Introduction\_Machine\_Learning\_Python\_Andreas\_C02

APTER 2

Supervised Learning

./t.sh "As we mentioned earlier, supervised machine learning is one of the most commonly used and successful types of machine learning. In this chapter, we will describe supervised learning in more detail and explain several popular supervised learning algorithms. We already saw an application of supervised machine learning in Chapter 1: classifying iris flowers into several species using physical measurements of the flowers."

Como mencionamos anteriormente, el aprendizaje automático supervisado es uno de los tipos de aprendizaje automático más utilizados y exitosos. En este capítulo, describiremos el aprendizaje supervisado con más detalle y explicaremos varios algoritmos de aprendizaje supervisado populares. Ya vimos una aplicación del aprendizaje automático supervisado en el Capítulo 1: clasificar las flores de iris en varias especies utilizando mediciones físicas de las flores.

./t.sh "Remember that supervised learning is used whenever we want to predict a certain outcome from a given input, and we have examples of input/output pairs. We build a machine learning model from these input/output pairs, which comprise our training set. Our goal is to make accurate predictions for new, never-before-seen data. Supervised learning often requires human effort to build the training set, but afterward automates and often speeds up an otherwise laborious or infeasible task."

Recuerde que el aprendizaje supervisado se utiliza siempre que queremos predecir un determinado resultado a partir de una entrada determinada, y tenemos ejemplos de pares de entrada/salida. Construimos un modelo de aprendizaje automático a partir de estos pares de entrada/salida, que componen nuestro conjunto de entrenamiento. Nuestro objetivo es hacer predicciones precisas para datos nuevos y nunca antes vistos. El aprendizaje supervisado a menudo requiere esfuerzo humano para construir el conjunto de capacitación, pero luego automatiza y a menudo acelera una tarea que de otro modo sería laboriosa o inviable.

**Classification and Regression**

./t.sh "There are two major types of supervised machine learning problems, called classification and regression."

Hay dos tipos principales de problemas de aprendizaje automático supervisados, llamados clasificación y regresión.

./t.sh "In classification, the goal is to predict a class label, which is a choice from a predefined list of possibilities. In Chapter 1 we used the example of classifying irises into one of three possible species. Classification is sometimes separated into binary classification, which is the special case of distinguishing between exactly two classes, and multiclass classification, which is classification between more than two classes. You can think of binary classification as trying to answer a yes/no question. Classifying emails as either spam or not spam is an example of a binary classification problem. In this binary classification task, the yes/no question being asked would be “Is this email spam?”"

En la clasificación, el objetivo es predecir una etiqueta de clase, que es una elección de una lista predefinida de posibilidades. En el Capítulo 1 utilizamos el ejemplo de clasificar los lirios en una de tres especies posibles. La clasificación a veces se separa en clasificación binaria, que es el caso especial de distinguir exactamente dos clases, y clasificación multiclase, que es la clasificación entre más de dos clases. Puede pensar en la clasificación binaria como un intento de responder una pregunta de sí o no. Clasificar correos electrónicos como spam o no spam es un ejemplo de un problema de clasificación binaria. En esta tarea de clasificación binaria, la pregunta de sí o no sería "¿Este correo electrónico es spam?"

./t.sh "In binary classification we often speak of one class being the positive class and the other class being the negative class. Here, positive doesn’t represent having benefit or value, but rather what the object of the study is. So, when looking for spam, “positive” could mean the spam class. Which of the two classes is called positive is often a subjective matter, and specific to the domain."

En la clasificación binaria a menudo hablamos de que una clase es la clase positiva y la otra clase es la clase negativa. Aquí positivo no representa tener beneficio o valor, sino cuál es el objeto de estudio. Entonces, cuando se busca spam, "positivo" podría significar la clase de spam. Cuál de las dos clases se denomina positiva es a menudo una cuestión subjetiva y específica del dominio.

./t.sh "The iris example, on the other hand, is an example of a multiclass classification problem. Another example is predicting what language a website is in from the text on the website. The classes here would be a pre-defined list of possible languages."

El ejemplo del iris, por otro lado, es un ejemplo de un problema de clasificación multiclase. Otro ejemplo es predecir en qué idioma se encuentra un sitio web a partir del texto del sitio web. Las clases aquí serían una lista predefinida de posibles idiomas.

./t.sh "For regression tasks, the goal is to predict a continuous number, or a floating-point number in programming terms (or real number in mathematical terms). Predicting a person’s annual income from their education, their age, and where they live is an example of a regression task. When predicting income, the predicted value is an amount, and can be any number in a given range. Another example of a regression task is predicting the yield of a corn farm given attributes such as previous yields, weather, and number of employees working on the farm. The yield again can be an arbitrary number."

Para las tareas de regresión, el objetivo es predecir un número continuo o un número de punto flotante en términos de programación (o un número real en términos matemáticos). Predecir los ingresos anuales de una persona en función de su educación, su edad y el lugar donde vive es un ejemplo de tarea de regresión. Al predecir ingresos, el valor previsto es una cantidad y puede ser cualquier número dentro de un rango determinado. Otro ejemplo de una tarea de regresión es predecir el rendimiento de una granja de maíz teniendo en cuenta atributos como rendimientos anteriores, clima y cantidad de empleados que trabajan en la granja. El rendimiento nuevamente puede ser un número arbitrario.

./t.sh "An easy way to distinguish between classification and regression tasks is to ask whether there is some kind of continuity in the output. If there is continuity between possible outcomes, then the problem is a regression problem. Think about predicting annual income. There is a clear continuity in the output. Whether a person makes $40,000 or $40,001 a year does not make a tangible difference, even though these are different amounts of money; if our algorithm predicts $39,999 or $40,001 when it should have predicted $40,000, we don’t mind that much."

Una manera fácil de distinguir entre tareas de clasificación y regresión es preguntar si existe algún tipo de continuidad en el resultado. Si hay continuidad entre los posibles resultados, entonces el problema es un problema de regresión. Piense en predecir los ingresos anuales. Hay una clara continuidad en la producción. Que una persona gane 0,000 o 0,001 al año no supone una diferencia tangible, aunque se trate de cantidades diferentes de dinero; Si nuestro algoritmo predice 9,999 o 0,001 cuando debería haber predicho 0,000, no nos importa mucho.

./t.sh "By contrast, for the task of recognizing the language of a website (which is a classifi‐ cation problem), there is no matter of degree. A website is in one language, or it is in another. There is no continuity between languages, and there is no language that is between English and French.1"

Por el contrario, para la tarea de reconocer el idioma de un sitio web (que es un problema de clasificación), no hay cuestión de grado. Un sitio web está en un idioma o está en otro. No hay continuidad entre idiomas y no hay ningún idioma que esté entre el inglés y el francés.1

**Generalization, Overfitting, and Underfitting**

./t.sh "In supervised learning, we want to build a model on the training data and then be able to make accurate predictions on new, unseen data that has the same characteristics as the training set that we used. If a model is able to make accurate predictions on unseen data, we say it is able to generalize from the training set to the test set. We want to build a model that is able to generalize as accurately as possible." >> tt.txt

En el aprendizaje supervisado, queremos construir un modelo a partir de los datos de entrenamiento y luego poder hacer predicciones precisas a partir de datos nuevos, no vistos, que tengan las mismas características que el conjunto de entrenamiento que usamos. Si un modelo puede hacer predicciones precisas a partir de datos no vistos, decimos que puede generalizar desde el conjunto de entrenamiento al conjunto de prueba. Queremos construir un modelo que pueda generalizar con la mayor precisión posible.

./t.sh "1 We ask linguists to excuse the simplified presentation of languages as distinct and fixed entities." >> tt.txt

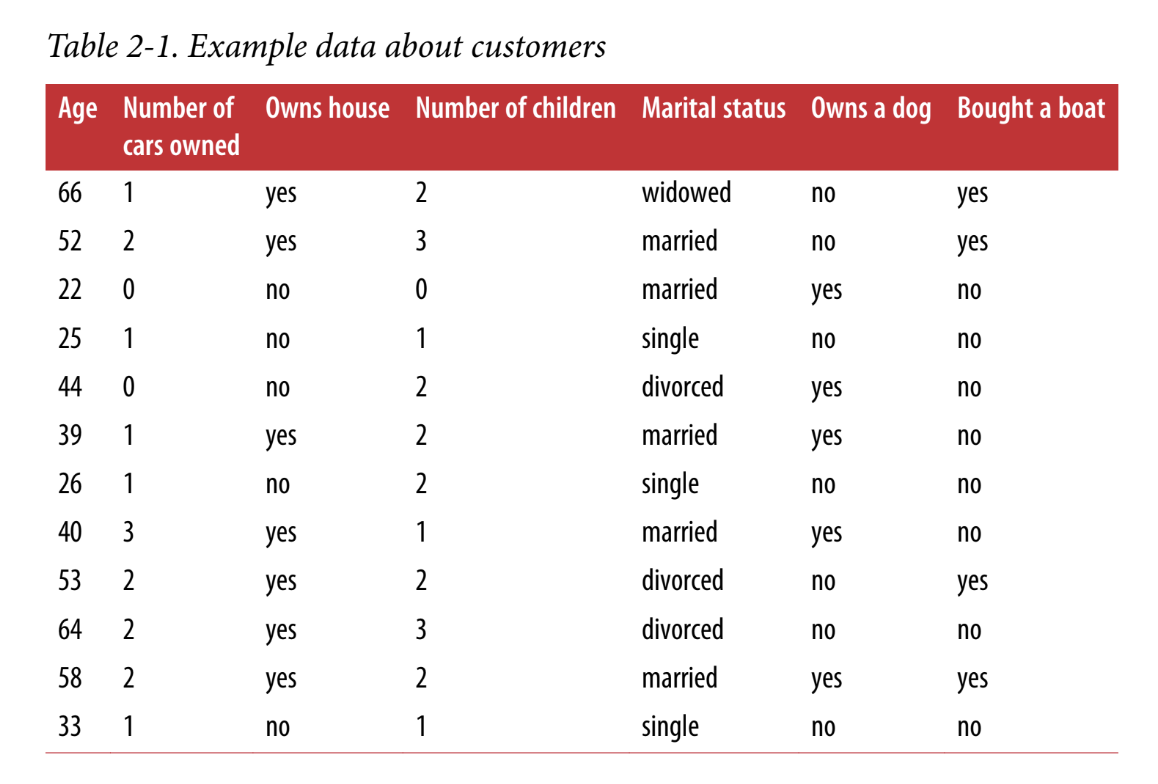
./t.sh "Usually we build a model in such a way that it can make accurate predictions on the training set. If the training and test sets have enough in common, we expect the model to also be accurate on the test set. However, there are some cases where this can go wrong. For example, if we allow ourselves to build very complex models, we can always be as accurate as we like on the training set." >> tt.txt

Normalmente, construimos un modelo de tal manera que pueda hacer predicciones precisas en el conjunto de entrenamiento. Si los conjuntos de entrenamiento y prueba tienen suficientes cosas en común, esperamos que el modelo también sea preciso en el conjunto de prueba. Sin embargo, hay algunos casos en los que esto puede salir mal. Por ejemplo, si nos permitimos construir modelos muy complejos, siempre podemos ser tan precisos como queramos en el conjunto de entrenamiento.

./t.sh "Let’s take a look at a made-up example to illustrate this point. Say a novice data scientist wants to predict whether a customer will buy a boat, given records of previous boat buyers and customers who we know are not interested in buying a boat.2 The goal is to send out promotional emails to people who are likely to actually make a purchase, but not bother those customers who won’t be interested." >> tt.txt

Veamos un ejemplo inventado para ilustrar este punto. Digamos que un científico de datos novato quiere predecir si un cliente comprará un barco, a partir de los registros de compradores de barcos anteriores y de clientes que sabemos que no están interesados ​​en comprar un barco.2 El objetivo es enviar correos electrónicos promocionales a personas que probablemente realicen una compra, pero no molestar a aquellos clientes que no estarán interesados.

Suppose we have the customer records shown in Table 2-1.



./t.sh "After looking at the data for a while, our novice data scientist comes up with the following rule: “If the customer is older than 45, and has less than 3 children or is not divorced, then they want to buy a boat.” When asked how well this rule of his does, our data scientist answers, “It’s 100 percent accurate!” And indeed, on the data that is in the table, the rule is perfectly accurate. There are many possible rules we could come up with that would explain perfectly if someone in this dataset wants to buy a boat. No age appears twice in the data, so we could say people who are 66, 52, 53, or" >> tt.txt

Después de observar los datos durante un rato, nuestro científico de datos novato propone la siguiente regla: "Si el cliente tiene más de 45 años y menos de 3 hijos o no está divorciado, entonces quiere comprar un barco". Cuando le preguntamos qué tan bien funciona esta regla, nuestro científico de datos responde: "¡Es 100 por ciento precisa!" Y, de hecho, en los datos que están en la tabla, la regla es perfectamente precisa. Hay muchas reglas posibles que podríamos idear que explicarían perfectamente si alguien en este conjunto de datos quiere comprar un barco. Ninguna edad aparece dos veces en los datos, por lo que podríamos decir que las personas que tienen 66, 52, 53 o 64 años quieren comprar un barco.

./t.sh "2 In the real world, this is actually a tricky problem. While we know that the other customers haven’t bought a boat from us yet, they might have bought one from someone else, or they may still be saving and plan to buy one in the future." >> tt.txt

En el mundo real, este es un problema complicado. Si bien sabemos que otros clientes aún no nos han comprado un barco, es posible que hayan comprado uno a otra persona o que aún estén ahorrando y planeen comprar uno en el futuro.

./t.sh "58 years old want to buy a boat, while all others don’t. While we can make up many rules that work well on this data, remember that we are not interested in making predictions for this dataset; we already know the answers for these customers. We want to know if new customers are likely to buy a boat. We therefore want to find a rule that will work well for new customers, and achieving 100 percent accuracy on the training set does not help us there. We might not expect that the rule our data scientist came up with will work very well on new customers. It seems too complex, and it is supported by very little data. For example, the “or is not divorced” part of the rule hinges on a single customer." >> tt.txt

58 años quiere comprar un barco, mientras que el resto no. Si bien podemos crear muchas reglas que funcionen bien con estos datos, recuerde que no nos interesa hacer predicciones para este conjunto de datos; ya conocemos las respuestas para estos clientes. Queremos saber si es probable que los nuevos clientes compren un barco. Por lo tanto, queremos encontrar una regla que funcione bien para los nuevos clientes, y lograr una precisión del 100 por ciento en el conjunto de entrenamiento no nos ayuda en ese aspecto. Es posible que no esperemos que la regla que se le ocurrió a nuestro científico de datos funcione muy bien con los nuevos clientes. Parece demasiado compleja y está respaldada por muy pocos datos. Por ejemplo, la parte de la regla que dice “o no está divorciado” depende de un solo cliente.

./t.sh "The only measure of whether an algorithm will perform well on new data is the evaluation on the test set. However, intuitively3 we expect simple models to generalize better to new data. If the rule was “People older than 50 want to buy a boat,” and this would explain the behavior of all the customers, we would trust it more than the rule involving children and marital status in addition to age. Therefore, we always want to find the simplest model. Building a model that is too complex for the amount of information we have, as our novice data scientist did, is called overfitting. Overfitting occurs when you fit a model too closely to the particularities of the training set and obtain a model that works well on the training set but is not able to generalize to new data. On the other hand, if your model is too simple—say, “Everybody who owns a house buys a boat”—then you might not be able to capture all the aspects of and variability in the data, and your model will do badly even on the training set. Choosing too simple a model is called underfitting." >> tt.txt

La única medida de si un algoritmo funcionará bien con nuevos datos es la evaluación en el conjunto de prueba. Sin embargo, intuitivamente3 esperamos que los modelos simples se generalicen mejor a nuevos datos. Si la regla fuera “Las personas mayores de 50 años quieren comprar un barco”, y esto explicaría el comportamiento de todos los clientes, confiaríamos más en ella que en la regla que incluye hijos y estado civil además de la edad. Por lo tanto, siempre queremos encontrar el modelo más simple. Construir un modelo que sea demasiado complejo para la cantidad de información que tenemos, como hizo nuestro científico de datos novato, se llama sobreajuste. El sobreajuste ocurre cuando ajustas un modelo demasiado de cerca a las particularidades del conjunto de entrenamiento y obtienes un modelo que funciona bien en el conjunto de entrenamiento pero que no es capaz de generalizarse a nuevos datos. Por otro lado, si tu modelo es demasiado simple (por ejemplo, “Todos los que tienen una casa compran un barco”), entonces es posible que no puedas capturar todos los aspectos y la variabilidad de los datos, y tu modelo funcionará mal incluso en el conjunto de entrenamiento. Elegir un modelo demasiado simple se llama subajuste.

./t.sh "The more complex we allow our model to be, the better we will be able to predict on the training data. However, if our model becomes too complex, we start focusing too much on each individual data point in our training set, and the model will not generalize well to new data." >> tt.txt

Cuanto más complejo sea nuestro modelo, mejor podremos hacer predicciones a partir de los datos de entrenamiento. Sin embargo, si nuestro modelo se vuelve demasiado complejo, comenzaremos a centrarnos demasiado en cada punto de datos individual de nuestro conjunto de entrenamiento y el modelo no se generalizará bien a los nuevos datos.

./t.sh "There is a sweet spot in between that will yield the best generalization performance. This is the model we want to find." >> tt.txt

Existe un punto intermedio que dará como resultado el mejor rendimiento de generalización. Este es el modelo que queremos encontrar.

./t.sh "The trade-off between overfitting and underfitting is illustrated in Figure 2-1." >> tt.txt

La compensación entre sobreajuste y subajuste se ilustra en la Figura 2-1.

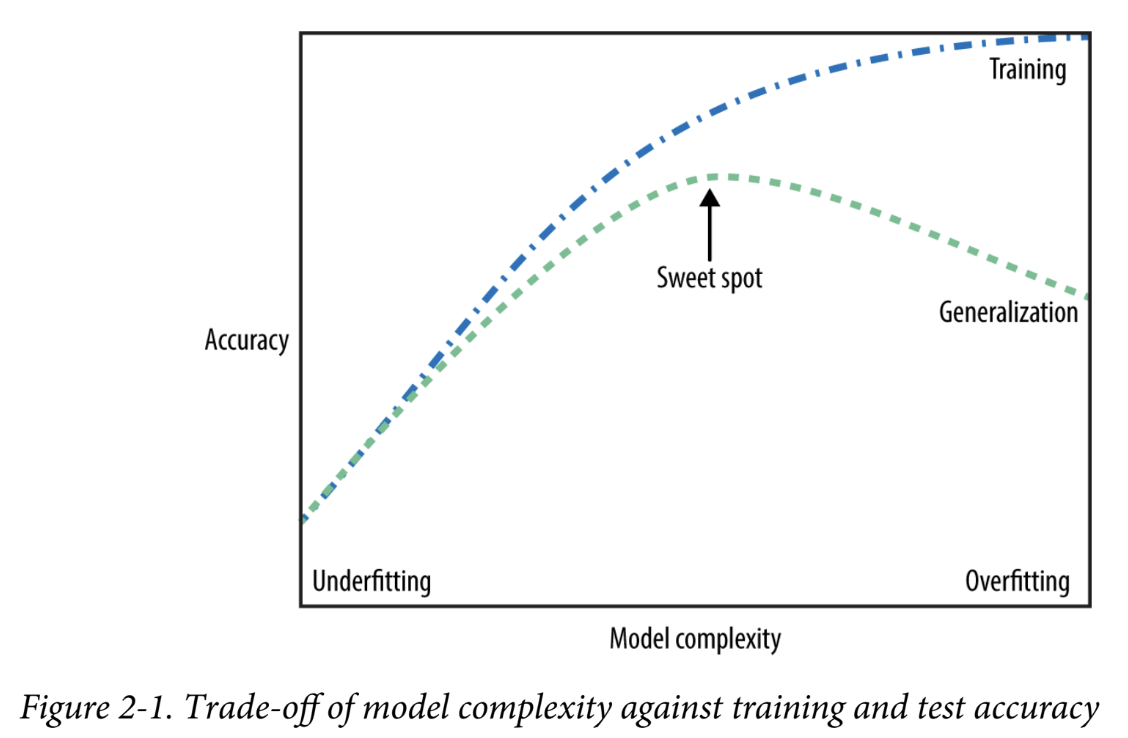


Figure 2-1. Trade-off of model complexity against training and test accuracy

**Relation of Model Complexity to Dataset Size**

./t.sh "It’s important to note that model complexity is intimately tied to the variation of inputs contained in your training dataset: the larger variety of data points your data‐set contains, the more complex a model you can use without overfitting. Usually, collecting more data points will yield more variety, so larger datasets allow building more complex models. However, simply duplicating the same data points or collecting very similar data will not help." >> tt.txt

Es importante tener en cuenta que la complejidad del modelo está íntimamente ligada a la variación de las entradas contenidas en el conjunto de datos de entrenamiento: cuanto mayor sea la variedad de puntos de datos que contenga el conjunto de datos, más complejo será el modelo que se puede utilizar sin sobreajustar. Por lo general, la recopilación de más puntos de datos dará como resultado una mayor variedad, por lo que los conjuntos de datos más grandes permiten construir modelos más complejos. Sin embargo, simplemente duplicar los mismos puntos de datos o recopilar datos muy similares no ayudará.

./t.sh "Going back to the boat selling example, if we saw 10,000 more rows of customer data, and all of them complied with the rule “If the customer is older than 45, and has less than 3 children or is not divorced, then they want to buy a boat,” we would be much more likely to believe this to be a good rule than when it was developed using only the 12 rows in Table 2-1." >> tt.txt

Volviendo al ejemplo de la venta de barcos, si viéramos 10.000 filas más de datos de clientes y todas ellas cumplieran con la regla “Si el cliente es mayor de 45 años, tiene menos de 3 hijos o no está divorciado, entonces quiere comprar un barco”, sería mucho más probable que creyéramos que se trata de una buena regla que cuando se desarrolló utilizando solo las 12 filas de la Tabla 2-1.

./t.sh "Having more data and building appropriately more complex models can often work wonders for supervised learning tasks. In this book, we will focus on working with datasets of fixed sizes. In the real world, you often have the ability to decide how much data to collect, which might be more beneficial than tweaking and tuning your model. Never underestimate the power of more data." >> tt.txt

Contar con más datos y crear modelos más complejos de forma adecuada puede resultar muy útil para las tareas de aprendizaje supervisado. En este libro, nos centraremos en trabajar con conjuntos de datos de tamaño fijo. En el mundo real, a menudo se puede decidir cuántos datos se van a recopilar, lo que puede resultar más beneficioso que modificar y ajustar el modelo. Nunca subestime el poder de contar con más datos.

**Supervised Machine Learning Algorithms**

./t.sh "We will now review the most popular machine learning algorithms and explain how they learn from data and how they make predictions. We will also discuss how the concept of model complexity plays out for each of these models, and provide an overview of how each algorithm builds a model. We will examine the strengths and weaknesses of each algorithm, and what kind of data they can best be applied to. We will also explain the meaning of the most important parameters and options.4 Many algorithms have a classification and a regression variant, and we will describe both." >> tt.txt

Ahora revisaremos los algoritmos de aprendizaje automático más populares y explicaremos cómo aprenden de los datos y cómo hacen predicciones. También analizaremos cómo se aplica el concepto de complejidad del modelo a cada uno de estos modelos y ofreceremos una descripción general de cómo cada algoritmo construye un modelo. Examinaremos las fortalezas y debilidades de cada algoritmo y a qué tipo de datos se pueden aplicar mejor. También explicaremos el significado de los parámetros y opciones más importantes.4 Muchos algoritmos tienen una clasificación y una variante de regresión, y describiremos ambas.

./t.sh "It is not necessary to read through the descriptions of each algorithm in detail, but understanding the models will give you a better feeling for the different ways machine learning algorithms can work. This chapter can also be used as a reference guide, and you can come back to it when you are unsure about the workings of any of the algorithms." >> tt.txt

No es necesario leer detalladamente las descripciones de cada algoritmo, pero comprender los modelos le dará una mejor idea de las diferentes formas en que pueden funcionar los algoritmos de aprendizaje automático. Este capítulo también se puede utilizar como guía de referencia y puede volver a él cuando no esté seguro del funcionamiento de cualquiera de los algoritmos.

**Some Sample Datasets**

./t.sh "We will use several datasets to illustrate the different algorithms. Some of the datasets will be small and synthetic (meaning made-up), designed to highlight particular aspects of the algorithms. Other datasets will be large, real-world examples." >> tt.txt

Usaremos varios conjuntos de datos para ilustrar los diferentes algoritmos. Algunos de los conjuntos de datos serán pequeños y sintéticos (es decir, inventados), diseñados para resaltar aspectos particulares de los algoritmos. Otros conjuntos de datos serán grandes ejemplos del mundo real.

./t.sh "An example of a synthetic two-class classification dataset is the forge dataset, which has two features. The following code creates a scatter plot (Figure 2-2) visualizing all of the data points in this dataset. The plot has the first feature on the x-axis and the second feature on the y-axis. As is always the case in scatter plots, each data point is represented as one dot. The color and shape of the dot indicates its class:" >> tt.txt

Un ejemplo de un conjunto de datos de clasificación sintético de dos clases es el conjunto de datos de forge, que tiene dos características. El siguiente código crea un diagrama de dispersión (Figura 2-2) que visualiza todos los puntos de datos en este conjunto de datos. La gráfica tiene la primera característica en el eje x y la segunda característica en el eje y. Como siempre ocurre en los diagramas de dispersión, cada punto de datos se representa como un punto. El color y la forma del punto indican su clase:

In[1]:

# generate dataset

X, y = mglearn.datasets.make\_forge()

# plot dataset

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y)

plt.legend(["Class 0", "Class 1"], loc=4)

plt.xlabel("First feature")

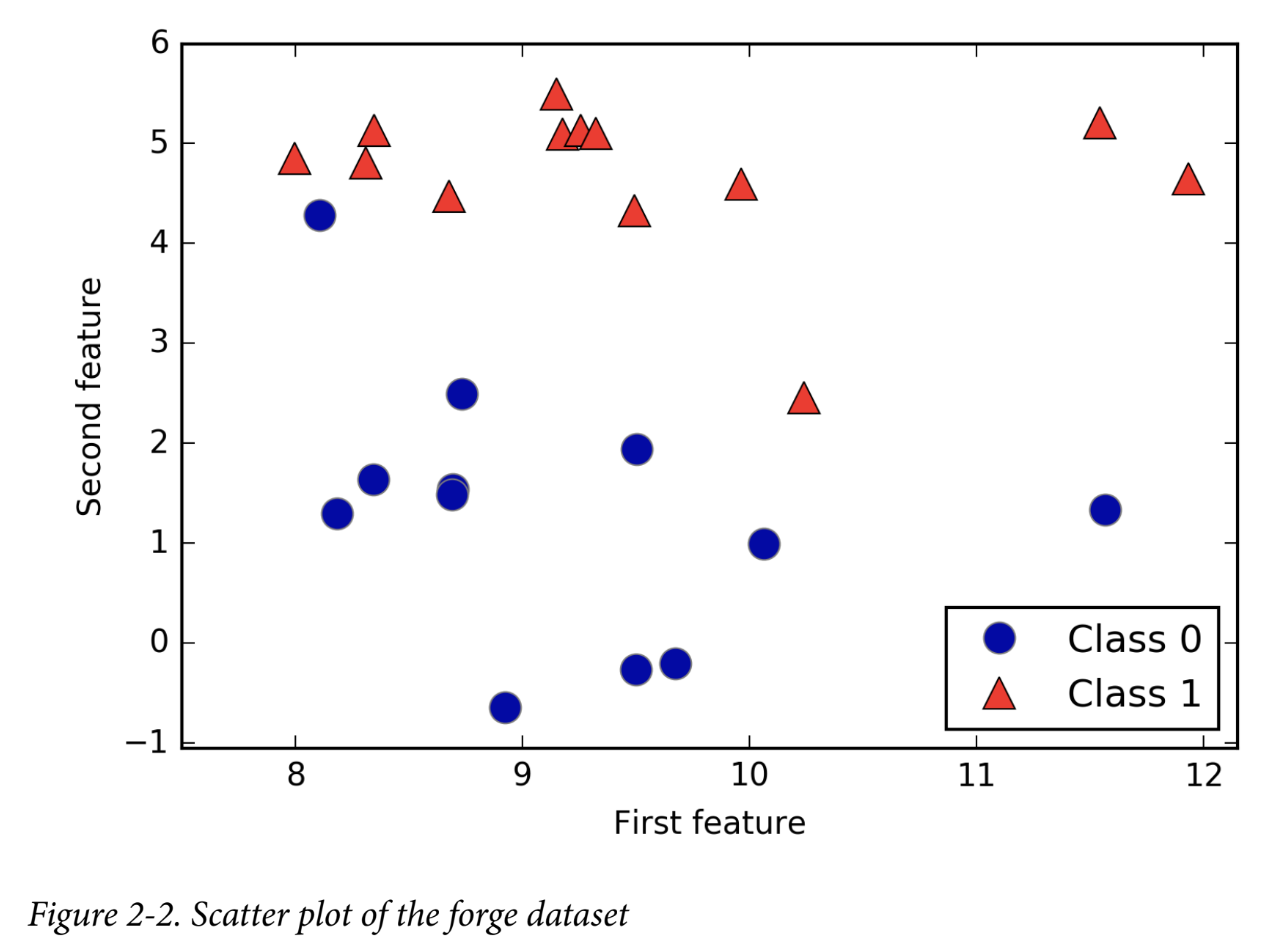
plt.ylabel("Second feature")

print("X.shape: {}".format(X.shape))

Out[1]:

X.shape: (26, 2)

4 Discussing all of them is beyond the scope of the book, and we refer you to the scikit-learn documentation for more details.



./t.sh "As you can see from X.shape, this dataset consists of 26 data points, with 2 features. To illustrate regression algorithms, we will use the synthetic wave dataset. The wave dataset has a single input feature and a continuous target variable (or response) that we want to model. The plot created here (Figure 2-3) shows the single feature on the x-axis and the regression target (the output) on the y-axis:" >> tt.txt

Como puede ver en X.shape, este conjunto de datos consta de 26 puntos de datos, con 2 características. Para ilustrar los algoritmos de regresión, utilizaremos el conjunto de datos de ondas sintéticas. El conjunto de datos de ondas tiene una única característica de entrada y una variable objetivo continua (o respuesta) que queremos modelar. El gráfico creado aquí (Figura 2-3) muestra la característica única en el eje x y el objetivo de regresión (la salida) en el eje y:

In[2]:

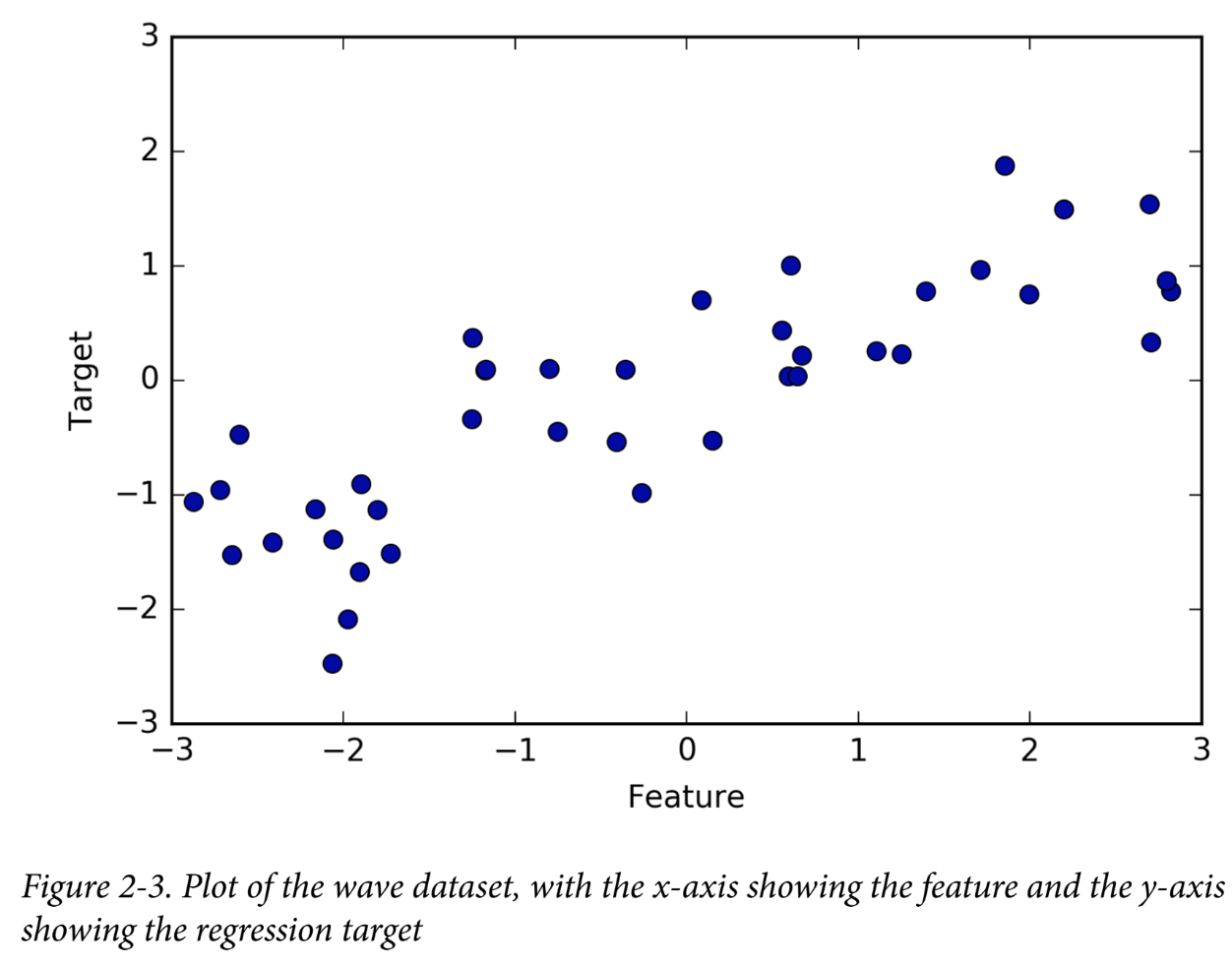
X, y = mglearn.datasets.make\_wave(n\_samples=40)

plt.plot(X, y, 'o')

plt.ylim(-3, 3)

plt.xlabel("Feature")

plt.ylabel("Target")



./t.sh "We are using these very simple, low-dimensional datasets because we can easily visualize them—a printed page has two dimensions, so data with more than two features is hard to show. Any intuition derived from datasets with few features (also called low-dimensional datasets) might not hold in datasets with many features (high- dimensional datasets). As long as you keep that in mind, inspecting algorithms on low-dimensional datasets can be very instructive." >> tt.txt

Utilizamos estos conjuntos de datos muy simples y de baja dimensión porque podemos visualizarlos fácilmente: una página impresa tiene dos dimensiones, por lo que es difícil mostrar datos con más de dos características. Cualquier intuición derivada de conjuntos de datos con pocas características (también llamados conjuntos de datos de baja dimensión) podría no ser válida en conjuntos de datos con muchas características (conjuntos de datos de alta dimensión). Siempre que tenga esto en cuenta, inspeccionar algoritmos en conjuntos de datos de baja dimensión puede ser muy instructivo.

./t.sh "We will complement these small synthetic datasets with two real-world datasets that are included in scikit-learn. One is the Wisconsin Breast Cancer dataset (cancer, for short), which records clinical measurements of breast cancer tumors. Each tumor is labeled as “benign” (for harmless tumors) or “malignant” (for cancerous tumors), and the task is to learn to predict whether a tumor is malignant based on the measurements of the tissue." >> tt.txt

Complementaremos estos pequeños conjuntos de datos sintéticos con dos conjuntos de datos del mundo real que se incluyen en scikit-learn. Uno es el conjunto de datos de cáncer de mama de Wisconsin (cáncer, para abreviar), que registra mediciones clínicas de tumores de cáncer de mama. Cada tumor está etiquetado como “benigno” (para tumores inofensivos) o “maligno” (para tumores cancerosos), y la tarea es aprender a predecir si un tumor es maligno basándose en las mediciones del tejido.

The data can be loaded using the load\_breast\_cancer function from scikit-learn:

In[3]:

from sklearn.datasets import load\_breast\_cancer

cancer = load\_breast\_cancer()

print("cancer.keys(): \n{}".format(cancer.keys()))

Out[3]:

cancer.keys():

dict\_keys(['feature\_names', 'data', 'DESCR', 'target', 'target\_names'])

./t.sh "Datasets that are included in scikit-learn are usually stored as Bunch objects, which contain some information about the dataset as well as the actual data. All you need to know about Bunch objects is that they behave like dictionaries, with the added benefit that you can access values using a dot (as in bunch.key instead of bunch['key'])." >> tt.txt

Los conjuntos de datos que se incluyen en scikit-learn generalmente se almacenan como objetos Bunch, que contienen información sobre el conjunto de datos y los datos reales. Todo lo que necesitas saber sobre los objetos Bunch es que se comportan como diccionarios, con el beneficio adicional de que puedes acceder a los valores usando un punto (como en ramo.key en lugar de ramo['clave']).

The dataset consists of 569 data points, with 30 features each:

In[4]:

print("Shape of cancer data: {}".format(cancer.data.shape))

Out[4]:

Shape of cancer data: (569, 30)

Of these 569 data points, 212 are labeled as malignant and 357 as benign:

In[5]:

print("Sample counts per class:\n{}".format(

{n: v for n, v in zip(cancer.target\_names, np.bincount(cancer.target))}))

Out[5]:

Sample counts per class:

{'benign': 357, 'malignant': 212}

To get a description of the semantic meaning of each feature, we can have a look at the feature\_names attribute:

In[6]:

print("Feature names:\n{}".format(cancer.feature\_names))

Out[6]:

Feature names:

['mean radius' 'mean texture' 'mean perimeter' 'mean area'

'mean smoothness' 'mean compactness' 'mean concavity'

'mean concave points' 'mean symmetry' 'mean fractal dimension'

'radius error' 'texture error' 'perimeter error' 'area error'

'smoothness error' 'compactness error' 'concavity error'

'concave points error' 'symmetry error' 'fractal dimension error'

'worst radius' 'worst texture' 'worst perimeter' 'worst area'

'worst smoothness' 'worst compactness' 'worst concavity'

'worst concave points' 'worst symmetry' 'worst fractal dimension']

./t.sh "You can find out more about the data by reading cancer.DESCR if you are interested. We will also be using a real-world regression dataset, the Boston Housing dataset. The task associated with this dataset is to predict the median value of homes in several Boston neighborhoods in the 1970s, using information such as crime rate, proximity to the Charles River, highway accessibility, and so on. The dataset contains 506 data points, described by 13 features:" >> tt.txt

Puede obtener más información sobre los datos leyendo cancer.DESCR si está interesado. También utilizaremos un conjunto de datos de regresión del mundo real, el conjunto de datos de Boston Housing. La tarea asociada con este conjunto de datos es predecir el valor medio de las viviendas en varios vecindarios de Boston en la década de 1970, utilizando información como la tasa de criminalidad, la proximidad al río Charles, la accesibilidad a las carreteras, etc. El conjunto de datos contiene 506 puntos de datos, descritos por 13 características:

In[7]:

from sklearn.datasets import load\_boston

boston = load\_boston()

print("Data shape: {}".format(boston.data.shape))

Out[7]:

Data shape: (506, 13)

./t.sh "Again, you can get more information about the dataset by reading the DESCR attribute of boston. For our purposes here, we will actually expand this dataset by not only considering these 13 measurements as input features, but also looking at all products (also called interactions) between features. In other words, we will not only consider crime rate and highway accessibility as features, but also the product of crime rate and highway accessibility. Including derived feature like these is called feature engineering, which we will discuss in more detail in Chapter 4. This derived dataset can be loaded using the load\_extended\_boston function::" >> tt.txt

Nuevamente, puede obtener más información sobre el conjunto de datos leyendo el atributo DESCR de Boston. Para nuestros propósitos aquí, en realidad ampliaremos este conjunto de datos no solo considerando estas 13 mediciones como características de entrada, sino también analizando todos los productos (también llamados interacciones) entre características. En otras palabras, no sólo consideraremos la tasa de criminalidad y la accesibilidad a las carreteras como características, sino también el producto de la tasa de criminalidad y la accesibilidad a las carreteras. Incluir características derivadas como estas se llama ingeniería de características, que discutiremos con más detalle en el Capítulo 4. Este conjunto de datos derivados se puede cargar usando la función load\_extended\_boston::

In[8]:

X, y = mglearn.datasets.load\_extended\_boston()

print("X.shape: {}".format(X.shape))

Out[8]:

X.shape: (506, 104)

./t.sh "The resulting 104 features are the 13 original features together with the 91 possible combinations of two features within those 13 (with replacement).5 We will use these datasets to explain and illustrate the properties of the different machine learning algorithms. But for now, let’s get to the algorithms themselves. First, we will revisit the k-nearest neighbors (k-NN) algorithm that we saw in the previous chapter." >> tt.txt

Las 104 características resultantes son las 13 características originales junto con las 91 combinaciones posibles de dos características dentro de esas 13 (con reemplazo).5 Usaremos estos conjuntos de datos para explicar e ilustrar las propiedades de los diferentes algoritmos de aprendizaje automático. Pero por ahora, vayamos a los algoritmos en sí. Primero, revisaremos el algoritmo de k vecinos más cercanos (k-NN) que vimos en el capítulo anterior.

5 This is 13 interactions for the first feature, plus 12 for the second not involving the first, plus 11 for the third and so on (13 + 12 + 11 + … + 1 = 91).

**k-Nearest Neighbors**

./t.sh "The k-NN algorithm is arguably the simplest machine learning algorithm. Building the model consists only of storing the training dataset. To make a prediction for a new data point, the algorithm finds the closest data points in the training dataset—its “nearest neighbors.”" >> tt.txt

El algoritmo k-NN es posiblemente el algoritmo de aprendizaje automático más simple. La construcción del modelo consiste únicamente en almacenar el conjunto de datos de entrenamiento. Para hacer una predicción para un nuevo punto de datos, el algoritmo encuentra los puntos de datos más cercanos en el conjunto de datos de entrenamiento: sus "vecinos más cercanos".

**k-Neighbors classification**

./t.sh "In its simplest version, the k-NN algorithm only considers exactly one nearest neighbor, which is the closest training data point to the point we want to make a prediction for. The prediction is then simply the known output for this training point. Figure 2-4 illustrates this for the case of classification on the forge dataset:" >> tt.txt

En su versión más simple, el algoritmo k-NN solo considera exactamente un vecino más cercano, que es el punto de datos de entrenamiento más cercano al punto para el que queremos hacer una predicción. La predicción es entonces simplemente el resultado conocido para este punto de entrenamiento. La Figura 2-4 ilustra esto para el caso de clasificación en el conjunto de datos de Forge:

In[9]:

mglearn.plots.plot\_knn\_classification(n\_neighbors=1)

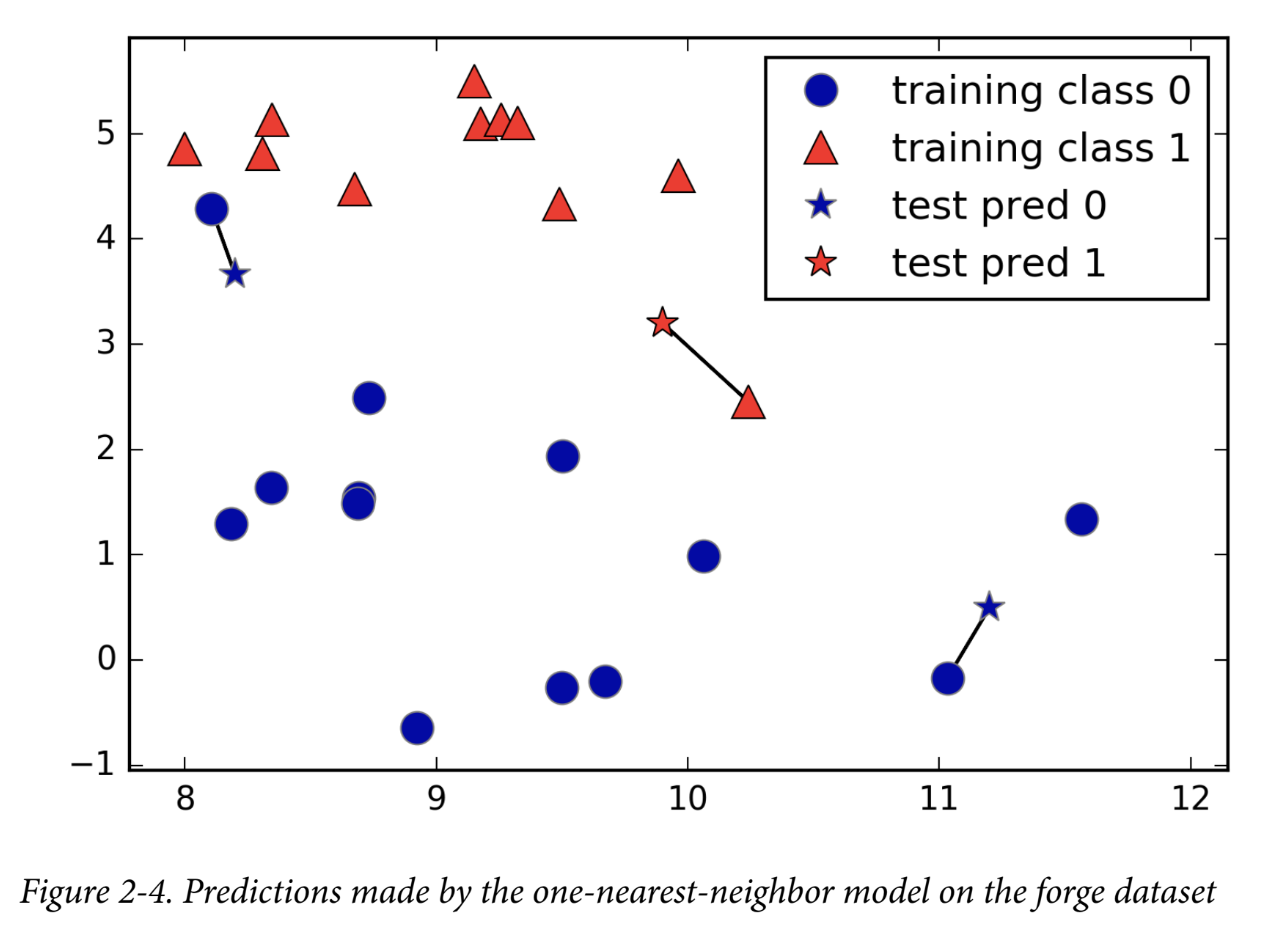


Figure 2-4. Predictions made by the one-nearest-neighbor model on the forge dataset

./t.sh "Here, we added three new data points, shown as stars. For each of them, we marked the closest point in the training set. The prediction of the one-nearest-neighbor algorithm is the label of that point (shown by the color of the cross)." >> tt.txt

Aquí, agregamos tres nuevos puntos de datos, que se muestran como estrellas. Para cada uno de ellos, marcamos el punto más cercano en el conjunto de entrenamiento. La predicción del algoritmo de un vecino más cercano es la etiqueta de ese punto (que se muestra con el color de la cruz).

./t.sh "Instead of considering only the closest neighbor, we can also consider an arbitrary number, k, of neighbors. This is where the name of the k-nearest neighbors algorithm comes from. When considering more than one neighbor, we use voting to assign a label. This means that for each test point, we count how many neighbors belong to class 0 and how many neighbors belong to class 1. We then assign the class that is more frequent: in other words, the majority class among the k-nearest neighbors. The following example (Figure 2-5) uses the three closest neighbors:" >> tt.txt

En lugar de considerar sólo el vecino más cercano, también podemos considerar un número arbitrario, k, de vecinos. De aquí proviene el nombre del algoritmo de k vecinos más cercanos. Cuando consideramos más de un vecino, utilizamos la votación para asignar una etiqueta. Esto significa que para cada punto de prueba, contamos cuántos vecinos pertenecen a la clase 0 y cuántos vecinos pertenecen a la clase 1. Luego asignamos la clase que es más frecuente: en otras palabras, la clase mayoritaria entre los k vecinos más cercanos. El siguiente ejemplo (Figura 2-5) utiliza los tres vecinos más cercanos:

In[10]:

mglearn.plots.plot\_knn\_classification(n\_neighbors=3)

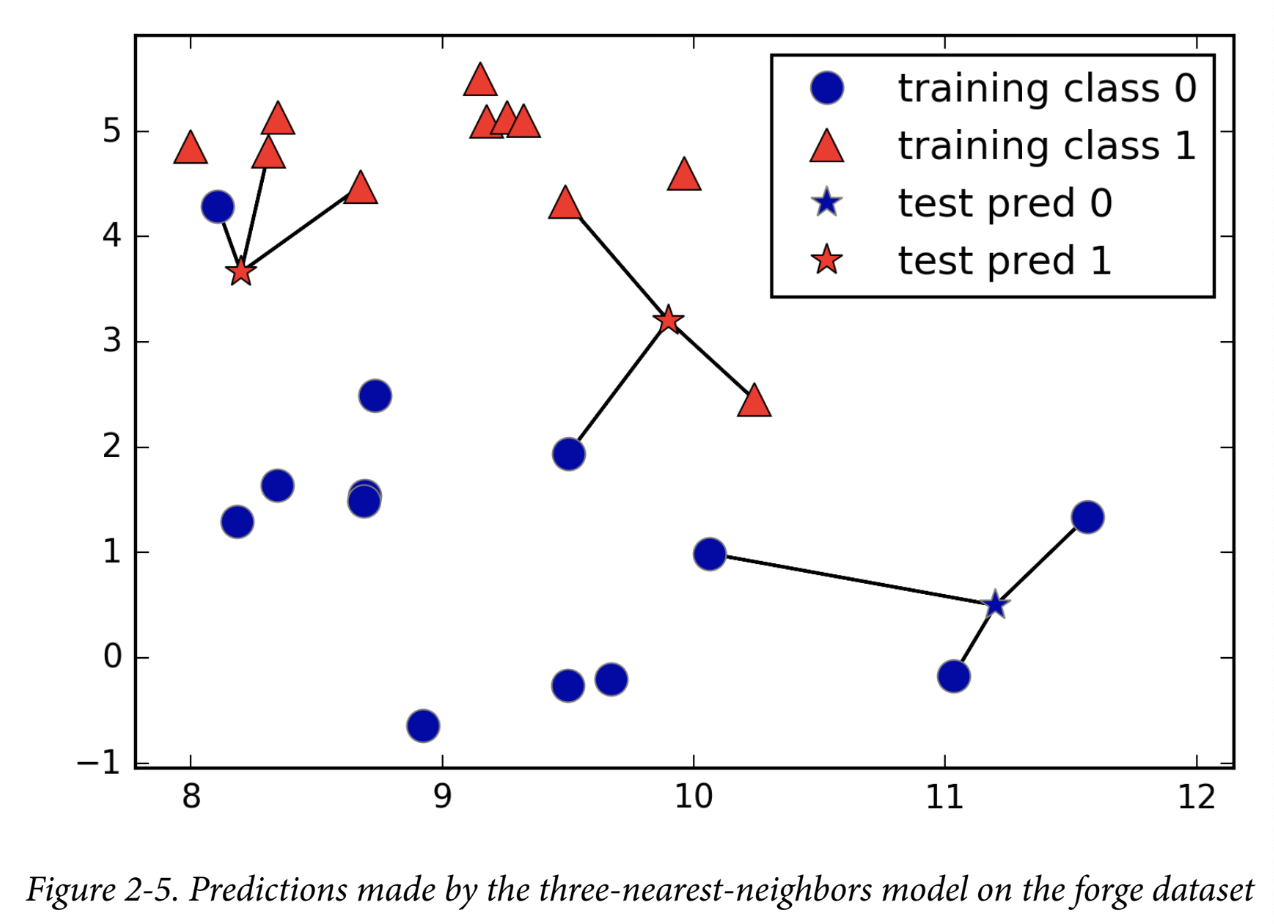


Figure 2-5. Predictions made by the three-nearest-neighbors model on the forge dataset

./t.sh "Again, the prediction is shown as the color of the cross. You can see that the prediction for the new data point at the top left is not the same as the prediction when we used only one neighbor." >> tt.txt

Nuevamente, la predicción se muestra como el color de la cruz. Puedes ver que la predicción para el nuevo punto de datos en la parte superior izquierda no es la misma que la predicción cuando usamos solo un vecino.

./t.sh "While this illustration is for a binary classification problem, this method can be applied to datasets with any number of classes. For more classes, we count how many neighbors belong to each class and again predict the most common class." >> tt.txt

Si bien esta ilustración corresponde a un problema de clasificación binaria, este método se puede aplicar a conjuntos de datos con cualquier cantidad de clases. Para más clases, contamos cuántos vecinos pertenecen a cada clase y nuevamente predecimos la clase más común.

./t.sh "Now let’s look at how we can apply the k-nearest neighbors algorithm using scikit- learn. First, we split our data into a training and a test set so we can evaluate generalization performance, as discussed in Chapter 1:" >> tt.txt

Ahora veamos cómo podemos aplicar el algoritmo de k vecinos más cercanos usando scikit-learn. Primero, dividimos nuestros datos en un conjunto de entrenamiento y de prueba para que podamos evaluar el rendimiento de la generalización, como se analiza en el Capítulo 1:

In[11]:

from sklearn.model\_selection import train\_test\_split

X, y = mglearn.datasets.make\_forge()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

./t.sh "Next, we import and instantiate the class. This is when we can set parameters, like the number of neighbors to use. Here, we set it to 3:" >> tt.txt

In[12]:

from sklearn.neighbors import KNeighborsClassifier

clf = KNeighborsClassifier(n\_neighbors=3)

Now, we fit the classifier using the training set. For KNeighborsClassifier this

means storing the dataset, so we can compute neighbors during prediction:

In[13]:

clf.fit(X\_train, y\_train)

./t.sh "To make predictions on the test data, we call the predict method. For each data point in the test set, this computes its nearest neighbors in the training set and finds the most common class among these:" >> tt.txt

Para hacer predicciones sobre los datos de prueba, utilizamos el método de predicción. Para cada punto de datos del conjunto de prueba, calcula sus vecinos más cercanos en el conjunto de entrenamiento y encuentra la clase más común entre ellos:

In[14]:

print("Test set predictions: {}".format(clf.predict(X\_test)))

Out[14]:

Test set predictions: [1 0 1 0 1 0 0]

./t.sh "To evaluate how well our model generalizes, we can call the score method with the test data together with the test labels:" >> tt.txt

Para evaluar qué tan bien se generaliza nuestro modelo, podemos llamar al método de puntuación con los datos de prueba junto con las etiquetas de prueba:

In[15]:

print("Test set accuracy: {:.2f}".format(clf.score(X\_test, y\_test)))

Out[15]:

Test set accuracy: 0.86

./t.sh "We see that our model is about 86% accurate, meaning the model predicted the class correctly for 86% of the samples in the test dataset." >> tt.txt

Vemos que nuestro modelo tiene una precisión de aproximadamente el 86 %, lo que significa que el modelo predijo la clase correctamente para el 86 % de las muestras en el conjunto de datos de prueba.

**Analyzing KNeighborsClassifier**

For two-dimensional datasets, we can also illustrate the prediction for all possible test points in the xy-plane. We color the plane according to the class that would be assigned to a point in this region. This lets us view the decision boundary, which is the divide between where the algorithm assigns class 0 versus where it assigns class 1.

En el caso de conjuntos de datos bidimensionales, también podemos ilustrar la predicción para todos los puntos de prueba posibles en el plano xy. Coloreamos el plano según la clase que se asignaría a un punto en esta región. Esto nos permite ver el límite de decisión, que es la división entre el lugar donde el algoritmo asigna la clase 0 y el lugar donde asigna la clase 1.

The following code produces the visualizations of the decision boundaries for one, three, and nine neighbors shown in Figure 2-6:

El siguiente código produce las visualizaciones de los límites de decisión para uno, tres y nueve vecinos que se muestran en la Figura 2-6:

In[16]:

fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n\_neighbors, ax in zip([1, 3, 9], axes):

# the fit method returns the object self, so we can instantiate

# and fit in one line

clf = KNeighborsClassifier(n\_neighbors=n\_neighbors).fit(X, y)

mglearn.plots.plot\_2d\_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)

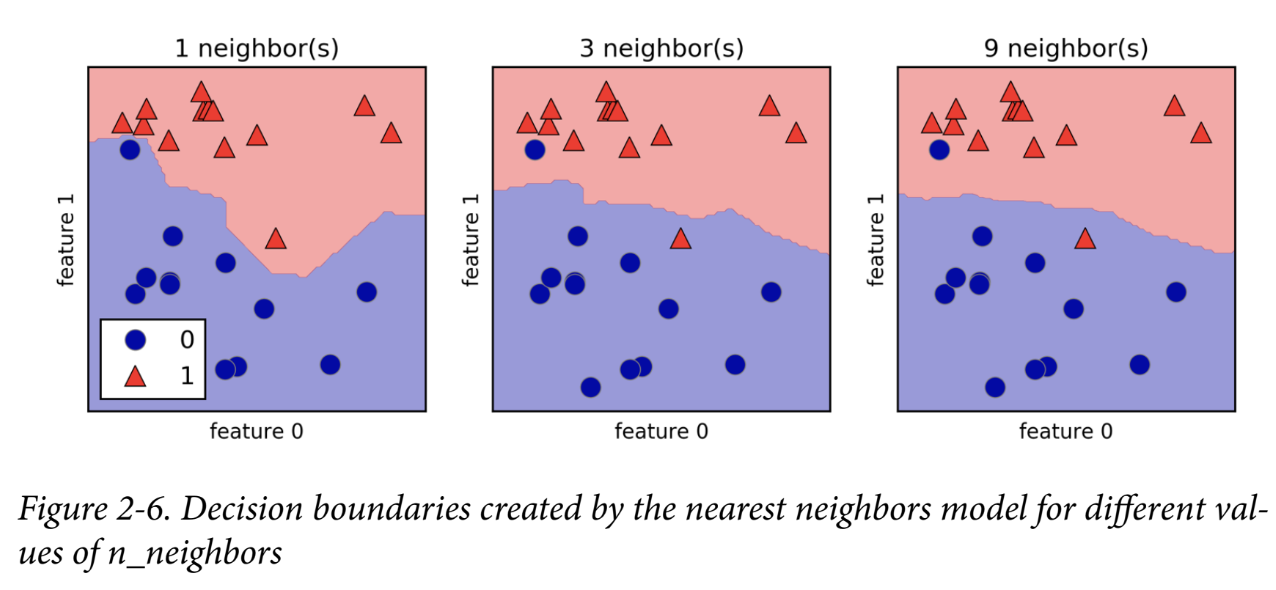
mglearn.discrete\_scatter(X[:, 0], X[:, 1], y, ax=ax)

ax.set\_title("{} neighbor(s)".format(n\_neighbors))

ax.set\_xlabel("feature 0")

ax.set\_ylabel("feature 1")

axes[0].legend(loc=3)



As you can see on the left in the figure, using a single neighbor results in a decision boundary that follows the training data closely. Considering more and more neighbors leads to a smoother decision boundary. A smoother boundary corresponds to a simpler model. In other words, using few neighbors corresponds to high model complexity (as shown on the left side of Figure 2-1), and using many neighbors corresponds to low model complexity (as shown on the right side of Figure 2-1). If you consider the extreme case where the number of neighbors is the number of all data points in the training set, each test point would have exactly the same neighbors (all training points) and all predictions would be the same: the class that is most frequent in the training set.

Como puede ver a la izquierda de la figura, el uso de un solo vecino da como resultado un límite de decisión que sigue de cerca los datos de entrenamiento. Si se consideran más y más vecinos, se obtiene un límite de decisión más suave. Un límite más suave corresponde a un modelo más simple. En otras palabras, el uso de pocos vecinos corresponde a una alta complejidad del modelo (como se muestra en el lado izquierdo de la Figura 2-1), y el uso de muchos vecinos corresponde a una baja complejidad del modelo (como se muestra en el lado derecho de la Figura 2-1). Si considera el caso extremo en el que la cantidad de vecinos es la cantidad de todos los puntos de datos en el conjunto de entrenamiento, cada punto de prueba tendría exactamente los mismos vecinos (todos los puntos de entrenamiento) y todas las predicciones serían las mismas: la clase que es más frecuente en el conjunto de entrenamiento.

Let’s investigate whether we can confirm the connection between model complexity and generalization that we discussed earlier. We will do this on the real-world Breast Cancer dataset. We begin by splitting the dataset into a training and a test set. Then we evaluate training and test set performance with different numbers of neighbors.

Investiguemos si podemos confirmar la conexión entre la complejidad del modelo y la generalización que analizamos anteriormente. Lo haremos con el conjunto de datos de cáncer de mama del mundo real. Comenzamos dividiendo el conjunto de datos en un conjunto de entrenamiento y uno de prueba. Luego evaluamos el rendimiento del conjunto de entrenamiento y de prueba con diferentes cantidades de vecinos.

The results are shown in Figure 2-7:

In[17]:

from sklearn.datasets import load\_breast\_cancer

cancer = load\_breast\_cancer()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

cancer.data, cancer.target, stratify=cancer.target, random\_state=66)

training\_accuracy = []

test\_accuracy = []

# try n\_neighbors from 1 to 10

neighbors\_settings = range(1, 11)

for n\_neighbors in neighbors\_settings:

# build the model

clf = KNeighborsClassifier(n\_neighbors=n\_neighbors)

clf.fit(X\_train, y\_train)

# record training set accuracy

training\_accuracy.append(clf.score(X\_train, y\_train))

# record generalization accuracy

test\_accuracy.append(clf.score(X\_test, y\_test))

plt.plot(neighbors\_settings, training\_accuracy, label="training accuracy")

plt.plot(neighbors\_settings, test\_accuracy, label="test accuracy")

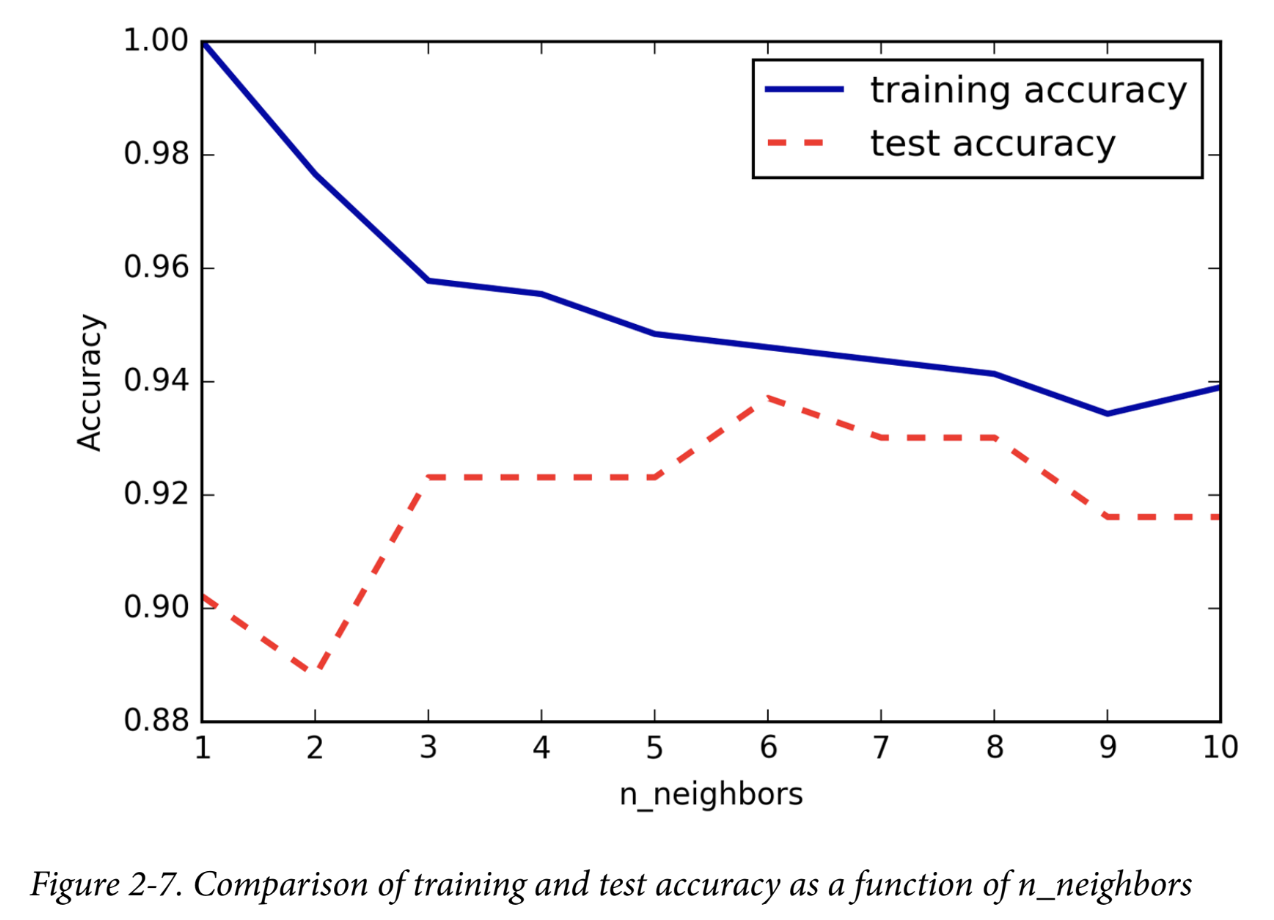
plt.ylabel("Accuracy")

plt.xlabel("n\_neighbors")

plt.legend()

The plot shows the training and test set accuracy on the y-axis against the setting of n\_neighbors on the x-axis. While real-world plots are rarely very smooth, we can still recognize some of the characteristics of overfitting and underfitting (note that because considering fewer neighbors corresponds to a more complex model, the plot is horizontally flipped relative to the illustration in Figure 2-1). Considering a single nearest neighbor, the prediction on the training set is perfect. But when more neighbors are considered, the model becomes simpler and the training accuracy drops. The test set accuracy for using a single neighbor is lower than when using more neighbors, indicating that using the single nearest neighbor leads to a model that is too complex. On the other hand, when considering 10 neighbors, the model is too simple and performance is even worse. The best performance is somewhere in the middle, using around six neighbors. Still, it is good to keep the scale of the plot in mind. The worst performance is around 88% accuracy, which might still be acceptable.

El gráfico muestra la precisión del conjunto de entrenamiento y prueba en el eje y frente a la configuración de n\_vecinos en el eje x. Si bien los gráficos del mundo real rara vez son muy uniformes, aún podemos reconocer algunas de las características del sobreajuste y el subajuste (tenga en cuenta que, dado que considerar menos vecinos corresponde a un modelo más complejo, el gráfico está invertido horizontalmente en relación con la ilustración de la Figura 2-1). Si se considera un solo vecino más cercano, la predicción en el conjunto de entrenamiento es perfecta. Pero cuando se consideran más vecinos, el modelo se vuelve más simple y la precisión del entrenamiento disminuye. La precisión del conjunto de prueba para usar un solo vecino es menor que cuando se usan más vecinos, lo que indica que usar el único vecino más cercano conduce a un modelo demasiado complejo. Por otro lado, cuando se consideran 10 vecinos, el modelo es demasiado simple y el rendimiento es incluso peor. El mejor rendimiento está en algún punto intermedio, utilizando alrededor de seis vecinos. Aun así, es bueno tener en cuenta la escala del gráfico. El peor rendimiento está en torno al 88 % de precisión, que aún podría ser aceptable.

|

**k-neighbors regression**

There is also a regression variant of the k-nearest neighbors algorithm. Again, let’s start by using the single nearest neighbor, this time using the wave dataset. We’ve added three test data points as green stars on the x-axis. The prediction using a single neighbor is just the target value of the nearest neighbor. These are shown as blue stars in Figure 2-8:

También existe una variante de regresión del algoritmo de los k vecinos más cercanos. Nuevamente, comencemos utilizando el vecino más cercano, esta vez utilizando el conjunto de datos de ondas. Hemos agregado tres puntos de datos de prueba como estrellas verdes en el eje x. La predicción utilizando un solo vecino es solo el valor objetivo del vecino más cercano. Estos se muestran como estrellas azules en la Figura 2-8:

In[18]:

mglearn.plots.plot\_knn\_regression(n\_neighbors=1)

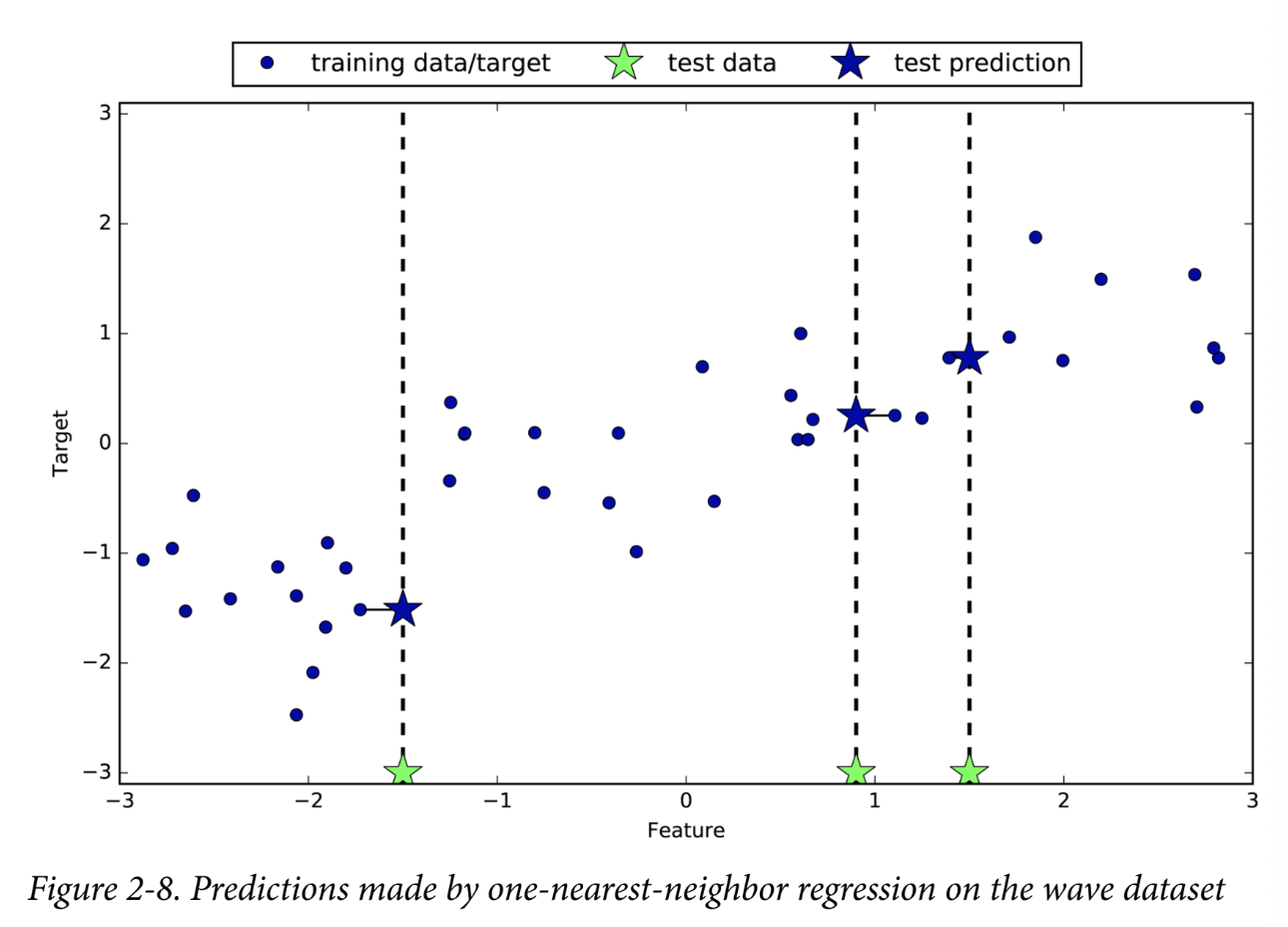


Figure 2-8. Predictions made by one-nearest-neighbor regression on the wave dataset

Again, we can use more than the single closest neighbor for regression. When using multiple nearest neighbors, the prediction is the average, or mean, of the relevant neighbors (Figure 2-9):

Nuevamente, podemos utilizar más de un vecino más cercano para la regresión. Cuando se utilizan varios vecinos más cercanos, la predicción es el promedio o la media de los vecinos relevantes (Figura 2-9):

In[19]:

mglearn.plots.plot\_knn\_regression(n\_neighbors=3)

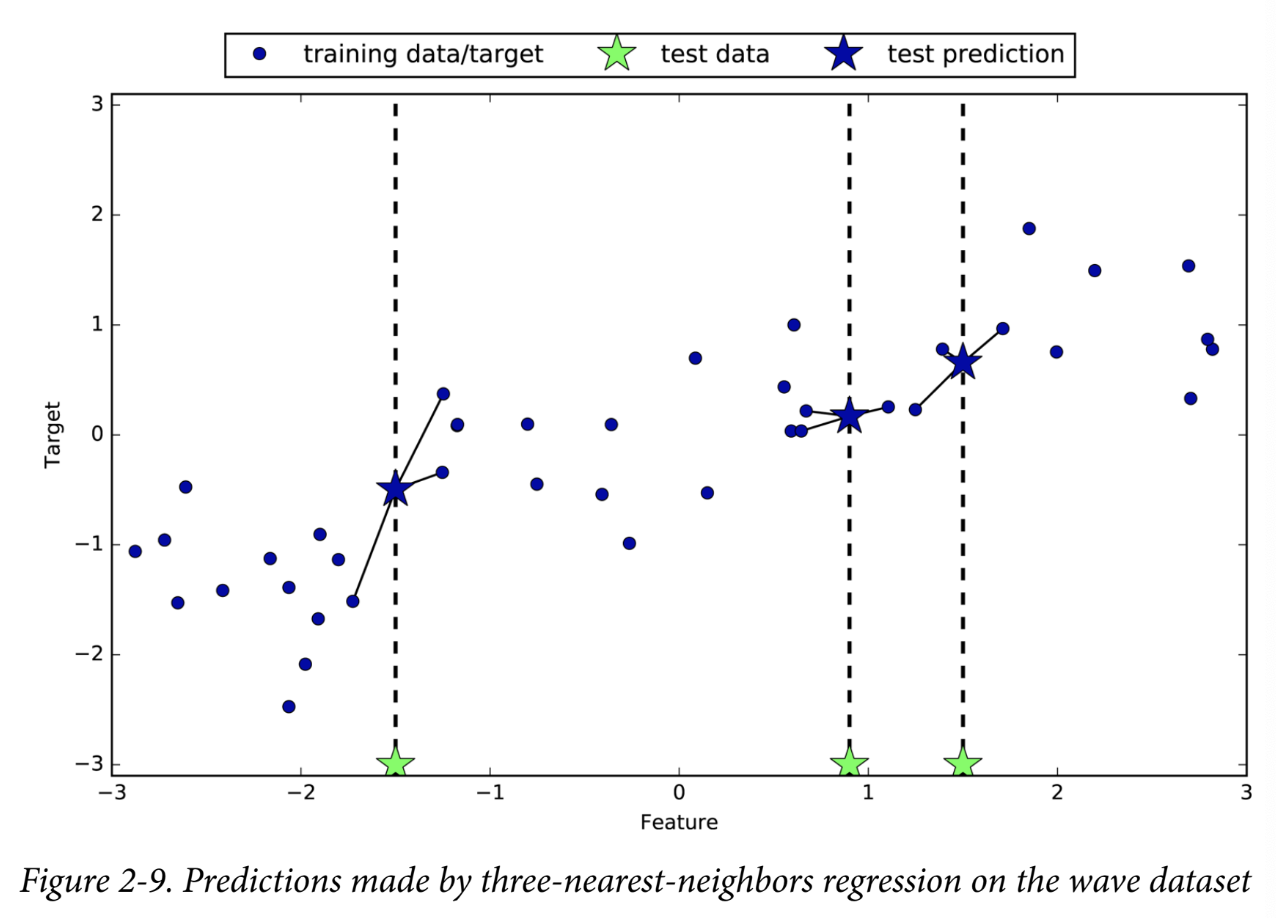


Figure 2-9. Predictions made by three-nearest-neighbors regression on the wave dataset

The k-nearest neighbors algorithm for regression is implemented in the KNeighbors Regressor class in scikit-learn. It’s used similarly to KNeighborsClassifier:

El algoritmo de k vecinos más cercanos para la regresión se implementa en la clase KNeighbors Regressor en scikit-learn. Se utiliza de forma similar a KNeighborsClassifier:

In[20]:

from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make\_wave(n\_samples=40)

# split the wave dataset into a training and a test set

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

# instantiate the model and set the number of neighbors to consider to 3

reg = KNeighborsRegressor(n\_neighbors=3)

# fit the model using the training data and training targets

reg.fit(X\_train, y\_train)

Now we can make predictions on the test set:

In[21]:

print("Test set predictions:\n{}".format(reg.predict(X\_test)))

Out[21]:

Test set predictions:

[-0.054 0.357 1.137 -1.894 -1.139 -1.631

0.357

0.912 -0.447 -1.139]

We can also evaluate the model using the score method, which for regressors returns the R2 score. The R2 score, also known as the coefficient of determination, is a measure of goodness of a prediction for a regression model, and yields a score between 0 and 1. A value of 1 corresponds to a perfect prediction, and a value of 0 corresponds to a constant model that just predicts the mean of the training set responses, y\_train:

También podemos evaluar el modelo utilizando el método de puntuación, que para los regresores devuelve la puntuación R2. La puntuación R2, también conocida como coeficiente de determinación, es una medida de la bondad de una predicción para un modelo de regresión y arroja una puntuación entre 0 y 1. Un valor de 1 corresponde a una predicción perfecta y un valor de 0 corresponde a un modelo constante que solo predice la media de las respuestas del conjunto de entrenamiento, y\_train:

In[22]:

print("Test set R^2: {:.2f}".format(reg.score(X\_test, y\_test)))

Out[22]:

Test set R^2: 0.83

Here, the score is 0.83, which indicates a relatively good model fit.

**Analyzing KNeighborsRegressor**

For our one-dimensional dataset, we can see what the predictions look like for all possible feature values (Figure 2-10). To do this, we create a test dataset consisting of many points on the x-axis, which corresponds to the single feature:

Para nuestro conjunto de datos unidimensional, podemos ver cómo se ven las predicciones para todos los valores de características posibles (Figura 2-10). Para ello, creamos un conjunto de datos de prueba que consta de muchos puntos en el eje x, que corresponde a la característica única:

In[23]:

fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# create 1,000 data points, evenly spaced between -3 and 3

line = np.linspace(-3, 3, 1000).reshape(-1, 1)

for n\_neighbors, ax in zip([1, 3, 9], axes):

# make predictions using 1, 3, or 9 neighbors

reg = KNeighborsRegressor(n\_neighbors=n\_neighbors)

reg.fit(X\_train, y\_train)

ax.plot(line, reg.predict(line))

ax.plot(X\_train, y\_train, '^', c=mglearn.cm2(0), markersize=8)

ax.plot(X\_test, y\_test, 'v', c=mglearn.cm2(1), markersize=8)

ax.set\_title(

"{} neighbor(s)\n train score: {:.2f} test score: {:.2f}".format(

n\_neighbors, reg.score(X\_train, y\_train),

reg.score(X\_test, y\_test)))

ax.set\_xlabel("Feature")

ax.set\_ylabel("Target")

axes[0].legend(["Model predictions", "Training data/target",

"Test data/target"], loc="best")

Supervised Machine Learning Algorithms

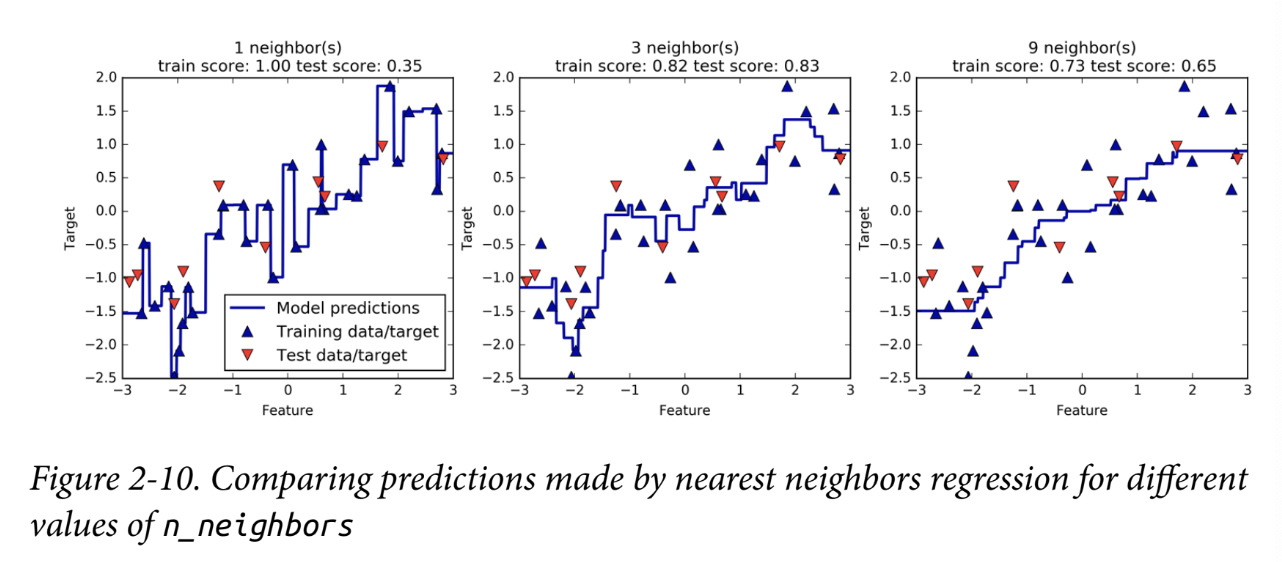
|

Figure 2-10. Comparing predictions made by nearest neighbors regression for different values of n\_neighbors

As we can see from the plot, using only a single neighbor, each point in the training set has an obvious influence on the predictions, and the predicted values go through all of the data points. This leads to a very unsteady prediction. Considering more neighbors leads to smoother predictions, but these do not fit the training data as well.

Como podemos ver en el gráfico, al utilizar un solo vecino, cada punto del conjunto de entrenamiento tiene una influencia obvia en las predicciones, y los valores predichos pasan por todos los puntos de datos. Esto genera una predicción muy inestable. Si se consideran más vecinos, se obtienen predicciones más uniformes, pero estas no se ajustan tan bien a los datos de entrenamiento.

**Strengths, weaknesses, and parameters**

In principle, there are two important parameters to the KNeighbors classifier: the number of neighbors and how you measure distance between data points. In practice, using a small number of neighbors like three or five often works well, but you should certainly adjust this parameter. Choosing the right distance measure is somewhat beyond the scope of this book. By default, Euclidean distance is used, which works well in many settings.

En principio, el clasificador KNeighbors tiene dos parámetros importantes: la cantidad de vecinos y la forma de medir la distancia entre los puntos de datos. En la práctica, utilizar una cantidad pequeña de vecinos, como tres o cinco, suele funcionar bien, pero conviene ajustar este parámetro. Elegir la medida de distancia adecuada queda fuera del alcance de este libro. De forma predeterminada, se utiliza la distancia euclidiana, que funciona bien en muchos entornos.

One of the strengths of k-NN is that the model is very easy to understand, and often gives reasonable performance without a lot of adjustments. Using this algorithm is a good baseline method to try before considering more advanced techniques. Building the nearest neighbors model is usually very fast, but when your training set is very large (either in number of features or in number of samples) prediction can be slow. When using the k-NN algorithm, it’s important to preprocess your data (see Chapter 3). This approach often does not perform well on datasets with many features (hundreds or more), and it does particularly badly with datasets where most features are 0 most of the time (so-called sparse datasets).

Una de las ventajas de k-NN es que el modelo es muy fácil de entender y, a menudo, ofrece un rendimiento razonable sin muchos ajustes. El uso de este algoritmo es un buen método de referencia para probar antes de considerar técnicas más avanzadas. La creación del modelo de vecinos más cercanos suele ser muy rápida, pero cuando el conjunto de entrenamiento es muy grande (ya sea en número de características o en número de muestras), la predicción puede ser lenta. Al utilizar el algoritmo k-NN, es importante preprocesar los datos (consulte el Capítulo 3). Este enfoque a menudo no funciona bien en conjuntos de datos con muchas características (cientos o más) y funciona particularmente mal con conjuntos de datos donde la mayoría de las características son 0 la mayor parte del tiempo (los denominados conjuntos de datos dispersos).

So, while the nearest k-neighbors algorithm is easy to understand, it is not often used in practice, due to prediction being slow and its inability to handle many features. The method we discuss next has neither of these drawbacks.

Por lo tanto, si bien el algoritmo de los k vecinos más próximos es fácil de entender, no se suele utilizar en la práctica debido a que la predicción es lenta y no puede manejar muchas características. El método que analizaremos a continuación no tiene ninguno de estos inconvenientes.

**Linear Models**

Linear models are a class of models that are widely used in practice and have been studied extensively in the last few decades, with roots going back over a hundred years. Linear models make a prediction using a linear function of the input features, which we will explain shortly.

Los modelos lineales son una clase de modelos que se utilizan ampliamente en la práctica y se han estudiado en profundidad en las últimas décadas, con orígenes que se remontan a más de cien años. Los modelos lineales realizan una predicción utilizando una función lineal de las características de entrada, que explicaremos en breve.

**Linear models for regression**

For regression, the general prediction formula for a linear model looks as follows:

Para la regresión, la fórmula de predicción general para un modelo lineal es la siguiente:

ŷ = w[0] \* x[0] + w[1] \* x[1] + ... + w[p] \* x[p] + b

Here, x[0] to x[p] denotes the features (in this example, the number of features is p+1) of a single data point, w and b are parameters of the model that are learned, and ŷ is the prediction the model makes. For a dataset with a single feature, this is:

Aquí, x[0] a x[p] denotan las características (en este ejemplo, la cantidad de características es p+1) de un único punto de datos, w y b son parámetros del modelo que se aprenden, y ŷ es la predicción que hace el modelo. Para un conjunto de datos con una única característica, esto es:

ŷ = w[0] \* x[0] + b

which you might remember from high school mathematics as the equation for a line. Here, w[0] is the slope and b is the y-axis offset. For more features, w contains the slopes along each feature axis. Alternatively, you can think of the predicted response as being a weighted sum of the input features, with weights (which can be negative) given by the entries of w.

que quizás recuerdes de las matemáticas de la escuela secundaria como la ecuación de una línea. Aquí, w[0] es la pendiente y b es el desplazamiento del eje y. Para más características, w contiene las pendientes a lo largo de cada eje de características. Alternativamente, puedes pensar en la respuesta predicha como una suma ponderada de las características de entrada, con pesos (que pueden ser negativos) dados por las entradas de w.

Trying to learn the parameters w[0] and b on our one-dimensional wave dataset might lead to the following line (see Figure 2-11):

Intentar aprender los parámetros w[0] y b en nuestro conjunto de datos de ondas unidimensionales podría conducir a la siguiente línea (ver Figura 2-11):

In[24]:

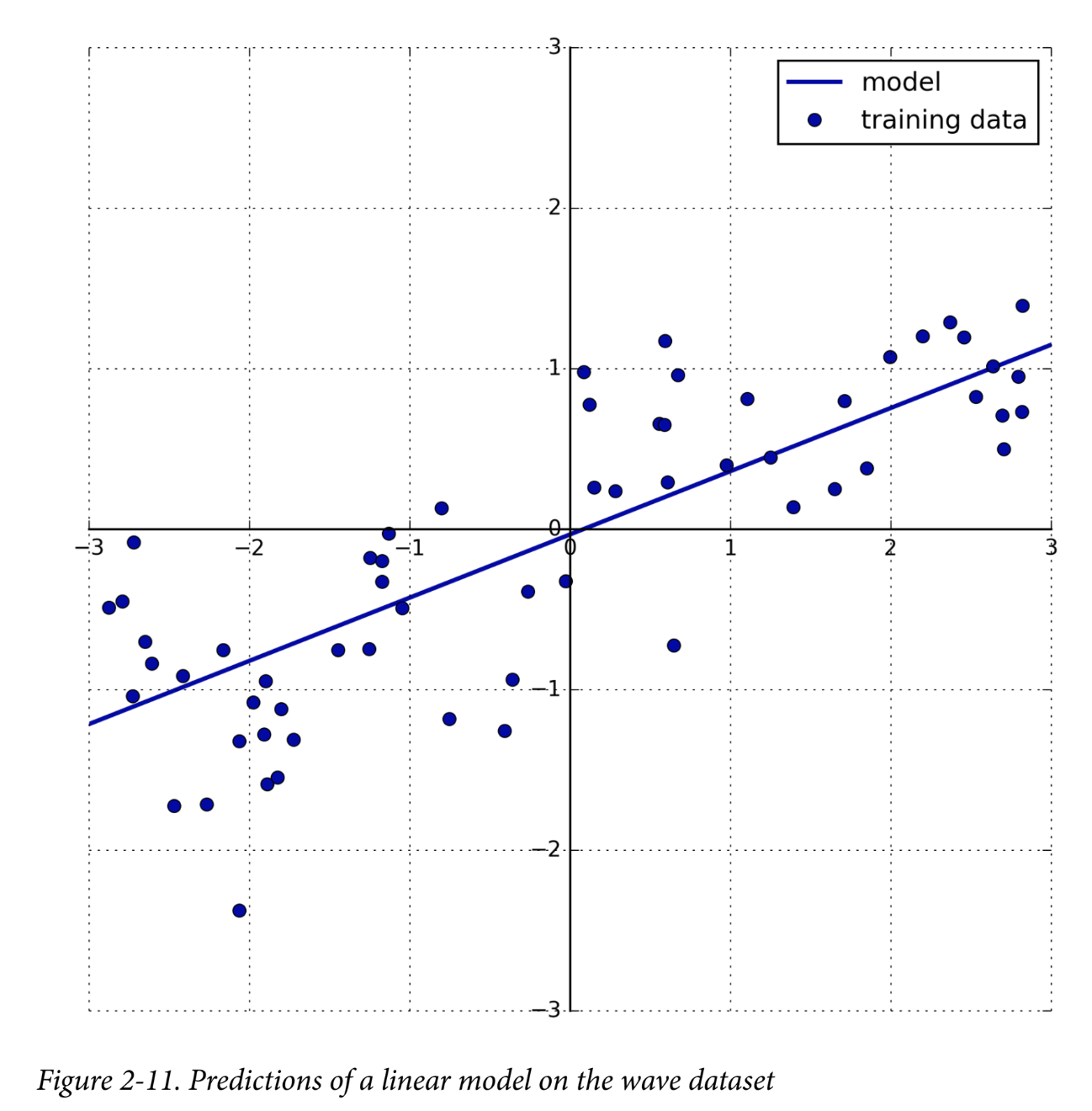
mglearn.plots.plot\_linear\_regression\_wave()

Out[24]:

w[0]: 0.393906

b: -0.031804

Supervised Machine Learning Algorithms



47Figure 2-11. Predictions of a linear model on the wave dataset

We added a coordinate cross into the plot to make it easier to understand the line. Looking at w[0] we see that the slope should be around 0.4, which we can confirm visually in the plot. The intercept is where the prediction line should cross the y-axis: this is slightly below zero, which you can also confirm in the image.

Agregamos una cruz de coordenadas al gráfico para que sea más fácil entender la línea. Al observar w[0], vemos que la pendiente debería estar alrededor de 0,4, lo que podemos confirmar visualmente en el gráfico. La intersección es donde la línea de predicción debería cruzar el eje y: esto está ligeramente por debajo de cero, lo que también se puede confirmar en la imagen.

Linear models for regression can be characterized as regression models for which the prediction is a line for a single feature, a plane when using two features, or a hyperplane in higher dimensions (that is, when using more features).

Los modelos lineales de regresión se pueden caracterizar como modelos de regresión para los cuales la predicción es una línea para una sola característica, un plano cuando se utilizan dos características o un hiperplano en dimensiones superiores (es decir, cuando se utilizan más características).

If you compare the predictions made by the straight line with those made by the KNeighborsRegressor in Figure 2-10, using a straight line to make predictions seems very restrictive. It looks like all the fine details of the data are lost. In a sense, this is true. It is a strong (and somewhat unrealistic) assumption that our target y is a linear combination of the features. But looking at one-dimensional data gives a somewhat skewed perspective. For datasets with many features, linear models can be very powerful. In particular, if you have more features than training data points, any target y can be perfectly modeled (on the training set) as a linear function.6

Si comparamos las predicciones realizadas con la línea recta con las realizadas con el KNeighborsRegressor en la Figura 2-10, el uso de una línea recta para realizar predicciones parece muy restrictivo. Parece que se pierden todos los detalles finos de los datos. En cierto sentido, esto es cierto. Es una suposición sólida (y algo irreal) que nuestro objetivo y sea una combinación lineal de las características. Pero observar datos unidimensionales ofrece una perspectiva algo sesgada. Para conjuntos de datos con muchas características, los modelos lineales pueden ser muy potentes. En particular, si tiene más características que puntos de datos de entrenamiento, cualquier objetivo y se puede modelar perfectamente (en el conjunto de entrenamiento) como una función lineal.6

There are many different linear models for regression. The difference between these models lies in how the model parameters w and b are learned from the training data, and how model complexity can be controlled. We will now take a look at the most popular linear models for regression.

Existen muchos modelos lineales diferentes para la regresión. La diferencia entre estos modelos radica en cómo se aprenden los parámetros w y b del modelo a partir de los datos de entrenamiento y cómo se puede controlar la complejidad del modelo. Ahora analizaremos los modelos lineales más populares para la regresión.

**Linear regression (aka ordinary least squares)**

Linear regression, or ordinary least squares (OLS), is the simplest and most classic linear method for regression. Linear regression finds the parameters w and b that minimize the mean squared error between predictions and the true regression targets, y, on the training set. The mean squared error is the sum of the squared differences between the predictions and the true values, divided by the number of samples. Linear regression has no parameters, which is a benefit, but it also has no way to control model complexity.

La regresión lineal, o mínimos cuadrados ordinarios (MCO), es el método lineal más simple y clásico para la regresión. La regresión lineal encuentra los parámetros w y b que minimizan el error cuadrático medio entre las predicciones y los objetivos de regresión reales, y, en el conjunto de entrenamiento. El error cuadrático medio es la suma de las diferencias al cuadrado entre las predicciones y los valores reales, dividida por el número de muestras. La regresión lineal no tiene parámetros, lo cual es una ventaja, pero tampoco tiene forma de controlar la complejidad del modelo.

Here is the code that produces the model you can see in Figure 2-11:

In[25]:

from sklearn.linear\_model import LinearRegression

X, y = mglearn.datasets.make\_wave(n\_samples=60)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=42)

lr = LinearRegression().fit(X\_train, y\_train)

The “slope” parameters (w), also called weights or coefficients, are stored in the coef attribute, while the offset or intercept (b) is stored in the intercept\_ attribute:

In[26]:

print("lr.coef\_: {}".format(lr.coef\_))

print("lr.intercept\_: {}".format(lr.intercept\_))

Out[26]:

lr.coef\_: [ 0.394]

lr.intercept\_: -0.031804343026759746

6 This is easy to see if you know some linear algebra.

Supervised Machine Learning Algorithms

|

You might notice the strange-looking trailing underscore at the end of coef\_ and intercept\_. scikit-learn always stores anything that is derived from the training data in attributes that end with a trailing underscore. That is to separate them from parameters that are set by the user.

Es posible que notes el extraño guión bajo final al final de coef\_ e intercept\_. scikit-learn siempre almacena todo lo que se deriva de los datos de entrenamiento en atributos que terminan con un guión bajo final. Esto es para separarlos de los parámetros que establece el usuario.

The intercept\_ attribute is always a single float number, while the coef\_ attribute is a NumPy array with one entry per input feature. As we only have a single input feature in the wave dataset, lr.coef\_ only has a single entry.

El atributo intercept\_ siempre es un único número de punto flotante, mientras que el atributo coef\_ es una matriz NumPy con una entrada por cada característica de entrada. Como solo tenemos una única característica de entrada en el conjunto de datos de ondas, lr.coef\_ solo tiene una única entrada.

Let’s look at the training set and test set performance:

Veamos el rendimiento del conjunto de entrenamiento y del conjunto de prueba:

In[27]:

print("Training set score: {:.2f}".format(lr.score(X\_train, y\_train)))

print("Test set score: {:.2f}".format(lr.score(X\_test, y\_test)))

Out[27]:

Training set score: 0.67

Test set score: 0.66

An R2 of around 0.66 is not very good, but we can see that the scores on the training and test sets are very close together. This means we are likely underfitting, not overfitting. For this one-dimensional dataset, there is little danger of overfitting, as the model is very simple (or restricted). However, with higher-dimensional datasets (meaning datasets with a large number of features), linear models become more powerful, and there is a higher chance of overfitting. Let’s take a look at how LinearRegression performs on a more complex dataset, like the Boston Housing dataset. Remember that this dataset has 506 samples and 105 derived features. First, we load the dataset and split it into a training and a test set. Then we build the linear regression model as before:

Un R2 de alrededor de 0,66 no es muy bueno, pero podemos ver que las puntuaciones en los conjuntos de entrenamiento y prueba están muy cerca una de la otra. Esto significa que es probable que estemos subajusteando, no sobreajusteando. Para este conjunto de datos unidimensional, hay poco peligro de sobreajuste, ya que el modelo es muy simple (o restringido). Sin embargo, con conjuntos de datos de dimensiones superiores (es decir, conjuntos de datos con una gran cantidad de características), los modelos lineales se vuelven más potentes y hay una mayor probabilidad de sobreajuste. Echemos un vistazo a cómo se desempeña LinearRegression en un conjunto de datos más complejo, como el conjunto de datos de Boston Housing. Recuerde que este conjunto de datos tiene 506 muestras y 105 características derivadas. Primero, cargamos el conjunto de datos y lo dividimos en un conjunto de entrenamiento y uno de prueba. Luego, construimos el modelo de regresión lineal como antes:

In[28]:

X, y = mglearn.datasets.load\_extended\_boston()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)

lr = LinearRegression().fit(X\_train, y\_train)

When comparing training set and test set scores, we find that we predict very accu‐

rately on the training set, but the R2 on the test set is much worse:

In[29]:

print("Training set score: {:.2f}".format(lr.score(X\_train, y\_train)))

print("Test set score: {:.2f}".format(lr.score(X\_test, y\_test)))

Out[29]:

Training set score: 0.95

Test set score: 0.61

This discrepancy between performance on the training set and the test set is a clear sign of overfitting, and therefore we should try to find a model that allows us to control complexity. One of the most commonly used alternatives to standard linear regression is ridge regression, which we will look into next.

Esta discrepancia entre el rendimiento en el conjunto de entrenamiento y el de prueba es una clara señal de sobreajuste, por lo que deberíamos intentar encontrar un modelo que nos permita controlar la complejidad. Una de las alternativas más utilizadas a la regresión lineal estándar es la regresión de cresta, que analizaremos a continuación.

**Ridge regression**

Ridge regression is also a linear model for regression, so the formula it uses to make predictions is the same one used for ordinary least squares. In ridge regression, though, the coefficients (w) are chosen not only so that they predict well on the training data, but also to fit an additional constraint. We also want the magnitude of coefficients to be as small as possible; in other words, all entries of w should be close to zero. Intuitively, this means each feature should have as little effect on the outcome as possible (which translates to having a small slope), while still predicting well. This constraint is an example of what is called regularization. Regularization means explicitly restricting a model to avoid overfitting. The particular kind used by ridge regression is known as L2 regularization.7

La regresión de cresta también es un modelo lineal de regresión, por lo que la fórmula que utiliza para hacer predicciones es la misma que se utiliza para los mínimos cuadrados ordinarios. Sin embargo, en la regresión de cresta, los coeficientes (w) se eligen no solo para que predigan bien sobre los datos de entrenamiento, sino también para que se ajusten a una restricción adicional. También queremos que la magnitud de los coeficientes sea lo más pequeña posible; en otras palabras, todas las entradas de w deben ser cercanas a cero. Intuitivamente, esto significa que cada característica debe tener el menor efecto posible en el resultado (lo que se traduce en tener una pendiente pequeña), sin dejar de predecir bien. Esta restricción es un ejemplo de lo que se llama regularización. Regularización significa restringir explícitamente un modelo para evitar el sobreajuste. El tipo particular utilizado por la regresión de cresta se conoce como regularización L2.7

Ridge regression is implemented in linear\_model.Ridge. Let’s see how well it does on the extended Boston Housing dataset:

La regresión de crestas se implementa en linear\_model.Ridge. Veamos qué tan bien funciona en el conjunto de datos ampliado de Boston Housing:

In[30]:

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("Test set score: {:.2f}".format(ridge.score(X\_test, y\_test)))

Out[30]:

Training set score: 0.89

Test set score: 0.75

As you can see, the training set score of Ridge is lower than for LinearRegression, while the test set score is higher. This is consistent with our expectation. With linear regression, we were overfitting our data. Ridge is a more restricted model, so we are less likely to overfit. A less complex model means worse performance on the training set, but better generalization. As we are only interested in generalization performance, we should choose the Ridge model over the LinearRegression model.

Como puede ver, la puntuación del conjunto de entrenamiento de Ridge es menor que la de la regresión lineal, mientras que la puntuación del conjunto de prueba es mayor. Esto es coherente con nuestra expectativa. Con la regresión lineal, estábamos sobreajustando nuestros datos. Ridge es un modelo más restringido, por lo que es menos probable que lo hagamos. Un modelo menos complejo significa un peor rendimiento en el conjunto de entrenamiento, pero una mejor generalización. Como solo nos interesa el rendimiento de la generalización, deberíamos elegir el modelo Ridge en lugar del modelo de regresión lineal.

7 Mathematically, Ridge penalizes the squared L2 norm of the coefficients, or the Euclidean length of w.

The Ridge model makes a trade-off between the simplicity of the model (near-zero coefficients) and its performance on the training set. How much importance the model places on simplicity versus training set performance can be specified by the user, using the alpha parameter. In the previous example, we used the default parameter alpha=1.0. There is no reason why this will give us the best trade-off, though. The optimum setting of alpha depends on the particular dataset we are using. Increasing alpha forces coefficients to move more toward zero, which decreases training set performance but might help generalization. For example:

El modelo Ridge establece un equilibrio entre la simplicidad del modelo (coeficientes cercanos a cero) y su rendimiento en el conjunto de entrenamiento. El usuario puede especificar cuánta importancia le da el modelo a la simplicidad en comparación con el rendimiento del conjunto de entrenamiento mediante el parámetro alfa. En el ejemplo anterior, usamos el parámetro predeterminado alfa=1.0. Sin embargo, no hay ninguna razón por la que esto nos brinde el mejor equilibrio. La configuración óptima de alfa depende del conjunto de datos en particular que estemos usando. Aumentar alfa obliga a los coeficientes a moverse más hacia cero, lo que disminuye el rendimiento del conjunto de entrenamiento pero puede ayudar a la generalización. Por ejemplo:

In[31]:

ridge10 = Ridge(alpha=10).fit(X\_train, y\_train)

print("Training set score: {:.2f}".format(ridge10.score(X\_train, y\_train)))

print("Test set score: {:.2f}".format(ridge10.score(X\_test, y\_test)))

Out[31]:

Training set score: 0.79

Test set score: 0.64

Decreasing alpha allows the coefficients to be less restricted, meaning we move right in Figure 2-1. For very small values of alpha, coefficients are barely restricted at all, and we end up with a model that resembles LinearRegression:

La disminución de alfa permite que los coeficientes estén menos restringidos, lo que significa que nos movemos hacia la derecha en la Figura 2-1. Para valores muy pequeños de alfa, los coeficientes apenas están restringidos y terminamos con un modelo que se parece a la regresión lineal:

In[32]:

ridge01 = Ridge(alpha=0.1).fit(X\_train, y\_train)

print("Training set score: {:.2f}".format(ridge01.score(X\_train, y\_train)))

print("Test set score: {:.2f}".format(ridge01.score(X\_test, y\_test)))

Out[32]:

Training set score: 0.93

Test set score: 0.77

Here, alpha=0.1 seems to be working well. We could try decreasing alpha even more to improve generalization. For now, notice how the parameter alpha corresponds to the model complexity as shown in Figure 2-1. We will discuss methods to properly select parameters in Chapter 5.

Aquí, alfa=0,1 parece funcionar bien. Podríamos intentar reducir alfa aún más para mejorar la generalización. Por ahora, observe cómo el parámetro alfa corresponde a la complejidad del modelo, como se muestra en la Figura 2-1. Analizaremos métodos para seleccionar parámetros correctamente en el Capítulo 5.

We can also get a more qualitative insight into how the alpha parameter changes the model by inspecting the coef\_ attribute of models with different values of alpha. A higher alpha means a more restricted model, so we expect the entries of coef\_ to have smaller magnitude for a high value of alpha than for a low value of alpha. This is confirmed in the plot in Figure 2-12:

También podemos obtener una perspectiva más cualitativa de cómo el parámetro alfa cambia el modelo inspeccionando el atributo coef\_ de los modelos con diferentes valores de alfa. Un alfa más alto significa un modelo más restringido, por lo que esperamos que las entradas de coef\_ tengan una magnitud menor para un valor alto de alfa que para un valor bajo de alfa. Esto se confirma en el gráfico de la Figura 2-12:

In[33]:

plt.plot(ridge.coef\_, 's', label="Ridge alpha=1")

plt.plot(ridge10.coef\_, '^', label="Ridge alpha=10")

plt.plot(ridge01.coef\_, 'v', label="Ridge alpha=0.1")

plt.plot(lr.coef\_, 'o', label="LinearRegression")

plt.xlabel("Coefficient index")

plt.ylabel("Coefficient magnitude")

plt.hlines(0, 0, len(lr.coef\_))

plt.ylim(-25, 25)

plt.legend()

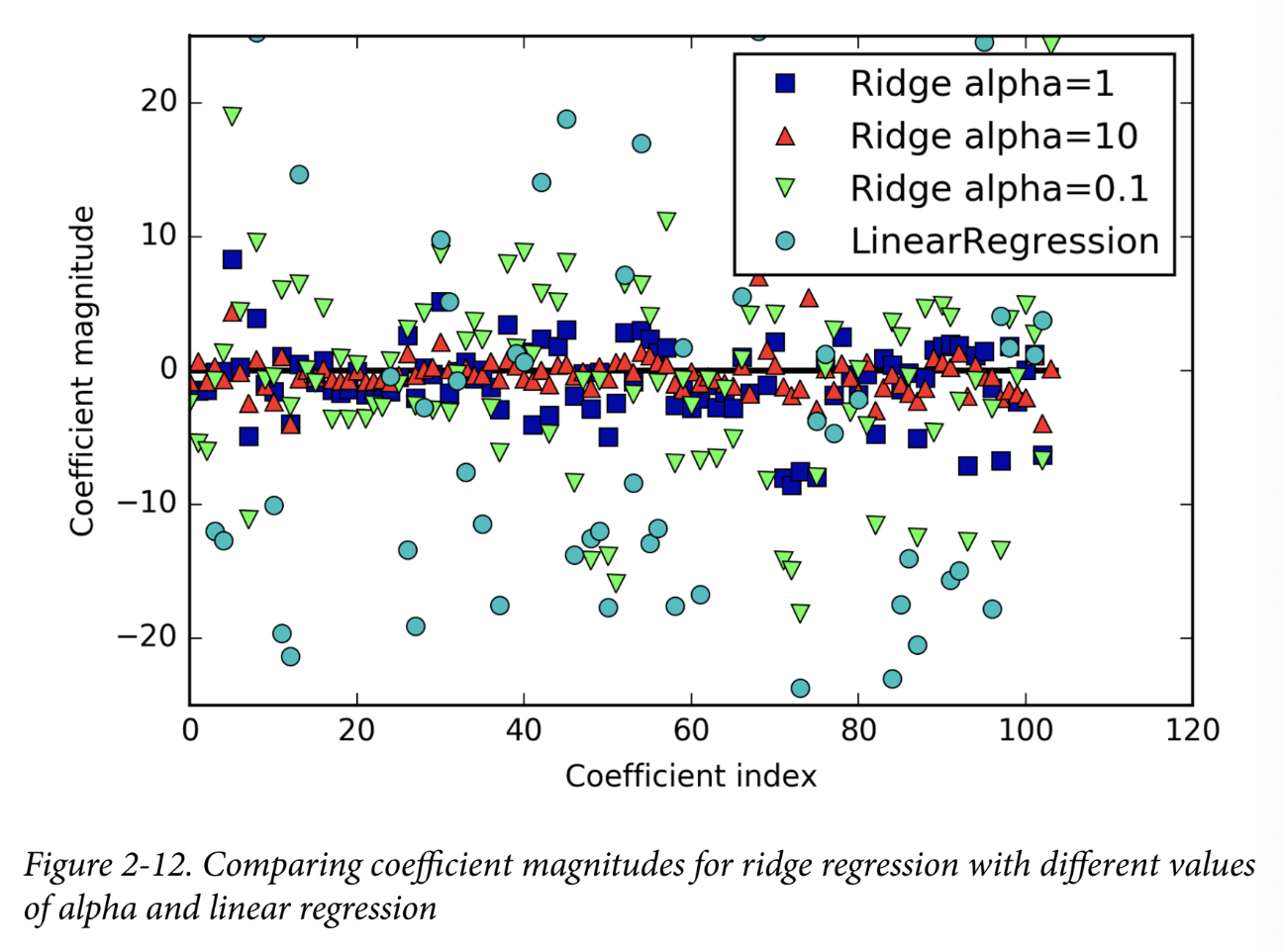


Figure 2-12. Comparing coefficient magnitudes for ridge regression with different values

of alpha and linear regression

Here, the x-axis enumerates the entries of coef\_: x=0 shows the coefficient associated with the first feature, x=1 the coefficient associated with the second feature, and so on up to x=100. The y-axis shows the numeric values of the corresponding values of the coefficients. The main takeaway here is that for alpha=10, the coefficients are mostly between around –3 and 3. The coefficients for the Ridge model with alpha=1 are somewhat larger. The dots corresponding to alpha=0.1 have larger magnitude still, and many of the dots corresponding to linear regression without any regularization (which would be alpha=0) are so large they are outside of the chart.

Aquí, el eje x enumera las entradas de coef\_: x=0 muestra el coeficiente asociado con la primera característica, x=1 el coeficiente asociado con la segunda característica, y así sucesivamente hasta x=100. El eje y muestra los valores numéricos de los valores correspondientes de los coeficientes. La principal conclusión aquí es que para alfa=10, los coeficientes están en su mayoría entre alrededor de -3 y 3. Los coeficientes para el modelo Ridge con alfa=1 son algo más grandes. Los puntos correspondientes a alfa=0,1 tienen una magnitud aún mayor, y muchos de los puntos correspondientes a la regresión lineal sin ninguna regularización (que sería alfa=0) son tan grandes que están fuera del gráfico.

Another way to understand the influence of regularization is to fix a value of alpha but vary the amount of training data available. For Figure 2-13, we subsampled the Boston Housing dataset and evaluated LinearRegression and Ridge(alpha=1) on subsets of increasing size (plots that show model performance as a function of dataset size are called learning curves):

Otra forma de entender la influencia de la regularización es fijar un valor de alfa pero variar la cantidad de datos de entrenamiento disponibles. Para la Figura 2-13, tomamos una submuestra del conjunto de datos de Boston Housing y evaluamos LinearRegression y Ridge(alpha=1) en subconjuntos de tamaño creciente (los gráficos que muestran el rendimiento del modelo en función del tamaño del conjunto de datos se denominan curvas de aprendizaje):

In[34]:

mglearn.plots.plot\_ridge\_n\_samples()

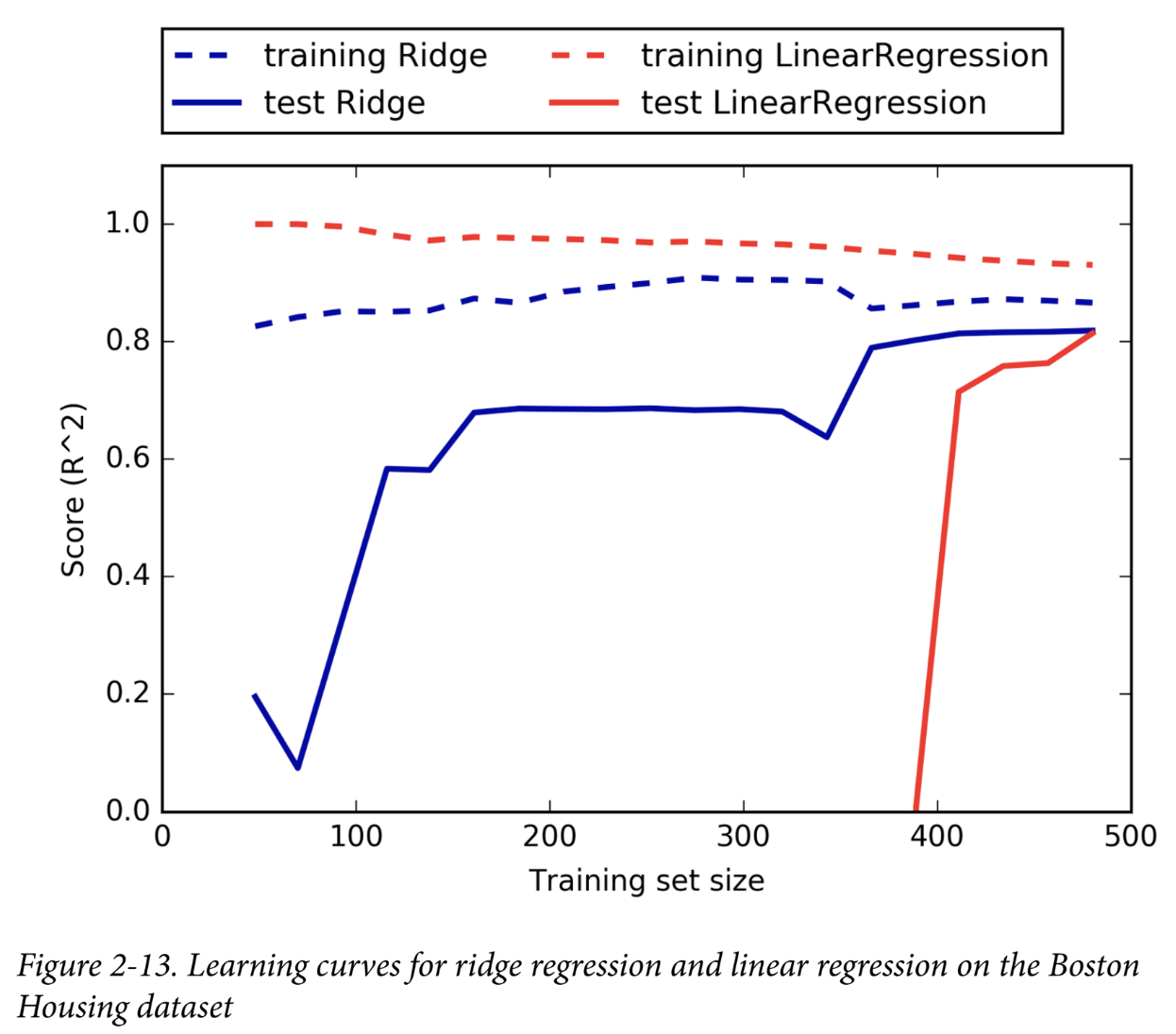


Figure 2-13. Learning curves for ridge regression and linear regression on the Boston

Housing dataset

As one would expect, the training score is higher than the test score for all dataset sizes, for both ridge and linear regression. Because ridge is regularized, the training score of ridge is lower than the training score for linear regression across the board. However, the test score for ridge is better, particularly for small subsets of the data. For less than 400 data points, linear regression is not able to learn anything. As more and more data becomes available to the model, both models improve, and linear regression catches up with ridge in the end. The lesson here is that with enough training data, regularization becomes less important, and given enough data, ridge and linear regression will have the same performance (the fact that this happens here when using the full dataset is just by chance). Another interesting aspect of Figure 2-13 is the decrease in training performance for linear regression. If more data is added, it becomes harder for a model to overfit, or memorize the data.

Como era de esperar, la puntuación de entrenamiento es mayor que la puntuación de prueba para todos los tamaños de conjuntos de datos, tanto para la regresión lineal como para la regresión ridge. Debido a que la regresión ridge está regularizada, la puntuación de entrenamiento de la regresión ridge es menor que la puntuación de entrenamiento para la regresión lineal en general. Sin embargo, la puntuación de prueba para la regresión ridge es mejor, en particular para pequeños subconjuntos de los datos. Para menos de 400 puntos de datos, la regresión lineal no puede aprender nada. A medida que más y más datos están disponibles para el modelo, ambos modelos mejoran y la regresión lineal alcanza a la regresión ridge al final. La lección aquí es que con suficientes datos de entrenamiento, la regularización se vuelve menos importante y, dados suficientes datos, la regresión ridge y la regresión lineal tendrán el mismo rendimiento (el hecho de que esto suceda aquí cuando se utiliza el conjunto de datos completo es solo por casualidad). Otro aspecto interesante de la Figura 2-13 es la disminución del rendimiento de entrenamiento para la regresión lineal. Si se agregan más datos, se vuelve más difícil para un modelo sobreajustar o memorizar los datos.

**Lasso**

An alternative to Ridge for regularizing linear regression is Lasso. As with ridge regression, using the lasso also restricts coefficients to be close to zero, but in a slightly different way, called L1 regularization.8 The consequence of L1 regularization is that when using the lasso, some coefficients are exactly zero. This means some features are entirely ignored by the model. This can be seen as a form of automatic feature selection. Having some coefficients be exactly zero often makes a model easier to interpret, and can reveal the most important features of your model.

Una alternativa a Ridge para regularizar la regresión lineal es Lasso. Al igual que con la regresión Ridge, el uso de Lasso también restringe los coeficientes para que sean cercanos a cero, pero de una manera ligeramente diferente, llamada regularización L1.8 La consecuencia de la regularización L1 es que cuando se usa Lasso, algunos coeficientes son exactamente cero. Esto significa que el modelo ignora por completo algunas características. Esto puede verse como una forma de selección automática de características. Tener algunos coeficientes exactamente cero a menudo hace que un modelo sea más fácil de interpretar y puede revelar las características más importantes de su modelo.

Let’s apply the lasso to the extended Boston Housing dataset:

Apliquemos el lazo al conjunto de datos ampliado de Boston Housing:

In[35]:

from sklearn.linear\_model import Lasso

lasso = Lasso().fit(X\_train, y\_train)

print("Training set score: {:.2f}".format(lasso.score(X\_train, y\_train)))

print("Test set score: {:.2f}".format(lasso.score(X\_test, y\_test)))

print("Number of features used: {}".format(np.sum(lasso.coef\_ != 0)))

Out[35]:

Training set score: 0.29

Test set score: 0.21

Number of features used: 4

As you can see, Lasso does quite badly, both on the training and the test set. This indicates that we are underfitting, and we find that it used only 4 of the 105 features. Similarly to Ridge, the Lasso also has a regularization parameter, alpha, that controls how strongly coefficients are pushed toward zero. In the previous example, we used the default of alpha=1.0. To reduce underfitting, let’s try decreasing alpha. When we do this, we also need to increase the default setting of max\_iter (the maximum number of iterations to run):

Como puede ver, Lasso funciona bastante mal, tanto en el conjunto de entrenamiento como en el de prueba. Esto indica que estamos subadaptando y descubrimos que solo utilizó 4 de las 105 características. De manera similar a Ridge, Lasso también tiene un parámetro de regularización, alpha, que controla la fuerza con la que los coeficientes se acercan a cero. En el ejemplo anterior, usamos el valor predeterminado de alpha=1.0. Para reducir el subajuste, intentemos disminuir alpha. Cuando hagamos esto, también debemos aumentar la configuración predeterminada de max\_iter (la cantidad máxima de iteraciones a ejecutar):

8 The lasso penalizes the L1 norm of the coefficient vector—or in other words, the sum of the absolute values of the coefficients.

In[36]:

# we increase the default setting of "max\_iter",

# otherwise the model would warn us that we should increase max\_iter.

lasso001 = Lasso(alpha=0.01, max\_iter=100000).fit(X\_train, y\_train)

print("Training set score: {:.2f}".format(lasso001.score(X\_train, y\_train)))

print("Test set score: {:.2f}".format(lasso001.score(X\_test, y\_test)))

print("Number of features used: {}".format(np.sum(lasso001.coef\_ != 0)))

Out[36]:

Training set score: 0.90

Test set score: 0.77

Number of features used: 33

A lower alpha allowed us to fit a more complex model, which worked better on the training and test data. The performance is slightly better than using Ridge, and we are using only 33 of the 105 features. This makes this model potentially easier to understand.

Un alfa más bajo nos permitió ajustar un modelo más complejo, que funcionó mejor en los datos de entrenamiento y prueba. El rendimiento es ligeramente mejor que con Ridge, y estamos utilizando solo 33 de las 105 características. Esto hace que este modelo sea potencialmente más fácil de entender.

If we set alpha too low, however, we again remove the effect of regularization and end up overfitting, with a result similar to LinearRegression:

Sin embargo, si establecemos un alfa demasiado bajo, eliminamos nuevamente el efecto de la regularización y terminamos sobreajustando, con un resultado similar a la regresión lineal:

In[37]:

lasso00001 = Lasso(alpha=0.0001, max\_iter=100000).fit(X\_train, y\_train)

print("Training set score: {:.2f}".format(lasso00001.score(X\_train, y\_train)))

print("Test set score: {:.2f}".format(lasso00001.score(X\_test, y\_test)))

print("Number of features used: {}".format(np.sum(lasso00001.coef\_ != 0)))

Out[37]:

Training set score: 0.95

Test set score: 0.64

Number of features used: 94

Again, we can plot the coefficients of the different models, similarly to Figure 2-12. The result is shown in Figure 2-14:

In[38]:

plt.plot(lasso.coef\_, 's', label="Lasso alpha=1")

plt.plot(lasso001.coef\_, '^', label="Lasso alpha=0.01")

plt.plot(lasso00001.coef\_, 'v', label="Lasso alpha=0.0001")

plt.plot(ridge01.coef\_, 'o', label="Ridge alpha=0.1")

plt.legend(ncol=2, loc=(0, 1.05))

plt.ylim(-25, 25)

plt.xlabel("Coefficient index")

plt.ylabel("Coefficient magnitude")

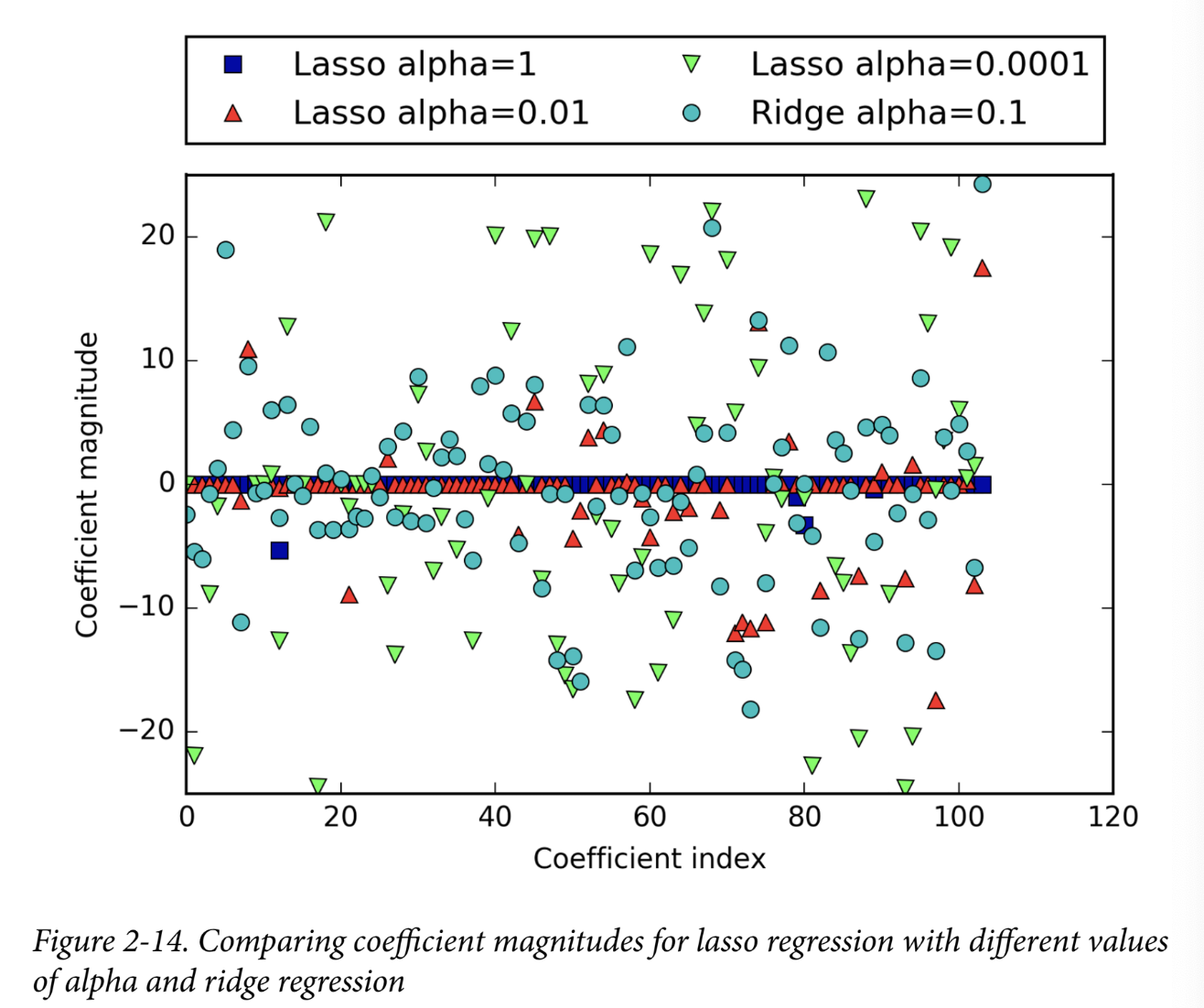


Figure 2-14. Comparing coefficient magnitudes for lasso regression with different values

of alpha and ridge regression

For alpha=1, we not only see that most of the coefficients are zero (which we already knew), but that the remaining coefficients are also small in magnitude. Decreasing alpha to 0.01, we obtain the solution shown as an upward pointing triangle, which causes most features to be exactly zero. Using alpha=0.0001, we get a model that is quite unregularized, with most coefficients nonzero and of large magnitude. For comparison, the best Ridge solution is shown as circles. The Ridge model with alpha=0.1 has similar predictive performance as the lasso model with alpha=0.01, but using Ridge, all coefficients are nonzero.

Para alfa=1, no solo vemos que la mayoría de los coeficientes son cero (lo cual ya sabíamos), sino que los coeficientes restantes también son de pequeña magnitud. Al disminuir alfa a 0,01, obtenemos la solución que se muestra como un triángulo que apunta hacia arriba, lo que hace que la mayoría de las características sean exactamente cero. Usando alfa=0,0001, obtenemos un modelo que está bastante desregularizado, con la mayoría de los coeficientes distintos de cero y de gran magnitud. A modo de comparación, la mejor solución de Ridge se muestra como círculos. El modelo Ridge con alfa=0,1 tiene un rendimiento predictivo similar al del modelo Lasso con alfa=0,01, pero usando Ridge, todos los coeficientes son distintos de cero.

In practice, ridge regression is usually the first choice between these two models. However, if you have a large amount of features and expect only a few of them to be important, Lasso might be a better choice. Similarly, if you would like to have a model that is easy to interpret, Lasso will provide a model that is easier to understand, as it will select only a subset of the input features. scikit-learn also provides the ElasticNet class, which combines the penalties of Lasso and Ridge. In practice, this combination works best, though at the price of having two parameters to adjust: one for the L1 regularization, and one for the L2 regularization.

En la práctica, la regresión de cresta suele ser la primera opción entre estos dos modelos. Sin embargo, si tiene una gran cantidad de características y espera que solo algunas de ellas sean importantes, Lasso puede ser una mejor opción. De manera similar, si desea tener un modelo que sea fácil de interpretar, Lasso le proporcionará un modelo que sea más fácil de entender, ya que seleccionará solo un subconjunto de las características de entrada. scikit-learn también proporciona la clase ElasticNet, que combina las penalizaciones de Lasso y Ridge. En la práctica, esta combinación funciona mejor, aunque al precio de tener que ajustar dos parámetros: uno para la regularización L1 y otro para la regularización L2.

**Linear models for classification**

Linear models are also extensively used for classification. Let’s look at binary classification first. In this case, a prediction is made using the following formula:

Los modelos lineales también se utilizan ampliamente para la clasificación. Veamos primero la clasificación binaria. En este caso, se realiza una predicción utilizando la siguiente fórmula:

ŷ = w[0] \* x[0] + w[1] \* x[1] + ... + w[p] \* x[p] + b > 0

The formula looks very similar to the one for linear regression, but instead of just returning the weighted sum of the features, we threshold the predicted value at zero. If the function is smaller than zero, we predict the class –1; if it is larger than zero, we predict the class +1. This prediction rule is common to all linear models for classification. Again, there are many different ways to find the coefficients (w) and the intercept (b).

La fórmula es muy similar a la de la regresión lineal, pero en lugar de devolver simplemente la suma ponderada de las características, establecemos el valor predicho en cero. Si la función es menor que cero, predecimos la clase -1; si es mayor que cero, predecimos la clase +1. Esta regla de predicción es común a todos los modelos lineales de clasificación. Nuevamente, hay muchas formas diferentes de encontrar los coeficientes (w) y la intersección (b).

For linear models for regression, the output, ŷ, is a linear function of the features: a line, plane, or hyperplane (in higher dimensions). For linear models for classification, the decision boundary is a linear function of the input. In other words, a (binary) linear classifier is a classifier that separates two classes using a line, a plane, or a hyperplane. We will see examples of that in this section. There are many algorithms for learning linear models. These algorithms all differ in the following two ways:

En el caso de los modelos lineales de regresión, la salida, ŷ, es una función lineal de las características: una línea, un plano o un hiperplano (en dimensiones superiores). En el caso de los modelos lineales de clasificación, el límite de decisión es una función lineal de la entrada. En otras palabras, un clasificador lineal (binario) es un clasificador que separa dos clases mediante una línea, un plano o un hiperplano. Veremos ejemplos de ello en esta sección. Existen muchos algoritmos para aprender modelos lineales. Todos estos algoritmos difieren en las dos formas siguientes:

• The way in which they measure how well a particular combination of coefficients and intercept fits the training data

• If and what kind of regularization they use

Different algorithms choose different ways to measure what “fitting the training set well” means. For technical mathematical reasons, it is not possible to adjust w and b to minimize the number of misclassifications the algorithms produce, as one might hope. For our purposes, and many applications, the different choices for item 1 in the preceding list (called loss functions) are of little significance.

Los distintos algoritmos eligen distintas formas de medir lo que significa “ajustarse bien al conjunto de entrenamiento”. Por razones matemáticas técnicas, no es posible ajustar w y b para minimizar la cantidad de clasificaciones erróneas que producen los algoritmos, como sería de esperar. Para nuestros propósitos, y para muchas aplicaciones, las distintas opciones para el elemento 1 de la lista anterior (denominadas funciones de pérdida) tienen poca importancia.

The two most common linear classification algorithms are logistic regression, implemented in linear\_model.LogisticRegression, and linear support vector machines (linear SVMs), implemented in svm.LinearSVC (SVC stands for support vector classifier). Despite its name, LogisticRegression is a classification algorithm and not a regression algorithm, and it should not be confused with LinearRegression. We can apply the LogisticRegression and LinearSVC models to the forge dataset, and visualize the decision boundary as found by the linear models (Figure 2-15):

Los dos algoritmos de clasificación lineal más comunes son la regresión logística, implementada en linear\_model.LogisticRegression, y las máquinas de vectores de soporte lineales (SVM lineales), implementadas en svm.LinearSVC (SVC significa clasificador de vectores de soporte). A pesar de su nombre, LogisticRegression es un algoritmo de clasificación y no un algoritmo de regresión, y no debe confundirse con LinearRegression. Podemos aplicar los modelos LogisticRegression y LinearSVC al conjunto de datos de Forge y visualizar el límite de decisión que encuentran los modelos lineales (Figura 2-15):

[39]:

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make\_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(), LogisticRegression()], axes):

clf = model.fit(X, y)

mglearn.plots.plot\_2d\_separator(clf, X, fill=False, eps=0.5,

ax=ax, alpha=.7)

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y, ax=ax)

ax.set\_title("{}".format(clf.\_\_class\_\_.\_\_name\_\_))

ax.set\_xlabel("Feature 0")

ax.set\_ylabel("Feature 1")

axes[0].legend()

Figure 2-15. Decision boundaries of a linear SVM and logistic regression on the forge dataset with the default parameters

In this figure, we have the first feature of the forge dataset on the x-axis and the second feature on the y-axis, as before. We display the decision boundaries found by LinearSVC and LogisticRegression respectively as straight lines, separating the area classified as class 1 on the top from the area classified as class 0 on the bottom. In other words, any new data point that lies above the black line will be classified as class 1 by the respective classifier, while any point that lies below the black line will be classified as class 0.

En esta figura, tenemos la primera característica del conjunto de datos de forja en el eje x y la segunda característica en el eje y, como antes. Mostramos los límites de decisión encontrados por LinearSVC y LogisticRegression respectivamente como líneas rectas, separando el área clasificada como clase 1 en la parte superior del área clasificada como clase 0 en la parte inferior. En otras palabras, cualquier punto de datos nuevo que se encuentre por encima de la línea negra será clasificado como clase 1 por el clasificador respectivo, mientras que cualquier punto que se encuentre por debajo de la línea negra será clasificado como clase 0.

The two models come up with similar decision boundaries. Note that both misclassify two of the points. By default, both models apply an L2 regularization, in the same way that Ridge does for regression.

Los dos modelos presentan límites de decisión similares. Nótese que ambos clasifican incorrectamente dos de los puntos. De manera predeterminada, ambos modelos aplican una regularización L2, de la misma manera que Ridge lo hace para la regresión.

For LogisticRegression and LinearSVC the trade-off parameter that determines the strength of the regularization is called C, and higher values of C correspond to less regularization. In other words, when you use a high value for the parameter C, LogisticRegression and LinearSVC try to fit the training set as best as possible, while with low values of the parameter C, the models put more emphasis on finding a coefficient vector (w) that is close to zero.

En el caso de LogisticRegression y LinearSVC, el parámetro de compensación que determina la fuerza de la regularización se denomina C, y los valores más altos de C corresponden a una menor regularización. En otras palabras, cuando se utiliza un valor alto para el parámetro C, LogisticRegression y LinearSVC intentan ajustar el conjunto de entrenamiento lo mejor posible, mientras que con valores bajos del parámetro C, los modelos ponen más énfasis en encontrar un vector de coeficientes (w) cercano a cero.

There is another interesting aspect of how the parameter C acts. Using low values of C will cause the algorithms to try to adjust to the “majority” of data points, while using a higher value of C stresses the importance that each individual data point be classified correctly. Here is an illustration using LinearSVC (Figure 2-16):

Hay otro aspecto interesante de cómo actúa el parámetro C. El uso de valores bajos de C hará que los algoritmos intenten ajustarse a la “mayoría” de los puntos de datos, mientras que el uso de un valor más alto de C enfatiza la importancia de que cada punto de datos individual se clasifique correctamente. Aquí hay una ilustración utilizando LinearSVC (Figura 2-16):

In[40]:

mglearn.plots.plot\_linear\_svc\_regularization()

Figure 2-16. Decision boundaries of a linear SVM on the forge dataset for different values of C

On the lefthand side, we have a very small C corresponding to a lot of regularization. Most of the points in class 0 are at the bottom, and most of the points in class 1 are at the top. The strongly regularized model chooses a relatively horizontal line, misclassifying two points. In the center plot, C is slightly higher, and the model focuses more on the two misclassified samples, tilting the decision boundary. Finally, on the righthand side, the very high value of C in the model tilts the decision boundary a lot, now correctly classifying all points in class 0. One of the points in class 1 is still misclassified, as it is not possible to correctly classify all points in this dataset using a straight line. The model illustrated on the righthand side tries hard to correctly classify all points, but might not capture the overall layout of the classes well. In other words, this model is likely overfitting.

En el lado izquierdo, tenemos un valor C muy pequeño que corresponde a una gran regularización. La mayoría de los puntos de la clase 0 están en la parte inferior, y la mayoría de los puntos de la clase 1 están en la parte superior. El modelo fuertemente regularizado elige una línea relativamente horizontal, lo que clasifica incorrectamente dos puntos. En el gráfico central, C es ligeramente más alto y el modelo se centra más en las dos muestras mal clasificadas, lo que inclina el límite de decisión. Finalmente, en el lado derecho, el valor muy alto de C en el modelo inclina mucho el límite de decisión, lo que ahora clasifica correctamente todos los puntos de la clase 0. Uno de los puntos de la clase 1 sigue estando mal clasificado, ya que no es posible clasificar correctamente todos los puntos de este conjunto de datos utilizando una línea recta. El modelo ilustrado en el lado derecho intenta clasificar correctamente todos los puntos, pero es posible que no capture bien la disposición general de las clases. En otras palabras, es probable que este modelo esté sobreajustado.

Similarly to the case of regression, linear models for classification might seem very restrictive in low-dimensional spaces, only allowing for decision boundaries that are straight lines or planes. Again, in high dimensions, linear models for classification become very powerful, and guarding against overfitting becomes increasingly important when considering more features.

De manera similar al caso de la regresión, los modelos lineales para la clasificación pueden parecer muy restrictivos en espacios de baja dimensión, ya que solo permiten límites de decisión que son líneas rectas o planos. Nuevamente, en dimensiones altas, los modelos lineales para la clasificación se vuelven muy poderosos y la protección contra el sobreajuste se vuelve cada vez más importante cuando se consideran más características.

Let’s analyze LogisticRegression in more detail on the Breast Cancer dataset:

In[41]:

from sklearn.datasets import load\_breast\_cancer

cancer = load\_breast\_cancer()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

cancer.data, cancer.target, stratify=cancer.target, random\_state=42)

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)))

Out[41]:

Training set score: 0.953

Test set score: 0.958

The default value of C=1 provides quite good performance, with 95% accuracy on both the training and the test set. But as training and test set performance are very close, it is likely that we are underfitting. Let’s try to increase C to fit a more flexible model:

El valor predeterminado de C=1 ofrece un rendimiento bastante bueno, con un 95 % de precisión tanto en el conjunto de entrenamiento como en el de prueba. Pero como el rendimiento del conjunto de entrenamiento y el de prueba son muy similares, es probable que no estemos ajustando bien el modelo. Intentemos aumentar C para ajustarlo a un modelo más flexible:

In[42]:

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)))

Out[42]:

Training set score: 0.972

Test set score: 0.965

Using C=100 results in higher training set accuracy, and also a slightly increased test set accuracy, confirming our intuition that a more complex model should perform better.

El uso de C=100 da como resultado una mayor precisión del conjunto de entrenamiento y también una precisión ligeramente mayor del conjunto de prueba, lo que confirma nuestra intuición de que un modelo más complejo debería funcionar mejor.

We can also investigate what happens if we use an even more regularized model than the default of C=1, by setting C=0.01:

También podemos investigar qué sucede si utilizamos un modelo aún más regularizado que el predeterminado de C=1, estableciendo C=0,01:

In[43]:

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)))

Out[43]:

Training set score: 0.934

Test set score: 0.930

As expected, when moving more to the left along the scale shown in Figure 2-1 from an already underfit model, both training and test set accuracy decrease relative to the default parameters.

Como era de esperar, al moverse más hacia la izquierda a lo largo de la escala que se muestra en la Figura 2-1 desde un modelo ya subadaptado, la precisión del conjunto de entrenamiento y de prueba disminuye en relación con los parámetros predeterminados.

Finally, let’s look at the coefficients learned by the models with the three different settings of the regularization parameter C (Figure 2-17):

Finalmente, veamos los coeficientes aprendidos por los modelos con las tres configuraciones diferentes del parámetro de regularización C (Figura 2-17):

In[44]:

plt.plot(logreg.coef\_.T, 'o', label="C=1")

plt.plot(logreg100.coef\_.T, '^', label="C=100")

plt.plot(logreg001.coef\_.T, 'v', label="C=0.001")

plt.xticks(range(cancer.data.shape[1]), cancer.feature\_names, rotation=90)

plt.hlines(0, 0, cancer.data.shape[1])

plt.ylim(-5, 5)

plt.xlabel("Feature")

plt.ylabel("Coefficient magnitude")

plt.legend()

As LogisticRegression applies an L2 regularization by default, the result looks similar to that produced by Ridge in Figure 2-12. Stronger regularization pushes coefficients more and more toward zero, though coefficients never become exactly zero. Inspecting the plot more closely, we can also see an interesting effect in the third coefficient, for “mean perimeter.” For C=100 and C=1, the coefficient is negative, while for C=0.001, the coefficient is positive, with a magnitude that is even larger than for C=1. Interpreting a model like this, one might think the coefficient tells us which class a feature is associated with. For example, one might think that a high “texture error” feature is related to a sample being “malignant.” However, the change of sign in the coefficient for “mean perimeter” means that depending on which model we look at, a high “mean perimeter” could be taken as being either indicative of “benign” or indicative of “malignant.” This illustrates that interpretations of coefficients of linear models should always be taken with a grain of salt.

Como LogisticRegression aplica una regularización L2 por defecto, el resultado parece similar al producido por Ridge en la Figura 2-12. Una regularización más fuerte empuja los coeficientes cada vez más hacia cero, aunque los coeficientes nunca llegan a ser exactamente cero. Inspeccionando el gráfico más de cerca, también podemos ver un efecto interesante en el tercer coeficiente, para el “perímetro medio”. Para C=100 y C=1, el coeficiente es negativo, mientras que para C=0,001, el coeficiente es positivo, con una magnitud que es incluso mayor que para C=1. Al interpretar un modelo como este, uno podría pensar que el coeficiente nos dice con qué clase está asociada una característica. Por ejemplo, uno podría pensar que una característica de “error de textura” alto está relacionada con que una muestra sea “maligna”. Sin embargo, el cambio de signo en el coeficiente para el “perímetro medio” significa que, dependiendo del modelo que miremos, un “perímetro medio” alto podría tomarse como indicativo de “benigno” o indicativo de “maligno”. Esto ilustra que las interpretaciones de los coeficientes de los modelos lineales siempre deben tomarse con cautela.

Figure 2-17. Coefficients learned by logistic regression on the Breast Cancer dataset for different values of C

If we desire a more interpretable model, using L1 regularization might help, as it limits the model to using only a few features. Here is the coefficient plot and classification accuracies for L1 regularization (Figure 2-18):

Si deseamos un modelo más interpretable, puede resultar útil utilizar la regularización L1, ya que limita el modelo a utilizar solo unas pocas características. Aquí se muestra el gráfico de coeficientes y las precisiones de clasificación para la regularización L1 (Figura 2-18):

In[45]:

for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):

lr\_l1 = LogisticRegression(C=C, penalty="l1").fit(X\_train, y\_train)

print("Training accuracy of l1 logreg with C={:.3f}: {:.2f}".format(

C, lr\_l1.score(X\_train, y\_train)))

print("Test accuracy of l1 logreg with C={:.3f}: {:.2f}".format(

C, lr\_l1.score(X\_test, y\_test)))

plt.plot(lr\_l1.coef\_.T, marker, label="C={:.3f}".format(C))

plt.xticks(range(cancer.data.shape[1]), cancer.feature\_names, rotation=90)

plt.hlines(0, 0, cancer.data.shape[1])

plt.xlabel("Feature")

plt.ylabel("Coefficient magnitude")

plt.ylim(-5, 5)

plt.legend(loc=3)

Out[45]:

Training accuracy of l1 logreg with C=0.001: 0.91

Test accuracy of l1 logreg with C=0.001: 0.92

Training accuracy of l1 logreg with C=1.000: 0.96

Test accuracy of l1 logreg with C=1.000: 0.96

Training accuracy of l1 logreg with C=100.000: 0.99

Test accuracy of l1 logreg with C=100.000: 0.98

As you can see, there are many parallels between linear models for binary classification and linear models for regression. As in regression, the main difference between the models is the penalty parameter, which influences the regularization and whether the model will use all available features or select only a subset.

Como puede ver, existen muchos paralelismos entre los modelos lineales para la clasificación binaria y los modelos lineales para la regresión. Al igual que en la regresión, la principal diferencia entre los modelos es el parámetro de penalización, que influye en la regularización y en si el modelo utilizará todas las características disponibles o seleccionará solo un subconjunto.

Figure 2-18. Coefficients learned by logistic regression with L1 penalty on the Breast Cancer dataset for different values of C

**Linear models for multiclass classification**

Many linear classification models are for binary classification only, and don’t extend naturally to the multiclass case (with the exception of logistic regression). A common technique to extend a binary classification algorithm to a multiclass classification algorithm is the one-vs.-rest approach. In the one-vs.-rest approach, a binary model is learned for each class that tries to separate that class from all of the other classes, resulting in as many binary models as there are classes. To make a prediction, all binary classifiers are run on a test point. The classifier that has the highest score on its single class “wins,” and this class label is returned as the prediction.

Muchos modelos de clasificación lineal son solo para clasificación binaria y no se extienden naturalmente al caso de múltiples clases (con la excepción de la regresión logística). Una técnica común para extender un algoritmo de clasificación binaria a un algoritmo de clasificación multiclase es el enfoque de uno contra el resto. En el enfoque de uno contra el resto, se aprende un modelo binario para cada clase que intenta separar esa clase de todas las demás clases, lo que da como resultado tantos modelos binarios como clases haya. Para hacer una predicción, todos los clasificadores binarios se ejecutan en un punto de prueba. El clasificador que tiene la puntuación más alta en su clase única "gana", y esta etiqueta de clase se devuelve como la predicción.

Having one binary classifier per class results in having one vector of coefficients (w) and one intercept (b) for each class. The class for which the result of the classification confidence formula given here is highest is the assigned class label:

Tener un clasificador binario por clase da como resultado un vector de coeficientes (w) y una intersección (b) para cada clase. La clase para la que el resultado de la fórmula de confianza de clasificación que se proporciona aquí es más alto es la etiqueta de clase asignada:

w[0] \* x[0] + w[1] \* x[1] + ... + w[p] \* x[p] + b

The mathematics behind multiclass logistic regression differ somewhat from the one- vs.-rest approach, but they also result in one coefficient vector and intercept per class, and the same method of making a prediction is applied.

Las matemáticas detrás de la regresión logística multiclase difieren un poco del enfoque de uno contra el resto, pero también dan como resultado un vector de coeficientes e intersección por clase, y se aplica el mismo método para hacer una predicción.

Let’s apply the one-vs.-rest method to a simple three-class classification dataset. We use a two-dimensional dataset, where each class is given by data sampled from a Gaussian distribution (see Figure 2-19):

Apliquemos el método de uno contra el resto a un conjunto de datos de clasificación de tres clases simple. Usamos un conjunto de datos bidimensional, donde cada clase está dada por datos muestreados de una distribución gaussiana (ver Figura 2-19):

In[46]:

from sklearn.datasets import make\_blobs

X, y = make\_blobs(random\_state=42)

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

plt.legend(["Class 0", "Class 1", "Class 2"])

Figure 2-19. Two-dimensional toy dataset containing three classes

66

|

Chapter 2: Supervised LearningNow, we train a LinearSVC classifier on the dataset:

In[47]:

linear\_svm = LinearSVC().fit(X, y)

print("Coefficient shape: ", linear\_svm.coef\_.shape)

print("Intercept shape: ", linear\_svm.intercept\_.shape)

Out[47]:

Coefficient shape: (3, 2)

Intercept shape: (3,)

We see that the shape of the coef\_ is (3, 2), meaning that each row of coef\_ contains the coefficient vector for one of the three classes and each column holds the coefficient value for a specific feature (there are two in this dataset). The intercept\_is now a one-dimensional array, storing the intercepts for each class.

Vemos que la forma de coef\_ es (3, 2), lo que significa que cada fila de coef\_ contiene el vector de coeficientes para una de las tres clases y cada columna contiene el valor del coeficiente para una característica específica (hay dos en este conjunto de datos). El intercept\_ es ahora una matriz unidimensional que almacena los interceptos para cada clase.

Let’s visualize the lines given by the three binary classifiers (Figure 2-20):

In[48]:

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y)

line = np.linspace(-15, 15)

for coef, intercept, color in zip(linear\_svm.coef\_, linear\_svm.intercept\_,

mglearn.cm3.colors):

plt.plot(line, -(line \* coef[0] + intercept) / coef[1], c=color)

plt.ylim(-10, 15)

plt.xlim(-10, 8)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',

'Line class 2'], loc=(1.01, 0.3))

You can see that all the points belonging to class 0 in the training data are above the line corresponding to class 0, which means they are on the “class 0” side of this binary classifier. The points in class 0 are above the line corresponding to class 2, which means they are classified as “rest” by the binary classifier for class 2. The points belonging to class 0 are to the left of the line corresponding to class 1, which means the binary classifier for class 1 also classifies them as “rest.” Therefore, any point in this area will be classified as class 0 by the final classifier (the result of the classification confidence formula for classifier 0 is greater than zero, while it is smaller than zero for the other two classes).

Se puede observar que todos los puntos pertenecientes a la clase 0 en los datos de entrenamiento están por encima de la línea correspondiente a la clase 0, lo que significa que están en el lado de la “clase 0” de este clasificador binario. Los puntos de la clase 0 están por encima de la línea correspondiente a la clase 2, lo que significa que el clasificador binario de la clase 2 los clasifica como “resto”. Los puntos pertenecientes a la clase 0 están a la izquierda de la línea correspondiente a la clase 1, lo que significa que el clasificador binario de la clase 1 también los clasifica como “resto”. Por lo tanto, cualquier punto de esta zona será clasificado como clase 0 por el clasificador final (el resultado de la fórmula de confianza de la clasificación para el clasificador 0 es mayor que cero, mientras que es menor que cero para las otras dos clases).

But what about the triangle in the middle of the plot? All three binary classifiers classify points there as “rest.” Which class would a point there be assigned to? The answer is the one with the highest value for the classification formula: the class of the closest line.

Pero ¿qué ocurre con el triángulo que se encuentra en el centro del gráfico? Los tres clasificadores binarios clasifican los puntos que se encuentran allí como “en reposo”. ¿A qué clase se asignaría un punto que se encuentra allí? La respuesta es la que tenga el valor más alto para la fórmula de clasificación: la clase de la línea más cercana.

Figure 2-20. Decision boundaries learned by the three one-vs.-rest classifiers

The following example (Figure 2-21) shows the predictions for all regions of the 2D space:

In[49]:

mglearn.plots.plot\_2d\_classification(linear\_svm, X, fill=True, alpha=.7)

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y)

line = np.linspace(-15, 15)

for coef, intercept, color in zip(linear\_svm.coef\_, linear\_svm.intercept\_,

mglearn.cm3.colors):

plt.plot(line, -(line \* coef[0] + intercept) / coef[1], c=color)

plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',

'Line class 2'], loc=(1.01, 0.3))

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

68

| Chapter 2: Supervised LearningFigure 2-21. Multiclass decision boundaries derived from the three one-vs.-rest classifiers

**Strengths, weaknesses, and parameters**

The main parameter of linear models is the regularization parameter, called alpha in the regression models and C in LinearSVC and LogisticRegression. Large values for alpha or small values for C mean simple models. In particular for the regression models, tuning these parameters is quite important. Usually C and alpha are searched for on a logarithmic scale. The other decision you have to make is whether you want to use L1 regularization or L2 regularization. If you assume that only a few of your features are actually important, you should use L1. Otherwise, you should default to L2. L1 can also be useful if interpretability of the model is important. As L1 will use only a few features, it is easier to explain which features are important to the model, and what the effects of these features are.

El parámetro principal de los modelos lineales es el parámetro de regularización, llamado alfa en los modelos de regresión y C en LinearSVC y LogisticRegression. Los valores altos de alfa o los valores bajos de C significan modelos simples. En particular para los modelos de regresión, ajustar estos parámetros es bastante importante. Por lo general, C y alfa se buscan en una escala logarítmica. La otra decisión que debe tomar es si desea utilizar la regularización L1 o L2. Si asume que solo algunas de sus características son realmente importantes, debe utilizar L1. De lo contrario, debe utilizar L2 de manera predeterminada. L1 también puede ser útil si la interpretabilidad del modelo es importante. Como L1 utilizará solo algunas características, es más fácil explicar qué características son importantes para el modelo y cuáles son los efectos de estas características.

Linear models are very fast to train, and also fast to predict. They scale to very large datasets and work well with sparse data. If your data consists of hundreds of thousands or millions of samples, you might want to investigate using the solver='sag' option in LogisticRegression and Ridge, which can be faster than the default on large datasets. Other options are the SGDClassifier class and the SGDRegressor class, which implement even more scalable versions of the linear models described here.

Los modelos lineales son muy rápidos de entrenar y también rápidos de predecir. Se escalan a conjuntos de datos muy grandes y funcionan bien con datos dispersos. Si sus datos constan de cientos de miles o millones de muestras, es posible que desee investigar utilizando la opción solver='sag' en LogisticRegression y Ridge, que puede ser más rápida que la opción predeterminada en conjuntos de datos grandes. Otras opciones son la clase SGDClassifier y la clase SGDRegressor, que implementan versiones aún más escalables de los modelos lineales descritos aquí.

Another strength of linear models is that they make it relatively easy to understand how a prediction is made, using the formulas we saw earlier for regression and classification. Unfortunately, it is often not entirely clear why coefficients are the way they are. This is particularly true if your dataset has highly correlated features; in these cases, the coefficients might be hard to interpret.

Otra ventaja de los modelos lineales es que permiten comprender con relativa facilidad cómo se hace una predicción, utilizando las fórmulas que vimos antes para la regresión y la clasificación. Lamentablemente, a menudo no queda del todo claro por qué los coeficientes son como son. Esto es particularmente cierto si el conjunto de datos tiene características altamente correlacionadas; en estos casos, los coeficientes pueden ser difíciles de interpretar.

Linear models often perform well when the number of features is large compared to the number of samples. They are also often used on very large datasets, simply because it’s not feasible to train other models. However, in lower-dimensional spaces, other models might yield better generalization performance. We will look at some examples in which linear models fail in “Kernelized Support Vector Machines” on page 94.

Los modelos lineales suelen tener un buen rendimiento cuando la cantidad de características es grande en comparación con la cantidad de muestras. También se suelen utilizar en conjuntos de datos muy grandes, simplemente porque no es posible entrenar otros modelos. Sin embargo, en espacios de menor dimensión, otros modelos pueden ofrecer un mejor rendimiento de generalización. Veremos algunos ejemplos en los que los modelos lineales fallan en “Máquinas de vectores de soporte kernelizadas” en la página 94.

**Method Chaining**

The fit method of all scikit-learn models returns self. This allows you to write code like the following, which we’ve already used extensively in this chapter:

El método de ajuste de todos los modelos de scikit-learn devuelve self. Esto le permite escribir código como el siguiente, que ya hemos utilizado ampliamente en este capítulo:

In[50]:

# instantiate model and fit it in one line

logreg = LogisticRegression().fit(X\_train, y\_train)

Here, we used the return value of fit (which is self) to assign the trained model to the variable logreg. This concatenation of method calls (here \_\_init\_\_ and then fit) is known as method chaining. Another common application of method chaining in scikit-learn is to fit and predict in one line:

Aquí, usamos el valor de retorno de fit (que es self) para asignar el modelo entrenado a la variable logreg. Esta concatenación de llamadas de método (aquí \_\_init\_\_ y luego fit) se conoce como encadenamiento de métodos. Otra aplicación común del encadenamiento de métodos en scikit-learn es ajustar y predecir en una línea:

In[51]:

logreg = LogisticRegression()

y\_pred = logreg.fit(X\_train, y\_train).predict(X\_test)

Finally, you can even do model instantiation, fitting, and predicting in one line:

In[52]:

y\_pred = LogisticRegression().fit(X\_train, y\_train).predict(X\_test)

This very short variant is not ideal, though. A lot is happening in a single line, which might make the code hard to read. Additionally, the fitted logistic regression model isn’t stored in any variable, so we can’t inspect it or use it to predict on any other data.

Sin embargo, esta variante tan corta no es ideal. En una sola línea suceden muchas cosas, lo que puede dificultar la lectura del código. Además, el modelo de regresión logística ajustado no se almacena en ninguna variable, por lo que no podemos inspeccionarlo ni usarlo para hacer predicciones sobre otros datos.

**Naive Bayes Classifiers**

Naive Bayes classifiers are a family of classifiers that are quite similar to the linear models discussed in the previous section. However, they tend to be even faster in training. The price paid for this efficiency is that naive Bayes models often provide generalization performance that is slightly worse than that of linear classifiers like LogisticRegression and LinearSVC.

Los clasificadores Naive Bayes son una familia de clasificadores que son bastante similares a los modelos lineales analizados en la sección anterior. Sin embargo, tienden a ser incluso más rápidos en el entrenamiento. El precio que se paga por esta eficiencia es que los modelos Naive Bayes suelen proporcionar un rendimiento de generalización ligeramente peor que el de los clasificadores lineales como LogisticRegression y LinearSVC.

The reason that naive Bayes models are so efficient is that they learn parameters by looking at each feature individually and collect simple per-class statistics from each feature. There are three kinds of naive Bayes classifiers implemented in scikit-learn: GaussianNB, BernoulliNB, and MultinomialNB. GaussianNB can be applied to any continuous data, while BernoulliNB assumes binary data and MultinomialNB assumes count data (that is, that each feature represents an integer count of something, like how often a word appears in a sentence). BernoulliNB and MultinomialNB are mostly used in text data classification.

La razón por la que los modelos bayesianos ingenuos son tan eficientes es que aprenden los parámetros al observar cada característica individualmente y recopilan estadísticas simples por clase de cada característica. Hay tres tipos de clasificadores bayesianos ingenuos implementados en scikit-learn: GaussianNB, BernoulliNB y MultinomialNB. GaussianNB se puede aplicar a cualquier dato continuo, mientras que BernoulliNB supone datos binarios y MultinomialNB supone datos de conteo (es decir, que cada característica representa un conteo entero de algo, como la frecuencia con la que aparece una palabra en una oración). BernoulliNB y MultinomialNB se utilizan principalmente en la clasificación de datos de texto.

The BernoulliNB classifier counts how often every feature of each class is not zero. This is most easily understood with an example:

In[53]:

X = np.array([[0, 1, 0, 1],

[1, 0, 1, 1],

[0, 0, 0, 1],

[1, 0, 1, 0]])

y = np.array([0, 1, 0, 1])

Here, we have four data points, with four binary features each. There are two classes, 0 and 1. For class 0 (the first and third data points), the first feature is zero two times and nonzero zero times, the second feature is zero one time and nonzero one time, and so on. These same counts are then calculated for the data points in the second class. Counting the nonzero entries per class in essence looks like this:

Aquí, tenemos cuatro puntos de datos, con cuatro características binarias cada uno. Hay dos clases, 0 y 1. Para la clase 0 (el primer y tercer punto de datos), la primera característica es cero dos veces y cero veces distinta de cero, la segunda característica es cero una vez y una vez distinta de cero, y así sucesivamente. Estos mismos recuentos se calculan luego para los puntos de datos de la segunda clase. El recuento de las entradas distintas de cero por clase en esencia se ve así:

In[54]:

counts = {}

for label in np.unique(y):

# iterate over each class

# count (sum) entries of 1 per feature

counts[label] = X[y == label].sum(axis=0)

print("Feature counts:\n{}".format(counts))

Out[54]:

Feature counts:

{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}

The other two naive Bayes models, MultinomialNB and GaussianNB, are slightly different in what kinds of statistics they compute. MultinomialNB takes into account the average value of each feature for each class, while GaussianNB stores the average value as well as the standard deviation of each feature for each class.

Los otros dos modelos bayesianos ingenuos, MultinomialNB y GaussianNB, son ligeramente diferentes en cuanto al tipo de estadísticas que calculan. MultinomialNB tiene en cuenta el valor promedio de cada característica para cada clase, mientras que GaussianNB almacena el valor promedio y la desviación estándar de cada característica para cada clase.

To make a prediction, a data point is compared to the statistics for each of the classes, and the best matching class is predicted. Interestingly, for both MultinomialNB and BernoulliNB, this leads to a prediction formula that is of the same form as in the linear models (see “Linear models for classification” on page 58). Unfortunately, coef\_for the naive Bayes models has a somewhat different meaning than in the linear models, in that coef\_ is not the same as w.

Para hacer una predicción, se compara un punto de datos con las estadísticas de cada una de las clases y se predice la clase que mejor coincide. Curiosamente, tanto para MultinomialNB como para BernoulliNB, esto conduce a una fórmula de predicción que tiene la misma forma que en los modelos lineales (consulte “Modelos lineales para la clasificación” en la página 58). Desafortunadamente, coef\_ para los modelos bayesianos ingenuos tiene un significado algo diferente que en los modelos lineales, ya que coef\_ no es lo mismo que w.

**Strengths, weaknesses, and parameters**

MultinomialNB and BernoulliNB have a single parameter, alpha, which controls model complexity. The way alpha works is that the algorithm adds to the data alpha many virtual data points that have positive values for all the features. This results in a “smoothing” of the statistics. A large alpha means more smoothing, resulting in less complex models. The algorithm’s performance is relatively robust to the setting of alpha, meaning that setting alpha is not critical for good performance. However, tuning it usually improves accuracy somewhat.

MultinomialNB y BernoulliNB tienen un único parámetro, alfa, que controla la complejidad del modelo. La forma en que funciona alfa es que el algoritmo agrega a los datos alfa muchos puntos de datos virtuales que tienen valores positivos para todas las características. Esto da como resultado una “suavizada” de las estadísticas. Un alfa alto significa más suavizado, lo que da como resultado modelos menos complejos. El rendimiento del algoritmo es relativamente robusto a la configuración de alfa, lo que significa que la configuración de alfa no es crítica para un buen rendimiento. Sin embargo, ajustarlo generalmente mejora un poco la precisión.

GaussianNB is mostly used on very high-dimensional data, while the other two variants of naive Bayes are widely used for sparse count data such as text. MultinomialNB usually performs better than BernoulliNB, particularly on datasets with a relatively large number of nonzero features (i.e., large documents).

GaussianNB se utiliza principalmente en datos de dimensiones muy altas, mientras que las otras dos variantes de Naive Bayes se utilizan ampliamente para datos de recuento disperso, como texto. MultinomialNB suele tener un mejor rendimiento que BernoulliNB, en particular en conjuntos de datos con una cantidad relativamente grande de características distintas de cero (es decir, documentos grandes).

The naive Bayes models share many of the strengths and weaknesses of the linear models. They are very fast to train and to predict, and the training procedure is easy to understand. The models work very well with high-dimensional sparse data and are relatively robust to the parameters. Naive Bayes models are great baseline models and are often used on very large datasets, where training even a linear model might take too long.

Los modelos bayesianos ingenuos comparten muchas de las fortalezas y debilidades de los modelos lineales. Son muy rápidos de entrenar y predecir, y el procedimiento de entrenamiento es fácil de entender. Los modelos funcionan muy bien con datos dispersos de alta dimensión y son relativamente robustos a los parámetros. Los modelos bayesianos ingenuos son excelentes modelos de referencia y se utilizan a menudo en conjuntos de datos muy grandes, donde el entrenamiento incluso de un modelo lineal podría llevar demasiado tiempo.

**Decision Trees**

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision. These questions are similar to the questions you might ask in a game of 20 Questions. Imagine you want to distinguish between the following four animals: bears, hawks, penguins, and dolphins. Your goal is to get to the right answer by asking as few if/else questions as possible. You might start off by asking whether the animal has feathers, a question that narrows down your possible animals to just two. If the answer is “yes,” you can ask another question that could help you distinguish between hawks and penguins. For example, you could ask whether the animal can fly. If the animal doesn’t have feathers, your possible animal choices are dolphins and bears, and you will need to ask a question to distinguish between these two animals—for example, asking whether the animal has fins.

Los árboles de decisión son modelos muy utilizados en tareas de clasificación y regresión. Básicamente, aprenden una jerarquía de preguntas if/else que conducen a una decisión. Estas preguntas son similares a las que podrías hacer en un juego de 20 preguntas. Imagina que quieres distinguir entre los siguientes cuatro animales: osos, halcones, pingüinos y delfines. Tu objetivo es llegar a la respuesta correcta haciendo la menor cantidad posible de preguntas if/else. Puedes empezar preguntando si el animal tiene plumas, una pregunta que reduce tus posibles animales a solo dos. Si la respuesta es "sí", puedes hacer otra pregunta que podría ayudarte a distinguir entre halcones y pingüinos. Por ejemplo, podrías preguntar si el animal puede volar. Si el animal no tiene plumas, tus posibles opciones de animales son delfines y osos, y tendrás que hacer una pregunta para distinguir entre estos dos animales, por ejemplo, preguntar si el animal tiene aletas.

This series of questions can be expressed as a decision tree, as shown in Figure 2-22.

In[55]:

mglearn.plots.plot\_animal\_tree()

72

|

Chapter 2: Supervised LearningFigure 2-22. A decision tree to distinguish among several animals

In this illustration, each node in the tree either represents a question or a terminal node (also called a leaf) that contains the answer. The edges connect the answers to a question with the next question you would ask. In machine learning parlance, we built a model to distinguish between four classes of animals (hawks, penguins, dolphins, and bears) using the three features “has feathers,” “can fly,” and “has fins.” Instead of building these models by hand, we can learn them from data using supervised learning.

En esta ilustración, cada nodo del árbol representa una pregunta o un nodo terminal (también llamado hoja) que contiene la respuesta. Los bordes conectan las respuestas a una pregunta con la siguiente pregunta que harías. En el lenguaje del aprendizaje automático, construimos un modelo para distinguir entre cuatro clases de animales (halcones, pingüinos, delfines y osos) utilizando las tres características "tiene plumas", "puede volar" y "tiene aletas". En lugar de construir estos modelos a mano, podemos aprenderlos a partir de datos mediante el aprendizaje supervisado.

**Building decision trees**

Let’s go through the process of building a decision tree for the 2D classification dataset shown in Figure 2-23. The dataset consists of two half-moon shapes, with each class consisting of 75 data points. We will refer to this dataset as two\_moons.

Repasemos el proceso de creación de un árbol de decisiones para el conjunto de datos de clasificación 2D que se muestra en la Figura 2-23. El conjunto de datos consta de dos formas de media luna, y cada clase consta de 75 puntos de datos. Nos referiremos a este conjunto de datos como two\_moons.

Learning a decision tree means learning the sequence of if/else questions that gets us to the true answer most quickly. In the machine learning setting, these questions are called tests (not to be confused with the test set, which is the data we use to test to see how generalizable our model is). Usually data does not come in the form of binary yes/no features as in the animal example, but is instead represented as continuous features such as in the 2D dataset shown in Figure 2-23. The tests that are used on continuous data are of the form “Is feature i larger than value a?”

Aprender un árbol de decisiones significa aprender la secuencia de preguntas if/else que nos lleva a la respuesta verdadera más rápidamente. En el contexto del aprendizaje automático, estas preguntas se denominan pruebas (no deben confundirse con el conjunto de pruebas, que son los datos que utilizamos para probar y ver qué tan generalizable es nuestro modelo). Por lo general, los datos no vienen en forma de características binarias de sí/no como en el ejemplo del animal, sino que se representan como características continuas, como en el conjunto de datos 2D que se muestra en la Figura 2-23. Las pruebas que se utilizan en datos continuos son del tipo "¿La característica i es mayor que el valor a?"

Figure 2-23. Two-moons dataset on which the decision tree will be built

To build a tree, the algorithm searches over all possible tests and finds the one that is most informative about the target variable. Figure 2-24 shows the first test that is picked. Splitting the dataset horizontally at x[1]=0.0596 yields the most information; it best separates the points in class 0 from the points in class 1. The top node, also called the root, represents the whole dataset, consisting of 75 points belonging to class 0 and 75 points belonging to class 1. The split is done by testing whether x[1] <= 0.0596, indicated by a black line. If the test is true, a point is assigned to the left node, which contains 2 points belonging to class 0 and 32 points belonging to class 1. Otherwise the point is assigned to the right node, which contains 48 points belonging to class 0 and 18 points belonging to class 1. These two nodes correspond to the top and bottom regions shown in Figure 2-24. Even though the first split did a good job of separating the two classes, the bottom region still contains points belonging to class 0, and the top region still contains points belonging to class 1. We can build a more accurate model by repeating the process of looking for the best test in both regions. Figure 2-25 shows that the most informative next split for the left and the right region is based on x[0].

Para construir un árbol, el algoritmo busca en todas las pruebas posibles y encuentra la que es más informativa sobre la variable objetivo. La Figura 2-24 muestra la primera prueba que se elige. Dividir el conjunto de datos horizontalmente en x[1]=0,0596 produce la mayor cantidad de información; separa mejor los puntos de la clase 0 de los puntos de la clase 1. El nodo superior, también llamado raíz, representa el conjunto de datos completo, que consta de 75 puntos que pertenecen a la clase 0 y 75 puntos que pertenecen a la clase 1. La división se realiza probando si x[1] <= 0,0596, indicado por una línea negra. Si la prueba es verdadera, se asigna un punto al nodo izquierdo, que contiene 2 puntos que pertenecen a la clase 0 y 32 puntos que pertenecen a la clase 1. De lo contrario, el punto se asigna al nodo derecho, que contiene 48 puntos que pertenecen a la clase 0 y 18 puntos que pertenecen a la clase 1. Estos dos nodos corresponden a las regiones superior e inferior que se muestran en la Figura 2-24. Aunque la primera división hizo un buen trabajo al separar las dos clases, la región inferior aún contiene puntos que pertenecen a la clase 0 y la región superior aún contiene puntos que pertenecen a la clase 1. Podemos construir un modelo más preciso repitiendo el proceso de búsqueda de la mejor prueba en ambas regiones. La Figura 2-25 muestra que la siguiente división más informativa para la región izquierda y la derecha se basa en x[0].

74

|

Chapter 2: Supervised LearningFigure 2-24. Decision boundary of tree with depth 1 (left) and corresponding tree (right)

Figure 2-25. Decision boundary of tree with depth 2 (left) and corresponding decision

tree (right)

This recursive process yields a binary tree of decisions, with each node containing a test. Alternatively, you can think of each test as splitting the part of the data that is currently being considered along one axis. This yields a view of the algorithm as building a hierarchical partition. As each test concerns only a single feature, the regions in the resulting partition always have axis-parallel boundaries.

Este proceso recursivo genera un árbol binario de decisiones, en el que cada nodo contiene una prueba. Otra posibilidad es pensar en cada prueba como si dividiera la parte de los datos que se está considerando actualmente a lo largo de un eje. Esto genera una visión del algoritmo como si estuviera construyendo una partición jerárquica. Como cada prueba se refiere solo a una única característica, las regiones de la partición resultante siempre tienen límites paralelos a los ejes.

The recursive partitioning of the data is repeated until each region in the partition (each leaf in the decision tree) only contains a single target value (a single class or a single regression value). A leaf of the tree that contains data points that all share the same target value is called pure. The final partitioning for this dataset is shown in Figure 2-26.

La partición recursiva de los datos se repite hasta que cada región de la partición (cada hoja del árbol de decisión) solo contenga un único valor objetivo (una única clase o un único valor de regresión). Una hoja del árbol que contiene puntos de datos que comparten el mismo valor objetivo se denomina pura. La partición final de este conjunto de datos se muestra en la Figura 2-26.

Supervised Machine Learning Algorithms

|

75Figure 2-26. Decision boundary of tree with depth 9 (left) and part of the corresponding

tree (right); the full tree is quite large and hard to visualize

A prediction on a new data point is made by checking which region of the partition of the feature space the point lies in, and then predicting the majority target (or the single target in the case of pure leaves) in that region. The region can be found by traversing the tree from the root and going left or right, depending on whether the test is fulfilled or not.

Se realiza una predicción sobre un nuevo punto de datos comprobando en qué región de la partición del espacio de características se encuentra el punto y, a continuación, prediciendo el objetivo mayoritario (o el objetivo único en el caso de hojas puras) en esa región. La región se puede encontrar recorriendo el árbol desde la raíz y yendo hacia la izquierda o hacia la derecha, según se cumpla o no la prueba.

It is also possible to use trees for regression tasks, using exactly the same technique. To make a prediction, we traverse the tree based on the tests in each node and find the leaf the new data point falls into. The output for this data point is the mean target of the training points in this leaf.

También es posible utilizar árboles para tareas de regresión, utilizando exactamente la misma técnica. Para hacer una predicción, recorremos el árbol en función de las pruebas en cada nodo y encontramos la hoja en la que se encuentra el nuevo punto de datos. El resultado de este punto de datos es el objetivo medio de los puntos de entrenamiento en esta hoja.

**Controlling complexity of decision trees**

Typically, building a tree as described here and continuing until all leaves are pure leads to models that are very complex and highly overfit to the training data. The presence of pure leaves mean that a tree is 100% accurate on the training set; each data point in the training set is in a leaf that has the correct majority class. The overfitting can be seen on the left of Figure 2-26. You can see the regions determined to belong to class 1 in the middle of all the points belonging to class 0. On the other hand, there is a small strip predicted as class 0 around the point belonging to class 1 to the very right. This is not how one would imagine the decision boundary to look, and the decision boundary focuses a lot on single outlier points that are far away from the other points in that class.

Por lo general, construir un árbol como se describe aquí y continuar hasta que todas las hojas sean puras conduce a modelos que son muy complejos y altamente sobreajustados a los datos de entrenamiento. La presencia de hojas puras significa que un árbol es 100% preciso en el conjunto de entrenamiento; cada punto de datos en el conjunto de entrenamiento está en una hoja que tiene la clase mayoritaria correcta. El sobreajuste se puede ver a la izquierda de la Figura 2-26. Puede ver las regiones determinadas como pertenecientes a la clase 1 en el medio de todos los puntos pertenecientes a la clase 0. Por otro lado, hay una pequeña franja predicha como clase 0 alrededor del punto perteneciente a la clase 1 a la derecha. Así no es como uno imaginaría que se vería el límite de decisión, y el límite de decisión se centra mucho en puntos atípicos individuales que están lejos de los otros puntos de esa clase.

There are two common strategies to prevent overfitting: stopping the creation of the tree early (also called pre-pruning), or building the tree but then removing or collapsing nodes that contain little information (also called post-pruning or just pruning). Possible criteria for pre-pruning include limiting the maximum depth of the tree, limiting the maximum number of leaves, or requiring a minimum number of points in a node to keep splitting it.

Existen dos estrategias comunes para evitar el sobreajuste: detener la creación del árbol antes de tiempo (también llamada poda previa) o construir el árbol pero luego eliminar o contraer los nodos que contienen poca información (también llamada poda posterior o simplemente poda). Los posibles criterios para la poda previa incluyen limitar la profundidad máxima del árbol, limitar la cantidad máxima de hojas o requerir una cantidad mínima de puntos en un nodo para seguir dividiéndolo.

Decision trees in scikit-learn are implemented in the DecisionTreeRegressor and DecisionTreeClassifier classes. scikit-learn only implements pre-pruning, not post-pruning.

Let’s look at the effect of pre-pruning in more detail on the Breast Cancer dataset. As always, we import the dataset and split it into a training and a test part. Then we build a model using the default setting of fully developing the tree (growing the tree until all leaves are pure). We fix the random\_state in the tree, which is used for tie-breaking internally:

Veamos el efecto de la poda previa con más detalle en el conjunto de datos de cáncer de mama. Como siempre, importamos el conjunto de datos y lo dividimos en una parte de entrenamiento y otra de prueba. Luego, construimos un modelo utilizando la configuración predeterminada de desarrollo completo del árbol (haciendo crecer el árbol hasta que todas las hojas sean puras). Arreglamos el random\_state en el árbol, que se utiliza para desempatar internamente:

In[56]:

from sklearn.tree import DecisionTreeClassifier

cancer = load\_breast\_cancer()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

cancer.data, cancer.target, stratify=cancer.target, random\_state=42)

tree = DecisionTreeClassifier(random\_state=0)

tree.fit(X\_train, y\_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X\_train, y\_train)))

print("Accuracy on test set: {:.3f}".format(tree.score(X\_test, y\_test)))

Out[56]:

Accuracy on training set: 1.000

Accuracy on test set: 0.937

As expected, the accuracy on the training set is 100%—because the leaves are pure, the tree was grown deep enough that it could perfectly memorize all the labels on the training data. The test set accuracy is slightly worse than for the linear models we looked at previously, which had around 95% accuracy.

Como se esperaba, la precisión en el conjunto de entrenamiento es del 100 % (debido a que las hojas son puras, el árbol creció lo suficientemente profundo como para poder memorizar perfectamente todas las etiquetas en los datos de entrenamiento). La precisión del conjunto de prueba es ligeramente peor que la de los modelos lineales que analizamos anteriormente, que tenían una precisión de alrededor del 95 %.

If we don’t restrict the depth of a decision tree, the tree can become arbitrarily deep and complex. Unpruned trees are therefore prone to overfitting and not generalizing well to new data. Now let’s apply pre-pruning to the tree, which will stop developing the tree before we perfectly fit to the training data. One option is to stop building the tree after a certain depth has been reached. Here we set max\_depth=4, meaning only four consecutive questions can be asked (cf. Figures 2-24 and 2-26). Limiting the depth of the tree decreases overfitting. This leads to a lower accuracy on the training set, but an improvement on the test set:

Si no restringimos la profundidad de un árbol de decisión, este puede volverse arbitrariamente profundo y complejo. Por lo tanto, los árboles sin podar son propensos a sobreajustarse y no generalizarse bien a nuevos datos. Ahora apliquemos una poda previa al árbol, que detendrá el desarrollo del árbol antes de que se ajuste perfectamente a los datos de entrenamiento. Una opción es detener la construcción del árbol después de que se haya alcanzado cierta profundidad. Aquí establecemos max\_depth=4, lo que significa que solo se pueden realizar cuatro preguntas consecutivas (consulte las Figuras 2-24 y 2-26). Limitar la profundidad del árbol disminuye el sobreajuste. Esto conduce a una menor precisión en el conjunto de entrenamiento, pero una mejora en el conjunto de prueba:

In[57]:

tree = DecisionTreeClassifier(max\_depth=4, random\_state=0)

tree.fit(X\_train, y\_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X\_train, y\_train)))

print("Accuracy on test set: {:.3f}".format(tree.score(X\_test, y\_test)))

Supervised Machine Learning Algorithms

|

77Out[57]:

Accuracy on training set: 0.988

Accuracy on test set: 0.951

**Analyzing decision trees**

We can visualize the tree using the export\_graphviz function from the tree module. This writes a file in the .dot file format, which is a text file format for storing graphs. We set an option to color the nodes to reflect the majority class in each node and pass the class and features names so the tree can be properly labeled:

Podemos visualizar el árbol utilizando la función export\_graphviz del módulo tree. Esto escribe un archivo en formato .dot, que es un formato de archivo de texto para almacenar gráficos. Establecemos una opción para colorear los nodos para reflejar la clase mayoritaria en cada nodo y pasamos los nombres de clase y características para que el árbol pueda etiquetarse correctamente:

In[58]:

from sklearn.tree import export\_graphviz

export\_graphviz(tree, out\_file="tree.dot", class\_names=["malignant", "benign"],

feature\_names=cancer.feature\_names, impurity=False, filled=True)

We can read this file and visualize it, as seen in Figure 2-27, using the graphviz module (or you can use any program that can read .dot files):

Podemos leer este archivo y visualizarlo, como se ve en la Figura 2-27, usando el módulo graphviz (o puedes usar cualquier programa que pueda leer archivos .dot):

In[59]:

import graphviz

with open("tree.dot") as f:

dot\_graph = f.read()

display(graphviz.Source(dot\_graph))

Figure 2-27. Visualization of the decision tree built on the Breast Cancer dataset

The visualization of the tree provides a great in-depth view of how the algorithm makes predictions, and is a good example of a machine learning algorithm that is easily explained to nonexperts. However, even with a tree of depth four, as seen here, the tree can become a bit overwhelming. Deeper trees (a depth of 10 is not uncommon) are even harder to grasp. One method of inspecting the tree that may be helpful is to find out which path most of the data actually takes. The samples shown in each node in Figure 2-27 gives the number of samples in that node, while value provides the number of samples per class. Following the branches to the right, we see that worst radius > 16.795 creates a node that contains only 8 benign but 134 malignant samples. The rest of this side of the tree then uses some finer distinctions to split off these 8 remaining benign samples. Of the 142 samples that went to the right in the initial split, nearly all of them (132) end up in the leaf to the very right.

La visualización del árbol proporciona una excelente visión en profundidad de cómo el algoritmo hace predicciones y es un buen ejemplo de un algoritmo de aprendizaje automático que se explica fácilmente a los no expertos. Sin embargo, incluso con un árbol de profundidad cuatro, como se ve aquí, el árbol puede volverse un poco abrumador. Los árboles más profundos (una profundidad de 10 no es poco común) son aún más difíciles de comprender. Un método para inspeccionar el árbol que puede ser útil es averiguar qué camino sigue realmente la mayoría de los datos. Las muestras que se muestran en cada nodo en la Figura 2-27 indican la cantidad de muestras en ese nodo, mientras que el valor indica la cantidad de muestras por clase. Siguiendo las ramas hacia la derecha, vemos que el peor radio > 16,795 crea un nodo que contiene solo 8 muestras benignas pero 134 malignas. El resto de este lado del árbol utiliza algunas distinciones más finas para separar estas 8 muestras benignas restantes. De las 142 muestras que fueron a la derecha en la división inicial, casi todas (132) terminan en la hoja de la extrema derecha.

Taking a left at the root, for worst radius <= 16.795 we end up with 25 malignant and 259 benign samples. Nearly all of the benign samples end up in the second leaf from the left, with most of the other leaves containing very few samples.

Si tomamos la raíz hacia la izquierda, para el peor radio <= 16,795, obtenemos 25 muestras malignas y 259 benignas. Casi todas las muestras benignas terminan en la segunda hoja desde la izquierda, y la mayoría de las otras hojas contienen muy pocas muestras.

**Feature importance in trees**

Instead of looking at the whole tree, which can be taxing, there are some useful properties that we can derive to summarize the workings of the tree. The most commonly used summary is feature importance, which rates how important each feature is for the decision a tree makes. It is a number between 0 and 1 for each feature, where 0 means “not used at all” and 1 means “perfectly predicts the target.” The feature importances always sum to 1:

En lugar de observar el árbol completo, lo que puede resultar complicado, existen algunas propiedades útiles que podemos derivar para resumir el funcionamiento del árbol. El resumen más utilizado es la importancia de las características, que califica la importancia de cada característica para la decisión que toma un árbol. Es un número entre 0 y 1 para cada característica, donde 0 significa “no se utiliza en absoluto” y 1 significa “predice perfectamente el objetivo”. La importancia de las características siempre suma 1:

In[60]:

print("Feature importances:\n{}".format(tree.feature\_importances\_))

Out[60]:

Feature importances:

[ 0.

0.

0.

0.

0.

0.

0.

0.

0.048 0.

0.

0.002 0.

0.

0.

0.

0.

0.

0.014 0.

0.018 0.122 0.012 0.

0.

0.

0.

0.01

0.727 0.046

]

We can visualize the feature importances in a way that is similar to the way we visualize the coefficients in the linear model (Figure 2-28):

Podemos visualizar la importancia de las características de una manera similar a la forma en que visualizamos los coeficientes en el modelo lineal (Figura 2-28):

In[61]:

def plot\_feature\_importances\_cancer(model):

n\_features = cancer.data.shape[1]

plt.barh(range(n\_features), model.feature\_importances\_, align='center')

plt.yticks(np.arange(n\_features), cancer.feature\_names)

plt.xlabel("Feature importance")

plt.ylabel("Feature")

plt.ylim(-1, n\_features)

plot\_feature\_importances\_cancer(tree)

Supervised Machine Learning Algorithms

|

Figure 2-28. Feature importances computed from a decision tree learned on the Breast Cancer dataset

Here we see that the feature used in the top split (“worst radius”) is by far the most important feature. This confirms our observation in analyzing the tree that the first level already separates the two classes fairly well.

Aquí vemos que la característica utilizada en la división superior (“peor radio”) es, con diferencia, la característica más importante. Esto confirma nuestra observación al analizar el árbol de que el primer nivel ya separa bastante bien las dos clases.

However, if a feature has a low value in feature\_importance\_, it doesn’t mean that this feature is uninformative. It only means that the feature was not picked by the tree, likely because another feature encodes the same information.

Sin embargo, si una característica tiene un valor bajo en feature\_importance\_, no significa que esa característica no sea informativa. Solo significa que el árbol no la eligió, probablemente porque otra característica codifica la misma información.

In contrast to the coefficients in linear models, feature importances are always positive, and don’t encode which class a feature is indicative of. The feature importances tell us that “worst radius” is important, but not whether a high radius is indicative of a sample being benign or malignant. In fact, there might not be such a simple relationship between features and class, as you can see in the following example (Figures 2-29 and 2-30):

A diferencia de los coeficientes de los modelos lineales, las importancias de las características son siempre positivas y no codifican a qué clase corresponde una característica. Las importancias de las características nos indican que el “peor radio” es importante, pero no si un radio alto es indicativo de que una muestra es benigna o maligna. De hecho, puede que no exista una relación tan simple entre las características y la clase, como se puede ver en el siguiente ejemplo (Figuras 2-29 y 2-30):

In[62]:

tree = mglearn.plots.plot\_tree\_not\_monotone()

display(tree)

Out[62]:

Feature importances: [ 0.

80

|

Chapter 2: Supervised Learning

1.]Figure 2-29. A two-dimensional dataset in which the feature on the y-axis has a nonmo‐

notonous relationship with the class label, and the decision boundaries found by a deci‐

sion tree

Figure 2-30. Decision tree learned on the data shown in Figure 2-29

The plot shows a dataset with two features and two classes. Here, all the information is contained in X[1], and X[0] is not used at all. But the relation between X[1] and the output class is not monotonous, meaning we cannot say “a high value of X[1] means class 0, and a low value means class 1” (or vice versa).

El gráfico muestra un conjunto de datos con dos características y dos clases. Aquí, toda la información está contenida en X[1] y X[0] no se utiliza en absoluto. Pero la relación entre X[1] y la clase de salida no es monótona, lo que significa que no podemos decir “un valor alto de X[1] significa clase 0 y un valor bajo significa clase 1” (o viceversa).

While we focused our discussion here on decision trees for classification, all that was said is similarly true for decision trees for regression, as implemented in Decision TreeRegressor. The usage and analysis of regression trees is very similar to that of classification trees. There is one particular property of using tree-based models for regression that we want to point out, though. The DecisionTreeRegressor (and all other tree-based regression models) is not able to extrapolate, or make predictions outside of the range of the training data.

Si bien centramos nuestra discusión aquí en los árboles de decisión para la clasificación, todo lo que se dijo es igualmente cierto para los árboles de decisión para la regresión, tal como se implementa en Decision TreeRegressor. El uso y análisis de los árboles de regresión es muy similar al de los árboles de clasificación. Sin embargo, hay una propiedad particular del uso de modelos basados ​​en árboles para la regresión que queremos señalar. DecisionTreeRegressor (y todos los demás modelos de regresión basados ​​en árboles) no puede extrapolar ni hacer predicciones fuera del rango de los datos de entrenamiento.

Let’s look into this in more detail, using a dataset of historical computer memory (RAM) prices. Figure 2-31 shows the dataset, with the date on the x-axis and the price of one megabyte of RAM in that year on the y-axis:

In[63]:

import os

ram\_prices = pd.read\_csv(os.path.join(mglearn.datasets.DATA\_PATH,

"ram\_price.csv"))

plt.semilogy(ram\_prices.date, ram\_prices.price)

plt.xlabel("Year")

plt.ylabel("Price in $/Mbyte")

Figure 2-31. Historical development of the price of RAM, plotted on a log scale

82

|

Chapter 2: Supervised LearningNote the logarithmic scale of the y-axis. When plotting logarithmically, the relation

seems to be quite linear and so should be relatively easy to predict, apart from some

bumps.

We will make a forecast for the years after 2000 using the historical data up to that

point, with the date as our only feature. We will compare two simple models: a

DecisionTreeRegressor and LinearRegression. We rescale the prices using a loga‐

rithm, so that the relationship is relatively linear. This doesn’t make a difference for

the DecisionTreeRegressor, but it makes a big difference for LinearRegression (we

will discuss this in more depth in Chapter 4). After training the models and making

predictions, we apply the exponential map to undo the logarithm transform. We

make predictions on the whole dataset for visualization purposes here, but for a

quantitative evaluation we would only consider the test dataset:

In[64]:

from sklearn.tree import DecisionTreeRegressor

# use historical data to forecast prices after the year 2000

data\_train = ram\_prices[ram\_prices.date < 2000]

data\_test = ram\_prices[ram\_prices.date >= 2000]

# predict prices based on date

X\_train = data\_train.date[:, np.newaxis]

# we use a log-transform to get a simpler relationship of data to target

y\_train = np.log(data\_train.price)

tree = DecisionTreeRegressor().fit(X\_train, y\_train)

linear\_reg = LinearRegression().fit(X\_train, y\_train)

# predict on all data

X\_all = ram\_prices.date[:, np.newaxis]

pred\_tree = tree.predict(X\_all)

pred\_lr = linear\_reg.predict(X\_all)

# undo log-transform

price\_tree = np.exp(pred\_tree)

price\_lr = np.exp(pred\_lr)

Figure 2-32, created here, compares the predictions of the decision tree and the linear

regression model with the ground truth:

In[65]:

plt.semilogy(data\_train.date, data\_train.price, label="Training data")

plt.semilogy(data\_test.date, data\_test.price, label="Test data")

plt.semilogy(ram\_prices.date, price\_tree, label="Tree prediction")

plt.semilogy(ram\_prices.date, price\_lr, label="Linear prediction")

plt.legend()

Supervised Machine Learning Algorithms

|

83Figure 2-32. Comparison of predictions made by a linear model and predictions made

by a regression tree on the RAM price data

The difference between the models is quite striking. The linear model approximates

the data with a line, as we knew it would. This line provides quite a good forecast for

the test data (the years after 2000), while glossing over some of the finer variations in

both the training and the test data. The tree model, on the other hand, makes perfect

predictions on the training data; we did not restrict the complexity of the tree, so it

learned the whole dataset by heart. However, once we leave the data range for which

the model has data, the model simply keeps predicting the last known point. The tree

has no ability to generate “new” responses, outside of what was seen in the training

data. This shortcoming applies to all models based on trees.9

Strengths, weaknesses, and parameters

As discussed earlier, the parameters that control model complexity in decision trees

are the pre-pruning parameters that stop the building of the tree before it is fully

developed. Usually, picking one of the pre-pruning strategies—setting either

9 It is actually possible to make very good forecasts with tree-based models (for example, when trying to predict

whether a price will go up or down). The point of this example was not to show that trees are a bad model for

time series, but to illustrate a particular property of how trees make predictions.

84

|

Chapter 2: Supervised Learningmax\_depth, max\_leaf\_nodes, or min\_samples\_leaf—is sufficient to prevent overfit‐

ting.

Decision trees have two advantages over many of the algorithms we’ve discussed so

far: the resulting model can easily be visualized and understood by nonexperts (at

least for smaller trees), and the algorithms are completely invariant to scaling of the

data. As each feature is processed separately, and the possible splits of the data don’t

depend on scaling, no preprocessing like normalization or standardization of features

is needed for decision tree algorithms. In particular, decision trees work well when

you have features that are on completely different scales, or a mix of binary and con‐

tinuous features.

The main downside of decision trees is that even with the use of pre-pruning, they

tend to overfit and provide poor generalization performance. Therefore, in most

applications, the ensemble methods we discuss next are usually used in place of a sin‐

gle decision tree.

Ensembles of Decision Trees

Ensembles are methods that combine multiple machine learning models to create

more powerful models. There are many models in the machine learning literature

that belong to this category, but there are two ensemble models that have proven to

be effective on a wide range of datasets for classification and regression, both of

which use decision trees as their building blocks: random forests and gradient boos‐

ted decision trees.

Random forests

As we just observed, a main drawback of decision trees is that they tend to overfit the

training data. Random forests are one way to address this problem. A random forest

is essentially a collection of decision trees, where each tree is slightly different from

the others. The idea behind random forests is that each tree might do a relatively

good job of predicting, but will likely overfit on part of the data. If we build many

trees, all of which work well and overfit in different ways, we can reduce the amount

of overfitting by averaging their results. This reduction in overfitting, while retaining

the predictive power of the trees, can be shown using rigorous mathematics.

To implement this strategy, we need to build many decision trees. Each tree should do

an acceptable job of predicting the target, and should also be different from the other

trees. Random forests get their name from injecting randomness into the tree build‐

ing to ensure each tree is different. There are two ways in which the trees in a random

forest are randomized: by selecting the data points used to build a tree and by select‐

ing the features in each split test. Let’s go into this process in more detail.

Supervised Machine Learning Algorithms

|

85Building random forests. To build a random forest model, you need to decide on the

number of trees to build (the n\_estimators parameter of RandomForestRegressor or

RandomForestClassifier). Let’s say we want to build 10 trees. These trees will be

built completely independently from each other, and the algorithm will make differ‐

ent random choices for each tree to make sure the trees are distinct. To build a tree,

we first take what is called a bootstrap sample of our data. That is, from our n\_samples

data points, we repeatedly draw an example randomly with replacement (meaning the

same sample can be picked multiple times), n\_samples times. This will create a data‐

set that is as big as the original dataset, but some data points will be missing from it

(approximately one third), and some will be repeated.

To illustrate, let’s say we want to create a bootstrap sample of the list ['a', 'b',

'c', 'd']. A possible bootstrap sample would be ['b', 'd', 'd', 'c']. Another

possible sample would be ['d', 'a', 'd', 'a'].

Next, a decision tree is built based on this newly created dataset. However, the algo‐

rithm we described for the decision tree is slightly modified. Instead of looking for

the best test for each node, in each node the algorithm randomly selects a subset of

the features, and it looks for the best possible test involving one of these features. The

number of features that are selected is controlled by the max\_features parameter.

This selection of a subset of features is repeated separately in each node, so that each

node in a tree can make a decision using a different subset of the features.

The bootstrap sampling leads to each decision tree in the random forest being built

on a slightly different dataset. Because of the selection of features in each node, each

split in each tree operates on a different subset of features. Together, these two mech‐

anisms ensure that all the trees in the random forest are different.

A critical parameter in this process is max\_features. If we set max\_features to n\_fea

tures, that means that each split can look at all features in the dataset, and no ran‐

domness will be injected in the feature selection (the randomness due to the

bootstrapping remains, though). If we set max\_features to 1, that means that the

splits have no choice at all on which feature to test, and can only search over different

thresholds for the feature that was selected randomly. Therefore, a high max\_fea

tures means that the trees in the random forest will be quite similar, and they will be

able to fit the data easily, using the most distinctive features. A low max\_features

means that the trees in the random forest will be quite different, and that each tree

might need to be very deep in order to fit the data well.

To make a prediction using the random forest, the algorithm first makes a prediction

for every tree in the forest. For regression, we can average these results to get our final

prediction. For classification, a “soft voting” strategy is used. This means each algo‐

rithm makes a “soft” prediction, providing a probability for each possible output

86

|

Chapter 2: Supervised Learninglabel. The probabilities predicted by all the trees are averaged, and the class with the

highest probability is predicted.

Analyzing random forests. Let’s apply a random forest consisting of five trees to the

two\_moons dataset we studied earlier:

In[66]:

from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import make\_moons

X, y = make\_moons(n\_samples=100, noise=0.25, random\_state=3)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y,

random\_state=42)

forest = RandomForestClassifier(n\_estimators=5, random\_state=2)

forest.fit(X\_train, y\_train)

The trees that are built as part of the random forest are stored in the estimator\_

attribute. Let’s visualize the decision boundaries learned by each tree, together with

their aggregate prediction as made by the forest (Figure 2-33):

In[67]:

fig, axes = plt.subplots(2, 3, figsize=(20, 10))

for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators\_)):

ax.set\_title("Tree {}".format(i))

mglearn.plots.plot\_tree\_partition(X\_train, y\_train, tree, ax=ax)

mglearn.plots.plot\_2d\_separator(forest, X\_train, fill=True, ax=axes[-1, -1],

alpha=.4)

axes[-1, -1].set\_title("Random Forest")

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train)

You can clearly see that the decision boundaries learned by the five trees are quite dif‐

ferent. Each of them makes some mistakes, as some of the training points that are

plotted here were not actually included in the training sets of the trees, due to the

bootstrap sampling.

The random forest overfits less than any of the trees individually, and provides a

much more intuitive decision boundary. In any real application, we would use many

more trees (often hundreds or thousands), leading to even smoother boundaries.

Supervised Machine Learning Algorithms

|

87Figure 2-33. Decision boundaries found by five randomized decision trees and the deci‐

sion boundary obtained by averaging their predicted probabilities

As another example, let’s apply a random forest consisting of 100 trees on the Breast

Cancer dataset:

In[68]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

cancer.data, cancer.target, random\_state=0)

forest = RandomForestClassifier(n\_estimators=100, random\_state=0)

forest.fit(X\_train, y\_train)

print("Accuracy on training set: {:.3f}".format(forest.score(X\_train, y\_train)))

print("Accuracy on test set: {:.3f}".format(forest.score(X\_test, y\_test)))

Out[68]:

Accuracy on training set: 1.000

Accuracy on test set: 0.972

The random forest gives us an accuracy of 97%, better than the linear models or a

single decision tree, without tuning any parameters. We could adjust the max\_fea

tures setting, or apply pre-pruning as we did for the single decision tree. However,

often the default parameters of the random forest already work quite well.

Similarly to the decision tree, the random forest provides feature importances, which

are computed by aggregating the feature importances over the trees in the forest. Typ‐

ically, the feature importances provided by the random forest are more reliable than

the ones provided by a single tree. Take a look at Figure 2-34.

88

|

Chapter 2: Supervised LearningIn[69]:

plot\_feature\_importances\_cancer(forest)

Figure 2-34. Feature importances computed from a random forest that was fit to the

Breast Cancer dataset

As you can see, the random forest gives nonzero importance to many more features

than the single tree. Similarly to the single decision tree, the random forest also gives

a lot of importance to the “worst radius” feature, but it actually chooses “worst perim‐

eter” to be the most informative feature overall. The randomness in building the ran‐

dom forest forces the algorithm to consider many possible explanations, the result

being that the random forest captures a much broader picture of the data than a sin‐

gle tree.

Strengths, weaknesses, and parameters. Random forests for regression and classifica‐

tion are currently among the most widely used machine learning methods. They are

very powerful, often work well without heavy tuning of the parameters, and don’t

require scaling of the data.

Essentially, random forests share all of the benefits of decision trees, while making up

for some of their deficiencies. One reason to still use decision trees is if you need a

compact representation of the decision-making process. It is basically impossible to

interpret tens or hundreds of trees in detail, and trees in random forests tend to be

deeper than decision trees (because of the use of feature subsets). Therefore, if you

need to summarize the prediction making in a visual way to nonexperts, a single

decision tree might be a better choice. While building random forests on large data‐

sets might be somewhat time consuming, it can be parallelized across multiple CPU

Supervised Machine Learning Algorithms

|

89cores within a computer easily. If you are using a multi-core processor (as nearly all

modern computers do), you can use the n\_jobs parameter to adjust the number of

cores to use. Using more CPU cores will result in linear speed-ups (using two cores,

the training of the random forest will be twice as fast), but specifying n\_jobs larger

than the number of cores will not help. You can set n\_jobs=-1 to use all the cores in

your computer.

You should keep in mind that random forests, by their nature, are random, and set‐

ting different random states (or not setting the random\_state at all) can drastically

change the model that is built. The more trees there are in the forest, the more robust

it will be against the choice of random state. If you want to have reproducible results,

it is important to fix the random\_state.

Random forests don’t tend to perform well on very high dimensional, sparse data,

such as text data. For this kind of data, linear models might be more appropriate.

Random forests usually work well even on very large datasets, and training can easily

be parallelized over many CPU cores within a powerful computer. However, random

forests require more memory and are slower to train and to predict than linear mod‐

els. If time and memory are important in an application, it might make sense to use a

linear model instead.

The important parameters to adjust are n\_estimators, max\_features, and possibly

pre-pruning options like max\_depth. For n\_estimators, larger is always better. Aver‐

aging more trees will yield a more robust ensemble by reducing overfitting. However,

there are diminishing returns, and more trees need more memory and more time to

train. A common rule of thumb is to build “as many as you have time/memory for.”

As described earlier, max\_features determines how random each tree is, and a

smaller max\_features reduces overfitting. In general, it’s a good rule of thumb to use

the default values: max\_features=sqrt(n\_features) for classification and max\_fea

tures=n\_features for regression. Adding max\_features or max\_leaf\_nodes might

sometimes improve performance. It can also drastically reduce space and time

requirements for training and prediction.

Gradient boosted regression trees (gradient boosting machines)

The gradient boosted regression tree is another ensemble method that combines mul‐

tiple decision trees to create a more powerful model. Despite the “regression” in the

name, these models can be used for regression and classification. In contrast to the

random forest approach, gradient boosting works by building trees in a serial man‐

ner, where each tree tries to correct the mistakes of the previous one. By default, there

is no randomization in gradient boosted regression trees; instead, strong pre-pruning

is used. Gradient boosted trees often use very shallow trees, of depth one to five,

which makes the model smaller in terms of memory and makes predictions faster.

90

|

Chapter 2: Supervised LearningThe main idea behind gradient boosting is to combine many simple models (in this

context known as weak learners), like shallow trees. Each tree can only provide good

predictions on part of the data, and so more and more trees are added to iteratively

improve performance.

Gradient boosted trees are frequently the winning entries in machine learning com‐

petitions, and are widely used in industry. They are generally a bit more sensitive to

parameter settings than random forests, but can provide better accuracy if the param‐

eters are set correctly.

Apart from the pre-pruning and the number of trees in the ensemble, another impor‐

tant parameter of gradient boosting is the learning\_rate, which controls how

strongly each tree tries to correct the mistakes of the previous trees. A higher learning

rate means each tree can make stronger corrections, allowing for more complex mod‐

els. Adding more trees to the ensemble, which can be accomplished by increasing

n\_estimators, also increases the model complexity, as the model has more chances

to correct mistakes on the training set.

Here is an example of using GradientBoostingClassifier on the Breast Cancer

dataset. By default, 100 trees of maximum depth 3 and a learning rate of 0.1 are used:

In[70]:

from sklearn.ensemble import GradientBoostingClassifier

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

cancer.data, cancer.target, random\_state=0)

gbrt = GradientBoostingClassifier(random\_state=0)

gbrt.fit(X\_train, y\_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X\_train, y\_train)))

print("Accuracy on test set: {:.3f}".format(gbrt.score(X\_test, y\_test)))

Out[70]:

Accuracy on training set: 1.000

Accuracy on test set: 0.958

As the training set accuracy is 100%, we are likely to be overfitting. To reduce overfit‐

ting, we could either apply stronger pre-pruning by limiting the maximum depth or

lower the learning rate:

Supervised Machine Learning Algorithms

|

91In[71]:

gbrt = GradientBoostingClassifier(random\_state=0, max\_depth=1)

gbrt.fit(X\_train, y\_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X\_train, y\_train)))

print("Accuracy on test set: {:.3f}".format(gbrt.score(X\_test, y\_test)))

Out[71]:

Accuracy on training set: 0.991

Accuracy on test set: 0.972

In[72]:

gbrt = GradientBoostingClassifier(random\_state=0, learning\_rate=0.01)

gbrt.fit(X\_train, y\_train)

print("Accuracy on training set: {:.3f}".format(gbrt.score(X\_train, y\_train)))

print("Accuracy on test set: {:.3f}".format(gbrt.score(X\_test, y\_test)))

Out[72]:

Accuracy on training set: 0.988

Accuracy on test set: 0.965

Both methods of decreasing the model complexity reduced the training set accuracy,

as expected. In this case, lowering the maximum depth of the trees provided a signifi‐

cant improvement of the model, while lowering the learning rate only increased the

generalization performance slightly.

As for the other decision tree–based models, we can again visualize the feature

importances to get more insight into our model (Figure 2-35). As we used 100 trees, it

is impractical to inspect them all, even if they are all of depth 1:

In[73]:

gbrt = GradientBoostingClassifier(random\_state=0, max\_depth=1)

gbrt.fit(X\_train, y\_train)

plot\_feature\_importances\_cancer(gbrt)

92

|

Chapter 2: Supervised LearningFigure 2-35. Feature importances computed from a gradient boosting classifier that was

fit to the Breast Cancer dataset

We can see that the feature importances of the gradient boosted trees are somewhat

similar to the feature importances of the random forests, though the gradient boost‐

ing completely ignored some of the features.

As both gradient boosting and random forests perform well on similar kinds of data,

a common approach is to first try random forests, which work quite robustly. If ran‐

dom forests work well but prediction time is at a premium, or it is important to

squeeze out the last percentage of accuracy from the machine learning model, mov‐

ing to gradient boosting often helps.

If you want to apply gradient boosting to a large-scale problem, it might be worth

looking into the xgboost package and its Python interface, which at the time of writ‐

ing is faster (and sometimes easier to tune) than the scikit-learn implementation of

gradient boosting on many datasets.

Strengths, weaknesses, and parameters. Gradient boosted decision trees are among the

most powerful and widely used models for supervised learning. Their main drawback

is that they require careful tuning of the parameters and may take a long time to

train. Similarly to other tree-based models, the algorithm works well without scaling

and on a mixture of binary and continuous features. As with other tree-based models,

it also often does not work well on high-dimensional sparse data.

The main parameters of gradient boosted tree models are the number of trees, n\_esti

mators, and the learning\_rate, which controls the degree to which each tree is

allowed to correct the mistakes of the previous trees. These two parameters are highly

Supervised Machine Learning Algorithms

|

93interconnected, as a lower learning\_rate means that more trees are needed to build

a model of similar complexity. In contrast to random forests, where a higher n\_esti

mators value is always better, increasing n\_estimators in gradient boosting leads to a

more complex model, which may lead to overfitting. A common practice is to fit

n\_estimators depending on the time and memory budget, and then search over dif‐

ferent learning\_rates.

Another important parameter is max\_depth (or alternatively max\_leaf\_nodes), to

reduce the complexity of each tree. Usually max\_depth is set very low for gradient

boosted models, often not deeper than five splits.

Kernelized Support Vector Machines

The next type of supervised model we will discuss is kernelized support vector

machines. We explored the use of linear support vector machines for classification in

“Linear models for classification” on page 58. Kernelized support vector machines

(often just referred to as SVMs) are an extension that allows for more complex mod‐

els that are not defined simply by hyperplanes in the input space. While there are sup‐

port vector machines for classification and regression, we will restrict ourselves to the

classification case, as implemented in SVC. Similar concepts apply to support vector

regression, as implemented in SVR.

The math behind kernelized support vector machines is a bit involved, and is beyond

the scope of this book. You can find the details in Chapter 12 of Hastie, Tibshirani,

and Friedman’s The Elements of Statistical Learning. However, we will try to give you

some sense of the idea behind the method.

Linear models and nonlinear features

As you saw in Figure 2-15, linear models can be quite limiting in low-dimensional

spaces, as lines and hyperplanes have limited flexibility. One way to make a linear

model more flexible is by adding more features—for example, by adding interactions

or polynomials of the input features.

Let’s look at the synthetic dataset we used in “Feature importance in trees” on page 79

(see Figure 2-29):

In[74]:

X, y = make\_blobs(centers=4, random\_state=8)

y = y % 2

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

94

|

Chapter 2: Supervised LearningFigure 2-36. Two-class classification dataset in which classes are not linearly separable

A linear model for classification can only separate points using a line, and will not be

able to do a very good job on this dataset (see Figure 2-37):

In[75]:

from sklearn.svm import LinearSVC

linear\_svm = LinearSVC().fit(X, y)

mglearn.plots.plot\_2d\_separator(linear\_svm, X)

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

Now let’s expand the set of input features, say by also adding feature1 \*\* 2, the

square of the second feature, as a new feature. Instead of representing each data point

as a two-dimensional point, (feature0, feature1), we now represent it as a three-

dimensional point, (feature0, feature1, feature1 \*\* 2).10 This new representa‐

tion is illustrated in Figure 2-38 in a three-dimensional scatter plot:

10 We picked this particular feature to add for illustration purposes. The choice is not particularly important.

Supervised Machine Learning Algorithms

|

95Figure 2-37. Decision boundary found by a linear SVM

In[76]:

# add the squared second feature

X\_new = np.hstack([X, X[:, 1:] \*\* 2])

from mpl\_toolkits.mplot3d import Axes3D, axes3d

figure = plt.figure()

# visualize in 3D

ax = Axes3D(figure, elev=-152, azim=-26)

# plot first all the points with y == 0, then all with y == 1

mask = y == 0

ax.scatter(X\_new[mask, 0], X\_new[mask, 1], X\_new[mask, 2], c='b',

cmap=mglearn.cm2, s=60, edgecolor='k')

ax.scatter(X\_new[~mask, 0], X\_new[~mask, 1], X\_new[~mask, 2], c='r', marker='^',

cmap=mglearn.cm2, s=60, edgecolor='k')

ax.set\_xlabel("feature0")

ax.set\_ylabel("feature1")

ax.set\_zlabel("feature1 \*\* 2")

96

|

Chapter 2: Supervised LearningFigure 2-38. Expansion of the dataset shown in Figure 2-37, created by adding a third

feature derived from feature1

In the new representation of the data, it is now indeed possible to separate the two

classes using a linear model, a plane in three dimensions. We can confirm this by fit‐

ting a linear model to the augmented data (see Figure 2-39):

In[77]:

linear\_svm\_3d = LinearSVC().fit(X\_new, y)

coef, intercept = linear\_svm\_3d.coef\_.ravel(), linear\_svm\_3d.intercept\_

# show linear decision boundary

figure = plt.figure()

ax = Axes3D(figure, elev=-152, azim=-26)

xx = np.linspace(X\_new[:, 0].min() - 2, X\_new[:, 0].max() + 2, 50)

yy = np.linspace(X\_new[:, 1].min() - 2, X\_new[:, 1].max() + 2, 50)

XX, YY = np.meshgrid(xx, yy)

ZZ = (coef[0] \* XX + coef[1] \* YY + intercept) / -coef[2]

ax.plot\_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)

ax.scatter(X\_new[mask, 0], X\_new[mask, 1], X\_new[mask, 2], c='b',

cmap=mglearn.cm2, s=60, edgecolor='k')

ax.scatter(X\_new[~mask, 0], X\_new[~mask, 1], X\_new[~mask, 2], c='r', marker='^',

cmap=mglearn.cm2, s=60, edgecolor='k')

ax.set\_xlabel("feature0")

ax.set\_ylabel("feature1")

ax.set\_zlabel("feature1 \*\* 2")

Supervised Machine Learning Algorithms

|

97Figure 2-39. Decision boundary found by a linear SVM on the expanded three-

dimensional dataset

As a function of the original features, the linear SVM model is not actually linear any‐

more. It is not a line, but more of an ellipse, as you can see from the plot created here

(Figure 2-40):

In[78]:

ZZ = YY \*\* 2

dec = linear\_svm\_3d.decision\_function(np.c\_[XX.ravel(), YY.ravel(), ZZ.ravel()])

plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],

cmap=mglearn.cm2, alpha=0.5)

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

98

| Chapter 2: Supervised LearningFigure 2-40. The decision boundary from Figure 2-39 as a function of the original two

features

The kernel trick

The lesson here is that adding nonlinear features to the representation of our data can

make linear models much more powerful. However, often we don’t know which fea‐

tures to add, and adding many features (like all possible interactions in a 100-

dimensional feature space) might make computation very expensive. Luckily, there is

a clever mathematical trick that allows us to learn a classifier in a higher-dimensional

space without actually computing the new, possibly very large representation. This is

known as the kernel trick, and it works by directly computing the distance (more pre‐

cisely, the scalar products) of the data points for the expanded feature representation,

without ever actually computing the expansion.

There are two ways to map your data into a higher-dimensional space that are com‐

monly used with support vector machines: the polynomial kernel, which computes all

possible polynomials up to a certain degree of the original features (like feature1 \*\*

2 \* feature2 \*\* 5); and the radial basis function (RBF) kernel, also known as the

Gaussian kernel. The Gaussian kernel is a bit harder to explain, as it corresponds to

an infinite-dimensional feature space. One way to explain the Gaussian kernel is that

Supervised Machine Learning Algorithms

|

99it considers all possible polynomials of all degrees, but the importance of the features

decreases for higher degrees.11

In practice, the mathematical details behind the kernel SVM are not that important,

though, and how an SVM with an RBF kernel makes a decision can be summarized

quite easily—we’ll do so in the next section.

Understanding SVMs

During training, the SVM learns how important each of the training data points is to

represent the decision boundary between the two classes. Typically only a subset of

the training points matter for defining the decision boundary: the ones that lie on the

border between the classes. These are called support vectors and give the support vec‐

tor machine its name.

To make a prediction for a new point, the distance to each of the support vectors is

measured. A classification decision is made based on the distances to the support vec‐

tor, and the importance of the support vectors that was learned during training

(stored in the dual\_coef\_ attribute of SVC).

The distance between data points is measured by the Gaussian kernel:

krbf(x1, x2) = exp (–ɣǁx1 - x2ǁ2)

Here, x1 and x2 are data points, ǁ x1 - x2 ǁ denotes Euclidean distance, and ɣ (gamma)

is a parameter that controls the width of the Gaussian kernel.

Figure 2-41 shows the result of training a support vector machine on a two-

dimensional two-class dataset. The decision boundary is shown in black, and the sup‐

port vectors are larger points with the wide outline. The following code creates this

plot by training an SVM on the forge dataset:

In[79]:

from sklearn.svm import SVC

X, y = mglearn.tools.make\_handcrafted\_dataset()

svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)

mglearn.plots.plot\_2d\_separator(svm, X, eps=.5)

mglearn.discrete\_scatter(X[:, 0], X[:, 1], y)

# plot support vectors

sv = svm.support\_vectors\_

# class labels of support vectors are given by the sign of the dual coefficients

sv\_labels = svm.dual\_coef\_.ravel() > 0

mglearn.discrete\_scatter(sv[:, 0], sv[:, 1], sv\_labels, s=15, markeredgewidth=3)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

11 This follows from the Taylor expansion of the exponential map.

100

|

Chapter 2: Supervised LearningFigure 2-41. Decision boundary and support vectors found by an SVM with RBF kernel

In this case, the SVM yields a very smooth and nonlinear (not a straight line) bound‐

ary. We adjusted two parameters here: the C parameter and the gamma parameter,

which we will now discuss in detail.

Tuning SVM parameters

The gamma parameter is the one shown in the formula given in the previous section,

which corresponds to the inverse of the width of the Gaussian kernel. Intuitively, the

gamma parameter determines how far the influence of a single training example rea‐

ches, with low values meaning corresponding to a far reach, and high values to a limi‐

ted reach. In other words, the wider the radius of the Gaussian kernel, the further the

influence of each training example. The C parameter is a regularization parameter,

similar to that used in the linear models. It limits the importance of each point (or

more precisely, their dual\_coef\_).

Let’s have a look at what happens when we vary these parameters (Figure 2-42):

In[80]:

fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):

for a, gamma in zip(ax, range(-1, 2)):

Supervised Machine Learning Algorithms

|

101mglearn.plots.plot\_svm(log\_C=C, log\_gamma=gamma, ax=a)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"],

ncol=4, loc=(.9, 1.2))

Figure 2-42. Decision boundaries and support vectors for different settings of the param‐

eters C and gamma

Going from left to right, we increase the value of the parameter gamma from 0.1 to 10.

A small gamma means a large radius for the Gaussian kernel, which means that many

points are considered close by. This is reflected in very smooth decision boundaries

on the left, and boundaries that focus more on single points further to the right. A

low value of gamma means that the decision boundary will vary slowly, which yields a

model of low complexity, while a high value of gamma yields a more complex model.

Going from top to bottom, we increase the C parameter from 0.1 to 1000. As with the

linear models, a small C means a very restricted model, where each data point can

only have very limited influence. You can see that at the top left the decision bound‐

ary looks nearly linear, with the misclassified points barely having any influence on

the line. Increasing C, as shown on the bottom left, allows these points to have a

stronger influence on the model and makes the decision boundary bend to correctly

classify them.

102

|

Chapter 2: Supervised LearningLet’s apply the RBF kernel SVM to the Breast Cancer dataset. By default, C=1 and

gamma=1/n\_features:

In[81]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

cancer.data, cancer.target, random\_state=0)

svc = SVC()

svc.fit(X\_train, y\_train)

print("Accuracy on training set: {:.2f}".format(svc.score(X\_train, y\_train)))

print("Accuracy on test set: {:.2f}".format(svc.score(X\_test, y\_test)))

Out[81]:

Accuracy on training set: 1.00

Accuracy on test set: 0.63

The model overfits quite substantially, with a perfect score on the training set and

only 63% accuracy on the test set. While SVMs often perform quite well, they are

very sensitive to the settings of the parameters and to the scaling of the data. In par‐

ticular, they require all the features to vary on a similar scale. Let’s look at the mini‐

mum and maximum values for each feature, plotted in log-space (Figure 2-43):

In[82]:

plt.boxplot(X\_train, manage\_xticks=False)

plt.yscale("symlog")

plt.xlabel("Feature index")

plt.ylabel("Feature magnitude")

From this plot we can determine that features in the Breast Cancer dataset are of

completely different orders of magnitude. This can be somewhat of a problem for

other models (like linear models), but it has devastating effects for the kernel SVM.

Let’s examine some ways to deal with this issue.

Supervised Machine Learning Algorithms

|

103Figure 2-43. Feature ranges for the Breast Cancer dataset (note that the y axis has a log‐

arithmic scale)

Preprocessing data for SVMs

One way to resolve this problem is by rescaling each feature so that they are all

approximately on the same scale. A common rescaling method for kernel SVMs is to

scale the data such that all features are between 0 and 1. We will see how to do this

using the MinMaxScaler preprocessing method in Chapter 3, where we’ll give more

details. For now, let’s do this “by hand”:

In[83]:

# compute the minimum value per feature on the training set

min\_on\_training = X\_train.min(axis=0)

# compute the range of each feature (max - min) on the training set

range\_on\_training = (X\_train - min\_on\_training).max(axis=0)

# subtract the min, and divide by range

# afterward, min=0 and max=1 for each feature

X\_train\_scaled = (X\_train - min\_on\_training) / range\_on\_training

print("Minimum for each feature\n{}".format(X\_train\_scaled.min(axis=0)))

print("Maximum for each feature\n {}".format(X\_train\_scaled.max(axis=0)))

104

|

Chapter 2: Supervised LearningOut[83]:

Minimum for each feature

[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.

0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Maximum for each feature

[ 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.

1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

In[84]:

# use THE SAME transformation on the test set,

# using min and range of the training set (see Chapter 3 for details)

X\_test\_scaled = (X\_test - min\_on\_training) / range\_on\_training

In[85]:

svc = SVC()

svc.fit(X\_train\_scaled, y\_train)

print("Accuracy on training set: {:.3f}".format(

svc.score(X\_train\_scaled, y\_train)))

print("Accuracy on test set: {:.3f}".format(svc.score(X\_test\_scaled, y\_test)))

Out[85]:

Accuracy on training set: 0.948

Accuracy on test set: 0.951

Scaling the data made a huge difference! Now we are actually in an underfitting

regime, where training and test set performance are quite similar but less close to

100% accuracy. From here, we can try increasing either C or gamma to fit a more com‐

plex model. For example:

In[86]:

svc = SVC(C=1000)

svc.fit(X\_train\_scaled, y\_train)

print("Accuracy on training set: {:.3f}".format(

svc.score(X\_train\_scaled, y\_train)))

print("Accuracy on test set: {:.3f}".format(svc.score(X\_test\_scaled, y\_test)))

Out[86]:

Accuracy on training set: 0.988

Accuracy on test set: 0.972

Here, increasing C allows us to improve the model significantly, resulting in 97.2%

accuracy.

Supervised Machine Learning Algorithms

|

105Strengths, weaknesses, and parameters

Kernelized support vector machines are powerful models and perform well on a vari‐

ety of datasets. SVMs allow for complex decision boundaries, even if the data has only

a few features. They work well on low-dimensional and high-dimensional data (i.e.,

few and many features), but don’t scale very well with the number of samples. Run‐

ning an SVM on data with up to 10,000 samples might work well, but working with

datasets of size 100,000 or more can become challenging in terms of runtime and

memory usage.

Another downside of SVMs is that they require careful preprocessing of the data and

tuning of the parameters. This is why, these days, most people instead use tree-based

models such as random forests or gradient boosting (which require little or no pre‐

processing) in many applications. Furthermore, SVM models are hard to inspect; it

can be difficult to understand why a particular prediction was made, and it might be

tricky to explain the model to a nonexpert.

Still, it might be worth trying SVMs, particularly if all of your features represent

measurements in similar units (e.g., all are pixel intensities) and they are on similar

scales.

The important parameters in kernel SVMs are the regularization parameter C, the

choice of the kernel, and the kernel-specific parameters. Although we primarily

focused on the RBF kernel, other choices are available in scikit-learn. The RBF

kernel has only one parameter, gamma, which is the inverse of the width of the Gaus‐

sian kernel. gamma and C both control the complexity of the model, with large values

in either resulting in a more complex model. Therefore, good settings for the two

parameters are usually strongly correlated, and C and gamma should be adjusted

together.

Neural Networks (Deep Learning)

A family of algorithms known as neural networks has recently seen a revival under

the name “deep learning.” While deep learning shows great promise in many machine

learning applications, deep learning algorithms are often tailored very carefully to a

specific use case. Here, we will only discuss some relatively simple methods, namely

multilayer perceptrons for classification and regression, that can serve as a starting

point for more involved deep learning methods. Multilayer perceptrons (MLPs) are

also known as (vanilla) feed-forward neural networks, or sometimes just neural

networks.

The neural network model

MLPs can be viewed as generalizations of linear models that perform multiple stages

of processing to come to a decision.

106

|

Chapter 2: Supervised LearningRemember that the prediction by a linear regressor is given as:

ŷ = w[0] \* x[0] + w[1] \* x[1] + ... + w[p] \* x[p] + b

In plain English, ŷ is a weighted sum of the input features x[0] to x[p], weighted by

the learned coefficients w[0] to w[p]. We could visualize this graphically as shown in

Figure 2-44:

In[87]:

display(mglearn.plots.plot\_logistic\_regression\_graph())

Figure 2-44. Visualization of logistic regression, where input features and predictions are

shown as nodes, and the coefficients are connections between the nodes

Here, each node on the left represents an input feature, the connecting lines represent

the learned coefficients, and the node on the right represents the output, which is a

weighted sum of the inputs.

In an MLP this process of computing weighted sums is repeated multiple times, first

computing hidden units that represent an intermediate processing step, which are

again combined using weighted sums to yield the final result (Figure 2-45):

In[88]:

display(mglearn.plots.plot\_single\_hidden\_layer\_graph())

Supervised Machine Learning Algorithms

|

107Figure 2-45. Illustration of a multilayer perceptron with a single hidden layer

This model has a lot more coefficients (also called weights) to learn: there is one

between every input and every hidden unit (which make up the hidden layer), and

one between every unit in the hidden layer and the output.

Computing a series of weighted sums is mathematically the same as computing just

one weighted sum, so to make this model truly more powerful than a linear model,

we need one extra trick. After computing a weighted sum for each hidden unit, a

nonlinear function is applied to the result—usually the rectifying nonlinearity (also

known as rectified linear unit or relu) or the tangens hyperbolicus (tanh). The result of

this function is then used in the weighted sum that computes the output, ŷ. The two

functions are visualized in Figure 2-46. The relu cuts off values below zero, while tanh

saturates to –1 for low input values and +1 for high input values. Either nonlinear

function allows the neural network to learn much more complicated functions than a

linear model could:

In[89]:

line = np.linspace(-3, 3, 100)

plt.plot(line, np.tanh(line), label="tanh")

plt.plot(line, np.maximum(line, 0), label="relu")

plt.legend(loc="best")

plt.xlabel("x")

plt.ylabel("relu(x), tanh(x)")

108

|

Chapter 2: Supervised LearningFigure 2-46. The hyperbolic tangent activation function and the rectified linear activa‐

tion function

For the small neural network pictured in Figure 2-45, the full formula for computing

ŷ in the case of regression would be (when using a tanh nonlinearity):

h[0] = tanh(w[0, 0] \* x[0] + w[1, 0] \* x[1] + w[2, 0] \* x[2] + w[3, 0] \* x[3] + b[0])

h[1] = tanh(w[0, 1] \* x[0] + w[1, 1] \* x[1] + w[2, 1] \* x[2] + w[3, 1] \* x[3] + b[1])

h[2] = tanh(w[0, 2] \* x[0] + w[1, 2] \* x[1] + w[2, 2] \* x[2] + w[3, 2] \* x[3] + b[2])

ŷ = v[0] \* h[0] + v[1] \* h[1] + v[2] \* h[2] + b

Here, w are the weights between the input x and the hidden layer h, and v are the

weights between the hidden layer h and the output ŷ. The weights v and w are learned

from data, x are the input features, ŷ is the computed output, and h are intermediate

computations. An important parameter that needs to be set by the user is the number

of nodes in the hidden layer. This can be as small as 10 for very small or simple data‐

sets and as big as 10,000 for very complex data. It is also possible to add additional

hidden layers, as shown in Figure 2-47:

Supervised Machine Learning Algorithms

|

109In[90]:

mglearn.plots.plot\_two\_hidden\_layer\_graph()

Figure 2-47. A multilayer perceptron with two hidden layers

Having large neural networks made up of many of these layers of computation is

what inspired the term “deep learning.”

Tuning neural networks

Let’s look into the workings of the MLP by applying the MLPClassifier to the

two\_moons dataset we used earlier in this chapter. The results are shown in

Figure 2-48:

In[91]:

from sklearn.neural\_network import MLPClassifier

from sklearn.datasets import make\_moons

X, y = make\_moons(n\_samples=100, noise=0.25, random\_state=3)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, stratify=y,

random\_state=42)

mlp = MLPClassifier(solver='lbfgs', random\_state=0).fit(X\_train, y\_train)

mglearn.plots.plot\_2d\_separator(mlp, X\_train, fill=True, alpha=.3)

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

110

|

Chapter 2: Supervised LearningFigure 2-48. Decision boundary learned by a neural network with 100 hidden units on

the two\_moons dataset

As you can see, the neural network learned a very nonlinear but relatively smooth

decision boundary. We used solver='lbfgs', which we will discuss later.

By default, the MLP uses 100 hidden nodes, which is quite a lot for this small dataset.

We can reduce the number (which reduces the complexity of the model) and still get

a good result (Figure 2-49):

In[92]:

mlp = MLPClassifier(solver='lbfgs', random\_state=0, hidden\_layer\_sizes=[10])

mlp.fit(X\_train, y\_train)

mglearn.plots.plot\_2d\_separator(mlp, X\_train, fill=True, alpha=.3)

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

Supervised Machine Learning Algorithms

|

111Figure 2-49. Decision boundary learned by a neural network with 10 hidden units on

the two\_moons dataset

With only 10 hidden units, the decision boundary looks somewhat more ragged. The

default nonlinearity is relu, shown in Figure 2-46. With a single hidden layer, this

means the decision function will be made up of 10 straight line segments. If we want

a smoother decision boundary, we could add more hidden units (as in Figure 2-48),

add a second hidden layer (Figure 2-50), or use the tanh nonlinearity (Figure 2-51):

In[93]:

# using two hidden layers, with 10 units each

mlp = MLPClassifier(solver='lbfgs', random\_state=0,

hidden\_layer\_sizes=[10, 10])

mlp.fit(X\_train, y\_train)

mglearn.plots.plot\_2d\_separator(mlp, X\_train, fill=True, alpha=.3)

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

112

|

Chapter 2: Supervised LearningIn[94]:

# using two hidden layers, with 10 units each, now with tanh nonlinearity

mlp = MLPClassifier(solver='lbfgs', activation='tanh',

random\_state=0, hidden\_layer\_sizes=[10, 10])

mlp.fit(X\_train, y\_train)

mglearn.plots.plot\_2d\_separator(mlp, X\_train, fill=True, alpha=.3)

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train)

plt.xlabel("Feature 0")

plt.ylabel("Feature 1")

Figure 2-50. Decision boundary learned using 2 hidden layers with 10 hidden units

each, with rect activation function

Supervised Machine Learning Algorithms

|

113Figure 2-51. Decision boundary learned using 2 hidden layers with 10 hidden units

each, with tanh activation function

Finally, we can also control the complexity of a neural network by using an l2 penalty

to shrink the weights toward zero, as we did in ridge regression and the linear classifi‐

ers. The parameter for this in the MLPClassifier is alpha (as in the linear regression

models), and it’s set to a very low value (little regularization) by default. Figure 2-52

shows the effect of different values of alpha on the two\_moons dataset, using two hid‐

den layers of 10 or 100 units each:

In[95]:

fig, axes = plt.subplots(2, 4, figsize=(20, 8))

for axx, n\_hidden\_nodes in zip(axes, [10, 100]):

for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):

mlp = MLPClassifier(solver='lbfgs', random\_state=0,

hidden\_layer\_sizes=[n\_hidden\_nodes, n\_hidden\_nodes],

alpha=alpha)

mlp.fit(X\_train, y\_train)

mglearn.plots.plot\_2d\_separator(mlp, X\_train, fill=True, alpha=.3, ax=ax)

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train, ax=ax)

ax.set\_title("n\_hidden=[{}, {}]\nalpha={:.4f}".format(

n\_hidden\_nodes, n\_hidden\_nodes, alpha))

114

|

Chapter 2: Supervised LearningFigure 2-52. Decision functions for different numbers of hidden units and different set‐

tings of the alpha parameter

As you probably have realized by now, there are many ways to control the complexity

of a neural network: the number of hidden layers, the number of units in each hidden

layer, and the regularization (alpha). There are actually even more, which we won’t

go into here.

An important property of neural networks is that their weights are set randomly

before learning is started, and this random initialization affects the model that is

learned. That means that even when using exactly the same parameters, we can

obtain very different models when using different random seeds. If the networks are

large, and their complexity is chosen properly, this should not affect accuracy too

much, but it is worth keeping in mind (particularly for smaller networks).

Figure 2-53 shows plots of several models, all learned with the same settings of the

parameters:

In[96]:

fig, axes = plt.subplots(2, 4, figsize=(20, 8))

for i, ax in enumerate(axes.ravel()):

mlp = MLPClassifier(solver='lbfgs', random\_state=i,

hidden\_layer\_sizes=[100, 100])

mlp.fit(X\_train, y\_train)

mglearn.plots.plot\_2d\_separator(mlp, X\_train, fill=True, alpha=.3, ax=ax)

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train, ax=ax)

Supervised Machine Learning Algorithms

|

115Figure 2-53. Decision functions learned with the same parameters but different random

initializations

To get a better understanding of neural networks on real-world data, let’s apply the

MLPClassifier to the Breast Cancer dataset. We start with the default parameters:

In[97]:

print("Cancer data per-feature maxima:\n{}".format(cancer.data.max(axis=0)))

Out[97]:

Cancer data per-feature maxima:

[

28.110

39.280

188.500 2501.000

0.201

0.304

0.097

2.873

0.031

0.135

0.396

0.053

49.540

251.200 4254.000

0.223

0.664

0.207]

0.163

4.885

0.079

1.058

0.345

21.980

0.030

1.252

0.427

542.200

36.040

0.291

In[98]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

cancer.data, cancer.target, random\_state=0)

mlp = MLPClassifier(random\_state=42)

mlp.fit(X\_train, y\_train)

print("Accuracy on training set: {:.2f}".format(mlp.score(X\_train, y\_train)))

print("Accuracy on test set: {:.2f}".format(mlp.score(X\_test, y\_test)))

Out[98]:

Accuracy on training set: 0.92

Accuracy on test set: 0.90

The accuracy of the MLP is quite good, but not as good as the other models. As in the

earlier SVC example, this is likely due to scaling of the data. Neural networks also

expect all input features to vary in a similar way, and ideally to have a mean of 0, and

116

|

Chapter 2: Supervised Learninga variance of 1. We must rescale our data so that it fulfills these requirements. Again,

we will do this by hand here, but we’ll introduce the StandardScaler to do this auto‐

matically in Chapter 3:

In[99]:

# compute the mean value per feature on the training set

mean\_on\_train = X\_train.mean(axis=0)

# compute the standard deviation of each feature on the training set

std\_on\_train = X\_train.std(axis=0)

# subtract the mean, and scale by inverse standard deviation

# afterward, mean=0 and std=1

X\_train\_scaled = (X\_train - mean\_on\_train) / std\_on\_train

# use THE SAME transformation (using training mean and std) on the test set

X\_test\_scaled = (X\_test - mean\_on\_train) / std\_on\_train

mlp = MLPClassifier(random\_state=0)

mlp.fit(X\_train\_scaled, y\_train)

print("Accuracy on training set: {:.3f}".format(

mlp.score(X\_train\_scaled, y\_train)))

print("Accuracy on test set: {:.3f}".format(mlp.score(X\_test\_scaled, y\_test)))

Out[99]:

Accuracy on training set: 0.991

Accuracy on test set: 0.965

ConvergenceWarning:

Stochastic Optimizer: Maximum iterations reached and the optimization

hasn't converged yet.

The results are much better after scaling, and already quite competitive. We got a

warning from the model, though, that tells us that the maximum number of iterations

has been reached. This is part of the adam algorithm for learning the model, and tells

us that we should increase the number of iterations:

In[100]:

mlp = MLPClassifier(max\_iter=1000, random\_state=0)

mlp.fit(X\_train\_scaled, y\_train)

print("Accuracy on training set: {:.3f}".format(

mlp.score(X\_train\_scaled, y\_train)))

print("Accuracy on test set: {:.3f}".format(mlp.score(X\_test\_scaled, y\_test)))

Out[100]:

Accuracy on training set: 0.995

Accuracy on test set: 0.965

Supervised Machine Learning Algorithms

|

117Increasing the number of iterations only increased the training set performance, not

the generalization performance. Still, the model is performing quite well. As there is

some gap between the training and the test performance, we might try to decrease the

model’s complexity to get better generalization performance. Here, we choose to

increase the alpha parameter (quite aggressively, from 0.0001 to 1) to add stronger

regularization of the weights:

In[101]:

mlp = MLPClassifier(max\_iter=1000, alpha=1, random\_state=0)

mlp.fit(X\_train\_scaled, y\_train)

print("Accuracy on training set: {:.3f}".format(

mlp.score(X\_train\_scaled, y\_train)))

print("Accuracy on test set: {:.3f}".format(mlp.score(X\_test\_scaled, y\_test)))

Out[101]:

Accuracy on training set: 0.988

Accuracy on test set: 0.972

This leads to a performance on par with the best models so far.12

While it is possible to analyze what a neural network has learned, this is usually much

trickier than analyzing a linear model or a tree-based model. One way to introspect

what was learned is to look at the weights in the model. You can see an example of

this in the scikit-learn example gallery. For the Breast Cancer dataset, this might

be a bit hard to understand. The following plot (Figure 2-54) shows the weights that

were learned connecting the input to the first hidden layer. The rows in this plot cor‐

respond to the 30 input features, while the columns correspond to the 100 hidden

units. Light colors represent large positive values, while dark colors represent nega‐

tive values:

In[102]:

plt.figure(figsize=(20, 5))

plt.imshow(mlp.coefs\_[0], interpolation='none', cmap='viridis')

plt.yticks(range(30), cancer.feature\_names)

plt.xlabel("Columns in weight matrix")

plt.ylabel("Input feature")

plt.colorbar()

12 You might have noticed at this point that many of the well-performing models achieved exactly the same

accuracy of 0.972. This means that all of the models make exactly the same number of mistakes, which is four.

If you compare the actual predictions, you can even see that they make exactly the same mistakes! This might

be a consequence of the dataset being very small, or it may be because these points are really different from

the rest.

118

|

Chapter 2: Supervised LearningFigure 2-54. Heat map of the first layer weights in a neural network learned on the

Breast Cancer dataset

One possible inference we can make is that features that have very small weights for

all of the hidden units are “less important” to the model. We can see that “mean

smoothness” and “mean compactness,” in addition to the features found between

“smoothness error” and “fractal dimension error,” have relatively low weights com‐

pared to other features. This could mean that these are less important features or pos‐

sibly that we didn’t represent them in a way that the neural network could use.

We could also visualize the weights connecting the hidden layer to the output layer,

but those are even harder to interpret.

While the MLPClassifier and MLPRegressor provide easy-to-use interfaces for the

most common neural network architectures, they only capture a small subset of what

is possible with neural networks. If you are interested in working with more flexible

or larger models, we encourage you to look beyond scikit-learn into the fantastic

deep learning libraries that are out there. For Python users, the most well-established

are keras, lasagna, and tensor-flow. lasagna builds on the theano library, while

keras can use either tensor-flow or theano. These libraries provide a much more

flexible interface to build neural networks and track the rapid progress in deep learn‐

ing research. All of the popular deep learning libraries also allow the use of high-

performance graphics processing units (GPUs), which scikit-learn does not

support. Using GPUs allows us to accelerate computations by factors of 10x to 100x,

and they are essential for applying deep learning methods to large-scale datasets.

Strengths, weaknesses, and parameters

Neural networks have reemerged as state-of-the-art models in many applications of

machine learning. One of their main advantages is that they are able to capture infor‐

mation contained in large amounts of data and build incredibly complex models.

Given enough computation time, data, and careful tuning of the parameters, neural

networks often beat other machine learning algorithms (for classification and regres‐

sion tasks).

Supervised Machine Learning Algorithms

|

119This brings us to the downsides. Neural networks—particularly the large and power‐

ful ones—often take a long time to train. They also require careful preprocessing of

the data, as we saw here. Similarly to SVMs, they work best with “homogeneous”

data, where all the features have similar meanings. For data that has very different

kinds of features, tree-based models might work better. Tuning neural network

parameters is also an art unto itself. In our experiments, we barely scratched the sur‐

face of possible ways to adjust neural network models and how to train them.

Estimating complexity in neural networks. The most important parameters are the num‐

ber of layers and the number of hidden units per layer. You should start with one or

two hidden layers, and possibly expand from there. The number of nodes per hidden

layer is often similar to the number of input features, but rarely higher than in the low

to mid-thousands.

A helpful measure when thinking about the model complexity of a neural network is

the number of weights or coefficients that are learned. If you have a binary classifica‐

tion dataset with 100 features, and you have 100 hidden units, then there are 100 \*

100 = 10,000 weights between the input and the first hidden layer. There are also

100 \* 1 = 100 weights between the hidden layer and the output layer, for a total of

around 10,100 weights. If you add a second hidden layer with 100 hidden units, there

will be another 100 \* 100 = 10,000 weights from the first hidden layer to the second

hidden layer, resulting in a total of 20,100 weights. If instead you use one layer with

1,000 hidden units, you are learning 100 \* 1,000 = 100,000 weights from the input to

the hidden layer and 1,000 \* 1 weights from the hidden layer to the output layer, for a

total of 101,000. If you add a second hidden layer you add 1,000 \* 1,000 = 1,000,000

weights, for a whopping total of 1,101,000—50 times larger than the model with two

hidden layers of size 100.

A common way to adjust parameters in a neural network is to first create a network

that is large enough to overfit, making sure that the task can actually be learned by

the network. Then, once you know the training data can be learned, either shrink the

network or increase alpha to add regularization, which will improve generalization

performance.

In our experiments, we focused mostly on the definition of the model: the number of

layers and nodes per layer, the regularization, and the nonlinearity. These define the

model we want to learn. There is also the question of how to learn the model, or the

algorithm that is used for learning the parameters, which is set using the algorithm

parameter. There are two easy-to-use choices for algorithm. The default is 'adam',

which works well in most situations but is quite sensitive to the scaling of the data (so

it is important to always scale your data to 0 mean and unit variance). The other one

is 'lbfgs', which is quite robust but might take a long time on larger models or

larger datasets. There is also the more advanced 'sgd' option, which is what many

deep learning researchers use. The 'sgd' option comes with many additional param‐

120

| Chapter 2: Supervised Learningeters that need to be tuned for best results. You can find all of these parameters and

their definitions in the user guide. When starting to work with MLPs, we recommend

sticking to 'adam' and 'lbfgs'.

fit Resets a Model

An important property of scikit-learn models is that calling fit

will always reset everything a model previously learned. So if you

build a model on one dataset, and then call fit again on a different

dataset, the model will “forget” everything it learned from the first

dataset. You can call fit as often as you like on a model, and the

outcome will be the same as calling fit on a “new” model.

Uncertainty Estimates from Classifiers

Another useful part of the scikit-learn interface that we haven’t talked about yet is

the ability of classifiers to provide uncertainty estimates of predictions. Often, you are

not only interested in which class a classifier predicts for a certain test point, but also

how certain it is that this is the right class. In practice, different kinds of mistakes lead

to very different outcomes in real-world applications. Imagine a medical application

testing for cancer. Making a false positive prediction might lead to a patient undergo‐

ing additional tests, while a false negative prediction might lead to a serious disease

not being treated. We will go into this topic in more detail in Chapter 6.

There are two different functions in scikit-learn that can be used to obtain uncer‐

tainty estimates from classifiers: decision\_function and predict\_proba. Most (but

not all) classifiers have at least one of them, and many classifiers have both. Let’s look

at what these two functions do on a synthetic two-dimensional dataset, when build‐

ing a GradientBoostingClassifier classifier, which has both a decision\_function

and a predict\_proba method:

In[103]:

from sklearn.ensemble import GradientBoostingClassifier

from sklearn.datasets import make\_circles

X, y = make\_circles(noise=0.25, factor=0.5, random\_state=1)

# we rename the classes "blue" and "red" for illustration purposes

y\_named = np.array(["blue", "red"])[y]

# we can call train\_test\_split with arbitrarily many arrays;

# all will be split in a consistent manner

X\_train, X\_test, y\_train\_named, y\_test\_named, y\_train, y\_test = \

train\_test\_split(X, y\_named, y, random\_state=0)

# build the gradient boosting model

gbrt = GradientBoostingClassifier(random\_state=0)

gbrt.fit(X\_train, y\_train\_named)

Uncertainty Estimates from Classifiers |

121The Decision Function

In the binary classification case, the return value of decision\_function is of shape

(n\_samples,), and it returns one floating-point number for each sample:

In[104]:

print("X\_test.shape: {}".format(X\_test.shape))

print("Decision function shape: {}".format(

gbrt.decision\_function(X\_test).shape))

Out[104]:

X\_test.shape: (25, 2)

Decision function shape: (25,)

This value encodes how strongly the model believes a data point to belong to the

“positive” class, in this case class 1. Positive values indicate a preference for the posi‐

tive class, and negative values indicate a preference for the “negative” (other) class:

In[105]:

# show the first few entries of decision\_function

print("Decision function:\n{}".format(gbrt.decision\_function(X\_test)[:6]))

Out[105]:

Decision function:

[ 4.136 -1.683 -3.951 -3.626

4.29

3.662]

We can recover the prediction by looking only at the sign of the decision function:

In[106]:

print("Thresholded decision function:\n{}".format(

gbrt.decision\_function(X\_test) > 0))

print("Predictions:\n{}".format(gbrt.predict(X\_test)))

Out[106]:

Thresholded decision function:

[ True False False False True True False True True True False True

True False True False False False True True True True True False

False]

Predictions:

['red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red' 'blue'

'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red' 'red'

'red' 'blue' 'blue']

For binary classification, the “negative” class is always the first entry of the classes\_

attribute, and the “positive” class is the second entry of classes\_. So if you want to

fully recover the output of predict, you need to make use of the classes\_ attribute:

122

|

Chapter 2: Supervised LearningIn[107]:

# make the boolean True/False into 0 and 1

greater\_zero = (gbrt.decision\_function(X\_test) > 0).astype(int)

# use 0 and 1 as indices into classes\_

pred = gbrt.classes\_[greater\_zero]

# pred is the same as the output of gbrt.predict

print("pred is equal to predictions: {}".format(

np.all(pred == gbrt.predict(X\_test))))

Out[107]:

pred is equal to predictions: True

The range of decision\_function can be arbitrary, and depends on the data and the

model parameters:

In[108]:

decision\_function = gbrt.decision\_function(X\_test)

print("Decision function minimum: {:.2f} maximum: {:.2f}".format(

np.min(decision\_function), np.max(decision\_function)))

Out[108]:

Decision function minimum: -7.69 maximum: 4.29

This arbitrary scaling makes the output of decision\_function often hard to

interpret.

In the following example we plot the decision\_function for all points in the 2D

plane using a color coding, next to a visualization of the decision boundary, as we saw

earlier. We show training points as circles and test data as triangles (Figure 2-55):

In[109]:

fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot\_2d\_separator(gbrt, X, ax=axes[0], alpha=.4,

fill=True, cm=mglearn.cm2)

scores\_image = mglearn.tools.plot\_2d\_scores(gbrt, X, ax=axes[1],

alpha=.4, cm=mglearn.ReBl)

for ax in axes:

# plot training and test points

mglearn.discrete\_scatter(X\_test[:, 0], X\_test[:, 1], y\_test,

markers='^', ax=ax)

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train,

markers='o', ax=ax)

ax.set\_xlabel("Feature 0")

ax.set\_ylabel("Feature 1")

cbar = plt.colorbar(scores\_image, ax=axes.tolist())

axes[0].legend(["Test class 0", "Test class 1", "Train class 0",

"Train class 1"], ncol=4, loc=(.1, 1.1))

Uncertainty Estimates from Classifiers |

123Figure 2-55. Decision boundary (left) and decision function (right) for a gradient boost‐

ing model on a two-dimensional toy dataset

Encoding not only the predicted outcome but also how certain the classifier is pro‐

vides additional information. However, in this visualization, it is hard to make out the

boundary between the two classes.

Predicting Probabilities

The output of predict\_proba is a probability for each class, and is often more easily

understood than the output of decision\_function. It is always of shape (n\_samples,

2) for binary classification:

In[110]:

print("Shape of probabilities: {}".format(gbrt.predict\_proba(X\_test).shape))

Out[110]:

Shape of probabilities: (25, 2)

The first entry in each row is the estimated probability of the first class, and the sec‐

ond entry is the estimated probability of the second class. Because it is a probability,

the output of predict\_proba is always between 0 and 1, and the sum of the entries

for both classes is always 1:

In[111]:

# show the first few entries of predict\_proba

print("Predicted probabilities:\n{}".format(

gbrt.predict\_proba(X\_test[:6])))

124

|

Chapter 2: Supervised LearningOut[111]:

Predicted probabilities:

[[ 0.016 0.984]

[ 0.843 0.157]

[ 0.981 0.019]

[ 0.974 0.026]

[ 0.014 0.986]

[ 0.025 0.975]]

Because the probabilities for the two classes sum to 1, exactly one of the classes will

be above 50% certainty. That class is the one that is predicted.13

You can see in the previous output that the classifier is relatively certain for most

points. How well the uncertainty actually reflects uncertainty in the data depends on

the model and the parameters. A model that is more overfitted tends to make more

certain predictions, even if they might be wrong. A model with less complexity usu‐

ally has more uncertainty in its predictions. A model is called calibrated if the

reported uncertainty actually matches how correct it is—in a calibrated model, a pre‐

diction made with 70% certainty would be correct 70% of the time.

In the following example (Figure 2-56) we again show the decision boundary on the

dataset, next to the class probabilities for the class 1:

In[112]:

fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot\_2d\_separator(

gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)

scores\_image = mglearn.tools.plot\_2d\_scores(

gbrt, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl, function='predict\_proba')

for ax in axes:

# plot training and test points

mglearn.discrete\_scatter(X\_test[:, 0], X\_test[:, 1], y\_test,

markers='^', ax=ax)

mglearn.discrete\_scatter(X\_train[:, 0], X\_train[:, 1], y\_train,

markers='o', ax=ax)

ax.set\_xlabel("Feature 0")

ax.set\_ylabel("Feature 1")

cbar = plt.colorbar(scores\_image, ax=axes.tolist())

axes[0].legend(["Test class 0", "Test class 1", "Train class 0",

"Train class 1"], ncol=4, loc=(.1, 1.1))

13 Because the probabilities are floating-point numbers, it is unlikely that they will both be exactly 0.500. How‐

ever, if that happens, the prediction is made at random.

Uncertainty Estimates from Classifiers |

125Figure 2-56. Decision boundary (left) and predicted probabilities for the gradient boost‐

ing model shown in Figure 2-55

The boundaries in this plot are much more well-defined, and the small areas of

uncertainty are clearly visible.

The scikit-learn website has a great comparison of many models and what their

uncertainty estimates look like. We’ve reproduced this in Figure 2-57, and we encour‐

age you to go though the example there.

Figure 2-57. Comparison of several classifiers in scikit-learn on synthetic datasets (image

courtesy http://scikit-learn.org)

Uncertainty in Multiclass Classification

So far, we’ve only talked about uncertainty estimates in binary classification. But the

decision\_function and predict\_proba methods also work in the multiclass setting.

Let’s apply them on the Iris dataset, which is a three-class classification dataset:

126

|

Chapter 2: Supervised LearningIn[113]:

from sklearn.datasets import load\_iris

iris = load\_iris()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

iris.data, iris.target, random\_state=42)

gbrt = GradientBoostingClassifier(learning\_rate=0.01, random\_state=0)

gbrt.fit(X\_train, y\_train)

In[114]:

print("Decision function shape: {}".format(gbrt.decision\_function(X\_test).shape))

# plot the first few entries of the decision function

print("Decision function:\n{}".format(gbrt.decision\_function(X\_test)[:6, :]))

Out[114]:

Decision function shape: (38, 3)

Decision function:

[[-0.529 1.466 -0.504]

[ 1.512 -0.496 -0.503]

[-0.524 -0.468 1.52 ]

[-0.529 1.466 -0.504]

[-0.531 1.282 0.215]

[ 1.512 -0.496 -0.503]]

In the multiclass case, the decision\_function has the shape (n\_samples,

n\_classes) and each column provides a “certainty score” for each class, where a large

score means that a class is more likely and a small score means the class is less likely.

You can recover the predictions from these scores by finding the maximum entry for

each data point:

In[115]:

print("Argmax of decision function:\n{}".format(

np.argmax(gbrt.decision\_function(X\_test), axis=1)))

print("Predictions:\n{}".format(gbrt.predict(X\_test)))

Out[115]:

Argmax of decision function:

[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]

Predictions:

[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]

The output of predict\_proba has the same shape, (n\_samples, n\_classes). Again,

the probabilities for the possible classes for each data point sum to 1:

Uncertainty Estimates from Classifiers |

127In[116]:

# show the first few entries of predict\_proba

print("Predicted probabilities:\n{}".format(gbrt.predict\_proba(X\_test)[:6]))

# show that sums across rows are one

print("Sums: {}".format(gbrt.predict\_proba(X\_test)[:6].sum(axis=1)))

Out[116]:

Predicted probabilities:

[[ 0.107 0.784 0.109]

[ 0.789 0.106 0.105]

[ 0.102 0.108 0.789]

[ 0.107 0.784 0.109]

[ 0.108 0.663 0.228]

[ 0.789 0.106 0.105]]

Sums: [ 1. 1. 1. 1. 1.

1.]

We can again recover the predictions by computing the argmax of predict\_proba:

In[117]:

print("Argmax of predicted probabilities:\n{}".format(

np.argmax(gbrt.predict\_proba(X\_test), axis=1)))

print("Predictions:\n{}".format(gbrt.predict(X\_test)))

Out[117]:

Argmax of predicted probabilities:

[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]

Predictions:

[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1 0]

To summarize, predict\_proba and decision\_function always have shape (n\_sam

ples, n\_classes)—apart from decision\_function in the special binary case. In the

binary case, decision\_function only has one column, corresponding to the “posi‐

tive” class classes\_[1]. This is mostly for historical reasons.

You can recover the prediction when there are n\_classes many columns by comput‐

ing the argmax across columns. Be careful, though, if your classes are strings, or you

use integers but they are not consecutive and starting from 0. If you want to compare

results obtained with predict to results obtained via decision\_function or pre

dict\_proba, make sure to use the classes\_ attribute of the classifier to get the actual

class names:

128

|

Chapter 2: Supervised LearningIn[118]:

logreg = LogisticRegression()

# represent each target by its class name in the iris dataset

named\_target = iris.target\_names[y\_train]

logreg.fit(X\_train, named\_target)

print("unique classes in training data: {}".format(logreg.classes\_))

print("predictions: {}".format(logreg.predict(X\_test)[:10]))

argmax\_dec\_func = np.argmax(logreg.decision\_function(X\_test), axis=1)

print("argmax of decision function: {}".format(argmax\_dec\_func[:10]))

print("argmax combined with classes\_: {}".format(

logreg.classes\_[argmax\_dec\_func][:10]))

Out[118]:

unique classes in training data: ['setosa' 'versicolor' 'virginica']

predictions: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor'

'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']

argmax of decision function: [1 0 2 1 1 0 1 2 1 1]

argmax combined with classes\_: ['versicolor' 'setosa' 'virginica' 'versicolor'

'versicolor' 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']

Summary and Outlook

We started this chapter with a discussion of model complexity, then discussed gener‐

alization, or learning a model that is able to perform well on new, previously unseen

data. This led us to the concepts of underfitting, which describes a model that cannot

capture the variations present in the training data, and overfitting, which describes a

model that focuses too much on the training data and is not able to generalize to new

data very well.

We then discussed a wide array of machine learning models for classification and

regression, what their advantages and disadvantages are, and how to control model

complexity for each of them. We saw that for many of the algorithms, setting the right

parameters is important for good performance. Some of the algorithms are also sensi‐

tive to how we represent the input data, and in particular to how the features are

scaled. Therefore, blindly applying an algorithm to a dataset without understanding

the assumptions the model makes and the meanings of the parameter settings will

rarely lead to an accurate model.

This chapter contains a lot of information about the algorithms, and it is not neces‐

sary for you to remember all of these details for the following chapters. However,

some knowledge of the models described here—and which to use in a specific situa‐

tion—is important for successfully applying machine learning in practice. Here is a

quick summary of when to use each model:

Summary and Outlook

|

129Nearest neighbors

For small datasets, good as a baseline, easy to explain.

Linear models

Go-to as a first algorithm to try, good for very large datasets, good for very high-

dimensional data.

Naive Bayes

Only for classification. Even faster than linear models, good for very large data‐

sets and high-dimensional data. Often less accurate than linear models.

Decision trees

Very fast, don’t need scaling of the data, can be visualized and easily explained.

Random forests

Nearly always perform better than a single decision tree, very robust and power‐

ful. Don’t need scaling of data. Not good for very high-dimensional sparse data.

Gradient boosted decision trees

Often slightly more accurate than random forests. Slower to train but faster to

predict than random forests, and smaller in memory. Need more parameter tun‐

ing than random forests.

Support vector machines

Powerful for medium-sized datasets of features with similar meaning. Require

scaling of data, sensitive to parameters.

Neural networks

Can build very complex models, particularly for large datasets. Sensitive to scal‐

ing of the data and to the choice of parameters. Large models need a long time to

train.

When working with a new dataset, it is in general a good idea to start with a simple

model, such as a linear model or a naive Bayes or nearest neighbors classifier, and see

how far you can get. After understanding more about the data, you can consider

moving to an algorithm that can build more complex models, such as random forests,

gradient boosted decision trees, SVMs, or neural networks.

You should now be in a position where you have some idea of how to apply, tune, and

analyze the models we discussed here. In this chapter, we focused on the binary clas‐

sification case, as this is usually easiest to understand. Most of the algorithms presen‐

ted have classification and regression variants, however, and all of the classification

algorithms support both binary and multiclass classification. Try applying any of

these algorithms to the built-in datasets in scikit-learn, like the boston\_housing or

diabetes datasets for regression, or the digits dataset for multiclass classification.

Playing around with the algorithms on different datasets will give you a better feel for

130

|

Chapter 2: Supervised Learninghow long they need to train, how easy it is to analyze the models, and how sensitive

they are to the representation of the data.

While we analyzed the consequences of different parameter settings for the algo‐

rithms we investigated, building a model that actually generalizes well to new data in

production is a bit trickier than that. We will see how to properly adjust parameters

and how to find good parameters automatically in Chapter 6.

First, though, we will dive in more detail into unsupervised learning and preprocess‐

ing in the next chapter.

Summary and Outlook

|

131CHAPTER 3

Unsupervised Learning and Preprocessing

The second family of machine learning algorithms that we will