In [1]:

```
import numpy as np
import pandas as pd
import sklearn.linear_model as skl_lm
import matplotlib.pyplot as plt

# To get nicer plots
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('svg') # Output as svg. Else you can try png
from IPython.core.pylabtools import figsize
figsize(10, 6) # Width and hight
np.set_printoptions(precision=3);
```

2.1 Problem 1.1 using matrix multiplications

Implement the linear regression problems from Exercises 1.1(a), (b), (c), (d) and (e) in Python using matrix multiplications. A matrix

$$\mathbf{X} = \begin{bmatrix} 1 & 2 \\ 1 & 3 \end{bmatrix}$$

can be constructed with numpy as `X=np.array([[1, 2], [1, 3]])` (Make sure that `numpy` has been imported. Here it is imported as `np`). The commands for matrix multiplication and transpose in `numpy` are `@` or `np.matmul` and `.T` or `np.transpose()` respectively. A system of linear equations $\mathbf{A}x = \mathbf{b}$ can be solved using `np.linalg.solve(A,b)`. A $k \times k$ unit matrix can be constructed with `np.eye(k)`.

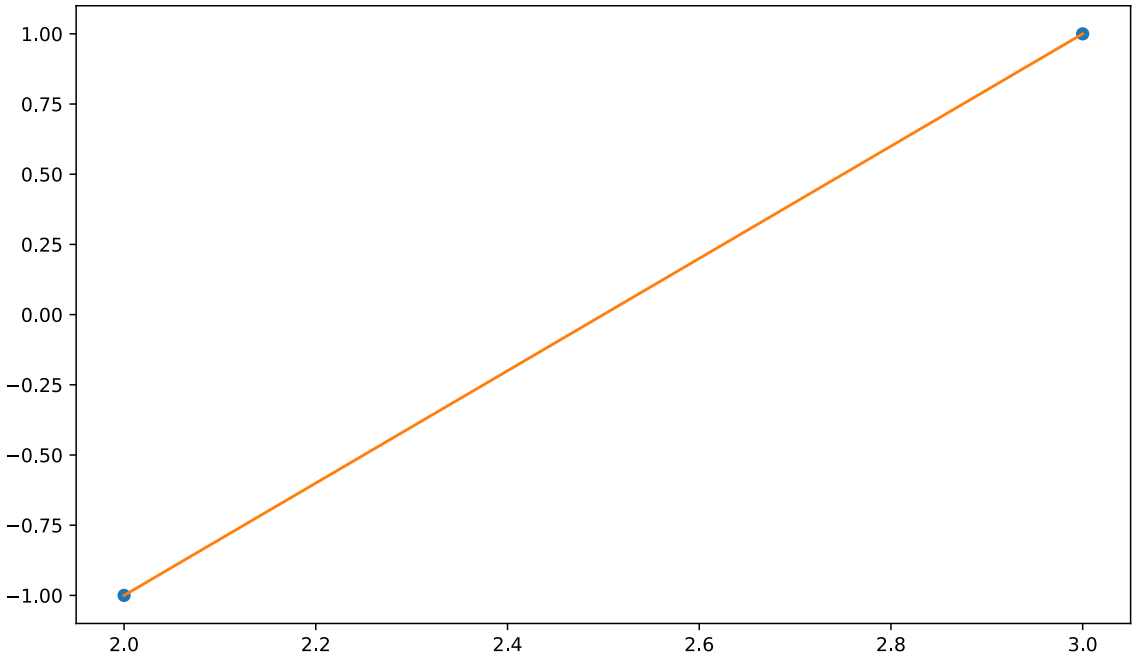
(a)

Assume that you record a scalar input x and a scalar output y. First, you record $x_1 = 2, y_1 = -1$, and thereafter $x_2 = 3, y_2 = 1$. Assume a linear regression model $y = \theta_0 + \theta_1 x + \epsilon$ and learn the parameters with maximum likelihood $\widehat{\boldsymbol{\theta}}$ with the assumption $\epsilon \sim \mathcal{N}(0, \sigma_\epsilon^2)$. Use the model to predict the output for the test input $x_\star = 4$, and plot the data and the model.

In [2]:

```
# Construct the data matrix
X = np.array([[1, 2], [1, 3]])
y = np.array([-1, 1])

# Solve the normal equations
theta_hat = np.linalg.solve(X.T@X, X.T@y)

# Print the solution
print(f"theta_hat = {theta_hat}")

# Compute prediction for x = 4
y_hat = theta_hat@np.array([1,4])

# Print the prediction
print(f"y_hat = {y_hat:.3f}")

# Plot the data and the model
plt.plot(X[:,1], y, 'o')
prediction = X@theta_hat
plt.plot(X[:,1], prediction);
```

```
theta_hat = [-5.  2.]
y_hat = 3.000
```



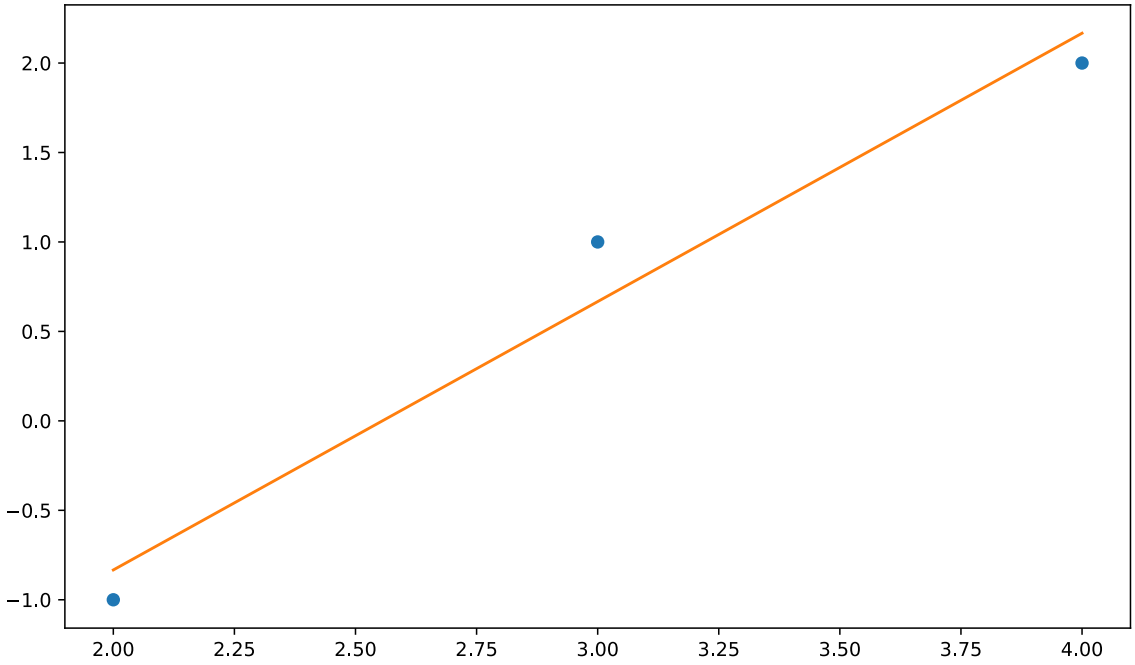
(b)

Now, assume you have made a third observation $y_3 = 2$ for $x_3 = 4$ (is that what you predicted in (a)?). Update the parameters $\widehat{\boldsymbol{\theta}}$ to all 3 data samples, add the new model to the plot (together with the new data point) and find the prediction for $x_\star = 5$.

In [3]:

```
# Construct the data matrices
X = np.array([[1, 2], [1, 3], [1, 4]])
y = np.array([-1, 1 ,2])

# Compute the solution using the normal equation
theta_hat = np.linalg.solve(X.T@X,X.T@y)

# Print the solution
print(f"theta_hat = {theta_hat}")

# Compute prediction for x = 5
y_hat = theta_hat@np.array([1,5])

# Print the prediction
print(f"y_hat =  {y_hat:.3f}")

# Plot the data and the model
plt.plot(X[:,1], y, 'o')
prediction = X@theta_hat
plt.plot(X[:,1], prediction);
```

```
theta_hat = [-3.833  1.5  ]
y_hat =  3.667
```



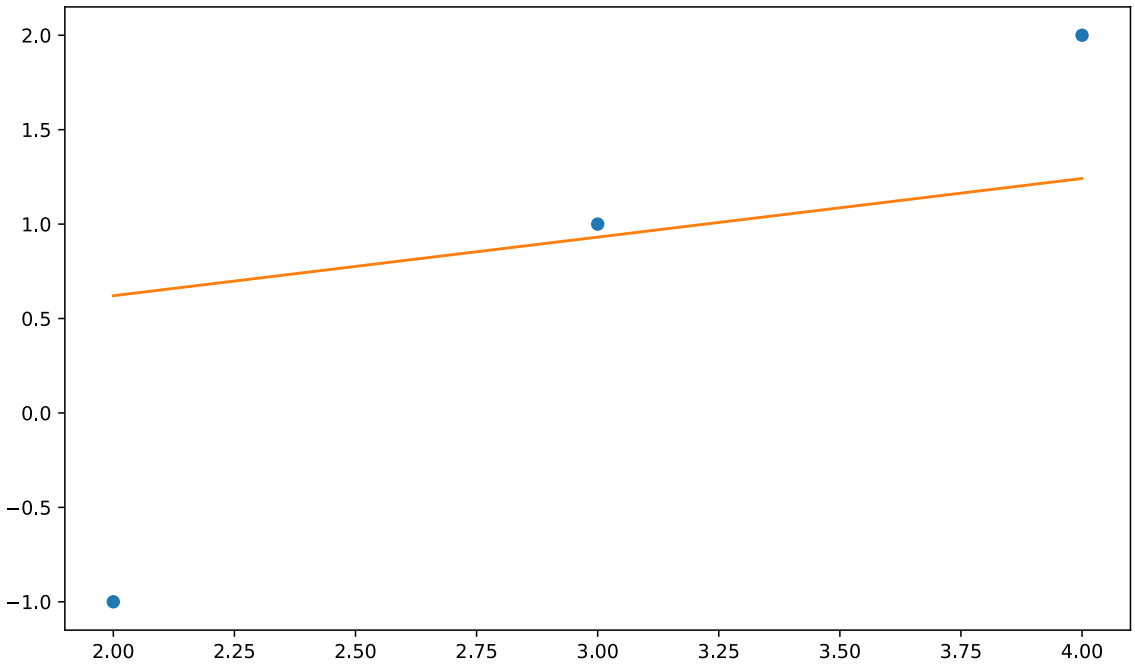
(c)

Repeat (b), but this time using a model without intercept term, i.e., $y = \theta_1 x + \epsilon$.

In [4]:

```
# Consctruct the data matrices
# Reshape the array to 2-dim so we can use np.linalg.solve()
X = np.array([2, 3, 4]).reshape(-1, 1)
y = np.array([-1, 1, 2])

# Compute the solution
theta_hat = np.linalg.solve(X.T@X,X.T@y)

# Print the solution
print(f"theta_hat = {theta_hat}")

# Compute prediction for x = 5
y_hat = theta_hat@np.array([5])

# Print the prediction
print(f"y_hat = {y_hat:.3f}")

# Plot the data and the model
plt.plot(X, y, 'o')
prediction = X@theta_hat
plt.plot(X, prediction);
```

```
theta_hat = [0.31]
y_hat = 1.552
```



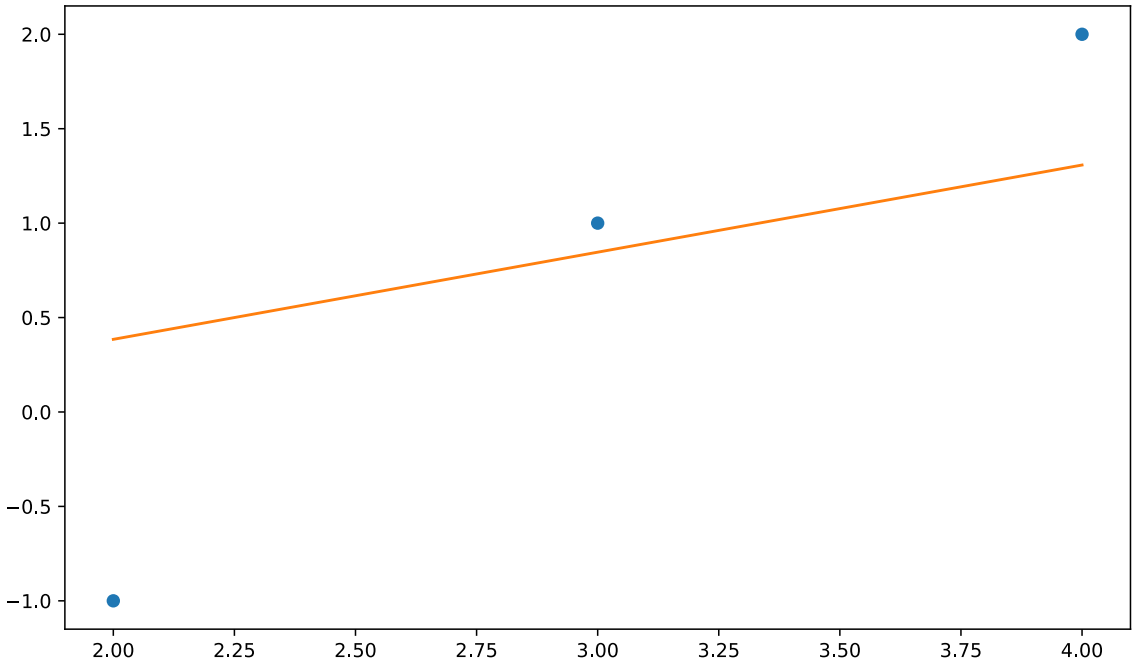
(d)

Repeat (b), but this time using Ridge Regression with $\gamma = 1$ instead.

In [5]:

```
# Use the solution to the Ridge Regression problem
# Construct the data matrices
X = np.array([[1, 2], [1, 3], [1, 4]])
y = np.array([-1, 1, 2])
lambda_ = 1

# Compute the solution
theta_hat = np.linalg.solve(X.T@X + np.eye(2),X.T@y)

# Print the solution
print(f"theta_hat = {theta_hat}")

# Compute prediction for x = 5
y_hat = theta_hat@np.array([1,5])

# Print the prediction
print(f"y_hat = {y_hat:.3f}")

# Plot the data and the model
plt.plot(X[:,1], y, 'o')
prediction = X@theta_hat
plt.plot(X[:,1], prediction);
```

```
theta_hat = [-0.538  0.462]
y_hat = 1.769
```



(e)

You realize that there are actually *two* output variables in the problem you are studying. In total, you have made the following observations:

sample	input x	first output y_1	second output y_2
(1)	2	-1	0
(2)	3	1	2
(3)	4	2	-1

You want to model this as a linear regression with multidimensional outputs (without regularization), i.e.,

$$y_1 = \theta_{01} + \theta_{11} x + \epsilon_1$$
$$y_2 = \theta_{02} + \theta_{12} x + \epsilon_2$$

By introducing, for the general case of p inputs and q outputs, the matrices

$$\underbrace{\begin{bmatrix} y_{11} & \cdots & y_{1q} \\ y_{21} & \cdots & y_{2q} \\ \vdots & & \vdots \\ y_{n1} & \cdots & y_{nq} \end{bmatrix}}_{\mathbf{Y}} = \underbrace{\begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} \theta_{01} & \theta_{02} & \cdots & \theta_{0q} \\ \theta_{11} & \theta_{12} & \cdots & \theta_{1q} \\ \theta_{21} & \theta_{22} & \cdots & \theta_{2q} \\ \vdots & \vdots & & \vdots \\ \theta_{p1} & \theta_{p2} & \cdots & \theta_{pq} \end{bmatrix}}_{\mathbf{\Theta}} + \boldsymbol{\epsilon}$$

try to make an educated guess how the normal equations can be generalized to the multidemsional output case. (A more thorough derivation is found in problem 1.5). Use your findings to compute the least square solution $\widehat{\mathbf{\Theta}}$ to the problem now including both the first output y_1 and the second output y_2.

In [6]:

```
# Construct the data matrices
X = np.array([[1, 2], [1, 3], [1, 4]])
Y = np.array([[-1, 0], [1, 2], [2, -1]])

# Compute the solution
Theta_hat = np.linalg.solve(X.T@X,X.T@Y)

# Print the solution
print(f"Theta_hat =\n{Theta_hat}")
```

```
Theta_hat =
[[-3.833  1.833]
 [ 1.5   -0.5  ]]
```

2.2 Problem 1.1 using the linear_model.LinearRegression() command

Implement the linear regression problem from Exercises 1.1(b) and (c) using the command `LinearRegression()` from `sklearn.linear_model` .

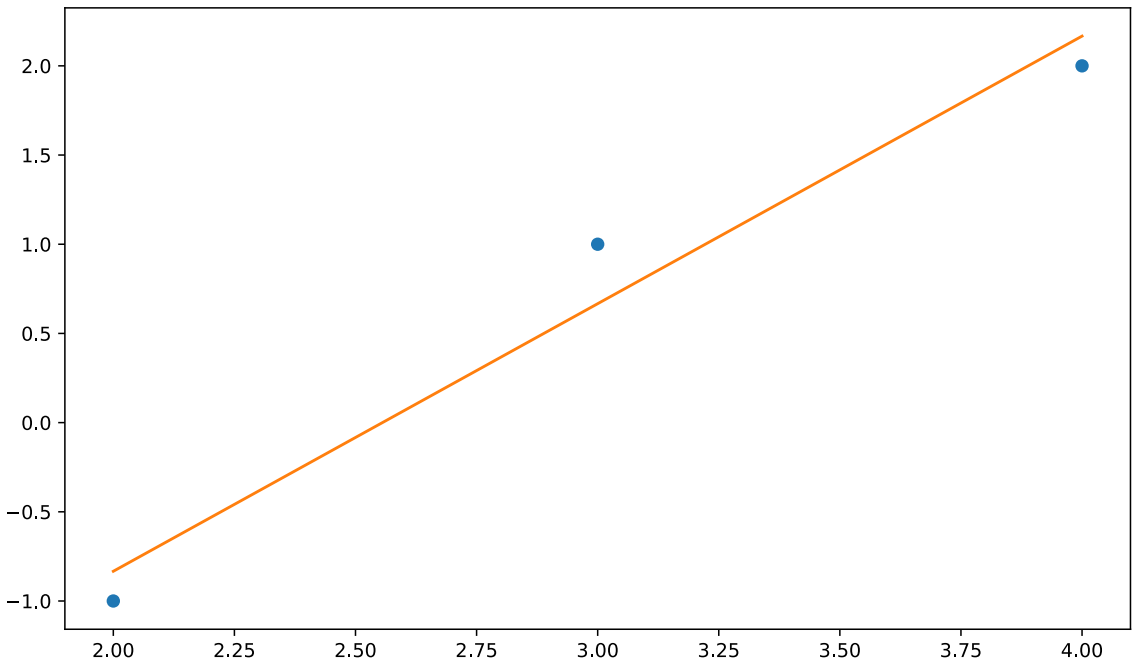
(b)

See above.

In [7]:

```
X = np.array([2, 3, 4]).reshape(-1, 1)
y = np.array([-1, 1 , 2])

# Learn the model using the skl_lm() command
model = skl_lm.LinearRegression()
model.fit(X, y)

# Print the solution
print(f'The coeficient for X is : {model.coef_[0]:.3f}')
print(f'The offset is: {model.intercept_:.3f}')

# Plot the data and the model
plt.plot(X, y, 'o')
prediction = model.predict(X)
plt.plot(X, prediction);
```

```
The coeficient for X is : 1.500
The offset is: -3.833
```



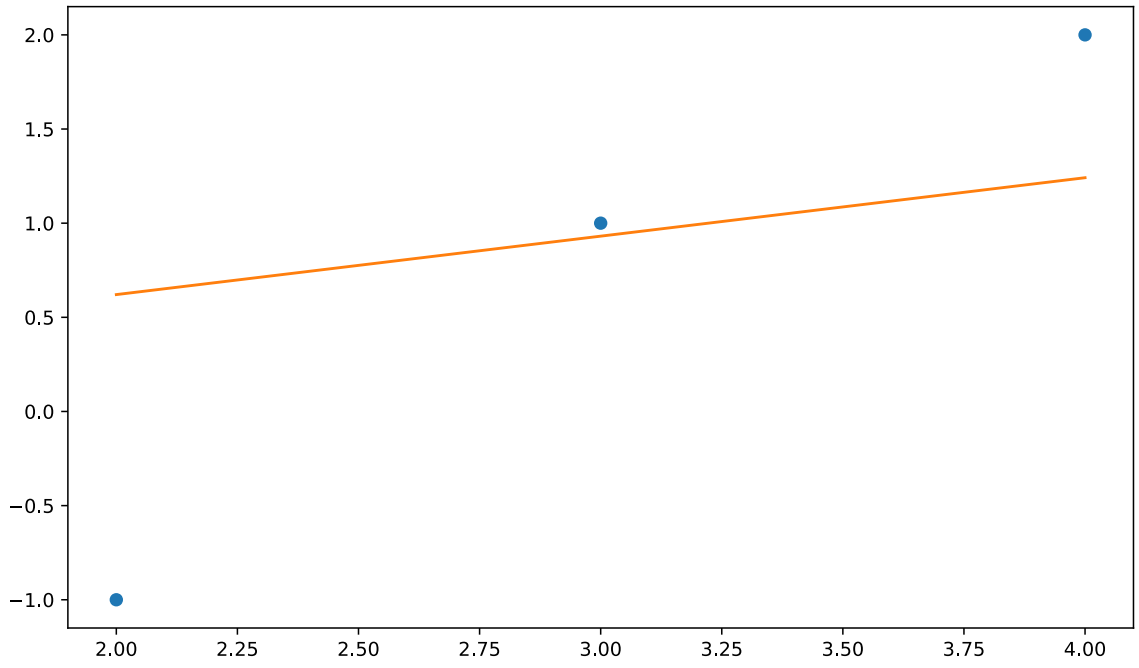
(c)

See above.

In [8]:

```
# Learn the model using the skl_lm() command without intercept
model = skl_lm.LinearRegression(fit_intercept=False)
model.fit(X, y)

# Print the solution
print(f'The coeficient for X is : {model.coef_[0]:.3f}')
print(f'The offset is: {model.intercept_:.3f}')

# Plot the data and the model
plt.plot(X, y, 'o')
prediction = model.predict(X)
plt.plot(X, prediction);
```

```
The coeficient for X is : 0.310
The offset is: 0.000
```



2.3 The Auto data set

(a)

Load the dataset `'data/auto.csv'`. Familiarize yourself with the dataset using `auto.info()`. The dataset:

Description: Gas mileage, horsepower, and other information for 392 vehicles.
Format: A data frame with 392 observations on the following 9 variables.

- `mpg` : miles per gallon
- `cylinders` : Number of cylinders between 4 and 8
- `displacement` : Engine displacement (cu. inches)
- `horsepower` : Engine horsepower
- `weight` : Vehicle weight (lbs.)
- `acceleration` : Time to accelerate from 0 to 60 mph (sec.)
- `year` : Model year (modulo 100)
- `origin` : Origin of car (1. American, 2. European, 3. Japanese)
- `name` : Vehicle name

  The orginal data contained 408 observations but 16 observations with missing values were removed.

In [9]:

```
# Load library
# The null values are '?' in the dataset. `na_values="?"` recognize the null values.
# There are null values that will mess up the computation. Easier to drop them by `drop
na()`.

# url = 'data/auto.csv'
url = 'https://uu-sml.github.io/course-sml-public/data/auto.csv'

auto = pd.read_csv(url, na_values='?').dropna()
```

In [10]:

```
auto.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 392 entries, 0 to 396
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   mpg           392 non-null    float64
 1   cylinders     392 non-null    int64
 2   displacement  392 non-null    float64
 3   horsepower    392 non-null    float64
 4   weight        392 non-null    int64
 5   acceleration  392 non-null    float64
 6   year          392 non-null    int64
 7   origin        392 non-null    int64
 8   name          392 non-null    object
dtypes: float64(4), int64(4), object(1)
memory usage: 30.6+ KB
```

(b)

Divide the data set randomly into two approximately equally sized subsets, `train` and `test` by generating the random indices using `np.random.choice()` .

In [11]:

```
print(f"auto.shape: {auto.shape}") #(No. of rows, No. of columns)

# Set seed to get reproducible results
np.random.seed(1)

trainI = np.random.choice(auto.shape[0], size=200, replace=False)
trainIndex = auto.index.isin(trainI)
train = auto.iloc[trainIndex]
test = auto.iloc[~trainIndex]
```

```
auto.shape: (392, 9)
```

(c)

Perform linear regression with `mpg` as the output and all other variables except name as input. How well (in terms of root-mean-square-error) does the model perform on test data and training data, respectively?

In [12]:

```
# Ignore RuntimeWarning: internal gelsd driver lwork query error. Harmless

# Linear regression
model = skl_lm.LinearRegression(fit_intercept = True)      # Add an offset
X_train = train[['cylinders', 'displacement', 'horsepower', 'weight',\
                 'acceleration', 'year', 'origin']]
Y_train = train['mpg']
model.fit(X_train, Y_train)
print(model)

# Evaluate on training data
train_predict = model.predict(X_train)
train_RMSE = np.sqrt(np.mean((train_predict - train.mpg)**2))
print(f'Train RMSE:\t{train_RMSE:.4f}')

## Evaluate on test data
X_test = test[['cylinders', 'displacement', 'horsepower', 'weight',\
               'acceleration', 'year', 'origin']]
test_predict = model.predict(X_test)
test_RMSE = np.sqrt(np.mean((test_predict - test.mpg)**2))
print(f'Test RMSE:\t{test_RMSE:.4f}')
```

```
LinearRegression()
Train RMSE:     3.2821
Test RMSE:      3.3370
```

(d)

Now, consider the input variable `origin`. What do the different numbers represent? By running `auto.origin.sample(30)` we see the 30 samples of the variable and that the input variables is quantitative. Does it really makes sense to treat it as a quantitative input? Use `pd.get_dummies()` to split it into dummy variables and do the linear regression again.

In [13]:

```
# Examples of the origin variable
print('auto origin:')
print(auto.origin.sample(30).tolist(), '\n')

X_train = pd.get_dummies(train, columns=['origin'])
print('X after transformation (origin has been split in three dummy variables):')
print(X_train.head(), '\n')
# Pick out the input variables
X_train = X_train[['cylinders', 'displacement', 'horsepower', 'weight',\
                   'acceleration', 'year', 'origin_1', 'origin_2', 'origin_3']]

X_test = pd.get_dummies(test, columns=['origin'])
X_test = X_test[['cylinders', 'displacement', 'horsepower', 'weight',\
                 'acceleration', 'year', 'origin_1', 'origin_2', 'origin_3']]

# look at how sci-kit learn transforms the qualitative input
print(X_train.sample(5), '\n')

# Repeat c) create and evaluate the model now using encoded categorical data
model1 = skl_lm.LinearRegression()
model1.fit(X_train, Y_train)
print(model1,'\n')

# Evaluate on training data
train_predict = model1.predict(X_train)
train_RMSE = np.sqrt(np.mean((train_predict - train.mpg)**2))
print(f'Train RMSE:\t{train_RMSE:.4f}')

## Evaluate on test data
test_predict = model1.predict(X_test)
test_RMSE = np.sqrt(np.mean((test_predict - test.mpg)**2))
print(f'Test RMSE:\t{test_RMSE:.4f}')
```

```
auto origin:
[1, 1, 1, 1, 1, 2, 1, 3, 1, 2, 3, 1, 3, 2, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 3]

X after transformation (origin has been split in three dummy variables):
    mpg  cylinders  displacement  horsepower  weight  acceleration  year
\
0  18.0          8         307.0       130.0    3504          12.0    70
4  17.0          8         302.0       140.0    3449          10.5    70
5  15.0          8         429.0       198.0    4341          10.0    70
6  14.0          8         454.0       220.0    4354           9.0    70
8  14.0          8         455.0       225.0    4425          10.0    70

                       name  origin_1  origin_2  origin_3
0  chevrolet chevelle malibu         1         0         0
4               ford torino         1         0         0
5          ford galaxie 500         1         0         0
6          chevrolet impala         1         0         0
8          pontiac catalina         1         0         0

     cylinders  displacement  horsepower  weight  acceleration  year  \
98           6         250.0       100.0    3278          18.0    73
285          8         305.0       130.0    3840          15.4    79
236          4         140.0        89.0    2755          15.8    77
213          8         350.0       145.0    4055          12.0    76
373          4         140.0        92.0    2865          16.4    82

     origin_1  origin_2  origin_3
98          1         0         0
285         1         0         0
236         1         0         0
213         1         0         0
373         1         0         0

LinearRegression()

Train RMSE:     3.2630
Test RMSE:      3.3108
```

(e)

Try obtain a better RMSE on test data by removing some inputs (explore what happens if you remove, e.g, `year`, `weight` and `acceleration`)

In [14]:

```
# First write a funcion that takes the prediction model, training and test data
# and computes RMSE to simplify the process

def computeRMSE(model, X, Y):
    Y_predict = model.predict(X)
    RMSE = np.sqrt(np.mean((Y_predict - Y)**2))
    return RMSE

# The following function streamlines the procedure of testing with dropping
# different varibles. It is optional. But if you want to skip the function
# keep in mind that when you declare e.g. X=X.drop(columns=column_name),
# you are manipulating the original X
def RMSE_with_drop_col(model, X, Y, X_test, Y_test, drop_col):
    # drop_col takes a list of string or strings
    print(f'Results without the variable {drop_col}:')
    X = X.drop(columns=drop_col)
    model.fit(X, Y)
    train_RMSE = computeRMSE(model, X, Y)
    print(f'Train RMSE: \t{train_RMSE:.4f}')

    X_test = X_test.drop(columns=drop_col)
    test_RMSE = computeRMSE(model, X_test, Y_test)
    print(f'Test RMSE:\t{test_RMSE:.4f}')
    print()

# Test output (has not been declared)
Y_test = test.mpg

# Remove weight
model2 = skl_lm.LinearRegression()
RMSE_with_drop_col(model2, X_train, Y_train, X_test, Y_test,\
                   ['weight', 'acceleration'])

# Remove year
model3 = skl_lm.LinearRegression()
RMSE_with_drop_col(model3, X_train, Y_train, X_test, Y_test, ['year'])

# Remove acceleration
model4 = skl_lm.LinearRegression()
RMSE_with_drop_col(model4, X_train, Y_train, X_test, Y_test, ['acceleration'])
```

```
Results without the variable ['weight', 'acceleration']:
Train RMSE:     3.7496
Test RMSE:      3.8356

Results without the variable ['year']:
Train RMSE:     4.1714
Test RMSE:      4.0975

Results without the variable ['acceleration']:
Train RMSE:     3.2650
Test RMSE:      3.3122
```

(f)

Try to obtain a better RMSE on test data by adding some transformations of inputs, such as $log(x)$, $\sqrt{x}$, $x_1 x_2$ etc.

In [15]:

```
# A small function to simplify the process
def RMSE_with_cols(model, X, Y, cols):
    print(f'RMSE with the variables {cols}')
    X = X[cols]
    model.fit(X, Y)
    RMSE = computeRMSE(model, X, Y)
    print(f'RMSE\t{RMSE:.4f}')

# horsepower*acceleration
print('comp = horsepower * acceleration')
model = skl_lm.LinearRegression()
X_train_copy = X_train.copy()     # A hard copy to avoid manipulating the original X
X_train_copy['comp'] = X_train_copy.horsepower * X_train_copy.acceleration
cols = ['cylinders', 'displacement','comp', 'origin_1', 'origin_2', 'origin_3']
RMSE_with_cols(model, X_train_copy, Y_train, cols)

print()

# sqrt(horsepower) and weight^2
print('sqrt(horsepower) and weight^2')
model = skl_lm.LinearRegression()
X_train_copy = X_train.copy()
X_train_copy['sqrt_horsepower'] = np.sqrt(X_train_copy.horsepower)
X_train_copy['weight_sqr'] = X_train_copy.weight**2
cols = ['cylinders', 'displacement','sqrt_horsepower', 'weight_sqr',\
        'origin_1', 'origin_2', 'origin_3']
RMSE_with_cols(model, X_train_copy, Y_train, cols)
```

```
comp = horsepower * acceleration
RMSE with the variables ['cylinders', 'displacement', 'comp', 'origin_1',
'origin_2', 'origin_3']
RMSE    4.1873

sqrt(horsepower) and weight^2
RMSE with the variables ['cylinders', 'displacement', 'sqrt_horsepower',
'weight_sqr', 'origin_1', 'origin_2', 'origin_3']
RMSE    4.1824
```

2.4 Nonlinear transformations of input variables

In [16]:

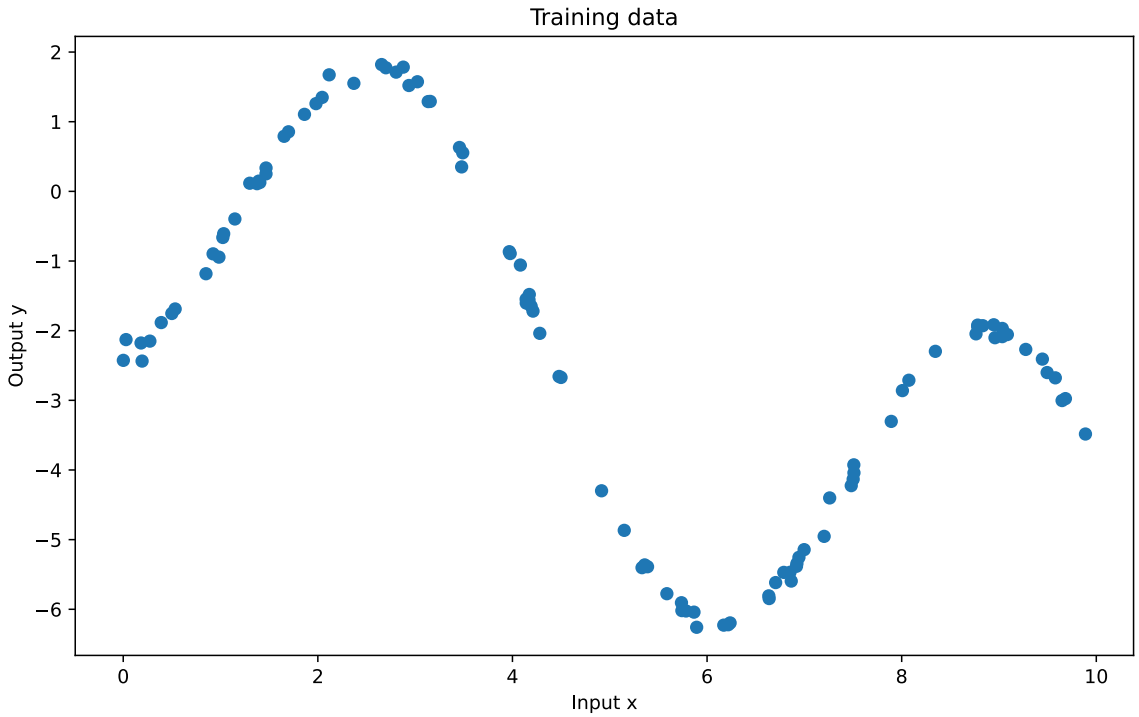
```
#Start by running the following code to generate your training data
np.random.seed(1)
x_train = np.random.uniform(0, 10, 100)
y_train = .4 \
          - .6 * x_train \
          + 3. * np.sin(x_train - 1.2) \
          + np.random.normal(0, 0.1, 100)
```

(a)

Plot the training output `y_train` versus the training input `x_train`.

In [17]:

```
# Plot
plt.plot(x_train, y_train, 'o')
plt.title('Training data')
plt.xlabel('Input x')
plt.ylabel('Output y');
```



(b)

Learn a model on the form

$$y = a + bx + c\sin(x + \phi) + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, 0, 1^2) \qquad (2.1)$$

where all parameters a, b, c and ϕ are to be learned from the training data `x_train` and `y_train`. Refrain from using the `linear_model()` command, but implement the normal equations yourself as in problem 2.1. Hint: Even though (2.1) is not a linear regression model, you can use the fact that $c\sin(x + \phi) = c\cos(\phi)\sin(x) + c\sin(\phi)\cos(x)$ to transform it into one.

In [18]:

```
X_train = np.column_stack([np.ones(100), x_train,\
                           np.cos(x_train), np.sin(x_train)])
#y_train = np.array(y_train).reshape(-1, 1)
theta_hat = np.linalg.solve(X_train.T@X_train, X_train.T@y_train)
print(f'theta_hat ={theta_hat}')
```

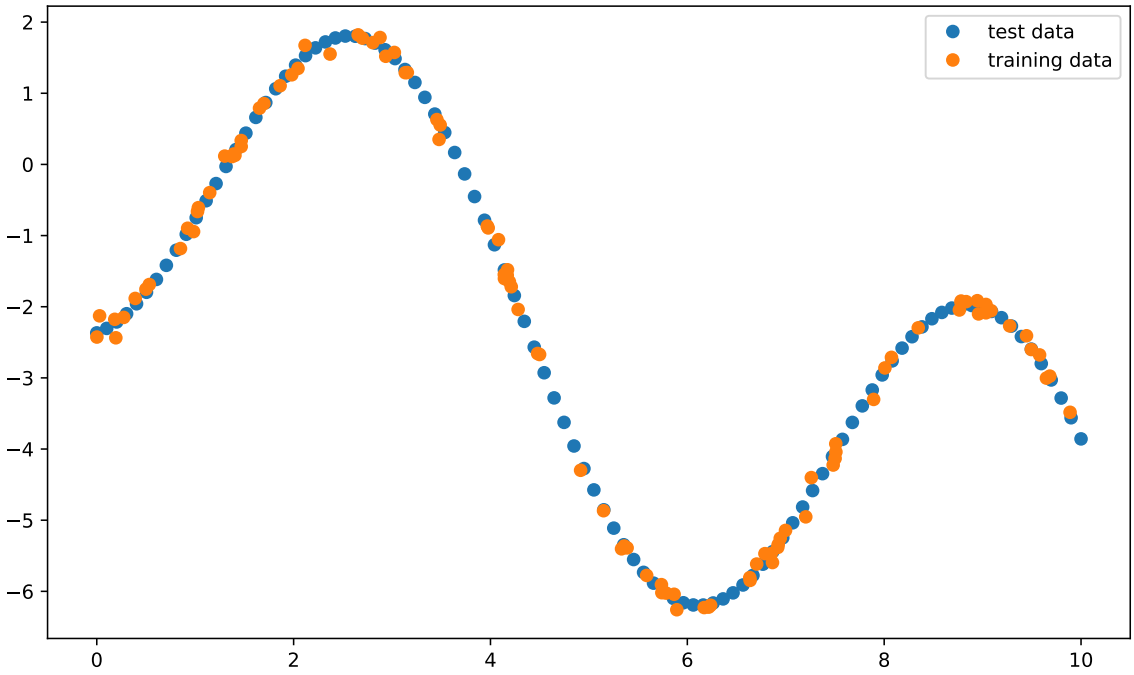
```
theta_hat =[ 0.421 -0.603 -2.789  1.088]
```

(c)

Construct 100 test inputs `x_test` in the span from 0 to 10 by using the `np.linspace()` function. Predict the outputs corresponding to these inputs and plot them together with the training data.

In [19]:

```
# c) Do prediction
x_test = np.linspace(0, 10, 100)
X_test = np.column_stack([np.ones(100), x_test, np.cos(x_test), np.sin(x_test)])
y_test_hat = X_test@theta_hat
plt.plot(x_test, y_test_hat, 'o', label='test data')
plt.plot(x_train, y_train, 'o', label='training data')
plt.legend();
```



(d)

Do a least squares fit by instead using the `linear_model()` function in `Python` . Check that you get the same estimates as in (b).

In [20]:

```
model = skl_lm.LinearRegression()
model.fit(X_train[:,1:], y_train)
prediction = model.predict(X_test[:,1:])

# Compute root mean squared error of predictions and learned theta
print(f"RMS y_test_hat:\t{np.sqrt(np.mean(np.square(prediction-y_test_hat)))}")
print(f"RMS theta_hat:\t{np.sqrt(np.mean(np.square(theta_hat - np.hstack((model.interce
pt_, model.coef_)))))}")
```

```
RMS y_test_hat: 2.799991492504061e-15
RMS theta_hat:  2.039043296240461e-15
```

2.5 Regularization

In this exercise we will apply Ridge regression and Lasso for fitting a polynomial to a scalar data set. We will have a setting where we first generate synthetic training data from

$$y = x^3 + 2x^2 + 6 + \epsilon, \qquad (2.2)$$

and later try to learn model for the data.

(a)

Write a function that implements the polynomial (2.2), i.e., takes x as argument and returns $x^3 + 2x^2 + 6$.

In [21]:

```
def f(x):
    return x**3 + 2*x**2 + 6
```

(b)

Use `np.random.seed()` to set the random seed. Use the function `np.linspace()` to construct a vector `x` with `n = 12` elements equally spaced from -2.3 to 1. Then use your function from (a) to construct a vector $\mathbf{y} = [y_1, \ldots, y_n]^T$ with 12 elements, where $y = x^3 + 2x^2 + 6 + \epsilon$, with $\epsilon \sim \mathcal{N}(0, 1^2)$. This is our training data.

In [22]:

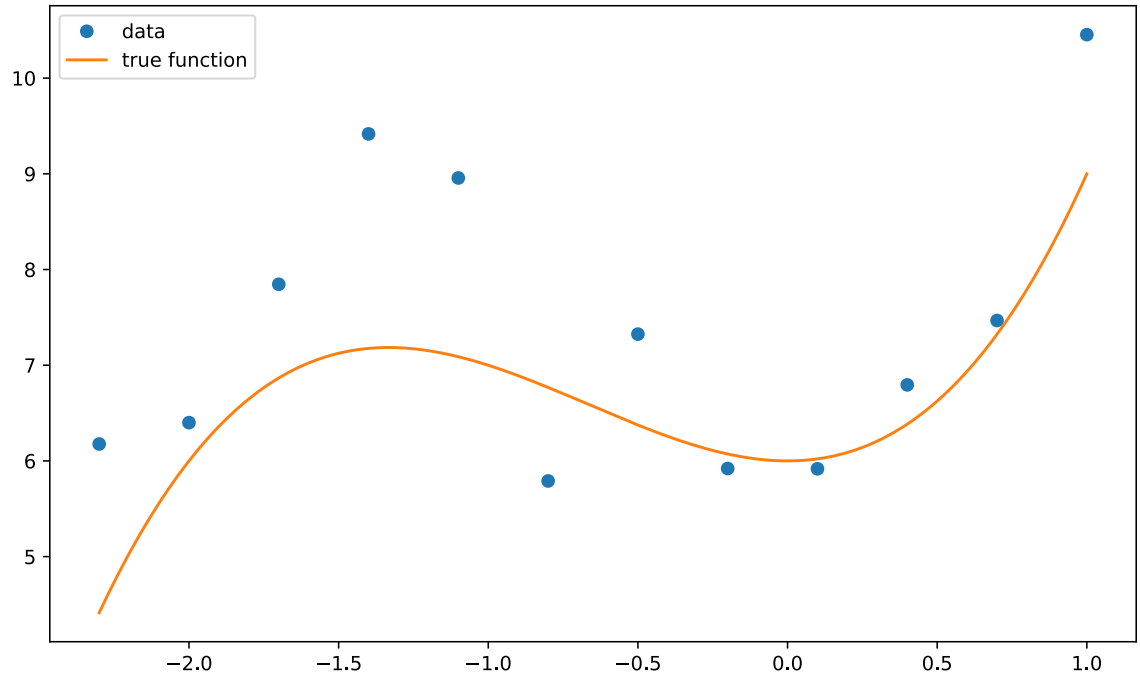
```
np.random.seed(0)
x_train = np.linspace(-2.3, 1, 12)
y_train = f(x_train) + np.random.normal(0, 1, 12)
```

(c)

Plot the training data $\mathcal{T} = \{x_i, y_i\}_{i=1}^{12}$ together with the "true" function.

In [23]:

```
x_test = np.linspace(-2.3, 1, 400)
y_test = f(x_test)

plt.plot(x_train, y_train, 'o', label='data')
plt.plot(x_test, y_test, label='true function')
plt.legend()
plt.show()
```

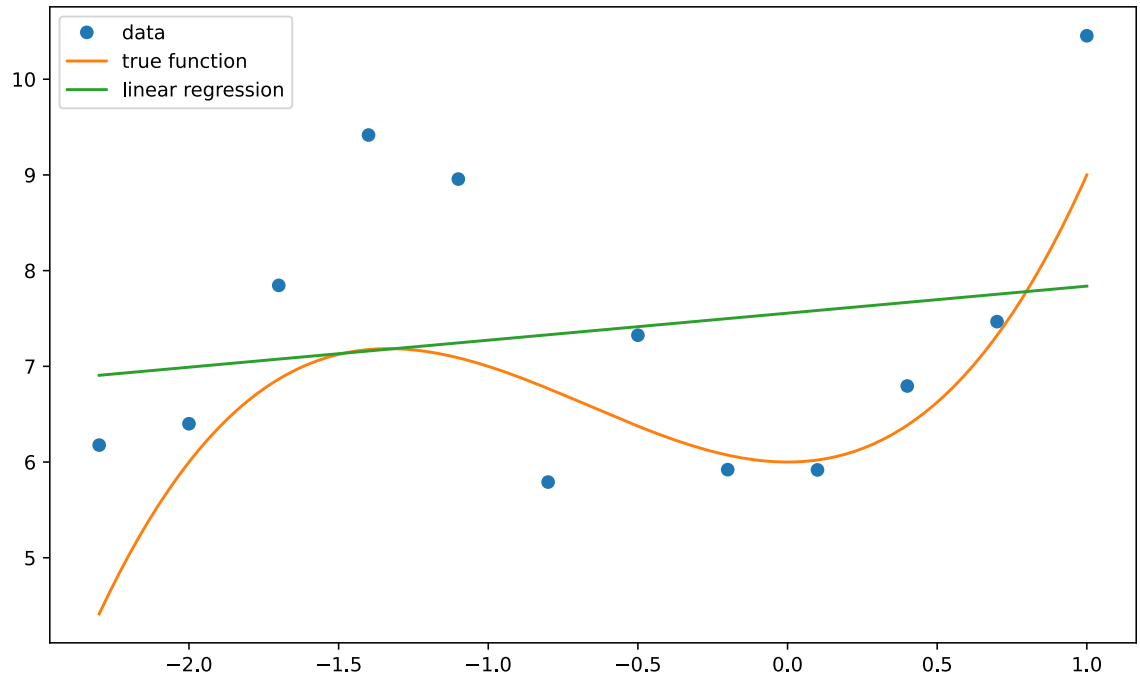


(d)

Fit a straight line to the data with y as output and x as input and plot the predicted output $\hat{y}_\star$ for densely spaced $x_\star$ values between -2.3 and 1. Plot these predictions in the same plot window.

In [24]:

```
# Linear regression
model = skl_lm.LinearRegression()
model.fit(x_train.reshape(-1,1), y_train)
prediction = model.predict(x_test.reshape(-1,1))

# Plots
plt.plot(x_train, y_train, 'o', label='data')
plt.plot(x_test, y_test, label='true function')
plt.plot(x_test, prediction, label='linear regression')

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

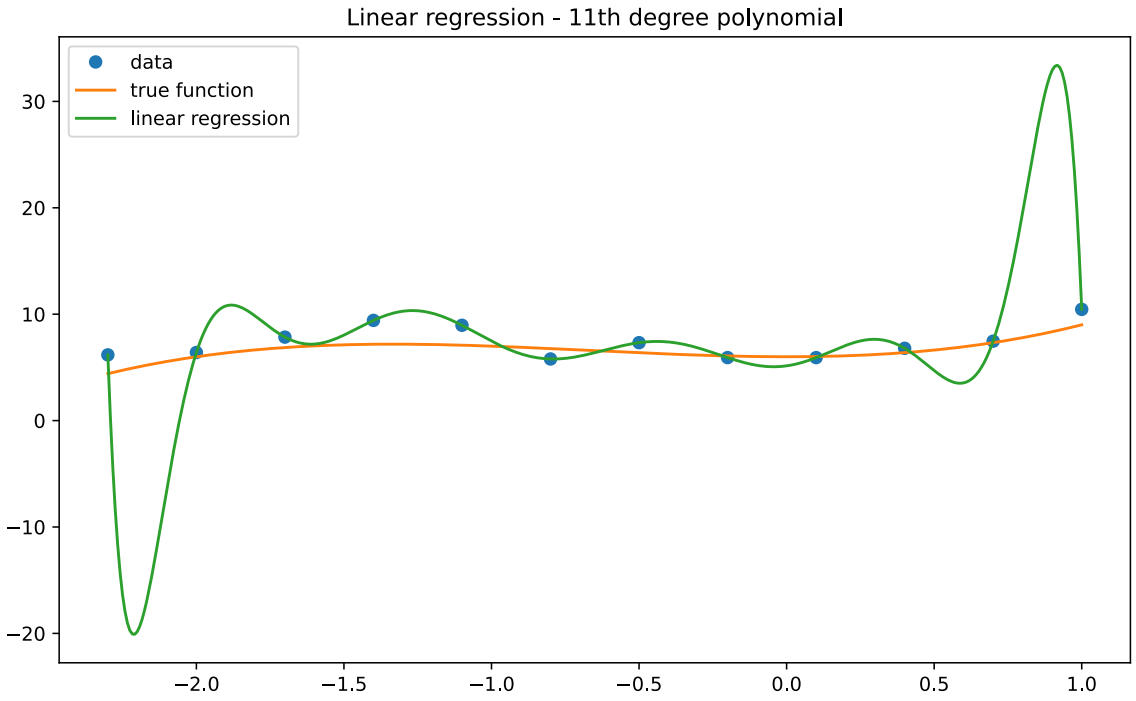


(e)

Fit an 11th degree polynomial to the data with linear regression. Plot the corresponding predictions.

In [25]:

```
# x^[0, 1, 2,..., 11]
x_train_ext = np.power(x_train.reshape(-1,1), np.arange(12))
x_test_ext = np.power(x_test.reshape(-1,1), np.arange(12))

model = skl_lm.LinearRegression()
model.fit(x_train_ext, y_train)
prediction = model.predict(x_test_ext)

# Plots
plt.plot(x_train, y_train, 'o', label='data')
plt.plot(x_test, y_test, label='true function')
plt.plot(x_test, prediction, label='linear regression')

plt.title('Linear regression - 11th degree polynomial')
plt.legend()
plt.show()
# Fitting an 11th degree polynomial to the data gives zero training error but has overf
itted on training data.
```



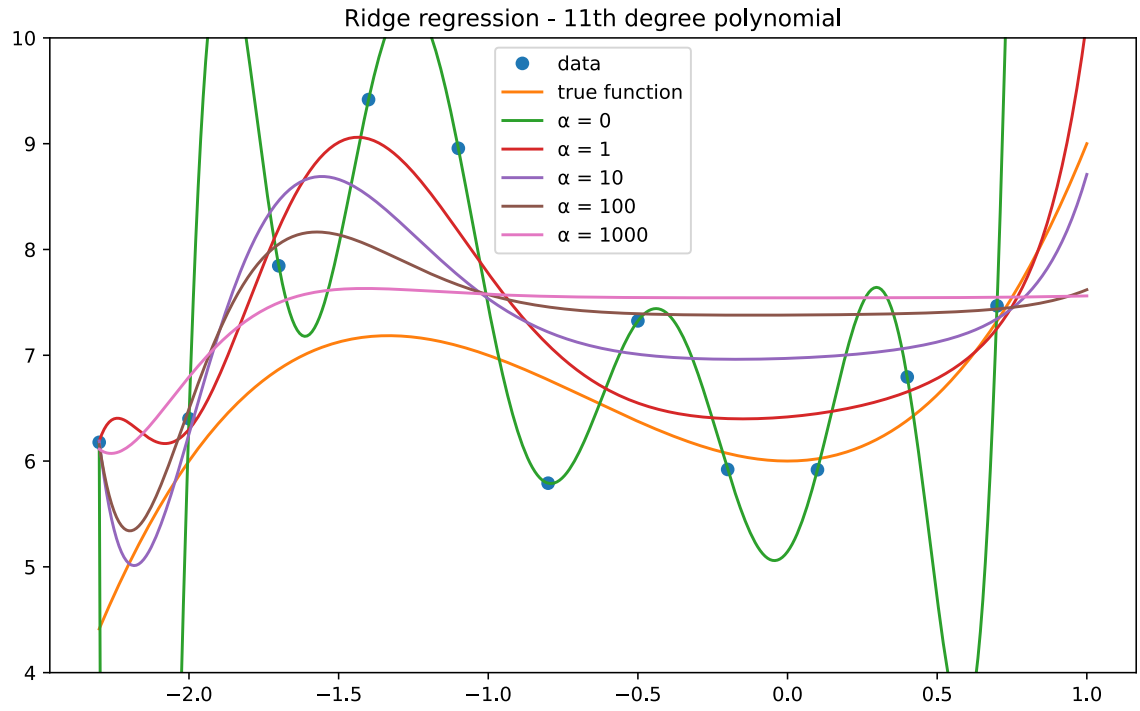
(f)

Use the fucntion `sklearn.linear_model.Ridge` and `sklearn.linear_model.Lasso` to fit a 11th degree polynomial. Also inspect the estimated coefficients. Try different values of penalty term α. What do you observe?

In [26]:

```
# Ridge

plt.plot(x_train, y_train, 'o', label='data')
plt.plot(x_test, y_test, label='true function')
print('Model coefficients:')
for alpha in [0, 1, 10, 100, 1000]:
    model = skl_lm.Ridge(alpha=alpha)
    model.fit(x_train_ext, y_train)
    print(f'\u03b1 = {alpha}')
    print(model.coef_)

    prediction = model.predict(x_test_ext)

    plt.plot(x_test, prediction, label=f'\u03b1 = {alpha}')

plt.title('Ridge regression - 11th degree polynomial')
plt.ylim([4,10])
plt.legend()
plt.show()
```

```
Model coefficients:
α = 0
[   0.      3.866   43.648  -21.823 -267.289 -103.323  462.711  417.687
 -115.039 -277.961 -120.216  -16.951]
α = 1
[ 0.     0.24   0.757 -0.051  0.797 -0.042  0.51   0.385  0.013  0.564
  0.472  0.097]
α = 10
[0.    0.1   0.284 0.04  0.326 0.035 0.283 0.131 0.14  0.269 0.118 0.013]
α = 100
[ 0.     0.013  0.05  -0.003  0.067 -0.016  0.073 -0.008  0.043  0.049
 -0.016 -0.011]
α = 1000
[ 0.     0.     0.006 -0.002  0.009 -0.005  0.011 -0.004  0.006  0.007
 -0.007 -0.003]
```

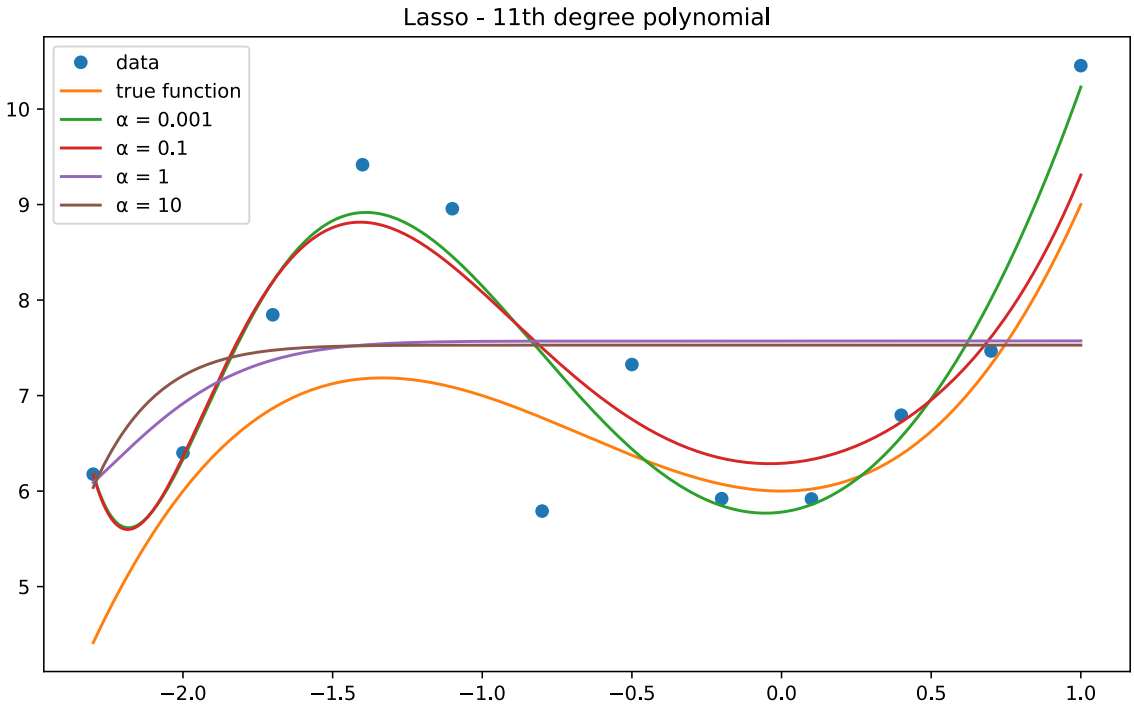


In [27]:

```
# Lasso
plt.plot(x_train, y_train, 'o', label='data')
plt.plot(x_test, y_test, label='true function')
print("Model coefficients:")
for alpha in [0.001, 0.1, 1, 10]:
    model = skl_lm.Lasso(alpha=alpha)
    model.fit(x_train_ext, y_train)
    print(f'\u03b1 = {alpha}:', model.coef_)

    prediction = model.predict(x_test_ext)
    plt.plot(x_test, prediction, label=f'\u03b1 = {alpha}')

    plt.title('Lasso - 11th degree polynomial')
plt.legend()
plt.show()
```

```
Model coefficients:
α = 0.001: [ 0.000e+00  4.025e-01  3.781e+00  4.316e-01 -3.597e-01  2.060e
-01
 -1.741e-02  3.742e-03  3.589e-03 -1.815e-03  9.611e-04 -3.874e-04]
α = 0.1: [ 0.000e+00  1.757e-01  2.201e+00  0.000e+00  1.995e-01  4.209e-0
1
  0.000e+00  1.858e-02  3.575e-03 -1.737e-03  1.491e-03 -4.590e-04]
α = 1: [ 0.     0.     0.    -0.     0.    -0.     0.    -0.     0.     0.
003
 -0.    -0.   ]
α = 10: [ 0.  0.  0. -0.  0.  0. -0.  0. -0.  0. -0.  0.]
```

```
/home/david/.conda/envs/sml/lib/python3.7/site-packages/sklearn/linear_mod
el/_coordinate_descent.py:531: ConvergenceWarning: Objective did not conve
rge. You might want to increase the number of iterations. Duality gap: 2.3
07135274913895, tolerance: 0.0025855884051094904
  positive)
/home/david/.conda/envs/sml/lib/python3.7/site-packages/sklearn/linear_mod
el/_coordinate_descent.py:531: ConvergenceWarning: Objective did not conve
rge. You might want to increase the number of iterations. Duality gap: 4.6
53079500512072, tolerance: 0.0025855884051094904
  positive)
/home/david/.conda/envs/sml/lib/python3.7/site-packages/sklearn/linear_mod
el/_coordinate_descent.py:531: ConvergenceWarning: Objective did not conve
rge. You might want to increase the number of iterations. Duality gap: 0.0
04300875005100124, tolerance: 0.0025855884051094904
  positive)
```



In [28]:

```
# α = 0.01 seems to give a good results for both ridge regression and LASSO.
# For both ridge regression and LASSO all coefficient except the first four
# are very small. Consequently, the remaining coefficients could be decreased
# without increasing the training error too much. This is natural since the
# data was generated from a third degree polynomial. For Lasso, some
# coefficients are estimated to be exactly equal to zero, which is a property
# of Lasso.
```