00 Linear Regression via Least Squares

June 3, 2021

Information: *Basic concepts and simple examples of linear regression via least squares*

Written by: *Zihao Xu*

Last update date: *06.03.2021*

1 Basic concepts

1.1 Regression

- **Regression** refers to a set of methods for modeling the relationship between one or more independent variables and a dependent variable.
  - In the natural sciences and social sciences, the purpose of regression is most often to **characterize** the relationship between the inputs and outputs
  - In machine learning, it is most often concerned with **prediction**
- Regression problems pop up whenever a prediction for a **numerical value** is wanted.
  - Predicting prices
  - Predicting length of stay
  - Demand forecasting
  - ...
- Mathematical Representation
  - Given n observations consisting of

    $$\mathbf{X}_{1:n} = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n\}$$

    which is usually known as *inputs*, *features*, etc. and

    $$\mathbf{y}_{1:n} = \{y_1, y_2, \cdots, y_n\}$$

    which is **continuous** and is usually known as *outputs*, *targets*, etc.
  - The Regression Problem is to use the data to learn the **map** between $\mathbf{x}$ and y

1.2 Linear Regression

- May be both the **simplest** and most **popular** among the standard tools to regression
- A kind of traditional **supervised machine learning**
- **Assumptions**:
  - The relationship between the independent variables $\mathbf{x}$ and the dependent variable y is **linear**
    - ∗ That is to say, y can be expressed as a **weighted sum** of the elements in x

    - ∗ Or in other words, *Linear regression* models the output as a line ($\dim(\mathbf{x}) = 1$) or hyperplane ($\dim(\mathbf{x} > 1)$)
  - There exist some noise on the observations and any noise is **well-behaved** (following a Gaussian distribution)
- **Linear Regression Model**:
  - The linear regression model is defined by the coefficients (or **parameters**) for each feature
  - For $\mathbf{x} \in \mathbb{R}^d, \mathbf{x} = [x_1, x_2, \cdots, x_d]^T$, denote the parameters to the θ:

$$\hat{y} = f(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_d x_d$$

    Let $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \cdots, \theta_d]^T$ and augmented $\tilde{\mathbf{x}} = [1, x_1, x_2, \cdots, x_d]^T$ The the model can be written as

$$\hat{y} = f(x) = \boldsymbol{\theta}^T \tilde{\mathbf{x}}$$

  - This is known as **parametric model**
- **Goal of Linear Regression**:
  - Notice that in **assumption**, we assume that there exist some noises in the observations following a Gaussian distribution. Therefore, we are **not** going to directly solve the equation and expect to get a result $\boldsymbol{\theta}$ that

$$y_i = \boldsymbol{\theta} \tilde{\mathbf{x}}_i \text{ for } 1 \leq i \leq n$$

  - Even when we are confident that the underlying relationship is linear, the noise term should be taken into consideration
  - The goal of linear regression is to find the parameters $\boldsymbol{\theta}$ that **minimize the prediction error**

1.3 Loss function

The most *popular* loss function in regression problems is the **squared error**.

- When the prediction for an example i is $\hat{y}_i$ and the corresponding true label is y_i, the squared error is given by

$$l_i(\boldsymbol{\theta}) = (\hat{y}_i - y_i)^2 = \left(\boldsymbol{\theta}^T \tilde{\mathbf{x}}_i - y_i\right)^2$$

  - Note: In some notations, there is a $\frac{1}{2}$ term for the convenience of notating derivatives, but it makes no difference as optimization
- To measure the quality of a model on the entire dataset of n examples, we simply sum (or **equivalently**, average) the losses on dataset

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} l_i(\boldsymbol{\theta}) = \sum_{i=1}^{n} \left(\boldsymbol{\theta}^T \tilde{\mathbf{x}}_i - y_i\right)^2$$

  In matrix notation:

$$L(\boldsymbol{\theta}) = \|\mathbf{y} - \tilde{\mathbf{X}}\boldsymbol{\theta}\|_2^2$$

  Where $\mathbf{y} = [y_1, y_2, \cdots, y_n]^T \in \mathbb{R}^n$, $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \cdots, \theta_d]^T \in \mathbb{R}^{d+1}$, $\tilde{\mathbf{X}} = [\tilde{\mathbf{x}}_1, \cdots, \tilde{\mathbf{x}}_n]^T \in \mathbb{R}^{n \times (d+1)}$, $\tilde{\mathbf{x}} = [1, x_1, x_2, \cdots, x_d]^T$
  - When averaging the losses on dataset, it's called **Mean Squared Error**
  - $\tilde{\mathbf{X}}$ is called **Design Matrix**
- Known as **Ordinary Least Squares (OLS)**

1.4 Normal Equation: closed-form solution for OLS

Calculate the gradient of OLS

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \|\mathbf{y} - \tilde{\mathbf{X}}\boldsymbol{\theta}\|_2^2 &= \nabla_{\boldsymbol{\theta}} \left[\left(\mathbf{y} - \tilde{\mathbf{X}}\boldsymbol{\theta} \right)^T \left(\mathbf{y} - \tilde{\mathbf{X}}\boldsymbol{\theta} \right) \right] \\
&= \left[2 \left(\mathbf{y} - \tilde{\mathbf{X}}\boldsymbol{\theta} \right)^T \cdot \nabla_{\boldsymbol{\theta}} \left[\mathbf{y} - \tilde{\mathbf{X}}\boldsymbol{\theta} \right] \right]^T \\
&= 2 \left(\mathbf{y} - \tilde{\mathbf{X}}\boldsymbol{\theta} \right) \cdot \left(-\tilde{\mathbf{X}} \right)^T \\
&= 2 \left(-\tilde{\mathbf{X}}^T \mathbf{y} + \tilde{\mathbf{X}}^T \tilde{\mathbf{X}} \boldsymbol{\theta} \right)
\end{aligned}$$

Set the gradient to zero *(first-order optimization)* and solve:

$$\begin{aligned}
2 \left(-\tilde{\mathbf{X}}^T \mathbf{y} + \tilde{\mathbf{X}}^T \tilde{\mathbf{X}} \boldsymbol{\theta}^* \right) &= 0 \\
\tilde{\mathbf{X}}^T \tilde{\mathbf{X}} \boldsymbol{\theta}^* &= \tilde{\mathbf{X}}^T \mathbf{y} \\
\boldsymbol{\theta}^* &= \left(\tilde{\mathbf{X}}^T \tilde{\mathbf{X}} \right)^{-1} \tilde{\mathbf{X}}^T \mathbf{y}
\end{aligned}$$

- This is known as **normal equation**, finds the regression coefficients **analytically**.
- It's an one-step learning algorithm (as opposed to Gradient Descent)

1.5 Normal Equation vs Gradient Descent

1.5.1 Gradient Descent

- Needs to choose GD-based algorithms and set appropriate parameters
- Needs to do a lot of iterations
- Works well with large d (dimension of input data)

1.5.2 Normal Equation

- Gets rid of setting parameters
- Does not need to iterate - compute in one step
- Slow if d is large ($n \leq 10^4$)
- Needs to compute inverse of $\tilde{\mathbf{X}}^T \tilde{\mathbf{X}}$, which is very slow
  - Sometimes use math tricks like *QR factorization* to speed up computation
- Leads to problems if $\tilde{\mathbf{X}}^T \tilde{\mathbf{X}}$ is not invertible

1.6 Improvements for Normal Equation

1.6.1 Orthogonalization

To speed up computation of $\left(\tilde{\mathbf{X}}^T \tilde{\mathbf{X}} \right)^{-1}$

- Make the columns of $\tilde{\mathbf{X}}$ orthonormal - orthogonal to each other and of length 1
- Do the **QR factorization** and obtain matrix $\tilde{\mathbf{X}} = \mathbf{QR}$

- With the property $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$, the calculation can be simplified a lot:

$$\begin{aligned}
&\tilde{\mathbf{X}}^T\tilde{\mathbf{X}} = \mathbf{R}^T\mathbf{Q}^T\mathbf{Q}\mathbf{R} = \mathbf{R}^T\mathbf{R} \\
&\Rightarrow \mathbf{R}^T\mathbf{R}\boldsymbol{\theta}^* = \mathbf{R}^T\mathbf{Q}^T\mathbf{y} \\
&\Rightarrow \boldsymbol{\theta}^* = \mathbf{R}^{-1}\mathbf{Q}^T\mathbf{y}
\end{aligned}$$

- Do not need to invert $\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$ directly

1.6.2 Singular Value Decomposition

Apply SVD to $\tilde{\mathbf{X}}$

$$\begin{aligned}
\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y} &= \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T\boldsymbol{\theta} - \mathbf{y} \\
&= \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T\boldsymbol{\theta} - \mathbf{U}\mathbf{U}^T\mathbf{y} \\
&= \mathbf{U}\left(\boldsymbol{\Sigma}\mathbf{V}^T\boldsymbol{\theta} - \mathbf{U}^T\mathbf{y}\right)
\end{aligned}$$

Denote $\mathbf{v} = \mathbf{V}^T\boldsymbol{\theta}$ and $\mathbf{z} = \mathbf{U}^T\mathbf{y}$

$$\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y} = \mathbf{U}\left(\boldsymbol{\Sigma}\mathbf{v} - \mathbf{z}\right)$$

Notice that orthogonal matrices preserve the L^2 norm

$$\left\|\mathbf{U}\left(\boldsymbol{\Sigma}\mathbf{v} - \mathbf{z}\right)\right\|_2^2 = \left\|\boldsymbol{\Sigma}\mathbf{v} - \mathbf{z}\right\|_2^2$$

That is to say

$$\left\|\tilde{\mathbf{X}}\boldsymbol{\theta} - \mathbf{y}\right\|_2^2 = \left\|\boldsymbol{\Sigma}\mathbf{v} - \mathbf{z}\right\|_2^2$$

In this way, the OLS regression problem is reduced to a diagonal form.

To minimize $\left\|\boldsymbol{\Sigma}\mathbf{v} - \mathbf{z}\right\|_2^2$, first denote

$$\operatorname{diag}\left(\boldsymbol{\Sigma}\right) = \left(\sigma_1, \sigma_2, \cdots, \sigma_r, 0, \cdots, 0\right)$$

(Always remember $\boldsymbol{\Sigma}$ is not necessarily a diagonal square matrix). Then we can get

$$\boldsymbol{\Sigma}\mathbf{v} = \begin{bmatrix} \sigma_1 v_1 \\ \sigma_2 v_2 \\ \vdots \\ \sigma_r v_r \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad \boldsymbol{\Sigma}\mathbf{v} - \mathbf{z} = \begin{bmatrix} \sigma_1 v_1 - z_1 \\ \sigma_2 v_2 - z_2 \\ \vdots \\ \sigma_r v_r - z_r \\ -z_{r+1} \\ \vdots \\ -z_m \end{bmatrix}$$

Since we're minimizing w.r.t. $\mathbf{v}$, only first r components of $\boldsymbol{\Sigma}\mathbf{v} - \mathbf{z}$ matter. Therefore, we can make these $\sigma_i v_i - z_i$ as small as possible by using $v_i = \frac{z_i}{\sigma_i}$, which makes the first r components become 0 and the rest are $-c_i$, thus

$$\left\|\boldsymbol{\Sigma}\mathbf{v} - \mathbf{z}\right\|_2^2 = \sum_{i=r+1}^{m} c_i^2$$

The compact solution in this way is

$$\boldsymbol{\theta}^* = \tilde{\mathbf{X}}^+\mathbf{y}$$

where $\tilde{\mathbf{X}}^+$ is the **Moore-Penrose Pseudoinverse**

1.7 Basis Function Regression

1.7.1 Motivation

To adapt linear regression to nonlinear relationships between variables

1.7.2 Definition

A **generalization of linear regression** that essentially replaces each input with a basis function of the input.

- A linear basis function model that uses the identity function is just linear regression.
- The model is **linear in the coefficients** of basis functions. The basis functions can be non-linear.

1.7.3 Mathematical Representation

The form of the basis function regression is

$$y = f(\mathbf{x}) = \sum_{i=1}^{m} \theta_i \phi_i(\mathbf{x}) = \boldsymbol{\theta}^T \boldsymbol{\phi}(\mathbf{x})$$

where $\boldsymbol{\theta} = (\theta_1, \cdots, \theta_m)^T$ are coefficients and $\boldsymbol{\phi} = (\phi_1, \cdots, \phi_m)^T$ are arbitrary basis functions

1.7.4 Common Basis Function Regression Model

- **Polynomials**
  - In the polynomial model, the basis functions are polynomial functions. For example, take
$$\phi_1 = 1, \phi_2 = x, \phi_3 = x^2, \cdots$$
- **Multi-dimensional polynomials**
  - $\phi_i(\mathbf{x}) = \Sigma_{\alpha \in A_i} \beta_\alpha \mathbf{x}^\alpha$
- **Radial basis functions**
  - $\phi_i(\mathbf{x}) = \exp\left\{-\frac{\|\mathbf{x}-x_i\|^2}{2l^2}\right\}$
- **Fourier Series**
  - $\phi_{2j}(x) = \cos\left(\frac{2j\pi}{L}x\right)$ and $\phi_{2j+1}(x) = \sin\left(\frac{2j\pi}{L}x\right)$

2 Analytically solve linear regression problems

The general process of analytically solving linear regression problems are

- Get the dataset
- Select the model to fit the data with (choose basis functions)
- Build the *design matrix* accordingly
- Solve the problem by applying SVD to the design matrix
- Check the regression result on some evaluation data

2.1 Simple linear regression with a single variable

We'll start with pairs of $x$ and y which definitely have a linear relationship while y may be contaminated with Gaussian noise. In particular, we generate the data from

$$y_i = -0.5 + 2x_i + 0.1\epsilon_i$$

where $\epsilon_i \in \mathcal{N}(0,1)$ and we samples $x_i \sim U([0,1])$

First, import necessary modules we would use in such a simple regression problem.

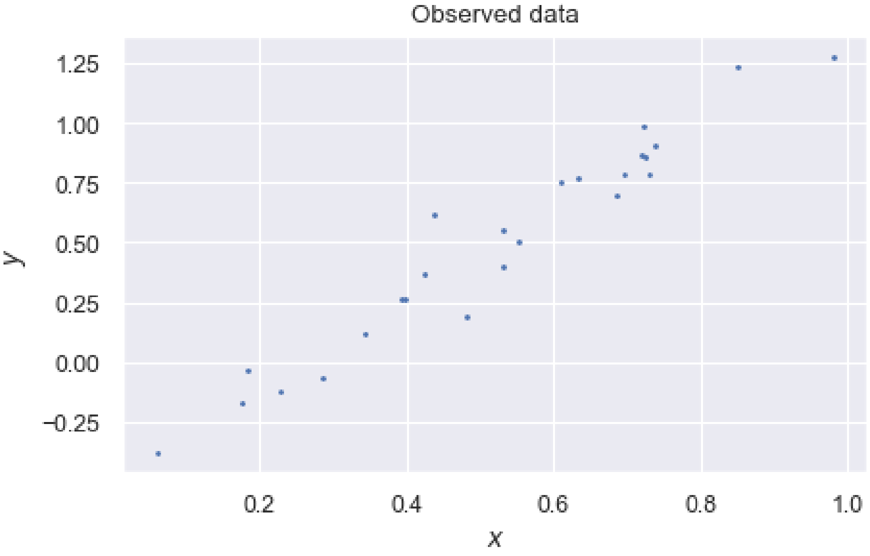
[1]:
```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(123)
```

seaborn is a Python data visualization library based on **matplotlib**. It provides a high-level interface for drawing attractive and informative statistical graphics. Here we simply use it to set the figure styles and line widths.

[2]:
```python
import seaborn as sns
# Use the default figure style in seaborn
sns.set()
# Set the line width
sns.set_context('paper')
```

Generate the synthetic dataset and visualize it.

[3]:
```python
# Number of observations
num_obs = 25
# Sample x
x = np.random.rand(num_obs).reshape(-1, 1)
# True parameters
theta_true = np.array([-0.5, 2.0]).reshape(-1, 1)
# Calculate y samples
y = theta_true[0] + theta_true[1] * x + 0.1 * np.random.randn(num_obs).reshape(
    -1, 1)
# Visulize the dataset
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.plot(x, y, '.', markersize=2)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title('Observed data')
plt.show()
```



To use least squares to fit the data into the model

$$y = \theta_0 + \theta_1 x$$

First build the $N \times 2$ design matrix $\tilde{\mathbf{X}}$

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix}$$

[4]:

```
X = np.hstack([np.ones((num_obs, 1)), x])
print(X[0, :])
```

```
[1.         0.69646919]
```

To get the analytic solution by the means of singular value decomposition mentioned above, call the `numpy.linalg.lstsq()` method

[5]:

```
# We only cares the solutions
theta = np.linalg.lstsq(X, y, rcond=None)[0]
print('The estimated theta0 is: %1.3f' % theta[0])
print('The estimated theta1 is: %1.3f' % theta[1])
```

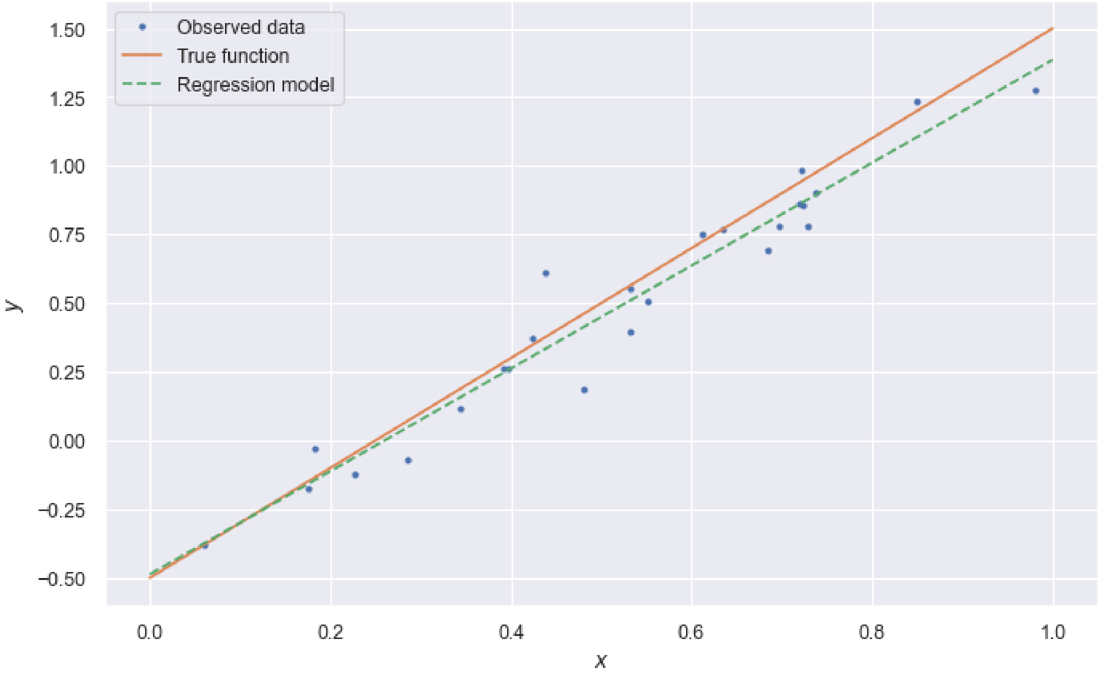
```
The estimated theta0 is: -0.487
The estimated theta1 is: 1.872
```

Considering that the true values are $\theta_0 = -0.5, \theta_1 = 2$, the values we found are very close to the correct values, but not exactly the same. This is good enough for such a small dataset containing only 25 samples with noises. In general, the more noise there is, the more observations it would take to identify the regression coefficients correctly.

Now visualize the regression function.

[6]:

```python
# Some points on which to evaluate the regression function
xe = np.linspace(0, 1, 100)
# True function
ye_true = theta_true[0] + theta_true[1] * xe
# The regression model
ye = theta[0] + theta[1] * xe
# Visulization
fig, ax = plt.subplots(figsize=(8, 5), dpi=100)
ax.plot(x, y, '.', markersize=4, label='Observed data')
ax.plot(xe, ye_true, label='True function')
ax.plot(xe, ye, '--', label='Regression model')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc='best')
plt.show()
```



2.2 Polynomial Regression with a single variable

Try to fit a linear regression model to the data generated from

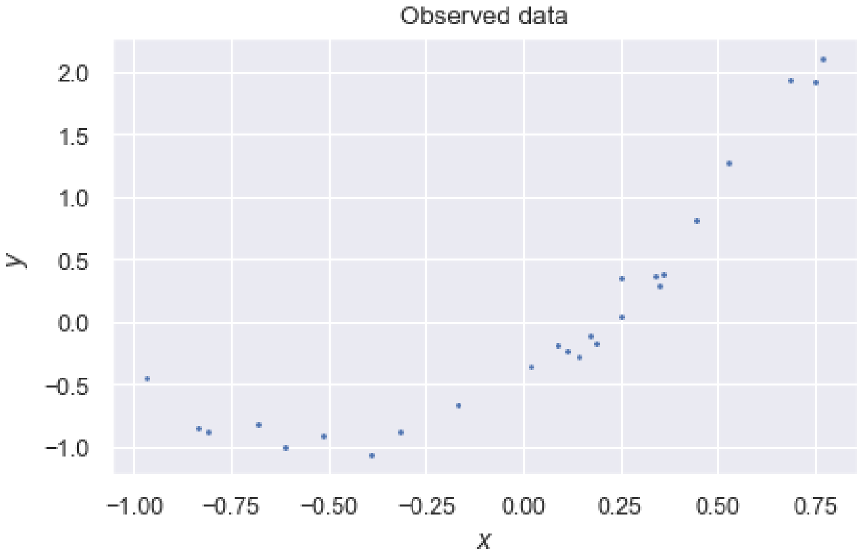
$$y_i = -0.5 + 2x_i + 2x_i^2 + 0.1\epsilon_i$$

where $\epsilon_i \sim \mathcal{N}(0,1)$ and we sample $x_i \sim U([-1,1])$

First generate the dataset and visualize it

[7]:

```python
# Number of observations
num_obs = 25
# Sample x
x = (2 * np.random.rand(num_obs) - 1.0).reshape(-1, 1)
# True parameters
theta_true = np.array([-0.5, 2.0, 2.0]).reshape(-1, 1)
# Calculate y samples
y = theta_true[0] + theta_true[1] * x + theta_true[2] * x**2 \
+ 0.1*np.random.randn(num_obs).reshape(-1, 1)
# Visulize the dataset
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.plot(x, y, '.', markersize=2)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title('Observed data')
plt.show()
```



To use least squares to fit the data into the model

$$y = \theta_0 + \theta_1 x + \theta_2 x^2$$

First build the $N \times 3$ design matrix $\tilde{\mathbf{X}}$

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}$$

[8]:

```
X = np.hstack([np.ones((num_obs, 1)), x, x**2])
print(X[0, :])
```

```
[1.         0.09013601 0.0081245 ]
```

Then solve this regression problem with the same method

[9]:

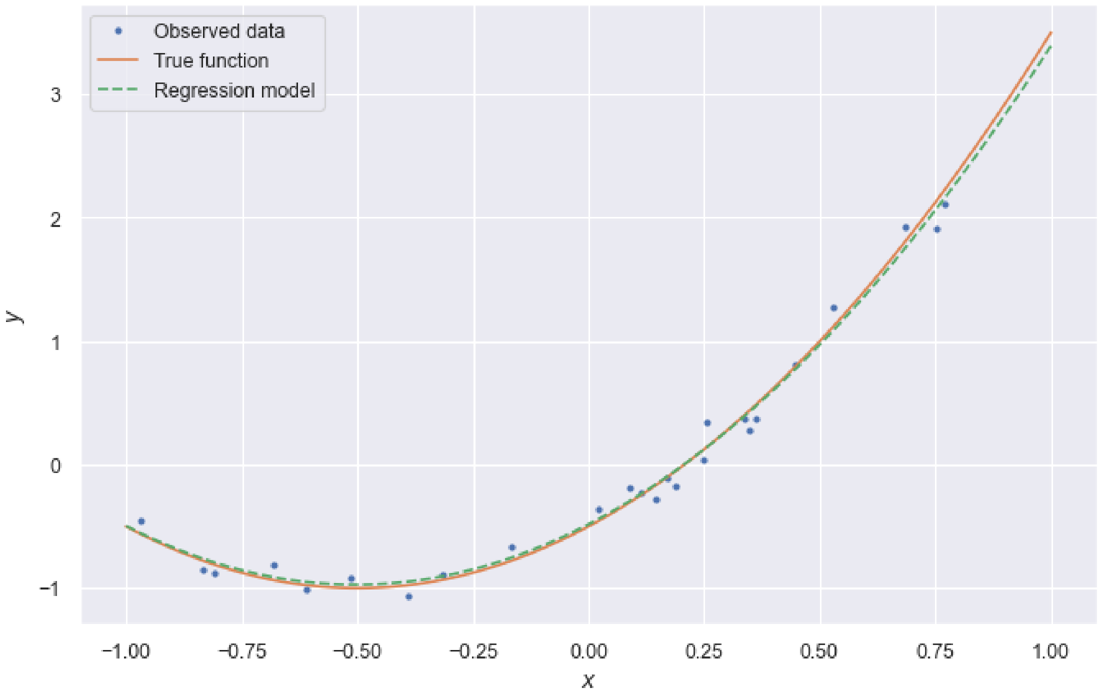
```
theta = np.linalg.lstsq(X, y, rcond=None)[0]
print('The estimated theta0 is: %1.3f' % theta[0])
print('The estimated theta1 is: %1.3f' % theta[1])
print('The estimated theta2 is: %1.3f' % theta[2])
```

```
The estimated theta0 is: -0.482
The estimated theta1 is: 1.944
The estimated theta2 is: 1.930
```

Check the performance on some evaluation points

[10]:

```
# Some points on which to evaluate the regression function
xe = np.linspace(-1, 1, 100)
# True function
ye_true = theta_true[0] + theta_true[1] * xe + theta_true[2] * xe**2
# The regression model
ye = theta[0] + theta[1] * xe + theta[2] * xe**2
# Visulization
fig, ax = plt.subplots(figsize=(8, 5), dpi=100)
ax.plot(x, y, '.', markersize=4, label='Observed data')
ax.plot(xe, ye_true, label='True function')
ax.plot(xe, ye, '--', label='Regression model')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc='best')
plt.show()
```



2.3 Linear regression with Fourier basis

Try to fit the data generated from

$$y_i = -0.5 + 2x_i + 2x_i^2 + sin(x_i) - sin(2x_i) + \epsilon_i$$

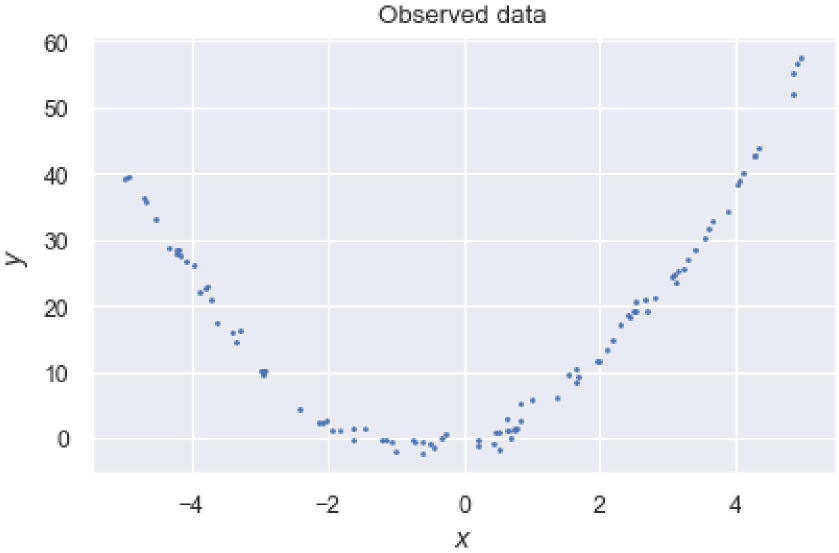
where $\epsilon_i \sim \mathcal{N}(0, 1)$ and we sample $x_i \sim U([-5, 5])$

First generate the dataset and visualize it.

[11]:

```
# Number of observations
num_obs = 100
# Sample x
x = (10 * np.random.rand(num_obs) - 5.0).reshape(-1, 1)
# True parameters
theta_true = np.array([-0.5, 2.0, 2.0, 1, -1]).reshape(-1, 1)
# Calculate y samples
y = theta_true[0] + theta_true[1] * x + theta_true[2] * x**2 \
+ theta_true[3]*np.sin(x) + theta_true[4]*np.sin(2*x)\
+ np.random.randn(num_obs).reshape(-1, 1)
# Visulize the dataset
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.plot(x, y, '.', markersize=2)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title('Observed data')
```

```
plt.show()
```



Next step is to build the design matrix for Fourier basis

$$\phi_{2j}(x) = \cos\left(\frac{2j\pi}{L}x\right), \phi_{2j+1}(x) = \sin\left(\frac{2j\pi}{L}x\right)$$

for $j = 1, \ldots, m/2$

Here we choose $m = 4$ arbitrarily (how to select m would be introduced later)

[12]:

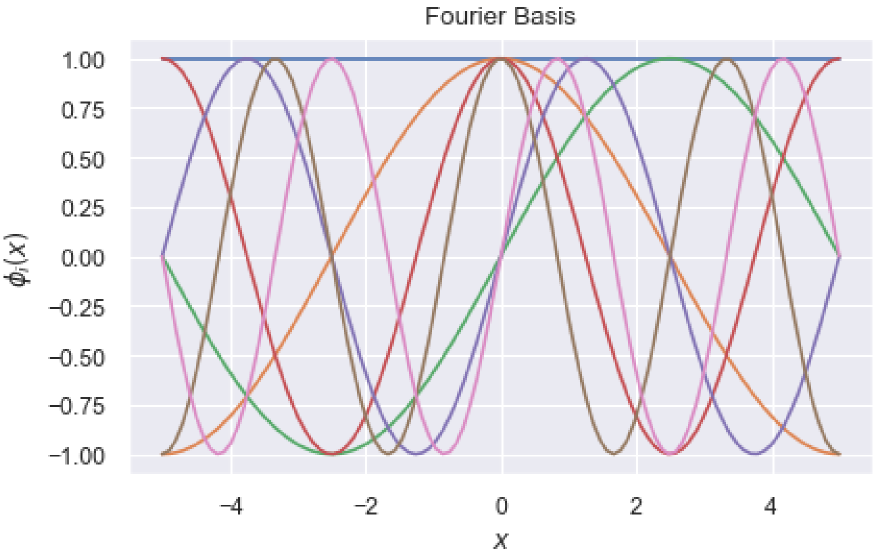
```
# Define the function since we need to call this repeatedly
def get_fourier_design_matrix(input_x, length, num_terms):
    cols = [np.ones((input_x.shape[0], 1))]
    for ii in range(int(num_terms / 2)):
        cols.append(np.cos(2 * (ii + 1) * np.pi / length * input_x))
        cols.append(np.sin(2 * (ii + 1) * np.pi / length * input_x))
    return np.hstack(cols)


# Length of domain
L = 10
m = 6
Phi = get_fourier_design_matrix(x, L, m)
```

Visualize the Fourier basis

[13]:

```
xe = np.linspace(-5, 5, 100).reshape(-1, 1)
Phi_xe = get_fourier_design_matrix(xe, L, m)
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
```

```
ax.plot(xe, Phi_xe)
ax.set_xlabel('$x$')
ax.set_ylabel('$\phi_i(x)$')
ax.set_title('Fourier Basis')
plt.show()
```

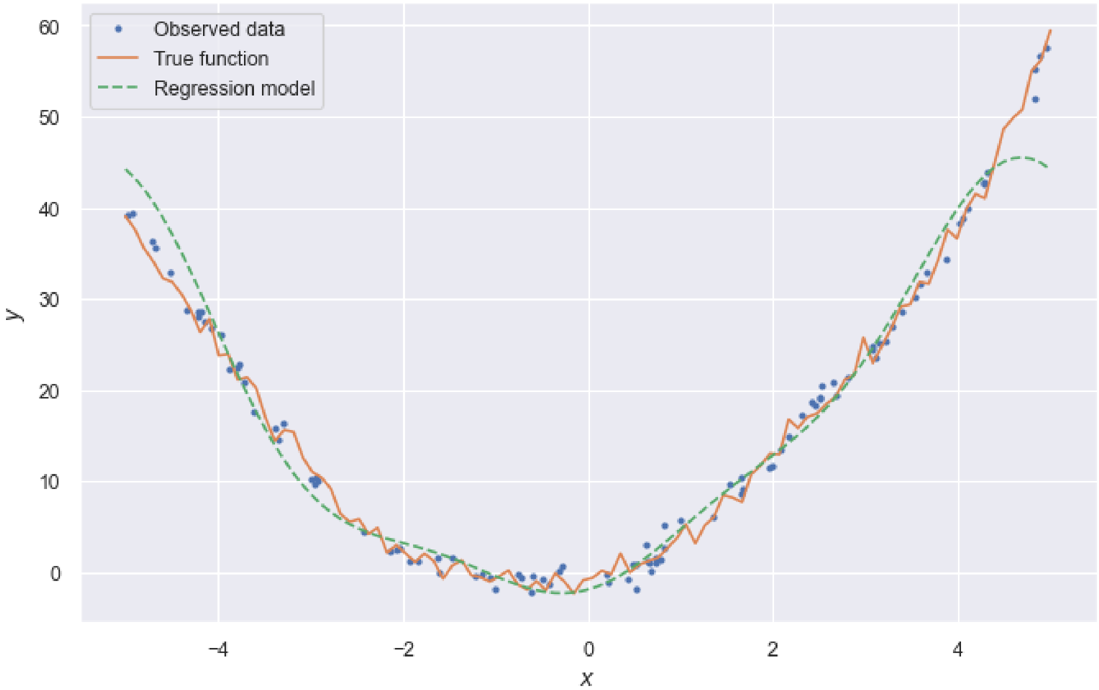


Again, solve the problem with the same method

[14]:
```
theta = np.linalg.lstsq(Phi, y, rcond=None)[0]
```

Check the performance on some evaluation points

[15]:
```
# True function
ye_true = theta_true[0] + theta_true[1] * xe + theta_true[2] * xe**2 \
+ theta_true[3]*np.sin(xe) + theta_true[4]*np.sin(2*xe)\
+ np.random.randn(num_obs).reshape(-1, 1)
# Regression result
ye = np.dot(Phi_xe, theta)
# Visulization
fig, ax = plt.subplots(figsize=(8, 5), dpi=100)
ax.plot(x, y, '.', markersize=4, label='Observed data')
ax.plot(xe, ye_true, label='True function')
ax.plot(xe, ye, '--', label='Regression model')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc='best')
plt.show()
```



2.4 Make use of Scikit-Learn package

While the process of solving linear regression problems are similar, the `Scikit-Learn` package provides several encapsulated method which can analytically solve linear regression models in several lines.

- **`linear_model`** module implements a variety of linear models
- **`preprocessing`** modules provides a simple way to build design matrices for different basis functions
- **`pipeline`** module implements utilities to build a composite estimator, as a chain of transforms and estimators

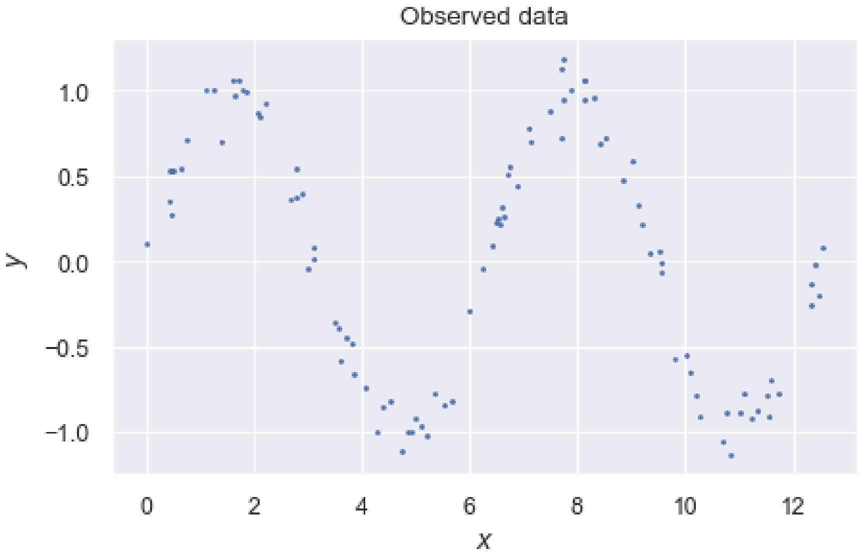
Here is a simple example of using the package to solve a simple regression problems with polynomial features. The dataset to be fit is a noisy version of a sin wave

$$y_i = \sin(x_i) + 0.1 * \epsilon$$

where $\epsilon_i \sim \mathcal{N}(0, 1)$ and we sample $x_i \sim U([0, 4\pi])$

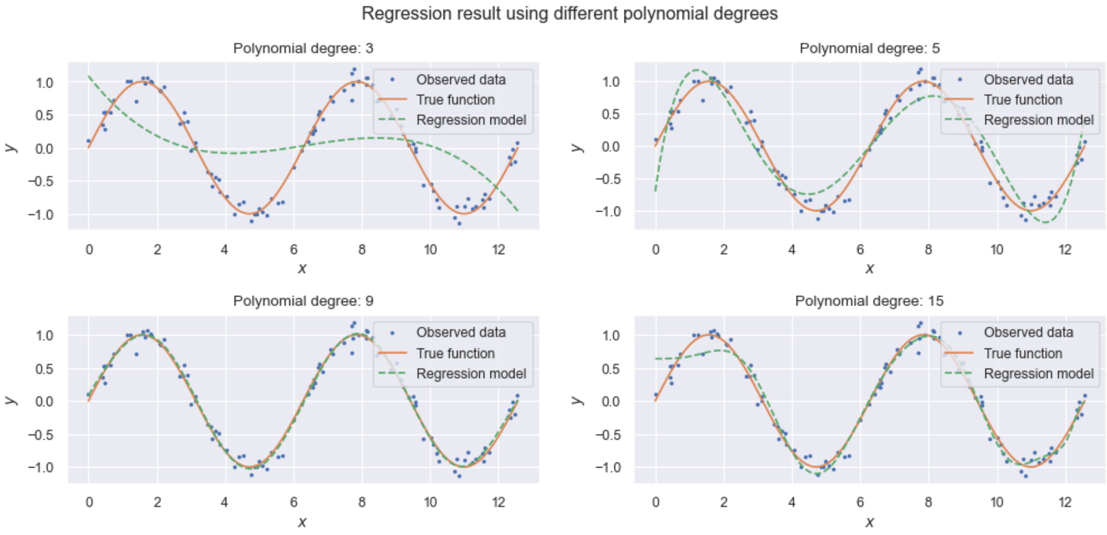
[16]:
```
# Number of observations
num_obs = 100
# Sample x
x = 4 * np.pi * np.random.rand(num_obs).reshape(-1, 1)
# Calculate y samples
y = np.sin(x) + 0.1 * np.random.randn(num_obs).reshape(-1, 1)
# Visulize the dataset
```

```
fig, ax = plt.subplots(figsize=(5, 3), dpi=100)
ax.plot(x, y, '.', markersize=2)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title('Observed data')
plt.show()
```



Solve the problem with Scikit-Learn package

[17]:
```
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures

# Try with different degrees easily
degrees = np.array([3, 5, 9, 15])
# Some evaluation points
xe = np.linspace(0, 4 * np.pi, 100).reshape(-1, 1)
# True value at the evaluation points
ye_true = np.sin(xe)
fig, axes = plt.subplots(2, 2, figsize=(12, 5), dpi=100)
axes = axes.flatten()
for index, degree in enumerate(degrees):
    # Assign the regression model and preprocessing method
    estimator = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    # Fit with the created model
    estimator.fit(x, y)
    # See the performance on evaluation points
    ye = estimator.predict(xe)
    # Visualization
    ax = axes[index]
    ax.plot(x, y, '.', markersize=3, label='Observed data')
    ax.plot(xe, ye_true, label='True function')
    ax.plot(xe, ye, '--', label='Regression model')
    ax.set_xlabel('$x$')
    ax.set_ylabel('$y$')
    ax.set_title('Polynomial degree: %d' % degree)
    ax.legend(loc='upper right')
plt.subplots_adjust(hspace=0.5)
plt.suptitle('Regression result using different polynomial degrees')
plt.show()
```



[ ]: