Week 7: Regression 1

Univariate Regression

Dr Giuseppe Brandi

Northeastern University London

What is regression?

A type of supervised learning, where each input example has a known target:

- If target is **discrete**, then it is a classification problem
  - "Which one?"
- If target is **continuous**, then it is regression
  - "How many?" or "How much?"

Example problems

"Predict

- CO_2 emissions[1]
- Bicycle traffic

in London given *datetime*, *weather season*, etc."

So, we collect samples – **both** input and target values – and we wish to understand a trend.

[1] Note that targets are continuous random variables.

Linear regression, Part I

The simple **linear regression** model involves a linear combination of input variables:

$$y(\mathrm{x}, \mathrm{w}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d$$

where $\mathrm{x} = (x_1, \ldots, x_d)^T$ is a d-dimensional input vector. The key idea is that $y(\mathrm{x}, \mathrm{w})$ is a **linear function** of parameters w

Fitting a straight line

The simplest example is to fit a straight line to data:

$$y = w_0 + w_1 x$$

Parameter w_0 is known as the **intercept** of the line; and w_1 as the **slope** of the line

Let's try an example...

Fitting a straight line

Consider the task of approximating the line $y(x) = -5 + 2x$.

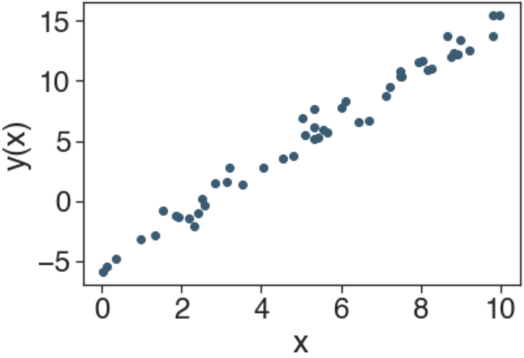
Generating sample data

```python
import numpy as np

# Create a random number generator (with seed)
rng = np.random.RandomState(123456789)
# Generage 50 random values for x, scaled x10
x = 10 * rng.rand(50)
# Compute y(x), adding random noise
y = 2 * x - 5 + rng.randn(50)
```

Fitting a straight line

Consider the task of approximating the line $y(x) = -5 + 2x$.



Solving it in Python

Consider the task of approximating the line $y(x) = -5 + 2x$.

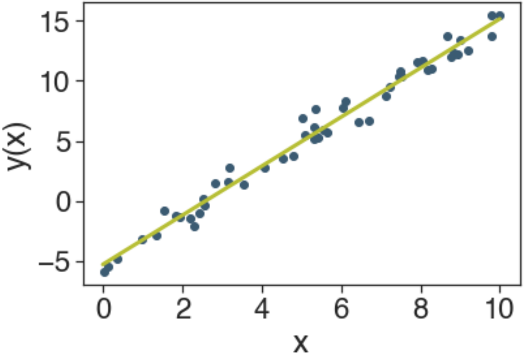
Fit a straight line

```python
from sklearn.linear_model import LinearRegression

# Create a linear regression model
model = LinearRegression(fit_intercept=True)
# Fit the training data
model.fit(x[:, np.newaxis], y)
# Predict y-value for 1,000 data points
u = np.linspace(0, 10, 1000)
v = model.predict(u[:, np.newaxis])
```

Fitting a straight line

Consider the task of approximating the line $y(x) = -5 + 2x$.



Solving it in Python

Consider the task of approximating the line $y(x) = -5 + 2x$.

Fit a straight line

```python
print('y(x) = {:.3f} + {:.3f}x'.format(
    model.intercept_,
    model.coef_[0]))
```

```
y(x) = -5.259 + 2.041x
```

The result is pretty close. *Is it only about straight lines?*

Linear regression, Part II

The simple linear regression model is also a linear function of input variables **x**, which is a limiting factor.

So, we consider linear combinations of non-linear functions:

$$y(\mathrm{x}, \mathrm{w}) = w_0 + w_1\phi_1(\mathrm{x}) + w_2\phi_2(\mathrm{x}) + \cdots + w_{m-1}\phi_{m-1}(\mathrm{x})$$

where $\phi_i(\mathrm{x})$ are known as **basis functions**. The size of the model (that is, the number of parameters) is m.

Linear regression, Part II

For example, if $\phi_j(x) = x^j$ then:

$$y(\mathrm{x}, \mathrm{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_{m-1} x^{m-1}$$

is a **polynomial regression** model.

In a similar fashion, we can use a Gaussian basis function (see our *handbook*)

Polynomial curve fitting

Let's construct a more complicated example, fitting a **curve**, using the function $y(x) = \sin(2\pi x)$.

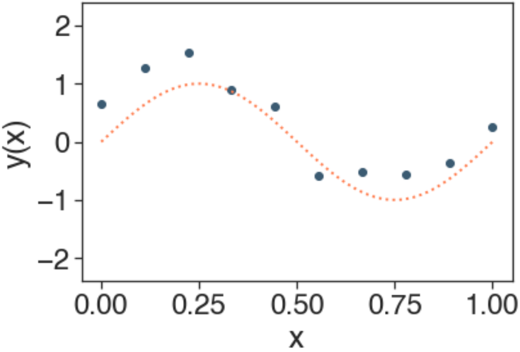
We will generate 10 sample values for x and $y(x)$, adding random normal noise to the latter.

Generating sample data

```
# Generate 10 equally-spaced values in range 0 to 1
x = np.linspace(0, 1, 10)
# Compute y(x), adding noise from
# a normal distribution N(0, 0.3)
y = np.sin(2 * np.pi * x) +
    rng.normal(scale=0.3, size=10)
```

Polynomial curve fitting

Let's construct a more complicated example, fitting a **curve**, using the function $y(x) = \sin(2\pi x)$.



Polynomial curve fitting

Let's construct a more complicated example, fitting a **curve**, using the function $y(x) = \sin(2\pi x)$.

Our task is to discover the function $y(x)$ from a finite dataset by **fitting the training data** with a polynomial function.

Let's consider, for now, a polynomial of *degree* (or *order*) 4:

$$y = w_1 x + w_2 x^2 + w_3 x^3 + w_4 x^4$$

The polynomial will be determined by $w_1, \ldots, w_4$.

Polynomial curve fitting

Let's construct a more complicated example, fitting a **curve**, using the function $y(x) = \sin(2\pi x)$.

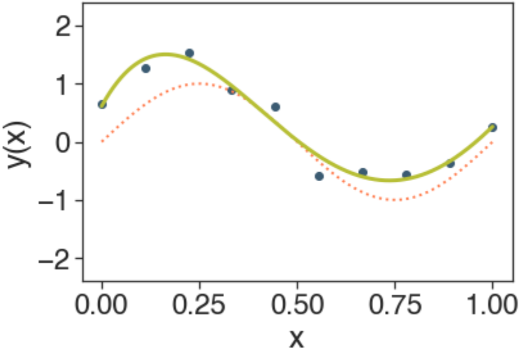
Solving it in Python

```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

# A polynomial regression model of degree 4
model = make_pipeline(PolynomialFeatures(4),
                      LinearRegression())
model.fit(x[:, np.newaxis], y)
# Predict y-value for 1,000 data points
u = np.linspace(0, 1, 1000)
v = model.predict(u[:, np.newaxis])
```

Polynomial curve fitting

Let's construct a more complicated example, fitting a **curve**, using the function $y(x) = \sin(2\pi x)$.



How does it work?

The values of the coefficients (in our case, $w_1, \ldots, w_4$) are determined by minimising an **error function**.

The error measures the misfit between $\hat{y}$ and y, the expected result.

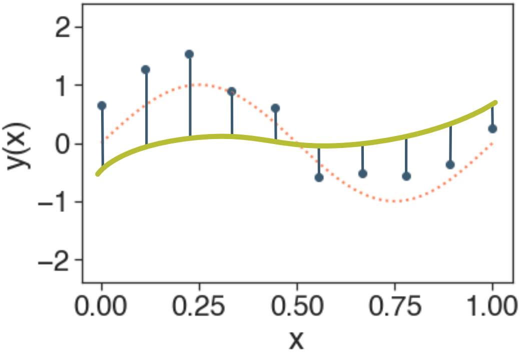
A widely used error function is the sum-of-squares error:

$$E(\mathrm{w}) = \frac{1}{2} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

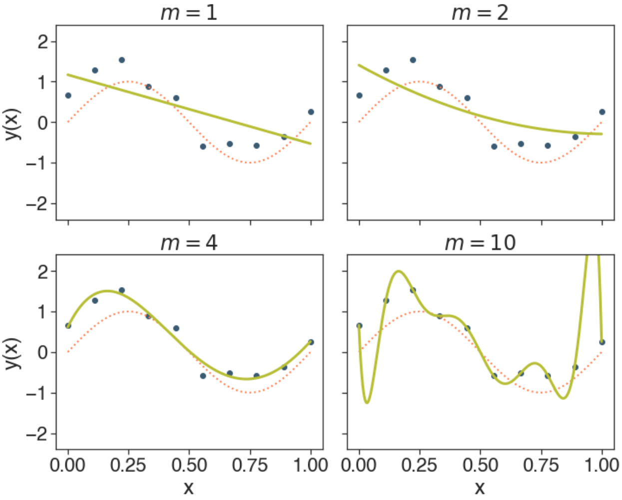
where n is the number of input samples.

How does it work?

The error measures the misfit between $\hat{y}$ and y, the expected result.

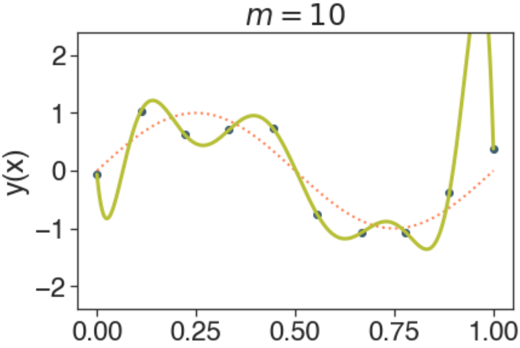


Why $m = 4$?



What has happened when $m = 10$?

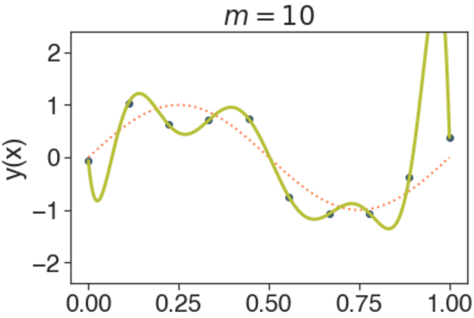
We have 10 sample data points (the blue points in the graph); and 10 parameters $w_1, \ldots, w_{10}$. So, each parameter w can be fine-tuned to match exactly each point (including noise).



What has happened when $m = 10$?

But for points in the middle, the polynomial oscillates a lot!

By fitting each data point, values of w become too large!



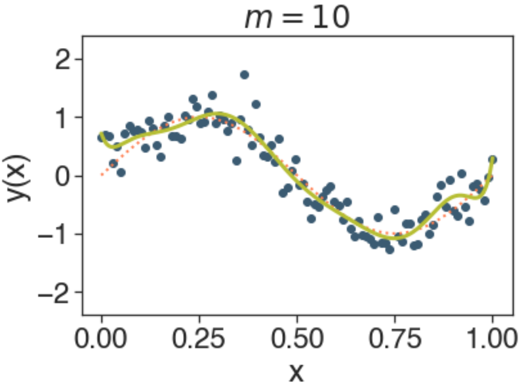
This behaviour is known as **over-fitting**.

The goal is to generalise, not overfit

Tip

The **goal** of machine learning is **good generalisation**; that is, the ability to predict new data.

Throwing more data to the problem helps, but...



We can say: "the size of the dataset should be 5× or 10× the size of the model." But this is not good...

Regularisation

We want to the size of the model to be related to the complexity of the problem solved, not the size of the dataset, right?

Regularisation is a technique to address over-fitting.

It introduces a penalty term to the error function so that values of w do not get too large:

$$E(\mathrm{w}) = \frac{1}{2}\sum_{i=1}^{n}(y - \hat{y})^2 + \alpha \text{ penalty}$$

Regularisation

- Ridge regression

$$E(\mathrm{w}) = \frac{1}{2}\sum_{i=1}^{n}(y-\hat{y})^2 + \frac{\alpha}{2}\sum_{j=1}^{m}{w_j}^2$$

- Lasso regression

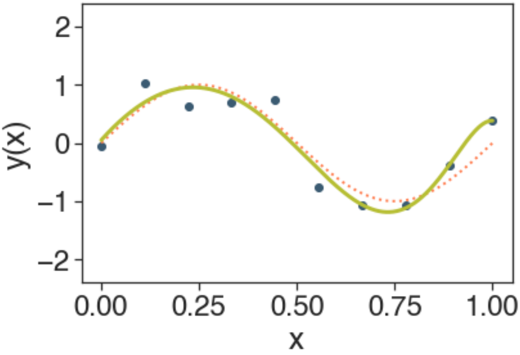
$$E(\mathrm{w}) = \frac{1}{2}\sum_{i=1}^{n}(y-\hat{y})^2 + \frac{\alpha}{2}\sum_{j=1}^{m}|w_j|$$

Parameter α controls the **strength of the penalty**. Also, this technique is known as *weight decay*.

Solving it in Python

```python
from sklearn.linear_model import Ridge
# Back to 10 input data points
x = np.linspace(0, 1, 10)
y = np.sin(2 * np.pi * x) +
    rng.normal(scale=0.3, size=10)
# Create a rigde regression model with 10 params
model = make_pipeline(PolynomialFeatures(10),
                      Ridge(alpha=0.0001))
model.fit(x[:, np.newaxis], y)
# Predict y-value for 1,000 data points
u = np.linspace(0, 1, 1000)
v = model.predict(u[:, np.newaxis])
```

Even with few training data and 10 parameters, regularisation improves curve fitting.



Lasso regression

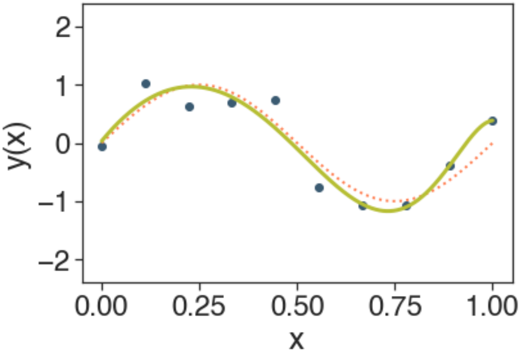
Solving it in Python

```python
from sklearn.linear_model import Lasso

# Create a lasso regression model with 10 params
model = make_pipeline(PolynomialFeatures(10),
                      Lasso(alpha=0.0001,
                            max_iter=100000))
model.fit(x[:, np.newaxis], y)
# Predict y-value for 1,000 data points
u = np.linspace(0, 1, 1000)
v = model.predict(u[:, np.newaxis])
```

Lasso regression

Even with few training data and 10 parameters, regularisation improves curve fitting.



Why is regularisation important?

A regulariser is one of the methods to **shrink parameter values towards zero**.

Thus, it is an opportunity to derive **sparse models**.

Tip

We do not have to worry (much) about the size of the model with respect to the input data.

The *effective number of parameters* (that is, the non-zero ones) **automatically adapts** to the problem.

Hyper-parameters

Let's recap the number of **hyper-parameters** that we have to tune to solve a regression problem:

- The choice of a basis function e.g., polynomial
- The model size, i.e., the number of parameters, m
- The parameter α of the regulariser (set $\alpha = 0$ to disable it)
- The number of iterations (depending on the solver)

Summary

We learned the fundamental of linear regression models, plus:

- What is an over-parameterised model?
- What is the over-fitting problem?
- What is regularisation?