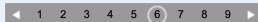


## Copyright and UWA unit content

Is it OK to download and share course material such as lectures, unit outlines, exam papers, articles and ebooks?



UWA is committed to providing easy access to learning material and many of your lectures are available for online access via the Lecture Capture System (LCS), accessible through the LMS. Your unit coordinator may make their lecture recordings available to download if they wish. You are allowed to access recorded lectures in the format they are supplied on the LCS – so if they are not made available to download, you must not use any software or devices to attempt to download them.

All recorded lectures and other course material, such as presentation slides, lecture and tutorial handouts, unit outlines and exam papers, are protected under the Copyright Act and remain the property of the University. You are not allowed to share these materials outside of the LMS – for example, by uploading them to study resource file sharing websites or emailing them to friends at other universities. Distributing course material outside of the LMS is a breach of the [University Policy on Academic Conduct](#) and students found to be sharing material on these sites will be penalised. University data, emails and software are also protected by copyright and should not be accessed, copied or destroyed without the permission of the copyright owner.

Other material accessible from the LMS or via the Library, such as ebooks and journal articles, are made available to you under licensing agreements that allow you to access them for personal educational use, but not to share with others.

### Can I share my login details?

No! Pheme is your key to accessing a number of UWA's online services, including LMS, studentConnect, UWA email, your Library account, and Unifi. These services hold copyright material as well as your personal information, including your unit marks, enrolment information and contact details, so it is important that you do not share the access credentials with anyone else.

[https://www.student.uwa.edu.au/learning/resources/ace/  
respect-intellectual-property/copyright-and-uwa-unit-content](https://www.student.uwa.edu.au/learning/resources/ace/respect-intellectual-property/copyright-and-uwa-unit-content)

# CITS5508 Machine Learning

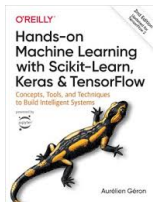
Débora Corrêa (Unit Coordinator and Lecturer)

2024

# Today

## Chapter 4.

# Hands-on Machine Learning with Scikit-Learn & TensorFlow



### Regression Models

We start to look at how Machine Learning algorithms work, starting with some regression models. We will cover:

- Linear Regression
- Gradient Descent
- Polynomial Regression
- Learning Curves
- Regularised Linear Model
- Logistic Regression

# Linear Regression

This is a **linear model** which makes a prediction by simply computing a weighted sum of the input features, plus a constant called the **bias** term (also called the **intercept** term),

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n,$$

where

- $\hat{y}$  is the predicted value;

and the known quantities are:

- $n$  is the number of features;
- $x_i$  is the  $i^{\text{th}}$  feature value; and
- $\theta_j$  is the  $j^{\text{th}}$  model parameter.

## Linear Regression: vectorized form of model

...Or in (column) vectors ...

$$\hat{y} = h_{\theta}(\mathbf{x}) = \theta^{\top} \mathbf{x},$$

where

- $\hat{y}$  is the predicted value;
- $\theta$  is the model's parameter vector, containing the bias term  $\theta_0$  and the feature weights  $\theta_1$  to  $\theta_n$ .
- $\mathbf{x}$  is the instance's feature vector, containing  $x_0$  to  $x_n$ , with  $x_0$  always equal to 1;
- $\theta^{\top} \mathbf{x}$  is the matrix multiplication of  $\theta^{\top}$  and  $\mathbf{x}$ ; and
- $h_{\theta}$  is the hypothesis function, using the model parameters  $\theta$ .

## Linear Regression: training the model

- Training a model means setting its parameters so that the model best fits the training set.
- Thus, we first need a measure of how well (or poorly) the model fits the training data. A common performance measure of a regression model is the *Root Mean Square Error* (*RMSE*).
- Therefore, to train a Linear Regression model, you need to find the value of  $\theta$  that minimizes the RMSE.
- In practice, it is simpler to minimize the *Mean Square Error* (*MSE*) than the RMSE. It leads to the same result (the value that minimizes a function also minimizes its square root).

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m \left( \theta^{\top} \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

## Linear Regression: the Normal Equation

To find the value of  $\theta$  that minimizes the cost function, there is a closed-form solution—in other words, a mathematical equation that gives the result directly. This is called the **Normal Equation**.

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

where

- $\mathbf{X}$  is a  $m \times (n + 1)$  matrix – the  $i^{\text{th}}$  row of  $\mathbf{X}$  is the training instance  $\mathbf{x}^{(i)}$ ;
- $\hat{\theta}$  is the value that minimizes the cost function;
- $\mathbf{y}$  is the vector of target values containing  $y^{(1)}$  to  $y^{(m)}$ .

Once the optimal model parameter  $\hat{\theta}$  is known, given a new test instance  $\mathbf{x}_{\text{test}}$ , we can predict  $y_{\text{test}}$  using the formula:

$$y_{\text{test}} = \hat{\theta}^\top \mathbf{x}_{\text{test}}.$$



## A random dataset to use as an example

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

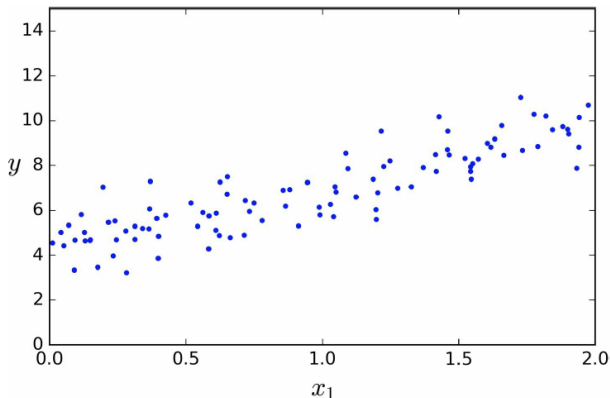


Figure 4-1. Randomly generated linear dataset

## Example: Fitting the model

Now let's compute  $\hat{\theta}$  using the Normal Equation.

```
X_b = np.c_[np.ones((100, 1)), X]    # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

```
theta_best
array([[4.21509616],
       [2.77011339]])
```

# Making Predictions

Just using basic numpy python:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
>>> y_predict = X_new_b @ theta_best
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Let's plot this model's predictions:

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

# Plot of Predictions

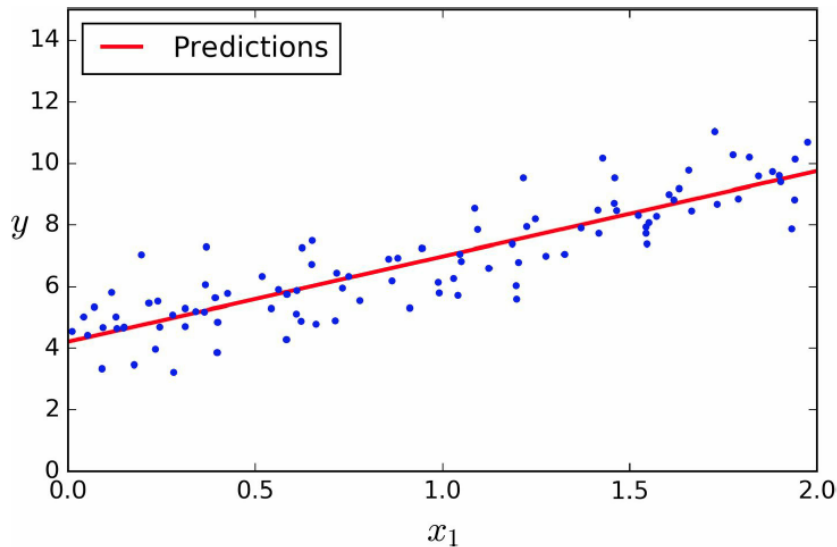


Figure 4-2. Linear Regression model predictions

The equivalent code using Scikit-Learn looks like this:

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

**Beware:** running time of normal equation can be cubic in number of features but linear with respect to the number of instances in the training set. Predictions are fast.

# Gradient Descent

Gradient Descent is a very generic optimization algorithm for which the general idea is to tweak parameters iteratively in order to minimize a cost function.

It measures the local gradient of the error function with regards to the parameter vector  $\theta$ , and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

It starts by filling  $\theta$  with random values (this is called **random initialization**), and then improve it gradually, taking one step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum.

## Partial Derivatives and the Gradient Vector

To implement Gradient Descent, we need to compute the gradient of the cost function with regards to each model parameter  $\theta_j$ . That is, we need to calculate how much the cost function will change if you change  $\theta_j$  just a little bit. This is called a *partial derivative*.

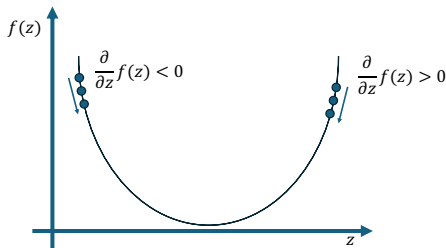
$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m \left( \theta^\top \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

# Partial Derivatives and the Gradient Vector

The idea:

when the  $\frac{\partial}{\partial z} f(z)$  is negative at  $z^{(t)}$ , then  $z^{(t+1)}$  need to be  $z^{(t)} + \text{"something"}$  to go in the direction of the minimum of  $f(z)$ ;

when the  $\frac{\partial}{\partial z} f(z)$  is positive at  $z^{(t)}$ , then  $z^{(t+1)}$  need to be  $z^{(t)} - \text{"something"}$  to go in the direction of the minimum of  $f(z)$ ;





## Partial Derivatives and the Gradient Vector

The gradient vector contains all the partial derivatives of the cost function:

$$\text{grad}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^{\top} (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \text{grad}(\boldsymbol{\theta}^{(t)}),$$

where  $\boldsymbol{\theta}^{(t)}$  is the value of  $\boldsymbol{\theta}$  at the previous step, and  $\boldsymbol{\theta}^{(t+1)}$  is the new value of  $\boldsymbol{\theta}$  (next step).

# Gradient Descent

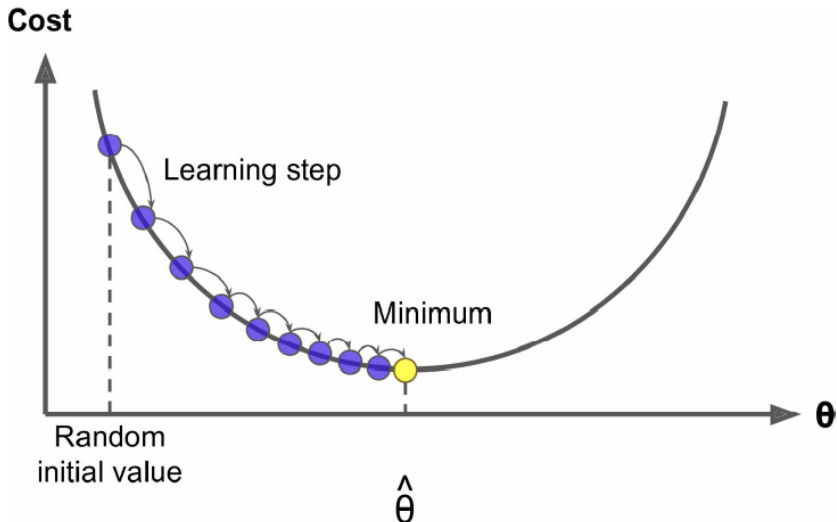


Figure 4-3. Gradient Descent

# Learning Rate

An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter ( $\eta$ ).

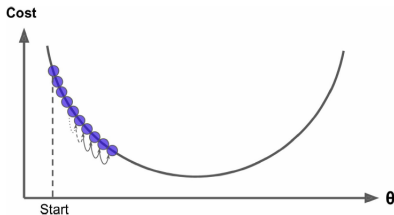


Figure 4-4. Learning rate too small

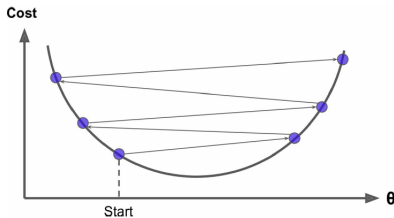


Figure 4-5. Learning rate too large

## Gradient Descent: pitfalls

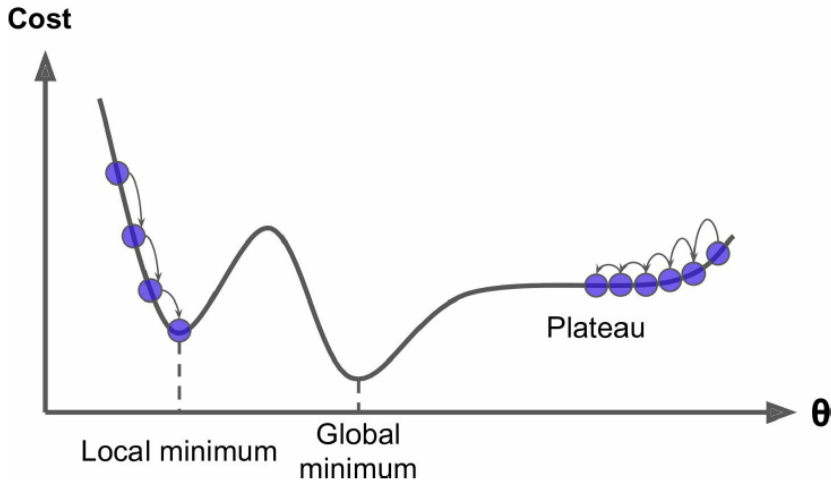


Figure 4-6. Gradient Descent pitfalls

# Gradient Descent for a Regression with an MSE cost function

Will be a bowl shape (good) but may be elongated if there are different scales.

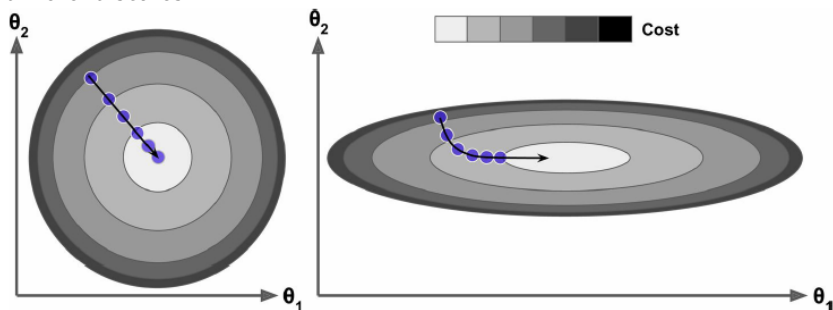


Figure 4-7. Gradient Descent with and without feature scaling

## GD – implementation

```
eta = 0.1 # learning rate
n_epochs = 1000
m = len(X_b) # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1) # randomly initialized model parameters

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients

>>> theta
array([[4.21509616],
       [2.77011339]])
```

## GD with various learning rates

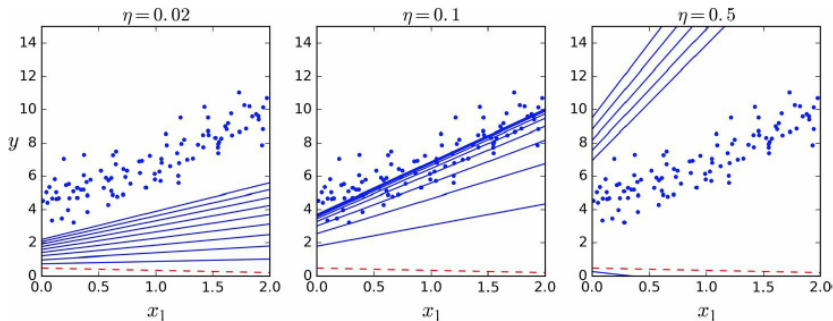
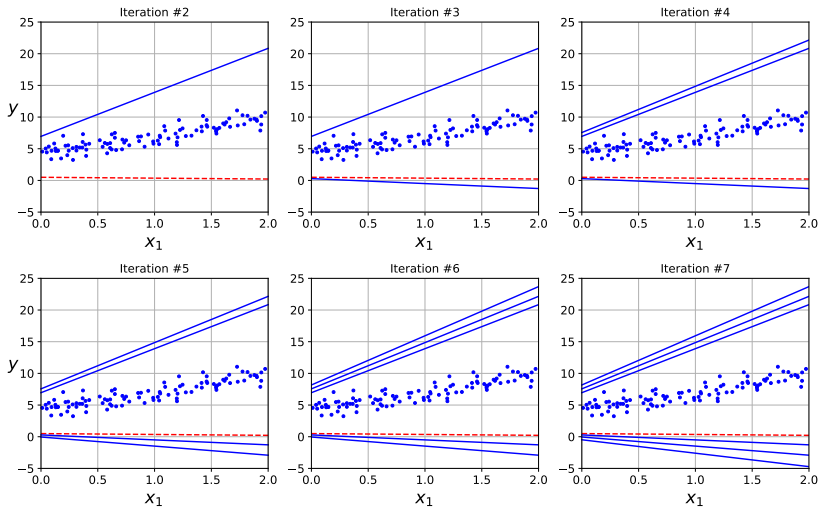


Figure 4-8. Gradient Descent with various learning rates

# GD with various learning rates

A divergence example -- when the learning rate is too high ( $\eta = 0.5$  in this plot)





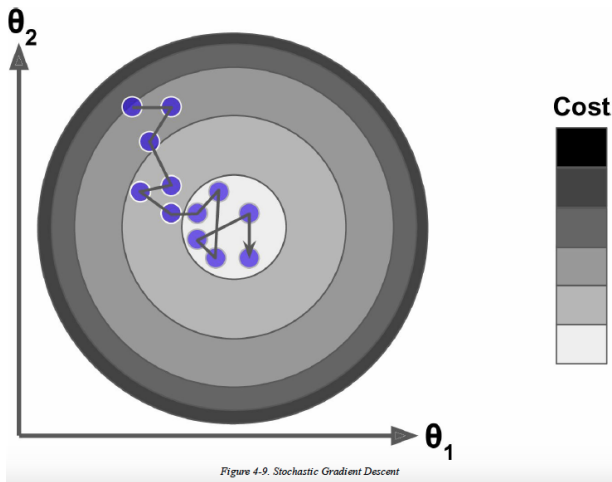
## 3 Ways to do Gradient Descent

**Batch GD:** Uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.

**Stochastic GD:** Just picks a random instance in the training set at every step and computes the gradients based only on that single instance. Makes the algorithm much faster since it has very little data to manipulate at every iteration. Also possible to train on huge training sets.

**Mini-batch GD:** At each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), computes the gradients on small random sets of instances (*mini-batches*). Get a performance boost from hardware optimization of matrix operations (cf GPUs).

## Stochastic Gradient Descent (SGD) jumps around:



One option is to gradually reduce the learning rate to allow the algorithm to settle to the global minimum.

# Stochastic Descent

## Reducing the learning rate gradually

```
n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

np.random.seed(42)
theta = np.random.randn(2, 1) # random initialization

for epoch in range(n_epochs):
    for iteration in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index : random_index + 1]
        yi = y[random_index : random_index + 1]
        gradients = 2 * xi.T @ (xi @ theta - yi)
        eta = learning_schedule(epoch * m + iteration)
        theta = theta - eta * gradients
```

(*m* is the size of the training set)

# Stochastic Descent in Action

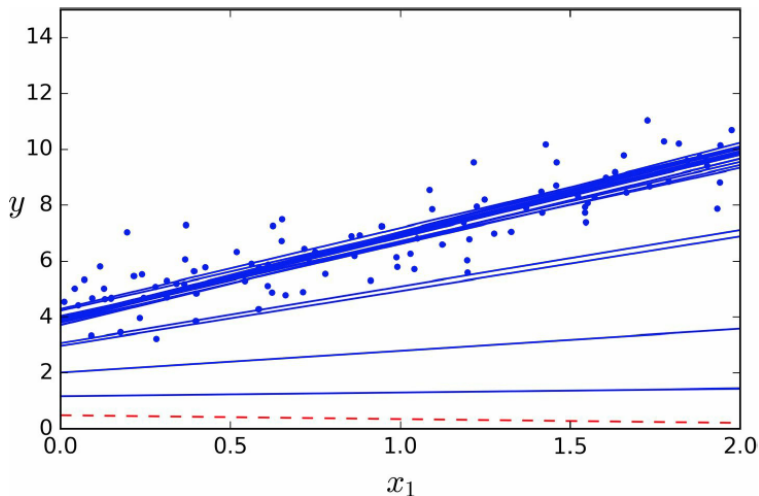


Figure 4-10. Stochastic Gradient Descent first 10 steps

# Paths in Parameter Space

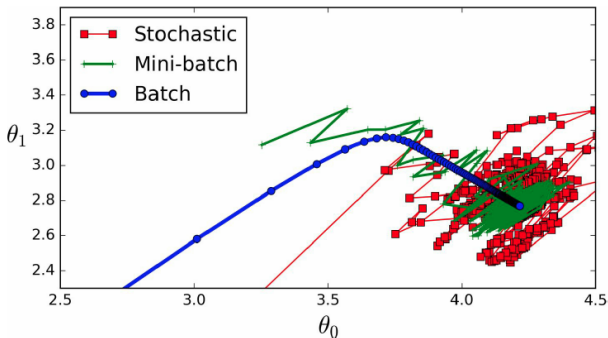


Figure 4-11. Gradient Descent paths in parameter space

# Polynomial Regression

What if your data is actually more complex than a straight line?

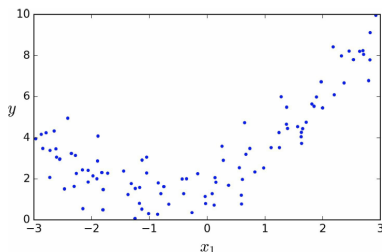


Figure 4-12. Generated nonlinear and noisy dataset

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 +
    np.random.randn(m, 1)
```

In the above example,  $y = 0.5x_1^2 + x_1 + 2 + \text{Gaussian noise}$ .

... We can still use linear regression

But introduce a new feature which is  $x_1^2$ .

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features =
    PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929, 0.56664654])

>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.781]), array([[0.933, 0.564]]))
```

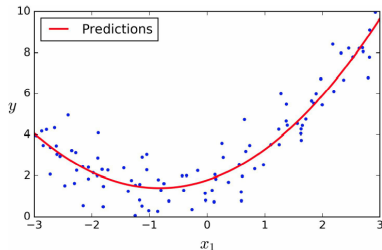


Figure 4-13. Polynomial Regression model predictions

The fitted curve (red) is:  $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78 = 0$ .

See Scikit-Learn's [PolynomialFeatures](#) class.

...And go to higher polynomial degrees

A polynomial of a large degree can fit data better

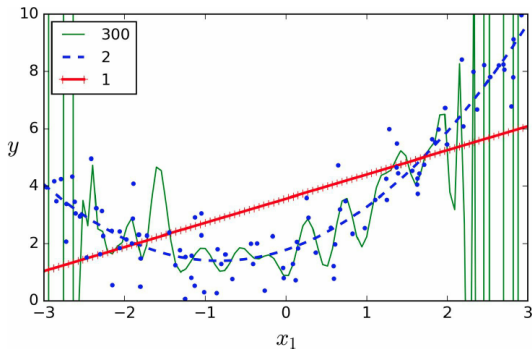


Figure 4-14. High-degree Polynomial Regression



# Learning Curves

In Chapter 2 you used [cross-validation](#) to get an estimate of a model's generalization performance.

- If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting.
- If it performs poorly on both, then it is underfitting.

This is one way to tell when a model is too complex or too simple.

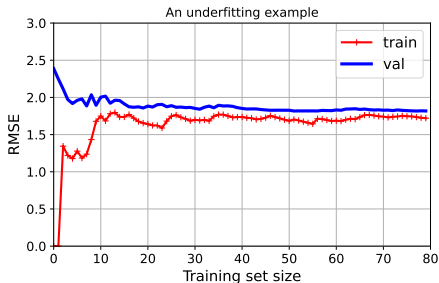
Another way is to look at the [learning curves](#).

# Learning Curves

**Learning Curves** are plots of the model's performance on the training set and the validation set **as a function of the training set size**. To generate the plots, train the model several times on different sized subsets of the training set.

See the code in the text book for `plot_learning_curves(model, X, y)`.

## Plot of learning curves: plain Linear Regression model



(Simulated data is:

$$y = 0.5 * X^2 + X + 2 + \text{noise}$$

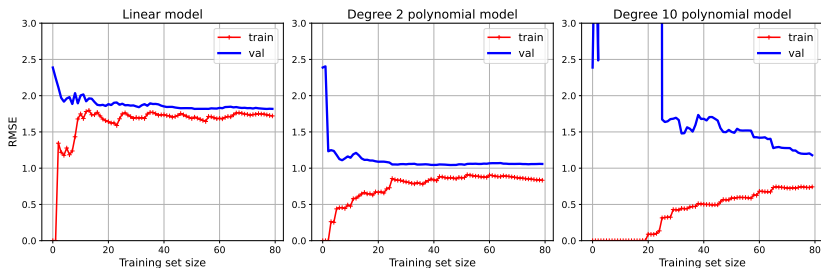
Try to fit a linear model)

Both curves reach a plateau; they are close and fairly high<sup>1</sup>. A clear under-fitting model.

---

<sup>1</sup>Note that the vertical axis is RMSE so the errors for both curves are fairly high

# Plots of learning curves



The simulated data is:  $y = 0.5 * X^2 + X + 2 + \text{noise}$

# Logistic Regression

Some regression algorithms can be used for classification as well (and vice versa).

*Logistic Regression* (also called *Logit Regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the **positive class**, labelled “1”), or else it predicts that it does not (i.e., it belongs to the **negative class**, labelled “0”). This makes it a *binary classifier*.

## How does a logistic model make a prediction?

Just like a *Linear Regression* model, a *Logistic Regression* model computes a weighted sum of the input features (plus a bias term), The probability  $\hat{p}$  estimated by the Logistic Regression model is given by:

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\theta^T \mathbf{x})$$

where the logistic function  $\sigma(t)$  is defined as follows:

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

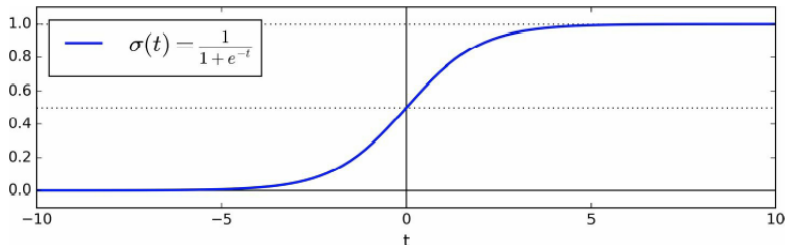


Figure 4-21. Logistic function

## How does a logistic model make a prediction?

The probability  $\hat{p}$  estimated by the Logistic Regression model is given by:

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^{\top} \mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^{\top} \mathbf{x})}$$

The Logistic Regression model prediction is:

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

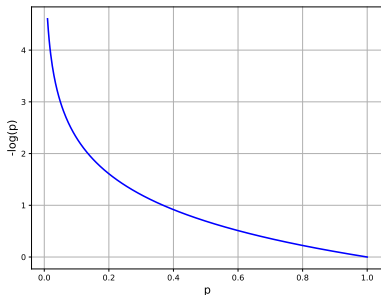
A Logistic Regression model predicts 1 if  $\boldsymbol{\theta}^{\top} \mathbf{x}$  is positive, and 0 if it is negative.

# Training and Cost Function

We want to find the parameter vector  $\theta$  so that the model gives high probabilities for instances in the positive class and low probabilities for instances in the negative class.

Cost function of a single instance:

$$C(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$



Therefore,  $\hat{p} \in [0, 1]$ .



## Training and Cost Function

$$C(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

**Example 1:**  $\hat{p} = 0.1$ :

If  $y = 1$ :

$$C(\theta) = -\log(0.1) = 2.30$$

If  $y = 0$ :

$$C(\theta) = -\log(1 - 0.1) = 0.10$$

**Example 2:**  $\hat{p} = 0.8$ :

If  $y = 1$ :

$$C(\theta) = -\log(0.8) = 0.22$$

If  $y = 0$ :

$$C(\theta) = -\log(1 - 0.8) = 1.6$$

# Training and Cost Function

The *Logistic Regression cost function* (*log loss*) is:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

- We want that the model estimates high probabilities for positive instances ( $y = 1$ ) and low probabilities for negative instances.
- Can do this using a simple log function of the probability.
- Average over all instances.
- No known closed-form equation to compute the value of  $\theta$  that minimizes this cost function.
- But cost function is convex, so GD is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough).

## Iris Example: three species

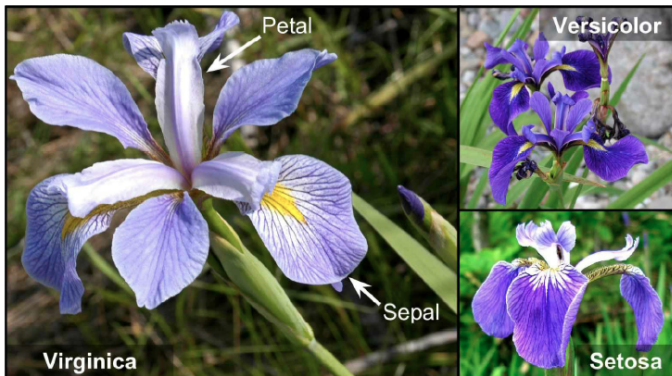


Figure 4-22. Flowers of three iris plant species<sup>16</sup>

Some flower properties vary with the species.

A **3-class classification** problem, with 4 attributes (all in cm): *sepal length*, *sepal width*, *petal length*, and *petal width*.

## Just using petal length feature

Suppose that we want to do a binary classification: to classify *Iris-Virginica* type versus *not Iris-Virginica* type, using just the *petal width* feature. After training a Logistic Regression classifier, we get a *decision boundary* at about 1.6cm.

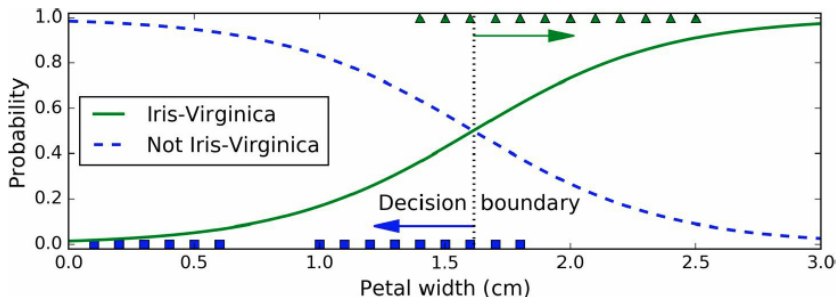


Figure 4-23. Estimated probabilities and decision boundary  
green triangles: *Iris-Virginica*; blue squares: other iris species

## With two features...

Again, we use the Logistic Regression classifier but on two features: *petal length* and *petal width*.  
... we also get a linear boundary.

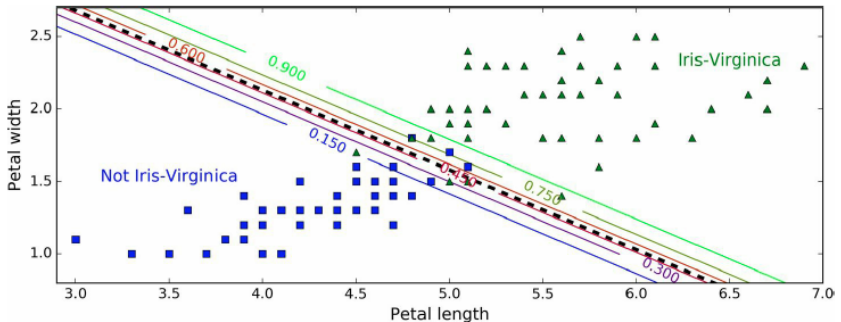


Figure 4-24. Linear decision boundary

The black dashed line represents the model's decision boundary (50% probability)

## For next week

Work through your lab sheet and attend the supervised lab.

Read up to Chapter 4 Training Models.

Have a good week.