Python机器学习基础教程

Stephen CUI[1]

February 17, 2023

[1]cuixuanStephen@gmail.com

Contents

Chapter 1

监督学习

1.1 分类与回归

监督机器学习问题主要有两种，分别叫作分类（classification）与回归（regression）。

分类问题的目标是预测类别标签（class label），这些标签来自预定义的可选列表。分类问题有时可分为二分类（binary classification，在两个类别之间进行区分的一种特殊情况）和多分类（multiclass classification，在两个以上的类别之间进行区分）。在二分类问题中，我们通常将其中一个类别称为正类（positive class），另一个类别称为反类（negative class）。这里的“正”并不代表好的方面或正数，而是代表研究对象。

回归任务的目标是预测一个连续值，编程术语叫作浮点数（floating-point number），数学术语叫作实数（real number）。

区分分类任务和回归任务有一个简单方法，就是问一个问题：输出是否具有某种连续性。如果在可能的结果之间具有连续性，那么它就是一个回归问题。

1.2 泛化、过拟合与欠拟合

在监督学习中，我们想要在训练数据上构建模型，然后能够对没见过的新数据（这些新数据与训练集具有相同的特性）做出准确预测。如果一个模型能够对没见过的数据做出准确预测，我们就说它能够从训练集**泛化**（generalize）到测试集。我们想要构建一个泛化精度尽可能高的模型。

判断一个算法在新数据上表现好坏的唯一度量，就是在测试集上的评估。然而从直觉上看[1]，我们认为简单的模型对新数据的泛化能力更好。因此，我们总想找到最简单的模型。构建一个对现有信息量来说过于复杂的模型，这被称为**过拟合**（overfitting）。如果你在拟合模型时过分关注训练集的细节，得到了一个在训练集上表现很好、但不能泛化到新数据上的模型，那么就存在过拟合。与之相反，如果你的模型过于简单，那么你可能无法抓住数据的全部内容以及数据中的变化，你的模型甚至在训练集上的表现就很差。选择过于简单的模型被称为**欠拟合**（underfitting）。

模型复杂度与数据集大小的关系

需要注意，模型复杂度与训练数据集中输入的变化密切相关：数据集中包含的数据点的变化范围越大，在不发生过拟合的前提下你可以使用的模型就越复杂。通常来说，收集更多的数据点可以有更大的

[1] 在数学上也可以证明这一点。

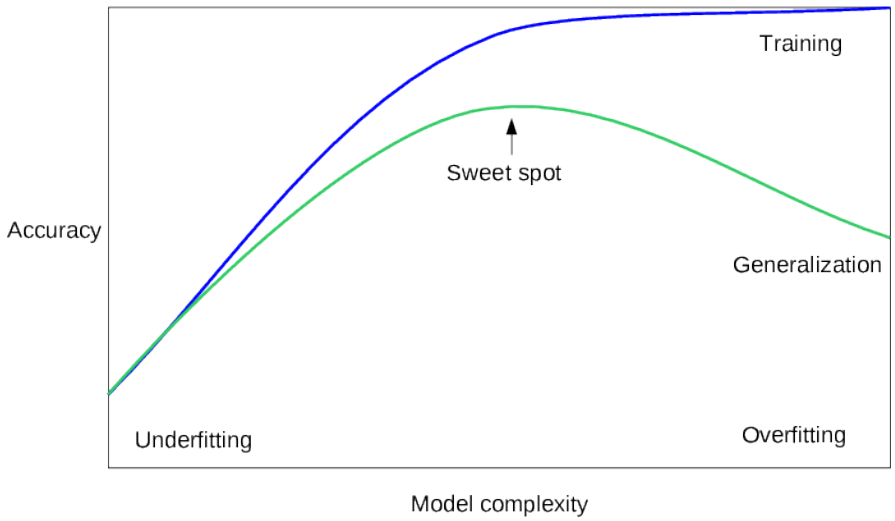

Figure 1.1: Trade-off of model complexity against training and test accuracy

Table 1.1: 一些样本数据集

名称	来源	特征量	用途
forge	模拟	26×2	二分类数据集
wave	模拟	40×1	回归算法数据集
cancer	真实	569×30	二分类数据集
boston	真实	506×13	回归数据集

变化范围，所以更大的数据集可以用来构建更复杂的模型。但是，仅复制相同的数据点或收集非常相似的数据是无济于事的。

收集更多数据，适当构建更复杂的模型，对监督学习任务往往特别有用。本书主要关注固定大小的数据集。在现实世界中，你往往能够决定收集多少数据，这可能比模型调参更为有效。永远不要低估更多数据的力量!

1.3 监督学习算法

现在开始介绍最常用的机器学习算法，并解释这些算法如何从数据中学习以及如何预测。我们还会讨论每个模型的复杂度如何变化，并概述每个算法如何构建模型。我们将说明每个算法的优点和缺点，以及它们最适应用于哪类数据。此外还会解释最重要的参数和选项的含义，许多算法都有分类和回归两种形式。

1.3.1 一些样本数据集

我们将使用一些数据集来说明不同的算法。其中一些数据集很小，而且是模拟的，其目的是强调算法的某个特定方面。其他数据集都是现实世界的大型数据集。

从特征较少的数据集（也叫低维数据集）中得出的结论可能并不适用于特征较多的数据集（也叫高维数据集）。只要你记住这一点，那么在低维数据集上研究算法也是很有启发的。

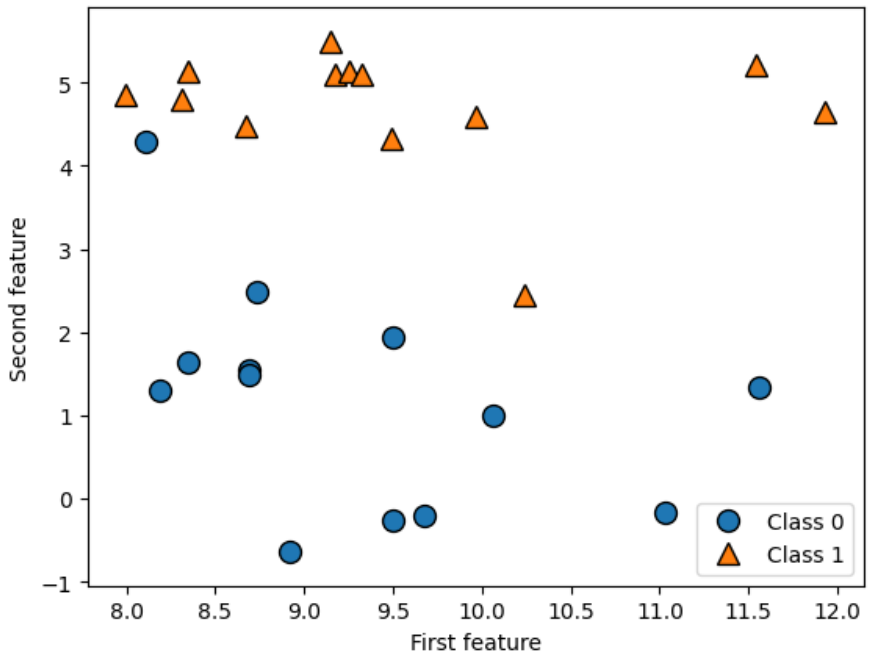

Figure 1.2: Scatter plot of the forge dataset

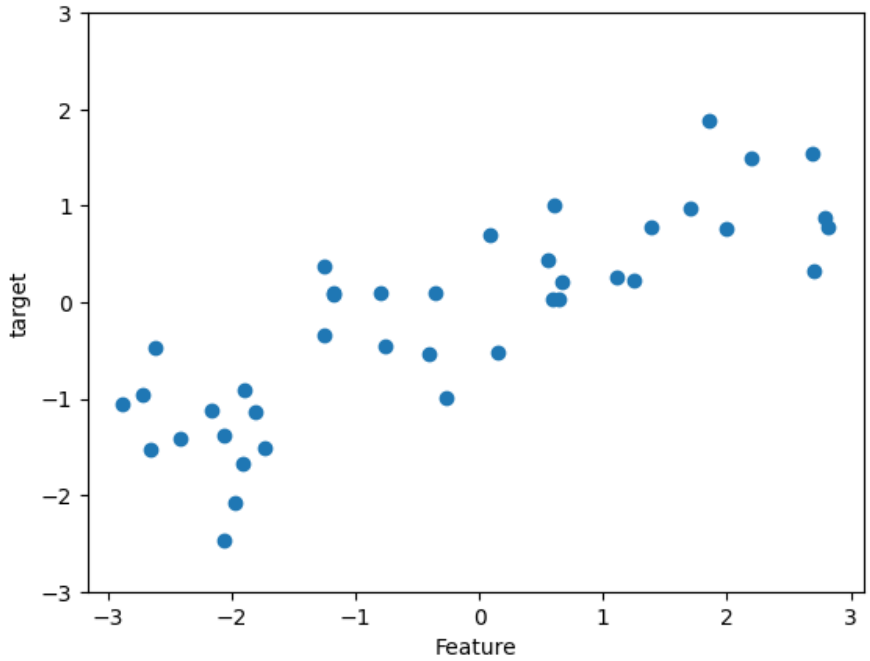

Figure 1.3: Plot of the wave dataset

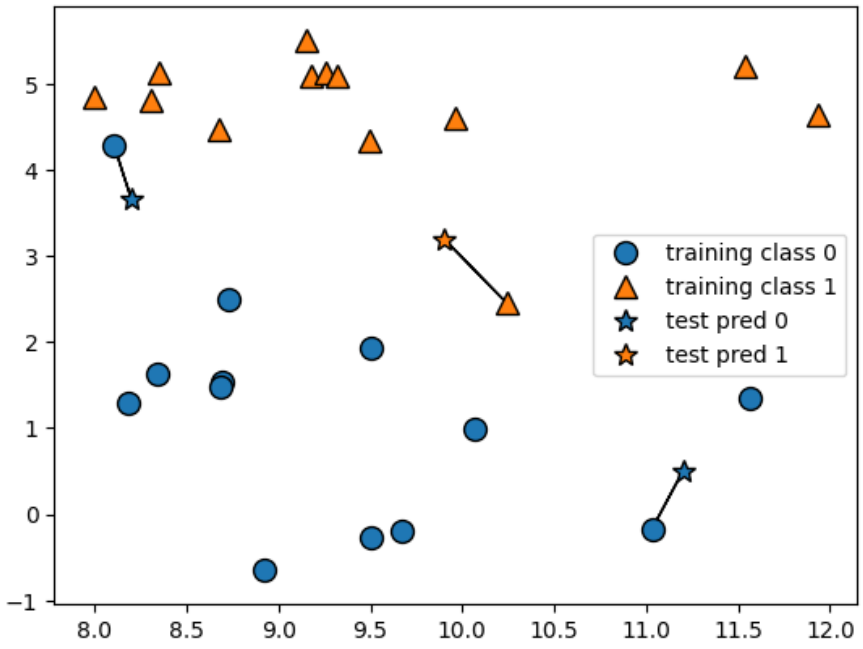

Figure 1.4: Predictions made by the one-nearest-neighbor model on the forge dataset

> 包含在 scikit-learn 中的数据集通常被保存为 Bunch 对象，里面包含真实数据以及一些数据集信息。关于 Bunch 对象，你只需要知道它与字典很相似，而且还有一个额外的好处，就是你可以用点操作符来访问对象的值（比如用 `bunch.key` 来代替 `bunch['key']`）。

对于我们的目的而言，我们需要扩展boston数据集，输入特征不仅包括这 13 个测量结果，还包括这些特征之间的乘积（也叫**交互项**）。换句话说，我们不仅将犯罪率和公路可达性作为特征，还将犯罪率和公路可达性的乘积作为特征。像这样包含导出特征的方法叫作**特征工程**（feature engineering）。

1.3.2 k近邻

k-NN 算法可以说是最简单的机器学习算法。构建模型只需要保存训练数据集即可。想要对新数据点做出预测，算法会在训练数据集中找到最近的数据点，也就是它的“最近邻”。

k近邻分类 k-NN 算法最简单的版本只考虑一个最近邻，也就是与我们想要预测的数据点最近的训练数据点。预测结果就是这个训练数据点的已知输出。Figure 1.4给出了这种分类方法在 forge 数据集上的应用。

除了仅考虑最近邻，我还可以考虑任意个（k 个）邻居。这也是 k 近邻算法名字的来历。在考虑多于一个邻居的情况时，我们用“投票法”（voting）来指定标签。将出现次数更多的类别（也就是 k 个近邻中占多数的类别）作为预测结果。

分析KNeighborsClassifier 分别将 1 个、3 个和 9 个邻居三种情况的决策边界可视化，见Figure 1.6。

从左图可以看出，使用单一邻居绘制的决策边界紧跟着训练数据。随着邻居个数越来越多，决策边界也越来越平滑。更平滑的边界对应更简单的模型。换句话说，使用更少的邻居对应更高的模型复杂度（如Figure 1.1右侧所示），而使用更多的邻居对应更低的模型复杂度（如Figure 1.1左侧所示）。假如考虑

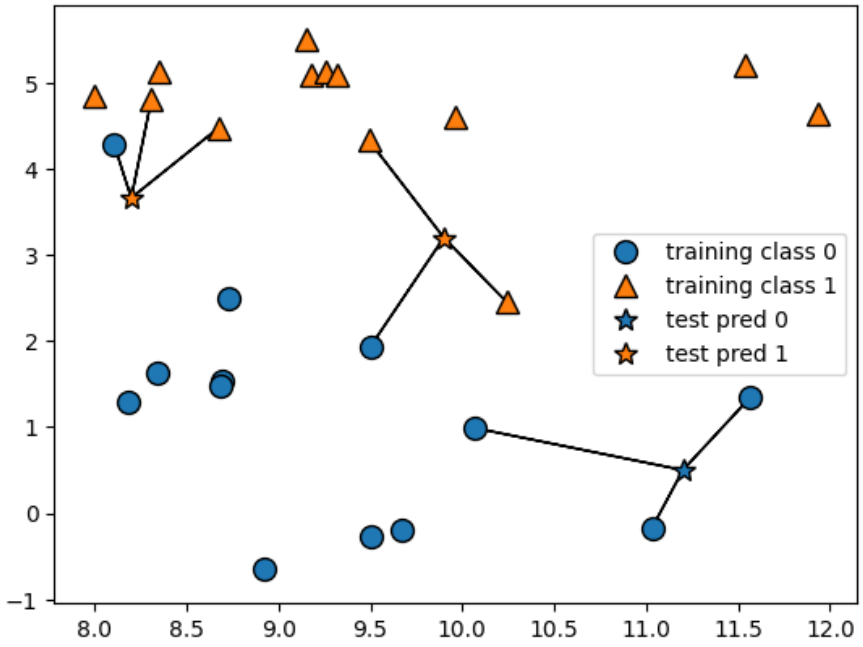

Figure 1.5: Predictions made by the three-nearest-neighbors model on the forge dataset

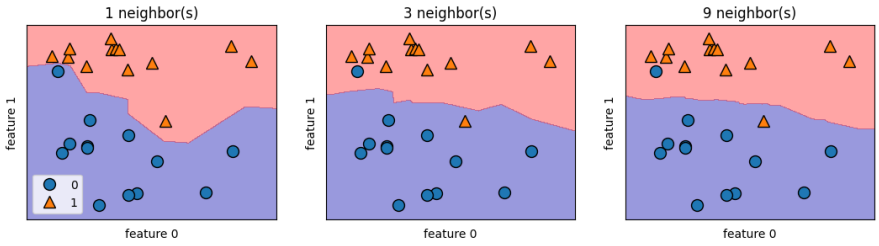

Figure 1.6: Decision boundaries created by the nearest neighbors model

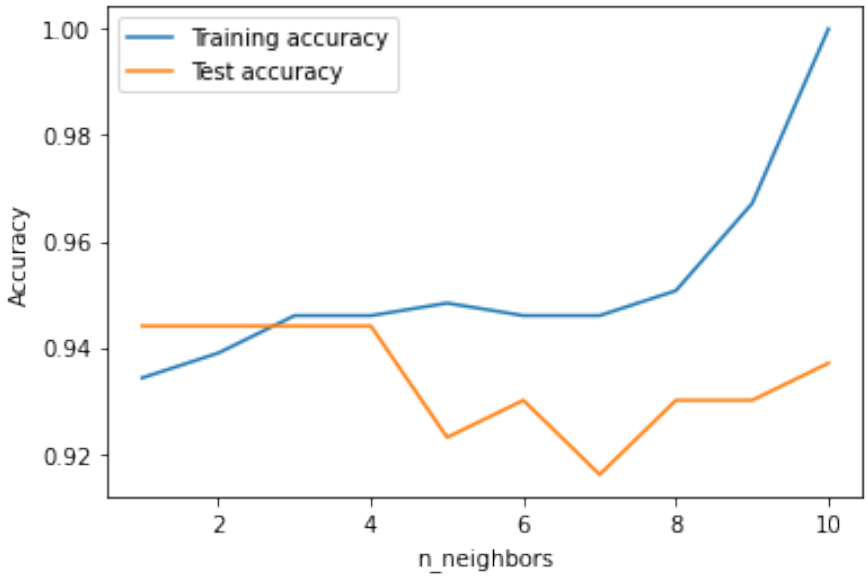

Figure 1.7: Comparison of training and test accuracy

极端情况，即邻居个数等于训练集中所有数据点的个数，那么每个测试点的邻居都完全相同（即所有训练点），所有预测结果也完全相同（即训练集中出现次数最多的类别）。

我们来研究一下能否证实之前讨论过的模型复杂度和泛化能力之间的关系。先将数据集分成训练集和测试集，然后用不同的邻居个数对训练集和测试集的性能进行评估。输出结果见Figure 1.7：

注意，由于更少的邻居对应更复杂的模型，所以做了水平翻转。仅考虑单一近邻时，训练集上的预测结果十分完美。但随着邻居个数的增多，模型变得更简单，训练集精度也随之下降。单一邻居时的测试集精度比使用更多邻居时要低，这表示单一近邻的模型过于复杂。与之相反，当考虑 10 个邻居时，模型又过于简单，性能甚至变得更差。最佳性能在中间的某处，邻居个数大约为 6。

k近邻回归 k 近邻算法还可以用于回归。我们还是先从单一近邻开始，这次使用 wave 数据集。我们添加了 3 个测试数据点，在 x 轴上用绿色五角星表示。利用单一邻居的预测结果就是最近邻的目标值。在Figure 1.8中用蓝色五角星表示。

我们还可以用score方法来评估模型，对于回归问题，这一方法返回的是 R^2分数。R^2分数也叫作决定系数，是回归模型预测的优度度量，位于0到1之间。R^2等于1对应完美预测，R^2等于0对应常数模型，即总是预测训练集响应（`y_train`）的平均值：

分析KNeighborsRegressor 对于我们的一维数据集，可以查看所有特征取值对应的预测结果（图Figure 1.8）。

从图中可以看出，仅使用单一邻居，训练集中的每个点都对预测结果有显著影响，预测结果的图像经过所有数据点。这导致预测结果非常不稳定。考虑更多的邻居之后，预测结果变得更加平滑，但对训练数据的拟合也不好。

优点、缺点和参数 一般来说，KNeighbors 分类器有 2 个重要参数：邻居个数与数据点之间距离的度量方法。在实践中，使用较小的邻居个数（比如 3 个或 5 个）往往可以得到比较好的结果，但你应该调节

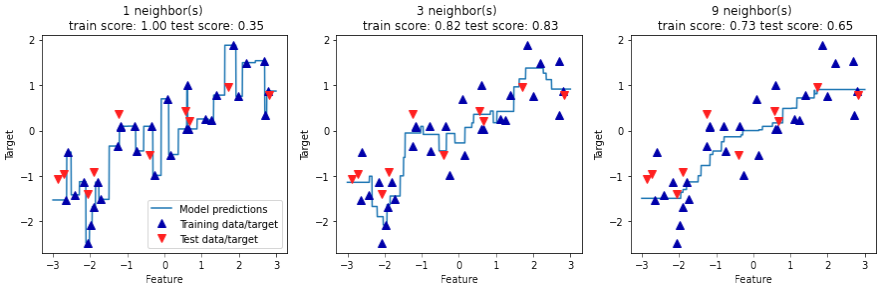

Figure 1.8: Comparing predictions made by nearest neighbors regression

这个参数。选择合适的距离度量方法超出了本书的范围。默认使用欧式距离，它在许多情况下的效果都很好。

k-NN 的优点之一就是模型很容易理解，通常不需要过多调节就可以得到不错的性能。在考虑使用更高级的技术之前，尝试此算法是一种很好的基准方法。构建最近邻模型的速度通常很快，但如果训练集很大（特征数很多或者样本数很大），预测速度可能会比较慢。使用 k-NN 算法时，对数据进行预处理是很重要的（见第 3 章）。这一算法对于有很多特征（几百或更多）的数据集往往效果不好，对于大多数特征的大多数取值都为 0 的数据集（所谓的稀疏数据集）来说，这一算法的效果尤其不好。

虽然 k 近邻算法很容易理解，但由于预测速度慢且不能处理具有很多特征的数据集，所以在实践中往往不会用到。

1.3.3 线性模型

线性模型利用输入特征的**线性函数**（linear function）进行预测

用于回归的线性模型 对于回归问题，线性模型预测的一般公式如下：

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \cdots + w[p] * x[p] + b$$

这里 $x[0]$ 到 $x[p]$ 表示单个数据点的特征（本例中特征个数为 $p+1$），w 和 b 是学习模型的参数，$\hat{y}$ 是模型的预测结果。对于单一特征的数据集，公式如下：

$$\hat{y} = w[0] * x[0] + b$$

对于有更多特征的数据集，w 包含沿每个特征坐标轴的斜率。或者，你也可以将预测的响应值看作输入特征的加权求和，权重由 w 的元素给出（可以取负值）。

用于回归的线性模型可以表示为这样的回归模型：对单一特征的预测结果是一条直线，两个特征时是一个平面，或者在更高维度（即更多特征）时是一个超平面。

如果将直线的预测结果与Figure 1.9 中 KNeighborsRegressor 的预测结果进行比较，你会发现直线的预测能力非常受限。似乎数据的所有细节都丢失了。从某种意义上来说，这种说法是正确的。假设目标 y 是特征的线性组合，这是一个非常强的（也有点不现实的）假设。但观察一维数据得出的观点有些片面。对于有多个特征的数据集而言，线性模型可以非常强大。特别地，如果特征数量大于训练数据点的数量，任何目标 y 都可以（在训练集上）用线性函数完美拟合。

线性回归（又名普通最小二乘法） 线性回归，或者**普通最小二乘法**（ordinary least squares，OLS），是回归问题最简单也最经典的线性方法。线性回归寻找参数 w 和 b，使得对训练集的预测值与真实的回归

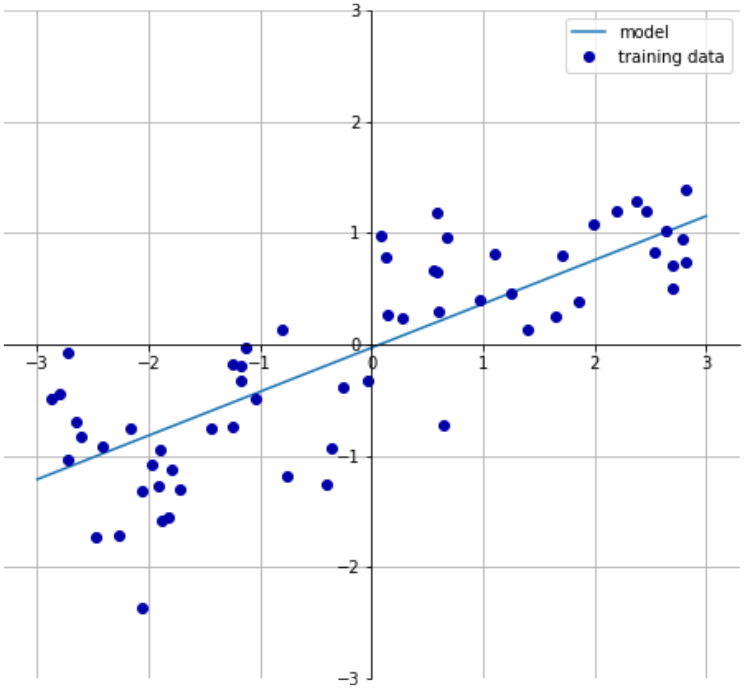

Figure 1.9: Predictions of a linear model on the wave dataset

目标值 y 之间的均方误差最小。**均方误差**（*mean squared error*）是预测值与真实值之差的平方和除以样本数。线性回归没有参数，这是一个优点，但也因此无法控制模型的复杂度。

斜率”参数（w，也叫作权重或系数）被保存在 `coef_` 属性中，而偏移或截距（b）被保存在 `intercept_` 属性中：

> 注意到了 `coef_` 和 `intercept_` 结尾处奇怪的下划线。scikit-learn总是将从训练数据中得出的值保存在以下划线结尾的属性中。这是为了将其与用户设置的参数区分开。

`intercept_` 属性是一个浮点数，而 `coef_` 属性是一个 NumPy 数组，每个元素对应一个输入特征。由于 wave 数据集中只有一个输入特征，所以 `lr.coef_` 中只有一个元素。

我们来看一下训练集和测试集的性能：

```
print('Training set score: {:.2f}'.format(lr.score(X_train, y_train)))
print('Training set score: {:.2f}'.format(lr.score(X_test, y_test)))
# 'Training set score: 0.67'
# 'Training set score: 0.66'
```

R^2 约为 0.66，这个结果不是很好，但我们可以看到，训练集和测试集上的分数非常接近。这说明可能存在欠拟合，而不是过拟合。对于这个一维数据集来说，过拟合的风险很小，因为模型非常简单（或受限）。然而，对于更高维的数据集（即有大量特征的数据集），线性模型将变得更加强大，过拟合的可能性也会变大。

```
print('Training set score: {:.2f}'.format(lr.score(X_train, y_train)))
print('Training set score: {:.2f}'.format(lr.score(X_test, y_test)))
# Training set score: 0.95
# Training set score: 0.61
```

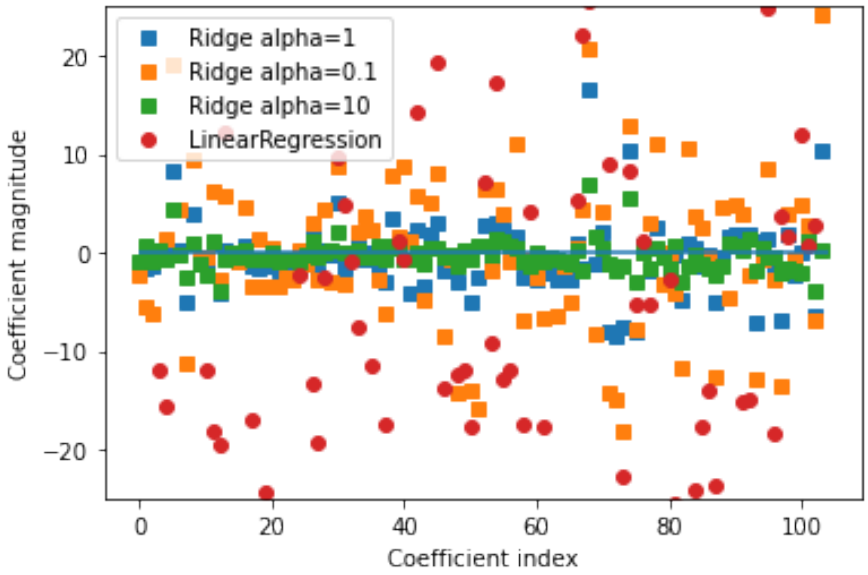

Figure 1.10: Comparing coefficient magnitudes for ridge regression

训练集和测试集之间的性能差异是过拟合的明显标志，因此我们应该试图找到一个可以控制复杂度的模型。标准线性回归最常用的替代方法之一就是岭回归（ridge regression）

岭回归 在岭回归中，对系数（w）的选择不仅要在训练数据上得到好的预测结果，而且还要拟合附加约束。我们还希望系数尽量小。换句话说，w 的所有元素都应接近于 0。直观上来看，这意味着每个特征对输出的影响应尽可能小（即斜率很小），同时仍给出很好的预测结果。这种约束是所谓正则化（regularization）的一个例子。正则化是指对模型做显式约束，以避免过拟合。岭回归用到的这种被称为 L2 正则化。

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train, y_train)
print('Training set score: {:.2f}'.format(ridge.score(X_train, y_train)))
print('Training set score: {:.2f}'.format(ridge.score(X_test, y_test)))
# Training set score: 0.89
# Training set score: 0.75
```

可以看出，Ridge 在训练集上的分数要低于 LinearRegression，但在测试集上的分数更高。这和我们的预期一致。线性回归对数据存在过拟合。Ridge 是一种约束更强的模型，所以更不容易过拟合。复杂度更小的模型意味着在训练集上的性能更差，但泛化性能更好。由于我们只对泛化性能感兴趣，所以应该选择 Ridge 模型而不是 LinearRegression 模型。

Ridge 模型在模型的简单性（系数都接近于 0）与训练集性能之间做出权衡。简单性和训练集性能二者对于模型的重要程度可以由用户通过设置 alpha 参数来指定。alpha 的最佳设定值取决于用到的具体数据集。增大 alpha 会使得系数更加趋向于 0，从而降低训练集性能，但可能会提高泛化性能。

减小 alpha 可以让系数受到的限制更小，即在Figure 1.1中向右移动。对于非常小的 alpha 值，系数几乎没有受到限制，我们得到一个与 LinearRegression 类似的模型。

我们还可以查看 alpha 取不同值时模型的 `coef_` 属性，从而更加定性地理解 alpha 参数是如何改变模型的。更大的 alpha 表示约束更强的模型，所以我们预计大 alpha 对应的`coef_`元素比小 alpha 对应的

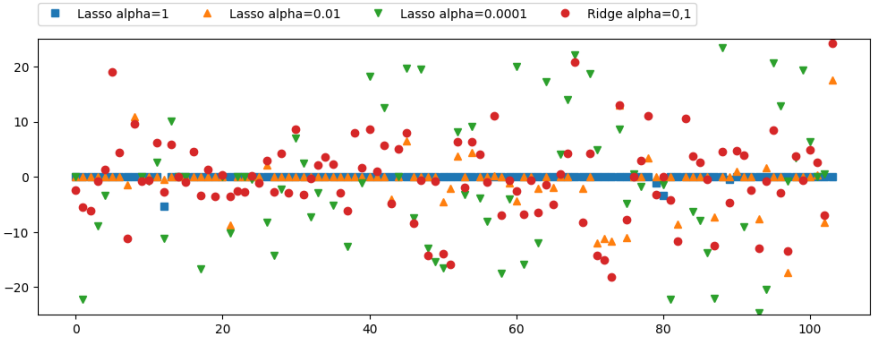

Figure 1.11: Comparing coefficient magnitudes for lasso regression

`coef_` 元素要小。这一点可以在Figure 1.10中得到证实。

还有一种方法可以用来理解正则化的影响，就是固定 alpha 值，但改变训练数据量。将模型性能作为数据集大小的函数进行绘图，这样的图像叫作**学习曲线**。

正如所预计的那样，无论是岭回归还是线性回归，所有数据集大小对应的训练分数都要高于测试分数。由于岭回归是正则化的，因此它的训练分数要整体低于线性回归的训练分数。但岭回归的测试分数要更高，特别是对较小的子数据集。如果少于 400 个数据点，线性回归学不到任何内容。随着模型可用的数据越来越多，两个模型的性能都在提升，最终线性回归的性能追上了岭回归。这里要记住的是，如果有足够多的训练数据，正则化变得不那么重要，并且岭回归和线性回归将具有相同的性能。**??**中还有一个有趣之处，就是线性回归的训练性能在下降。如果添加更多数据，模型将更加难以过拟合或记住所有的数据。

lasso 除了 Ridge，还有一种正则化的线性回归是 Lasso。与岭回归相同，使用 lasso 也是约束系数使其接近于 0，但用到的方法不同，叫作 L1 正则化。L1 正则化的结果是，使用 lasso 时某些系数刚好为 0。这说明某些特征被模型完全忽略。这可以看作是一种自动化的特征选择。某些系数刚好为 0，这样模型更容易解释，也可以呈现模型最重要的特征。

如你所见，Lasso 在训练集与测试集上的表现都很差。这表示存在欠拟合，我们发现模型只用到了 105 个特征中的 4 个。与 Ridge 类似，Lasso 也有一个正则化参数 alpha，可以控制系数趋向于 0 的强度。为了降低欠拟合，我们尝试减小 alpha。这么做的同时，我们还需要增加 `max_iter` 的值（运行迭代的最大次数）。

alpha 值变小，我们可以拟合一个更复杂的模型，在训练集和测试集上的表现也更好。模型性能比使用 Ridge 时略好一点，而且我们只用到了 105 个特征中的 33 个。这样模型可能更容易理解。

但如果把alpha设得太小，那么就会消除正则化的效果，并出现过拟合，得到LinearRegression 类似的结果：

在实践中，在两个模型中一般首选岭回归。但如果特征很多，你认为只有其中几个是重要的，那么选择 Lasso 可能更好。同样，如果你想要一个容易解释的模型，Lasso 可以给出更容易理解的模型，因为它只选择了一部分输入特征。scikit-learn 还提供了 ElasticNet类，结合了 Lasso 和 Ridge 的惩罚项。在实践中，这种结合的效果最好，不过代价是要调节两个参数：一个用于 L1 正则化，一个用于 L2 正则化。

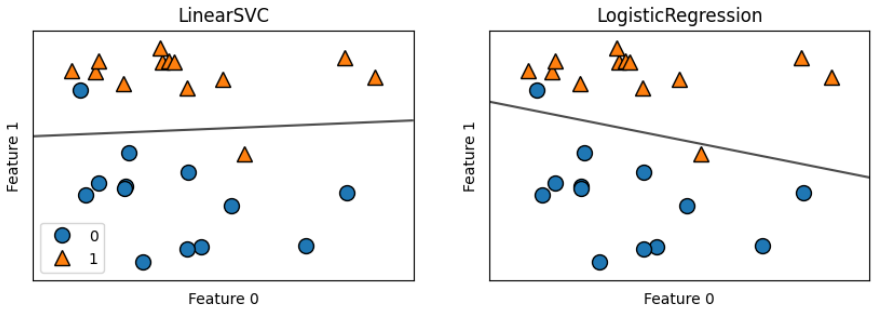

Figure 1.12: Decision boundaries of a linear SVM and logistic regression

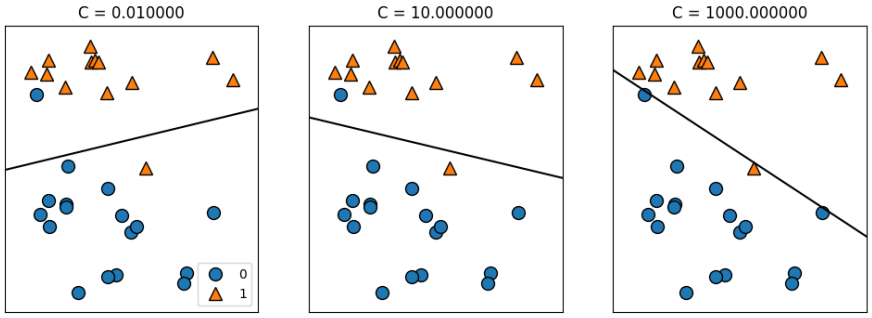

Figure 1.13: Decision boundaries of a linear SVM for different values of C

首先来看二分类。这时可以利用下面的公式进行预测：

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \cdots + w[p] * x[p] + b > 0$$

这个公式看起来与线性回归的公式非常相似，但我们没有返回特征的加权求和，而是为预测设置了阈值（0）。

对于用于回归的线性模型，输出 $\hat{y}$ 是特征的线性函数，是直线、平面或超平面（对于更高维的数据集）。对于用于分类的线性模型，**决策边界**是输入的线性函数。换句话说，（二元）线性分类器是利用直线、平面或超平面来分开两个类别的分类器。

学习线性模型有很多种算法。这些算法的区别在于以下两点：

- 系数和截距的特定组合对训练数据拟合好坏的度量方法；
- 是否使用正则化，以及使用哪种正则化方法。

由于数学上的技术原因，不可能调节 w 和 b 使得算法产生的误分类数量最少。对于我们的目的，以及对于许多应用而言，上面第一点（称为损失函数）的选择并不重要。

最常见的两种线性分类算法是 **Logistic回归**（logistic regression）和**线性支持向量机**（linear support vector machine，线性 SVM），

对于 LogisticRegression 和 LinearSVC，决定正则化强度的权衡参数叫作C。C值越大，对应的正则化越弱。换句话说，如果参数 C 值较大，那么 LogisticRegression和LinearSVC 将尽可能将训练集拟合到最好，而如果 C 值较小，那么模型更强调使系数量（*bmw*）接近于 0。

参数 C 的作用还有另一个有趣之处。较小的 C 值可以让算法尽量适应“大多数”数据点，而较大的 C 值更强调每个数据点都分类正确的重要性。

在Figure 1.13中，左图，C 值很小，对应强正则化。大部分属于类别 0 的点都位于底部，大部分属于类别 1 的点都位于顶部。强正则化的模型会选择一条相对水平的线，有两个点分类错误。中间图，C 值稍大，模型更关注两个分类错误的样本，使决策边界的斜率变大。最后，右图，模型的 C 值非常大，使得决策边界的斜率也很大，现在模型对类别 0 中所有点的分类都是正确的。类别 1 中仍有一个点分类错误，这是因为对这个数据集来说，不可能用一条直线将所有点都分类正确。右侧图中的模型尽量使所有点的分类都正确，但可能无法掌握类别的整体分布。换句话说，这个模型很可能过拟合。

与回归的情况类似，用于分类的线性模型在低维空间中看起来可能非常受限，决策边界只能是直线或平面。同样，在高维空间中，用于分类的线性模型变得非常强大，当考虑更多特征时，避免过拟合变得越来越重要。

我们在乳腺癌数据集上详细分析 LogisticRegression：

```
# ConvergenceWarning
logreg = LogisticRegression().fit(X_train, y_train)
print('Training set score: {:.3f}'.format(logreg.score(X_train, y_train)))
print('Test set score: {:.3f}'.format(logreg.score(X_test, y_test)))
# Training set score: 0.946
# Test set score: 0.965
```

C=1 的默认值给出了相当好的性能，在训练集和测试集上都达到 95% 的精度。但由于训练集和测试集的性能非常接近，所以模型很可能是欠拟合的。欠拟合的模型可以考虑增加C值来更好的拟合模型。

```
# ConvergenceWarning
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print('Training set score: {:.3f}'.format(logreg100.score(X_train, y_train)))
print('Test set score: {:.3f}'.format(logreg100.score(X_test, y_test)))
# Training set score: 0.944
# Test set score: 0.958
```

我们还可以研究使用正则化更强的模型时会发生什么。设置 $C = 0.01$：

```
# ConvergenceWarning
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print('Training set score: {:.3f}'.format(logreg001.score(X_train, y_train)))
print('Test set score: {:.3f}'.format(logreg001.score(X_test, y_test)))
# Training set score: 0.934
# Test set score: 0.930
```

第 3 个系数那里发现有趣之处，这个系数是“平均周长”（mean perimeter），它的回归系数在不同的正则参数中符号发生了改变。这也说明，对线性模型系数的解释应该始终持保留态度。

优点、缺点和参数 线性模型的主要参数是正则化参数，在回归模型中叫作 alpha，在 LinearSVC 和 Logistic-Regression 中叫作 C。alpha 值较大或 C 值较小，说明模型比较简单。特别是对于回归模型而言，调节这些参数非常重要。通常在对数尺度上对 C 和 alpha 进行搜索。你还需要确定的是用 L1 正则化还是 L2 正则化。如果你假定只有几个特征是真正重要的，那么你应该用L1 正则化，否则应默认使用 L2 正则化。如果模型的可解释性很重要的话，使用 L1 也会有帮助。由于 L1 只用到几个特征，所以更容易解释哪些特征对模型是重要的，以及这些特征的作用。

线性模型的训练速度非常快，预测速度也很快。这种模型可以推广到非常大的数据集，对稀疏数据也很有效。如果你的数据包含数十万甚至上百万个样本，你可能需要研究如何使用 LogisticRegression 和

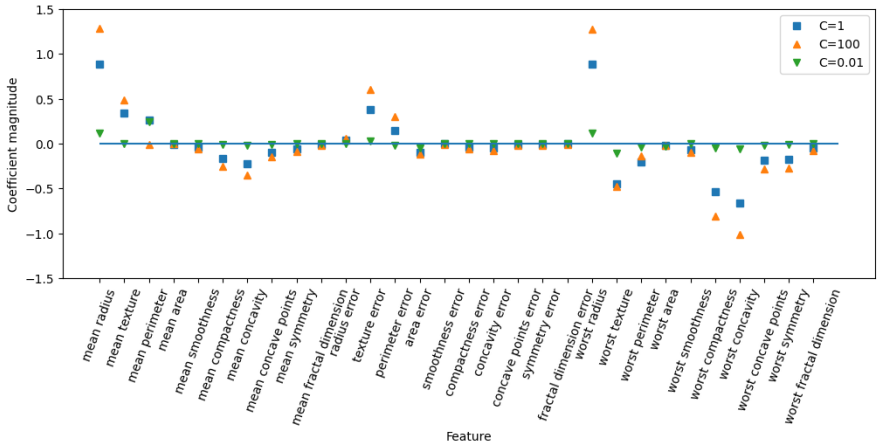

Figure 1.14: Coefficients learned by logistic regression for different C

Ridge 模型的 solver='sag' 选项，在处理大型数据时，这一选项比默认值要更快。其他选项还有 SGDClassifier 类和 SGDRegressor 类，它们对本节介绍的线性模型实现了可扩展性更强的版本。

线性模型的另一个优点在于，利用我们之间见过的用于回归和分类的公式，理解如何进行预测是相对比较容易的。不幸的是，往往并不完全清楚系数为什么是这样的。如果你的数据集中包含高度相关的特征，这一问题尤为突出。在这种情况下，可能很难对系数做出解释。

如果特征数量大于样本数量，线性模型的表现通常都很好。它也常用于非常大的数据集，只是因为训练其他模型并不可行。但在更低维的空间中，其他模型的泛化性能可能更好。

方法链

scikit-learn 中所有模型的 fit 方法返回的都是 self。

```
logreg = LinearRegression().fit(X_train, y_train)
```

这里我们利用 fit 的返回值（即 self）将训练后的模型赋值给变量 logreg。这种方法调用的拼接（先调用 `__init__`，然后调用 fit）被称为方法链（method chaining）。scikit-learn 中方法链的另一个常见用法是在一行代码中同时 fit 和 predict。

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

最后，你甚至可以在一行代码中完成模型初始化、拟合和预测：

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

不过这种非常简短的写法并不完美。一行代码中发生了很多事情，可能会使代码变得难以阅读。此外，拟合后的回归模型也没有保存在任何变量中，所以我们既不能查看它也不能用它来预测其他数据。

1.3.4 朴素贝叶斯分类器

朴素贝叶斯分类器是与线性模型非常相似的一种分类器，但它的训练速度往往更快。这种高效率所付出的代价是，朴素贝叶斯模型的泛化能力要比线性分类器（如LogisticRegression 和 LinearSVC）稍差。朴素贝叶斯模型如此高效的原因在于，它通过单独查看每个特征来学习参数，并从每个特征中收集简单的类别统计数据。

scikit-learn 中实现了三种朴素贝叶斯分类器：GaussianNB、BernoulliNB 和 MultinomialNB。GaussianNB 可应用于任意连续数据，而BernoulliNB 假定输入数据为二分类数据，MultinomialNB 假定输入数据为计数数据。BernoulliNB 和MultinomialNB 主要用于文本数据分类。

1.3.5 决策树

决策树是广泛用于分类和回归任务的模型。本质上，它从一层层的 if/else 问题中进行学习，并得出结论。这一系列问题可以表示为一棵决策树，如图 2-22 所示。

make_moons

构造决策树

控制决策树的复杂度 通常来说，构造决策树直到所有叶结点都是纯的叶结点，这会导致模型非常复杂，并且对训练数据高度过拟合。纯叶结点的存在说明这棵树在训练集上的精度是 100%。训练集中的每个数据点都位于分类正确的叶结点中。

防止过拟合有两种常见的策略：一种是及早停止树的生长，也叫预剪枝（pre-pruning）；另一种是先构造树，但随后删除或折叠信息量很少的结点，也叫后剪枝（post-pruning）或剪枝（pruning）。预剪枝的限制条件可能包括限制树的最大深度、限制叶结点的最大数目，或者规定一个结点中数据点的最小数目来防止继续划分。

scikit-learn 的决策树在 DecisionTreeRegressor 类和 DecisionTreeClassifier 类中实现。scikit-learn 只实现了预剪枝，没有实现后剪枝。

分析决策树