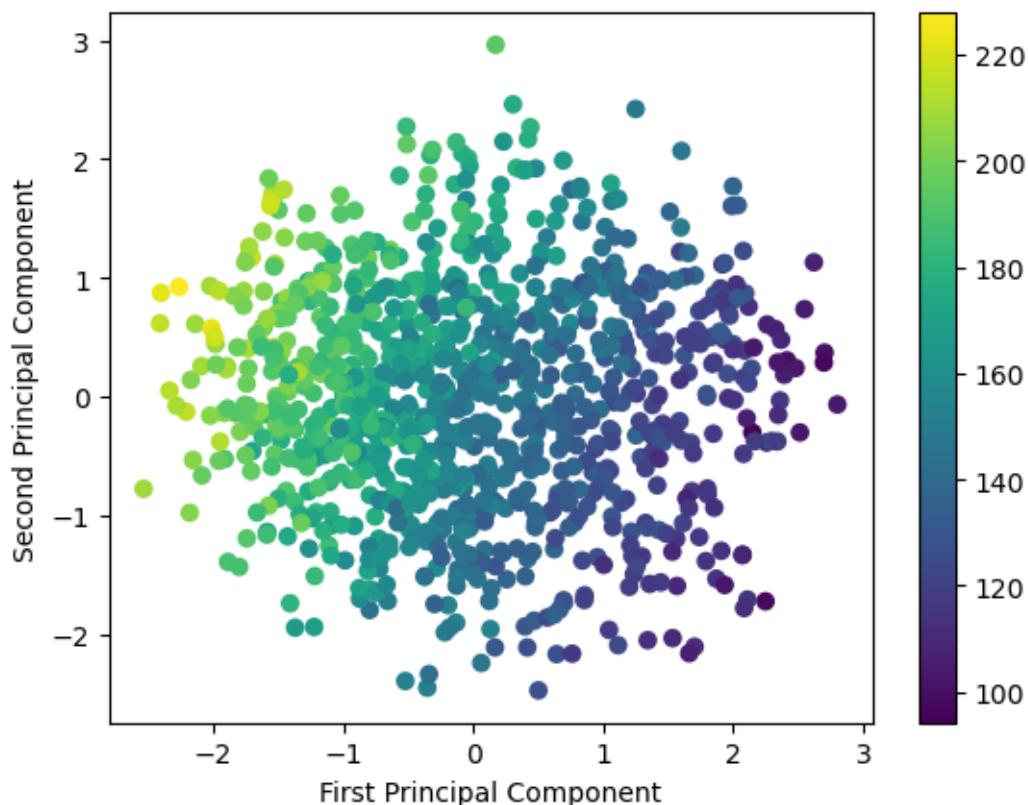
```
[21]: from sklearn.decomposition import PCA

mean_z, logvar_z = model.encode(x_test)
latent_z = model.reparameterize(mean_z, logvar_z)
target_pred = model.decoder_property(latent_z).numpy().flatten()

# Invert the scaling and convert the property unit to Kcal/mol
target_pred = (target_pred * band_gap_std + band_gap_mean) * 627

data_pca = PCA(n_components=2)
data_pca.fit(mean_z.numpy())
data_pca_transform = data_pca.transform(mean_z.numpy())

plt.scatter(data_pca_transform[:, 0], data_pca_transform[:, 1], c=target_pred)
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.colorbar()
plt.show()
```



In this plot we can see a **smooth transition of the property correspondence** making it easy to sample molecules of our interest. To this end, we can linearly interpolate between

two molecules.

Let us pick 1000 datapoints in the latent space.

```
[22]: num_inter_data = 1000

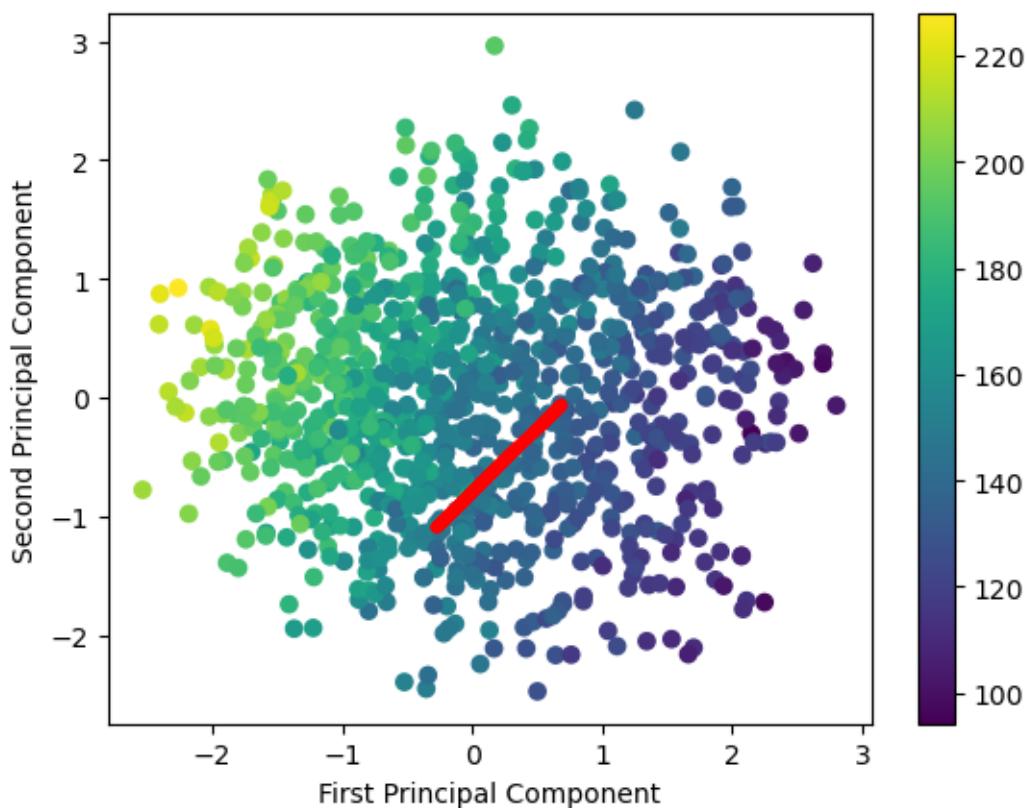
latent_z_inter = np.linspace(mean_z[7:8][0], mean_z[8:9][0], num_inter_data)
latent_z_inter_transform = data_pca.transform(latent_z_inter)

plt.scatter(data_pca_transform[:, 0], data_pca_transform[:, 1], c=target_pred)
plt.colorbar()

plt.scatter(latent_z_inter_transform[:, 0], latent_z_inter_transform[:, 1],
            marker='.', c='red')

plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')

plt.show()
```



We can now take these latent datapoints and decode molecules and their corresponding property value.

```
[23]: data_inter_pred = model.decoder(latent_z_inter)
target_inter_pred = model.decoder_property(latent_z_inter).numpy()

deep_smi_inter_pred = sparse_to_smiles(tf.argmax(data_inter_pred, axis=2).
                                         numpy())
deep_smi_inter_pred = [item.replace(stop_character, '') for item in
                        deep_smi_inter_pred]

deep_smi_inter_pred, idx = np.unique(deep_smi_inter_pred, return_index=True)

# Invert the scaling and convert the property unit to Kcal/mol
target_inter_pred = (target_inter_pred[idx] * band_gap_std + band_gap_mean) * 627
```

Parse generated molecules and sort out invalid samples.

```
[24]: import deepsmiles

converter = deepsmiles.Converter(rings=True, branches=True)

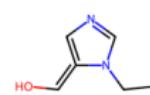
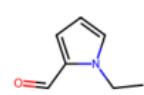
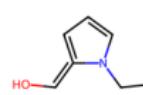
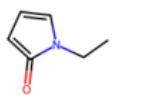
molecule_list = []
labels = []

for i, item in enumerate(deep_smi_inter_pred):
    try:
        smi = converter.decode(item)
        mol = Chem.MolFromSmiles(smi, sanitize = False)

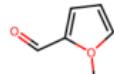
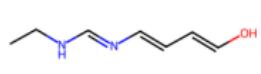
        if mol is not None:
            molecule_list.append(mol)
            p = target_inter_pred[i][0]
            labels.append(f'{smi} - {p:.2f} kcal/mol')
    except:
        pass

figure = Draw.MolsToGridImage(molecule_list, legends=labels, molsPerRow=4,
                                subImgSize=(300, 300), returnPNG=False)

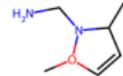
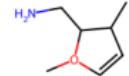
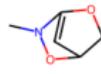
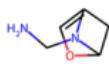
plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.axis('off')
plt.show()
```



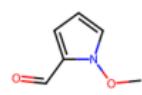
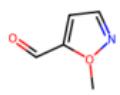
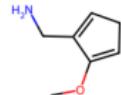
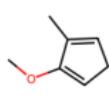
CCN1C=CC=C1=O - 139.47 kcal/mol CCN1C=CC=C1=CO - 139.47 kcal/mol CCN1C=CC=C1C=O - 139.46 kcal/mol CCN1C=NC=C1=CO - 139.48 kcal/mol



CCNC=NC=CC=CO - 139.48 kcal/mol CO1C=CC=C1C - 139.41 kcal/mol CO1C=CC=C1C=O - 139.41 kcal/mol C12OC=C(C1)N2C - 140.50 kcal/mol



C12OC=C(C1)N2CN - 140.41 kcal/mol C1OC2=CC1ON2C - 140.84 kcal/mol CO1C=CC(C)C1CN - 139.90 kcal/mol CO1C=CC(C)N1CN - 140.03 kcal/mol



COCl=CCC=C1C - 139.48 kcal/mol COCl=CCC=C1CN - 139.51 kcal/mol COIN=CC=C1C=O - 139.41 kcal/mol CON1C=CC=C1C=O - 139.40 kcal/mol

7-Image_Segmentation

May 7, 2024

1 7. Image Segmentation Example

In this last notebook, we see a medical imaging example for which we can make use of neural networks. To this end, we consider the [LIDC-IDRI](#) dataset of lung CT scans where anomalies, in particular lesions, were manually outlined by experts. The goal is to design a **neural network for image segmentation** which learns from previous outlines of lesions to predict anomalies in new lung CT scans.

Keywords: Semantic segmentation, U-Net, keras.layers.SeparableConv2D, keras.layers.BatchNormalization, keras.layers.UpSampling2D

1.1 U-Net-based Semantic Segmentation

Semantic segmentation describes the task of grouping pixels together which (semantically) belong to one object. In that regard, a particularly useful architecture for biomedical imaging tasks is the [U-Net](#) illustrated in the following figure. It is a **fully-convolutional neural network** and can be used also with comparably small datasets.

The architecture we use in this notebook is based on the U-Net. We adopt the *U-Net Xception-style model* proposed in a [Keras tutorial](#) on image segmentation.

With the following cell, we load and plot the dataset. Note that we only use a small subset (119 images) of the full dataset.

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

images = np.load('data/lidc_images_119.npy')
targets = np.load('data/lidc_labels_119.npy')

fig, axs = plt.subplots(10,2, figsize=(4,20))

for i in range(10):
    axs[i,0].imshow(images[i])
    axs[i,1].imshow(targets[i])

    axs[i,0].set_xticks([])
    axs[i,0].set_yticks([])
```

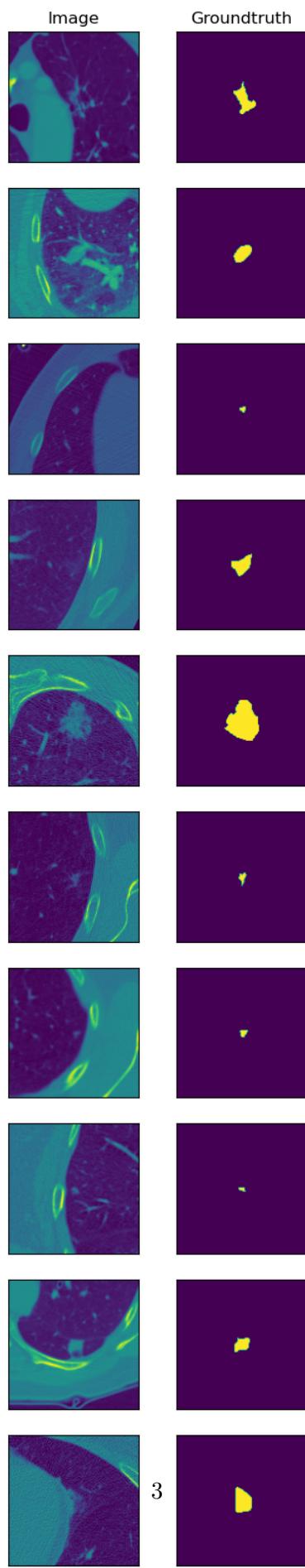
```
axs[i,1].set_xticks([])
axs[i,1].set_yticks([])

axs[0,0].set_title('Image')
axs[0,1].set_title('Groundtruth')

plt.show()

train_images = images[:100][...,np.newaxis]
train_targets = targets[:100][...,np.newaxis]

test_images = images[100:][...,np.newaxis]
test_targets = targets[100:][...,np.newaxis]
```



We use the following hyperparameters

```
[2]: learning_rate = 0.001
batch_size = 10
num_epochs = 10
```

and define the model as follows.

```
[4]: from tensorflow import keras

keras.utils.set_random_seed(123)

def segmentation_model(img_size=(128, 128), num_classes=2):
    inputs = keras.Input(shape=img_size + (1,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = keras.layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = keras.layers.BatchNormalization()(x)
    x = keras.layers.Activation("relu")(x)

    previous_block_activation = x # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = keras.layers.Activation("relu")(x)
        x = keras.layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = keras.layers.BatchNormalization()(x)

        x = keras.layers.Activation("relu")(x)
        x = keras.layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = keras.layers.BatchNormalization()(x)

        x = keras.layers.MaxPooling2D(3, strides=2, padding="same")(x)

    # Project residual
    residual = keras.layers.Conv2D(filters, 1, strides=2, padding="same")(
        previous_block_activation
    )
    x = keras.layers.add([x, residual]) # Add back residual
    previous_block_activation = x # Set aside next residual

    ### [Second half of the network: upsampling inputs] ###
```

```

for filters in [256, 128, 64, 32]:
    x = keras.layers.Activation("relu")(x)
    x = keras.layers.Conv2DTranspose(filters, 3, padding="same")(x)
    x = keras.layers.BatchNormalization()(x)

    x = keras.layers.Activation("relu")(x)
    x = keras.layers.Conv2DTranspose(filters, 3, padding="same")(x)
    x = keras.layers.BatchNormalization()(x)

    x = keras.layers.UpSampling2D(2)(x)

    # Project residual
    residual = keras.layers.UpSampling2D(2)(previous_block_activation)
    residual = keras.layers.Conv2D(filters, 1, padding="same")(residual)
    x = keras.layers.add([x, residual]) # Add back residual
    previous_block_activation = x # Set aside next residual

# Add a per-pixel classification layer
outputs = keras.layers.Conv2D(num_classes, 3, activation="softmax", ↴
    padding="same")(x)

# Define the model
model = keras.Model(inputs, outputs)

return model

```

2024-05-07 11:01:59.247029: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations. To enable the following instructions: SSE4.1 SSE4.2, in other operations, rebuild TensorFlow with the appropriate compiler flags.

Now, we build the model.

[5]: # Free up RAM in case the model definition cells were run multiple times
`keras.backend.clear_session()`

`model = segmentation_model(img_size=(128, 128), num_classes=2)`
`model.summary()`

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
<hr/>			
input_1 (InputLayer)	[(None, 128, 128, 1 0)]	[]	

```

conv2d (Conv2D)           (None, 64, 64, 32)  320
['input_1[0][0]']

batch_normalization (BatchNorm (None, 64, 64, 32) 128
['conv2d[0][0]']
alization)

activation (Activation)   (None, 64, 64, 32)  0
['batch_normalization[0][0]']

activation_1 (Activation) (None, 64, 64, 32)  0
['activation[0][0]']

separable_conv2d (SeparableCon (None, 64, 64, 64) 2400
['activation_1[0][0]']
v2D)

batch_normalization_1 (BatchNo (None, 64, 64, 64) 256
['separable_conv2d[0][0]']
rmalization)

activation_2 (Activation)  (None, 64, 64, 64)  0
['batch_normalization_1[0][0]']

separable_conv2d_1 (SeparableC (None, 64, 64, 64) 4736
['activation_2[0][0]']
onv2D)

batch_normalization_2 (BatchNo (None, 64, 64, 64) 256
['separable_conv2d_1[0][0]']
rmalization)

max_pooling2d (MaxPooling2D) (None, 32, 32, 64)  0
['batch_normalization_2[0][0]']

conv2d_1 (Conv2D)          (None, 32, 32, 64)  2112
['activation[0][0]']

add (Add)                 (None, 32, 32, 64)  0
['max_pooling2d[0][0]',
'conv2d_1[0][0]']

activation_3 (Activation)  (None, 32, 32, 64)  0      ['add[0][0]']

separable_conv2d_2 (SeparableC (None, 32, 32, 128) 8896
['activation_3[0][0]']
onv2D)

```

```

batch_normalization_3 (BatchNormal (None, 32, 32, 128) 512
['separable_conv2d_2[0][0]']
rmalization)

activation_4 (Activation)      (None, 32, 32, 128)  0
['batch_normalization_3[0][0]']

separable_conv2d_3 (SeparableConv2D (None, 32, 32, 128) 17664
['activation_4[0][0]']
onv2D)

batch_normalization_4 (BatchNormal (None, 32, 32, 128) 512
['separable_conv2d_3[0][0]']
rmalization)

max_pooling2d_1 (MaxPooling2D) (None, 16, 16, 128)  0
['batch_normalization_4[0][0]']

conv2d_2 (Conv2D)             (None, 16, 16, 128)  8320      ['add[0][0]']

add_1 (Add)                  (None, 16, 16, 128)  0
['max_pooling2d_1[0][0]',
'conv2d_2[0][0]']

activation_5 (Activation)     (None, 16, 16, 128)  0      ['add_1[0][0]']

separable_conv2d_4 (SeparableConv2D (None, 16, 16, 256) 34176
['activation_5[0][0]']
onv2D)

batch_normalization_5 (BatchNormal (None, 16, 16, 256) 1024
['separable_conv2d_4[0][0]']
rmalization)

activation_6 (Activation)     (None, 16, 16, 256)  0
['batch_normalization_5[0][0]']

separable_conv2d_5 (SeparableConv2D (None, 16, 16, 256) 68096
['activation_6[0][0]']
onv2D)

batch_normalization_6 (BatchNormal (None, 16, 16, 256) 1024
['separable_conv2d_5[0][0]']
rmalization)

max_pooling2d_2 (MaxPooling2D) (None, 8, 8, 256)   0
['batch_normalization_6[0][0]']

```

conv2d_3 (Conv2D)	(None, 8, 8, 256)	33024	['add_1[0] [0]']
add_2 (Add)	(None, 8, 8, 256)	0	
['max_pooling2d_2[0] [0]', 'conv2d_3[0] [0]']			
activation_7 (Activation)	(None, 8, 8, 256)	0	['add_2[0] [0]']
conv2d_transpose (Conv2DTranspose)	(None, 8, 8, 256)	590080	
['activation_7[0] [0]'] ose)			
batch_normalization_7 (BatchNormaliza	(None, 8, 8, 256)	1024	
['conv2d_transpose[0] [0]'] rmalization)			
activation_8 (Activation)	(None, 8, 8, 256)	0	
['batch_normalization_7[0] [0]']			
conv2d_transpose_1 (Conv2DTranspo	(None, 8, 8, 256)	590080	
['activation_8[0] [0]'] se)			
batch_normalization_8 (BatchNormaliz	(None, 8, 8, 256)	1024	
['conv2d_transpose_1[0] [0]'] rmalization)			
up_sampling2d_1 (UpSampling2D)	(None, 16, 16, 256)	0	['add_2[0] [0]']
up_sampling2d (UpSampling2D)	(None, 16, 16, 256)	0	
['batch_normalization_8[0] [0]']			
conv2d_4 (Conv2D)	(None, 16, 16, 256)	65792	
['up_sampling2d_1[0] [0]']			
add_3 (Add)	(None, 16, 16, 256)	0	
['up_sampling2d[0] [0]', 'conv2d_4[0] [0]']			
activation_9 (Activation)	(None, 16, 16, 256)	0	['add_3[0] [0]']
conv2d_transpose_2 (Conv2DTranspo	(None, 16, 16, 128)	295040	
['activation_9[0] [0]'] se)			
batch_normalization_9 (BatchNormaliz	(None, 16, 16, 128)	512	
['conv2d_transpose_2[0] [0]'] rmalization)			

```

activation_10 (Activation)      (None, 16, 16, 128)  0
['batch_normalization_9[0][0]']

conv2d_transpose_3 (Conv2DTran (None, 16, 16, 128)  147584
['activation_10[0][0]']
spose)

batch_normalization_10 (BatchN (None, 16, 16, 128)  512
['conv2d_transpose_3[0][0]']
ormalization)

up_sampling2d_3 (UpSampling2D) (None, 32, 32, 256)  0           ['add_3[0][0]']

up_sampling2d_2 (UpSampling2D) (None, 32, 32, 128)  0
['batch_normalization_10[0][0]']

conv2d_5 (Conv2D)             (None, 32, 32, 128)  32896
['up_sampling2d_3[0][0]']

add_4 (Add)                  (None, 32, 32, 128)  0
['up_sampling2d_2[0][0]',
'conv2d_5[0][0]']

activation_11 (Activation)    (None, 32, 32, 128)  0           ['add_4[0][0]']

conv2d_transpose_4 (Conv2DTran (None, 32, 32, 64)  73792
['activation_11[0][0]']
spose)

batch_normalization_11 (BatchN (None, 32, 32, 64)  256
['conv2d_transpose_4[0][0]']
ormalization)

activation_12 (Activation)    (None, 32, 32, 64)  0
['batch_normalization_11[0][0]']

conv2d_transpose_5 (Conv2DTran (None, 32, 32, 64)  36928
['activation_12[0][0]']
spose)

batch_normalization_12 (BatchN (None, 32, 32, 64)  256
['conv2d_transpose_5[0][0]']
ormalization)

up_sampling2d_5 (UpSampling2D) (None, 64, 64, 128)  0           ['add_4[0][0]']

up_sampling2d_4 (UpSampling2D) (None, 64, 64, 64)   0

```

```

['batch_normalization_12[0][0]']

conv2d_6 (Conv2D)           (None, 64, 64, 64)  8256
['up_sampling2d_5[0][0]']

add_5 (Add)                 (None, 64, 64, 64)  0
['up_sampling2d_4[0][0]', 
'conv2d_6[0][0]']

activation_13 (Activation)  (None, 64, 64, 64)  0          ['add_5[0][0]']

conv2d_transpose_6 (Conv2DTran (None, 64, 64, 32)  18464
['activation_13[0][0]']
spose)

batch_normalization_13 (BatchN (None, 64, 64, 32)  128
['conv2d_transpose_6[0][0]']
ormalization)

activation_14 (Activation)  (None, 64, 64, 32)  0
['batch_normalization_13[0][0]']

conv2d_transpose_7 (Conv2DTran (None, 64, 64, 32)  9248
['activation_14[0][0]']
spose)

batch_normalization_14 (BatchN (None, 64, 64, 32)  128
['conv2d_transpose_7[0][0]']
ormalization)

up_sampling2d_7 (UpSampling2D) (None, 128, 128, 64  0          ['add_5[0][0]']
)

up_sampling2d_6 (UpSampling2D) (None, 128, 128, 32  0
['batch_normalization_14[0][0]']
)

conv2d_7 (Conv2D)           (None, 128, 128, 32  2080
['up_sampling2d_7[0][0]']
)

add_6 (Add)                 (None, 128, 128, 32  0
['up_sampling2d_6[0][0]', 
])

'conv2d_7[0][0]'

conv2d_8 (Conv2D)           (None, 128, 128, 2)  578      ['add_6[0][0]']

```

```
=====
=====
Total params: 2,058,114
Trainable params: 2,054,338
Non-trainable params: 3,776
-----
```

With all of this in place, we can train our model. Note that **this can take a substantial time**, depending on your machine and the total amount of epochs.

```
[6]: model.compile(optimizer=keras.optimizers.SGD(learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

train_history = model.fit(x=train_images, y=train_targets,
                           batch_size=batch_size,
                           validation_data = (test_images, test_targets),
                           epochs=num_epochs)

fig, axs = plt.subplots(1,2, figsize=(9,3))

axs[0].plot(train_history.history['accuracy'], label='Training')
axs[0].plot(train_history.history['val_accuracy'], label='Validation')
axs[0].set_title('Accuracy')
axs[0].legend()

axs[1].plot(train_history.history['loss'])
axs[1].plot(train_history.history['val_loss'])
axs[1].set_title('Loss')

plt.show()
```

```
Epoch 1/10
10/10 [=====] - 5s 338ms/step - loss: 0.4990 -
accuracy: 0.8789 - val_loss: 0.5404 - val_accuracy: 0.9820
Epoch 2/10
10/10 [=====] - 3s 311ms/step - loss: 0.1389 -
accuracy: 0.9746 - val_loss: 0.4277 - val_accuracy: 0.9843
Epoch 3/10
```

```
KeyboardInterrupt
Cell In[6], line 5
  1 model.compile(optimizer=keras.optimizers.
  2                   SGD(learning_rate=learning_rate),
  3                   loss='sparse_categorical_crossentropy',
  4                   metrics=['accuracy'])
```

```

----> 5 train_history = model.fit(x=train_images, y=train_targets, u
↳batch_size=batch_size,
    6                               validation_data = (test_images, test_targets)
    7                               epochs=num_epochs)
    9 fig, axs = plt.subplots(1,2, figsize=(9,3))
   11 axs[0].plot(train_history.history['accuracy'], label='Training')

File ~/anaconda3/APML/lib/python3.11/site-packages/keras/utils/traceback_utils.
↳py:65, in filter_traceback.<locals>.error_handler(*args, **kwargs)
   63 filtered_tb = None
   64 try:
---> 65     return fn(*args, **kwargs)
   66 except Exception as e:
   67     filtered_tb = _process_traceback_frames(e.__traceback__)

File ~/anaconda3/APML/lib/python3.11/site-packages/keras/engine/training.py:
<1685, in Model.fit(self, x, y, batch_size, epochs, verbose, callbacks, u
↳validation_split, validation_data, shuffle, class_weight, sample_weight, u
↳initial_epoch, steps_per_epoch, validation_steps, validation_batch_size, u
↳validation_freq, max_queue_size, workers, use_multiprocessing)
  1677 with tf.profiler.experimental.Trace(
  1678     "train",
  1679     epoch_num=epoch,
  (...),
  1682     _r=1,
  1683 ):
  1684     callbacks.on_train_batch_begin(step)
-> 1685     tmp_logs = self.train_function(iterator)
  1686     if data_handler.should_sync:
  1687         context.async_wait()

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/util/
↳traceback_utils.py:150, in filter_traceback.<locals>.error_handler(*args, u
↳**kwargs)
  148 filtered_tb = None
  149 try:
--> 150     return fn(*args, **kwargs)
  151 except Exception as e:
  152     filtered_tb = _process_traceback_frames(e.__traceback__)

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳polymorphic_function/polymorphic_function.py:894, in Function.__call__(self, u
↳*args, **kwds)
  891 compiler = "xla" if self._jit_compile else "nonXla"
  893 with OptionalXlaContext(self._jit_compile):
--> 894     result = self._call(*args, **kwds)
  896 new_tracing_count = self.experimental_get_tracing_count()
  897 without_tracing = (tracing_count == new_tracing_count)

```

```

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳ polymorphic_function/polymorphic_function.py:926, in Function._call(self, *args, **kwds)
    923     self._lock.release()
    924     # In this case we have created variables on the first call, so we run
    ↵the
    925     # defunned version which is guaranteed to never create variables.
--> 926     return self._no_variable_creation_fn(*args, **kwds) # pylint:u
    ↵disable=not-callable
    927 elif self._variable_creation_fn is not None:
    928     # Release the lock early so that multiple threads can perform the cal
    929     # in parallel.
    930     self._lock.release()

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳ polymorphic_function/tracing_compiler.py:143, in TracingCompiler.
    ↵__call__(self, *args, **kwargs)
    140 with self._lock:
    141     (concrete_function,
    142      filtered_flat_args) = self._maybe_define_function(args, kwargs)
--> 143 return concrete_function._call_flat(
    144     filtered_flat_args, captured_inputs=concrete_function.
    ↵captured_inputs)

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳ polymorphic_function/monomorphic_function.py:1757, in ConcreteFunction.
    ↵_call_flat(self, args, captured_inputs, cancellation_manager)
    1753 possible_gradient_type = gradients_util.PossibleTapeGradientTypes(args)
    1754 if (possible_gradient_type == gradients_util.POSSIBLE_GRADIENT_TYPES_NONE
    1755     and executing_eagerly):
    1756     # No tape is watching; skip to running the function.
-> 1757     return self._build_call_outputs(self._inference_function.call(
    1758         ctx, args, cancellation_manager=cancellation_manager))
    1759 forward_backward = self._select_forward_and_backward_functions(
    1760     args,
    1761     possible_gradient_type,
    1762     executing_eagerly)
    1763 forward_function, args_with_tangents = forward_backward.forward()

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
↳ polymorphic_function/monomorphic_function.py:381, in _EagerDefinedFunction.
    ↵call(self, ctx, args, cancellation_manager)
    379 with _InterpolateFunctionError(self):
    380     if cancellation_manager is None:
--> 381         outputs = execute.execute(
    382             str(self.signature.name),
    383             num_outputs=self._num_outputs,
    384             inputs=args,
    385             attrs=attrs,

```

```

386         ctx=ctx)
387     else:
388         outputs = execute.execute_with_cancellation(
389             str(self.signature.name),
390             num_outputs=self._num_outputs,
391             ...)
392         ctx=ctx,
393         cancellation_manager=cancellation_manager)

File ~/anaconda3/APML/lib/python3.11/site-packages/tensorflow/python/eager/
→ execute.py:52, in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
  50 try:
  51     ctx.ensure_initialized()
--> 52     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name, op_name
  53                                         inputs, attrs, num_outputs)
  54 except core._NotOkStatusException as e:
  55     if name is not None:

```

KeyboardInterrupt:

Save the trained model:

```
[ ]: model.save_weights('output/Segmentation_model.h5', save_format='h5')
```

We provide again a pretrained model (500 epochs):

```
[7]: model.load_weights('output/Segmentation_model_pretrained.h5')
```

Make model prediction for both the training and test images.

```
[8]: train_prediction = model.predict(train_images)
test_prediction = model.predict(test_images)
```

```
4/4 [=====] - 1s 228ms/step
1/1 [=====] - 0s 199ms/step
```

First, we examine how well the model predicts the segmentation result on the training set.

```
[9]: fig, axs = plt.subplots(10,3, figsize=(6,20))

for i in range(10):

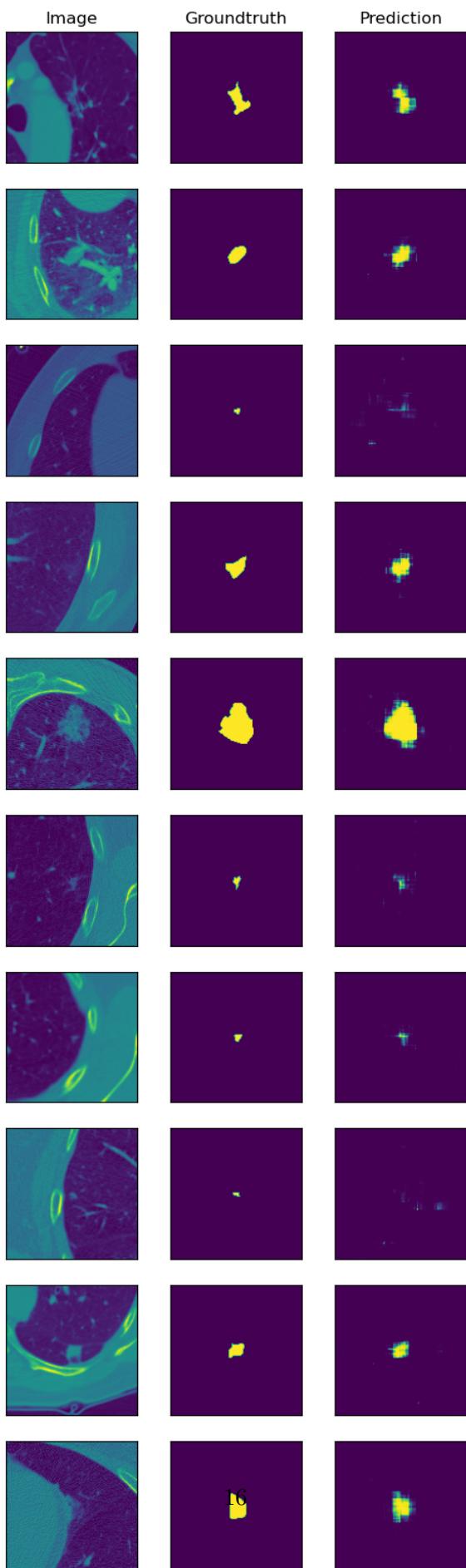
    axs[i,0].imshow(train_images[i])
    axs[i,1].imshow(train_targets[i])
    axs[i,2].imshow(train_prediction[i,...,1])

    axs[i,0].set_xticks([])
    axs[i,0].set_yticks([])
    axs[i,1].set_xticks([])
```

```
axs[i,1].set_yticks([])
axs[i,2].set_xticks([])
axs[i,2].set_yticks([])

axs[0,0].set_title('Image')
axs[0,1].set_title('Groundtruth')
axs[0,2].set_title('Prediction')

plt.show()
```



Lastly, let's examine the predictions of segments on the test images.

```
[10]: fig, axs = plt.subplots(10,3, figsize=(6,20))

for i in range(10):

    axs[i,0].imshow(test_images[i])
    axs[i,1].imshow(test_targets[i])
    axs[i,2].imshow(test_prediction[i,...,1])

    axs[i,0].set_xticks([])
    axs[i,0].set_yticks([])
    axs[i,1].set_xticks([])
    axs[i,1].set_yticks([])
    axs[i,2].set_xticks([])
    axs[i,2].set_yticks([])

    axs[0,0].set_title('Image')
    axs[0,1].set_title('Groundtruth')
    axs[0,2].set_title('Prediction')

plt.show()
```

