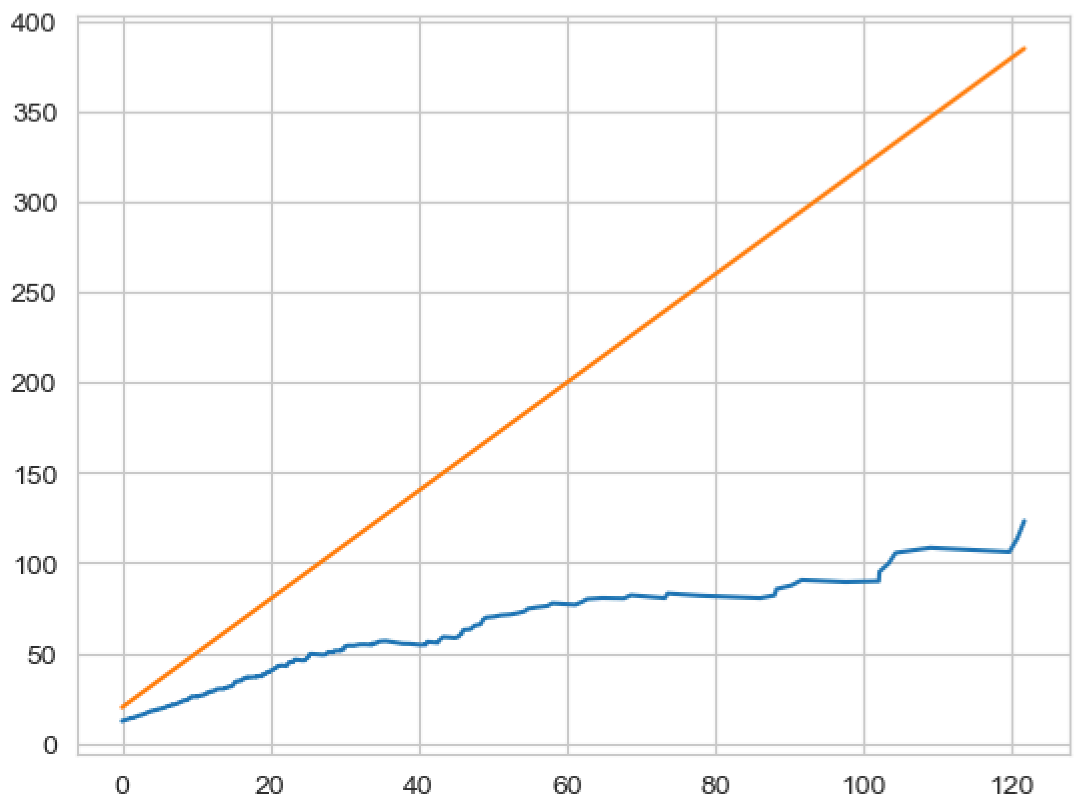
```python
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from statsmodels.sandbox.distributions.genpareto import meanexcess
```

```
[1.  0.5 0.  0.  0. ]
[1.  0.5 0.  0.  0. ]
[0.   0.75 1.   1.   1.  ]
[0.   0.75 1.   1.   1.  ]
[1.   0.25 0.   0.   0.  ]
[1.   0.25 0.   0.   0.  ]
[0.01002513 0.1026334  0.58578644 1.36754447 1.8       ]
[0.01002513 0.1026334  0.58578644 1.36754447 1.8       ]
[1.8        1.36754447 0.58578644 0.1026334  0.01002513]
[1.8        1.36754447 0.58578644 0.1026334  0.01002513]
(np.float64(0.6666666666666667), np.float64(0.22222222222222243), np.float64(0.565685
4249492306), np.float64(-0.5999999999999694))
(np.float64(0.6666666666666666), np.float64(0.2222222222222222), np.float64(0.5656854
24949238), np.float64(-0.6000000000000001))
0.49999999999999983
0.5
25.0
shape > -1 does not hold
-20.0
[ 90.58975456  89.44479883  89.93537016  95.15286424  99.8338829
 105.5815496  108.42828071 106.1187639  113.86339791 123.30913557
 133.25991457 142.96000261  87.26977449  97.03653367 111.8009111
 129.58208396 107.54166214 137.41517792 191.60503101   0.        ]
[ 91.64720504  97.56004259 102.03862675 102.11144857 103.37748392
 104.28540941 108.98021756 119.63037135 120.72896766 121.634448
 124.01458255 129.12115169 202.68138013 205.38173159 206.79010977
 211.36911913 265.80506194 271.77876688 286.29650275 669.50656477]
[1. 2. 3. 4. 5. 6. 7. 8. 9.]
[4.  3.5 3.  2.5 2.  1.5 1.  0.5 0. ]
[5.  5.5 6.  6.5 7.  7.5 8.  8.5 9. ]
1.1410777703680384
[0.         0.50916043 0.79788456 1.14107777]
-9.521272659185343e-18 0.509942585126642 0.7998363037131325 1.1409433927783235
```



```
RANDOM_SEED = 0xdeadbeef
```

Lab 02: Linear Regression

For the first tasks, we will work with synthetic univariate data. We generate 100 features $x_i \in [-1, 1]$ as `x` and two different regression targets `y1` and `y2`.

```
data_rng = np.random.default_rng(RANDOM_SEED)
n = 100
x = 2 * data_rng.random(n) - 1  # create n points between -1 and 1

# setup synthetic y1
true_offset = 0.5
true_slope = 1.25
noise = data_rng.normal(loc=0., scale=0.25, size=(n,))

y1 = true_offset + true_slope * x + noise


# setup synthetic y2
y2  = true_offset + np.sin(np.pi * x) + noise
```

Task 1: Scatter Plot

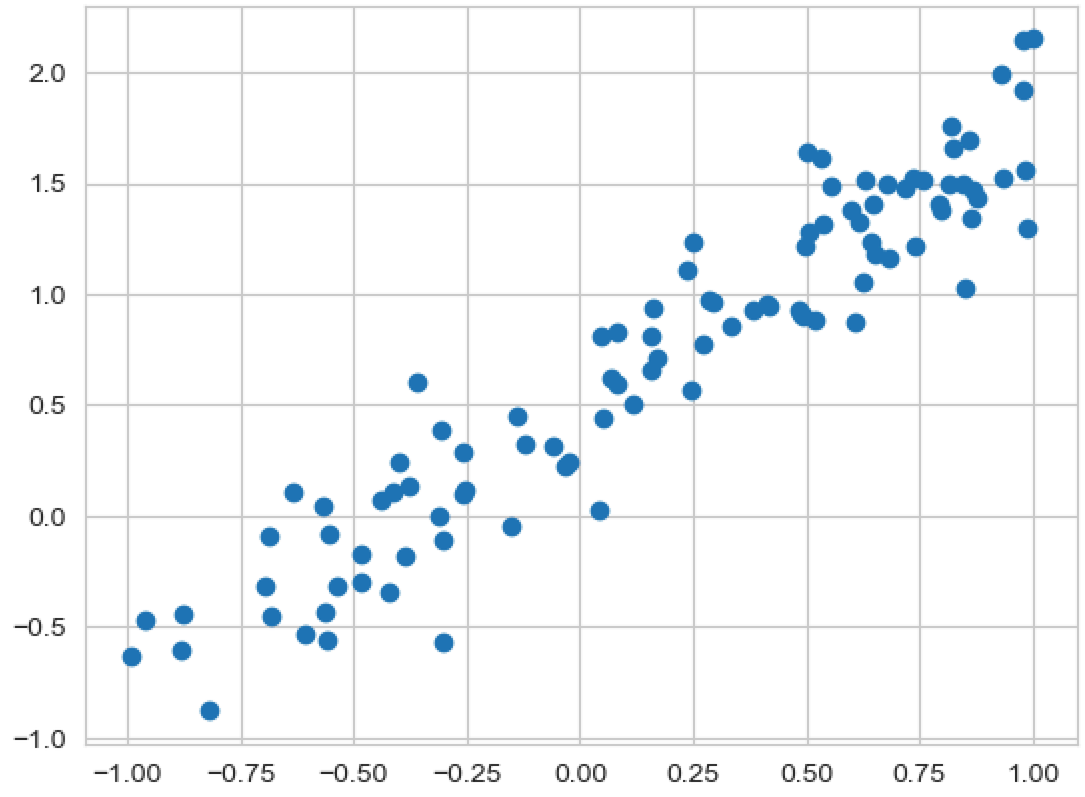
In the next cell, we show you how you can use `plt.scatter` to create scatter-plots.

A scatter plot is a graphical representation that displays individual data points based on two variables, with one variable plotted along the x-axis and the other plotted along the y-axis. It's commonly used to observe and show relationships between two numeric variables. Plot `x` against the target variable `y1`.

The simplest way to create a scatter-plot in `matplotlib` is by calling `plt.scatter(x, y)` where `x` is a list or array of x-coordinates and `y` is a list or array of y-coordinates.

```
# Let us create a scatter-plot of x and y1
plt.scatter(x, y1)
```

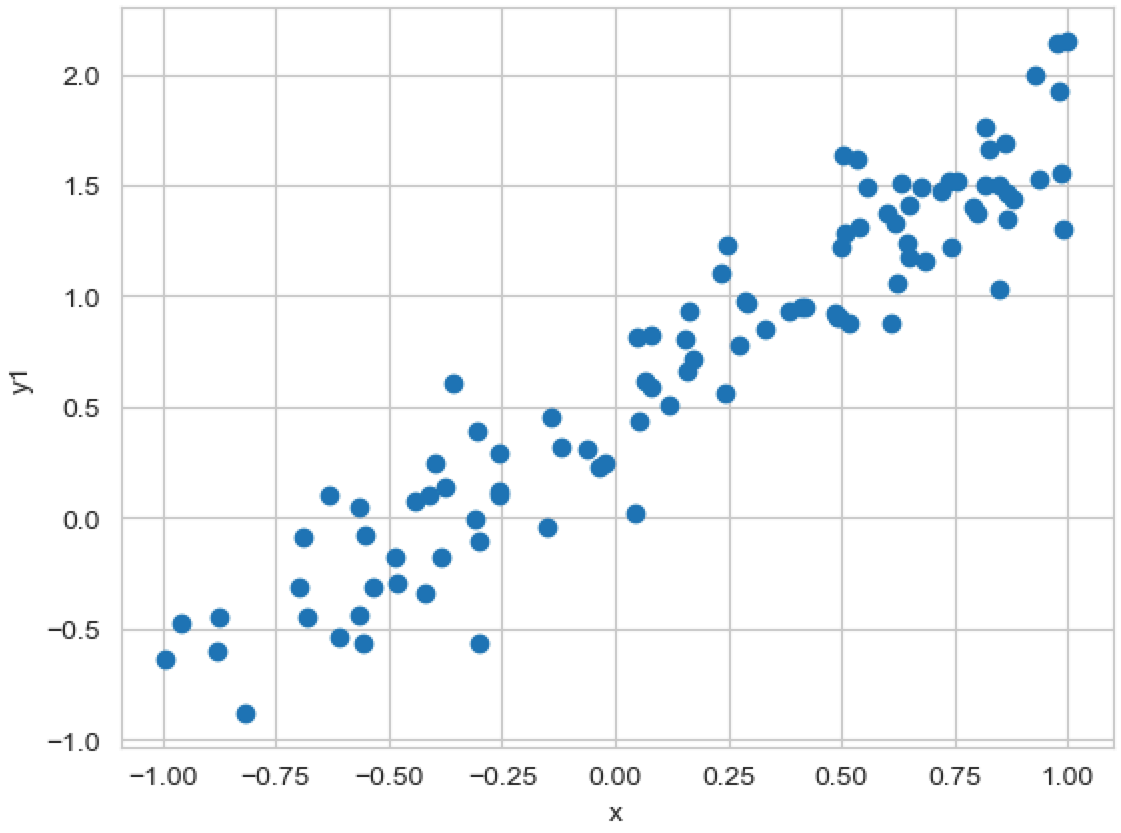
```
<matplotlib.collections.PathCollection at 0x30f31f4a0>
```



In the next cell, we re-create the same plot but also label the axes, which is generally a good practice.

In `matplotlib` it is common to build a plot incrementally by calling many functions (such as `plt.xlabel` and `plt.ylabel` here), before displaying the result using `plt.show()`.

```
plt.scatter(x, y1)
plt.xlabel("x")
plt.ylabel("y1")
plt.show()
```

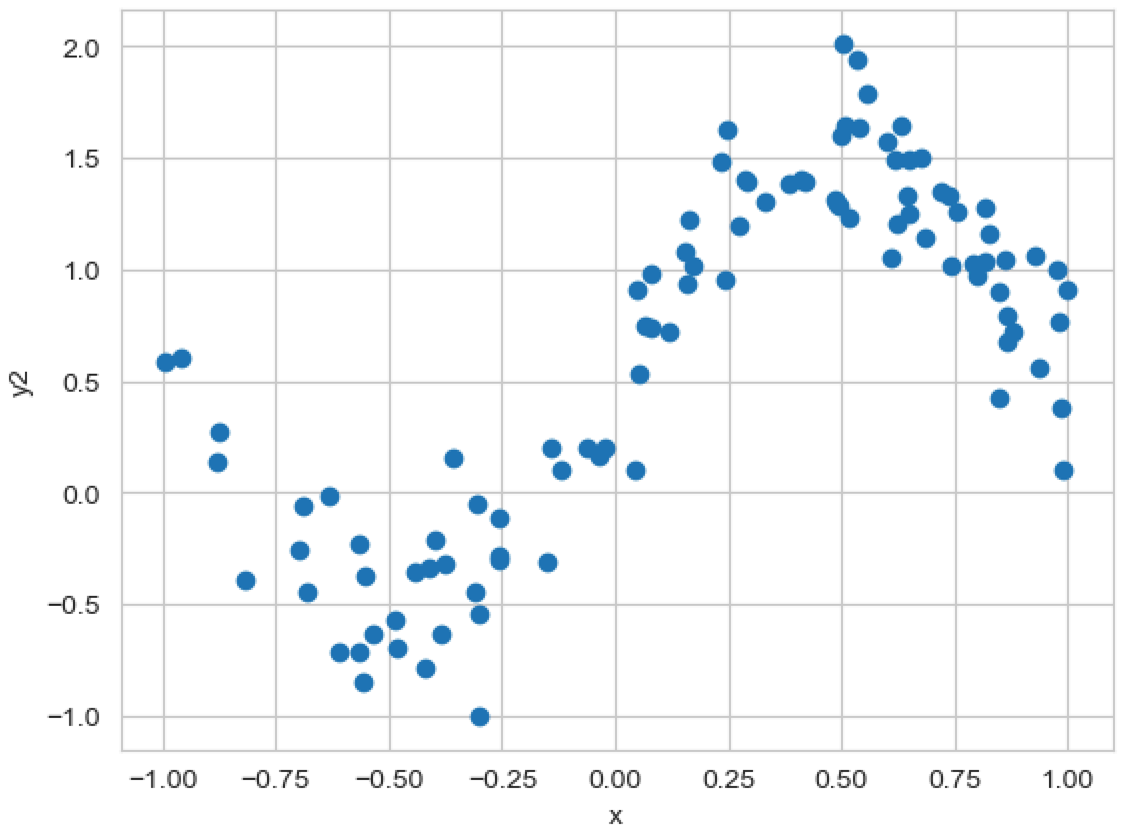


🚨 Task 1A (1 Point) 🚨

Your turn:

- create a scatter-plot of `x` and `y2`.
- Study the plot: do you think the relationship between `x` and `y2` is linear?

```
# TODO
plt.scatter(x, y2)
plt.xlabel("x")
plt.ylabel("y2")
plt.show()

# The graph definitely does not look linear, as it has a sinusoidal shape.
```



📢 **On Moodle** 📢

- Report whether the relationship between `x` and `y2` is linear.

Task 2: Univariate Linear Regression

You will now implement Linear Regression with a single variable. In class you have seen that the underlying model is: $y = \theta_0 + \theta_1 x$.

You also derived the closed formula for θ_0 and θ_1:

- $\hat{\theta}_1 = \frac{\sum_{i=1}^{m}(x_i - \mu(x))(y_i - \mu(y))}{\sum_{i=1}^{m}(x_i - \mu(x))^2}$
- $\hat{\theta}_0 = \mu(y) - \hat{\theta}_1 \mu(x)$

🚨 Task 2A (2 Points) 🚨

In the following cell, implement the `.fit` and `.predict` methods:

- In the `.predict` method you will have to apply the model to the input `x`
- In the `.fit` method you will have to compute $\hat{\theta}_0$ and $\hat{\theta}_1$.

```
class UnivariateLinearRegression:
  def __init__(self):
    self.theta_0: float = 0.
    self.theta_1: float = 0.

  def predict(self, x):
    y_pred = self.theta_0 + self.theta_1 * x
    return y_pred

  def fit(self, x, y):
    self.theta_1 = np.sum((x - np.mean(x)) * (y - np.mean(y))) / np.sum((x - np.mean(
    self.theta_0 = np.mean(y) - self.theta_1 * np.mean(x)


  def show_parameters(self):
    print(f"theta_0: {self.theta_0}, theta_1: {self.theta_1}")

    return self
```

📢 **On Moodle**: 📢

- Make a snapshot, or copy and submit the code you have written in the cell above.

Now we fit the linear model to `x` and the target `y1`:

- Create an instance of the class `UnivariateLinearRegression`
- fit the model using its `.fit` method
- get the predicted values, using `.predict`

```
lin_reg_uni = UnivariateLinearRegression()
lin_reg_uni.fit(x, y1)
y_pred = lin_reg_uni.predict(x)
lin_reg_uni.show_parameters()
```

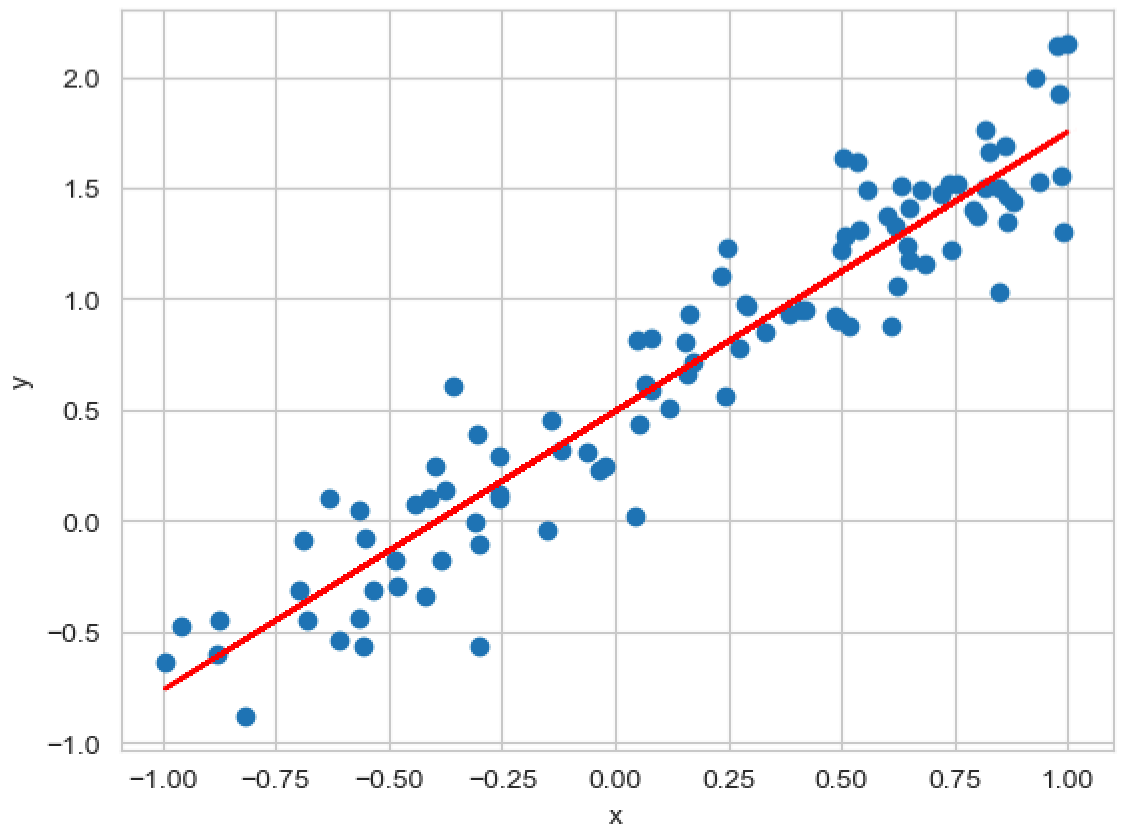
```
theta_0: 0.4931014561351984, theta_1: 1.2589807859200781
<__main__.UnivariateLinearRegression at 0x30f31fd40>
```

In the next cell, we provide a helper function to visualize your fitted linear model, based on `x`, the true values of `y` (`y_true`) and the predicted values of `y` (`y_pred`):

```
def plot_model(x, y_pred, y_true):
  # scatter plot of the true data points
  plt.scatter(x, y_true)
  # plot the fitted line
  plt.plot(x, y_pred, c="r")
  # label axes
  plt.xlabel("x")
  plt.ylabel("y")
  plt.show()
```

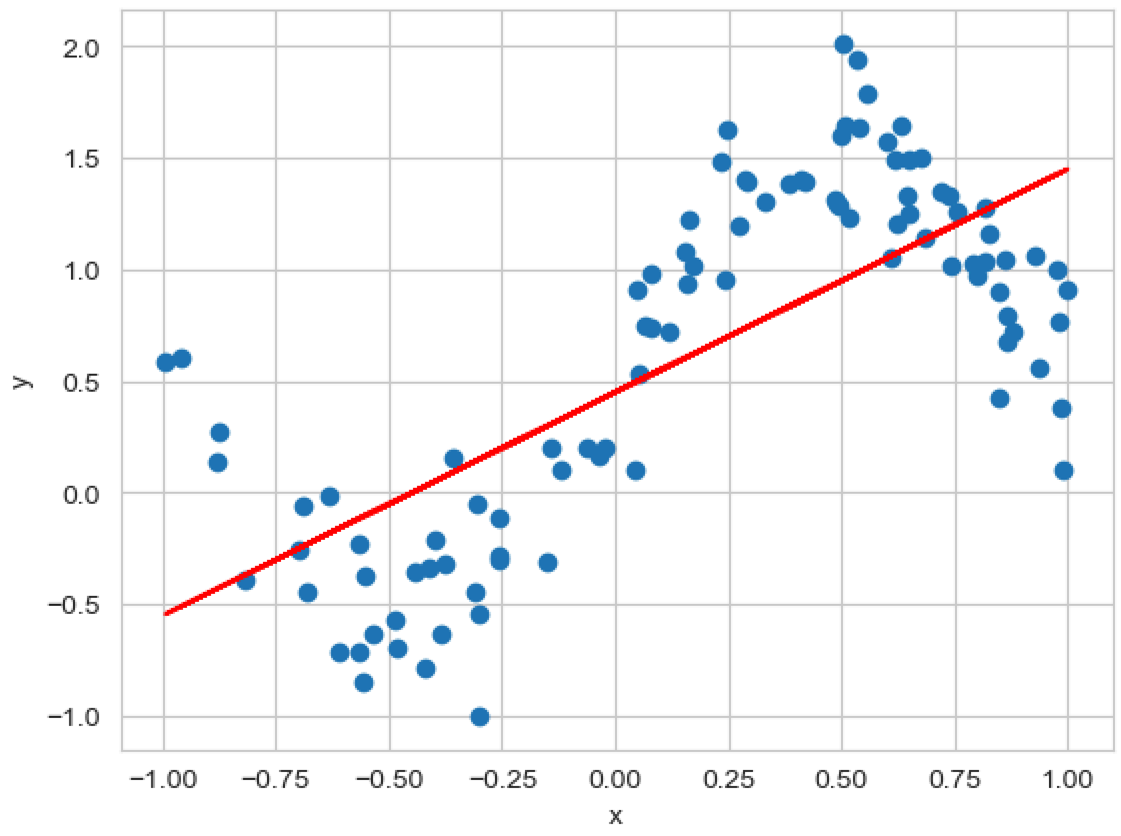
We use `plot_model` to visualize to outcome of our linear regression for `x` and `y1`.

```
plot_model(x=x, y_pred=y_pred, y_true=y1)
```



Now we repeat the same steps for `x` and `y2`. Inspect the plot and reflect on your answer to Task 2A.

```
lin_reg_uni = UnivariateLinearRegression()
lin_reg_uni.fit(x, y2)
y_pred = lin_reg_uni.predict(x)

plot_model(x, y_pred, y2)
```



Task 3: Multivariate Linear Regression using the Normal Equation

In the next cell, we provide a function to generate synthetic data for multivariate linear regression.

```python
def create_random_data(m: int, n: int, random_seed: int = RANDOM_SEED):
  """
  m: number of samples
  n: number of dimensions
  random_seed: seed to initialize random number generator
  """
  rng = np.random.default_rng(random_seed)
  # random data
  X = rng.standard_normal(size=(m, n))
  # random true model parameters
  theta = rng.standard_normal(n)
  # random noise
  noise = rng.normal(loc=0., scale=.25, size=m)
  # observed y values
  y = X @ theta + noise
  return X, y

# create synthetic dataset with 100 observations and 10 dimensions
X, y = create_random_data(100, 10)
```

In class you have seen that the underlying model for multivariate linear regression is: $y = X\theta$. Here $X \in \mathbb{R}^{m,n}$, $\theta \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$.

We also derived a closed formula for θ:

$$\hat{\theta} = (X^T X)^{-1} X^T y$$

This is known as the *normal equation*.

The normal equation can be used to solve univariate and also multivariate linear regression problems, and we have shown that it yields optimal parameters.

In the next cell we implement the normal equation $\hat{\theta} = (X^T X)^{-1} X^T y$ and apply it to our data `X` and `y`.

`numpy` reminder:

- the transpose of `X` is written `X.T`
- matrix multiplication is written with the `@` symbol, e.g. `A @ B`.
- you can compute the inverse of a matrix `A` using `np.linalg.inv(A)`. However, it is more numerically stable (and yes, this can really make a difference in practice!) to use a method such as `pinv` or `solve`, as noted below.

```
theta = (np.linalg.inv(X.T @ X) @ X.T) @ y
```

Computing the matrix inverse as part of a larger expression: when solving the normal equation, we don't actually care about the inverse $(X^T X)^{-1}$ by itself; rather, we care about its value *when multiplied by another vector*. In particular, the expression $(X^T X)^{-1} X^T$ is also known as the Moore-Penrose Pseudoinverse of X and can be computed directly using `np.linalg.pinv(X)`, which is generally more efficient. Alternatively, you can solve the normal equation as `np.linalg.solve(X.T @ X, X.T @ y)`.

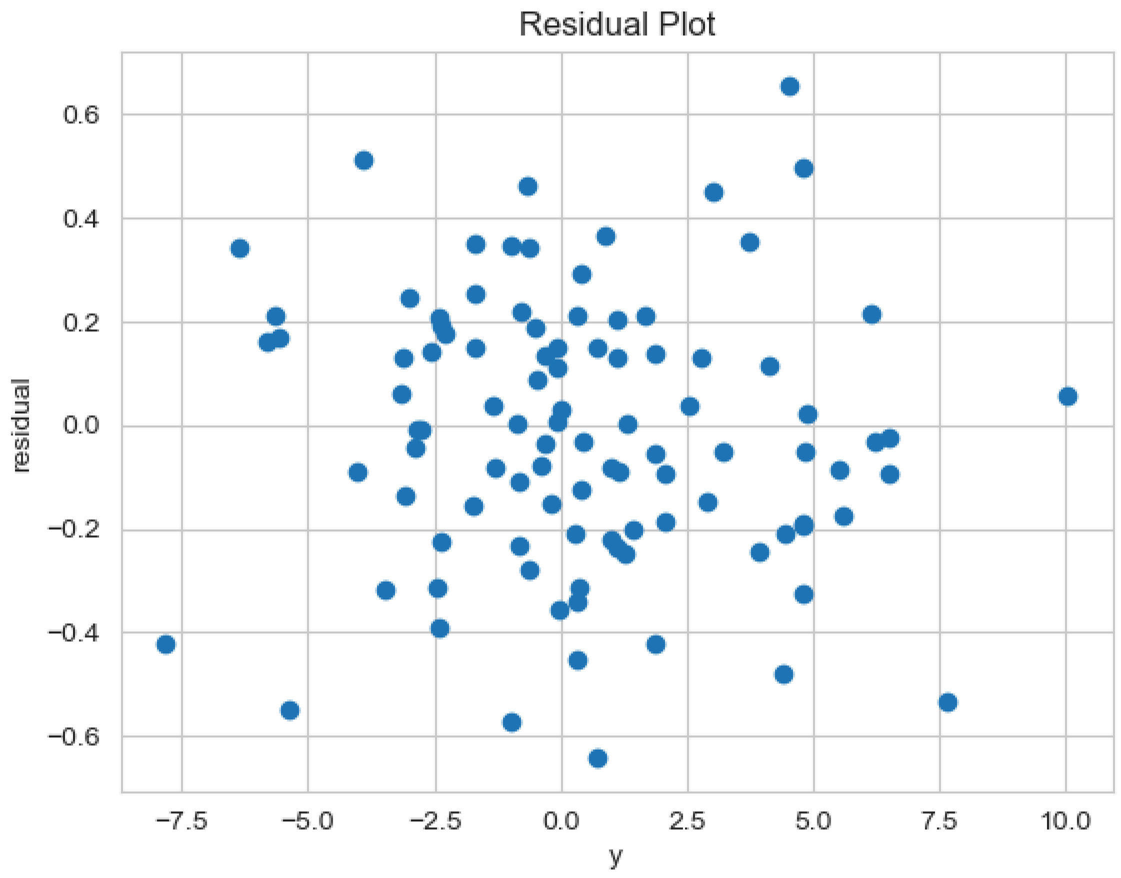
In the next cell, we provide a helper function to create a residual plot based on the true values of `y` (`y_true`) and the predicted values of `y` (`y_pred`).

```
def residual_plot(y_true, y_pred):
  residual = y_pred - y_true
  plt.scatter(y_true, residual)
  plt.xlabel("y")
  plt.ylabel("residual")
  plt.title("Residual Plot")
  plt.show()
```

Below we compute the predicted values of `y` based on the value of θ you computed using the normal equation. Then we plot the residuals using `residual_plot`.

```
y_pred = X @ theta
```

```
residual_plot(y_true=y, y_pred=y_pred)
```



Performance of Normal Equation

Max Clever has seen the slides for next lecture already – where we will discuss alternative methods such as "gradient descent" for linear regression.

Now he wonders: *Why do we need other algorithms, when the normal equation solves the problem already?*

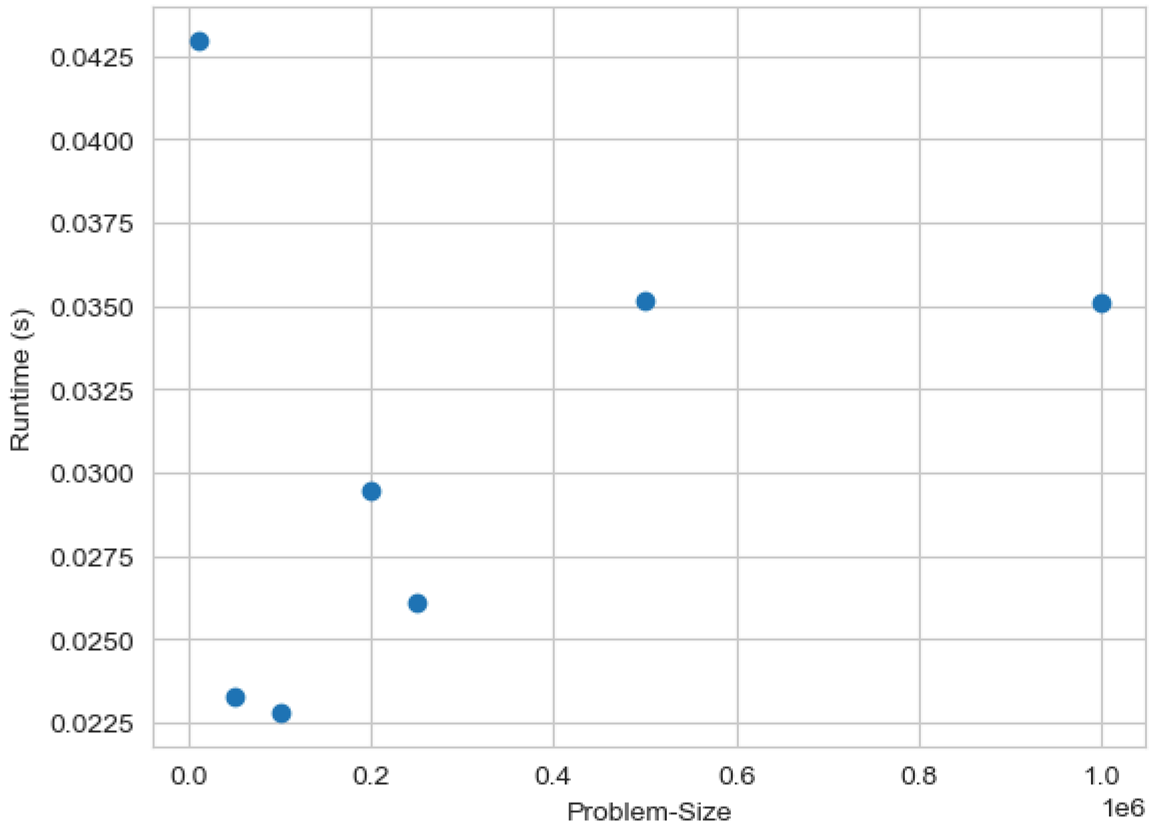
To answer this question, we will now explore how the runtime of computing the normal equation depends on the input size n and m.

In the next cell, we provide a skeleton to measure the runtime of computing the normal equation depending on the problem size n*m.

🚨 Task 3A (2 Point) 🚨

- fill in the blanks in the code below
- run the code and explore different values for m. For which values of m is the running time still below 30min?

```python
import time
sizes = []
times = []
n = 1000
for m in [10, 50, 100, 200, 250, 500, 1000]:
  # TODO create a dataset with m samples and n dimensions, we provide a helper functi
  X, y = create_random_data(m=m, n=n)

  start_time = time.monotonic()

  # enter the code you want to time here
  theta = (np.linalg.inv(X.T @ X) @ X.T) @ y

  elapsed = time.monotonic() - start_time

  # TODO what is the input size of the problem
  problem_size = m * n

  sizes.append(problem_size)
  times.append(elapsed)

plt.scatter(sizes, times)
plt.xlabel('Problem-Size')
plt.ylabel('Runtime (s)')
plt.show()
```



📢 **On Moodle** 📢

Answer the following question:

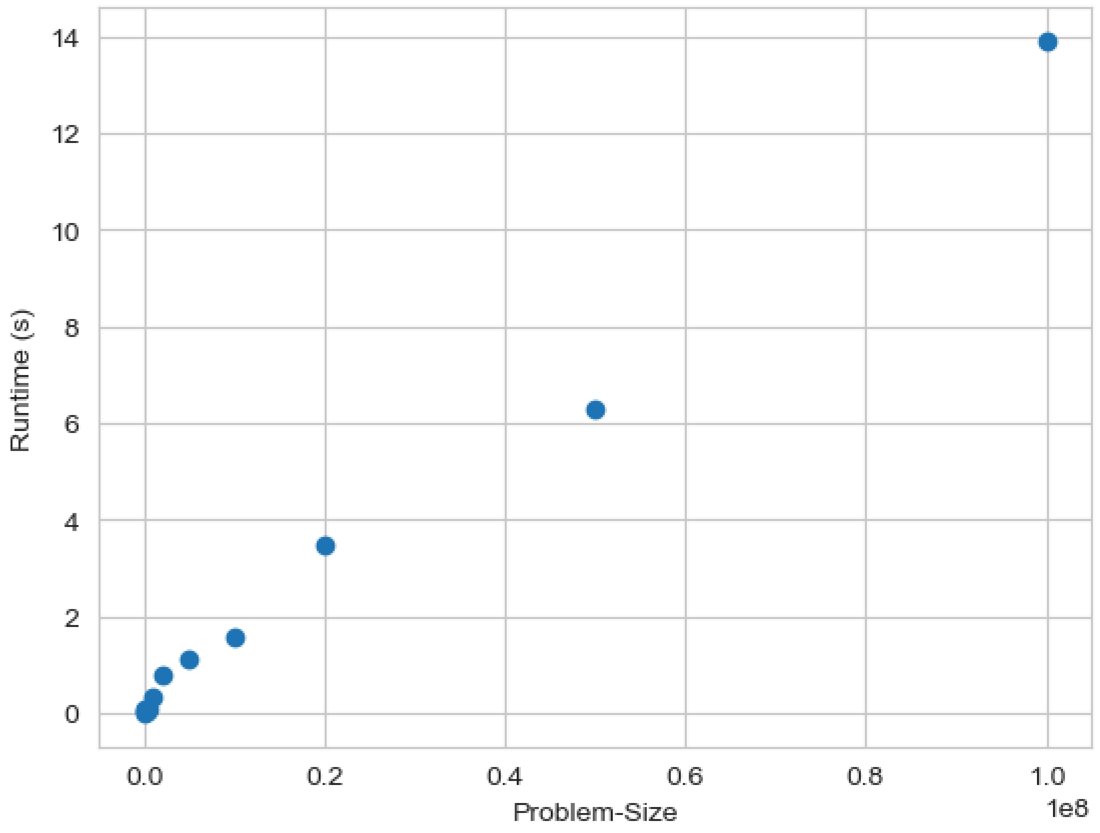
- Plot with different problem sizes
- Which problem size yields a running time of approximately 30 minutes?

🚨 Task 3B (2 Point) 🚨

We now explore the impact of the number of features, n, on the running time:

- Modify your solution to Task 3A to measure the runtime dependence on the number of features n instead of the number of samples m.
- What happens if you use large values for n, e.g. $n = 100'000$?

```python
import time
sizes = []
times = []
m = 1000
for n in [10, 50, 100, 200, 250, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000]:
  X, y = create_random_data(m=m, n=n)

  start_time = time.monotonic()

  theta = np.linalg.pinv(X) @ y

  elapsed = time.monotonic() - start_time

  problem_size = m * n

  sizes.append(problem_size)
  times.append(elapsed)

plt.scatter(sizes, times)
plt.xlabel('Problem-Size')
plt.ylabel('Runtime (s)')
plt.show()
```



📢 **On Moodle** 📢

Answer the following question:

- Upload your plot with a "very large" maximal value for n.

OPTIONAL QUESTIONS

- Do your responses to Tasks 3A and 3B change if you use `np.linalg.pinv(X)` ?
- Probably your plots for number of samples m and number of features n look very alike. Can you explain why?

Task 4: Multivariate Linear Regression using `scikit-learn`

In this task we will apply linear regression to non-synthetic data. The variable `X` is a `pandas Dataframe` containing features and `y` contains the target. Read through the description to get an idea of the different variables.

```
from sklearn.datasets import load_diabetes

data = load_diabetes(as_frame=True)
```

```python
X = data['data']
y = data['target']
description = data['DESCR']

print(description)
```

```
.. _diabetes_dataset:

Diabetes dataset
----------------

Ten baseline variables, age, sex, body mass index, average blood
pressure, and six blood serum measurements were obtained for each of n =
442 diabetes patients, as well as the response of interest, a
quantitative measure of disease progression one year after baseline.

**Data Set Characteristics:**

:Number of Instances: 442

:Number of Attributes: First 10 columns are numeric predictive values

:Target: Column 11 is a quantitative measure of disease progression one year after ba
seline

:Attribute Information:
    - age     age in years
    - sex
    - bmi     body mass index
    - bp      average blood pressure
    - s1      tc, total serum cholesterol
    - s2      ldl, low-density lipoproteins
    - s3      hdl, high-density lipoproteins
    - s4      tch, total cholesterol / HDL
    - s5      ltg, possibly log of serum triglycerides level
    - s6      glu, blood sugar level

Note: Each of these 10 feature variables have been mean centered and scaled by the st
andard deviation times the square root of `n_samples` (i.e. the sum of squares of eac
h column totals 1).

Source URL:
https://www4.stat.ncsu.edu/~boos/var.select/diabetes.html

For more information see:
Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) "Least Angl
e Regression," Annals of Statistics (with discussion), 407-499.
(https://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)
```

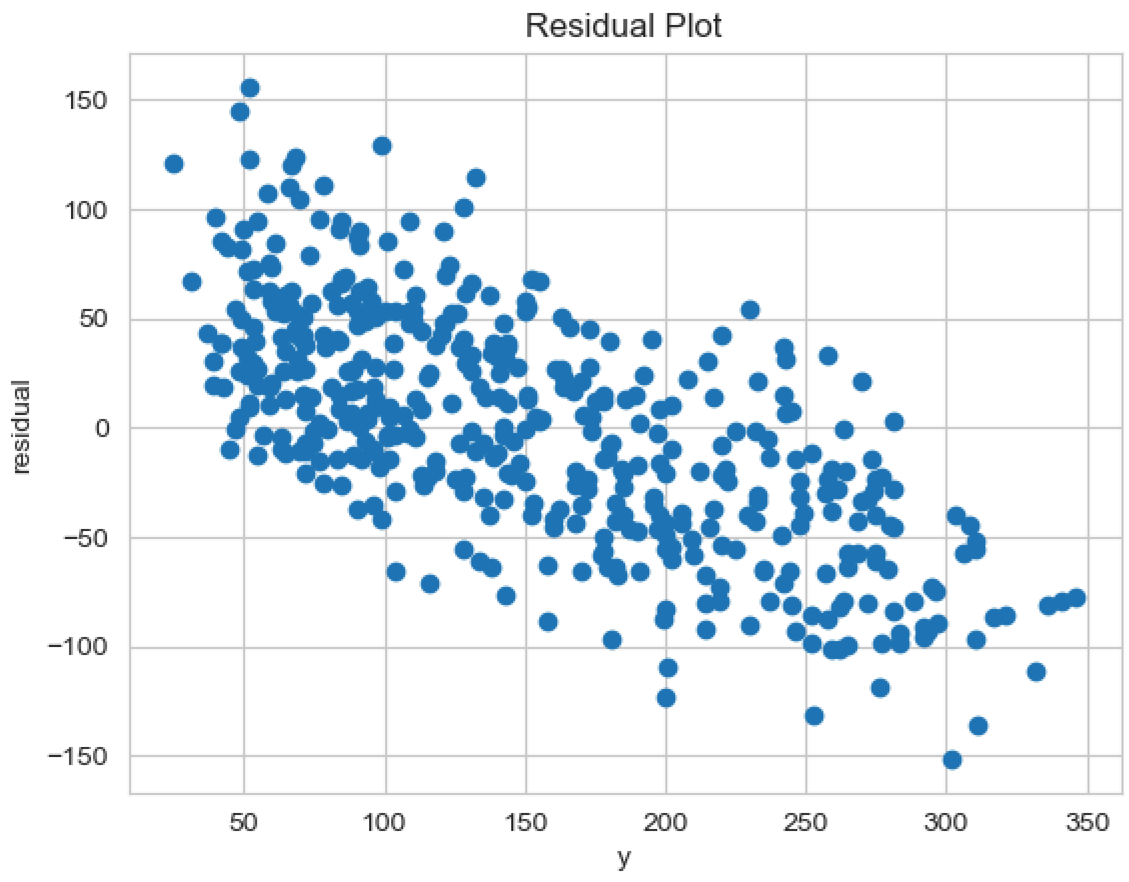
In the next cell, we will fit a linear regression model on this diabetes dataset using the implementation provided by the popular `scikit-learn` library.

Their implementation is contained in the `sklearn.linear_model.LinearRegression` class. The most important methods of any `sklearn` model are `.fit` and `.predict`. You will see them a lot in future labs.

The `.fit(X, y)` method trains the linear regression model and `.predict(X)` return the model's predictions for the data `X`.

You can see them in action in the next cell.

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
y_pred = lin_reg.predict(X)

residual_plot(y_true=y, y_pred=y_pred)
```



The numeric entries of the estimated parameter vector θ tell us something about the strength and direction of the relationship between the variables in `X` and the target `y`.

The estimated parameters θ of the linear model can be found in the `.coef_` member variable. The feature names can be found in the `.feature_names_in_` member variable. They are the same as the names of the columns of `X` and should be in the same order.

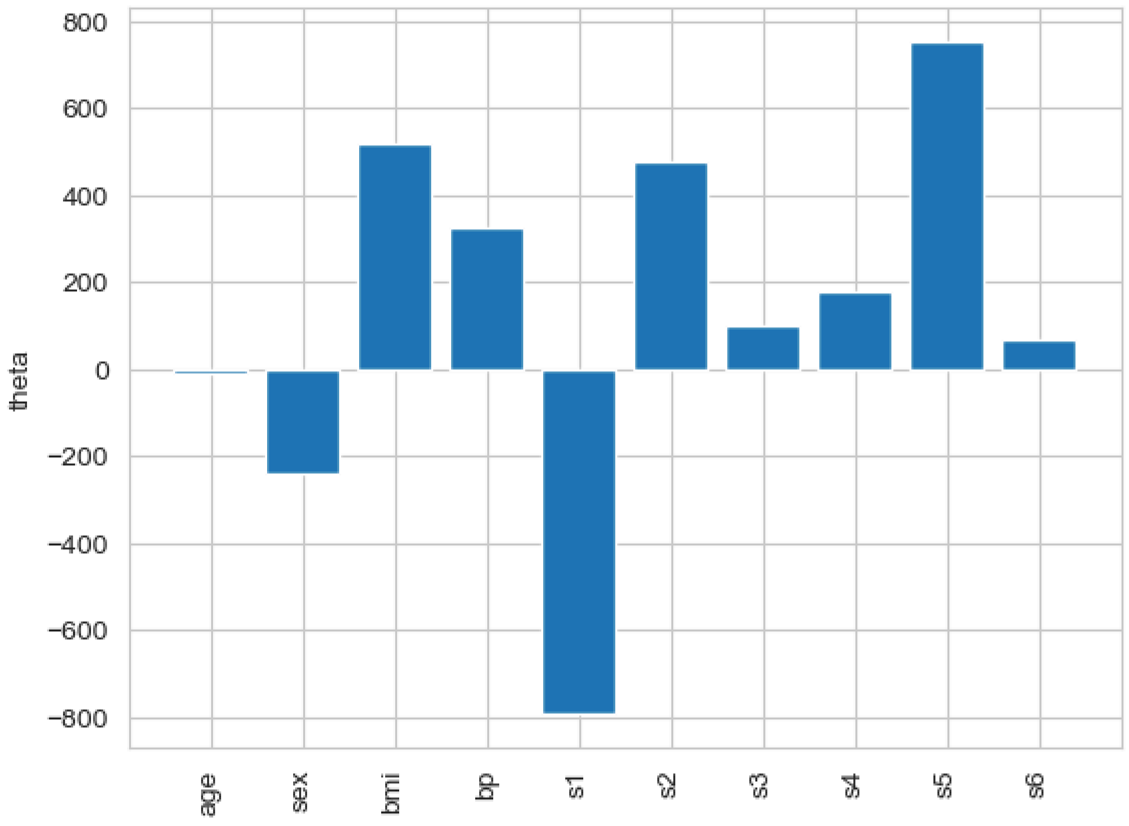
In the next code cell, we will plot the entries of θ paired with their corresponding feature name. We will also print out the same information.

🚨 Task 4A (3 Points) 🚨

Study the plot and printed output of the next cell and answer the following questions:

- Which are the 3 most influential features? bmi, s1 and s5: biggest absolute values
- How do you interpret the sign of the coefficients? A positive coefficient means that an increase in the feature value leads to an increase in the target value, while a negative coefficient indicates that an increase in the feature value leads to a decrease in the target value.
- If you had to exclude 1 feature, which one would you select and why? Exclude the feature with the smallest absolute coefficient, as it has the least influence on the target variable. -> age

```
thetas = lin_reg.coef_
feature_names = lin_reg.feature_names_in_

plt.bar(feature_names, thetas)
plt.xticks(rotation=90)
plt.ylabel("theta")
plt.show()

print()
for th, name in zip(thetas, feature_names):
  print(f"{name}\t{th}")
```



```
age      -10.009866299810273
sex      -239.8156436724229
bmi      519.8459200544602
bp       324.3846455023234
s1       -792.1756385522301
s2       476.7390210052567
s3       101.04326793803352
s4       177.0632376713459
s5       751.2736995571039
s6       67.62669218370512
```

📢 **On Moodle** 📢

- Report your answers to the 3 questions.