Python Machine Learning
Equation Reference

Sebastian Raschka
mail@sebastianraschka.com

05/04/2015
(last updated: 06/15/2016)
https://github.com/rasbt/python-machine-learning-book

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:

$$\begin{aligned}
&\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big)^2 \\
&= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big)^2 \\
&= \frac{1}{2} \sum_i 2 \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big) \frac{\partial J}{\partial w_j} \Big(y^{(i)} - \sum_i \Big(w_j^{(i)} x_j^{(i)} \Big) \Big) \\
&= \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big) \Big(- x_j^{(i)} \Big) \\
&= - \sum_i \Big(y^{(i)} - \phi\big(z^{(i)}\big) \Big) x_j^{(i)}
\end{aligned} \tag{2.1}$$

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 $P(y=0|\mathbf{x};\mathbf{w}) = 1 - P(y=1|\mathbf{x};\mathbf{w}) = 0.2 or 20 percent$. 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:

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

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

$$\begin{aligned} & \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z) \\ &= \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z) \big(1-\phi(z)\big) \frac{\partial}{\partial w_j} z \\ &= \Big(y \big(1-\phi(z)\big) - (1-y)\phi(z) \Big) x_j \\ &= \big(y - \phi(z)\big) x_j \end{aligned} \tag{3.1}$$

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

... to be continued ...