Machine Learning Methods

Yaning Liu

2024-08-01

Table of contents

Preface

This is a book based on the classnotes of MATH 4833/5833

1 Support Vector Machines

Support vector machines (SVM), developed by Cortes and Vapnik (Ref. 1), is a powerful machine learning tool that can be applied to both classification and regression problems. When applied as a classifier, it is called a *support vector classifier*, and a *support vector regressor* when dealing with regression tasks. In this chapter, we will explore the theoretical concepts, mathematical principles, and practical applications of SVM. We start with solving classification problems with SVM.

1.1 Linear Support Vector Machines for Classification

Consider a binary classification problem, where the data $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$, $y_i \in \{-1, 1\}$, are linearly separable. Our goal is to find a linear line (or a plane in 3D and a hyperplane in higher dimensions) that separates the data points. There can be infinitely many such choices, as shown in the figure below. In this example, the blue circles and red squares representing two different classes, with two features x_1 and x_2. Three lines, l_1, l_2 and l_3, are plotted to show possible ways of separating the data linearly.

We need some criterion to decide the "best" separating line (or plane in higher dimensions), called **the decision boundary**. Intuitively, we want one that lies far from both datasets, so it has the best tolerance for perturbations/noises in the training data. Such a linear decision boundary can be described as

$$s(x) = w^T x + b = 0$$

which is exactly the same form as those used for linear regression and logistic regression. Note that we separated the intercept b from the other coefficients w for an easier discussion of SVM. Assume a feature vector $x = (x_1, x_2, \dots, x_d)^T$ is d-dimensional and the training dataset comprises N instances $x_1, x_2, \dots, x_N$.

Suppose the two classes are **linearly separable**, i.e., they can be clearly separated by the linear decision boundary. The class of points with label $+1$ satisfies $s(x) > 0$ and the other class satisfies $s(x) < 0$. Predictions are made for new instances in the same fashion. Let $\widehat{y}(x)$ represent the predicted class for a new instance with features x. We have

$$\widehat{y}(x) = \begin{cases} -1 & \text{if } s(x) = w^T x + b < 0 \\ 1 & \text{if } s(x) = w^T x + b > 0 \end{cases} \tag{1.1}$$

1.2 Margin

As we saw in the figure above, there can be infinitely many ways to define a linear decision boundary. A natural way to select the "optimal" boundary is to find the boundary so that it is as far as possible from both classes of data points. Such a strategy can be described with the help of the concept of **margin**, which is defined as the minimum distance between the boundary and any data point. In the figure below, the margin m, corresponding to the distance between boundary l and the data instance closest to l (the red square in the upper right corner), was plotted. An "optimal" boundary can then be defined as the one that has the maximum margin and such a classifier is called a **maximum margin classifier**. For example, l^* represents the maximum margin classifier, and m^* represents the maximum margin. Notice that data instances of the two classes that are closest to the boundary l^* are equidistant from it.

We now mathematically describe the maximum margin. It can be shown that the distance between an arbitrary point x in the feature space and the decision boundary $s(x) = w^T x + b = 0$ is

$$\frac{|s(x)|}{||w||_2}$$

(see Excercise 5-1). Noting that $|s(x)| = y(x)s(x)$, the distance of x_i to the decision boundary can be rewritten as:

$$\frac{y_i s(x_i)}{||w||_2} = \frac{y_i(w^T x_i + b)}{||w||_2} \tag{1.2}$$

Margin, defined as the smallest distance between the decision boundary and any data point x_i, can be mathematically written as

$$\text{margin} = \min_{1 \le i \le N} \frac{y_i(w^T x_i + b)}{||w||_2} = \frac{1}{||w||_2} \min_{1 \le i \le N} y_i(w^T x_i + b).$$

Therefore, the maximum margin is

$$\max_{w,b} \text{margin} = \max_{w,b} \left\{ \frac{1}{||w||_2} \min_{1 \le i \le N} y_i(w^T x_i + b) \right\}$$

and our maximum margin classifier is found by computing

$$\arg\max_{w,b} \left\{ \frac{1}{||w||_2} \min_{1 \le i \le N} y_i(w^T x_i + b) \right\}$$

An important fact related to the distance of x_i to the decision boundary is that it is invariant to the linear transformation (rescaling) $w \to \alpha w$, and $b \to \alpha b$ (but it does change the value of $y_i(w^T x_i + b)$; see Excercise 5-2). As a result, we can choose the α such that

$$y_i(w^T x_i + b) = 1 \tag{1.3}$$

for the points that are closest to the decision boundary (these points are also called **support vectors**, and hence the name support vector machines), as shown in the figure below:

Then the maximum margin problem can be written as

$$\arg\max_{w,b} \frac{1}{||w||_2} = \arg\min_{w,b} ||w||_2^2 = \arg\min_{w,b} \frac{1}{2}||w||_2^2$$

where the last step is for the sake of simpler computation only. Now, the maximum margin criterion is to solve the following optimization problem:

$$\begin{aligned} &\arg\min_{w,b} \frac{1}{2}||w||_2^2 \\ &\text{subject to } y_i(w^T x_i + b) \geq 1, \quad 1 \leq i \leq N \end{aligned}$$

1.3 Dual Problem

The problem above is a **quadratic programming problem**, which (called the **primal problem**) can be converted to the **dual problem** by using the method of *Lagrange multipliers*. Using Lagrange multipliers $\alpha_i \geq 0$ for each of the restrains, the Lagrangian function can be written as

$$L(w, b, \alpha) = \frac{1}{2}||w||_2^2 + \sum_{i=1}^{N} \alpha_i (1 - y_i(w^T x_i + b))$$

where $\alpha = (\alpha_1, \ldots, \alpha_N)$. Taking derivative with respect to w and b, and set them to 0. After substitution and organization (see Excercise 5-3), we have the dual problem:

$$\begin{aligned} &\arg\max_{\alpha} L(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j x_i^T x_j \\ &\text{subject to } \sum_{i=1}^{N} \alpha_i y_i = 0, \quad \alpha_i \geq 0, \;\; 1 \leq i \leq N \end{aligned}$$

After the dual problem is solved (numerically), for example, by the Sequential Minimal Optimization (SMO) algorithm (see Platt 1998), predictions for new data instances can be made by evaluating

$$s(x) = w^T x + b = \sum_{i=1}^{N} \alpha_i y_i x_i^T x + b$$

We have not talked about how to compute b yet. Noting that for any support vector x_s, the following equation is satisfied:

$$y_i s(x_s) = y_i \left(\sum_{i=1}^{N} \alpha_i y_i x_i^T x_s + b \right) = 1$$

and b can be obtained from any of these equations. A more numerically robust way to calculate b is to take the average of the b's solved for from all these equations (see Excercise 5-4):

$$b = \frac{1}{N_{\mathcal{S}}} \sum_{i \in \mathcal{S}} \left(y_i - \sum_{j=1}^{N} \alpha_j y_j x_j^T x_i \right)$$

where $\mathcal{S}$ denotes the set of indices of the support vectors, and $N_{\mathcal{S}}$ is the total number of support vectors.

Excercise 5-1

Show that the distance between an arbitrary point x in the feature space and the decision boundary $s(x) = w^T x + b = 0$ is

$$\frac{|s(x)|}{||w||_2}$$

Excercise 5-2

Show that the distance of x to the decision boundary is invariant to the rescaling $w \to \alpha w$, and $b \to \alpha b$.

Excercise 5-3

Derive the dual problem from the primal problem for the maximum margin problem.

Excercise 5-4

Show the value of b can be computed by

$$b = \frac{1}{N_{\mathcal{S}}} \sum_{i \in \mathcal{S}} \left(y_i - \sum_{j=1}^{N} \alpha_j y_j x_j^T x_i \right)$$

1.4 Kernel Functions

A binary classification problem may not be linearly separable. In such cases, there may exist maps such that the original feature space is projected to a higher dimension and the data points become linearly separable in that higher-dimensional space. Let such a map be $\phi(x)$. All of the above formulas ($s(x)$, primal problem, and dual problem) need to be changed so that x, x_i, and x_j will be replaced by $\phi(x)$, $\phi(x_i)$, and $\phi(x_j)$, respectively. The problem is computing inner products such as $\phi(x_i)^T \phi(x_j)$ can be very expensive, especially considering that the dimension of the new feature space can be high, and even infinite. To avoid the direct calculation of the inner product, a trick is to imagine a function $k(x, y)$ exists and

$$k(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

That is, there is a function that calculates the inner product of mapped features $\phi(x)$ in the higher-dimensional space by evaluating a function defined in the original (low-dimensional) feature space. Such functions $k(\cdot, \cdot)$ are called **kernel functions**. With the kernel function, the dual problem can be rewritten as

$$\arg\max_{\alpha} L(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j k(x_i, x_j)$$

$$\text{subject to } \sum_{i=1}^{N} \alpha_i y_i = 0, \quad \alpha_i \geq 0, \;\; 1 \leq i \leq N$$

and the linear function $s(x)$ becomes

$$s(x) = \sum_{i=1}^{N} \alpha_i y_i k(x_i, x) + b$$

Since, we are not sure what $\phi(\cdot)$ looks like, and thus finding the kernel function corresponding to $\phi(\cdot)$ is not possible. As a result, we often turn to some commonly used kernel functions as shown below, and in fact, each of them implicitly defines a map $\phi(\cdot)$.

$$\begin{aligned}
\text{Linear: } & k(x, y) = x^T y \\
\text{Polynomial: } & k(x, y) = \left(\gamma x^T y + r\right)^d \\
\text{Gaussian RBF: } & k(x, y) = \exp\left(-\gamma \|x - y\|_2^2\right) \\
\text{Sigmoid: } & k(x, y) = \tanh\left(\gamma x^T y + r\right) \\
\text{Laplacian: } & k(x, y) = \exp\left(-\gamma \|x - y\|_1\right)
\end{aligned}$$

These standard kernel functions can be combined to form new kernel functions. For instance, a linear combination of kernel functions is still a kernel function.

1.5 Soft Margin

Even if the data points are linearly separable (in the original feature space or with a kernel function), it may still not be a good idea to use such a decision boundary, because of overfitting and poor generalization. To alleviate this problem, we may consider a more flexible model that allows data points to appear on the wrong side of the decision boundary ($y_i s(x_i) < 0$), while keeping the margin as large as possible. As the figure shown below, there are two instances of the blue circle class (target $+1$) on the wrong side, and one instance of the red square class (target -1). All of them are colored slightly differently in the figure. Data points can also be inside the margin boundary, although it is on the right side of the decision boundary (there is one such point in the red square class in the figure). Such a method is called **soft margin classification**.

A possible cost function to achieve a balance of maximum margin and limiting cases of being on the wrong side and inside the margin boundary (characterized by $y_i(w^T x_i + b) < 1$) is:

$$\arg\min_{w,b} \frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N}(1 - y_i(w^T x_i + b))^+$$

where $C > 0$ is a constant that controls the penalty to be applied for the instances that result in violations, and $(\cdot)^+$ is the *Heaviside step function* with $(0)^+ = 0$. However, it is nonconvex and not continuous, leading to difficulty in solving the optimization problem. Other functions can be used to replace the Heaviside step function. One example is the **hinge loss function**:

$$L_{\text{hinge}}(x) = \max(0, 1 - x)$$

```
# Plot of a hinge function
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

x = np.linspace(-2, 4, 100)
y = np.zeros(100)
for i in range(100):
    y[i] = max(0, 1-x[i])
plt.plot(x, y, 'b-')
plt.xlabel('x')
plt.title('Hinge Loss');
```

With the hinge loss, the soft margin objective function can be written as:

$$\begin{aligned}&\arg\min_{w,b}\frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N} L_{\text{hinge}}(y_i(w^T x_i + b))\\ &= \arg\min_{w,b}\frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N}\max(0, 1 - y_i(w^T x_i + b))\end{aligned}$$

The hinge loss function penalizes more those points farther away from the boundary. Introduce **slack variables** $\xi_i \geq 0$, $1 \leq i \leq N$, and $\xi_i = L_{\text{hinge}}(y_i(w^T x_i + b)) = |y_i - s(x_i)|$. The objective function can be rewritten as

$$\begin{aligned}&\arg\min_{w,b,\varepsilon_i}\frac{1}{2}||w||_2^2 + C\sum_{i=1}^{N}\varepsilon_i\\ &\text{subject to } y_i(w^T x_i + b) \geq 1 - \varepsilon_i, \quad \varepsilon_i \geq 0,\ i = 1, \ldots, N\end{aligned}$$

This is still a quadratic problem and similarly a dual problem can be obtained using Lagrange multiplier.

Excercise 5-5

Show the constraints $y_i(w^T x_i + b) \geq 1 - \varepsilon_i$, $i = 1, \ldots, N$ need to hold for the soft margin classification problem.

Example 5-1

Build a binary SVM classifier for the Iris Dataset to classify virginica and non-virginica. Consider two cases: 1) using sepal length and petal width as features; 2) using petal length and petal width as features.

```
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

# Load the data
iris = datasets.load_iris()
# Show what the iris dataset look like
print(iris)
# Use sepal length and petal width as features
X = iris["data"][:, [0, 3]]  # Sepal length and petal width
# 1 if Iris-Virginica, else 0
y = (iris["target"] == 2).astype('int')
# Build linear svm classifier
# Scaling is important for SVM.
# Build a Pipeline that first scales data and then trains the model
# For hard margin, set C=0.0
svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1.0, loss="hinge", dual="auto", random_state=36)),
    ])
# Train the model
svm_clf.fit(X, y);
```

```
{'data': array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 3.4, 1.5, 0.2],
       [4.4, 2.9, 1.4, 0.2],
       [4.9, 3.1, 1.5, 0.1],
       [5.4, 3.7, 1.5, 0.2],
       [4.8, 3.4, 1.6, 0.2],
       [4.8, 3. , 1.4, 0.1],
       [4.3, 3. , 1.1, 0.1],
```

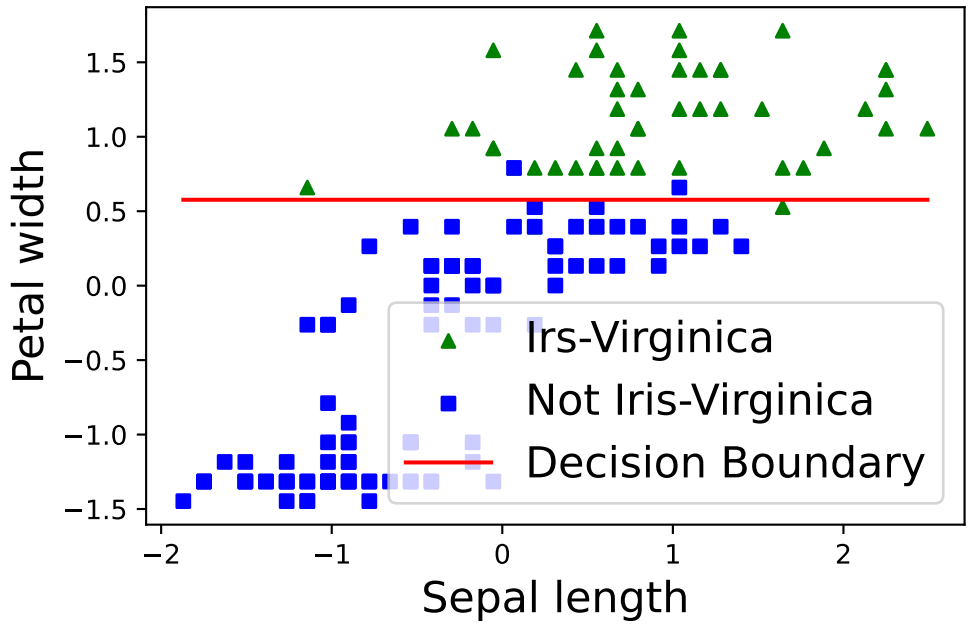
```
[5.8, 4. , 1.2, 0.2],
[5.7, 4.4, 1.5, 0.4],
[5.4, 3.9, 1.3, 0.4],
[5.1, 3.5, 1.4, 0.3],
[5.7, 3.8, 1.7, 0.3],
[5.1, 3.8, 1.5, 0.3],
[5.4, 3.4, 1.7, 0.2],
[5.1, 3.7, 1.5, 0.4],
[4.6, 3.6, 1. , 0.2],
[5.1, 3.3, 1.7, 0.5],
[4.8, 3.4, 1.9, 0.2],
[5. , 3. , 1.6, 0.2],
[5. , 3.4, 1.6, 0.4],
[5.2, 3.5, 1.5, 0.2],
[5.2, 3.4, 1.4, 0.2],
[4.7, 3.2, 1.6, 0.2],
[4.8, 3.1, 1.6, 0.2],
[5.4, 3.4, 1.5, 0.4],
[5.2, 4.1, 1.5, 0.1],
[5.5, 4.2, 1.4, 0.2],
[4.9, 3.1, 1.5, 0.2],
[5. , 3.2, 1.2, 0.2],
[5.5, 3.5, 1.3, 0.2],
[4.9, 3.6, 1.4, 0.1],
[4.4, 3. , 1.3, 0.2],
[5.1, 3.4, 1.5, 0.2],
[5. , 3.5, 1.3, 0.3],
[4.5, 2.3, 1.3, 0.3],
[4.4, 3.2, 1.3, 0.2],
[5. , 3.5, 1.6, 0.6],
[5.1, 3.8, 1.9, 0.4],
[4.8, 3. , 1.4, 0.3],
[5.1, 3.8, 1.6, 0.2],
[4.6, 3.2, 1.4, 0.2],
[5.3, 3.7, 1.5, 0.2],
[5. , 3.3, 1.4, 0.2],
[7. , 3.2, 4.7, 1.4],
[6.4, 3.2, 4.5, 1.5],
[6.9, 3.1, 4.9, 1.5],
[5.5, 2.3, 4. , 1.3],
[6.5, 2.8, 4.6, 1.5],
[5.7, 2.8, 4.5, 1.3],
[6.3, 3.3, 4.7, 1.6],
```

```
[4.9, 2.4, 3.3, 1. ],
[6.6, 2.9, 4.6, 1.3],
[5.2, 2.7, 3.9, 1.4],
[5. , 2. , 3.5, 1. ],
[5.9, 3. , 4.2, 1.5],
[6. , 2.2, 4. , 1. ],
[6.1, 2.9, 4.7, 1.4],
[5.6, 2.9, 3.6, 1.3],
[6.7, 3.1, 4.4, 1.4],
[5.6, 3. , 4.5, 1.5],
[5.8, 2.7, 4.1, 1. ],
[6.2, 2.2, 4.5, 1.5],
[5.6, 2.5, 3.9, 1.1],
[5.9, 3.2, 4.8, 1.8],
[6.1, 2.8, 4. , 1.3],
[6.3, 2.5, 4.9, 1.5],
[6.1, 2.8, 4.7, 1.2],
[6.4, 2.9, 4.3, 1.3],
[6.6, 3. , 4.4, 1.4],
[6.8, 2.8, 4.8, 1.4],
[6.7, 3. , 5. , 1.7],
[6. , 2.9, 4.5, 1.5],
[5.7, 2.6, 3.5, 1. ],
[5.5, 2.4, 3.8, 1.1],
[5.5, 2.4, 3.7, 1. ],
[5.8, 2.7, 3.9, 1.2],
[6. , 2.7, 5.1, 1.6],
[5.4, 3. , 4.5, 1.5],
[6. , 3.4, 4.5, 1.6],
[6.7, 3.1, 4.7, 1.5],
[6.3, 2.3, 4.4, 1.3],
[5.6, 3. , 4.1, 1.3],
[5.5, 2.5, 4. , 1.3],
[5.5, 2.6, 4.4, 1.2],
[6.1, 3. , 4.6, 1.4],
[5.8, 2.6, 4. , 1.2],
[5. , 2.3, 3.3, 1. ],
[5.6, 2.7, 4.2, 1.3],
[5.7, 3. , 4.2, 1.2],
[5.7, 2.9, 4.2, 1.3],
[6.2, 2.9, 4.3, 1.3],
[5.1, 2.5, 3. , 1.1],
[5.7, 2.8, 4.1, 1.3],
```

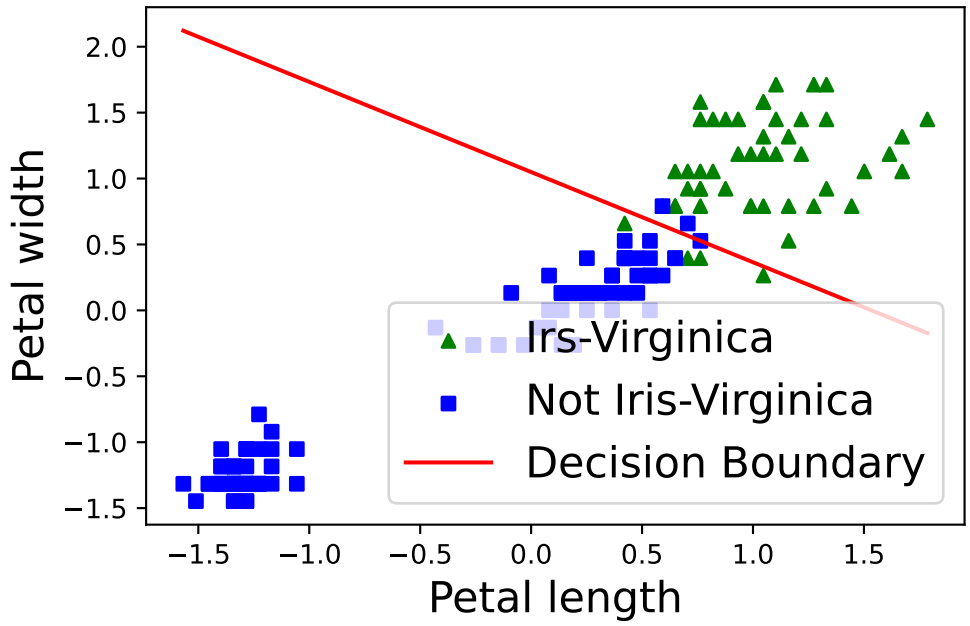
```
[6.3, 3.3, 6. , 2.5],
[5.8, 2.7, 5.1, 1.9],
[7.1, 3. , 5.9, 2.1],
[6.3, 2.9, 5.6, 1.8],
[6.5, 3. , 5.8, 2.2],
[7.6, 3. , 6.6, 2.1],
[4.9, 2.5, 4.5, 1.7],
[7.3, 2.9, 6.3, 1.8],
[6.7, 2.5, 5.8, 1.8],
[7.2, 3.6, 6.1, 2.5],
[6.5, 3.2, 5.1, 2. ],
[6.4, 2.7, 5.3, 1.9],
[6.8, 3. , 5.5, 2.1],
[5.7, 2.5, 5. , 2. ],
[5.8, 2.8, 5.1, 2.4],
[6.4, 3.2, 5.3, 2.3],
[6.5, 3. , 5.5, 1.8],
[7.7, 3.8, 6.7, 2.2],
[7.7, 2.6, 6.9, 2.3],
[6. , 2.2, 5. , 1.5],
[6.9, 3.2, 5.7, 2.3],
[5.6, 2.8, 4.9, 2. ],
[7.7, 2.8, 6.7, 2. ],
[6.3, 2.7, 4.9, 1.8],
[6.7, 3.3, 5.7, 2.1],
[7.2, 3.2, 6. , 1.8],
[6.2, 2.8, 4.8, 1.8],
[6.1, 3. , 4.9, 1.8],
[6.4, 2.8, 5.6, 2.1],
[7.2, 3. , 5.8, 1.6],
[7.4, 2.8, 6.1, 1.9],
[7.9, 3.8, 6.4, 2. ],
[6.4, 2.8, 5.6, 2.2],
[6.3, 2.8, 5.1, 1.5],
[6.1, 2.6, 5.6, 1.4],
[7.7, 3. , 6.1, 2.3],
[6.3, 3.4, 5.6, 2.4],
[6.4, 3.1, 5.5, 1.8],
[6. , 3. , 4.8, 1.8],
[6.9, 3.1, 5.4, 2.1],
[6.7, 3.1, 5.6, 2.4],
[6.9, 3.1, 5.1, 2.3],
[5.8, 2.7, 5.1, 1.9],
```

```
       [6.8, 3.2, 5.9, 2.3],
       [6.7, 3.3, 5.7, 2.5],
       [6.7, 3. , 5.2, 2.3],
       [6.3, 2.5, 5. , 1.9],
       [6.5, 3. , 5.2, 2. ],
       [6.2, 3.4, 5.4, 2.3],
       [5.9, 3. , 5.1, 1.8]]), 'target': array([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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 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, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]), 'frame': None, 'target_names'
```

```
# Plot the decision boundary

X_scaled = svm_clf["scaler"].transform(X)
x_db = [X_scaled[:,0].min(), X_scaled[:,0].max()]
beta0 = svm_clf["linear_svc"].intercept_[0]
beta1 = svm_clf["linear_svc"].coef_[0][0]
beta2 = svm_clf["linear_svc"].coef_[0][1]
y_db = -(beta0 + np.dot(beta1, x_db)) / beta2
plt.scatter(X_scaled[iris["target"] == 2, 0], X_scaled[iris["target"] == 2, 1],
            marker='^', c='g', s=24, label='Irs-Virginica')
plt.scatter(X_scaled[iris["target"] != 2, 0], X_scaled[iris["target"] != 2, 1],
            marker='s', c='b', s=24, label='Not Iris-Virginica')
plt.plot(x_db, y_db, label='Decision Boundary', c='r')
plt.legend(fontsize=16, loc='lower right')
plt.xlabel('Sepal length', fontsize=16)
plt.ylabel('Petal width', fontsize=16);
```



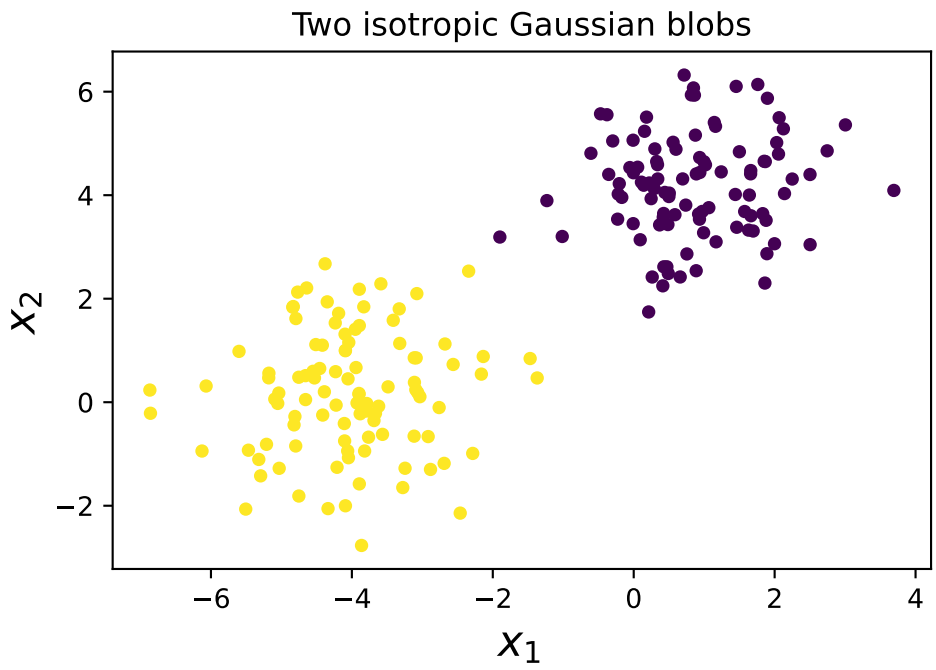

```
# Use petal length and width as features
X = iris["data"][:, 2:]  # petal length and width
# Train the model
svm_clf.fit(X, y)

# Plot the decision boundary

X_scaled = svm_clf["scaler"].transform(X)
x_db = [X_scaled[:,0].min(), X_scaled[:,0].max()]
beta0 = svm_clf["linear_svc"].intercept_[0]
beta1 = svm_clf["linear_svc"].coef_[0][0]
beta2 = svm_clf["linear_svc"].coef_[0][1]
y_db = -(beta0 + np.dot(beta1, x_db)) / beta2
plt.scatter(X_scaled[iris["target"] == 2, 0], X_scaled[iris["target"] == 2, 1],
            marker='^', c='g', s=24, label='Irs-Virginica')
plt.scatter(X_scaled[iris["target"] != 2, 0], X_scaled[iris["target"] != 2, 1],
            marker='s', c='b', s=24, label='Not Iris-Virginica')
plt.plot(x_db, y_db, label='Decision Boundary', c='r')
plt.legend(fontsize=16, loc='lower right')
plt.xlabel('Petal length', fontsize=16)
plt.ylabel('Petal width', fontsize=16);
```



Example 5-2

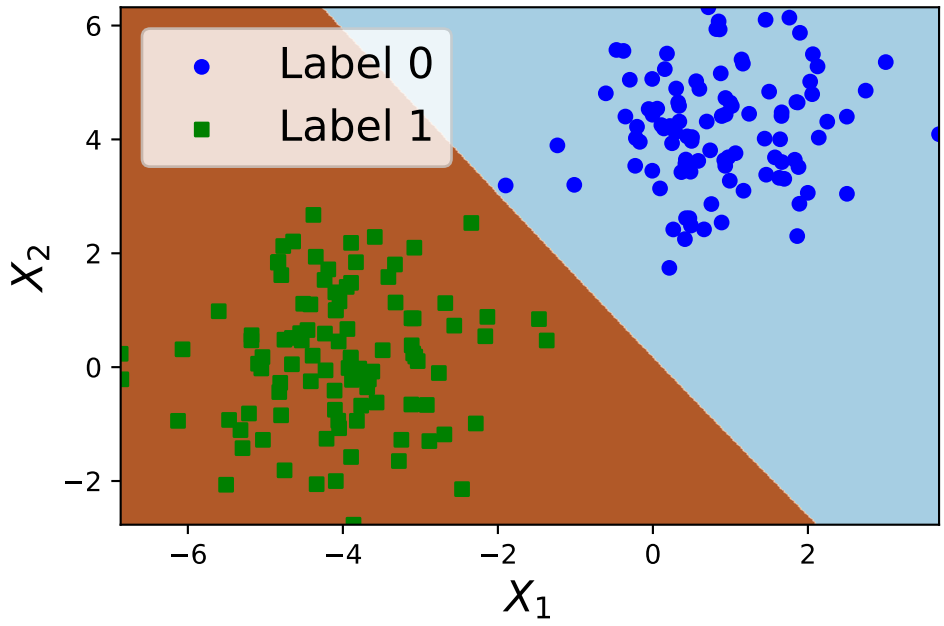
Consider isotropic Gaussian blobs for binary classification (use sklearn.datasets.make_blobs). Use a sigmoid kernel with γ set to "scale" to train a SVM.

```
from sklearn.svm import SVC

# Generate two isotropic Gaussian blobs with two features, each blob has 200 data points
X, y = datasets.make_blobs(n_samples=200, n_features=2, centers=2, random_state=3)
plt.scatter(X[:, 0], X[:, 1], marker='o', c=y, s=16)
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.title('Two isotropic Gaussian blobs');
```




```
# Train SVM with a sigmoid kernel
svm_classifier = SVC(kernel='sigmoid', gamma='scale')
svm_classifier.fit(X, y)

# Plot the decision boundary
# generage grid
x1 = np.linspace(X[:,0].min(), X[:,0].max(), 400)
x2 = np.linspace(X[:,1].min(), X[:,1].max(), 400)
X1, X2 = np.meshgrid(x1, x2)
# flatten X1 and X2
r1, r2 = X1.flatten(), X2.flatten()
# make r1 and r2 2D
r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
# horizontally stack r1 and r2
grid = np.hstack((r1,r2))
# now grid is a feature matrix
# get predicted labels for grid
yhat = svm_classifier.predict(grid)
# reshape yhat so that it has the same shape as X1 and X2
ZZ = yhat.reshape(X1.shape)
plt.contourf(X1, X2, ZZ, cmap='Paired')
plt.scatter(X[y == 0, 0], X[y == 0, 1], marker='o', c='b', s=24, label='Label 0')
```

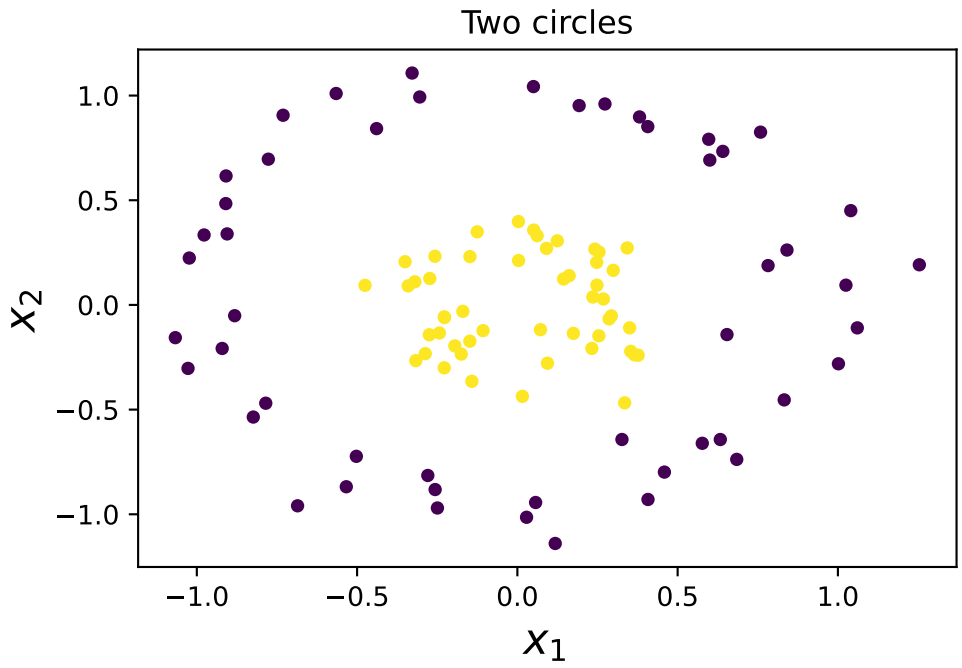
```
plt.scatter(X[y == 1, 0], X[y == 1, 1], marker='s', c='g', s=24, label='Label 1')
plt.legend(fontsize=16)
plt.xlabel('$X_1$', fontsize=16)
plt.ylabel('$X_2$', fontsize=16);
```



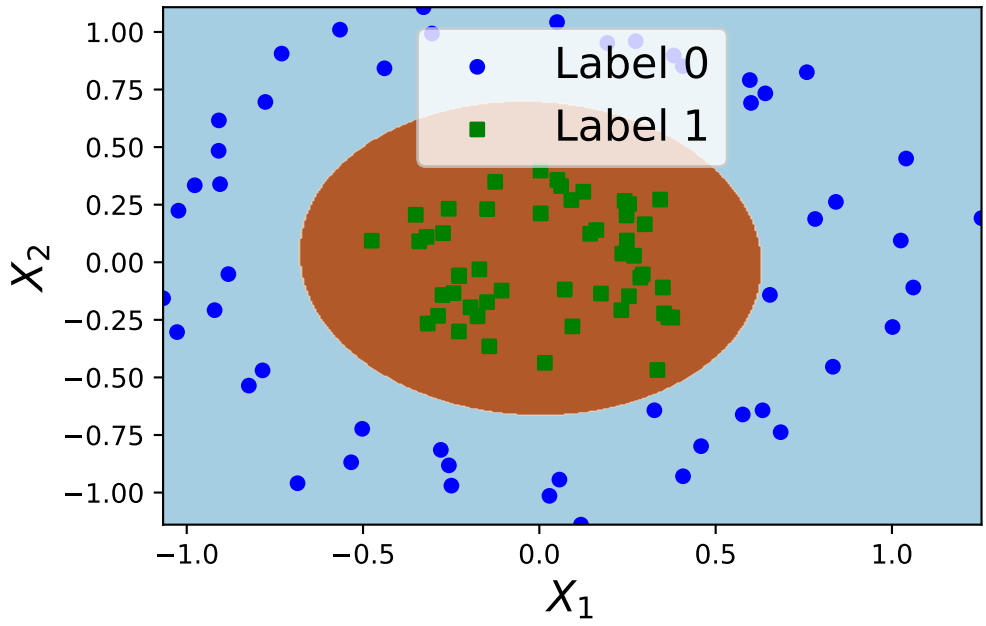
Example 5-3

Train an SVM for two circles (sklearn.datasets.make_circles) using a Gaussin rbf kernel with $\gamma = 0.7$.

```
# Create two circles with a total number of 200 points,
# a scale factor of 0.3 between inner and outer circle in the range [0, 1)
# and a standard deviation of 0.1 of Gaussian noise added to the data.
X, y = datasets.make_circles(n_samples=100, factor=0.3, noise=0.1, random_state=10)

plt.scatter(X[:, 0], X[:, 1], marker='o', c=y, s=16)
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.title('Two circles');

# Train SVM with a sigmoid kernel
svm_classifier = SVC(kernel='rbf', gamma=0.7)
svm_classifier.fit(X, y);
```




```
# Plot the decision boundary
# generage grid
x1 = np.linspace(X[:,0].min(), X[:,0].max(), 400)
x2 = np.linspace(X[:,1].min(), X[:,1].max(), 400)
X1, X2 = np.meshgrid(x1, x2)
# flatten X1 and X2
r1, r2 = X1.flatten(), X2.flatten()
# make r1 and r2 2D
r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
# horizontally stack r1 and r2
grid = np.hstack((r1,r2))
# now grid is a feature matrix
# get predicted labels for grid
yhat = svm_classifier.predict(grid)
# reshape yhat so that it has the same shape as X1 and X2
ZZ = yhat.reshape(X1.shape)
plt.contourf(X1, X2, ZZ, cmap='Paired')
plt.scatter(X[y == 0, 0], X[y == 0, 1], marker='o', c='b', s=24, label='Label 0')
plt.scatter(X[y == 1, 0], X[y == 1, 1], marker='s', c='g', s=24, label='Label 1')
plt.legend(fontsize=16)
plt.xlabel('$X_1$', fontsize=16)
plt.ylabel('$X_2$', fontsize=16);
```



1.5.1 SVM Regression

The SVM algorithm can handle both linear and nonlinear classification, as well as linear and nonlinear regression. In SVM for regression, the aim is to fit as many data points as possible within a margin, while minimizing the number of data points that fall outside the margin (see figure below; the points located outside of the margin are colored differently). The width of this margin boundary is controlled by a hyperparameter ϵ. The following figure illustrates a linear SVM Regression model with a hyperparameter ϵ. The idea is that we can tolerate a deviation that is less than ϵ for a prediction $f(x) - y$, where $f(x)$ is the SVM regression prediction, and y is the target corresponding to feature x. There will be a penalty when the deviation is greater than ϵ. The SVR regression problem can thus be described mathematically as:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} L_\epsilon(f(x_i) - y_i)$$

where L_ϵ is the ϵ-insensitive loss function defined as:

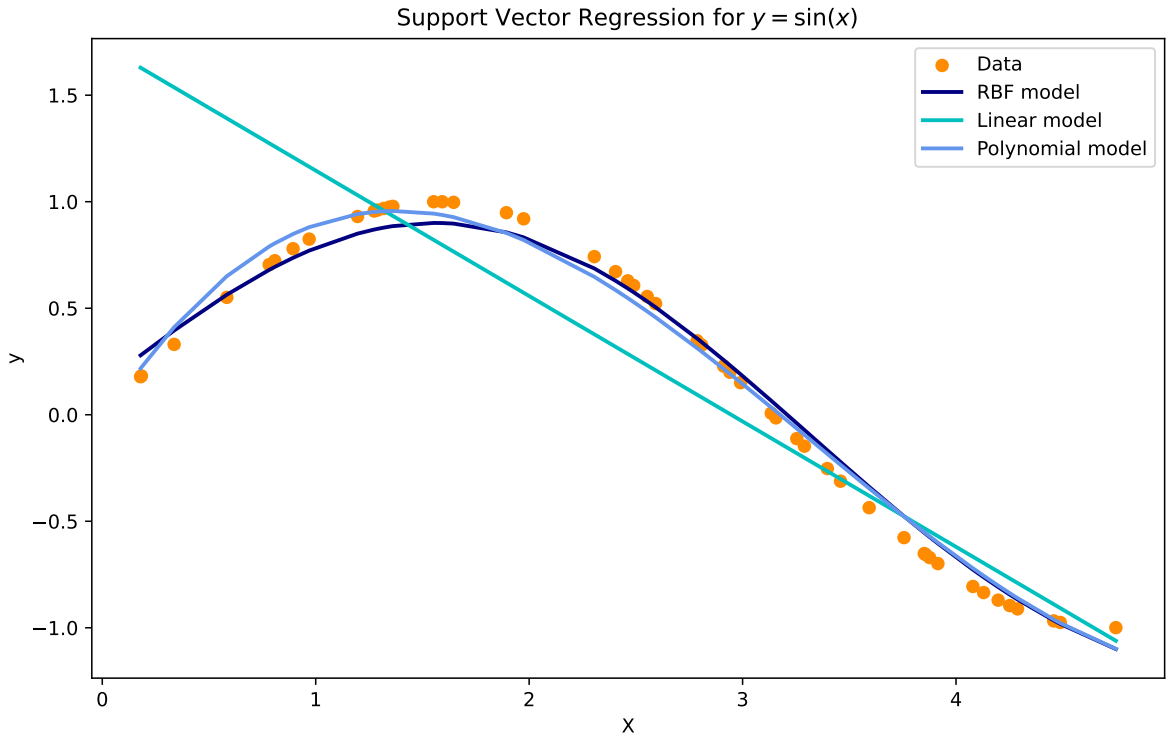
$$L_\epsilon(x) = \begin{cases} 0, & \text{if } |x| \leq \epsilon \\ |x| - \epsilon, & \text{otherwise} \end{cases}$$

Scikit-Learn's LinearSVR class can perform linear SVM Regression, and SVR can perform kernelized SVM model for nonlinear regression tasks.

Example 5-4

Consider the function $y = f(x) = \sin(x)$. Create a dataset of 50 instances with $x \in (0, 5)$ and Build SVM regression models based on the data using three kernels: linear, rbf and polynomial of degree 3.

```
from sklearn.svm import SVR

ndata = 50
# Create the training dataset
np.random.seed(20)
X = np.sort(5 * np.random.rand(ndata))
y = np.sin(X)
X = X[:, np.newaxis]

# Fit SVR models
svr_rbf = SVR(kernel='rbf', C=100, gamma=0.1, epsilon=0.1)
svr_lin = SVR(kernel='linear', C=100, epsilon=0.1)
svr_poly = SVR(kernel='poly', C=100, degree=3, epsilon=0.1, coef0=1)

# Train the SVR models
y_rbf = svr_rbf.fit(X, y).predict(X)
y_lin = svr_lin.fit(X, y).predict(X)
y_poly = svr_poly.fit(X, y).predict(X)

# Plot the original data with the three models
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='darkorange', label='Data')
plt.plot(X, y_rbf, color='navy', lw=2, label='RBF model')
plt.plot(X, y_lin, color='c', lw=2, label='Linear model')
plt.plot(X, y_poly, color='cornflowerblue', lw=2, label='Polynomial model')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Support Vector Regression for $y=\sin(x)$')
plt.legend();
```



1.6 References

1. Cortes, C. and Vapnik, V.N., Support vector networks. *Machine Learning*, 20(3): 273-297.

2. Platt, J.C., Sequential Minimal Optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft Research, 1998.

2 Decision Trees

A **decision tree** is a supervised machine learning algorithm that is used for both classification and regression tasks. It splits the data into subsets/branches recursively based on feature values. Each branch of the tree-like structure serves as a decision rule, and each ending point of the tree (called a **leaf**) leads to a prediction. Given a new data instance, one can start from the **root node** of the tree (that contains all the data points), follow the branches (decision rules), reach internal nodes (called **children**), and finally reach a leaf to make predictions or decisions. The following diagram shows a decision tree to decide weather and how to go for a walk.

Here is another example that shows how a decision tree can help determine if a credit card applicant should be approved or denied.

Decision tree can also be used to predict continuous quantities. The following diagram shows how a decision tree can help predict used car prices.

2.1 Decision Trees for Classification

To train a decision tree, the main goal is to decide which feature to choose at a node and at what value of the feature the node is divided. To this end, we need to define a measure that can be used to compare different choices and values of features for divisions. A *greedy optimization approach* is typically used, beginning with a single root node that represents the entire training dataset. The tree is then grown incrementally by adding nodes one at a time. At each step, there are a set of candidate features and their values in the input space for the split, and once they are chosen, a pair of leaf nodes will be added to the current tree corresponding to the split. Let $\{(x_1, y_1), \dots, (x_N, y_N)\}$ be a training dataset, where $y_i \in \{1, 2, \dots, K\}$, $1 \leq i \leq N$. Assume at a certain step, there has been a collection of leaf nodes $L = \{L_1, L_2, \dots, L_M\}$, For each of the leaf nodes L_i, and a candidate criterion for the split, c, there will be two children for L_i, denoted by $L_i^0(c)$ and $L_i^1(c)$. To quantify the performance of the split, there are two commonly used measures, namely, **information gain** and **Gini index** (also known as **Gini impurity**). To define information gain, we first define the **information entropy** (or just **entropy**) as

$$\mathrm{H}(L_i) = -\sum_{k=1}^{K} p_k \log p_k$$

where $p_k, k = 1, \ldots, K$ is the proportion of data points in L_i that belong to class k, and log is the natural logarithm (base 2 is also commonly used). For each of the two children of L_i, we can compute $\mathrm{H}(L_i^0(c))$ and $\mathrm{H}(L_i^1(c))$ in the same way. The information gain can then be defined as the difference between the entropy of L_i and the weighted some of the entropies of its two children:

$$\mathrm{IG}(L_i, c) = \mathrm{H}(L_i) - \sum_{k=0}^{1} \frac{|L_i^k(c)|}{|L_i|} \mathrm{H}(L_i^k(c))$$

where $|\cdot|$ denotes the number of data points associated with a node. Hence the weights are simply the proportion of data points in the children to the total number of instances in the parent. The larger the information gain, the purer the two children resulting from a split. Notice that if a split results in two pure children, then both children nodes have an entropy of 0. If a node has the same proportion of each class (i.e., $p_k = \frac{1}{K}$, $1 \leq k \leq K$), then its entropy achieves a maximum (see Exercise 6-1).

Another way to measure a potential split criterion c is to evaluate the Gini index, which is defined as

$$G(L_i) = \sum_{k=0}^{K} p_k(1 - p_k) = 1 - \sum_{k=0}^{K} p_k^2$$

$G(L_i)$ will be equal to 0 if L_i is pure, and it is largest when each class has the same proportion of the data instances (see Exercise 6-2). To evaluate the performance of the split criterion c, we calculate the weighted Gini index for the two children nodes:

$$G(L_i, c) = \sum_{k=0}^{1} \frac{|L_i^k(c)|}{|L_i|} G(L_i^k(c))$$

and we pick the best split that leads to the smallest $G(L_i, c)$. The process continues until certain stopping criterion is satisfied, for example, all leaf nodes are pure, or the depth of the tree (number of nodes along the longest path from the root node down to the farthest leaf node) reaches a preset value.

Notice that the algorithm works for both discrete and continuous features. For the latter, a discretization of the feature space needs to be performed before applying the algorithm.

Exercise 6-1

(a) Verify that a node's information entropy achieves its maximum if each of the K classes has the same proportion of its data instances.

(b) Verify that a pure node's information entropy is 0.

Exercise 6-2

(a) Verify that a node's Gini index achieves its maximum if each of the K classes has the same proportion of the data instances.

(b) Verify that a pure node's Gini index is 0.

2.2 Decision Trees for Regression

The algorithm for building a decision tree for regression is similar to that for classification. The only difference is that we need to define a new performance measure for a split. Given the training dataset $\{(x_1, y_1), ..., (x_N, y_N)\}$, assume we have obtained a collection of leaf nodes $L = \{L_1, L_2, ..., L_M\}$. For a node L_i and a candidate criterion for the split, c, the resulting two children nodes are $L_i^0(c)$ and $L_i^1(c)$. Predictions of the target corresponding to a children node should be made by taking the average of all the y_i's of those data points located in the node, which is equivalent to minimizing the sum-of-squares error in the node (see Excercise 6-3). Then, the optimal choice of c will be the one that leads to the smallest residual sum-of-squares error.

To stop the algorithm, we impose some stopping criterion, e.g., the reduction of residual sum-of-squares error falls below some threshold, or another one mentioned in the case of classification.

The following figure shows such an example, where $x = (x_1, x_2)$, i.e., the feature space is two-dimensional. The first step splits the entire feature space into two regions depending on $x_1 < x_{11}$ or not; the second steps splits the region $\{(x, y)|x < x_{11}\}$ into two subregions labeled 1 and 2, depending on $x_2 < x_{21}$ or not; the third step divides the region $\{(x, y)|x > x_{11}\}$ into two subregions depending on $x_2 < x_{22}$ or not, and the resulting two subregions are further divided into regions 3 and 4, and 5 and 6, respectively, depending on if $x_1 < x_{12}$ and $x_1 < x_{13}$.

Exercise 6-3

Show that taking the average of all the y_i's of the data points located in a node as the predicted target is equivalent to minimizing the sum-of-squares error in the node.

2.3 Tree Pruning

To avoid overfitting and increase generalization of decision trees, in the training process, it is a common practice to build a deep tree (e.g., to its maximum depth), and then trim the branches that contribute the least to the model's predictive power. One commonly used method is the **weakest link tree pruning** method (also known as **cost complexity pruning**) Assume a large tree has been built. To apply weakest link tree pruning, we evaluate each subtree in the

decision tree for its contribution to the tree's performance, for example, using the misclassification rate for classification trees or residual sum-of-squares for regressin trees. Once we identify the "weakest link", i.e., the subtree, if removed, that leads to the smallest increase in the error (misclassification rate or residual sum-of-squares), the weakest link is pruned (removed) from the tree. The process is repeated iteratively until a stopping criterion is satisfied, e.g. the smallest increase in error by removing a weakest link is bigger than a threshold.

Example 6-1

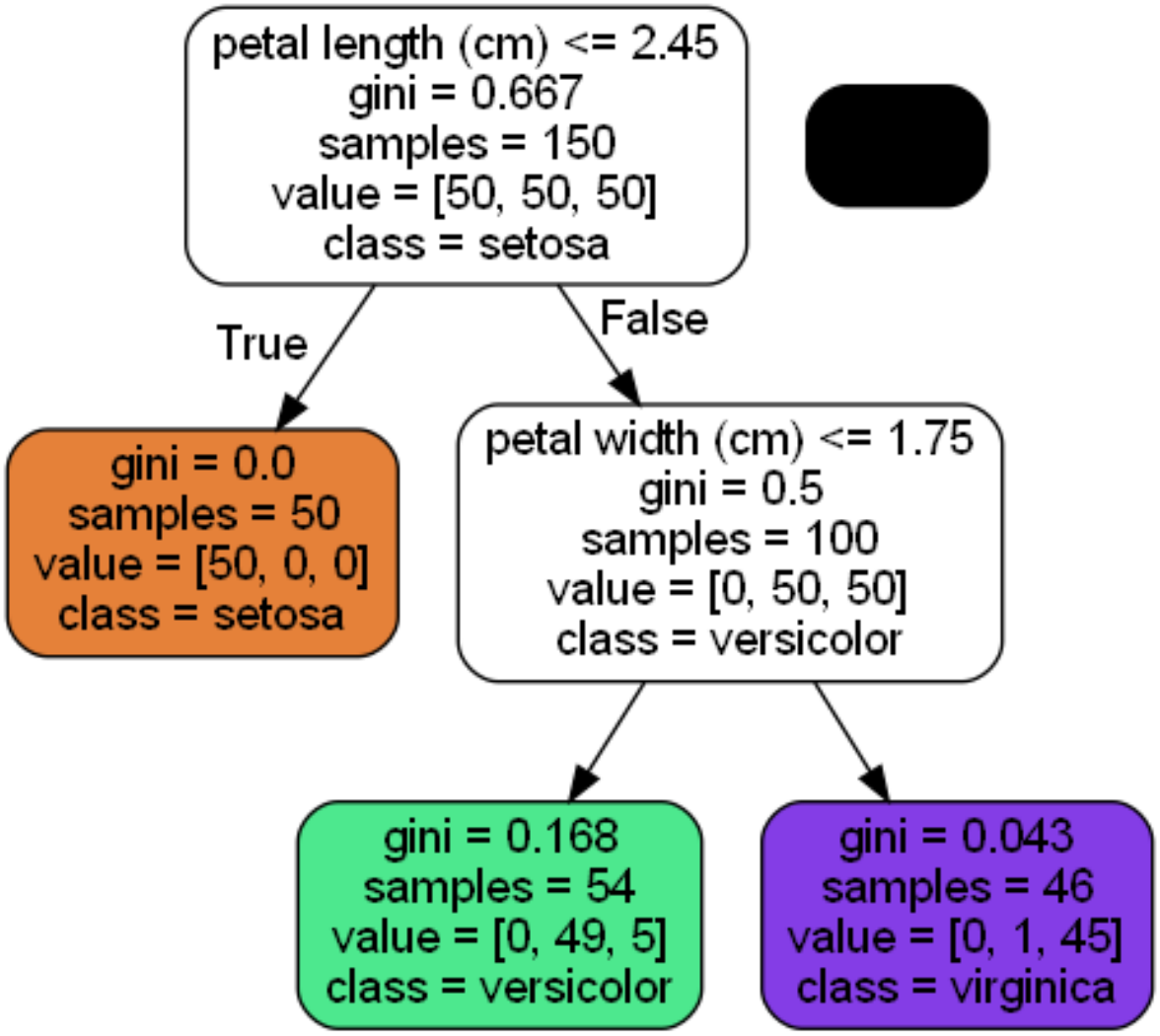
Build a decision tree classifier for the Iris Dataset to classify virginica and non-virginica. Use petal length and petal width as features.

```
import numpy as np
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
import matplotlib.pyplot as plt
%matplotlib inline
import graphviz
import pydotplus
from IPython.display import Image

# Load the data
iris = datasets.load_iris()
# Use sepal length and width as features
X = iris["data"][:, 2:] # sepal length and width
# 0: 'setosa', 1: 'versicolor', 2: 'virginica'
y = iris.target
# Build a decision tree. By default, it uses Gini index
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=32)
tree_clf.fit(X, y)
```

```
DecisionTreeClassifier(max_depth=2, random_state=32)
```

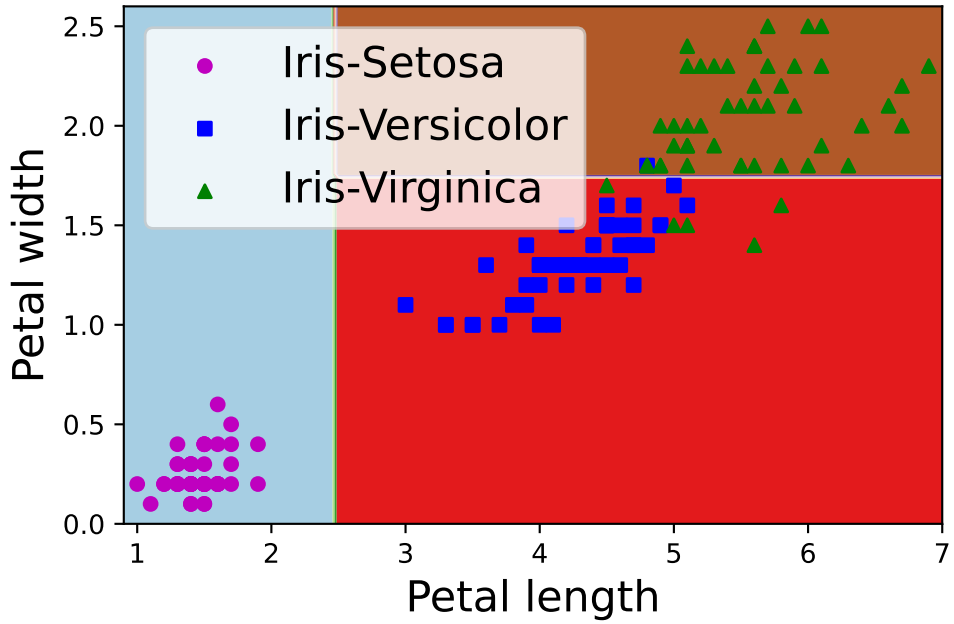
```
# Visualize the trained decision tree

# Export the tree to a dot file
dot_data = export_graphviz(tree_clf, out_file=None,
                           feature_names=iris.feature_names[2:],
                           class_names=iris.target_names, rounded=True,
                           filled=True)

# Convert the .dot file to a graph
```

```
graph = pydotplus.graph_from_dot_data(dot_data)

# Show the graph
Image(graph.create_png())
```




```
# plot decision boundary
# generage grid
x1 = np.linspace(X[:,0].min()-0.1, X[:,0].max()+0.1, 100)
x2 = np.linspace(X[:,1].min()-0.1, X[:,1].max()+0.1, 100)
X1, X2 = np.meshgrid(x1, x2)
# flatten X1 and X2
```

```
r1, r2 = X1.flatten(), X2.flatten()
# make r1 and r2 2D
r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
# horizontally stack r1 and r2
grid = np.hstack((r1,r2))
# now grid is a feature matrix
# get predicted labels for grid
yhat = tree_clf.predict(grid)
# reshape yhat so that it has the same shape as X1 and X2
ZZ = yhat.reshape(X1.shape)
plt.contourf(X1, X2, ZZ, cmap='Paired')
plt.scatter(X[iris["target"] == 0, 0], X[iris["target"] == 0, 1],
marker='o', c='m', s=24, label='Iris-Setosa')
plt.scatter(X[iris["target"] == 1, 0], X[iris["target"] == 1, 1],
marker='s', c='b', s=24, label='Iris-Versicolor')
plt.scatter(X[iris["target"] == 2, 0], X[iris["target"] == 2, 1],
marker='^', c='g', s=24, label='Iris-Virginica')
plt.legend(fontsize=16)
plt.xlabel('Petal length', fontsize=16)
plt.ylabel('Petal width', fontsize=16);
```



In addition, we can estimate the class probabilities for a prediction based on the proportion of data points belonging to the leaf node. For example,

```
tree_clf.predict_proba([[4.5, 1.6]])
```

```
array([[0.        , 0.90740741, 0.09259259]])
```

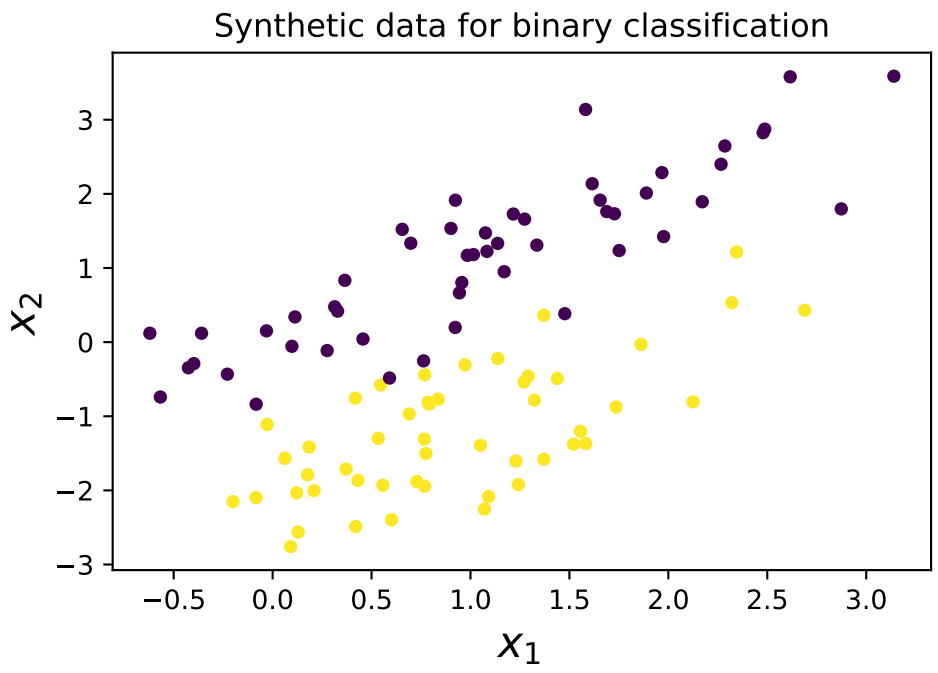
shows that in the leaf node where the data instance $x = (4.5, 1.6)$ fall, there are no Setosa cases, and there are 90.1% Versicolor cases, and 9.3% Virginica cases.

Example 6-2

Consider a binary classification problem where the data are generate with sklearn.datasets.make_classification.

```
from sklearn.datasets import make_classification

# Generate a synthetic dataset using make_classification with 2 features
X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0,
                           n_clusters_per_class=1, random_state=21)
```

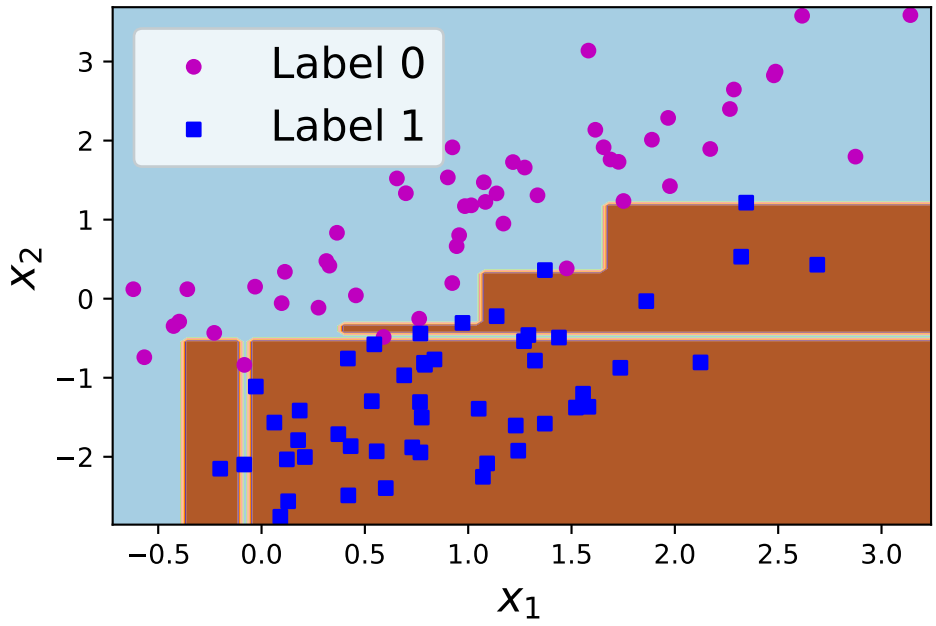
```
# Visualize the data

plt.scatter(X[:,0], X[:,1], c=y, s=16)
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16)
plt.title('Synthetic data for binary classification');
```



```
# Train a Decision Tree Classifier without regularization
clf = DecisionTreeClassifier(random_state=20)
clf.fit(X, y)
```

```
DecisionTreeClassifier(random_state=20)
```

```
# plot decision boundary
# generage grid
x1 = np.linspace(X[:,0].min()-0.1, X[:,0].max()+0.1, 100)
x2 = np.linspace(X[:,1].min()-0.1, X[:,1].max()+0.1, 100)
X1, X2 = np.meshgrid(x1, x2)
# flatten X1 and X2
r1, r2 = X1.flatten(), X2.flatten()
# make r1 and r2 2D
r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
# horizontally stack r1 and r2
grid = np.hstack((r1,r2))
# now grid is a feature matrix
# get predicted labels for grid
yhat = clf.predict(grid)
# reshape yhat so that it has the same shape as X1 and X2
ZZ = yhat.reshape(X1.shape)
plt.contourf(X1, X2, ZZ, cmap='Paired')
plt.scatter(X[y == 0, 0], X[y == 0, 1], marker='o', c='m', s=24, label='Label 0')
plt.scatter(X[y == 1, 0], X[y == 1, 1], marker='s', c='b', s=24, label='Label 1')
plt.legend(fontsize=16)
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16);
```



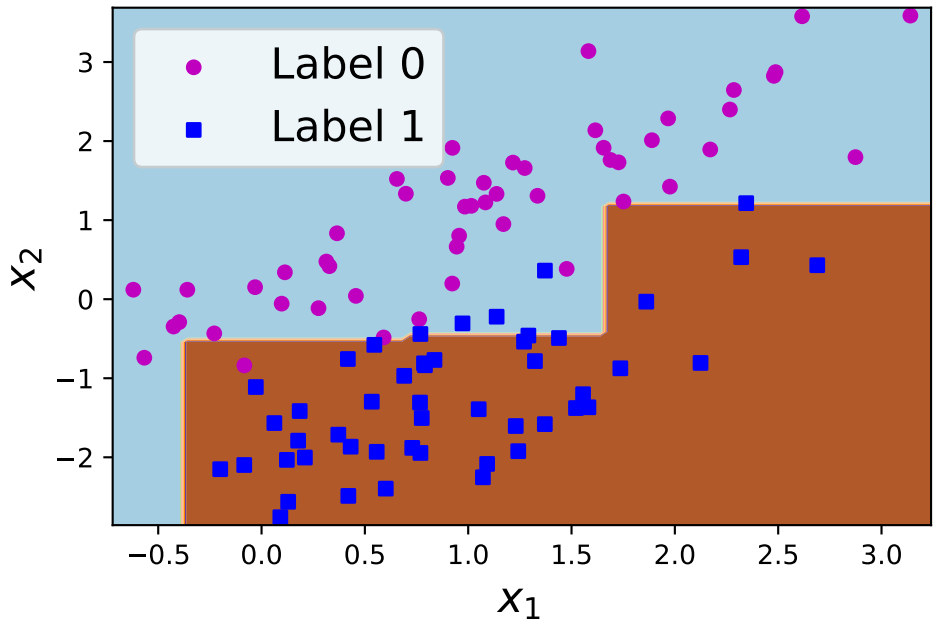
Clearly, the model is overfitting. Now we apply one of the regularization measures for decision trees.

```
# Train a Decision Tree Classifier with regularization
clf = DecisionTreeClassifier(random_state=20, max_depth=4)
clf.fit(X, y)
```

```
DecisionTreeClassifier(max_depth=4, random_state=20)
```

```
# plot decision boundary
# generage grid
x1 = np.linspace(X[:,0].min()-0.1, X[:,0].max()+0.1, 100)
x2 = np.linspace(X[:,1].min()-0.1, X[:,1].max()+0.1, 100)
X1, X2 = np.meshgrid(x1, x2)
# flatten X1 and X2
r1, r2 = X1.flatten(), X2.flatten()
# make r1 and r2 2D
r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
# horizontally stack r1 and r2
grid = np.hstack((r1,r2))
# now grid is a feature matrix
# get predicted labels for grid
yhat = clf.predict(grid)
```

```
# reshape yhat so that it has the same shape as X1 and X2
ZZ = yhat.reshape(X1.shape)
plt.contourf(X1, X2, ZZ, cmap='Paired')
plt.scatter(X[y == 0, 0], X[y == 0, 1], marker='o', c='m', s=24, label='Label 0')
plt.scatter(X[y == 1, 0], X[y == 1, 1], marker='s', c='b', s=24, label='Label 1')
plt.legend(fontsize=16)
plt.xlabel('$x_1$', fontsize=16)
plt.ylabel('$x_2$', fontsize=16);
```



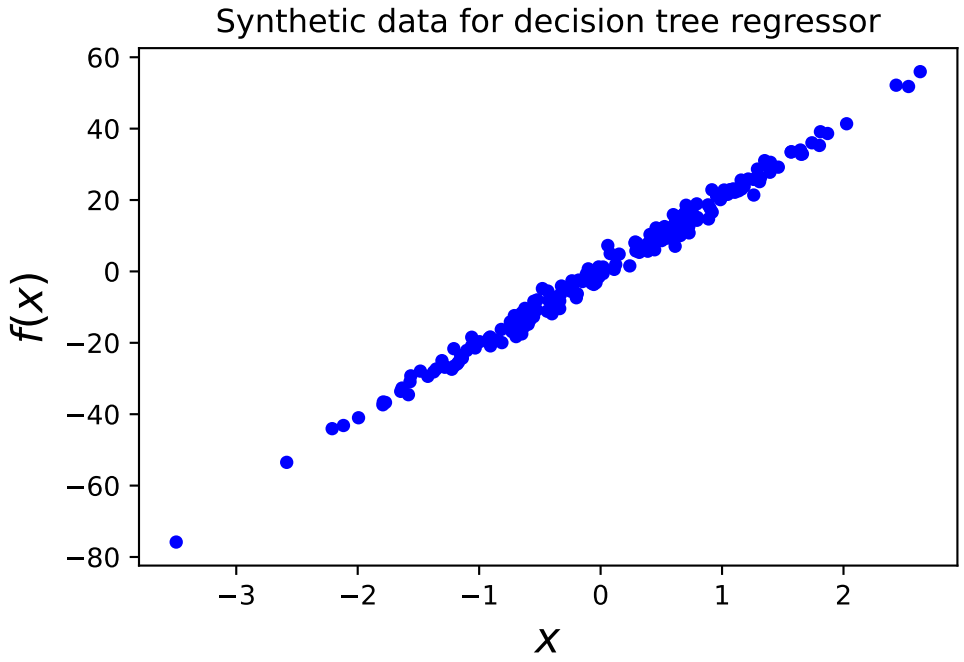
The second model definitely generalizes better.

Example 6-3

Consider a regression problem with decision trees where the data are generated from sklearn.datasets.make_regression.

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.datasets import make_regression

# Generate some sample data
X, y = make_regression(n_samples=200, n_features=1, noise=2.0, random_state=86)

# Visualize the data
```

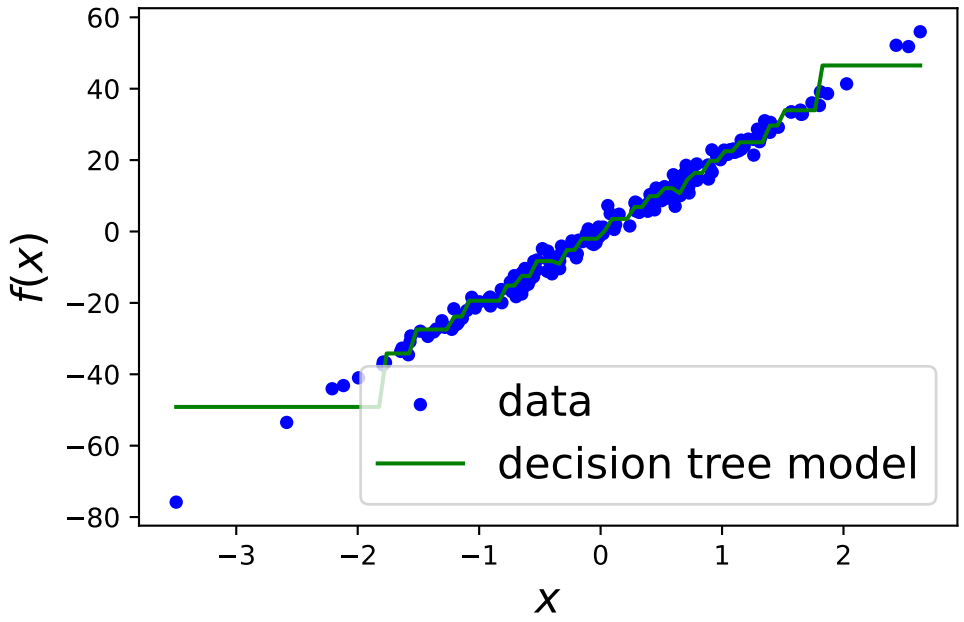
```
plt.scatter(X[:,0], y, c='b', s=16)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16)
plt.title('Synthetic data for decision tree regressor');
```




```
# Fit a Decision Tree Regressor
regressor = DecisionTreeRegressor(min_samples_leaf=6)
regressor.fit(X, y)
```

```
DecisionTreeRegressor(min_samples_leaf=6)
```

```
# plot the decision tree regression model
# generage grid
x = np.linspace(X[:,0].min(), X[:,0].max(), 100)
x = x[:, np.newaxis]
yhat = regressor.predict(x)
plt.scatter(X[:,0], y, c='b', s=16, label='data')
plt.plot(x, yhat, 'g', label='decision tree model')
plt.legend(fontsize=16)
plt.xlabel('$x$', fontsize=16)
plt.ylabel('$f(x)$', fontsize=16);
```


60
40
20
0
−20
−40
−60
−80
f(x)
−3
−2
−1
0
1
2
x
data
decision tree model

3 Ensemble Learning

Ensemble learning is a machine learning technique that combines multiple models (classifiers or regressors, often called **weak learners**) to produce a more accurate and robust one than any individual model. The key idea behind ensemble learning is that by aggregating the predictions from several models, the strengths of individual models can be leveraged, and their weaknesses can be mitigated, thus improving the overall performance of modeling.

3.1 Bagging (Bootstrap Aggregating)

The idea of **bagging**, short for **Bootstrap Aggregating**, is to train multiple models independently on different bootstrapped subsets of the training data, and then the predictions are averaged (for regressors) or voted upon (for classifiers). Bootstrapping is a statistical technique that generates multiple samples from a single (small) dataset by sampling with replacement so that estimates of the distribution of a statistic, e.g., the mean, variance, or confidence interval, can be made when the underlying distribution of the data is unknown. The following diagram illustrates how boostrapping works.

Bagging is particularly effective for reducing variance and preventing overfitting, especially in models with high variability such as decision trees. A high-variability model highly depends on the training dataset. If a different training set is used, then the model can behave quite differently. With bootstrapping, we are creating a group of approximately independent and identically distributed (i.i.d.) training sets, and an individual model (with high variance) is trained on each of the sets. By combining the models and averaging the predictions, the ensemble is likely less variable than any of its component learners.

Let $\{(x_1, y_1), \dots, (x_N, y_N)\}$ be a training dataset for a regression problem. Each feature x_i is d-dimensional. Suppose M samples each of size N are obtained by bootstrapping. For each bootstrapped samples $S_1, S_2, \dots, S_M$, we fit a model $f_i(x), i = 1, \dots, M$. Then the ensemble estimate $f^{\mathrm{E}}(x)$ for a new data instance x is computed as the average of the predictions from the individual models:

$$f^{\mathrm{E}}(x) = \frac{1}{M} \sum_{i=1}^{M} f_i(x)$$

For a K-class classification problem, the algorithm works similarly. With the individual learners $f_i(x), i = 1, \dots, M$, we can obtain a vector $(p_1(x), p_2(x), \dots, p_K(x))$, where $p_i(x)$ represents the proportion of the learners that predict class i for the new instance x. Then the ensemble estimate is:

$$f^{\mathrm{E}}(x) = \arg \max_{k \in \{1,2,\dots,K\}} p_k(x) = \arg \max_{k \in \{1,2,\dots,K\}} \sum_{i=1}^{M} \mathbb{I}(f_i(x) = k)$$

where $\mathbb{I}(\cdot)$ is the indicator function. This above voting method is called **hard voting** or **majority voting**. Another way of voting is to consider at the probability of predicting a class for a new instance (x) for each learner f_i, if the individual learners are equipped with such probabilities (e.g. decision trees). Let $p_{i,j}(x)$ denote the probability of learner f_i predicting class j for data instance (x), where $1 \leq i \leq N$, and $1 \leq j \leq K$. If we average these probabilities for each j, and find the class with the largest average, we can define the **soft voting** rule:

$$f^{\mathrm{E}}(x) = \arg \max_{k \in \{1,2,\dots,K\}} \left\{ \frac{1}{M} \sum_{i=1}^{M} p_{i,k} \right\}$$

For instance, suppose there are three models f_1, f_2, f_3 in the ensemble to predict two classes labeled 1 and 2, and the probabilities are:

$$p_{1,1} = 0.7, p_{1,2} = 0.3, p_{2,1} = 0.4, p_{1,2} = 0.6, p_{3,1} = 0.8, p_{3,2} = 0.2,$$

Then the average probabilities are $(0.7 + 0.4 + 0.8)/3 = 0.63$ for predicting class 1, and $(0.3 + 0.6 + 0.2)/3 = 0.37$ for predicting class 2. The final prediction is class 1 based on the soft voting rule. Soft voting considers the confidence levels of each model, and hence can lead to more accurate ensemble prediction, especially when the individual models are not in strong agreement. However, it does require the individual learners to be capable of outputting probabilities associated with predictions, which many models fail to do.

In the bagging algorithm above, we assume the sampling is done with replacement. In the case of no replacement, the method is called **pasting**. In some cases, we may want to sample from the feature space (i.e., use a subset of features), e.g., when the dimension of the feature space is large. If both the features and data points are randomly selected to create distinct training sets for individual models, then the method is called **random patches**. If only the features are sampled and all the data points are used for the individual models, then the method is called **random subspaces**. In addition, we noticed an obvious advantage of applying bagging (or pasting, random patches, random subspaces), which is training the individual learners can be easily parallelized. This property enables us to train a ensemble model with a large number of components.

Example 7-1

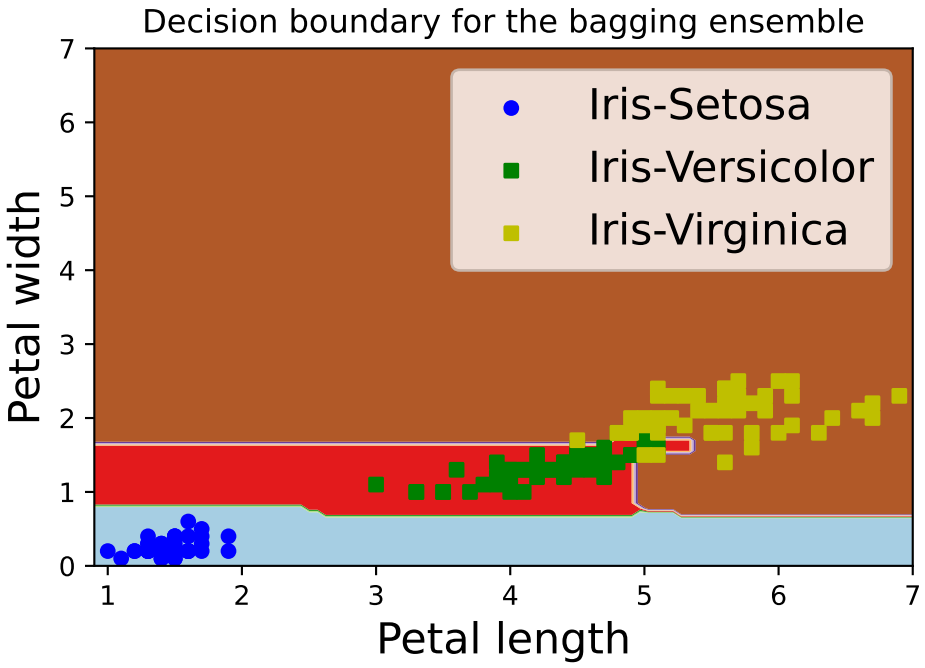
Construct a bagging ensemble of 50 decision trees with no regularization for the Iris dataset. Use the petal length and width as features. Each tree component is trained with 100 bootstrapped instances.

```
from sklearn.datasets import load_iris
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
import matplotlib.pyplot as plt
%matplotlib inline

# Load the Iris dataset
# Use only the last two features petal length and width
iris = load_iris()
X, y = iris.data[:, 2:], iris.target

# Creating the bagging ensemble,
# each member with a sample size of 100.
bag_clf = BaggingClassifier(estimator=DecisionTreeClassifier(), n_estimators=50,
                            max_samples=100, random_state=32)
bag_clf.fit(X, y)
```

```
BaggingClassifier(estimator=DecisionTreeClassifier(), max_samples=100,
                  n_estimators=50, random_state=32)
```

```
# Plot the decision boundary
# generage grid
x1 = np.linspace(X[:,0].min()-0.1, X[:,0].max()+0.1, 100)
x2 = np.linspace(X[:,1].min()-0.1, X[:,0].max()+0.1, 100)
X1, X2 = np.meshgrid(x1, x2)
# flatten X1 and X2
r1, r2 = X1.flatten(), X2.flatten()
# make r1 and r2 2D
r1, r2 = r1.reshape((len(r1), 1)), r2.reshape((len(r2), 1))
# horizontally stack r1 and r2
grid = np.hstack((r1,r2))
# now grid is a feature matrix
# get predicted labels for grid
yhat = bag_clf.predict(grid)
# reshape yhat so that it has the same shape as X1 and X2
ZZ = yhat.reshape(X1.shape)
plt.contourf(X1, X2, ZZ, cmap='Paired')
```

```
plt.scatter(X[y == 0, 0], X[y == 0, 1],
marker='o', c='b', s=24, label='Iris-Setosa')
plt.scatter(X[y == 1, 0], X[y == 1, 1],
marker='s', c='g', s=24, label='Iris-Versicolor')
plt.scatter(X[y == 2, 0], X[y == 2, 1],
marker='s', c='y', s=24, label='Iris-Virginica')
plt.legend(fontsize=16)
plt.xlabel('Petal length', fontsize=16)
plt.ylabel('Petal width', fontsize=16);
plt.title('Decision boundary for the bagging ensemble');
```



3.1.1 Out-of-Bag Score

A byproduct of bagging is that we have a measure to estimate how well the ensemble model performs for new data points, without actually having new data points or evaluating the ensemble model. The reason is that each individual model only sees part of the training data points, since the training set for each individual model is obtained from bootstrapping the entire training dataset. Hence, how the models perform on the data instances they did not see during the training process can be an estimator on how well the ensemble model generalizes. To be specific, a measure can be defined in this way: 1) for each data instance in the training set, we find all the models that did not use it during the training process; 2) evaluate these

models at the data instance, and take the majority vote; 3) the majority vote is either equal to the true label or not; we compute the proportion of the data instances for which the true labels equal the majority votes. The proportion is defined as the **out-of-bag score** (**OOB score**). Mathematically, let D_i, $1 \leq i \leq M$ be the set of data points used to train model f_i. Denote the out-of-bag majority vote for an instance x as $f^{\mathrm{E}}_{\mathrm{OOB}}(x$. Then

$$f^{\mathrm{E}}_{\mathrm{OOB}}(x) = \arg \max_{k \in \{1,2,\ldots,K\}} \sum_{i=1}^{M} \mathbb{I}(f_i(x) = k) \cdot \mathbb{I}(x \notin D_i)$$

and the OOB score can be computed as

$$s_{\mathrm{OOB}} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(f^{\mathrm{E}}_{\mathrm{OOB}}(x_i) = y_i)$$

Similary, we can define the **OOB error**, which is simply 1 minus the OOB score:

$$e_{\mathrm{OOB}} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(f^{\mathrm{E}}_{\mathrm{OOB}}(x_i) \neq y_i) = 1 - s_{\mathrm{OOB}}$$

We now continue with the previous example and compute the OOB score.

```
# The difference is to set oob_score=True
bag_clf = BaggingClassifier(estimator=DecisionTreeClassifier(), n_estimators=50,
                            max_samples=100, oob_score=True, random_state=32)
bag_clf.fit(X, y)
```

```
BaggingClassifier(estimator=DecisionTreeClassifier(), max_samples=100,
                  n_estimators=50, oob_score=True, random_state=32)
```

```
# Now we can see what the OOB socre is:
bag_clf.oob_score_
```

```
0.96
```

To see the detailed information on $f^{\mathrm{E}}_{\mathrm{OOB}}$ for each data instance x, we can do

```
bag_clf.oob_decision_function_
```

```
array([[1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
       [1.        , 0.        , 0.        ],
```

```
[1.        , 0.        , 0.        ],
[1.        , 0.        , 0.        ],
[1.        , 0.        , 0.        ],
[1.        , 0.        , 0.        ],
[1.        , 0.        , 0.        ],
[1.        , 0.        , 0.        ],
[1.        , 0.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 0.79310345, 0.20689655],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 0.93181818, 0.06818182],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 0.07692308, 0.92307692],
[0.        , 1.        , 0.        ],
[0.        , 0.71428571, 0.28571429],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 0.93333333, 0.06666667],
[0.        , 0.        , 1.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 0.125     , 0.875     ],
[0.        , 1.        , 0.        ],
[0.        , 0.9375    , 0.0625    ],
```

```
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 1.        , 0.        ],
[0.        , 0.        , 1.        ],
[0.        , 0.03448276, 0.96551724],
[0.        , 0.        , 1.        ],
[0.        , 0.00925926, 0.99074074],
[0.        , 0.        , 1.        ],
[0.        , 0.        , 1.        ],
[0.        , 0.76984127, 0.23015873],
[0.        , 0.00961538, 0.99038462],
[0.        , 0.00961538, 0.99038462],
[0.        , 0.        , 1.        ],
[0.        , 0.05263158, 0.94736842],
[0.        , 0.        , 1.        ],
[0.        , 0.        , 1.        ],
[0.        , 0.04761905, 0.95238095],
[0.        , 0.03225806, 0.96774194],
[0.        , 0.        , 1.        ],
[0.        , 0.00833333, 0.99166667],
[0.        , 0.        , 1.        ],
[0.        , 0.        , 1.        ],
[0.        , 0.96      , 0.04      ],
[0.        , 0.        , 1.        ],
[0.        , 0.15909091, 0.84090909],
[0.        , 0.        , 1.        ],
[0.        , 0.1484375 , 0.8515625 ],
[0.        , 0.        , 1.        ],
[0.        , 0.01      , 0.99      ],
[0.        , 0.50438596, 0.49561404],
[0.        , 0.15833333, 0.84166667],
[0.        , 0.        , 1.        ],
```

```
[0.        , 0.375     , 0.625    ],
[0.        , 0.        , 1.       ],
[0.        , 0.        , 1.       ],
[0.        , 0.        , 1.       ],
[0.        , 0.46153846, 0.53846154],
[0.        , 0.04      , 0.96     ],
[0.        , 0.        , 1.       ],
[0.        , 0.        , 1.       ],
[0.        , 0.01190476, 0.98809524],
[0.        , 0.3452381 , 0.6547619 ],
[0.        , 0.        , 1.       ],
[0.        , 0.        , 1.       ],
[0.        , 0.04347826, 0.95652174],
[0.        , 0.04545455, 0.95454545],
[0.        , 0.        , 1.       ],
[0.        , 0.        , 1.       ],
[0.        , 0.03846154, 0.96153846],
[0.        , 0.04166667, 0.95833333],
[0.        , 0.03125   , 0.96875  ],
[0.        , 0.        , 1.       ],
[0.        , 0.09782609, 0.90217391]])
```

For example, for the last data instance, among all the individual models not using it during the training stage, 90.2% of them predict it to be Verginica, while 9.8% predict it to be Versicolor ($f^{\mathrm{E}}_{\mathrm{OOB}} = 3$).

3.2 Random Forests

Random Forest (RF) is a variation of bagging. The motivation behind RF is to create training datasets for individual trees that are less dependent on each other (a collection of decorrelated trees). To this end, more randomness is introduced in the sampling process. For each individual model, a bootstrapped sample is first randomly selected. In the following process that construct a decision tree, for each node, instead of looking at all the possible choices of feature for a split and all the split points, RF randomly selects a subset of k features, where $k < d$, a split criterion is decided based on the subset of features. Note that if $k = d$, then the ensemble is simply a regular bagging of decision trees. Usually, k is chosen as $\log_2 d$. The randomness resulting from randomly sampling the features increase the independency of the individual models in the ensemble.

Now we use *RandomForestClassifier* to perform RF with Python for the wine data set (https://scikit-learn.org/stable/datasets/toy_dataset.html#wine-dataset).

```python
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# Load the Wine dataset
wine = load_wine()
X, y = wine.data, wine.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize and train the Random Forest Classifier
rf_clf = RandomForestClassifier(n_estimators=100, random_state=21)
rf_clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf_clf.predict(X_test)

# Evaluate the model's performance
accuracy = rf_clf.score(X_test, y_test)
print(f"Random Forest Classifier Accuracy: {accuracy:.2f}")
```

```
Random Forest Classifier Accuracy: 1.00
```

We achieved an accuracy of 1.0. It is not surprising that random forest performs so well. Actually, RF is used much more often than the other bagging techniques, and it is also commonly used as a baseline model, before more complicated models, such as deep neural networks, are attempted.

3.2.1 Importance Score associate with Random Forests

In Random Forests, feature importance scores are a byproduct that can be used to evaluate the significance of each feature in predicting the target variable. Feature importance is an important topic in machine learning interpretability/explainability. Many models do not carry built-in feature importance scores as RF, and hence one has to apply some model-agnostic methods to compute those scores. As a result, this is a key advantage of Random Forests, as they directly provide a way to understand which features are most influential in the model's predictions.

The most common method for calculating feature importance for RF is by considering the how much each feature contributes to reducing the impurity (e.g., Gini impurity) in the decision

trees within the Random Forest. For each feature, the decrease in impurity is averaged over all the trees in the forest. A feature that results in a significant decrease in impurity is considered more important.

```
# Print out the feature importances for all features:
print('feature importance:', rf_clf.feature_importances_)
print()
# A better print-out: including the feature names,
# so that we know which feature importance is for which feature
for i in range(rf_clf.feature_importances_.size):
    print(wine["feature_names"][i], rf_clf.feature_importances_[i])
```

```
feature importance: [0.12920959 0.02570678 0.01617244 0.02701698 0.02524581 0.06519636
 0.15131565 0.01371709 0.0223001  0.17238892 0.07560528 0.14544402
 0.13068099]

alcohol 0.1292095910497735
malic_acid 0.025706776014575165
ash 0.016172443959041227
alcalinity_of_ash 0.027016975258495418
magnesium 0.025245808545271965
total_phenols 0.06519636188629413
flavanoids 0.1513156473223214
nonflavanoid_phenols 0.013717093062667347
proanthocyanins 0.022300097134065243
color_intensity 0.17238891683084545
hue 0.07560527634220683
od280/od315_of_diluted_wines 0.1454440242300447
proline 0.1306809883643975
```

The results show that the features “alcohol”, “flavanoids”, “color_intensity”, “od280/od315_of_diluted_wines”, and “proline” play a more important role than the others in the predictions of the ensemble model.

3.3 Boosting

Boosting works differently from bagging, where a collection of independent weak learners with high variance are combined to produce a model that generalizes well. For boosting, the idea is that a collection of sequential models with high bias are combined to produce a stronger predictive model. Each new model in the ensemble is trained sequentially with the purpose

of correcting the errors made by the previous models. Two boosting techniques are commonly used: **AdaBoost** and **Gradient Boosting**. We start with the discussion of AdaBoost.

3.3.1 Adaptive Boosting (AdaBoost)

The idea of AdaBoost is to assign weights to training data instances. A subsequent model tries to put more weights on the instances that are predicted wrong by the previous models. The final ensemble model is a linear combination (weighted sum) of all the individual models with more accurate individual models assigned a larger coefficients (weights). The algorithm works as follows.

Step 1. Initialize the weights for data instances

- All data instances are initially assigned an equal weight, $w_i^{(1)} = \frac{1}{N}$, $1 \leq i \leq N$. Here the superscript denotes the iteration number.

Step 2. Train weak learners

For each iteration m, $1 \leq i \leq M$, where M is the number of individual models to be constructed,

- train a weak learner f_m, such as a shallow decision tree, based on the weighted samples. That is, minimizing a weighted error function (e.g., in Scikit-Learn the fit method for the classifier has a sample_weight optional input).
- compute the weighted error:

$$\epsilon_m = \frac{\sum_{i=1}^{N} w_i^{(m)} \cdot \mathbb{I}\{f_m(x_i) \neq y_i\}}{\sum_{i=1}^{N} w_i^{(m)}}$$

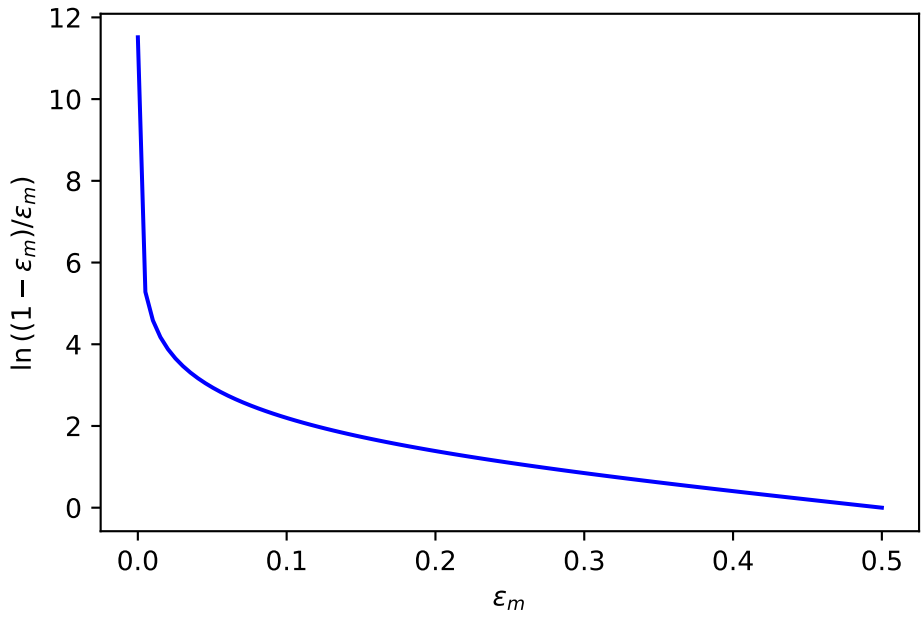
This can be explained as the weighted sum of misclassified instances. The denominator is always 1 as seen in the following steps. Note that if the weights are equal for all data instances, ϵ_m is simply the proportion of data instances that are predicted wrong by f_m.

- compute the weight of learner f_m

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1 - \epsilon_m}{\epsilon_m} \right)$$

The smaller ϵ_m, the larger the weight α_m, as seen below. As α_m approaches 0.5 (meaning approaching a random model), α_m is close to 0.

```
eps = np.linspace(0.00001, 0.5, 100)
plt.plot(eps, np.log((1-eps)/eps), 'b-')
plt.xlabel('$\epsilon_m$')
plt.ylabel('$\ln{((1-\epsilon_m)/\epsilon_m)}$');
```



Step 3. Update Sample Weights

- Increase the weights of f_m-misclassified samples:

$$w_i^{(m+1)} = w_i^{(m)} \cdot \exp\left(\alpha_m \cdot \mathbb{I}\{f_m(x_i) \neq y_i\}\right), \quad 1 \leq i \leq N$$

This means the weights of the misclassified instances are magnified, while the weights of the correctly classified instances get smaller, due to the normalization below.

- Normalize weights

$$w_i^{(m+1)} = \frac{w_i^{(m+1)}}{\sum_{j=1}^{N} w_j^{(m+1)}}, \quad 1 \leq i \leq N$$

Step 4. Combine Weak Learners

- The final ensemble model is:

$$f^{\mathrm{E}}(x) = \operatorname{sign}\left(\sum_{m=1}^{M} \alpha_m \cdot f_m(x)\right)$$

The principles behind these formulas are the minimization of an exponential loss function (see Friedman (2000)). Due to the complexity of the derivation, we will leave it out.

Example 7-2

Apply AdaBoost to the breast cancer dataset (https://scikit-learn.org/stable/datasets/toy_dataset.html#breast cancer-dataset). The weak learners are decision trees with a maximum depth of 1.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score

# Load the Breast Cancer dataset
data = load_breast_cancer()
X, y = data.data, data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize and train the weak learners
ada_model = AdaBoostClassifier(estimator=DecisionTreeClassifier(max_depth=1),
                               algorithm='SAMME',  # for the purpose of suppressing a warning
                               n_estimators=100, random_state=90)
ada_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = ada_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"AdaBoost Classifier Accuracy: {accuracy:.2f}")
```

```
AdaBoost Classifier Accuracy: 0.97
```

3.3.2 Gradient Boosting

Gradient Boosting builds models sequentially in the way that each subsequent model is trained to correct the residual errors of the combined predictions from previous models. The final

model is a weighted sum of all the individual models. The name comes from the fact that it uses gradient descent to minimize the loss function. Here are the details of the algorithm.

Step 1. Initialize a model

The initial model, $f_0(x)$, used in gradient boosting is typically a constant function, and the lost function L is typically the Mean Squared Error (MSE). Hence the initial model is:

$$f_0(x) = \arg\min_c \sum_{i=1}^{N} L(y_i, c)$$

We have seen in the chapter on decision trees that the solution to the optimization problem is trivial:

$$f_0(x) = \frac{1}{N} \sum_{i=1}^{N} y_i$$

Step 2. Compute the residuals

For each iteration m, $1 \leq i \leq M$, where M is the number of individual models to be constructed, we are trying to find the mth individual model f_m. By the motivation of gradient boosting, f_m approximates the residuals of the current model, which is the negative gradient of the loss function L with respect to the current prediction:

$$r_i^{(m)} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$$

where $F_{m-1}(x)$ is the ensemble model with $m-1$ individual models. The residual represents the direction where the ensemble model needs to move to fast decrease the cost function. In the case of L being the MSE function, the residual $r_i^{(m)}$ is simply $y_i - F_{m-1}(x_i)$, the difference between the true value (label) and the current ensemble prediction.

Step 3. Fit the new individual model f_m that approximates the residual $r_i^{(m)}$

$$f_m(x) = \arg\min_f \sum_{i=1}^{N} \left(r_i^{(m)} - f(x_i)\right)^2$$

Step 4. Update the ensemble model $F_m(x)$

$$F_m(x) = F_{m-1}(x) + \eta f_m(x)$$

where η is the learning rate.

At the end, we have the final gradient boost ensemble model, $F^{\mathrm{E}}(x) = F_M(x)$. It is clearly seen that the whole process is a gradient descent in the function space.

Now we use decision trees as the individual models in the gradient boosting ensemble for a synthetic problem. Such gradient boosting is called **Gradient Tree Boosting**, or **Gradient Boosted Regression Trees (GBRT)**.

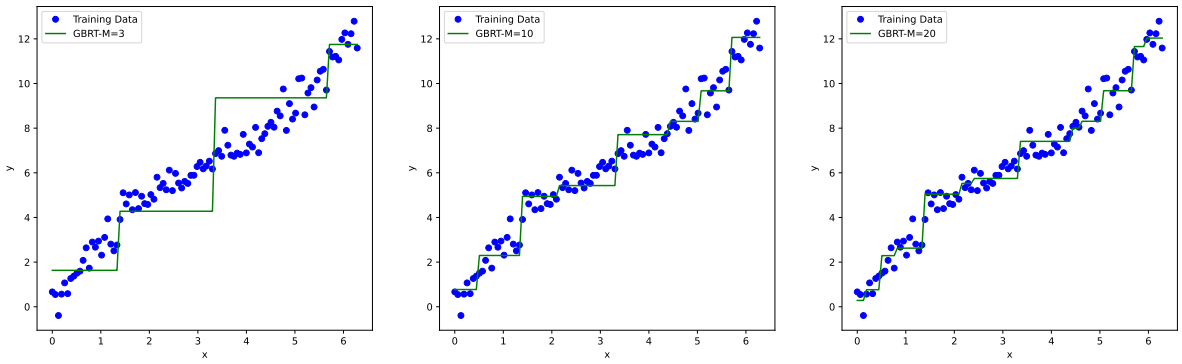
Example 7-3

Use sklearn.ensemble.GradientBoostingRegressor to build a gradient boosting model for noisy data generated from the underlying function

$$f(x) = 2x + \sin x$$

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error

# Create a synthetic dataset
ndata = 100  # 100 data points
np.random.seed(10)
X = np.linspace(0, 2*np.pi, ndata)  # data points are between 0 and 2pi
y = 2 * X + np.sin(X) + np.random.normal(0, 0.5, X.size)
X = X[:, np.newaxis]

# Initialize and train a Gradient Boosting Regressor with 3 components.
gbr3 = GradientBoostingRegressor(n_estimators=3, max_depth=1, learning_rate=1, random_state=4
gbr3.fit(X, y)

# Predictions
y_pred_3 = gbr3.predict(X)

# Plotting the results
fig, ax = plt.subplots(1, 3, figsize=(21, 6))

ax[0].plot(X, y, 'bo', label='Training Data')
ax[0].plot(X, y_pred_3, 'g-', label='GBRT-M=3')
ax[0].legend()
ax[0].set_xlabel('x')
ax[0].set_ylabel('y');

# Initialize and train a Gradient Boosting Regressor with 10 components.
gbr10 = GradientBoostingRegressor(n_estimators=10, max_depth=1, learning_rate=1, random_state
gbr10.fit(X, y)
```

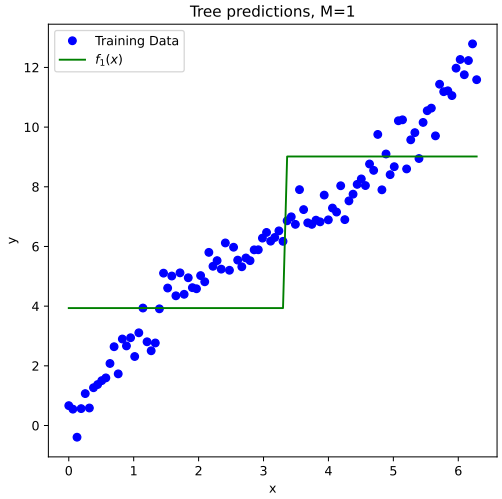
```
# Predictions
y_pred_10 = gbr10.predict(X)

# Plotting the results
ax[1].plot(X, y, 'bo', label='Training Data')
ax[1].plot(X, y_pred_10, 'g-', label='GBRT-M=10')
ax[1].legend()
ax[1].set_xlabel('x')
ax[1].set_ylabel('y');

# Initialize and train a Gradient Boosting Regressor with 20 components.
gbr20 = GradientBoostingRegressor(n_estimators=20, max_depth=1, learning_rate=1, random_state
gbr20.fit(X, y)

# Predictions
y_pred_20 = gbr20.predict(X)

# Plotting the results
ax[2].plot(X, y, 'bo', label='Training Data')
ax[2].plot(X, y_pred_20, 'g-', label='GBRT-M=20')
ax[2].legend()
ax[2].set_xlabel('x')
ax[2].set_ylabel('y');
```

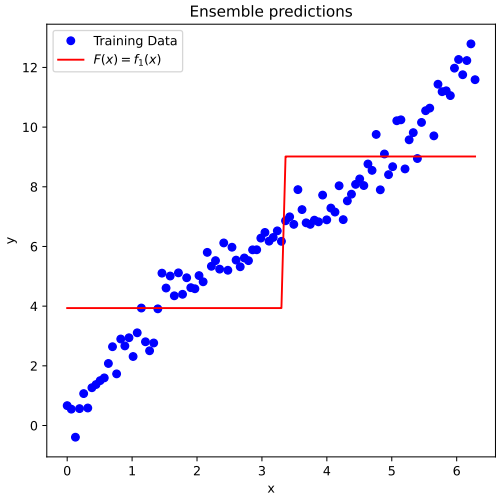


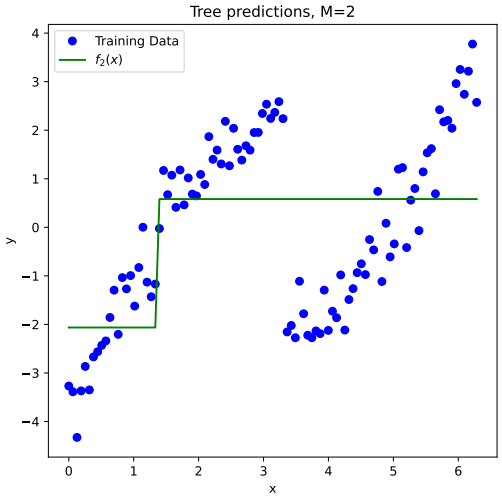
There is an obvious improvement by increasing M from 3 to 10. Further increasing M to 20 does not lead to obvious change of the ensemble model. The gradient boosting algorithm tells us that the ensemble model is obtained iteratively by adding the component that approximates the previous residuals. We verify this by not using the built-in GradientBoostingRegressor class.

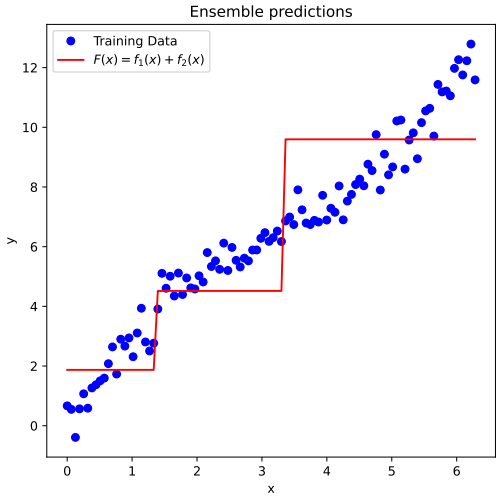
```
from sklearn.tree import DecisionTreeRegressor

fig, ax = plt.subplots(3, 2, figsize=(14, 21))

# Train the DT regressor on X and y
# It is the first decision tree regressor on the data
dt_reg1 = DecisionTreeRegressor(max_depth=1)
dt_reg1.fit(X, y)

ax[0,0].plot(X, y, 'bo', label='Training Data')
ax[0,0].plot(X, dt_reg1.predict(X), 'g-', label='$f_1(x)$')
ax[0,0].legend()
ax[0,0].set_xlabel('x')
ax[0,0].set_ylabel('y')
ax[0,0].set_title('Tree predictions, M=1')

# The ensemble will have only one individual model
ax[0,1].plot(X, y, 'bo', label='Training Data')
ax[0,1].plot(X, dt_reg1.predict(X), 'r-', label='$F(x) = f_1(x)$')
ax[0,1].legend()
ax[0,1].set_xlabel('x')
ax[0,1].set_ylabel('y');
ax[0,1].set_title('Ensemble predictions');


# Train the second DT regressor on the residual errors made by the first predictor
# It is the second decision tree regressor on the data
r = y - dt_reg1.predict(X)
dt_reg2 = DecisionTreeRegressor(max_depth=1)
dt_reg2.fit(X, r)

ax[1,0].plot(X, r, 'bo', label='Training Data')
ax[1,0].plot(X, dt_reg2.predict(X), 'g-', label='$f_2(x)$')
ax[1,0].legend()
ax[1,0].set_xlabel('x')
ax[1,0].set_ylabel('y')
ax[1,0].set_title('Tree predictions, M=2')

# The ensemble will have two individual models
ax[1,1].plot(X, y, 'bo', label='Training Data')
ax[1,1].plot(X, dt_reg1.predict(X)+dt_reg2.predict(X), 'r-', label='$F(x) = f_1(x)+f_2(x)$')
ax[1,1].legend()
```

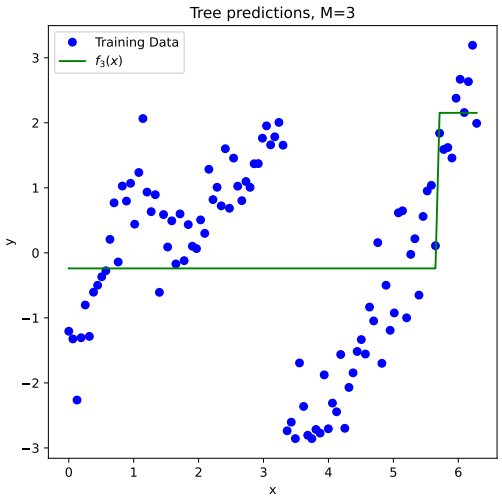
```
ax[1,1].set_xlabel('x')
ax[1,1].set_ylabel('y');
ax[1,1].set_title('Ensemble predictions');

# Train the third DT regressor on the residual errors made by the first two predictors
# It is the third decision tree regressor on the data
r = r - dt_reg2.predict(X)
dt_reg3 = DecisionTreeRegressor(max_depth=1)
dt_reg3.fit(X, r)

ax[2,0].plot(X, r, 'bo', label='Training Data')
ax[2,0].plot(X, dt_reg3.predict(X), 'g-', label='$f_3(x)$')
ax[2,0].legend()
ax[2,0].set_xlabel('x')
ax[2,0].set_ylabel('y')
ax[2,0].set_title('Tree predictions, M=3')

# The ensemble will have two individual models
ax[2,1].plot(X, y, 'bo', label='Training Data')
ax[2,1].plot(X, dt_reg1.predict(X)+dt_reg2.predict(X)+dt_reg3.predict(X),
             'r-', label='$F(x) = f_1(x)+f_2(x)+f_3(x)$')
ax[2,1].legend()
ax[2,1].set_xlabel('x')
ax[2,1].set_ylabel('y');
ax[2,1].set_title('Ensemble predictions');
```

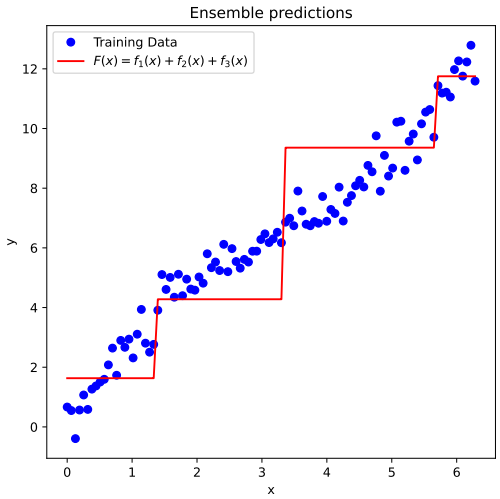

Tree predictions, M=1
Training Data
$f_1(x)$
x
y


Ensemble predictions
Training Data
$F(x) = f_1(x)$
x
y


Tree predictions, M=2
Training Data
$f_2(x)$
x
y


Ensemble predictions
Training Data
$F(x) = f_1(x) + f_2(x)$
x
y


Tree predictions, M=3
Training Data
$f_3(x)$
x
y


Ensemble predictions
Training Data
$F(x) = f_1(x) + f_2(x) + f_3(x)$
x
y

Note that the plot in the bottom right corner is the same as that for the gradient boosting regressor with 3 components trained using the built-in class in the previous cell, confirming that either way we obtain the same ensemble model.

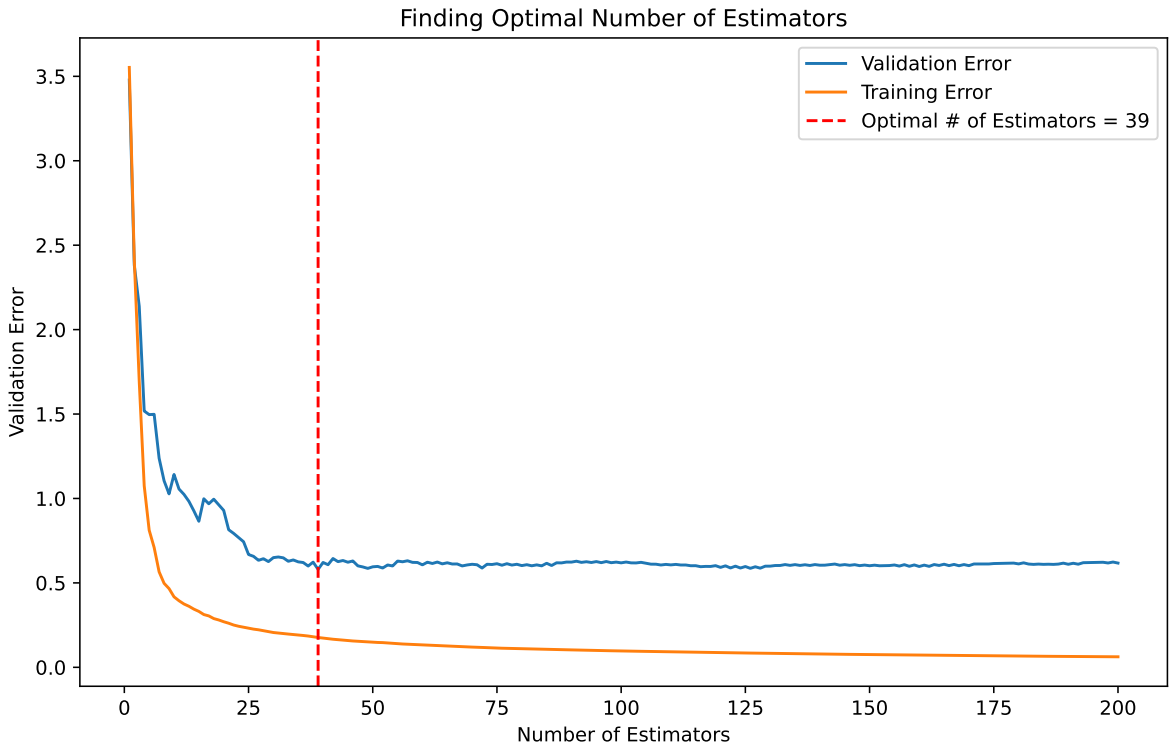
3.3.3 Optimal Ensemble Size

We can use a validation set to monitor the model's performance and perform *early stopping* when the model's performance on the validation set starts to decrease. The Python code below does the job.

```
from sklearn.metrics import mean_squared_error

# We use the same synthetic data

# Split the data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=30)

# Initialize a Gradient Boosting Regressor with a large number of estimators
gbr = GradientBoostingRegressor(n_estimators=200, learning_rate=1, max_depth=1, random_state=

# Fit the model while tracking the validation error at each stage
gbr.fit(X_train, y_train)

# Compute validation error after each stage
errors_val = [mean_squared_error(y_val, y_pred) for y_pred in gbr.staged_predict(X_val)]
# Compute training error after each stage
errors_train = [mean_squared_error(y_train, y_pred_train) for y_pred_train in gbr.staged_pred

# Find the optimal number of estimators (minimizing validation error)
best_n_estimators = np.argmin(errors_val)
print(f"Optimal number of estimators: {best_n_estimators+1}")  # Adding 1 because indexing st

# Plot the validation error
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(errors_val)+1), errors_val, label='Validation Error')
plt.plot(range(1, len(errors_train)+1), errors_train, label='Training Error')
plt.axvline(best_n_estimators+1, color='red', linestyle='--', label=f'Optimal # of Estimators
plt.xlabel('Number of Estimators')
plt.ylabel('Validation Error')
plt.title('Finding Optimal Number of Estimators')
plt.legend()
plt.show()
```

```
# Re-train the model with the optimal number of estimators
gbr_optimal = GradientBoostingRegressor(n_estimators=best_n_estimators, learning_rate=1, max_
gbr_optimal.fit(X_train, y_train)

# Evaluate on the validation set
y_val_pred = gbr_optimal.predict(X_val)
mse_val = mean_squared_error(y_val, y_val_pred)
print(f"Validation MSE with optimal estimators: {mse_val:.2f}")
```

```
Optimal number of estimators: 39
```




```
Validation MSE with optimal estimators: 0.62
```

3.4 References

1. J. Friedman, T. Hastie, and R. Tibshirani, Additive logistic regression: a statistical view of boosting (With discussion and a rejoinder by the authors), Annals of Statistics, 28(2): 337-407.

4 Dimension Reduction

In an age of data explosion, we often see data in high dimensions. High-dimensional data can be very complex, making it challenging to analyze and interpret. As the number of dimensions increases, it becomes more difficult to recognize patterns and the performance of machine learning algorithms is seriously affected. This is the so-called **Curse of Dimensionality**. In addition, high-dimensional data is challenging for visualization. As a result, dimensionality reduction, which involves reducing the number of features while retaining as much relevant information as possible, is a crucial step in machine learning. We will introduce a few of them, including principal component analysis, incremental principal component analysis, and kernel principal component analysis.

4.1 Principal Component Analysis (PCA)

Principal component analysis (PCA) transforms data of a higher dimension to a new coordinate system with a lower dimension. During the transformation, the variance of the data, which can be considered as the information the data contain, is conserved as much as possible. The core of the algorithm is an eigendecomposition of the covariance matrix constructed with the mean-removed/centered data, leading to a set of eigenvectors, called **principal components**, and eigenvalues, which provides important information on the dimension of the reduced coordinates.

Suppose we have a set of data points/observations $\{x_1, x_2, \dots, x_N\}$, and each data point is d-dimensional, i.e., $x_i = (x_i^1, x_i^2, \dots, x_i^d)^T$. Combining all the data points into a data matrix X,

$$X = \begin{bmatrix} x_1^1 & x_1^2 & \dots & x_1^d \\ x_2^1 & x_2^2 & \dots & x_2^d \\ \vdots & \vdots & \vdots & \vdots \\ x_N^1 & x_N^2 & \dots & x_N^d \end{bmatrix} = \begin{bmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_N^T \end{bmatrix}$$

Compute the mean of each feature, $\bar{x}^i = \frac{1}{N}\sum_{j=1}^{N} x_j^i$, $1 \le i \le d$. Subtracting the mean feature from each feature in the data matrix X, i.e. each x_i^j element in X is replaced by $x_i^j - \bar{x}^j$, we obtain the centered data matrix X_c. The covariance matrix of X_c can be computed as

$$C = \frac{1}{N} X_c^T X_c$$

Performing an eigenvalue decomposition on the covariance matrix, we obtain a set of eigenvalues $\{\lambda_1, \dots, \lambda_d\}$, and a set of d-dimensional normalized (unit) eigenvectors $\{u_1, u_2, \dots, u_d\}$. The eigenvectors are called principal components. These are the directions in the d-dimensional space the original data points x_i's will be projected to in order to get a new set of coordinates. Supposing we only keep a small set of the eigenvalues, $\{\lambda_1, \dots, \lambda_m\}$, and eigenvectors, $\{u_1, u_2, \dots, u_m\}$, $m \ll d$, then the projected data will have a much smaller dimension m. Let X_r be the dimension-reduced data. Then

$$X = \begin{bmatrix} x_1^T u_1 & x_1^T u_2 & \dots & x_1^T u_m \\ x_2^T u_1 & x_2^T u_2 & \dots & x_2^T u_m \\ \vdots & \vdots & \vdots & \vdots \\ x_N^T u_1 & x_N^T u_2 & \dots & x_N^T u_m \end{bmatrix}$$

Now we will see why the algorithm works, i.e. why the algorithm guarantees that the maximum variance is retained for any chosen m, $1 \leq m \leq d$.

Let v_1 be a unit vector in the d-dimensional space, and it is the vector such that when all the data points are projected to it, the resulting 1-dimensional transformed data points has the largest variance. We want to show $v_1 = u_1$, the normalized eigenvector of C corresponding to the largest eigenvalue. To see this, we can compute the mean of the transformed (reduced) data points by:

$$\frac{1}{N} \sum_{i=1}^{N} v_1^T x_i = v_1^T \frac{1}{N} \sum_{i=1}^{N} x_i = v_1^T \bar{x}$$

where $\bar{x}$ is the mean vector of all the data points (row mean of data matrix X). Based on the mean, we can compute the variance of the reduced coordinates as (see Exercise 8-1):

$$\frac{1}{N} \sum_{i=1}^{N} (v_1^T x_i - v_1^T \bar{x})^2 = v_1^T C v_1$$

where C is the data covariance matrix defined before

$$C = \frac{1}{N} X_c^T X_c = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})(x_i - \bar{x})^T$$

Our goal is to maximize $v_1^T C v_1$, with the constraint that $v_1^T v_1 = 1$. Using Lagrange multiplier, we can obtain the following equivalent problem:

$$\arg\max_{v_1} \; v_1^T C v_1 + \lambda(1 - v_1^T v_1)$$

Taking derivative with respect to v_1 and setting it to 0 lead to

$$Cv_1 = \lambda v_1,$$

which shows that v_1 is an eigenvector of C corresponding to the eigenvalue λ. Left multiplying both sides of the previous equation by v_1^T, we have

$$v_1^T C v_1 = \lambda$$

Since $v_1^T C v_1$ is the largest, λ must be equal to λ_1, the largest eigenvalue of C. Therefore, we have $v_1 = u_1$, the eigenvector corresponding to the largest eigenvalue, also called the first principal component.

Using a mathematical induction, we can show the variance of the m-dimensional projected data points is the largest when we use the first m eigenvectors corresponding to the largest m eigenvalues of C.

Exercise 8-1

Show the variance can be computed by

$$\frac{1}{N}\sum_{i=1}^{N}(v_1^T x_i - v_1^T \bar{x})^2 = v_1^T C v_1$$

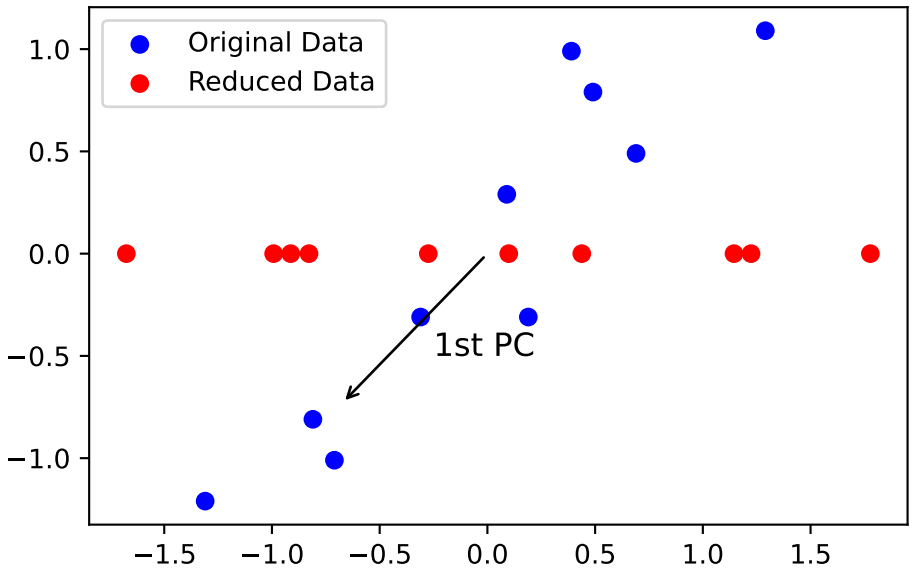
Example 8-1

We consider a simple 2D data that can be transformed to 1D without losing much variance. We use Python to transform the data and show how sklearn can be used to perform PCA.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Synthetic 2D data that is very linear
X = np.array([[2.5, 2.4],
              [0.5, 0.7],
              [2.2, 2.9],
              [1.9, 2.2],
              [3.1, 3.0],
              [2.3, 2.7],
```

```
                [2, 1.6],
                [1, 1.1],
                [1.5, 1.6],
                [1.1, 0.9]])

# mean removal
X_c = X - np.mean(X, axis=0)

# Perform PCA, m=1, i.e. only use the first principal component
pca = PCA(n_components=1)
X_r = pca.fit_transform(X_c)

# Explained variance, i.e. \lambda_1/(\lambda_1+\lambda_2)
# This is also the proportion of variance that is retained.
explained_variance = pca.explained_variance_ratio_

# This is the first principal component
print("Principal Components:\n", pca.components_)
print("Explained Variance Ratio:", explained_variance)

# Plot original and reduced data
plt.scatter(X_c[:, 0], X_c[:, 1], color='blue', label='Original Data')
plt.scatter(X_r[:, 0], np.zeros(X_r.shape), color='red', label='Reduced Data')
# Plot an arrow that shows the principal component direction
plt.annotate('', xy=pca.components_[0], xytext=(0,0),
             arrowprops=dict(facecolor='black', arrowstyle='->'))
plt.text(-0.25, -0.5, '1st PC', fontsize=12, color='k')

plt.legend()
plt.show()
```

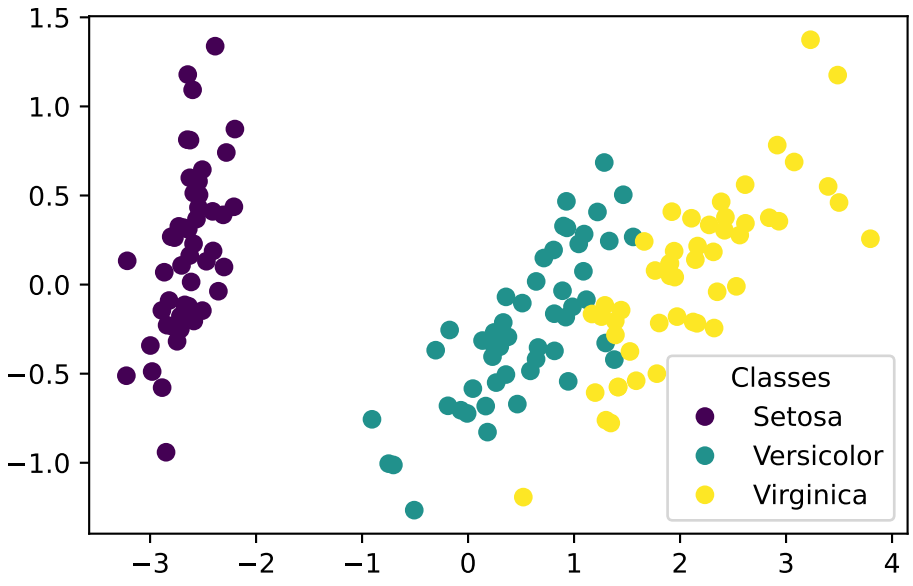
```
Principal Components:
 [[-0.6778734  -0.73517866]]
Explained Variance Ratio: [0.96318131]
```



Example 8-2

Visualize the iris dataset after transforming the data to 2D using PCA.

```
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
X = iris.data  # 4D features
y = iris.target

# Perform PCA to reduce the data from 4D to 2D
pca = PCA(n_components=2)
X_r = pca.fit_transform(X)

scatter = plt.scatter(X_r[:, 0], X_r[:, 1], c=y)
# Create a custom legend
handles, labels = scatter.legend_elements(prop="colors")
legend_labels = ['Setosa', 'Versicolor', 'Virginica']
plt.legend(handles, legend_labels, title="Classes");
```



4.2 Choosing m

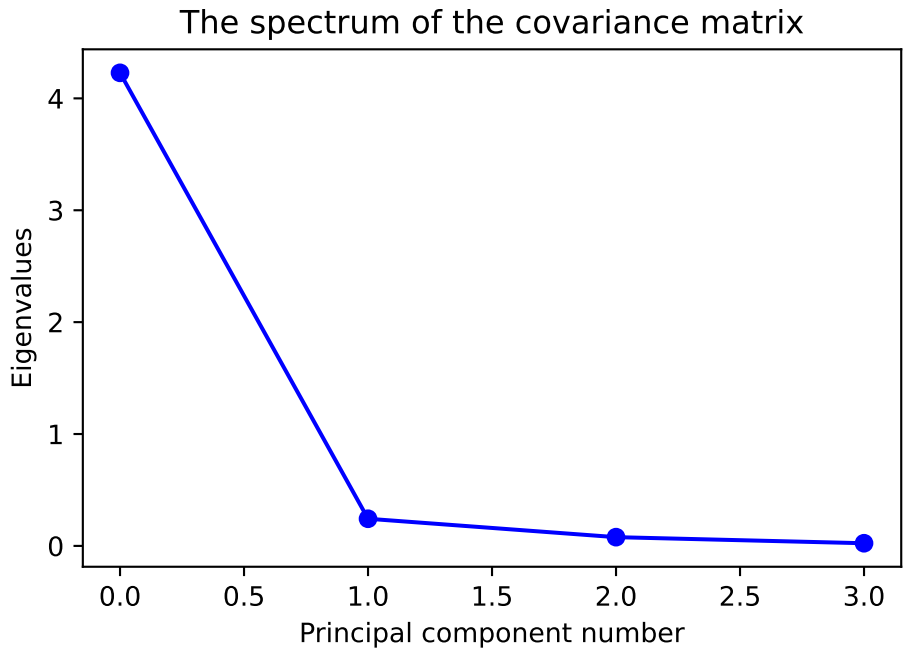
A common way to guide the choice of m is to use the "*cumulative explained variance ratio*". Suppose $\tilde{m}$ principal components are used (hence the reduced data has dimension $\tilde{m}$), then the cumulative explained variance ratio is defined as the sum of the first $\tilde{m}$ largest eigenvalues divided by the sum of all the eigenvalues. This is an indicator of what percentage of variance is retained after transforming the data to a smaller dimension. For a preset threshold, say 90%, we will choose the number of components where the cumulative explained variance ratio first reaches the threshold to be m.

We go back to the previous iris data set example to illustrate the process.

```
# Still for the Iris dataset
# Here we build a PCA without specifying the number of components
# The covariance matrix will be 4 by 4 now
# there will be 4 principal components, and hence 4 eigenvalues
pca = PCA()
pca.fit(X)
# Show the eigenvalues of the covariance matrix
print('The eigenvalues are: ', pca.explained_variance_)
```

```
The eigenvalues are:  [4.22824171 0.24267075 0.0782095  0.02383509]
```

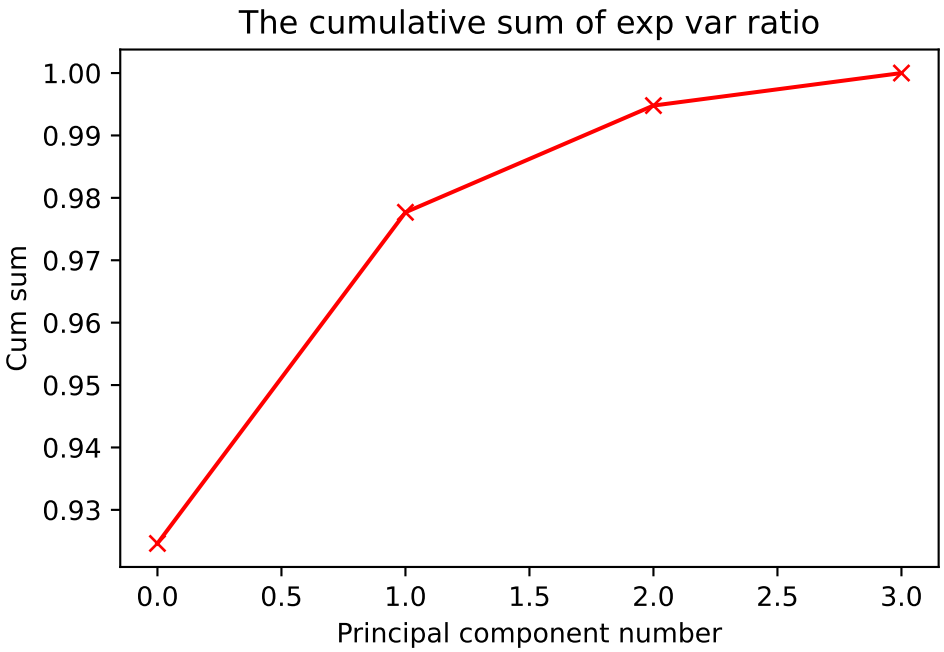
```
# Plot the spectrum of the covariance matrix
plt.plot(pca.explained_variance_, 'bo-')
plt.xlabel('Principal component number')
plt.ylabel('Eigenvalues')
plt.title('The spectrum of the covariance matrix');
```




```
cumsum = np.cumsum(pca.explained_variance_ratio_)
print('Cumulative explained variance ratios are: ', cumsum)
```

```
Cumulative explained variance ratios are:  [0.92461872 0.97768521 0.99478782 1.        ]
```

```
# Plot the cumulative explained variance ratio
plt.plot(cumsum, 'rx-')
plt.xlabel('Principal component number')
plt.ylabel('Cum sum')
plt.title('The cumulative sum of exp var ratio');
```



For example, if our threshold is chosen to be 95%, we would need $m = 2$, i.e., when the 4D data points are transformed to 2D with PCA, less than 5% of the variance is lost. The following line of code will do this automatically:

```
d = np.argmax(cumsum >= 0.95) + 1
print('To retain at least 95% of variance, m should be ', d)
```

```
To retain at least 95% of variance, m should be  2
```

Example 8-3

We consider an application of PCA to the MNIST dataset.

```
from sklearn.datasets import fetch_openml


# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
# Feature matrix:
X = mnist['data']
# Target vector:
y = mnist['target']
```

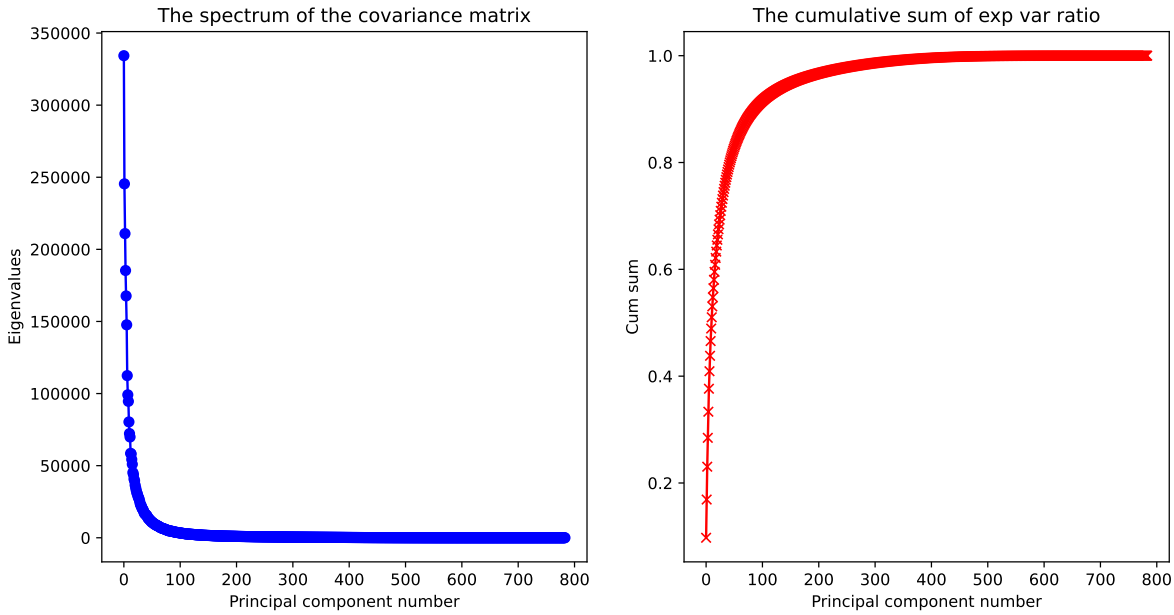
```
print('The shape of X is: ', X.shape)
print('The shape of y is: ', y.shape)
```

```
The shape of X is:  (70000, 784)
The shape of y is:  (70000,)
```

```
# Perform PCA and determine the m that gives us at least 95% variance
pca = PCA()
pca.fit(X)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
print('To retain at least 95% of variance, m should be ', d)
```

```
To retain at least 95% of variance, m should be  154
```

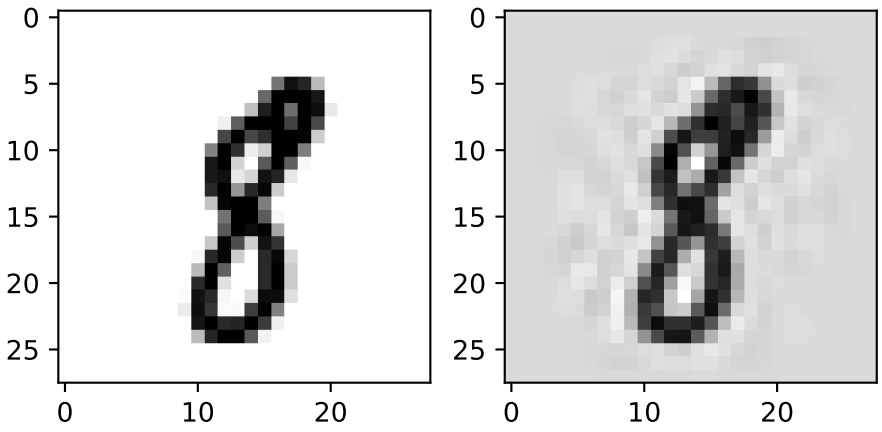
```
plt.figure(figsize=(12, 6))
plt.subplot(1,2,1)
# Plot the spectrum of the covariance matrix
plt.plot(pca.explained_variance_, 'bo-')
plt.xlabel('Principal component number')
plt.ylabel('Eigenvalues')
plt.title('The spectrum of the covariance matrix');
plt.subplot(1,2,2)
# Plot the cumulative sum of the explained variance ratio
plt.plot(cumsum, 'rx-')
plt.xlabel('Principal component number')
plt.ylabel('Cum sum')
plt.title('The cumulative sum of exp var ratio');
```




```
# First compress the data to 154 dimensions
pca = PCA(n_components=154)
X_reduced = pca.fit_transform(X)
# Reconstruct the data back to 784 dimension.
X_recovered = pca.inverse_transform(X_reduced)
print('Shape of the reconstructed data: ', X_recovered.shape)
```

```
Shape of the reconstructed data:  (70000, 784)
```

```
import matplotlib as mpl

# Take an arbitray instance of the original data for plotting
plt.subplot(1,2,1)
digit = X[300, :]
digit_image = digit.reshape(28,28)
plt.imshow(digit_image, cmap = mpl.cm.binary);

# Take an arbitray instance of the recovered data for plotting
plt.subplot(1,2,2)
digit = X_recovered[300, :]
digit_image = digit.reshape(28,28)
plt.imshow(digit_image, cmap = mpl.cm.binary);
```



The two figures, original and reconstructed, are very similar.

4.3 Incremental Principal Component Analysis (IPCA)

When a dataset is large, PCA can be very inefficient, due to the fact that the algorithm requires that all data have to be used at once. In some cases, data come in batches, and we would like the algorithm can handle incremental data, instead of repeating the training on all the data that are currently available. Incremental Principal Component Analysis (IPCA) is a variant of PCA that can handle data in smaller chunks. To achieve this, IPCA iteratively updates the data mean, covariance matrix, and the eigenvalues and eigenvectors through an incremental singular value decomposition (SVD) procedure, as new data arrive.

We use the following example to illustrate how IPCA can be implemented in Python

Example 8-4

Revisit the MNIST data using IPCA.

```
from sklearn.decomposition import IncrementalPCA

# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
# Feature matrix:
X = mnist['data']

# Define the number of principal components
n_components = 50

# Initialize IncrementalPCA
```

```
ipca = IncrementalPCA(n_components=n_components)

# Simulate batch processing
batch_size = 1000
n_batches = X.shape[0] // batch_size

# Process data in batches
for batch_idx in range(n_batches):
    X_batch = X[batch_idx * batch_size:(batch_idx + 1) * batch_size]
    ipca.partial_fit(X_batch)

# Transform the data using the fitted IncrementalPCA
X_ipca = ipca.transform(X)

# Print the cumulative explained variance ratio
print(f'Cumulative explained variance ratio by component: {np.cumsum(ipca.explained_variance_
```

```
Cumulative explained variance ratio by component: [0.09746102 0.16901519 0.23051014 0.284543
 0.40926634 0.43816185 0.46574444 0.48916443 0.51023025 0.53060464
 0.54767341 0.56461214 0.58044427 0.59530583 0.60849752 0.62128546
 0.63315565 0.64468204 0.65533938 0.66543267 0.67501973 0.68411242
 0.69294025 0.70132605 0.70942184 0.7172739  0.72466989 0.73156274
 0.73811856 0.74456287 0.75056215 0.75641254 0.76206722 0.76749188
 0.77251804 0.77737117 0.78214475 0.78680419 0.79133058 0.79574818
 0.79990946 0.80380231 0.80756828 0.81128911 0.81484134 0.81824902
 0.821381   0.824395  ]
```

4.4 Kernel Principal Component Analysis (KPCA)

PCA is effective for data whose features show a linear relationship. For a high-dimensional non-linear feature space (think about spirals or concentric circles), PCA may not perform well (m could be close to d). For non-linear feature space, we may use the idea illustrated in the discussion of kernelized SVM, where we use kernels to implicitly project linearly inseparable data to a higher-dimensional (can be infinitely dimensional) space where they become linearly separable. Similarly, we can borrow the idea and use a kernel to implicitly project a nonlinear feature space to a higher-dimensional one, which hopefully are more linear. We now introduce the main mathematics behind KPCA, which is very similar to that of PCA.

Suppose the implicitly defined mapping from the original feature space (d-dimensional) to a higher-dimensional (D-dimensional, $d \ll D$) one is $\phi(x)$. Then in the higher-dimensional space, the data points become $\{\phi(x_1), \phi(x_2), \ldots, \phi(x_N)\}$. We will then build a data matrix

each row of which represents a data points, and then build a covariance matrix C (assuming the features have a mean of 0 for now):

$$C = \frac{1}{N} \sum_{i=1}^{N} \phi(x_i)\phi(x_i)^T$$

Based on the covariance matrix, we find its eigendecomposition:

$$Cv_i = \lambda_i v_i$$

If we directly work in the D-dimensional feature space, this will be computationally intractable. We will find a way to use the kernel, which computes the inner product in the higher-dimensional space by working in the original space. Note that the previous equation can be rewritten as

$$\frac{1}{N} \sum_{i=1}^{N} \phi(x_i) \left(\phi(x_i)^T v_i\right) = \lambda_i v_i$$

by a substitution of C. if $\lambda_i > 0$ for all i, then we can write the eigenvectors as a linear combination of the features:

$$v_i = \sum_{n=1}^{N} a_{in}\phi(x_n)$$

Plugging the linear combination back into the eigendecomposition equation above, and after some manipulation (see Exercise 8-2), we have

$$\frac{1}{N} \sum_{n=1}^{N} k(x_l, x_n) \sum_{m=1}^{N} a_{im} k(x_n, x_m) = \lambda_i \sum_{n=1}^{N} a_{in} k(x_l, x_n)$$

where k is the kernel function. In matrix notation, this can be written as (see Exercise 8-3)

$$K^2 a_i = \lambda_i N K a_i$$

where $a_i = (a_{i1}, a_{i2}, ..., a_{iN})$ and K is the kernel matrix. The solutions of which can be obtained by solving

$$K a_i = \lambda_i N a_i$$

(The solutions are not exactly the same, but they are the same corresponding to non-zero eigenvalues, which is enough). Here, we have a different normalization condition for a_i (not $a_i^T a_i = 1$). Note that

$$\lambda_i N a_i^T a_i = a_i^T \lambda_i N a_i = a_i^T K a_i = \sum_{n=1}^{N} \sum_{m=1}^{N} a_{in} a_{im} \phi(x_n)^T \phi(x_m) = v_i^T v_i = 1$$

Hence we have the normalization condition for a_i:

$$a_i^T a_i = \frac{1}{\lambda_i N}$$

We can then project the original data x_i onto the principal components in the higher-dimensional feature space to obtain the reduced-dimensional representation by:

$$y_i = \phi(x)^T v_i = \sum_{j=1}^{N} a_{ij} \phi(x)^T \phi(x_j) = \sum_{j=1}^{N} a_{ij} k(x, x_j) \quad 1 \le i \le N$$

In the above discussion, we assumed the projected features have zero means, which is not the case in general. To take this into account, let $\bar{\phi}(x_i)$ be the mean-removed feature, which is obtained by

$$\phi(x_i) = \phi(x_i) - \frac{1}{N} \sum_{j=1}^{N} \phi(x_j)$$

Then the entries of the kernel matrix $\bar{K}$ is calculated by

$$\bar{K}_{ij} = \bar{\phi}(x_i)^T \bar{\phi}(x_j)$$

After some algebra, it can shown that (see Exercise 8-4)

$$\bar{K} = K - 1_N K - K 1_N + 1_N K 1_N$$

where 1_N is a $N \times N$ matrix whose entries are all $1/N$

Exercise 8-2

Derive the equation

$$\frac{1}{N} \sum_{n=1}^{N} k(x_l, x_n) \sum_{m=1}^{N} a_{im} k(x_n, x_m) = \lambda_i \sum_{n=1}^{N} a_{in} k(x_l, x_n)$$

Exercise 8-3

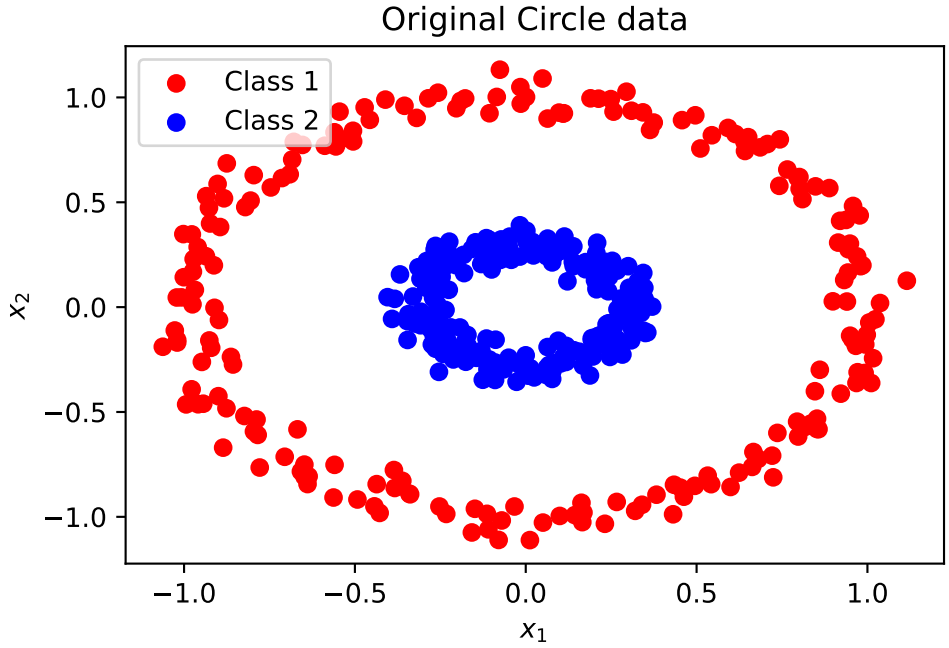
Show the covariance matrix in the higher dimension can be written as

$$C^2 a_i = \lambda_i N C a_i$$

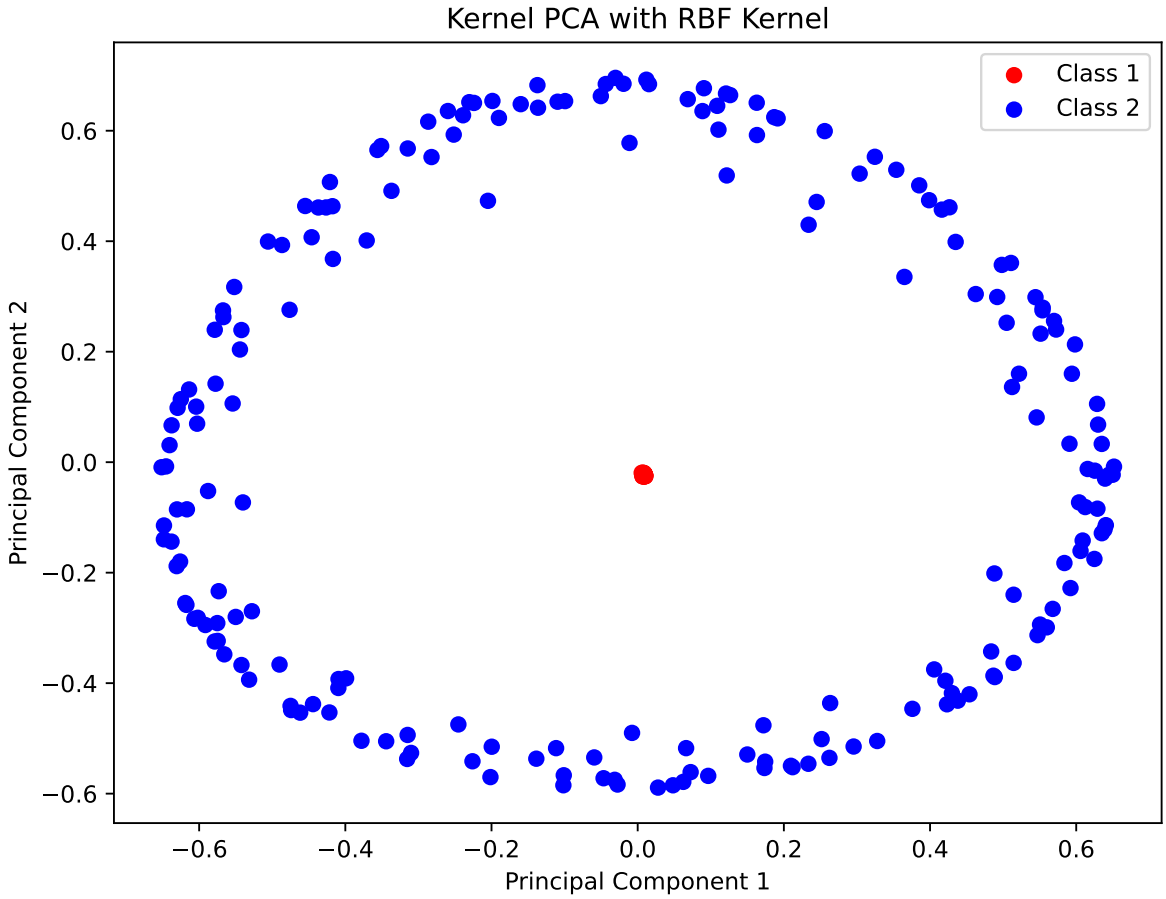
Example 8-5

Perform KPCA on the concentric circles data.

```
from sklearn.decomposition import KernelPCA
from sklearn.datasets import make_circles

# Generate non-linear data (e.g., concentric circles)
X, y = make_circles(n_samples=400, factor=0.3, noise=0.05, random_state=32)

# Plot the circle
plt.scatter(X[y == 0, 0], X[y == 0, 1], color='red', label='Class 1')
plt.scatter(X[y == 1, 0], X[y == 1, 1], color='blue', label='Class 2')
plt.title('Original Circle data')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.legend()
plt.show()
```

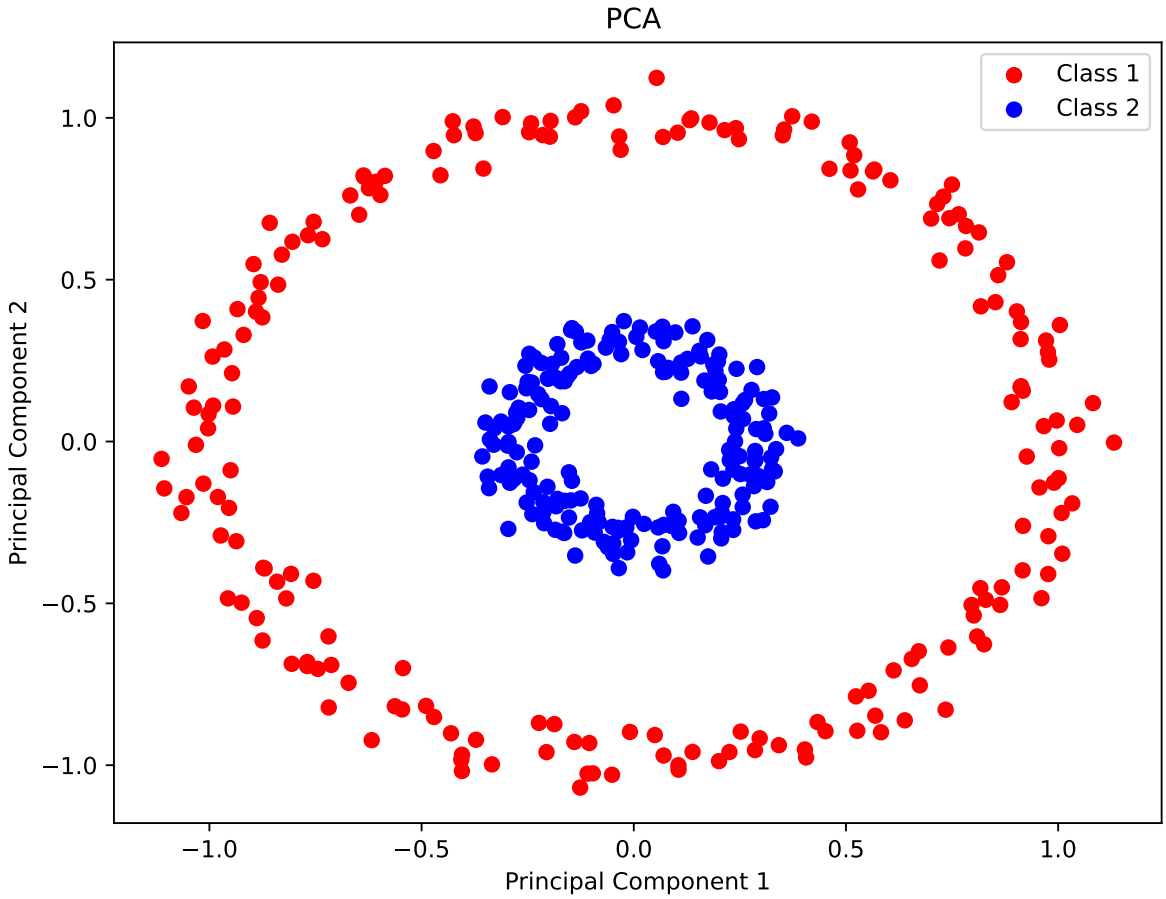


```
# Perform Kernel PCA
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_kpca = kpca.fit_transform(X)

# Plot the results
plt.figure(figsize=(8, 6))
plt.scatter(X_kpca[y == 0, 0], X_kpca[y == 0, 1], color='red', label='Class 1')
plt.scatter(X_kpca[y == 1, 0], X_kpca[y == 1, 1], color='blue', label='Class 2')
plt.title('Kernel PCA with RBF Kernel')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()
```



If we perform PCA on the concentric circle data, what will happen?

```
# Perform PCA with two components
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Plot the results
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[y == 0, 0], X_pca[y == 0, 1], color='red', label='Class 1')
plt.scatter(X_pca[y == 1, 0], X_pca[y == 1, 1], color='blue', label='Class 2')
plt.title('PCA ')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()
```

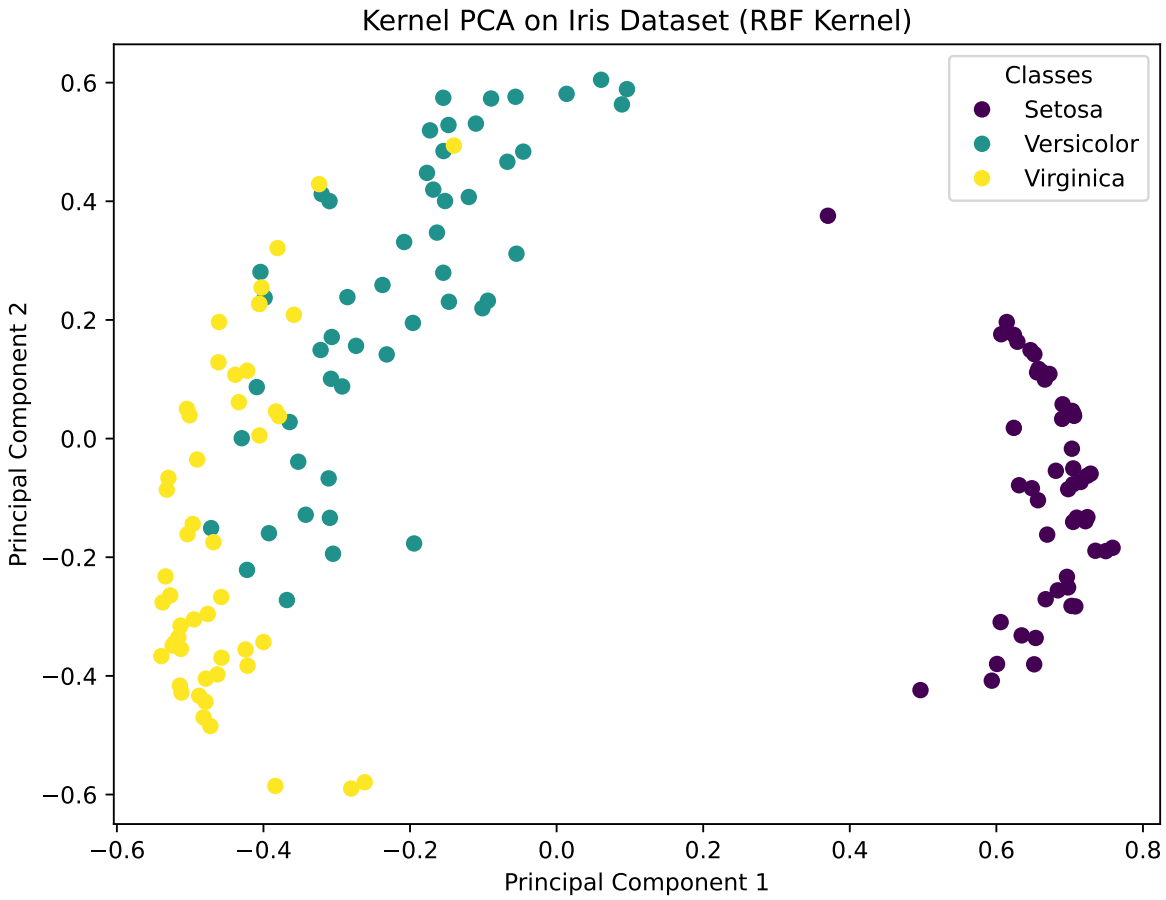


The shape of the new coordinates are very similar to the original ones.

Example 8-6

Perform KPCA for the iris dataset to reduce the dimensions of the feature space to 2.

```
from sklearn.preprocessing import StandardScaler


# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names

# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply Kernel PCA with RBF kernel
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=0.1)
X_kpca = kpca.fit_transform(X_scaled)

# Plot the results of Kernel PCA
plt.figure(figsize=(8, 6))
scatter = plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=y)
# Create a custom legend
handles, labels = scatter.legend_elements(prop="colors")
legend_labels = ['Setosa', 'Versicolor', 'Virginica']
plt.legend(handles, legend_labels, title="Classes");
plt.title('Kernel PCA on Iris Dataset (RBF Kernel)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()
```

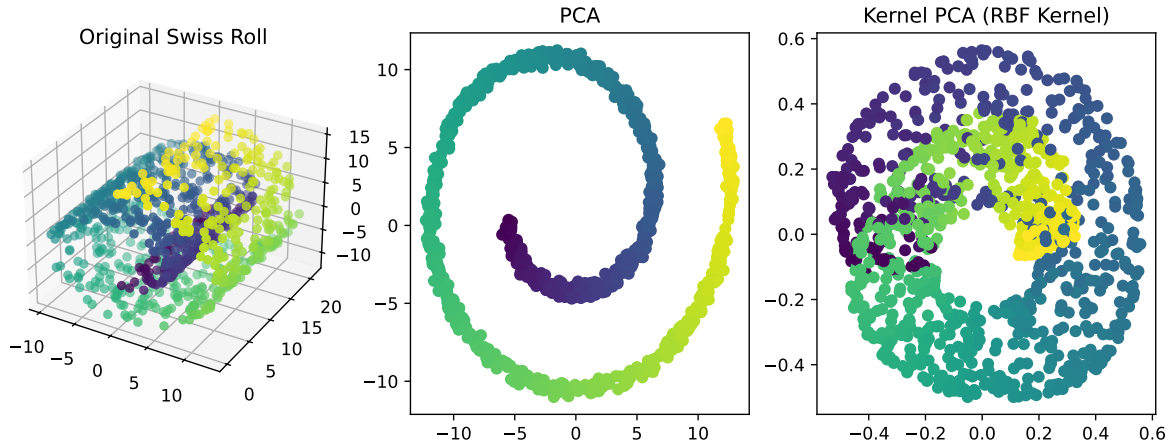


Example 8-7

Apply and compare PCA and KPCA applied to the Swiss Roll Dataset.

```
from sklearn.datasets import make_swiss_roll

# Generate the Swiss roll dataset
X, color = make_swiss_roll(n_samples=1000, noise=0.2, random_state=24)

# Apply standard PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Apply Kernel PCA with RBF kernel
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=0.02)
X_kpca = kpca.fit_transform(X)
```

```
# Plot the original Swiss roll in 3D
fig = plt.figure(figsize=(12, 4))
ax = fig.add_subplot(131, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color)
ax.set_title("Original Swiss Roll")

# Plot the results of standard PCA
ax = fig.add_subplot(132)
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=color)
plt.title("PCA")

# Plot the results of Kernel PCA
ax = fig.add_subplot(133)
plt.scatter(X_kpca[:, 0], X_kpca[:, 1], c=color)
plt.title("Kernel PCA (RBF Kernel)")

plt.show()
```



The 2D projection from standard PCA flattens the Swiss roll, but it does not effectively capture the underlying spiral structure. Points that were far apart on the roll may end up close together in the 2D projection. On the other hand, Kernel PCA successfully unrolls the Swiss roll, revealing a structure that better preserves the original relationships between points. The colors (representing positions along the roll) form a smooth gradient, indicating that the non-linear structure has been effectively captured.

5 Neural Networks

A neural network (NN) is a machine learning model that mimics the way biological neural networks in the human brain process information. These networks have various degrees of complexity, and are able to handle difficult problems, such as natural language processing and image recognition. A prototype of NN is shown in the following figure,

The circles are called **neurons** (also called nodes), which are the basic units of the NN. They are arranged in layers and here there are three layers: the *input layer*: features are passed into this layer; the output layer: the predicted quantities (discrete or continuous) come from this layer; and the hidden layer: any layers between the input and output layers, main computations are performed in these layers. Each neuron in the hidden layer in this structure is *fully connected* (or *dense*) with the input neurons (i.e., connected with all the neurons in the previous layer). The arrows represent the connections, and a weight is associates with each connection. The value of each neuron in the hidden layer is a nonlinear function (called **activation function**) of the weighted sum (with a constant term called **bias**) of the values of the input neurons. The output layer follows a similar pattern. Some commonly ued activation functions are:

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Rectified Linear Unit (ReLU): $\text{ReLU}(x) = \max(0, x)$
- hyperbolic tangent (Tanh): $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Depending on the structure of the NN, it can be as simple as a linear model, or can have millions or more hyperparameters that take days to be trained. We start from the simplest versions of NN and gradually move to more complex ones in later chapters.

5.1 The Perceptron

A perceptron has only an input layer and an output layer. As the figure shown below, we consider a perceptron with only one neuron in the output layer. The output neuron takes the weighted sum of the inputs ($w_1x_1 + \cdots + w_mx_m + b$), and applies an activation function Γ, usually the *Heaviside step function*, to the weighted sum to generate an output y:

$$y = \Gamma(w_1x_1 + \cdots + w_mx_m + b)$$

where

$$\Gamma(x) = \begin{cases} 1 & \text{if } x \geq 1 \\ 0 & \text{if } x < 1 \end{cases}$$

If the activation function Γ is taken to be the identity (linear activation) function $\Gamma(x) = x$, then perceptron is reduced to a linear model. Perceptron is mainly used for binary classification. Training a perceptron model involves finding the optimal weights and biases. The choice of the activation function also matters. The We now use an example to illustrate how to implement perceptron in Python.

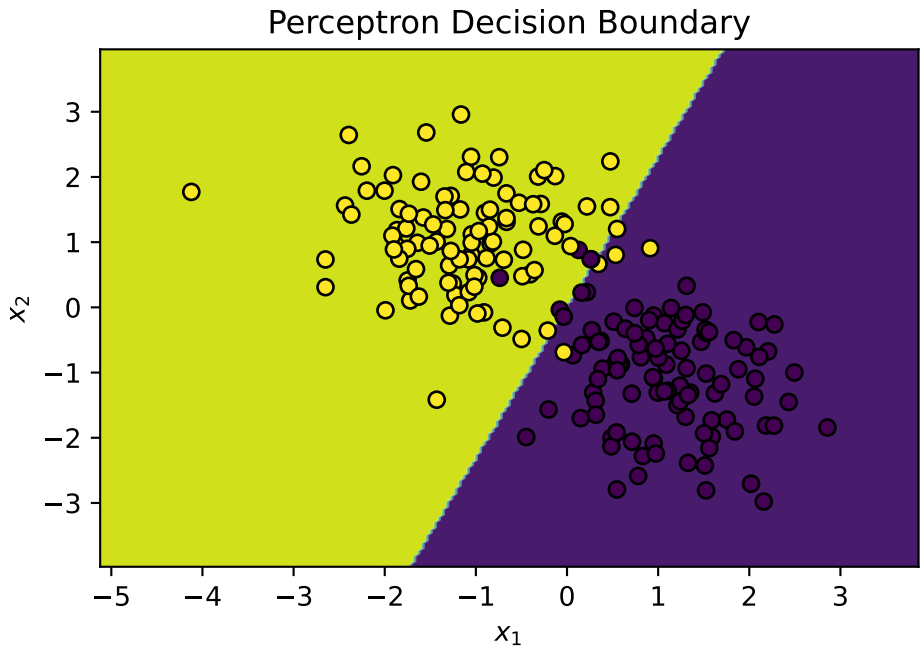
Example 10-1

Use perceptron for a random 2-class classification problem where the data are generated by sklearn.datasets.make_classification.

```
import numpy as np
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification

# Generate a synthetic dataset
X, y = make_classification(n_samples=200, n_features=2, n_informative=2, n_redundant=0,
                           n_clusters_per_class=1, n_classes=2, random_state=50)

# Initialize and train the perceptron model
per_clf = Perceptron(max_iter=1000, tol=1e-3, random_state=50)
per_clf.fit(X, y)

# Make predictions
y_pred = per_clf.predict(X)
```

```
# Create a mesh grid for plotting the decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
                     np.linspace(y_min, y_max, 200))

# Predict over the mesh grid
Z = per_clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```

```
# Plot the decision boundary
plt.contourf(xx, yy, Z)

# Plot the original data points
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')

# Add labels and title
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Perceptron Decision Boundary');
```



5.2 Multilayer Perceptron (MLP)

If we add one or more hidden layers to the perceptron structure, we obtain a **multilayer perceptron** (see the first figure above). With a more complex structure, MLP is capable of solving more difficult problems for both classification and regression. When the number of hidden layers is large, the model is called a **deep neural network (DNN)**. Using DNNs to perform machine learning tasks is called **deep learning** (although there is no agreement on how many hidden layers is deep). The input layer takes the input features, each node of each hidden layer takes the weighted sum of the nodes from the previous layer and applies an

activation function, and the output layer produces the output. There can be a single node in the output layer if the problem is a regression of binary classification. There can be more nodes for multi-class classification problems. For a multi-class classification problem, the activation function for the output layer is usually the **softa** function for a K-class classification problem is:

$$\Gamma(x) = \Gamma(x_1, x_2, \ldots, x_K) = \left(\frac{e^{x_1}}{\sum_{j=1}^{K} e^{x_j}}, \frac{e^{x_2}}{\sum_{j=1}^{K} e^{x_j}}, \ldots, \frac{e^{x_K}}{\sum_{j=1}^{K} e^{x_j}} \right) \in (0,1)^K$$

The training of an MLP involves finding the optimal weights and biases based on a selected loss function. The MSE function is natural for regression, and cross-entropy is appropriate for classification. To find the best hyperparameters, initial values (guesses) of the hyperparameters need to be first provided. Then the input layer receives the input features, and each node in the hidden layer closest to the input layer receives the weighted sum of the input nodes and applies the activation function. The subsequent hidden layers and the output layer follow the same process, and finally an output is produced corresponding to the initial hyperparameter values, and the loss function is evaluated at the output. This stage is called **forward propagation**. The second stage involves a process that reverses the previous one, called **back propagation**. Here, the gradient of the loss function with respect to the hyperparameters (weights and biases) are computed. With the gradient information, an efficient optimization algorithm such as *stochastic gradient descent (SGD)* can be used to search for the optimal hyperparameters. The core of the training of MLP is the back propagation. We use examples to illustrate how it works.

Example 10-2

In the simple MLP for regression below, we have initialized all the weights and biases, and the input values for Nodes I_1 and I_2 are given. For both the hidden and output layers, suppose we use the sigmoid activation. Also assume the true target values are 0.01 and 0.99 for Nodes O_1 and O_2, respectively. Perform a forward pass and evaluate the value of the cost function.

Calculate the weighted sum of the inputs I_1 and I_2 for H_1, called the **net input** of H_1:

$$\begin{aligned} \text{net}_{H_1} &= w_1 \cdot i_1 + w_2 \cdot i_2 + b_1 \\ \text{net}_{H_1} &= 0.15 \cdot 0.05 + 0.2 \cdot 0.1 + 0.35 = 0.3775 \end{aligned}$$

Use the sigmoid function to get the output of H_1:

$$\text{out}_{H_1} = \frac{1}{1 + e^{-\text{net}_{H_1}}} = \frac{1}{1 + e^{-0.3775}} = 0.593269992$$

Carrying out the same process for H_2:

$$\text{out}_{H_2} = 0.596884378$$

Repeat this process for the nodes in the output layer, using the output from the hidden layer nodes as inputs.

For the output of O_1, we have:

$$\begin{aligned}
\text{net}_{O_1} &= w_5 \cdot \text{out}_{H_1} + w_6 \cdot \text{out}_{H_2} + b_3 \\
\text{net}_{O_1} &= 0.4 \cdot 0.593269992 + 0.45 \cdot 0.596884378 + 0.6 = 1.105905967 \\
\text{out}_{O_1} &= \frac{1}{1 + e^{-\text{net}_{O_1}}} = \frac{1}{1 + e^{-1.105905967}} = 0.75136507
\end{aligned}$$

Similarly, for O_2 we get:

$$\text{out}_{O_2} = 0.772928465$$

We can now calculate the error for each output node using the MSE function and sum them to get the total error:

$$E_{\text{total}} = \sum \frac{1}{2}(\text{target} - \text{output})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 + \frac{1}{2}(0.99 - 0.772928465)^2 = 0.298371109$$

Example 10-3

Now we perform the back propagation process.

Output layer: Consider w_5. We want to know how much a change in w_5 affects the total error, i.e., $\frac{\partial \text{E}_{\text{total}}}{\partial w_5}$. Applying the chain rule, we know that

$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{O_1}} \cdot \frac{\partial \text{out}_{O_1}}{\partial \text{net}_{O_1}} \cdot \frac{\partial \text{out}_{O_1}}{\partial w_5}$$

We now find each piece in this equation:

$$\begin{aligned}
E_{\text{total}} &= \frac{1}{2}\left(\text{target}_{o1} - \text{ out }_{o1}\right)^2 + \frac{1}{2}\left(\text{target}_{o2} - \text{out }_{o2}\right)^2 \\
\frac{\partial E_{\text{total}}}{\partial \text{out}_{O_1}} &= -\left(\text{target}_{O_1} - \text{out}_{O_1}\right) = -(0.01 - 0.75136507) = 0.74136507
\end{aligned}$$

$$\begin{aligned}
\text{out }_{o1} &= \tfrac{1}{1+e^{-net_{o1}}} \\
\tfrac{\partial \text{ out }_{o1}}{\partial net_{o1}} &= \text{out}_{o1}\left(1 - \text{ out }_{o1}\right) = 0.75136507(1 - 0.75136507) = 0.186815602
\end{aligned}$$

$$\begin{aligned}
\text{net }_{o1} &= w_5 \cdot \text{ out }_{h1} + w_6 \cdot \text{ out }_{h2} + b_3 \\
\tfrac{\partial \text{net}_{O_1}}{\partial w_5} &= \text{ out }_{h1} = 0.593269992
\end{aligned}$$

Putting it all together:

$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{O_1}} \cdot \frac{\partial \text{out}_{O_1}}{\partial \text{net}_{O_1}} \cdot \frac{\partial \text{net}_{O_1}}{\partial w_5}$$
$$\frac{\partial E_{\text{total}}}{\partial w_5} = 0.74136507 \cdot 0.186815602 \cdot 0.593269992 = 0.082167041$$

Similarly, we can obtain $\frac{\partial \text{E}_{\text{total}}}{\partial w_6}$, $\frac{\partial \text{E}_{\text{total}}}{\partial w_7}$, $\frac{\partial \text{E}_{\text{total}}}{\partial w_8}$.

Now we deal with the hidden layer. First we find $\frac{\partial \text{E}_{\text{total}}}{\partial w_1}$ by

$$\frac{\partial E_{\text{total}}}{\partial w_1} = \frac{\partial E_{\text{total}}}{\partial \text{out}_{H_1}} \cdot \frac{\partial \text{out}_{H_1}}{\partial \text{net}_{H_1}} \cdot \frac{\partial \text{net}_{H_1}}{\partial w_1}$$

Note out_{H_1} affects both out_{O_1} and out_{O_2}. So

$$\frac{\partial E_{\text{total}}}{\partial \text{out}_{H_1}} = \frac{\partial E_{O_1}}{\partial \text{out}_{H_1}} + \frac{\partial E_{O_2}}{\partial \text{out}_{H_1}}$$

and

$$\frac{\partial E_{O_1}}{\partial \text{out}_{H_1}} = \frac{\partial E_{O_1}}{\partial \text{net}_{O_1}} \cdot \frac{\partial \text{net}_{O_1}}{\partial \text{out}_{H_1}} = \frac{\partial E_{O_1}}{\partial \text{out}_{O_1}} \cdot \frac{\partial \text{out}_{O_1}}{\partial \text{net}_{O_1}} \cdot \frac{\partial \text{net}_{O_1}}{\partial \text{out}_{H_1}} = 0.74136507 \cdot 0.186815602 \cdot w_5 = 0.055399425$$

Similarly, all the other quantities can be obtained. Once all the partial derivatives are obtained, SGD or other derivative-based optimization algorithm can be applied.

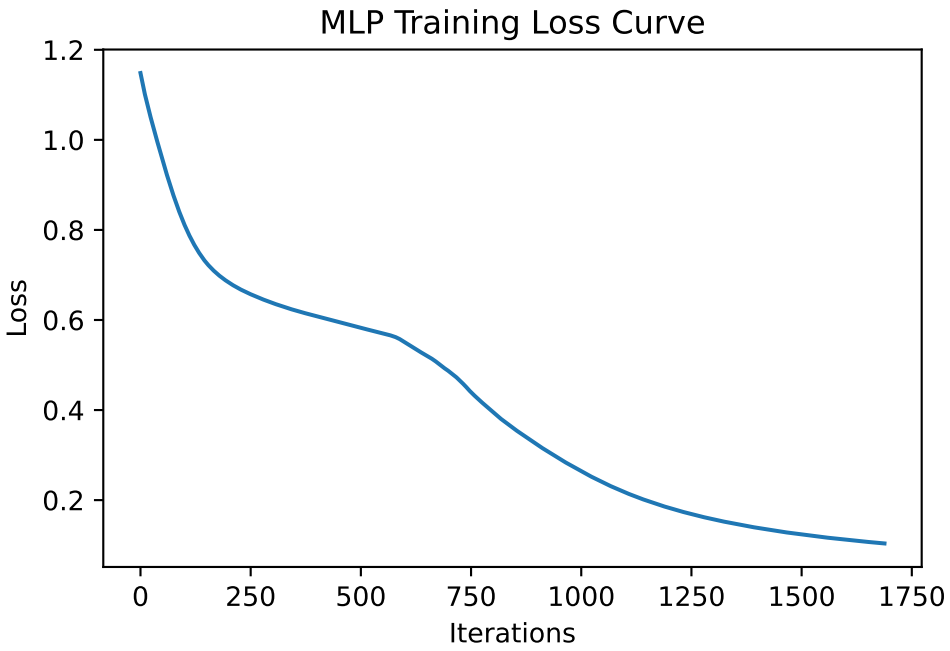
Example 10-4

We perform MLP for the iris data set. Here we use two hidden layers, each with 10 neurons.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix

# Load the Iris dataset
iris = load_iris()
X = iris.data
```

```
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=96)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize and train the MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(12, 6), max_iter=5000, random_state=86)
mlp.fit(X_train, y_train)

# Make predictions
y_pred = mlp.predict(X_test)

# Evaluate the model
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

# Plotting the loss curve. The loss function continues to drop as the model is being trained
plt.plot(mlp.loss_curve_)
plt.title('MLP Training Loss Curve')
plt.xlabel('Iterations')
plt.ylabel('Loss');
```

```
Confusion Matrix:
 [[20  0  0]
 [ 0 14  0]
 [ 0  0 11]]
```



We can check the optimal weights and biases of the model by:

```
# See the weight matrix
mlp.coefs_
```

```
[array([[-0.38639999, -0.47921984, -0.85369411, -0.09692643, -0.10745231,
          0.6204933 , -0.21007314, -0.42818818,  0.96059093, -0.33130668,
         -0.53353584,  0.34266489],
        [ 0.46759207,  0.43518618,  0.67255324, -0.64774433, -0.4716051 ,
         -0.47048044, -0.29232437,  0.65049607, -0.56107454,  0.57628054,
          0.43831958,  0.02490162],
        [-0.28121689, -0.19595681, -0.89872946,  1.24731719,  0.53192312,
         -1.76467841,  0.77030492, -0.27495727, -0.99168277, -0.79445805,
         -0.38833726, -0.01354722],
        [ 0.33654925,  0.20413659, -0.0604337 ,  0.53313068,  0.5098916 ,
         -0.95091831,  1.26242908, -0.5976721 ,  0.17004848, -0.59130148,
         -0.80525132,  0.21632437]]),
 array([[-5.22115239e-01, -1.01460772e-05,  6.63854561e-01,
         -5.58638660e-01, -1.17290631e-01, -2.67314255e-01],
        [-1.70766217e-01,  3.03615582e-06,  5.79073512e-01,
         -4.28465918e-02,  4.48677979e-01, -3.98988864e-01],
        [ 3.98023541e-01, -9.84730334e-07, -2.61874469e-01,
         -2.00233714e-10,  8.13262572e-01, -2.68644639e-01],
```

```
        [-3.08437189e-01, -3.39837556e-33, -1.15359523e+00,
         -1.46528864e-01, -7.39587434e-01, -2.04793275e-04],
        [-3.20656875e-02,  1.11437745e-20,  2.43235781e-01,
         -1.88461832e-01, -1.36292845e+00, -1.62083433e-01],
        [-6.06505850e-02, -5.57667625e-43,  1.92121725e+00,
         -2.66795860e-01,  1.63185967e+00, -1.06931924e-02],
        [ 4.66871444e-01, -2.90609369e-05, -1.57047082e+00,
         -4.92826853e-01,  2.43951395e-01, -2.49797927e-04],
        [-5.74238448e-05, -9.99580221e-05,  6.19060496e-01,
          1.63528938e-01,  1.15051811e+00,  1.15172677e-01],
        [-4.01003452e-01,  2.67118236e-13,  1.79810058e+00,
          9.06125332e-02, -3.17692857e-01,  1.49533069e-06],
        [-1.90776811e-16,  2.82001457e-16,  6.60208208e-01,
         -1.05162652e-10,  1.11280488e+00,  7.64862740e-02],
        [-3.90733121e-01, -4.16138798e-04,  6.93691343e-01,
          1.39816374e-39,  5.97359729e-01,  3.77181784e-01],
        [-5.27576431e-01, -4.57465937e-26, -1.34915774e-01,
          4.11310602e-01, -6.20370127e-02,  1.09531065e-01]]),
 array([[ 6.57551335e-01,  6.76243565e-02, -5.44790501e-01],
        [ 3.26215186e-07, -4.76841489e-04,  3.17413031e-05],
        [ 1.74927927e-01,  1.01649310e+00, -1.90269507e+00],
        [ 4.04607609e-01, -3.64797025e-01, -4.56734652e-01],
        [ 7.86811483e-01, -4.27109606e-01,  1.51893780e-02],
        [ 6.52852896e-02,  6.91145457e-01,  6.19429767e-01]])]
```

The first weight matrix is 4×12, since we have four input neurons and 12 neurons in the first hidden layer. For example, the first row represents the weight from the first neuron of the input layer to all 12 neurons in the first hidden layer. For the biases, simply call:

```
# Bias:
mlp.intercepts_
```

```
[array([ 1.064518  ,  1.00025039, -0.1233446 , -0.28436637,  1.38207967,
         0.95564239, -0.32248468,  0.68590364,  0.24189182,  0.01015541,
         0.90629805,  0.16900112]),
 array([ 0.30398812, -0.26807586,  0.58656479, -0.24204741, -0.25822549,
        -0.56513263]),
 array([-1.71106822, -0.53736298,  1.27126756])]
```

For example, the first array of 12 elements represents the bias for the 12 neurons in the first hidden layer.

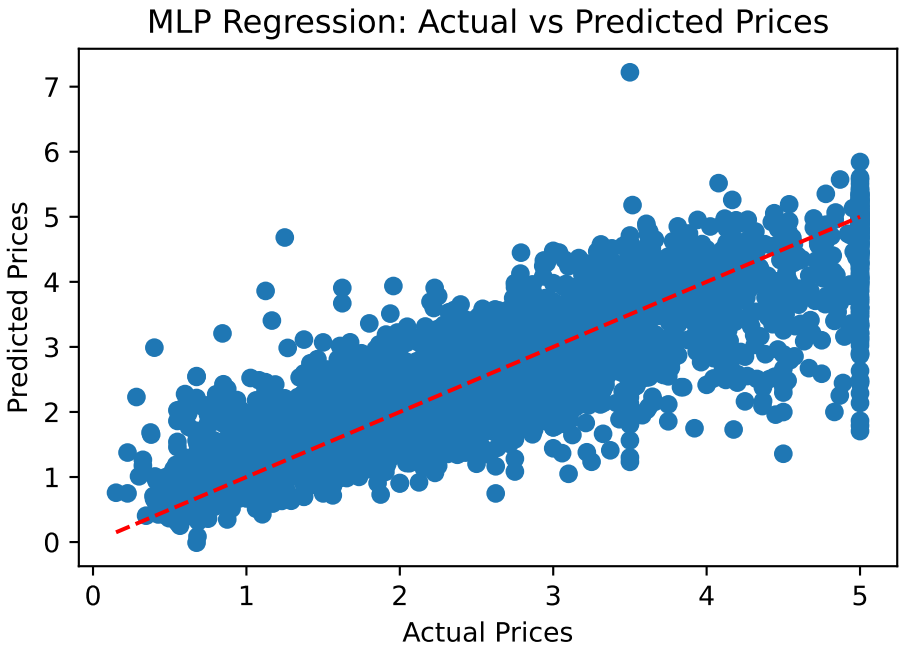
Example 10-5

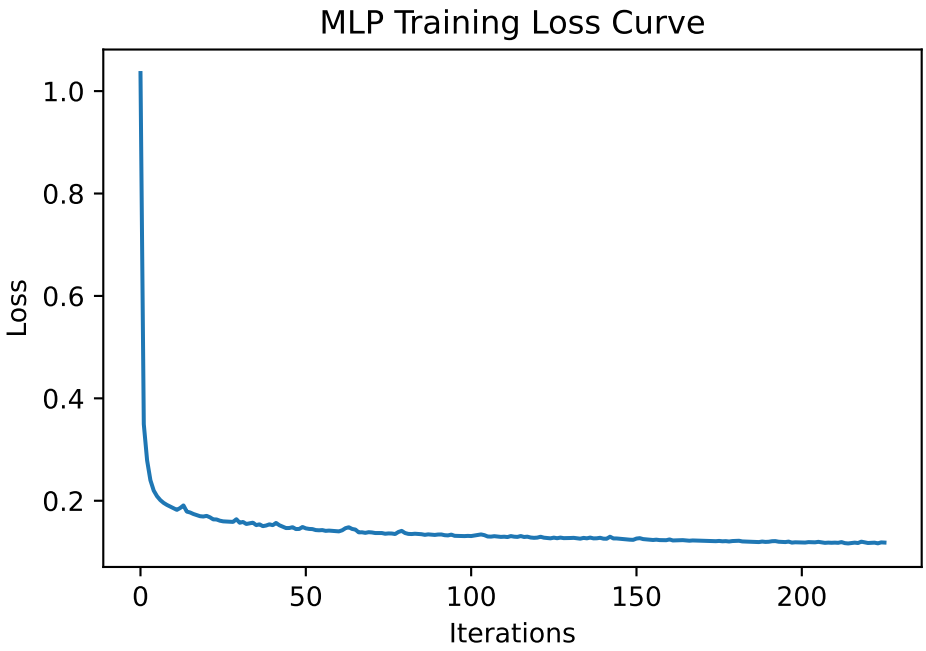
Apply MLP to the california housing dataset. Use two hidden layers each with 50 neurons.

```
from sklearn.datasets import fetch_california_housing
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error, r2_score

# Load the Boston Housing dataset
data = fetch_california_housing()
X = data.data
y = data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=22)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize and train the MLPRegressor
mlp = MLPRegressor(hidden_layer_sizes=(50, 50), max_iter=1000, random_state=77)
mlp.fit(X_train, y_train)

# Make predictions
y_pred = mlp.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Mean Squared Error: {mse:.2f}")
print(f"R^2 Score: {r2:.2f}")

# Plot the predictions vs actual values
plt.scatter(y_test, y_pred)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'r--')
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.title('MLP Regression: Actual vs Predicted Prices')
plt.show()
```

```
# Plot the loss curve
plt.plot(mlp.loss_curve_)
plt.title('MLP Training Loss Curve')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.show()
```

```
Mean Squared Error: 0.26
R^2 Score: 0.80
```




MLP Training Loss Curve
Loss
1.0
0.8
0.6
0.4
0.2
0
50
100
150
200
Iterations