Python Machine Learning
Equation Reference

Sebastian Raschka
mail@sebastianraschka.com

05/04/2015 (last updated: 06/19/2016)

Code Repository and Resources::
https://github.com/rasbt/python-machine-learning-book

@book{raschka2015python,
title={Python Machine Learning},
author={Raschka, Sebastian},
year={2015},
publisher={Packt Publishing} }

Chapter 1

Giving Computers the Ability to Learn from Data

The Iris dataset, consisting of 150 samples and 4 features, can then be written as a 150×4 matrix $\mathbf{X} \in \mathbb{R}^{150 \times 4}$:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & \dots & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & \dots & x_4^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & \dots & x_4^{(150)} \end{bmatrix}$$

For the rest of this book, unless noted otherwise, we will use the superscript (i) to refer to the ith training sample, and the subscript j to refer to the jth dimension of the training dataset.
We use lower-case, bold-face letters to refer to vectors ($\mathbf{x} \in \mathbb{R}^{n \times 1}$) and upper-case, bold-face letters to refer to matrices, respectively ($\mathbf{X} \in \mathbb{R}^{n \times m}$), where n refers to the number of rows, and m refers to the number of columns, respectively. To refer to single elements in a vector or matrix, we write the letters in italics $x^{(n)}$ or $x_m^{(n)}$, respectively. For example, x_1^{150} refers to the refers to the first dimension of the flower sample 150, the sepal length. Thus, each row in this feature matrix represents one flower instance and can be written as four-dimensional row vector $\mathbf{x}^{(i)} \in \mathbb{R}^{1 \times 4}$

$$\mathbf{x}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}.$$

Each feature dimension is a 150-dimensional column vector $\mathbf{x}_j \in \mathbb{R}^{150 \times 1}$, for example

$$\mathbf{x_j} = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}.$$

Similarly, we store the target variables (here: class labels) as a 150-dimensional column vector

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(150)} \end{bmatrix}, (y \in \{\text{Setosa, Versicolor, Virginica }\}).$$

1.7 A roadmap for building machine learning systems

1.7.1 Preprocessing – getting data into shape

1.7.2 Training and selecting a predictive model

1.7.3 Evaluating models and predicting unseen data instances

1.8 Using Python for machine learning

1.8.1 Installing Python packages

1.9 Summary

Chapter 2

Training Machine Learning Algorithms for Classification

2.1 Artificial neurons – a brief glimpse into the early history of machine learning

We can then define an activation function $\phi(z)$ that takes a linear combination of certain input values $\mathbf{x}$ and a corresponding weight vector $\mathbf{w}$ where z is the so-called net input ($z = w_1x_1 + \cdots + w_mx_m$):

$$\mathbf{w} = \begin{bmatrix} w^{(1)} \\ w^{(2)} \\ \vdots \\ w^{(m)} \end{bmatrix}, \mathbf{w} = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ \vdots \\ x^{(m)} \end{bmatrix}.$$

Now, if the activation of a particular sample $x^{(i)}$, that is, the output of $\phi(z)$, is greater than a defined threshold θ, we predict class 1 and class -1, otherwise. In the perceptron algorithm, the activation function $\phi(\cdot)$ is a simple *unit step function*, which is sometimes also called the *Heaviside step function*:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise .} \end{cases}$$

For simplicity, we can bring the threshold θ to the left side of the equation and define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$, so that we write $\mathbf{z}$ in a more compact form

$$z = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \mathbf{w^T x}$$

and

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise .} \end{cases}$$

In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in $\mathbf{x}$ and $\mathbf{w}$ using a *vector dot product*, whereas superscript T stands for *transpose*, which is an operation that transforms a column vector into a row vector and vice versa:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \mathbf{w^T x} = \sum_{j=0}^{m} \mathbf{w_j x_j} = \mathbf{w}^T \mathbf{x}.$$

For example:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32.$$

Furthermore, the transpose operation can also be applied to a matrix to reflect it over its diagonal, for example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Rosenblatt's initial perceptron rule is fairly simple and can be summarized by the following steps:

1. Initialize the weights to 0 or small random numbers.
2. For each training sample $\mathbf{x}^{(i)}$, perform the following steps:
   (a) Compute the output value $\hat{y}$.
   (b) Update the weights.

Here, the output value is the class label predicted by the unit step function that we de ned earlier, and the simultaneous update of each weight w_j in the weight vector $\mathbf{w}$ can be more formally written as:

$$w_j := w_j + \Delta w_j$$

The value of Δw_j, which is used to update the weight w_j, is calculated by the perceptron rule:

$$\Delta w_j = \eta \Big(y^{(i)} - \hat{y}^{(i)} \Big) x_j^{(i)}$$

Where η is the learning rate (a constant between 0.0 and 1.0), $y^{(i)}$ is the true class label of the ith training sample, and $\hat{y}^{(i)}$ is the predicted class label. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute $\hat{y}^{(i)}$ before all of the weights Δw_j were updated. Concretely, for a 2D dataset, we would write the update as follows:

$$\Delta w_0 = \eta\Big(y^{(i)} - \hat{y}^{(i)}\Big)$$

$$\Delta w_1 = \eta\Big(y^{(i)} - \hat{y}^{(i)}\Big)x_1^{(i)}$$

$$\Delta w_2 = \eta\Big(y^{(i)} - \hat{y}^{(i)}\Big)x_2^{(i)}$$

Before we implement the perceptron rule in Python, let us make a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged:

$$\Delta w_j = \eta\Big(-1 - -1\Big)x_j^{(i)} = 0$$

$$\Delta w_j = \eta\Big(1 - 1\Big)x_j^{(i)} = 0$$

However, in the case of a wrong prediction, the weights are being pushed towards the direction of the positive or negative target class, respectively:

$$\Delta w_j = \eta\Big(1 - -1\Big)x_j^{(i)} = \eta(2)x_j^{(i)}$$

$$\Delta w_j = \eta\Big(-1 - 1\Big)x_j^{(i)} = \eta(-2)x_j^{(i)}$$

To get a better intuition for the multiplicative factor $x_j^{(i)}$, let us go through another simple example, where:

$$y^{(i)} = +1, \quad \hat{y}^{(i)} = -1, \quad \eta = 1$$

Let's assume that $x_j^{(i)} = 0.5$ and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1 so that the activation $x_j^i \times w_j^{(i)}$ will be more positive the next time we encounter this sample and thus will be more likely to be above the threshold of the unit step function to classify the sample as $+1$:

$$\Delta w_j^{(i)} = (1 - -1)0.5 = (2)0.5 = 1$$

The weight update is proportional to the value of $x_j^{(i)}$. For example, if we have another sample $x_j^{(i)} = 2$ that is incorrectly classified as -1, we'd push the decision boundary by an even larger extent to classify this sample correctly the next time:

$$\Delta w_j^{(i)} = (1 - -1)2 = (2)2 = 4.$$

2.2 Implementing a perceptron learning algorithm in Python

2.2.1 Training a perceptron model on the Iris dataset

2.3 Adaptive linear neurons and the convergence of learning

The key difference between the Adaline rule (also known as the Widrow-Hoff rule) and Rosenblatt's perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function ϕz is simply the identity function of the net input so that

$$\phi\big(\mathbf{w}^T\mathbf{x}\big) = \mathbf{w}^T\mathbf{x}$$

2.3.1 Minimizing cost functions with gradient descent

One of the key ingredients of supervised machine learning algorithms is to define an objective function that is to be optimized during the learning process. This objective function is often a cost function that we want to minimize. In the case of Adaline, we can define the cost function $J(\cdot)$ to learn the weights as the Sum of Squared Errors (SSE) between the calculated outcomes and the true class labels

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big)\Big)^2.$$

Using gradient descent, we can now update the weights by taking a step away from the gradient $\nabla J(\mathbf{w})$ of our cost function $J(\cdot)$:

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}.$$

To compute the gradient of the cost function, we need to compute the partial derivative of the cost function with respect to each weight w_j,

$$\frac{\partial J}{\partial w_j} = -\sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big)\Big)x_j^{(i)},$$

so that we can write the update of weight w_j as

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big) x_j^{(i)}$$

Since we update all weights simultaneously, our Adaline learning rule becomes

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}.$$

For those who are familiar with calculus, the partial derivative of the SSE cost function with respect to the jth weight in can be obtained as follows:

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big)^2 \\
&= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big)^2 \\
&= \frac{1}{2} \sum_i 2\big(y^{(i)} - \phi(z^{(i)})\big) \frac{\partial}{\partial w_j} \Big(y^{(i)} - \phi(z^{(i)}) \Big) \\
&= \sum_i \big(y^{(i)} - \phi(z^{(i)})\big) \frac{\partial}{\partial w_j} \Big(y^{(i)} - \sum_i \big(w_j^{(i)} x_j^{(i)}\big) \Big) \\
&= \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big) \Big(- x_j^{(i)} \Big) \\
&= - \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big) x_j^{(i)}
\end{aligned}$$

Performing a matrix-vector multiplication is similar to calculating a vector dot product where each row in the matrix is treated as a single row vector. This vectorized approach represents a more compact notation and results in a more efficient computation using NumPy. For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

2.3.2 Implementing an Adaptive Linear Neuron in Python

Here, we will use a feature scaling method called standardization, which gives our data the property of a standard normal distribution. The mean of each feature is centered at value 0 and the feature column has a standard deviation of 1. For example, to standardize the jth feature, we simply need to subtract the sample mean μ_j from every training sample and divide it by its standard deviation σ_j:

$$\mathbf{x'}_j = \frac{\mathbf{x} - \mu_\mathbf{j}}{\sigma_j}.$$

Here $\mathbf{x}_j$ is a vector consisting of the jth feature values of all training samples n.

2.3.3 Large scale machine learning and stochastic gradient descent

A popular alternative to the batch gradient descent algorithm is stochastic gradient descent, sometimes also called iterative or on-line gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all samples $\mathbf{x}^{(i)}$:

$$\Delta \mathbf{w} = \eta \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big) \mathbf{x}^{(i)}.$$

We update the weights incrementally for each training sample:

$$\Delta \mathbf{w} = \eta \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big) \mathbf{x}^{(i)}.$$

2.4 Summary

Chapter 3

A Tour of Machine Learning Classifiers Using Scikit-learn

3.1 Choosing a classification algorithm

3.2 First steps with scikit-learn

3.2.1 Training a perceptron via scikit-learn

3.3 Modeling class probabilities via logistic regression

3.3.1 Logistic regression intuition and conditional probabilities

The odds ratio can be written as

$$\frac{p}{(1-p)},$$

where p stands for the probability of the positive (1? p) event. The term positive event does not necessarily mean good, but refers to the event that we want to predict, for example, the probability that a patient has a certain disease; we can think of the positive event as class label $y = 1$. We can then further define the logit function, which is simply the logarithm of the odds ratio (log-odds):

$$logit(p) = \log \frac{p}{1-p}$$

The logit function takes input values in the range 0 to 1 and transforms them to values over the entire real number range, which we can use to express a linear relationship between feature values and the log-odds:

$$logit(p(y = 1|\mathbf{x})) = w_0x_0 + w_1x_1 + \cdots + x_mw_m = \sum_{i=0}^{m} w_ix_i = \mathbf{w}^T\mathbf{x}.$$

Here, $p(y = 1|\mathbf{x})$ s the conditional probability that a particular sample belongs to class 1 given its features $\mathbf{x}$. Now what we are actually interested in is predicting the probability that a certain sample belongs to a particular class, which is the inverse form of the logit function. It is also called the logistic function, sometimes simply abbreviated as sigmoid function due to its characteristic S-shape

$$\phi(z) = \frac{1}{1 + e^{-z}}.$$

The output of the sigmoid function is then interpreted as the probability of particular sample belonging to class 1

$$\phi(z) = P(y = 1|\mathbf{x}; \mathbf{w})$$

given its features x parameterized by the weights w. For example, if we compute $\phi(z) = 0.8$ for a particular flower sample, it means that the chance that this sample is an Iris-Versicolor ower is 80 percent. Similarly, the probability that this ower is an Iris-Setosa ower can be calculated as $P(y = 0|\mathbf{x}; \mathbf{w}) = 1 - P(y = 1|\mathbf{x}; \mathbf{w}) = 0.2or20percent$. The predicted probability can then simply be converted into a binary outcome via a quantizer (unit step function):

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.5 \\ 0 & \text{otherwise .} \end{cases}$$

If we look at the preceding sigmoid plot, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & \text{if } \phi(z) \geq 0.0 \\ 0 & \text{otherwise .} \end{cases}$$

3.3.2 Learning the weights of the logistic cost function

In the previous chapter, we de ned the sum-squared-error cost function:

$$J(\mathbf{w}) = \frac{1}{2}\sum_i \left(\phi\big(z^{(i)}\big) - y^{(i)}\right)^2.$$

We minimized this in order to learn the weights w for our Adaline classification model. To explain how we can derive the cost function for logistic regression, let's first define the likelihood L that we want to maximize when we build a

logistic regression model, assuming that the individual samples in our dataset are independent of one another. The formula is as follows:

$$L(\mathbf{w}) = P(\mathbf{y}|\mathbf{x};\mathbf{w}) = \prod_{i=1}^{n} P\big(y^{(i)}|x^{(i)};\mathbf{w}\big) = \prod_{i=1}^{n} \Big(\phi\big(z^{(i)}\big)\Big)^{y^{(i)}} \Big(1-\phi\big(z^{(i)}\big)\Big)^{1-y^{(i)}}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the log-likelihood function:

$$l(\mathbf{w}) = \log L(\mathbf{w}) = \sum_{i=1}^{n} \Big[y^{(i)} \log\Big(\phi\big(z^{(i)}\big)\Big) + \Big(1-y^{(i)}\Big) \log\Big(1-\phi\big(z^{i()}\big)\Big)\Big]$$

Firstly, applying the log function reduces the potential for numerical under ow, which can occur if the likelihoods are very small. Secondly, we can convert the product of factors into a summation of factors, which makes it easier to obtain the derivative of this function via the addition trick, as you may remember from calculus.
Now we could use an optimization algorithm such as gradient ascent to maximize this log-likelihood function. Alternatively, let's rewrite the log-likelihood as a cost function $J(\cdot)$ that can be minimized using gradient descent as in *Chapter 2, Training Machine Learning Algorithms for Classification*:

$$J(\mathbf{w}) = \sum_{i=1}^{n} \Big[-y^{(i)} \log\Big(\phi\big(z^{(i)}\big)\Big) - \Big(1-y^{(i)}\Big) \log\Big(1-\phi\big(z^{i()}\big)\Big)\Big]$$

To get a better grasp on this cost function, let's take a look at the cost that we calculate for one single-sample instance:

$$J\big(\phi(z), y; \mathbf{w}\big) = -y \log\big(\phi(z)\big) - (1-y) \log\big(1-\phi(z)\big).$$

Looking at the preceding equation, we can see that the rst term becomes zero if $y = 0$, and the second term becomes zero if $y = 1$, respectively:

$$J\big(\phi(z), y; \mathbf{w}\big) = \begin{cases} -\log\big(\phi(z)\big) \text{ if } y = 1 \\ -\log\big(1-\phi(z)\big) \text{ if } y = 0 \end{cases}$$

3.3.3 Training a logistic regression model with scikit-learn

If we were to implement logistic regression ourselves, we could simply substitute the cost function $J(\cdot)$ in our Adaline implementation from *Chapter 2, Training Machine Learning Algorithms for Classification*, by the new cost function:

$$J(\mathbf{w}) = \sum_{i=1}^{n} \Big[-y^{(i)} \log\Big(\phi\big(z^{(i)}\big)\Big) - \Big(1-y^{(i)}\Big) \log\Big(1-\phi\big(z^{i()}\big)\Big)\Big]$$

We can show that the weight update in logistic regression via gradient descent is indeed equal to the equation that we used in Adaline in *Chapter 2, Training Machine Learning Algorithms for Classification*. Let's start by calculating the partial derivative of the log-likelihood function with respect to the jth weight:

$$\frac{\partial}{\partial w_j} l(\mathbf{w}) = \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

Before we continue, let's calculate the partial derivative of the sigmoid function first:

$$\frac{\partial}{\partial z} \phi(z) = \frac{\partial}{\partial z} \frac{1}{1+e^{-1}} \frac{1}{\big(1+e^{-z}\big)^2} e^{-z} = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}} \right)$$

$$= \phi(z)(1-\phi(z)).$$

Now we can resubstitute $\frac{\partial}{\partial z} \phi(z) = \phi(z)(1-\phi(z))$ in our first equation to obtain the following:

$$\begin{aligned}
&\left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z) \\
&= \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z) \big(1-\phi(z)\big) \frac{\partial}{\partial w_j} z \\
&= \Big(y \big(1-\phi(z)\big) - (1-y) \phi(z) \Big) x_j \\
&= \big(y - \phi(z)\big) x_j
\end{aligned}$$

Remember that the goal is to find the weights that maximize the log-likelihood so that we would perform the update for each weight as follows:

$$w_j := w_j + \eta \sum_{i=1}^{n} \Big(y^{(i)} - \phi(z^{(i)}) \Big) x_j^{(i)}$$

Since we update all weights simultaneously, we can write the general update rule as follows:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$

We define $\Delta \mathbf{w}$ as follows:

$$\Delta \mathbf{w} = \eta \nabla l(\mathbf{w})$$

Since maximizing the log-likelihood is equal to minimizing the cost function $J(\cdot)$ that we defined earlier, we can write the gradient descent update rule as follows:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^{n} \Big(y^{(i)} - \phi(z^{(i)}) \Big) x_j^{(i)}$$

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}, \; \Delta \mathbf{w} = -\eta \nabla J(\mathbf{w})$$

This is equal to the gradient descent rule in Adaline in *Chapter 2, Training Machine Learning Algorithms for Classification.*

3.3.4 Tackling overfitting via regularization

The most common form of regularization is the so-called L2 regularization (sometimes also called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2} \|\mathbf{w}\|^2 = \frac{\lambda}{2} \sum_{j=1}^{m} w_j^2$$

Here, λ is the so-called regularization parameter.
In order to apply regularization, we just need to add the regularization term to the cost function that we de ned for logistic regression to shrink the weights:

$$J(\mathbf{w}) = \sum_{i=1}^{n} \Big[- y^{(i)} \log \big(\phi(z^{(i)}) \big) - \big(1 - y^{(i)} \big) \log \big(1 - \phi(z^{(i)}) \big) \Big] + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Via the regularization parameter λ, we can then control how well we t the training data while keeping the weights small. By increasing the value of λ, we increase the regularization strength.
The parameter C that is implemented for the *LogisticRegression* class in scikit-learn comes from a convention in support vector machines, which will be the topic of the next section. C is directly related to the regularization parameter λ , which is its inverse:

$$C = \frac{1}{\lambda}$$

So, we can rewrite the regularized cost function of logistic regression as follows:

$$J(\mathbf{w}) = C \left[\sum_{i=1}^{n} \Big(- y^{(i)} \log \big(\phi(z^{(i)}) \big) - \big(1 - y^{(i)} \big) \Big) \log \Big(1 - \phi(z^{(i)}) \Big) \right] + \frac{1}{2} \|\mathbf{w}\|^2$$

3.4 Maximum margin classification with support vector machines

3.4.1 Maximum margin intuition

To get an intuition for the margin maximization, let's take a closer look at those *positive* and *negative* hyperplanes that are parallel to the decision boundary, which can be expressed as follows:

$$w_0 + \mathbf{w}^T \mathbf{x}_{pos} = 1 \quad (1)$$

$$w_0 + \mathbf{w}^T \mathbf{x}_{neg} = -1 \quad (2)$$

If we subtract those two linear equations (1) and (2) from each other, we get:

$$\Rightarrow \mathbf{w}^T \big(\mathbf{x}_{pos} - \mathbf{x}_{neg}\big) = 2$$

We can normalize this by the length of the vector $\mathbf{w}$, which is de ned as follows:

$$\|\mathbf{w}\| = \sqrt{\sum_{j=1}^{m} w_j^2}$$

So we arrive at the following equation:

$$\frac{\mathbf{w}^T(\mathbf{x}_{pos} - \mathbf{x}_{neg})}{\|\mathbf{w}\|} = \frac{2}{\|\mathbf{w}\|}$$

The left side of the preceding equation can then be interpreted as the distance between the positive and negative hyperplane, which is the so-called margin that we want to maximize.
Now the objective function of the SVM becomes the maximization of this margin by maximizing $\frac{2}{\|\mathbf{w}\|}$ under the constraint that the samples are classi ed correctly, which can be written as follows:

$$w_0 + \mathbf{w}^T \mathbf{x}^{(i)} \geq 1 \text{ if } y^{(i)} = 1$$

$$w_0 + \mathbf{w}^T \mathbf{x}^{(i)} < -1 \text{ if } y^{(i)} = -1$$

These two equations basically say that all negative samples should fall on one side of the negative hyperplane, whereas all the positive samples should fall behind the positive hyperplane. This can also be written more compactly as follows:

$$y^{(i)}\big(w_0 + \mathbf{w}^T \mathbf{x}^{(i)}\big) \geq 1 \quad \forall_i$$

In practice, though, it is easier to minimize the reciprocal term $\frac{1}{2}\|\mathbf{w}\|^2$, which can be solved by quadratic programming.

3.4.2 Dealing with the nonlinearly separable case using slack variables

The motivation for introducing the slack variable ξ was that the linear constraints need to be relaxed for nonlinearly separable data to allow convergence of the optimization in the presence of misclassifications under the appropriate cost penalization. The positive-values slack variable is simply added to the linear constraints:

$$\mathbf{w}^T\mathbf{x}^{(i)} \geq 1 - \xi^{(i)} \text{ if } y^{(i)} = 1$$

$$\mathbf{w}^T\mathbf{x}^{(i)} < -1 + \xi^{(i)} \text{ if } y^{(i)} = -1$$

So the new objective to be minimized (subject to the preceding constraints) becomes:

$$\frac{1}{2}\|\mathbf{w}\|^2 + C\Big(\sum_i \xi^{(i)}\Big)$$

3.4.3 Alternative implementations in scikit-learn

3.5 Solving nonlinear problems using a kernel SVM

As shown in the next figure, we can transform a two-dimensional dataset onto a new three-dimensional feature space where the classes become separable via the following projection:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

3.5.1 Using the kernel trick to find separating hyperplanes in higher dimensional space

To solve a nonlinear problem using an SVM, we transform the training data onto a higher dimensional feature space via a mapping function $\phi(\cdot)$ and train a linear SVM model to classify the data in this new feature space. Then we can use the same mapping function $\phi(\cdot)$ to transform new, unseen data to classify it using the linear SVM model.
However, one problem with this mapping approach is that the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data. This is where the so-called kernel trick comes into play. Although we didn't go into much detail about how to solve the quadratic programming task to train an SVM, in practice all we need is to replace the dot product

$$\mathbf{x}^{(i)\,T}\mathbf{x}^{(j)} \text{ by } \phi\big(\mathbf{x}^{(i)}\big)^T\phi\big(\mathbf{x}^{(j)}\big)$$

In order to save the expensive step of calculating this dot product between two points explicitly, we de define a so-called kernel function:

$$k\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \phi\big(\mathbf{x}^{(i)}\big)^T \phi\big(\mathbf{x}^{(j)}\big)$$

One of the most widely used kernels is the *Radial Basis Function kernel* (RBF kernel) or Gaussian kernel:

$$k\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \exp\left(- \frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2} \right)$$

This is often simplified to:

$$k\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \exp\left(- \gamma \, \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2 \right)$$

Here, $\gamma = \frac{1}{2\sigma^2}$ is a free parameter that is to be optimized.

3.6 Decision tree learning

In order to split the nodes at the most informative features, we need to define an objective function that we want to optimize via the tree learning algorithm. Here, our objective function is to maximize the information gain at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Here, f is the feature to perform the split, D_p and D_j are the dataset of the parent p and jth child node; I is our impurity measure, N_p is the total number of samples at the parent node, and N_j is the number of samples at the jth child node. As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities?the lower the impurity of the child nodes, the larger the information gain. However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that each parent node is split into two child nodes, D_{left} and D_{right}:

$$IG(D_p, f) = 1(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Now, the three impurity measures or splitting criteria that are commonly used in binary decision trees are *Gini impurity* (I_G), *Entropy* (I_H) and the *classification error* (I_E). Let's start with the definition of Entropy for all non-empty classes $p(i|t) \neq 0$:

$$I_H(t) = - \sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

Here, $p(i|t)$ is the proportion of the samples that belongs to class i for a particular node t. The entropy is therefore 0 if all samples at a node belong to the same class, and the entropy is maximal if we have a uniform class distribution. For example, in a binary class setting, the entropy is 0 if $p(i = 1|t) = 1$ or $p(i = 0|t) = 0$. If the classes are distributed uniformly with $p(i = 1|t) = 0.5$ and $p(i = 0|t) = 0.5$, the entropy is 1. Therefore, we can say that the entropy criterion attempts to maximize the mutual information in the tree.
Intuitively, the Gini impurity can be understood as a criterion to minimize the probability of misclassification:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Similar to entropy, the Gini impurity is maximal if the classes are perfectly mixed, for example, in a binary class setting ($c = 2$):

$$1 - \sum_{i=1}^{c} 0.5^2 = 0.5.$$

...
Another impurity measure is the classification error:

$$I_E = 1 - max\{p(i|t)\}$$

3.6.1 Maximizing information gain – getting the most bang for the buck

3.6.2 Building a decision tree

3.6.3 Combining weak to strong learners via random forests

3.7 K-nearest neighbors – a lazy learning algorithm

The *minkowski* distance that we used in the previous code example is just a generalization of the Euclidean and Manhattan distances that can be written as follows:

$$d\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \sqrt[p]{\sum_k \big|x_k^{(i)} - x_k^{(j)}\big|^p}$$

3.8 Summary

Chapter 4

Building Good Training Sets – Data Pre-Processing

4.1 Dealing with missing data

4.1.1 Eliminating samples or features with missing values

4.1.2 Imputing missing values

4.1.3 Understanding the scikit-learn estimator API

4.2 Handling categorical data

4.2.1 Mapping ordinal features

4.2.2 Encoding class labels

4.2.3 Performing one-hot encoding on nominal features

4.3 Partitioning a dataset in training and test sets

4.4 Bringing features onto the same scale

Now, there are two common approaches to bringing different features onto the same scale: *normalization* and *standardization*. Those terms are often used quite loosely in different fields, and the meaning has to be derived from the context. Most often, *normalization* refers to the rescaling of the features to a range of [0, 1], which is a special case of min-max scaling. To normalize our data, we can simply apply the min-max scaling to each feature column, where the new value $x^{(i)}_{norm}$ of a sample $x^{(i)}$:

$$x^{(i)}_{norm} = \frac{x^{(i)} - \mathbf{x}_{min}}{\mathbf{x}_{max} - \mathbf{x}_{min}}$$

Here, $x^{(i)}$ is a particular sample, x_{min} is the smallest value in a feature column, and x_{max} the largest value, respectively.
[...] Furthermore, standardization maintains useful information about outliers and makes the algorithm less sensitive to them in contrast to min-max scaling, which scales the data to a limited range of values.
The procedure of standardization can be expressed by the following equation:

$$x^{(i)}_{std} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Here, μ_x is the sample mean of a particular feature column and σ_x the corresponding standard deviation, respectively.

4.5 Selecting meaningful features

4.5.1 Sparse solutions with L1 regularization

We recall from *Chapter 3, A Tour of Machine Learning Classfiers Using Scikit-learn*, that L2 regularization is one approach to reduce the complexity of a model by penalizing large individual weights, where we de ned the L2 norm of our weight vector w as follows:

$$L2 : \|\mathbf{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$$

Another approach to reduce the model complexity is the related *L1 regularization*:

$$L1 : \|\mathbf{w}\|_1 = \sum_{j=1}^{m} |w_j|$$

4.5.2 Sequential feature selection algorithms

Based on the preceding definition of SBS, we can outline the algorithm in 4 simple steps:

1. Initialize the algorithm with $k = d$, where d is the dimensionality of the full feature space $\mathbf{X}_d$
2. Determine the feature x^- that maximizes the criterion $x^- = \arg\max J(\mathbf{X_k} - x)$, where $x \in \mathbf{X}_k$.
3. Remove the feature x^- from the feature set: $\mathbf{X}_{k-l} := \mathbf{X}_k - x^-; \quad k := k-1$.
4. Terminate if k equals the number of desired features, if not, got to step 2.

4.6 Assessing feature importance with random forests

4.7 Summary

Chapter 5

Compressing Data via Dimensionality Reduction

5.1 Unsupervised dimensionality reduction via principal component analysis

When we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix $\mathbf{W}$ that allows us to map a sample vector $\mathbf{x}$ onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space:

$$\mathbf{x} = [x_1, x_2, \ldots, x_j], \mathbf{x} \in \mathbb{R}^4$$

$$\downarrow \mathbf{x}\mathbf{W}, \quad \mathbf{W} \in \mathbb{R}^{d \times k}$$

$$\mathbf{z} = [z_1, z_2, \ldots, z_k], \quad \mathbf{z} \in \mathbb{R}^4$$

As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically $k << d$), the rst principal component will have the largest possible variance, and all consequent principal components will have the largest possible variance given that they are uncorrelated (orthogonal) to the other principal components. Note that the PCA directions are highly sensitive to data scaling, and we need to standardize the features prior to PCA if the features were measured on different scales and we want to assign equal importance to all features.
Before looking at the PCA algorithm for dimensionality reduction in more detail, let's summarize the approach in a few simple steps:

1. Standardize the d-dimensional dataset.
2. Construct the covariance matrix.

3. Decompose the covariance matrix into its eigenvectors and eigenvalues.
4. Select k eigenvectors that correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
5. Construct a projection matrix $\mathbf{W}$ from the "top" k eigenvectors.
6. Transform the d-dimensional input dataset $\mathbf{X}$ using the projection matrix $\mathbf{W}$ to obtain the new k-dimensional feature subspace.

5.1.1 Total and explained variance

After completing the mandatory preprocessing steps by executing the preceding code, let's advance to the second step: constructing the covariance matrix. The symmetric $d \times d$ -dimensional covariance matrix, where d is the number of dimensions in the dataset, stores the pairwise covariances between the different features. For example, the covariance between two features $\mathbf{x}_j$ and $\mathbf{x}_k$ on the population level can be calculated via the following equation:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^{n} \big(x_j^{(i)} - \mu_j\big)\big(x_k^{(i)} - \mu_k\big)$$

Here, μ_j and μ_k are the sample means of feature j and k, respectively. [...] For example, a covariance matrix of three features can then be written as

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

[...] an eigenvector $\mathbf{v}$ satisfies the following condition:

$$\Sigma\mathbf{v} = \lambda\mathbf{v}$$

Here, λ is a scalar: the eigenvector.
...
The variance explained ratio of an eigenvalue λ_j is simply the fraction of an eigenvalue λ_j and the total sum of the eigenvalues:

$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$$

5.1.2 Feature transformation

Using the projection matrix, we can now transform a sample $\mathbf{x}$ onto the PCA subspace obtaining $\mathbf{x}'$, a now two-dimensional sample vector consisting of two new features:

$$\mathbf{x}' = \mathbf{x}\mathbf{W}$$

5.1.3 Principal component analysis in scikit-learn

5.2 Supervised data compression via linear discriminant analysis

Before we take a look into the inner workings of LDA in the following subsections, let's summarize the key steps of the LDA approach:

1. Standardize the d-dimensional dataset (d is the number of features).
2. For each class, compute the d dimensional mean vector.
3. Construct the between-class scatter matrix $\mathbf{S}_B$ and the within-class scatter matrix $\mathbf{S}_W$.
4. Compute the eigenvectors and corresponding eigenvalues of the matrix $\mathbf{S}_W^{-1}\mathbf{S}_B$.
5. Choose the k eigenvectors that correspond to the k largest eigenvalues to construct a $d \times k$-dimensional transformation matrix $\mathbf{W}$; the eigenvectors are the columns of this matrix.
6. Project the samples onto the new feature subspace using the transformation matrix $\mathbf{W}$.

5.2.1 Computing the scatter matrices

Each mean vector $\mathbf{m}_i$ stores the mean feature value μ_m with respect to the samples of class i:

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{x \in D_i}^{c} \mathbf{x}_m$$

This results in three mean vectors:

$$\mathbf{m}_i = \begin{bmatrix} \mu_{i,\text{alcohol}} \\ \mu_{i,\text{malic-acid}} \\ \mu_{i,\text{proline}} \end{bmatrix}, i \in \{1, 2, 3\}$$

Using the mean vectors, we can now compute the within-class scatter matrix $\mathbf{S}_W$

$$\mathbf{S}_W = \sum_{i=1}^{c} \mathbf{S}_i$$

This is calculated by summing up the individual scatter matrices Si of each individual class i:

$$\mathbf{S}_i = \sum_{x \in D_i}^{c} (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T$$

The assumption that we are making when we are computing the scatter matrices is that the class labels in the training set are uniformly distributed. [...] Thus, we want to scale the individual scatter matrices $\mathbf{S}_i$ before we sum them up as scatter matrix $\mathbf{S}_W$ When we divide the scatter matrices by the number of class samples $\mathbf{N}_i$, we can see that computing the scatter matrix is in fact the same as computing the covariance matrix $\mathbf{Sigma}_i$ The covariance matrix is a normalized version of the scatter matrix:

$$\Sigma_i = \frac{1}{N_i}\mathbf{S}_W = \frac{1}{N_i} \sum_{x \in D_i}^{c} (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T$$

After we have computed the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute the between-class scatter matrix $\mathbf{S}_B$

$$\mathbf{S}_B = \sum_{i=1}^{c} N_i(\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$$

Here, $\mathbf{m}$ is the overall mean that is computed, including samples from all classes.

5.2.2 Selecting linear discriminants for the new feature subspace

5.2.3 Projecting samples onto the new feature space

$$\mathbf{X}' = \mathbf{XW}$$

5.2.4 LDA via scikit-learn

5.3 Using kernel principal component analysis for nonlinear mappings

5.3.1 Kernel functions and the kernel trick

To transform the samples $\mathbf{x} \in \mathbb{R}^d$ onto this higher k-dimensional subspace, we defined a nonlinear mapping function ϕ:

$$\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k >> d)$$

We can think of ϕ as a function that creates nonlinear combinations of the original features to map the original d-dimensional dataset onto a larger, k-dimensional feature space. For example, if we had feature vector $\mathbf{x} \in \mathbb{R}^d$ ($\mathbf{x}$ is a

column vector consisting of d features) with two dimensions ($d = 2$), a potential mapping onto a 3D space could be as follows:

$$\mathbf{x} = [x_1, x_2]^T$$

$$\downarrow \phi$$

$$\mathbf{z} = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]^T$$

[...] We computed the covariance between two features k and j as follows:

$$\sigma_{jk} = \frac{1}{n}\sum_{n}^{i=1}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

Since the standardizing of features centers them at mean zero, for instance, $\mu_j = 0$ and $\mu_k = 0$, we can simplify this equation as follows:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n} x_j^{(i)} x_k^{(i)}$$

Note that the preceding equation refers to the covariance between two features; now, let's write the general equation to calculate the covariance matrix Σ:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}^{(i)}\mathbf{x}^{(i)\,T}$$

Bernhard Scholkopf generalized this approach (B. Scholkopf, A. Smola, and K.-R. Muller. *Kernel Principal Component Analysis.* pages 583-588, 1997) so that we can replace the dot products between samples in the original feature space by the nonlinear feature combinations via ϕ:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} \phi\left(\mathbf{x}^{(i)}\right)\phi\left(\mathbf{x}^{(i)}\right)^T$$

To obtain the eigenvectors?the principal components?from this covariance matrix, we have to solve the following equation:

$$\begin{aligned}
&\Sigma\mathbf{v} = \lambda\mathbf{v} \\
&\Rightarrow \frac{1}{n}\sum_{i=1}^{n} \phi\left(\mathbf{x}^{(i)}\right)\phi\left(\mathbf{x}^{(i)}\right)^T\mathbf{v} = \lambda\mathbf{v} \\
&\Rightarrow \frac{1}{n\lambda}\sum_{i=1}^{n} \phi\left(\mathbf{x}^{(i)}\right)\left(\mathbf{x}^{(i)}\right)^T\mathbf{v} = \frac{1}{n}\sum_{(i=1)}^{n} \mathbf{a}^{(i)}\phi(\mathbf{x}^{(i)})
\end{aligned}$$

Here, λ and v are the eigenvalues and eigenvectors of the covariance matrix Σ, and $\mathbf{a}$ can be obtained by extracting the eigenvectors of the kernel (similarity) matrix $\mathbf{K}$ as we will see in the following paragraphs.
The derivation of the kernel matrix is as follows:

5.3.2 Implementing a kernel principal component analysis in Python

First, let's write the covariance matrix as in matrix notation, where $\phi(X)$ is an $n \times k$-dimensional matrix:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\phi\big(\mathbf{x}^{(i)}\big)\big(\mathbf{x}^{(i)}\big)^T = \frac{1}{n}\phi(\mathbf{X})^T\phi(\mathbf{X})$$

Now, we can write the eigenvector equation as follows:

$$mathbf{v} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{a}^{(i)}\phi(\mathbf{x}^{(i)}) = \lambda\phi(\mathbf{X})^T\mathbf{a}$$

Since $\Sigma\mathbf{v} = \lambda\mathbf{v}$, we get:

$$\frac{1}{n}\phi(\mathbf{X})^T\phi(\mathbf{X})\phi(\mathbf{X})^T\mathbf{a} = \lambda\phi(\mathbf{X})^T\mathbf{a}$$

Multiplying it by $\phi(\mathbf{X})$ on both sides yields the following result:

$$\begin{aligned}
&\frac{1}{n}\phi(\mathbf{X})\phi(\mathbf{X})^T\phi(\mathbf{X})\phi(\mathbf{X})^T\mathbf{a} = \lambda\phi(\mathbf{X})\phi(\mathbf{X})^T\mathbf{a}\\
&\Rightarrow \frac{1}{n}\phi(\mathbf{X})\phi(\mathbf{X})^T\mathbf{a} = \lambda\mathbf{a}\\
&\Rightarrow \frac{1}{n}\mathbf{K}\mathbf{a} = \lambda\mathbf{a}
\end{aligned}$$

Here, K is the similarity (kernel) matrix:

$$\mathbf{K} = \phi(\mathbf{X})\phi(\mathbf{X})^T$$

As we recall from the SVM section in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, we use the kernel trick to avoid calculating the pairwise dot products of the samples $\mathbf{x}$ under ϕ explicitly by using a kernel function $\kappa(\cdot)$ so that we don't need to calculate the eigenvectors explicitly:
[...] The most commonly used kernels are the following ones:

- The polynomial kernel:

  $$\kappa\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \big(\mathbf{x}^{(i)\,T}\mathbf{x}^{(j)} + \theta\big)^p$$

  Here, θ is the threshold and p is the power that has to be specified by the user.

- the hyperbolic tangent (sigmoid) kernel:

$$\kappa\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \tanh\big(\eta \mathbf{x}^{(i)\,T}\mathbf{x}^{(j)} + \theta\big)$$

- The *Radial Basis Function (RBF)* or Gaussian kernel that we will use in the following examples in the next subsection:

$$\kappa\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right),$$

  which is also often written as

$$\kappa\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \exp\big(-\gamma\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\big),$$

  where $\gamma = \frac{1}{2\sigma^2}$.

To summarize what we have discussed so far, we can define the following three steps to implement an RBF kernel PCA:

1. We compute the kernel (similarity) matrix $\mathbf{K}$, where we need to calculate the following:

$$\kappa\big(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}\big) = \exp\big(-\gamma\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\big)$$

   We do this for each pair of samples:

$$\mathbf{K} = \begin{bmatrix} \kappa\big(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}\big) & \kappa\big(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}\big) & \dots & \kappa\big(\mathbf{x}^{(1)}, \mathbf{x}^{(n)}\big) \\ \kappa\big(\mathbf{x}^{(2)}, \mathbf{x}^{(1)}\big) & \kappa\big(\mathbf{x}^{(2)}, \mathbf{x}^{(2)}\big) & \dots & \kappa\big(\mathbf{x}^{(2)}, \mathbf{x}^{(n)}\big) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa\big(\mathbf{x}^{(n)}, \mathbf{x}^{(1)}\big) & \kappa\big(\mathbf{x}^{(n)}, \mathbf{x}^{(2)}\big) & \dots & \kappa\big(\mathbf{x}^{(n)}, \mathbf{x}^{(n)}\big) \end{bmatrix}.$$

   For example, if our dataset contains 100 training samples, the symmetric kernel matrix of the pair-wise similarities would be 100×100 dimensional.

2. We center the kernel matrix $\mathbf{K}$ using the following equation:

$$\mathbf{K}' = \mathbf{K} - 1_n\mathbf{K} - \mathbf{K} - \mathbf{K}1_n + 1_n\mathbf{K}1_n$$

   Here, 1_n is an $n \times n$-dimensional matrix (the same dimensions as the kernel matrix) where all values are equal to $\frac{1}{n}$.

3. We collect the top k eigenvectors of the centered kernel matrix based on their corresponding eigenvalues, which are ranked by decreasing magnitude. In contrast to standard PCA, the eigenvectors are not the principal component axes but the samples projected onto those axes

Example 1 – separating half-moon shapes

Example 2 – separating concentric circles

5.3.3 Projecting new data points

[...] Thus, if we want to project a new sample $\mathbf{x}'$ onto this principal component axis, we'd need to compute the following:

$$\phi(\mathbf{x}')^T\mathbf{v}$$

Fortunately, we can use the kernel trick so that we don't have to calculate the projection $\phi(\mathbf{x}')^T\mathbf{v}$ explicitly. However, it is worth noting that kernel PCA, in contrast to standard PCA, is a memory-based method, which means that we have to reuse the original training set each time to project new samples. We have to calculate the pairwise RBF kernel (similarity) between each ith sample in the training dataset and the new sample $\mathbf{x}'$:

$$\phi(\mathbf{x}')^T\mathbf{v} = \sum_i \mathbf{a}^{(i)}\phi(\mathbf{x}')^T\phi(\mathbf{x}^{(\mathbf{i})})$$

$$= \sum_i \mathbf{a}^{(i)}k(\mathbf{x}', \mathbf{x}^{(i)})^T$$

Here, eigenvectors $\mathbf{a}$ and eigenvalues λ of the Kernel matrix $\mathbf{K}$ satisfy the following condition in the equation

$$\mathbf{Ka} = \lambda\mathbf{a}$$

5.3.4 Kernel principal component analysis in scikit-learn

5.4 Summary

Chapter 6

Learning Best Practices for Model Evaluation and Hyperparameter Tuning

the sum of all false predictions divided by the number of total predictions, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions, respectively:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

(TP = true positives, FP = false positives, TN = true negatives, FN = false negatives)
The prediction accuracy can then be calculated directly from the error:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

The true *positive rate* (TPR) and *false positive rate* (FPR) are performance metrics that are especially useful for imbalanced class problems:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Precision (PRE) and *recall* (REC) are performance metrics that are related to those true positive and true negative rates, and in fact, recall is synonymous to the true positive rate:

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In practice, often a combination of precision and recall is used, the so-called *F1-score*:

$$\text{F1} = 2 \times \frac{PRE \times REC}{PRE + REC}$$

6.5.3 Plotting a receiver operating characteristic

6.5.4 The scoring metrics for multiclass classification

he micro-average is calculated from the individual true positives, true negatives, false positives, and false negatives of the system. For example, the micro-average of the precision score in a k-class system can be calculated as follows:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

The macro-average is simply calculated as the average scores of the different systems:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

6.6 Summary

Chapter 7

Combining Different Models for Ensemble Learning

7.1 Learning with ensembles

To predict a class label via a simple majority or plurality voting, we combine the predicted class labels of each individual classifier C_j and select the class label $\hat{y}$ that received the most votes:

$$\hat{y} = mode\{C_1(\mathbf{x}), C_2(\mathbf{x}), \ldots, C_m(\mathbf{x})\}$$

For example, in a binary classification task where $class1 = -1$ and $class2 = +1$, we can write the majority vote prediction as follows:

$$C(\mathbf{x}) = sign\left[\sum_j^m C_j(\mathbf{x}\right] = \begin{cases} 1 & \text{if } \sum_j C_j(\mathbf{x}) \geq 0 \\ -1 & \text{otherwise .} \end{cases}$$

To illustrate why ensemble methods can work better than individual classifiers alone, let's apply the simple concepts of combinatorics. For the following example, we make the assumption that all n base classifiers for a binary classification task have an equal error rate ϵ. Furthermore, we assume that the classifiers are independent and the error rates are not correlated. Under those assumptions, we can simply express the error probability of an ensemble of base classifiers as a probability mass function of a binomial distribution:

$$P(y \geq k) = \sum_k^n \binom{n}{k} \epsilon^k (1-\epsilon)^{n-k} = \epsilon_{\text{ensemble}}$$

Here, $\binom{n}{k}$ is the binomial coefficient *n choose k*. In other words, we compute the probability that the prediction of the ensemble is wrong. Now let's take a look

at a more concrete example of 11 base classifiers ($n = 11$) with an error rate of 0.25 ($\epsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1-\epsilon)^{11-k} = 0.034$$

7.2 Implementing a simple majority vote classifier

Our goal is to build a stronger meta-classifier that balances out the individual classifiers' weaknesses on a particular dataset. In more precise mathematical terms, we can write the weighted majority vote as follows:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A \big(C_j(\mathbf{x}) = i\big)$$

Let's assume that we have an ensemble of three base classifiers $C_j (j \in 0, 1)$ and want to predict the class label of a given sample instance x. Two out of three base classi ers predict the class label 0, and one C_3 predicts that the sample belongs to class 1. If we weight the predictions of each base classifier equally, the majority vote will predict that the sample belongs to class 0:

$$C_1(\mathbf{x}) \rightarrow 0, C_2(\mathbf{x}) \rightarrow 0, C_3(\mathbf{x}) \rightarrow 1$$

$$\hat{y} = mode 0, 0, 1 = 0$$

Now let's assign a weight of 0.6 to C_3 and weight C_1 and C_2 by a coefficient of 0.2, respectively.

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A \big(C_j(\mathbf{x}) = i\big)$$

$$= \arg\max_i \big[0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1\big] = 1$$

More intuitively, since $3 \times 0.2 = 0.6$, we can say that the prediction made by C_3 has three times more weight than the predictions by C_1 or C_2 , respectively. We can write this as follows:

$$\hat{y} = mode\{0, 0, 1, 1, 1\} = 1$$

[...] The modified version of the majority vote for predicting class labels from probabilities can be written as follows:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

Here, p_{ij} is the predicted probability of the jth classifier for class label i.
To continue with our previous example, let's assume that we have a binary classification problem with class labels $i \in \{0, 1\}$ and an ensemble of three classifiers $C_j(j \in \{1, 2, 3\}$. Let's assume that the classifier C_j returns the following class membership probabilities for a particular sample $\mathbf{x}$:

$$C_1(\mathbf{x}) \rightarrow [0.9, 0.1], C_2(\mathbf{x}) \rightarrow [0.8, 0.2], C_3(\mathbf{x}) \rightarrow [0.4, 0.6]$$

We can then calculate the individual class probabilities as follows:

$$p(i_0|\mathbf{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1|\mathbf{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.06 = 0.42$$

$$\hat{y} = \arg\max_i \left[p(i_0|\mathbf{x}), p(i_1|\mathbf{x})\right] = 0$$

7.2.1 Combining different algorithms for classification with majority vote

7.3 Evaluating and tuning the ensemble classifier

7.4 Bagging – building an ensemble of classifiers from bootstrap samples

7.5 Leveraging weak learners via adaptive boosting

[...] The original boosting procedure is summarized in four key steps as follows:

1. Draw a random subset of training samples d_1 without replacement from the training set D to train a weak learner C_1.

2. Draw second random training subset d_2 without replacement from the training set and add 50 percent of the samples that were previously misclassified to train a weak learner C_2.

3. Find the training samples d_3 in the training set D on which C_1 and C_2 disagree to train a third weak learner C_3

4. Combine the weak learners C_1, C_2, and C_3 via majority voting.

[...] Now that have a better understanding behind the basic concept of AdaBoost, let's take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol ($\times$) and the dot product between two vectors by a dot symbol ($\cdot$), respectively. The steps are as follows:

1. Set weight vector $\mathbf{w}$ to uniform weights where $\sum_i w_i = 1$.
2. For j in m boosting rounds, do the following:
   (a) Train a weighted weak learner: $C_j = train(\mathbf{X}, \mathbf{y}, \mathbf{w})$.
   (b) Predict class labels: $\hat{y} = predict(C_j, \mathbf{X})$.
   (c) Compute the weighted error rate: $\epsilon = \mathbf{w} \cdot (\hat{\mathbf{y}} == \mathbf{y})$.
   (d) Compute the coefficient α_j: $\alpha_j = 0.5 \log \frac{1-\epsilon}{\epsilon}$.
   (e) Update the weights: $\mathbf{w} := \mathbf{w} \times \exp\big(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y}\big)$.
   (f) Normalize weights to sum to 1: $\mathbf{w} := \mathbf{w} / \sum_i w_i$.
3. Compute the final prediction: $\hat{\mathbf{y}} = \big(\sum_{j=1}^{m} \big(\alpha_j \times predict(C_j, \mathbf{X})\big) > 0\big)$.

Note that the expression $(\hat{\mathbf{y}} == \mathbf{y})$ in step 5 refers to a vector of 1s and 0s, where a 1 is assigned if the prediction is incorrect and 0 is assigned otherwise.

Sample indices	x	y	Weights	$\hat{y}(x \leq 3.0)$?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

Since the computation of the weight updates may look a little bit complicated at rst, we will now follow the calculation step by step. We start by computing the weighted error rate ϵ as described in step 5:

$$\epsilon = 0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times0+0.1\times1+0.1\times1+0.1\times1+0.1\times0$$

$$= \frac{3}{10} = 0.3$$

Next we compute the coefficient α_j (shown in step 6), which is later used in step 7 to update the weights as well as for the weights in majority vote prediction (step 10):

$$\alpha_j = 0.5 \log\left(\frac{1-\epsilon}{\epsilon}\right) \approx 0.424$$

After we have computed the coefficient α_j we can now update the weight vector using the following equation:

$$\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$$

Here, $\hat{\mathbf{y}} \times \mathbf{y}$ is an element-wise multiplication between the vectors of the predicted and true class labels, respectively. Thus, if a prediction $\hat{y}_i$ is correct, $\hat{y}_i \times y_i$ will have a positive sign so that we decrease the ith weight since α_j is a positive number as well:

$$0.1 \times \exp(-0.424 \times 1 \times 1) \approx 0.065$$

Similarly, we will increase the ith weight if $\hat{y}_i$ predicted the label incorrectly like this:

$$0.1 \times \exp(-0.424 \times 1 \times (-1)) \approx 0.153$$

Or like this:

$$0.1 \times \exp(-0.424 \times (-1) \times 1) \approx 0.153$$

After we update each weight in the weight vector, we normalize the weights so that they sum up to 1 (step 8):

$$\mathbf{w} := \frac{\mathbf{w}}{\sum_i w_i}$$

Here, $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$.
Thus, each weight that corresponds to a correctly classified sample will be reduced from the initial value of 0.1 to $0.065/0.914 \approx 0.071$ for the next round of boosting. Similarly, the weights of each incorrectly classified sample will increase from 0.1 to $0.153/0.914 \approx 0.167$.

7.6 Summary

Chapter 8

Applying Machine Learning to Sentiment Analysis

8.1 Obtaining the IMDb movie review dataset

8.2 Introducing the bag-of-words model

8.2.1 Transforming words into feature vectors

8.2.2 Assessing word relevancy via term frequency-inverse document frequency

The *tf-idf* can be de ned as the product of the *term frequency* and the *inverse document frequency*:

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t, d)$$

Here the $\text{tf}(t, d)$ is the term frequency that we introduced in the previous section, and the inverse document frequency $\text{idf}(t, d)$ can be calculated as:

$$\text{idf}(t, d) = \log \frac{n_d}{1 + \text{df}(d, t)},$$

where n_d is the total number of documents, and $\text{df}(d, t)$ is the number of documents d that contain the term t. Note that adding the constant 1 to the denominator is optional and serves the purpose of assigning a non-zero value to terms that occur in all training samples; the log is used to ensure that low document frequencies are not given too much weight.
However, if we'd manually calculated the tf-idfs of the individual terms in our feature vectors, we'd have noticed that the *TfidfTransformer* calculates the tf-idfs slightly differently compared to the standard textbook equations that we de ned earlier. The equations for the idf and tf-idf that were implemented in scikit-learn are:

$$\text{idf}(t, d) = \log \frac{1 + n_d}{1 + \text{df}(d, t)}$$

The tf-idf equation that was implemented in scikit-learn is as follows:

$$\text{tf-idf(t, d)} = \text{tf}(t, d) \times (\text{idf}(t, d) + 1).$$

While it is also more typical to normalize the raw term frequencies before calculating the tf-idfs, the *TfidfTransformer* normalizes the tf-idfs directly. By default (norm='l2'), scikit-learn's *TfidfTransformer* applies the L2-normalization, which returns a vector of length 1 by dividing an un-normalized feature vector $\mathbf{v}$ by its L2-norm:

$$\mathbf{v}_{norm} = \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \frac{\mathbf{v}}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}} = \frac{\mathbf{v}}{\left(\sum_{i=1}^{n} v_i^2\right)^{1/2}}$$

8.2.3 Cleaning text data

8.2.4 Processing documents into tokens

8.3 Training a logistic regression model for document classification

8.4 Working with bigger data - online algorithms and out-of-core learning

8.5 Summary

Chapter 9

Embedding a Machine Learning Model into a Web Application

Chapter 10

Predicting Continuous Target Variables with Regression Analysis

10.1 Introducing a simple linear regression model

The goal of simple (univariate) linear regression is to model the relationship between a single feature (explanatory variable x) and a continuous valued *response* (target variable y). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_0 + w_1 + x$$

Here, the weight w_0 represents the y axis intercepts and w_1 is the coefficient of the explanatory variable.
[...] The special case of one explanatory variable is also called *simple linear regression*, but of course we can also generalize the linear regression model to multiple explanatory variables. Hence, this process is called *multiple linear regression*:

$$y = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{i=0}^{m} w_i x_i = \mathbf{w}^T \mathbf{x}$$

Here, w_0 is the y-axis intercept with x_0 =1.

10.2 Exploring the Housing Dataset

10.2.1 Visualizing the important characteristics of a dataset

The correlation matrix is a square matrix that contains the Pearson product-moment correlation coeffcients (often abbreviated as Pearson's r), which mea-

sure the linear dependence between pairs of features. The correlation coefficients are bounded to the range -1 and 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r =?1$, respectively. As mentioned previously, Pearson's correlation coefficient can simply be calculated as the covariance between two features x and y (numerator) divided by the product of their standard deviations (denominator):

$$r = \frac{\sum_{i=1}^{n} \Big[\big(x^{(i)} - \mu_x\big)\big(y^{(i)} - \mu_y\big)\Big]}{\sqrt{\sum_{i=1}^{n} \big(x^{(i)} - \mu_x\big)^2}\sqrt{\sum_{i=1}^{n} \big(y^{(i)} - \mu_y\big)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Here, μ denotes the sample mean of the corresponding feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations, respectively.
We can show that the covariance between standardized features is in fact equal to their linear correlation coefficient. Let's first standardize the features x and y, to obtain their z-scores which we will denote as x' and y' , respectively:

$$x' = \frac{x - \mu_x}{\sigma_x},\ y' = \frac{y - \mu_y}{\sigma_y}$$

Remember that we calculate the (population) covariance between two features as follows:

$$\sigma_x y = \frac{1}{n} \sum_{i}^{n} \big(x^{(i)} - \mu_x\big)\big(y^{(i)} - \mu_y\big)$$

Since standardization centers a feature variable at mean 0, we can now calculate the covariance between the scaled features as follows:

$$\sigma'_{xy} = \frac{1}{n} \sum_{i}^{n} (x' - 0)(y' - 0)$$

Through resubstitution, we get the following result:

$$\frac{1}{n} \sum_{i}^{n} \Big(\frac{x - \mu_x}{\sigma_x}\Big)\Big(\frac{y - \mu_y}{\sigma_y}\Big)$$

$$= \frac{1}{n \cdot \sigma_x \sigma_y} \sum_{i}^{n} \sum_{i}^{n} \big(x^{(i)} - \mu_x\big)\big(y^{(i)} - \mu_y\big)$$

We can simplify it as follows:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

10.3 Implementing an ordinary least squares linear regression model

10.3.1 Solving regression for regression parameters with gradient descent

Consider our implementation of the *ADAptive LInear NEuron (Adaline)* from *Chapter 2, Training Machine Learning Algorithms for Classifcation*; we remember that the artificial neuron uses a linear activation function and we de ned a cost function $J(\cdot)$, which we minimized to learn the weights via optimization algorithms, such as *Gradient Descent (GD)* and *Stochastic Gradient Descent (SGD)*. This cost function in Adaline is the *Sum of Squared Errors (SSE)*. This is identical to the OLS cost function that we defined:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

Here, $\hat{y}$ is the predicted value $\hat{y} = \mathbf{w}^T\mathbf{x}$ (note that the term $1/2$ is just used for convenience to derive the update rule of GD). Essentially, OLS linear regression can be understood as Adaline without the unit step function so that we obtain continuous target values instead of the class labels -1 and 1.
[...] As an alternative to using machine learning libraries, there is also a closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{(-1)}\mathbf{X}^T\mathbf{y}$$

10.3.2 Estimating the coefficient of a regression model via scikit-learn

10.4 Fitting a robust regression model using RANSAC

10.5 Evaluating the performance of linear regression models

Another useful quantitative measure of a model's performance is the so-called *Mean Squared Error (MSE)*, which is simply the average value of the SSE cost function that we minimize to t the linear regression model. The MSE is useful to for comparing different regression models or for tuning their parameters via a grid search and cross-validation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

[...] Sometimes it may be more useful to report the coef cient of determination (R^2), which can be understood as a standardized version of the MSE, for better interpretability of the model performance. In other words, R^2 is the fraction of response variance that is captured by the model. The R^2 value is defined as follows:

$$R^2 = 1 - \frac{SSE}{SST}$$

Here, SSE is the sum of squared errors and SST is the total sum of squares $SST = \sum_{i=1}^{n} \left(y^{(i)} - \mu_y\right)^2$, or in other words, it is simply the variance of the response. Let's quickly show that R^2 is indeed just the rescaled version of the MSE:

$$R^2 = 1 - \frac{SSE}{SST}$$

$$= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\frac{1}{n}\sum_{i=1}^{n} \left(y^{(i)} - \mu_y\right)^2}$$

$$= 1 - \frac{MSE}{Var(y)}$$

For the training dataset, R^2 is bounded between 0 and 1, but it can become negative for the test set. If R^2 =1, the model fits the data perfectly with a corresponding $MSE = 0$.

10.6 Using regularized methods for regression

The most popular approaches to regularized linear regression are the so-called *Ridge Regression*, *Least Absolute Shrinkage and Selection Operator (LASSO)*, and the *Elastic Net* method.
Ridge regression is an L2 penalized model where we simply add the squared sum of the weights to our least-squares cost function:

$$J(\mathbf{w})_{ridge} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \|\mathbf{w}\|_2^2$$

Here:

$$\text{L2}: \quad \lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2$$

By increasing the value of the hyperparameter λ , we increase the regularization strength and shrink the weights of our model. Please note that we don't regularize the intercept term w_0.

An alternative approach that can lead to sparse models is the LASSO. Depending on the regularization strength, certain weights can become zero, which makes the LASSO also useful as a supervised feature selection technique:

$$J(\mathbf{w})_{LASSO} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda \|w\|_1$$

Here:

$$L1: \quad \lambda \|\mathbf{w}\|_1 = \lambda \sum_{j=1}^{m} |w_j|$$

However, a limitation of the LASSO is that it selects at most n variables if $m > n$. A compromise between Ridge regression and the LASSO is the Elastic Net, which has a L1 penalty to generate sparsity and a L2 penalty to overcome some of the limitations of the LASSO, such as the number of selected variables.

$$J(\mathbf{w})_{ElasticNet} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_1 \sum_{j=1}^{m} w_j^2 + \lambda_2 \sum_{j=1}^{m} |w_j|$$

10.7 Turning a linear regression model into a curve - polynomial regression

In the previous sections, we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

$$y = w_0 + w_1 x + w_2 x^2 x^2 + \cdots + w_d x^d,$$

where d denotes the degree of the polynomial.

10.7.1 Modeling nonlinear relationships in the Housing Dataset

10.7.2 Dealing with nonlinear relationships using random forests

Decision tree regression

When we used decision trees for classi cation, we de ned entropy as a measure of impurity to determine which feature split maximizes the *Information Gain (IG)*, which can be de ned as follows for a binary split:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

To use a decision tree for regression, we will replace entropy as the impurity measure of a node t by the MSE:

$$I(t) - MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} \left(y^{(i)} - \hat{y}_t \right)^2$$

Here, N_t is the number of training samples at node t, D_t is the training subset at node t, $y^{(i)}$ is the true target value, and $\hat{y}^{(i)}$ is the predicted target value (sample mean):

$$\hat{y}_t = \frac{1}{N} \sum_{i \in D_t} y^{(i)}$$

In the context of decision tree regression, the MSE is often also referred to as within-node variance, which is why the splitting criterion is also better known as *variance reduction*.

Random forest regression

10.8 Summary

Chapter 11

Working with Unlabeled Data – Clustering Analysis

11.1 Grouping objects by similarity using k-means

Thus, our goal is to group the samples based on their feature similarities, which we can be achieved using the k-means algorithm that can be summarized by the following four steps:

1. Randomly pick k centroids from the sample points as initial cluster centers.
2. Assign each sample to the nearest centroid $\mu^{(j)}, \quad j \in 1, ..., k$.
3. Move the centroids to the center of the samples that were assigned to it.
4. Repeat steps 2 and 3 until the cluster assignments do not change or a user-de ned tolerance or a maximum number of iterations is reached.

Now the next question is *how do we measure similarity between objects?* We can de ne similarity as the opposite of distance, and a commonly used distance for clustering samples with continuous features is the *squared Euclidean distance* between two points $\mathbf{x}$ and $\mathbf{y}$ in m-dimensional space:

$$d(\mathbf{x}, \mathbf{y})^2 = \sum_{j=1}^{m} \big(x_j - y_j\big)^2 = \|\mathbf{x} - \mathbf{y}\|_2^2.$$

Note that, in the preceding equation, the index j refers to the jth dimension (feature column) of the sample points x and y. In the rest of this section, we will use the superscripts i and j to refer to the sample index and cluster index, respectively.
Based on this Euclidean distance metric, we can describe the k-means algorithm as a simple optimization problem, an iterative approach for minimizing

the *within-cluster sum of squared errors (SSE)*, which is sometimes also called *cluster inertia*:

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\| \mathbf{x}^{(i)} - \mu^{(j)} \right\|_2^2$$

Here, $\mu^{(j)}$ is the representative point (centroid) for cluster j, and $w^{(i,j)} = 1$ if the sample $\mathbf{x}^{(i)}$ is in cluster j; $w^{(i,j)} = 0$ otherwise.

11.1.1 K-means++

[...] The initialization in k-means++ can be summarized as follows:

1. Initialize an empty set M to store the k centroids being selected.
2. Randomly choose the first centroid $\mu^{(j)}$ from the input samples and assign it to M
3. For each sample $\mathbf{x}^{(i)}$ that is not in M, find the minimum distance $d\big(x^{(i)}, M\big)^2$ to any of the centroids in M.
4. To randomly select the next centroid $\mu^{(p)}$, use a weighted probability distribution equal to $\frac{d(\mu^{(p)}, M)^2}{\sum_i d(x^{(i)} M)^2}$
5. Repeat steps 2 and 3 until k centroids are chosen.
6. Proceed with the classic k-means algorithm.

11.1.2 Hard versus soft clustering

The $fuzzyc-means(FCM)$ procedure is very similar to k-means. However, we replace the hard cluster assignment by probabilities for each point belonging to each cluster. In k-means, we could express the cluster membership of a sample x by a sparse vector of binary values:

$$\begin{bmatrix} \mu^{(1)} \rightarrow 0 \\ \mu^{(2)} \rightarrow 1 \\ \mu^{(3)} \rightarrow 0 \end{bmatrix}$$

Here, the index position with value 1 indicates the cluster centroid $\mu^{(j)}$ the sample is assigned to (assuming $k = 3,\ j \in \{1, 2, 3\}$). In contrast, a membership vector in FCM could be represented as follows:

$$\begin{bmatrix} \mu^{(1)} \rightarrow 0.1 \\ \mu^{(2)} \rightarrow 0.85 \\ \mu^{(3)} \rightarrow 0.05 \end{bmatrix}$$

Here, each value falls in the range $[0, 1]$ and represents a probability of membership to the respective cluster centroid. The sum of the memberships for a given

sample is equal to 1. Similarly to the k-means algorithm, we can summarize the FCM algorithm in four key steps:

1. Specify the number of k centroids and randomly assign the cluster memberships for each point.
2. Compute the cluster centroids $\mu^{(j)}, j \in \{1, \ldots, k\}$.
3. Update the cluster memberships for each point.
4. Repeat steps 2 and 3 until the membership coefficients do not change or a user-de ned tolerance or a maximum number of iterations is reached.

The objective function of FCM – we abbreviate it by J_m – looks very similar to the within cluster sum-squared-error that we minimize in k-means:

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{m(i,j)} \left\| \mathbf{x}^{(i)} - \mu^{(j)} \right\|_2^2 \; m \in [1, \infty)$$

However, note that the membership indicator $w^{(i,j)}$ is not a binary value as in k-means $\left(w^{(i,j)} \in \{0, 1\}\right)$ but a real value that denotes the cluster membership probability $\left(w^{(i,j)} \in [0, 1]\right)$. You also may have noticed that we added an additional exponent to $w^{(i,j)}$; the exponent m, any number greater or equal to 1 (typically $m = 2$), is the so-called *fuzziness coefficient* (or simply *fuzzifier*) that controls the degree of *fuzziness*. The larger the value of m, the smaller the cluster membership $w^{(i,j)}$ becomes, which leads to fuzzier clusters. The cluster membership probability itself is calculated as follows:

$$w^{(i,j)} = \left[\sum_{p=1}^{k} \left(\frac{\|\mathbf{x}^{(i)} - \mu^{(j)}\|_2}{\|\mathbf{x}^{(i)} - \mu^{(p)}\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

For example, if we chose three cluster centers as in the previous k-means example, we could calculate the membership of the $\mathbf{x}^{(i)}$ sample belonging to its own cluster:

$$w^{(i,j)} = \left[\sum_{p=1}^{k} \left(\frac{\|\mathbf{x}^{(i)} - \mu^{(j)}\|_2}{\|\mathbf{x}^{(i)} - \mu^{(1)}\|_2} \right)^{\frac{2}{m-1}} + \sum_{p=1}^{k} \left(\frac{\|\mathbf{x}^{(i)} - \mu^{(j)}\|_2}{\|\mathbf{x}^{(i)} - \mu^{(2)}\|_2} \right)^{\frac{2}{m-1}} + \sum_{p=1}^{k} \left(\frac{\|\mathbf{x}^{(i)} - \mu^{(j)}\|_2}{\|\mathbf{x}^{(i)} - \mu^{(3)}\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

The center $\mu^{(j)}$ of a cluster itself is calculated as the mean of all samples in the cluster weighted by the membership degree of belonging to its own cluster:

$$\mu^{(j)} = \frac{\sum_{i=1}^{n} w^{m(i,j)} \mathbf{x}^{(i)}}{\sum_{i=1}^{n} w^{m(i,j)}}$$

11.1.3 Using the elbow method to find the optimal number of clusters

11.1.4 Quantifying the quality of clustering via silhouette plots

To calculate the *silhouette coefficient* of a single sample in our dataset, we can apply the following three steps:

1. Calculate the cluster cohesion $a^{(i)}$ as the average distance between a sample $\mathbf{x}^{(i)}$ and all other points in the same cluster.
2. Calculate the cluster separation $b^{(i)}$ from the next closest cluster as the average distance between the sample $\mathbf{x}^{(i)}$ and all samples in the nearest cluster.
3. Calculate the silhouette $s^{(i)}$ as the difference between cluster cohesion and separation divided by the greater of the two, as shown here:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}.$$

The silhouette coefficient is bounded in the range -1 to 1. Based on the preceding formula, we can see that the silhouette coefficient is 0 if the cluster separation and cohesion are equal ($b^{(i)} = a^{(i)}$). Furthermore, we get close to an ideal silhouette coefficient of 1 if $b^{(i)} >> a^{(i)}$, since $b^{(i)}$ quantifies how dissimilar a sample is to other clusters, and $a^{(i)}$ tells us how similar it is to the other samples in its own cluster, respectively.

11.2 Organizing clusters as a hierarchical tree

11.2.1 Performing hierarchical clustering on a distance matrix

11.2.2 Attaching dendrograms to a heat map

11.2.3 Applying agglomerative clustering via scikit-learn

11.3 Locating regions of high density via DBSCAN

[...] In *Density-based Spatial Clustering of Applications with Noise* (DBSCAN), a special label is assigned to each sample (point) using the following criteria:

- A point is considered as *core point* if at least a specified number (*MinPts*) of neighboring points fall within the specified radius ϵ.

- A *border point* is a point that has fewer neighbors than MinPts within ϵ, but lies within the ϵ radius of a core point.
- All other points that are neither core nor border points are considered as *noise points*.

After labeling the points as core, border, or noise points, the DBSCAN algorithm can be summarized in two simple steps:

1. Form a separate cluster for each core point or a connected group of core points (core points are connected if they are no farther away than ϵ).
2. Assign each border point to the cluster of its corresponding core poin.

11.4 Summary

Chapter 12

Training Artificial Neural Networks for Image Recognition

12.1 Modeling complex functions with artificial neural networks

12.1.1 Single-layer neural network recap

In *Chapter 2, Training Machine Learning Algorithms for Classification*, we implemented the Adaline algorithm to perform binary classification, and we used a gradient descent optimization algorithm to learn the weight coefficients of the model. In every epoch (pass over the training set), we updated the weight vector $\mathbf{w}$ using the following update rule:

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, \quad \text{where } \Delta\mathbf{w} = -\eta\nabla J(\mathbf{w})$$

In other words, we computed the gradient based on the whole training set and updated the weights of the model by taking a step into the opposite direction of the gradient $\nabla J(\mathbf{w})$. In order to find the optimal weights of the model, we optimized an objective function that we defined as the *Sum of Squared Errors (SSE)* cost function $J(\mathbf{w})$. Furthermore, we multiplied the gradient by a factor, the learning rate η , which we chose carefully to balance the speed of learning against the risk of overshooting the global minimum of the cost function.
In gradient descent optimization, we updated all weights simultaneously after each epoch, and we de ned the partial derivative for each weight w_j in the weight vector $\mathbf{w}$ as follows:

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = \sum_i \big(y^{(i)} - a^{(i)}\big) x_j^{(i)}$$

Here $y^{(i)}$ is the target class label of a particular sample $x^{(i)}$, and $a^{(i)}$ is the *activation* of the neuron, which is a linear function in the special case of Adaline. Furthermore, we defined the *activation function* $\phi(\cdot)$ as follows:

$$\phi(z) = z = a$$

Here, the net input z is a linear combination of the weights that are connecting the input to the output layer:

$$z = \sum_j w_j x_j = \mathbf{w}^T \mathbf{x}$$

While we used the activation $\phi(z)$ to compute the gradient update, we implemented a *threshold function* (Heaviside function) $g(\cdot)$ to squash the continuous-valued output into binary class labels for prediction:

$$\phi(z) = \begin{cases} 1 & \text{if } g(z) \geq 0 \\ -1 & \text{otherwise .} \end{cases}$$

12.1.2 Introducing the multi-layer neural network architecture

[...] As shown in the preceding figure, we denote the ith activation unit in the lth layer as a_i^l , and the activation units a_0^1 and a_0^2 are the *bias units*, respectively, which we set equal to 1. The activation of the units in the input layer is just its input plus the bias unit:

$$\mathbf{a}^{(i)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$

Each unit in layer l is connected to all units in layer $l+1$ via a weight coefficient. For example, the connection between the kth unit in layer l to the jth unit in layer $l+1$ would be written as $w_{j,k}^{(l)}$. Please note that the superscript i in $x_m^{(i)}$ stands for the ith sample, not the ith layer. In the following paragraphs, we will often omit the superscript i for clarity.

[...] To better understand how this works, remember the one-hot representation of categorical variables that we introduced in *Chapter 4, Building Good Training Sets – Data Preprocessing*. For example, we would encode the three class labels in the familiar Iris dataset (*0=Setosa, 1=Versicolor, 2=Virginica*) as follows:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} .$$

This one-hot vector representation allows us to tackle classification tasks with an arbitrary number of unique class labels present in the training set.
[...] If you are new to neural network representations, the terminology around the indices (subscripts and superscripts) may look a little bit confusing at first. You may wonder why we wrote $w_{j,k}^{(l)}$ and not $w_{k,j}^{(l)}$ to refer to the weight coefficient that connects the kth unit in layer l to the jth unit in layer $l+1$. What may seem a little bit quirky at first will make much more sense in later sections when we vectorize the neural network representation. For example, we will summarize the weights that connect the input and hidden layer by a matrix $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times [m+1]}$, where h is the number of hidden units and $m+1$ is the number of input units plus bias unit.

12.1.3 Activating a neural network via forward propagation

[...] Now, let's walk through the individual steps of forward propagation to generate an output from the patterns in the training data. Since each unit in the hidden layer is connected to all units in the input layers, we first calculate the activation $a_1^{(2)}$ as follows:

$$z_1^{(2)} = a_0^{(1)} w_{1,0}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + \cdots + a_m^{(1)} w_{l,m}^{(1)}$$

$$a_1^{(2)} = \phi\big(z_1^{(2)}\big)$$

Here, $z_1^{(2)}$ is the net input and $\phi(\cdot)$ is the activation function, which has to be differentiable to learn the weights that connect the neurons using a gradient-based approach. To be able to solve complex problems such as image classification, we need nonlinear activation functions in our MLP model, for example, the sigmoid (logistic) function that we discussed in previous chapters:

$$\phi(z) = \frac{1}{1 + e^{-z}}.$$

For purposes of computational efficiency and code readability, we will now write the activation in a more compact form using the concepts of basic linear algebra, which will allow us to vectorize our code implementation:

$$\mathbf{z}^{(2)} = \mathbf{W}^{(1)} \mathbf{a}^{(1)}$$

$$\mathbf{a}^{(2)} = \phi\big(\mathbf{z}^{(2)}\big)$$

Note: Everywhere you read h in the following paragraphs of this section, you can think of h as $h+1$ to include the bias unit (and in order to get the dimensions right).
Here, $\mathbf{a}^{(1)}$ is our $[m+1] \times 1$ dimensional feature vector a sample $\mathbf{x}^{(i)}$ plus bias unit. $\mathbf{W}^{(i)}$ is an $h \times [m+1]$-dimensional weight matrix where h is the number

of hidden units in our neural network. After matrix-vector multiplication, we obtain the $h \times 1$-dimensional net input vector $\mathbf{z}^{(2)}$ to calculate the activation $\mathbf{a}^{(2)}$ (where $\mathbf{a}^{(2)} \in \mathbb{R}^{h \times 1}$). Furthermore, we can generalize this computation to all n samples in the training set:

$$\mathbf{Z}^{(2)} = \mathbf{W}^{(1)}\left[\mathbf{A}^{(1)}\right]^T$$

Here, $\mathbf{A}^{(1)}$ is now an $n \times [m+1]$ matrix, and the matrix-matrix multiplication will result in an $h \times n$-dimensional net input matrix $\mathbf{Z}^{(2)}$. Finally, we apply the activation function $\phi(\cdot)$ to each value in the net input matrix to get the $h \times n$ activation matrix $\mathbf{A}^{(2)}$ for the next layer (here, output layer):

$$\mathbf{A}^{(2)} = \phi\big(\mathbf{Z}^{(2)}\big)$$

Similarly, we can rewrite the activation of the output layer in the vectorized form:

$$\mathbf{Z}^{(3)}\mathbf{W}^{(2)}\mathbf{A}^{(2)}$$

Here, we multiply the $t \times h$ matrix $\mathbf{W}^{(2)}$ (t is the number of output units) by the $h \times n$ dimensional matrix $\mathbf{A}^{(2)}$ to obtain the $t \times n$ dimensional matrix $\mathbf{Z}^{(3)}$ (the columns in this matrix represent the outputs for each sample). Lastly, we apply the sigmoid activation function to obtain the continuous valued output of our network:

$$\mathbf{A}^{(3)} = \phi\big(\mathbf{Z}^{(3)}\big),\ \mathbf{A}^{(3)} \in \mathbb{R}^{t \times n}.$$

12.2 Classifying handwritten digits

12.2.1 Obtaining the MNIST dataset

12.2.2 Implementing a multi-layer perceptron

As you may have noticed, by going over our preceding MLP implementation, we also implemented some additional features, which are summarized here:

- *l2*: the λ parameter for L2 regularization to decrease the degree of overfitting; equivalently, *l1* is the λ parameter for L1 regularization.
- *epochs*: The number of passes over the training set.
- *eta*: The learning rate η
- *alpha*: A parameter for momentum learning to add a factor of the previous gradient to the weight update for faster learning

  $$\Delta\mathbf{w}_t = \eta\nabla J(\mathbf{w}_t) + \alpha\Delta\mathbf{w}_t - 1,$$

  where t is the current time step or epoch.

- *decrease_const*: The decrease constant d for an adaptive learning rate η that decreases over time for better convergence $\eta/1 + t \times d$.
- *shuffle*: Shuffling the training set prior to every epoch to prevent the algorithm from getting stuck in cycles.
- *Minibatches*: Splitting of the training data into k mini-batches in each epoch. The gradient is computed for each mini-batch separately instead of the entire training data for faster learning.

12.3 Training an artificial neural network

12.3.1 Computing the logistic cost function

The logistic cost function that we implemented as the *_get_cost* method is actually pretty simple to follow since it is the same cost function that we described in the logistic regression section in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn.*

$$J(\mathbf{w}) = -\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)$$

Here, $a^{(i)}$ is the sigmoid activation of the ith unit in one of the layers which we compute in the forward propagation step:

$$a^{(i)} = \phi\big(z^{(i)}\big).$$

Now, let's add a regularization term, which allows us to reduce the degree of over tting. As you will recall from earlier chapters, the L2 and L1 regularization terms are de ned as follows (remember that we don't regularize the bias units):

$$L2 = \lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2 \text{ and } L1 = \lambda \|\mathbf{w}\|_1 = \lambda \sum_{j=1}^{m} |w_j|.$$

[...] By adding the L2 regularization term to our logistic cost function, we obtain the following equation:

$$J(\mathbf{w}) = -\left[\sum_{i=1}^{n} y^{(i)} \log\left(a^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - a^{(i)}\right)\right] + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

Since we implemented an MLP for multi-class classification, this returns an output vector of t elements, which we need to compare with the $t \times 1$ dimensional target vector in the one-hot encoding representation. For example, the activation of the third layer and the target class (here: class 2) for a particular sample may look like this:

$$a^{(3)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \; \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Thus, we need to generalize the logistic cost function to all activation units j in our network. So our cost function (without the regularization term) becomes:

$$J(\mathbf{w}) = -\sum_{i=1}^{n}\sum_{j=1}^{t} = y_j^{(i)} \log\left(1 - a_j^{(1)}\right)$$

Here, the superscript i is the index of a particular sample in our training set. The following generalized regularization term may look a little bit complicated at first, but here we are just calculating the sum of all weights of a layer l (without the bias term) that we added to the first column:

$$J(\mathbf{w}) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{m} y_j^{(i)} \log\left(\phi\left(z_j^{(i)}\right)\right) + \left(1 - y_j^{(i)}\right) \log\left(1 - \phi\left(z_j^{(i)}\right)\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

The following expression represents the L2-penalty term:

$$\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(w_{j,i}^{(l)}\right)^2$$

Remember that our goal is to minimize the cost function $J(\mathbf{w})$. Thus, we need to calculate the partial derivative of matrix $\mathbf{W}$ with respect to each weight for every layer in the network:

$$\frac{\partial}{\partial w_{j,i}^{l}} J(\mathbf{W}).$$

12.3.2 Training neural networks via backpropagation

[...] As we recall from the beginning of this chapter, we first need to apply forward propagation in order to obtain the activation of the output layer, which we formulated as follows:

$$\mathbf{Z}^{(2)} = \mathbf{W}^{(1)}\left[\mathbf{A}^{(1)}\right]^T \quad \text{(net input of the hidden layer)}$$

$$\mathbf{A}^{(2)} = \phi\left(\mathbf{Z}^{(2)} = \right) \quad \text{(activation of the hidden layer)}$$

$$\mathbf{Z}^{(3)} = \mathbf{W}^{(2)}\mathbf{A}^{(2)} \quad \text{(net input of the output layer)}$$

$$\mathbf{A}^{(3)} = \phi\big(\mathbf{Z}^{(3)}\big) \quad \text{(activation of the output layer)}$$

[...] In backpropagation, we propagate the error from right to left. We start by calculating the error vector of the output layer:

$$\delta^{(3)} = \mathbf{a}^{(3)} - \mathbf{y}$$

Here, $\mathbf{y}$ is the vector of the true class labels. Next, we calculate the error term of the hidden layer:

$$\delta^{(2)} = \big(\mathbf{W}^{(2)}\big)^T \delta^{(3)} \cdot \frac{\partial \phi\big(z^{(2)}\big)}{\partial z^{(2)}}.$$

Here, $\frac{\partial \phi\left(z^{(2)}\right)}{\partial z^{(2)}}$ is simply the derivative of the sigmoid activation function, which we implemented as *_sigmoid_gradient*:

$$\frac{\partial \phi\big(z^{(2)}\big)}{\partial z^{(2)}} = \Big(a^{(2)} \cdot \big(1 - a^{(2)}\big)\Big).$$

Note that the asterisk symbol $(\cdot)$ means element-wise multiplication in this context.
Although, it is not important to follow the next equations, you may be curious as to how I obtained the derivative of the activation function. I summarized the derivation step by step here:

$$\begin{aligned}
\phi'(z) &= \frac{\partial}{\partial z}\Big(\frac{1}{1+e^{-z}}\Big) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}}{\big(1+e^{-z}\big)^2} - \Big(\frac{1}{1+e^{-z}}\Big)^2 \\
&= \frac{1}{\big(1+e^{-z}\big)^2} - \Big(\frac{1}{1+e^{-z}}\Big)^2 \\
&= \phi(z) - \big(\phi(z)\big)^2 \\
&= \phi(z) - \big(1 - \phi(z)\big) \\
&= a(1-a)
\end{aligned}$$

To better understand how we compute the $\delta^P(3)$ term, let's walk through it in more detail. In the preceding equation, we multiplied the transpose $(\mathbf{W}^{(2)})^T$ of the $t \times h$ dimensional matrix $\mathbf{W}^{(2)}$; t is the number of output class labels and h is the number of hidden units. Now, $(\mathbf{W}^{(2)})^T$ becomes an $h \times t$ dimensional matrix with $\delta^{(3)}$, which is a $t \times 1$ dimensional vector. We then performed a pair-wise multiplication between $(\mathbf{W}^{(2)})^T \delta^{(3)}$ and $\Big(a^{(2)} \cdot \big(1 - a^{(2)}\big)\Big)$, which is

also a $t \times 1$ dimensional vector. Eventually, after obtaining the ? terms, we can now write the derivation of the cost function as follows:

$$\frac{\partial}{\partial w_{i,j}^{l}} J(\mathbf{W}) = a_j^l \delta_i^{(l+1)}$$

Next, we need to accumulate the partial derivative of every jth node in layer l and the ith error of the node in layer $l+1$:

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

Remember that we need to compute $\Delta_{i,j}^{(l)}$ for every sample in the training set. Thus, it is easier to implement it as a vectorized version like in our preceding MLP code implementation:

$$\Delta^{(l)} := \Delta^{(l)} \delta^{(l+1)} \left(\mathbf{A}^{(l)}\right)^T$$

After we have accumulated the partial derivatives, we can add the regularization term as follows:

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \quad \text{(except for the bias term)}$$

Lastly, after we have computed the gradients, we can now update the weights by taking an opposite step towards the gradient:

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \eta \Delta^{(l)}$$

12.4 Developing your intuition for backpropagation

12.5 Debugging neural networks with gradient checking

In the previous sections, we de ned a cost function $J(\mathbf{W})$ where $\mathbf{W}$ is the matrix of the weight coefficients of an artificial network. Note that $J(\mathbf{W})$ is – roughly speaking – a "stacked" matrix consisting of the matrices $\mathbf{W}^{(1)}$ and $W^{(2)}$ in a multi-layer perceptron with one hidden unit. We de ned $\mathbf{W}^{(1)}$ as the $h \times [m+1]$-dimensional matrix that connects the input layer to the hidden layer, where h is the number of hidden units and m is the number of features (input units). The matrix $\mathbf{W}^{(2)}$ that connects the hidden layer to the output layer has the dimensions $t \times h$, where t is the number of output units. We then calculated the derivative of the cost function for a weight $w_{i,j}^{l}$ as follows:

$$\frac{\partial}{\partial w_{i,j}^{(i)}}$$

Remember that we are updating the weights by taking an opposite step towards the direction of the gradient. In gradient checking, we compare this analytical solution to a numerically approximated gradient:

$$\frac{\partial}{\partial w_{i,j}^{l}} J(\mathbf{W}) \approx \frac{J\big(w_{i,j}^{(l)} + \epsilon\big) - J\big(w_{i,j}^{(l)}\big)}{\epsilon}$$

Here, ϵ is typically a very small number, for example 1e-5 (note that 1e-5 is just a more convenient notation for 0.00001). Intuitively, we can think of this finite difference approximation as the slope of the secant line connecting the points of the cost function for the two weights w and $w + \epsilon$ (both are scalar values), as shown in the following figure. We are omitting the superscripts and subscripts for simplicity.
[...] An even better approach that yields a more accurate approximation of the gradient is to compute the symmetric (or centered) difference quotient given by the two-point formula:

$$\frac{J\big(w_{i,j}^{(l)} + \epsilon\big) - J\big(w_{i,j}^{(l)} - \epsilon\big)}{2\epsilon}$$

Typically, the approximated difference between the numerical gradient J'_n and analytical gradient J'_a is then calculated as the L2 vector norm. For practical a reasons, we unroll the computed gradient matrices into at vectors so that we can calculate the error (the difference between the gradient vectors) more conveniently:

$$\text{error} = \|J'_n - J'_a\|_2$$

One problem is that the error is not scale invariant (small errors are more significant if the weight vector norms are small, too). Thus, it is recommended to calculate a normalized difference:

$$\text{relative error} = \frac{\|J'_n - J'_a\|_2}{\|J'_n\|_2 + \|J'_a\|_2}$$

12.6 Convergence in neural networks

12.7 Other neural network architectures

12.7.1 Convolutional Neural Networks

12.7.2 Recurrent Neural Networks

12.8 A few last words about neural network implementation

12.9 Summary

Chapter 13

Parallelizing Neural Network Training with Theano

13.1 Building, compiling, and running expressions with Theano

13.1.1 What is Theano?

13.1.2 First steps with Theano

13.1.3 Configuring Theano

13.1.4 Working with array structures

13.1.5 Wrapping things up – a linear regression example

13.2 Choosing activation functions for feedforward neural networks

13.2.1 Logistic function recap

We recall from the section on logistic regression in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn* that we can use the logistic function to model the probability that sample x belongs to the positive class (class 1) in a binary classification task:

$$\phi_{logistic}(z) = \frac{1}{1 + e^{-z}}$$

Here, the scalar variable z is de ned as the net input:

$$z = w_0 x_0 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \mathbf{w}^T \mathbf{x}$$

Note that w_0 is the bias unit (y-axis intercept, $x_0 = 1$).

13.2.2 Estimating probabilities in multi-class classification via the softmax function

The softmax function is a generalization of the logistic function that allows us to compute meaningful class-probabilities in multi-class settings (multinomial logistic regression). In softmax, the probability of a particular sample with net input z belongs to the i th class can be computed with a normalization term in the denominator that is the sum of all M linear functions:
$P(y = i|z) = \phi_{softmax}(z) = \frac{e_i^z}{\sum_{m=1}^{M} e_m^z}$.

13.2.3 Broadening the output spectrum by using a hyperbolic tangent

Another sigmoid function that is often used in the hidden layers of artificial neural networks is the *hyperbolic tangent (tanh)*, which can be interpreted as a rescaled version of the logistic function.

$$\phi_{tanh}(z) = 2 \times \phi_{logistic}(2 \times z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\phi_{logistic}(z) = \frac{1}{1 + e^{-z}}$$

13.3 Training neural networks efficiently using Keras

13.4 Summary