Exercise 1: Polynomial Regression

```
import numpy as np
import matplotlib.pyplot as plt
import warnings
```

In this exercise our aim is to check what happens when we try to learn from data in a simple setting.

We do this by using polynomial functions.

Step 1: Define the hidden functions

First of all we generate the data from *hidden* functions: given a certain analytical function, we generate a subset of (x, y) tuples from it, and we take them as the starting ground for our exercise.

We will generate data from two *hidden* functions:

$$f_A(x) = 2x,$$

and

$$f_B(x) = 2x - 10x^5 + 15x^10.$$

```
# Defining the (hidden) functions to generate the data

def funcA(x):
    return 2*x

def funcB(x):
    return 2*x - 10*x**5 + 15*x**10

def getY(func, mu, sigma, x):
    return func(x) + rng.normal(mu, sigma, len(x))
```

```
funcs = {
    'A': funcA,
    'B': funcB
}
```

Step 2: Generate data from the hidden functions

We will generate 10 tuples. Since we want them to be the *exact* function, we will generate them from the general function

$$y_\mu = f(x_\mu) + \eta_\mu$$

where the η_μ is always identically zero (i.e. mean zero and variance zero).

We define a function `get_dataset` in order to get tuples from the above formula, given a domain of the x values, the number of tuples, and the mean and variance of the noise term η. It returns the x and y arrays, depending on the `func_name` name of the function generating the dataset (i.e. A and B in this case).

```
def get_dataset(x_min, x_max, n_pairs,
                mu, sigma, func_name):
    x = rng.uniform(x_min, x_max, n_pairs)
    y_dict = {}

    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        y = getY(funcs[func_name], mu,
                 sigma, x)

    return x, y
```

Step 3: Fit the data

The following function `get_polyfunc` yields the array of polynomial coefficients, given a degree of the polynomial and the name of the generating function. `get_polyxy` yields instead the tuple containing the x and y sample values of the polynomial function for plotting purposes with, say, 100 points, given the polynomial degree and the name function.

```
def get_polyfunc(deg, n_pairs, func_name):
    x, y = get_dataset(x_min=0, x_max=1,
                       n_pairs=n_pairs, mu=0,
```

```
                        sigma=0, func_name=func_name)
    p = np.polynomial.Polynomial.fit(x, y, deg=deg)
    coef = p.convert().coef
    return coef

def get_polyxy(x_min, x_max, deg, n_pairs, func_name):
    coef = get_polyfunc(deg=deg, n_pairs=n_pairs,
                        func_name=func_name)
    x = np.linspace(x_min, x_max, 100)
    y = np.polynomial.polynomial.polyval(x, coef, tensor=False)
    return (x, y)
```

Step 4: Generate the test dataset

We now generate the shape of the polynomial (which we supposedly know only by *observation* and our goal is to infer the model).

We generate the polynomials in the $[0, 1.25]$ interval so that we can compare them to the test data.

We do this for each of the two generating functions, `A` and `B` and call the values `x_c` and `y_c`. We also generate the so-called *test* dataset (`x_t` vs. `y_t`).

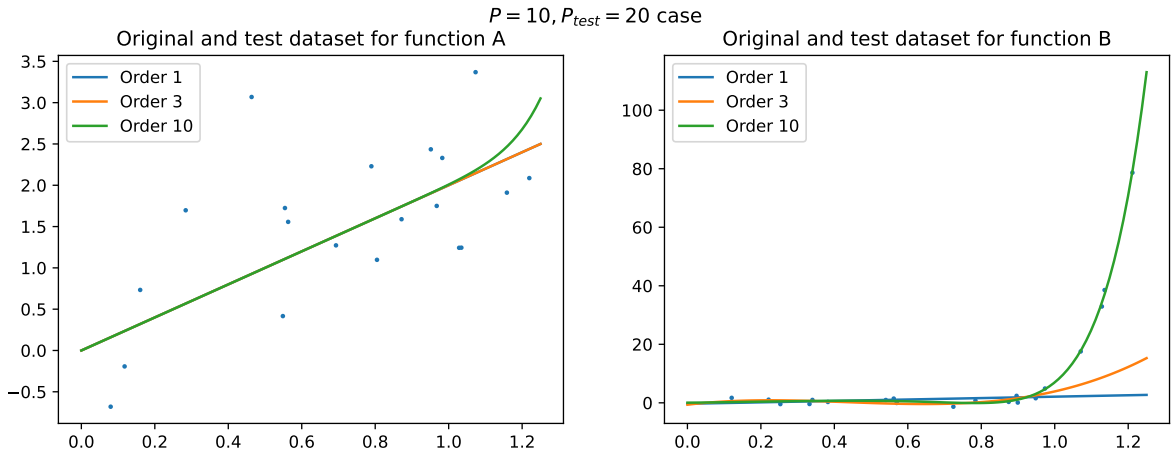
```
# this is for reproducibility
rng = np.random.default_rng(42)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,4))

fig.suptitle("$P=10, P_{test}=20$ case")
for func_name, ax in zip(['A', 'B'], [ax1, ax2]):
    ax.set_title(f'Original and test dataset for function {func_name}')
    x_t, y_t = get_dataset(x_min=0, x_max=1.25,
                           n_pairs=20, mu=0, sigma=1,
                           func_name=func_name)
    ax.scatter(x_t, y_t, s=3)
    for deg in [1, 3, 10]:
        x_c, y_c = get_polyxy(x_min=0, x_max=1.25,
                              deg=deg, n_pairs=10,
                              func_name=func_name)
        ax.plot(x_c, y_c, label=f'Order {deg}')
    ax.legend()
```

```
plt.show()
```

```
/home/shoichi/Documenti/FisicaLM/Cammarota/spml_exercises/env/lib/python3.11/site-packages/nu
  return pu._fit(polyvander, x, y, deg, rcond, full, w)
/home/shoichi/Documenti/FisicaLM/Cammarota/spml_exercises/env/lib/python3.11/site-packages/nu
  return pu._fit(polyvander, x, y, deg, rcond, full, w)
```



Step 5 and 6: How well does the fit describe the test dataset?

We can comment on the previous figures:

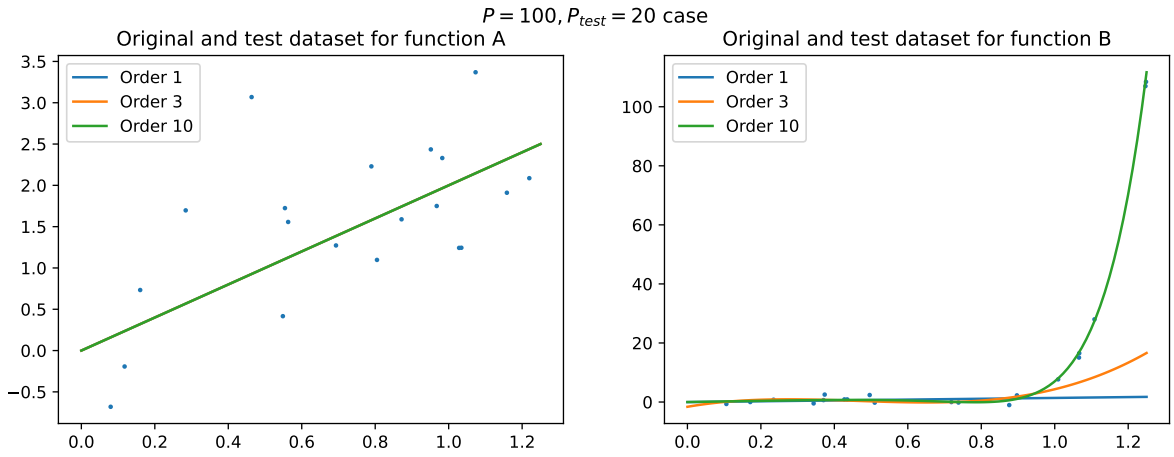
- For **function A**, order 1 and order 3 polynomial behave well (are nearly linear) in the whole domain. Order 10 instead diverges from the linear behaviour for values of $x > 1.0$, giving us a hint that probably the presence of high order terms for a dataset which is originally of lower order yields to an overfitting behaviour.
- Instead for **function B** it is the very polynomial of order 10 that well explains data points that are *out-of-sample* (i.e. for $x > 1.0$), while polynomials of order 3 and 1 poorly perform, especially in that domain.

Step 7: Let's check what happens for a bigger original dataset generating the polynomials

```
# this is for reproducibility
rng = np.random.default_rng(42)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,4))

fig.suptitle("$P=100, P_{test}=20$ case")
for func_name, ax in zip(['A', 'B'], [ax1, ax2]):
    ax.set_title(f'Original and test dataset for function {func_name}')
    x_t, y_t = get_dataset(x_min=0, x_max=1.25,
                           n_pairs=20, mu=0, sigma=1,
                           func_name=func_name)
    ax.scatter(x_t, y_t, s=3)
    for deg in [1, 3, 10]:
        x_c, y_c = get_polyxy(x_min=0, x_max=1.25,
                              deg=deg, n_pairs=100,
                              func_name=func_name)
        ax.plot(x_c, y_c, label=f'Order {deg}')
    ax.legend()
plt.show()
```



Step 8: How this affects the results?

For the $P = 100, P_{test} = 20$ case the **function B** plot doesn't seem to have changed much. This may be due to the fact that a higher number of points to generate the fits have killed

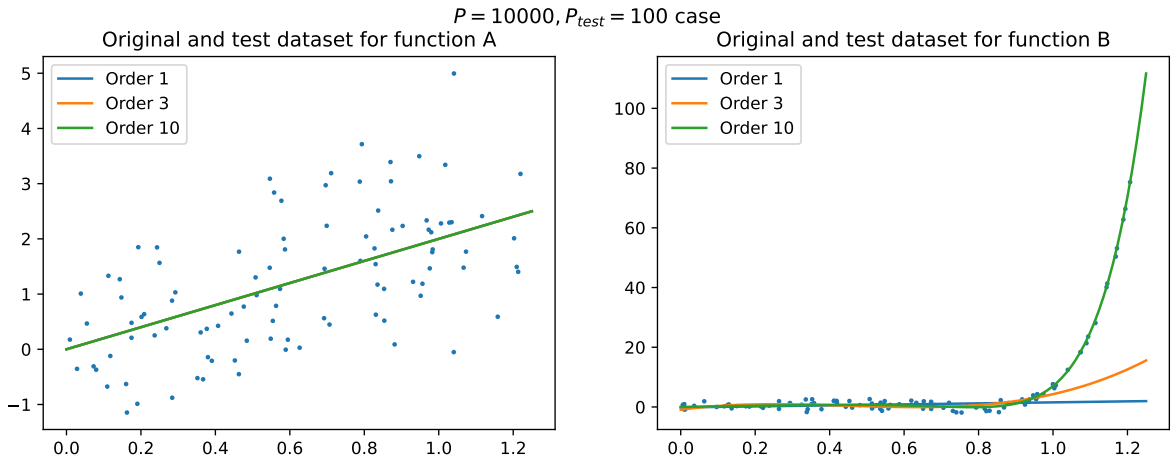
higher order terms. Instead, the **function A** has seen the Order 10 polynomial *straightened up*. This may be explained by the fact that, for polynomials of order 1 and 3 much hasn't changed since they are by construction very out of the way to fitting such a nonlinear function (with finite order 5 and order 10 terms). The polynomial of order 10 instead had previously already enough information to *catch the signal*.

Step 9: Let's check, instead, for the case of a much bigger original dataset, and a bigger test dataset as well

```
# this is for reproducibility
rng = np.random.default_rng(42)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,4))

fig.suptitle("$P=10000, P_{test}=100$ case")
for func_name, ax in zip(['A', 'B'], [ax1, ax2]):
    ax.set_title(f'Original and test dataset for function {func_name}')
    x_t, y_t = get_dataset(x_min=0, x_max=1.25,
                           n_pairs=100, mu=0, sigma=1,
                           func_name=func_name)
    ax.scatter(x_t, y_t, s=3)
    for deg in [1, 3, 10]:
        x_c, y_c = get_polyxy(x_min=0, x_max=1.25,
                              deg=deg, n_pairs=10_000,
                              func_name=func_name)
        ax.plot(x_c, y_c, label=f'Order {deg}')
    ax.legend()
plt.show()
```



Step 10: What happens when the dataset is larger?

When the dataset is larger we have more data points, which means that our predicted models have a stronger predictive power. The robustness of the predictions can be seen from the behaviour of the *out-of-sample* polynomial, which is consistent with the previous case as well.