7 Selected supervised learning algorithms

This chapter covers

- Markov models: page rank and HMM
- Imbalanced learning, including undersampling and oversampling strategies
- Active learning, including uncertainty sampling and query by committee strategies
- Model selection, including hyperparameter tuning
- Ensemble methods, including bagging, boosting, and stacking
- ML research, including supervised learning algorithms

In the previous two chapters, we looked at supervised algorithms for classification and regression. In this chapter, we focus on a selected set of supervised learning algorithms. The algorithms are selected to give exposure to a variety of applications—from time series models used in computational finance to imbalanced learning used in fraud detection to active learning used to reduce the number of training labels to model selection and ensemble methods used in all data science competitions. Finally, we conclude with ML research and exercises. Let's begin by reviewing the fundamentals of Markov models.

7.1 Markov models

In this section, we discuss probabilistic models for a sequence of observations. Time series models have a wide range of applications, including in computational finance, speech recognition, and computational biology. We'll start by looking at two popular algorithms built upon the properties of Markov chains: the page rank algorithm and the EM algorithm for hidden Markov models (HMMs).

However, before we dive into individual algorithms, let's start with the fundamentals. A Markov model for a sequence of random variables $x_1, ..., x_T$ of order 1 is a joint probability model that can be factorized as follows.

$$\begin{aligned} p(x_1, ..., x_T) &= p(x_1)p(x_2|x_1)\cdots p(x_T|x_{T-1}) \\ &= p(x_1)\prod_{t=2}^{T} p(x_t|x_{t-1}) \end{aligned} \tag{7.1}$$

Notice that each factor is a conditional distribution conditioned by 1 random variable (i.e., each factor only depends on the previous state and, therefore, has a memory of 1). We can also say that X_{t-1} serves as a sufficient statistic for X_t. A *sufficient statistic* is a function of sample data (e.g., a summary of the data, such as a sum or a mean) that contains all the information needed to estimate an unknown parameter in a statistical model.

Let's look at the transition probability $p(x_t|x_{t-1}))$ in more detail. For a discrete state sequence $x_t \in \{1, ..., K\}$, we can represent the transition probability as a $K \times K$ stochastic matrix (in which the rows sum to 1).

$$A_{ij} = p(X_t = j|X_{t-1} = i), \text{ where } \sum_j A_{ij} = 1 \ \forall i \tag{7.2}$$

Figure 7.1 shows a simple Markov model with two states.

Here, α is the transition probability out of state 1 and $1 - \alpha$ is the probability of staying in state 1. Notice how the transition probabilities out of each state add up to 1. We can write down the corresponding transition probability matrix as follows.

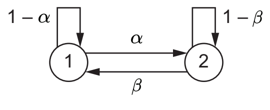

Figure 7.1 A two-state Markov model

$$A = \begin{pmatrix} 1-\alpha & \alpha \\ \beta & 1-\beta \end{pmatrix} \tag{7.3}$$

If α and β do not vary over time; in other words, if the transition matrix A is independent of time, we call the Markov chain *stationary* or *time invariant*.

We are often interested in the long-term behavior of the Markov chain—namely, a distribution over states based on the frequency of visits to each state as time passes. Such distribution is known as *stationary distribution*. Let's compute it! Let π_0 be the initial distribution over the states. Then, after the first transition, we have the following.

$$\pi_1(j) = \sum_i \pi_0(i) A_{ij} \tag{7.4}$$

Or in matrix notation, we have the following.

$$\pi_1 = \pi_0 A \tag{7.5}$$

We can keep going and write down the second transition as follows.

$$\pi_2 = \pi_1 A = \pi_0 A^2 \tag{7.6}$$

We see that raising the transition matrix A to a power of n is equivalent to modeling n hops (or transitions) of the Markov chain. After some time, we reach a state when left multiplying the row state vector π by the matrix A gives us the same vector π.

$$\pi = \pi A \tag{7.7}$$

In the preceding case, we found the stationary distribution; it is equal to π, which is an eigenvector of A that corresponds to the eigenvalue of 1. We will also state here (with proof left as an exercise) that a stationary distribution exists if and only if the chain is *recurrent* (i.e., it can return to any state with probability 1) and *aperiodic* (i.e., it doesn't oscillate).

7.1.1 Page rank algorithm

Google uses a page rank algorithm to order search results from millions of web pages. We can formulate a collection of n web pages as a graph $G = (V, E)$. Let every node $v \in V$ represent a web page, and let every edge $e \in E$ represent a directed link from one page to another. Then, G is a sparse graph with $|V| = n$. Intuitively, a web page that receives a lot of incoming links from reputable sources should be promoted toward the top of the ranked list. Knowing how the pages are linked allows us to construct a giant transition matrix A, where A_{ij} is the probability of following a link from page i to page j. Given this formulation, we can interpret the entry π_j as the importance or rank of page j. Considering the Markov chain discussion in the previous section, the page rank is the stationary distribution π (i.e., the eigenvector of A corresponding to the eigenvalue of 1).

$$\pi_j = \sum_i A_{ij} \pi_i \tag{7.8}$$

For the stationary distribution π (i.e., page rank) to exist, we need to guarantee that the Markov chain described by the transition matrix is recurrent and aperiodic. Let's look at how we can construct such transition matrix. Let's model the interaction with web pages as follows. If we have outgoing links for a specific page, then with probability $p > 0.5$ (p can be chosen using a simulation), we jump to one of the outgoing links (decided uniformly at random), and with probability $1-p$, we open a new page (also uniformly at random). If there are no outgoing links, we open a new page (decided uniformly at random). These two conditions ensure the Markov chain is recurrent and aperiodic, since random jumps can include self-transitions; thus, every state is reachable from every other state. Let G_{ij} be the adjacency matrix and $d_j = \sum_i G_{ij}$ represent the out-degree of page j. Then, if the out-degree is 0, we jump to a random page with probability $1/n$. With probability p we follow a link with probability equal to the out-degree $1/d_j$, and with probability $1-p$, we jump to a random page. Let's summarize our transition matrix in the following equation.

$$A_{ij} = \begin{cases} \frac{pG_{ij}}{d_j} + \frac{1-p}{n} & \text{if } d_j \neq 0 \\ \frac{1}{n} & \text{if } d_j = 0 \end{cases} \tag{7.9}$$

However, how do we find the eigenvector π given the enormous size of our transition matrix A? We will use an iterative algorithm called the *power method* for computing the page rank. Let v_0 be an arbitrary vector in the range of A—we can initialize v_0 at random. Consider now repeatedly multiplying v by `A` and renormalizing v.

$$\begin{aligned} v_t &= Av_{t-1} \\ v_t &= \frac{v_t}{||v_t||} \\ \lambda &= v_t^T A v_t \end{aligned} \tag{7.10}$$

If we repeat this algorithm until convergence ($||v_t|| \approx ||v_{t-1}||$) or until the maximum number of iterations T is reached, we get our stationary distribution $\pi = v_T$. The reason this works is that we can write out our matrix $A = U\lambda U^T$. Then, we get the following equation.

$$\begin{aligned} v_t &= Av_{t-1} = A^t v_0 = \left(U\Lambda U^T\right)^t v_0 = U\left(\Lambda^t U^T v_0\right) \\ &= a_1\lambda_1^t u_1 + a_2\lambda_2^t u_2 + \cdots + a_n\lambda_n^t u_n \\ &= \lambda_1^t \left(a_1 u_1 + a_2\left(\frac{\lambda_2}{\lambda_1}\right)^t u_2 + \cdots + a_n\left(\frac{\lambda_n}{\lambda_1}\right)^t u_n\right) \\ &\rightarrow \lambda_1^t a_1 u_1 \end{aligned} \tag{7.11}$$

Equation 7.11 is true for some coefficients a_i, and since U is an ortho-normal matrix (i.e., $U^T U = I$) and $|\lambda_k| / |\lambda_1| < 1$ for $k > 1$, v_t converges to u_1, which is equal to our stationary distribution! We can summarize the page rank algorithm in the pseudo-code in figure 7.2.

```
**class** page_rank
**function** power_iteration(A):
$v_0 \sim \text{Unif}[0, 1]^d$
**while** (not converged) and (iter $\leq$ max_iter):
    $v_t = A_{v_{t-1}}$
    $v_t = \frac{v_t}{\|v_t\|}$
    $\lambda = v_t^T A v_t$
    converged = $\|v_t - v_{t-1}\| \leq$ tolerance
**end while**
**return** $\lambda, v_t$
```

Figure 7.2 Page rank pseudo-code

We start by sampling the initial value of our vector v from a d-dimensional uniform distribution. Next, we iterate by multiplying A and v and renormalizing v. We repeat the process until we either converge or exceed the maximum number of iterations. We are now ready to implement the power method algorithm for computing page rank from scratch in the following listing.

Listing 7.1 Page rank algorithm

```
import numpy as np
from numpy.linalg import norm

np.random.seed(42)

class page_rank():

    def __init__(self):
        self.max_iter = 100
        self.tolerance = 1e-5

    def power_iteration(self, A):
        n = np.shape(A)[0]
        v = np.random.rand(n)
        converged = False
        iter = 0

        while (not converged) and (iter < self.max_iter):
            old_v = v
            v = np.dot(A, v)
            v = v / norm(v)
            lambd = np.dot(v, np.dot(A, v))
            converged = norm(v - old_v) < self.tolerance
            iter += 1
        #end while
        return lambd, v

if __name__ == "__main__":

    X = np.random.rand(10,5)
    A = np.dot(X.T, X)
```

Constructs a symmetric random real matrix for simplicity

```
    pr = page_rank()
    lambd, v = pr.power_iteration(A)

    print(lambd)
    print(v)

    #compare against np.linalg implementation
    eigval, eigvec = np.linalg.eig(A)
    idx = np.argsort(np.abs(eigval))[::-1]
    top_lambd = eigval[idx][0]
    top_v = eigvec[:,idx][0]
```

Compares against np.linalg implementation

As we can see, our implementation successfully finds the eigenvector of `A` corresponding to the eigenvalue of `1`, which is equal to the page rank.

7.1.2 Hidden Markov models

Hidden Markov models (HMMs) represent time-series data in applications ranging from stock market prediction to DNA sequencing. HMMs are based on a discrete-state Markov chain with latent states $z_t \in \{1, ..., K\}$, a transition matrix K, and an emission matrix E that models observed data X emitted from each state. An HMM graphical model is shown in figure 7.3.

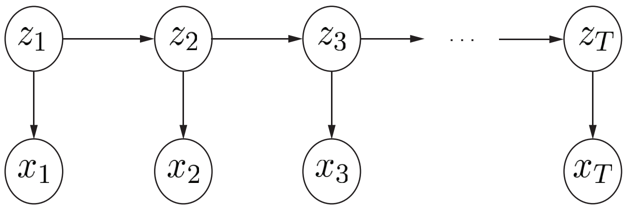

Figure 7.3 Hidden Markov model

We can write down the joint probability density as follows.

$$\begin{aligned}\overbrace{p(z_{1:T}, x_{1:T}|\theta)}^{\text{Joint distribution}} &= \overbrace{p(z_{1:T}|\theta)p(x_{1:T}|z_{1:T}, \theta)}^{\text{Factorization according to the HMM model}} \\ &= \underbrace{p(z_1|\pi)}_{\text{Initial state distribution}} \underbrace{\left[\prod_{t=2}^{T} p(z_t|z_{t-1}, A)\right]}_{\text{State transition distribution}} \underbrace{\left[\prod_{t=1}^{T} p(x_t|z_t, E)\right]}_{\text{State emission distribution}} \end{aligned} \tag{7.12}$$

Here, $\theta = \{\pi, A, E\}$ are HMM parameters, with π being the initial state distribution. The number of states K is often determined by the application. For example, we can model the sleep–wake cycle using $K = 2$ states. The data itself can be either discrete or continuous. We are going to focus on the discrete case in which the emission matrix $E_{kl} = p(x_t = 1 \mid z_t = k)$. The transition matrix is assumed to be time invariant and equal to $A_{ij} = p(z_t = j \mid z_{t-1}) = i)$.

We are typically interested in predicting the unobserved latent state z based on emitted observations x at any given time. In mathematical notation, we are after $p(z_t = j \mid x_{1:t})$. Let's call this quantity $\alpha_t(j)$.

$$\alpha_t(j) = p(z_t = j|x_{1:t}) \tag{7.13}$$

Let's compute it! Notice the recurrent relationship between successive values of alpha. Let's develop this recurrence.

$$\begin{aligned} p(z_t = j|x_{1:t-1}) &= \sum_{z_{t-1}} p(z_t = j, z_{t-1}|x_{1:t-1}) \\ &= \sum_i p(z_t = j|z_{t-1} = i)p(z_{t-1} = i|x_{1:t-1}) \\ &= \sum_i A(i,j)\alpha_{t-1}(i) \end{aligned} \tag{7.14}$$

In the prediction step, we are summing over all possible i states and multiplying the alpha by the transition matrix into j state. Let's use the result of the prediction step in the update step.

$$\begin{aligned} \alpha_t(j) &= p(z_t = j|x_{1:t}) = p(z_t = j|x_t, x_{1:t-1}) \\ &= \frac{p(x_t|z_t = j, x_{1:t-1})p(z_t = j|x_{1:t-1})}{p(x_t|x_{1:t-1})} \\ &= \frac{p(x_t|z_t = j)p(z_t = j|x_{1:t-1})}{\sum_{z_t} p(z_t, x_t|x_{1:t-1})} \\ &= \frac{p(x_t|z_t = j)p(z_t = j|x_{1:t-1})}{\sum_j p(x_t|z_t = j)p(z_t = j|x_{1:t-1})} \\ &= \frac{1}{Z_t} E_t(j) \sum_i A(i,j)\alpha_{t-1}(i) \end{aligned} \tag{7.15}$$

We can summarize our derivation in matrix notation as follows.

$$\alpha_t \propto E_t(A^T \alpha_{t-1}) \tag{7.16}$$

Given the recursion in alpha and the initial condition, we can compute the latent state marginals. This algorithm is referred to as the *forward algorithm,* due to its forward recursive pass to compute the alphas. It's a real-time algorithm suitable (with appropriate features) for applications such as handwriting or speech recognition. However, we can improve our estimates of the marginals by considering all the data up to time T. Let's figure out how we can do that by introducing the backward pass. The basic idea of the *forward–backward algorithm* is to partition the Markov chain into past and future by conditioning on z_t. Let T be the time our time series ends. Let's define the marginal probability over all observed data as the following.

$$\begin{aligned}\gamma_t(j) &= p(z_t = j|x_{1:T}) = p(z_t = j|x_{1:t}, x_{t+1:T}) \\ &= \frac{1}{Z_t} p(z_t = j|x_{1:t})p(x_{t+1:T}|z_t = j) \\ &\propto \alpha_t(j)\beta_t(j)\end{aligned} \tag{7.17}$$

Here, we defined $\beta_t(j) = p(x_{t+1:\mathrm{T}}|z_t = j)$ and used conditional independence of the past and future chain conditioned on state z_t. But our question remains: how can we compute $\beta_t(j)$? We can use a similar recursion going backward from time $t = T$.

$$\begin{aligned}\beta_{t-1}(i) &= p(x_{t:T}|z_{t-1} = i) = \sum_j p(z_t = j, x_t, x_{t+1:T}|z_{t-1} = i) \\ &= \sum_j p(x_{t+1:T}|z_t = j, x_t, z_{t-1} = i)p(z_t = j, x_t|z_{t-1} = i) \\ &= \sum_j p(x_{t+1:T}|z_t = j)p(z_t = j, x_t|z_{t-1} = i) \\ &= \sum_j p(x_{t+1:T}|z_t = j)p(x_t|z_t = j, z_{t-1} = i)p(z_t = j|z_{t-1} = i) \\ &= \sum_j \beta_t(j)E_t(j)A(i, j)\end{aligned} \tag{7.18}$$

We can summarize our derivation in the matrix notation as follows.

$$\beta_{t-1} = A(E_t\beta_t) \tag{7.19}$$

Here, the base case is given by the following.

$$\beta_T(i) = p(X_{T+1:T}|z_T = i) = 1 \tag{7.20}$$

This is true since the sequence ends at time T and $X_{T+1:T}$ is a nonevent with probability 1. Having computed both alpha and beta messages, we can combine them to produce our smoothed marginals: $\gamma_t(j) \propto \alpha_t(j)\ \beta_t(j)$.

Let's now look at how we can decode the maximum likelihood sequence of transitions between the latent state variables z_t. In other words, we would like to find the following.

$$z^* = \arg\max_{z_{1:T}} p(z_{1:T}|x_{1:T}) \tag{7.21}$$

Let $\delta_t(j)$ be the probability of ending up in state j given the most probable path sequence $z_{1:t-1}{}^*$.

$$\delta_t(j) = \max_{z_1,\ldots,z_{t-1}} p(z_t = j, z_{1:t-1}|x_{1:t}) \tag{7.22}$$

We can represent it recursively as the following.

$$\delta_t(j) = \max_i \delta_{t-1}(i)A(i,j)E_t(j) \tag{7.23}$$

This algorithm is known as the *Viterbi algorithm*. Let's summarize what we studied so far in the pseudo-code in figure 7.4.

```
1: class HMM
2: function forward_backward:
3: for $t = 1$ to $n$:
4:     $\alpha_t$ = normalize($E_t A^T \alpha_{t-1}$)
5: end for
6: for $t = n - 1$ to 1:
7:     $\beta_t$ = normalize($A\ (E_t \beta_{t+1})$)
8: end for
9: $\gamma = \alpha \cdot \beta$
10: return $\gamma, \alpha, \beta$
11: function viterbi:
12: for $t = 1$ to $n$:
13:     $\delta_t[j] = \max_i(\delta_{t-1}[i]A[i,j]E_t[j])$
14: end for
```

Figure 7.4 Hidden Markov model pseudo-code

We have two main functions: `forward_backward` and `viterbi`. In the `forward_backward` function, we construct a sparse matrix X of emission indicators and compute the forward probabilities α, normalizing in every iteration. Similarly, we compute the backward probabilities β as previously derived. Finally, we compute the marginal probabilities γ by multiplying α and β. In the `viterbi` function, we apply the log scale for numerical stability and replace multiplication with addition, computing the expression for δ as derived earlier. We are now ready to implement the inference of the HMM from scratch!

Listing 7.2 Forward–backward HMM algorithm

```
import numpy as np
from scipy.sparse import coo_matrix
import matplotlib.pyplot as plt

np.random.seed(42)

class HMM():
    def __init__(self, d=3, k=2, n=10000):
        self.d = d                          Dimension of data
        self.k = k                          Dimension of latent state
        self.n = n                          Number of data points

        self.A = np.zeros((k,k))            Transition matrix
        self.E = np.zeros((k,d))            Emission matrix
        self.s = np.zeros(k)                Initial state vector

        self.x = np.zeros(self.n)           Emitted observations

    def normalize_mat(self, X, dim=1):
        z = np.sum(X, axis=dim)
        Xnorm = X/z.reshape(-1,1)
        return Xnorm

    def normalize_vec(self, v):
        z = sum(v)
        u = v / z
        return u, z

    def init_hmm(self):

        #initialize matrices at random
        self.A = self.normalize_mat(np.random.rand(self.k,self.k))
        self.E = self.normalize_mat(np.random.rand(self.k,self.d))
        self.s, _ = self.normalize_vec(np.random.rand(self.k))

        #generate markov observations
        z = np.random.choice(self.k, size=1, p=self.s)
        self.x[0] = np.random.choice(self.d, size=1, p=self.E[z,:].ravel())
        for i in range(1, self.n):
            z = np.random.choice(self.k, size=1, p=self.A[z,:].ravel())
            self.x[i] = np.random.choice(self.d, size=1,
            ➥ p=self.E[z,:].ravel())
        #end for

    def forward_backward(self):

        #construct sparse matrix X of emission indicators
        data = np.ones(self.n)
        row = self.x
        col = np.arange(self.n)
        X = coo_matrix((data, (row, col)), shape=(self.d, self.n))

        M = self.E * X
        At = np.transpose(self.A)
        c = np.zeros(self.n)  #normalization constants
        alpha = np.zeros((self.k, self.n))  #alpha = p(z_t = j | x_{1:T})
        alpha[:,0], c[0] = self.normalize_vec(self.s * M[:,0])
        for t in range(1, self.n):
            alpha[:,t], c[t] = self.normalize_vec(np
            ➥ .dot(At, alpha[:,t-1]) * M[:,t])
        #end for

        beta = np.ones((self.k, self.n))
        for t in range(self.n-2, 0, -1):
            beta[:,t] = np.dot(self.A, beta[:,t+1] * M[:,t+1])/c[t+1]
        #end for
        gamma = alpha * beta

        return gamma, alpha, beta, c

    def viterbi(self):

        #construct sparse matrix X of emission indicators
        data = np.ones(self.n)
        row = self.x
        col = np.arange(self.n)
        X = coo_matrix((data, (row, col)), shape=(self.d, self.n))

        #log scale for numerical stability
        s = np.log(self.s)
        A = np.log(self.A)
        M = np.log(self.E * X)

        Z = np.zeros((self.k, self.n))
        Z[:,0] = np.arange(self.k)
        v = s + M[:,0]
        for t in range(1, self.n):
            Av = A + v.reshape(-1,1)
            v = np.max(Av, axis=0)
            idx = np.argmax(Av, axis=0)
            v = v.reshape(-1,1) + M[:,t].reshape(-1,1)
            Z = Z[idx,:]
            Z[:,t] = np.arange(self.k)
        #end for
        llh = np.max(v)
        idx = np.argmax(v)
        z = Z[idx,:]

        return z, llh


if __name__ == "__main__":

    hmm = HMM()
    hmm.init_hmm()

    gamma, alpha, beta, c = hmm.forward_backward()
    z, llh = hmm.viterbi()
```

After the code is executed, we return α, β, and γ parameters for a randomly initialized HMM model. After calling the viterbi decoder, we retrieve the maximum likelihood sequence of state transitions z.

In the following section, we will look at imbalanced learning. In particular, we will focus on undersampling and oversampling strategies for equalizing the number of samples among different classes.

7.2 Imbalanced learning

Most classification algorithms will only perform optimally when the number of samples in each class is roughly the same. Highly skewed datasets where the minority class is outnumbered by one or more classes commonly occur in fraud detection, medical diagnosis, and computational biology. One way of addressing this issue is by resampling the dataset to offset the imbalance and arrive at a more robust and accurate decision boundary. Resampling techniques can be broadly divided into four categories: undersampling the majority class, over-sampling the minority class, combining over and undersampling, and creating an ensemble of balanced datasets.

7.2.1 Undersampling strategies

Undersampling methods remove data from the majority class of the original dataset, as shown in figure 7.5. *Random undersampling* simply removes data points from the majority class uniformly at random. *Cluster centroids* is a method that replaces a cluster of samples by the cluster centroid of a K-means algorithm, where the number of clusters is set by the level of undersampling.

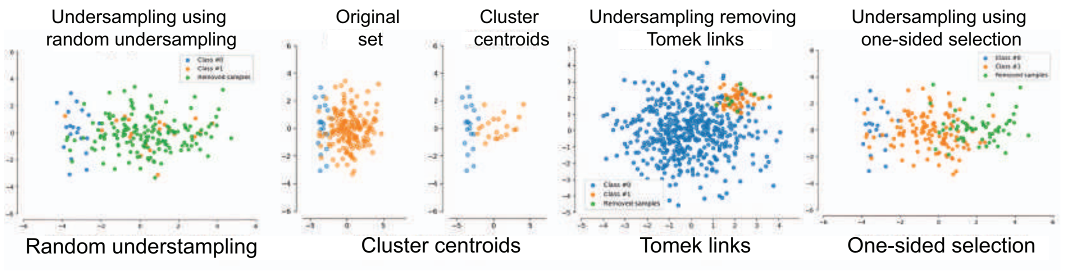

Figure 7.5 Undersampling strategies: Random, cluster centroids, Tomek links, and one-sided selection

Another effective undersampling method is *Tomek links*, which removes unwanted overlap between classes. Tomek links are removed until all minimally distanced nearest neighbor pairs are of the same class. A Tomek link is defined as follows: given an instance pair (x_i, x_j), where $x_i \in S_{min}$, $x_j \in S_{maj}$ and $d(x_i, x_j)$ is the distance between x_i and x_i, the (x_i, x_j) pair is called a Tomek link if there is no instance x_k such that $d(x_i, x_k) < d(x_i, x_k)$ or $d(x_j, x_k) < d(x_i, x_j)$. In this way, if two instances form a Tomek link, then either one of these instances is noise or both are near a border. Therefore, one can use Tomek links to clean up overlap between classes. By removing overlapping examples, one can establish well-defined clusters in the training set and lead to improved classification performance. The *one-sided selection* (OSS) method selects a representative subset of the majority class `E` and combines it with the set of all minority examples S_{min} to form $N = \{E \cup S_{min}\}$. The reduced set N is further processed to remove all majority class Tomek links. Let's experiment with Tomek links undersampling, using the imbalanced-learn library.

Listing 7.3 Tomek links algorithm

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from imblearn.under_sampling import TomekLinks

rng = np.random.RandomState(42)

def main():

    n_samples_1 = 500
    n_samples_2 = 50
    X_syn = np.r_[1.5 * rng.randn(n_samples_1, 2), 0.5 *
    ➥ rng.randn(n_samples_2, 2) + [2, 2]]                 <-- Generates data
    y_syn = np.array([0] * (n_samples_1) + [1] * (n_samples_2))
    X_syn, y_syn = shuffle(X_syn, y_syn)
    X_syn_train, X_syn_test, y_syn_train, y_syn_test =
    ➥ train_test_split(X_syn, y_syn)

    tl = TomekLinks(sampling_strategy='auto')                   Removes
    X_resampled, y_resampled = tl.fit_resample(X_syn, y_syn) <-- Tomek links
    idx_resampled = tl.sample_indices_
    idx_samples_removed =
     np.setdiff1d(np.arange(X_syn.shape[0]),idx_resampled)

    fig = plt.figure()          <-- Generates plots
    ax = fig.add_subplot(1, 1, 1)

    idx_class_0 = y_resampled == 0
    plt.scatter(X_resampled[idx_class_0, 0], X_resampled[idx_class_0, 1],
    ➥ alpha=.8, label='Class #0')
    plt.scatter(X_resampled[~idx_class_0, 0], X_resampled[~idx_class_0, 1],
    ➥ alpha=.8, label='Class #1')
    plt.scatter(X_syn[idx_samples_removed, 0], X_syn[idx_samples_removed, 1],
    ➥ alpha=.8, label='Removed samples')
    plt.title('Undersampling: Tomek links')
    plt.legend()
    plt.show()

if __name__ == "__main__":
    main()
```

Figure 7.6 shows the resulting output. Thus, we can see that removing unwanted class overlap can increase the robustness of our decision boundary and improve classification accuracy.

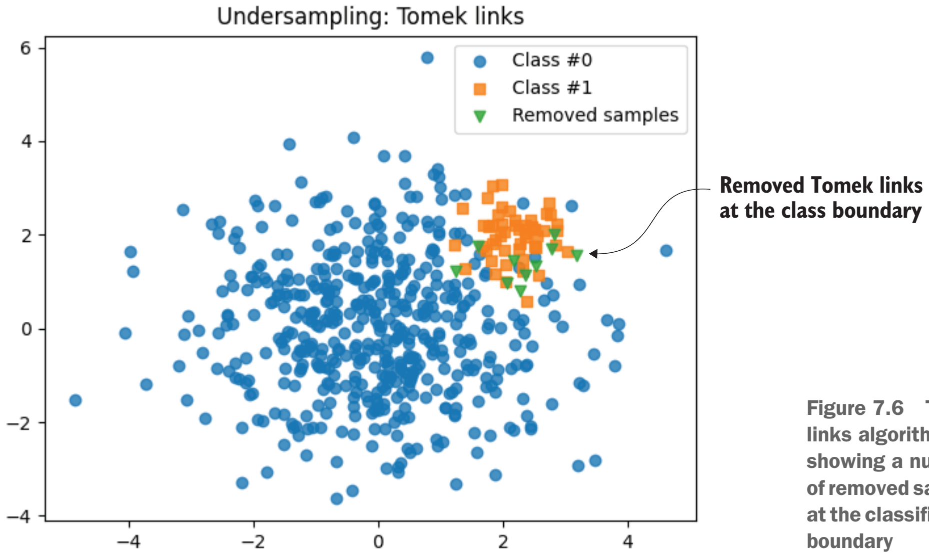

Figure 7.6 Tomek links algorithm showing a number of removed samples at the classification boundary

7.2.2 Oversampling strategies

Oversampling methods append data to the minority class of the original dataset, as shown in figure 7.7. *Random oversampling* simply adds data points to the minority class uniformly at random. The *synthetic minority oversampling technique* (SMOTE) generates synthetic examples by finding *K*-nearest neighbors in the feature space and generating a new data point along the line segments joining any of the *K*-minority class nearest neighbors. Synthetic samples are generated in the following way: take the difference between the feature vector (sample) under consideration and its nearest neighbor, multiply this difference by a random number between 0 and 1, and add it to the feature vector under consideration, augmenting the dataset with a new data point.

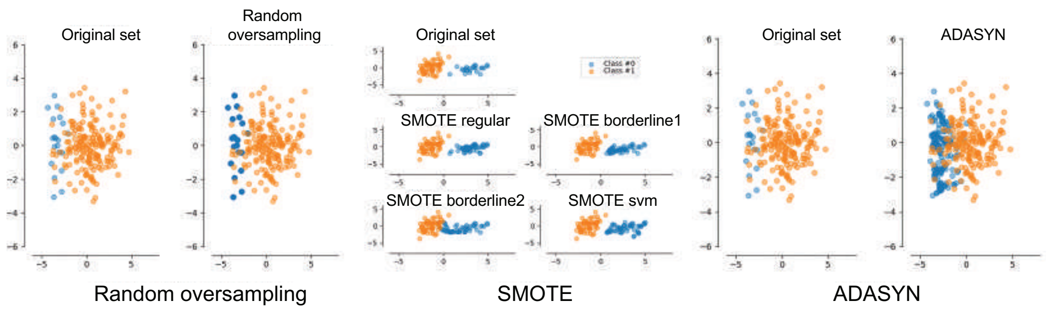

Figure 7.7 Oversampling strategies

Adaptive synthetic sampling (ADASYN) uses a weighted distribution for different minority class examples according to their level of difficulty in learning, where more synthetic data is generated for minority class examples that are harder to learn. As a result, the ADASYN approach improves learning of imbalanced datasets in two ways: reducing the bias introduced by class imbalance and adaptively shifting the classification decision boundary toward the difficult examples. Let's experiment with SMOTE oversampling using the imbalanced-learn library.

Listing 7.4 SMOTE algorithm

```
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.decomposition import PCA

from imblearn.over_sampling import SMOTE

def plot_resampling(ax, X, y, title):
    c0 = ax.scatter(X[y == 0, 0], X[y == 0, 1], label="Class #0", alpha=0.5)
    c1 = ax.scatter(X[y == 1, 0], X[y == 1, 1], label="Class #1", alpha=0.5)
    ax.set_title(title)
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    ax.spines['left'].set_position(('outward', 10))
    ax.spines['bottom'].set_position(('outward', 10))
    ax.set_xlim([-6, 8])
    ax.set_ylim([-6, 6])

    return c0, c1

def main():
    X, y = make_classification(n_classes=2, class_sep=2, weights=[0.3, 0.7],
                               n_informative=3, n_redundant=1, flip_y=0,
                               n_features=20, n_clusters_per_class=1,
                               n_samples=80,
                               random_state=10)     <-- Generates the dataset

    pca = PCA(n_components=2)
    X_vis = pca.fit_transform(X)     <-- Fits PCA for visualization

    method = SMOTE()
    X_res, y_res = method.fit_resample(X, y)     <-- Applies regular SMOTE
    X_res_vis = pca.transform(X_res)

    f, (ax1, ax2) = plt.subplots(1, 2)     <-- Generates plots
    c0, c1 = plot_resampling(ax1, X_vis, y, 'Original')
    plot_resampling(ax2, X_res_vis, y_res, 'SMOTE')
    ax1.legend((c0, c1), ('Class #0', 'Class #1'))
    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    main()
```

In listing 7.4, we generate a high dimensional dataset with two imbalanced classes. We apply a regular SMOTE oversampling algorithm and fit a two-component PCA to visualize the data in two dimensions. Figure 7.8 shows the resulting output.

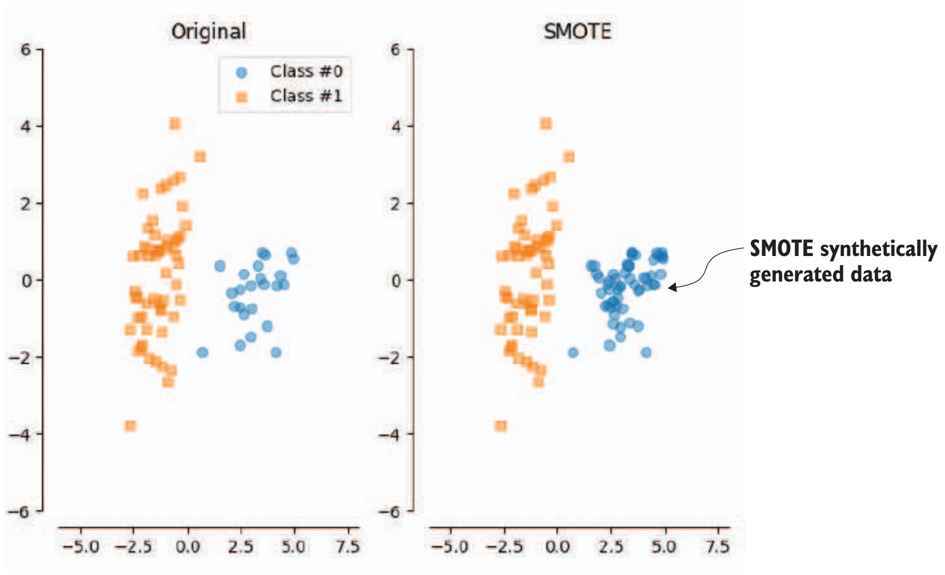

Figure 7.8 SMOTE algorithm

Thus, we can see that SMOTE densely populates the minority class with synthetic data. It's possible to combine oversampling and undersampling techniques into a hybrid strategy. Common examples include SMOTE and Tomek links or SMOTE and edited nearest neighbors (ENN). Additional ways of learning on imbalanced datasets include weighing training instances, introducing different misclassification costs for positive and negative examples, and bootstrapping. In the following section, we will focus on a very important principle in supervised ML that can reduce the number of required training examples: active learning.

7.3 Active learning

The key idea behind *active learning* is that a machine learning algorithm can achieve greater accuracy with fewer training labels if it is allowed to choose the data from which it learns. Active learning is well motivated in many modern machine learning problems where unlabeled data may be abundant, but labels are expensive to obtain. Active learning is sometimes called *query learning* or *optimal experimental design* because an active learner poses queries in the form of unlabeled data instances to be labeled by an oracle. In this way, the active learner seeks to achieve high accuracy using as few labeled instances as possible. For a review, see "Active Learning Literature Survey" by Burr Settles (University of Wisconsin-Madison, Department of Computer Sciences, 2009).

We focus on pool-based sampling that assumes that there is a small set of labeled data L and a large pool of unlabeled data U. Queries are selectively drawn from the pool according to an informativeness measure. Pool-based methods rank the entire collection of unlabeled data to select the best query. Therefore, for very large datasets, stream-based sampling, where the data is scanned sequentially and query decisions are evaluated individually, may be more appropriate. Figure 7.9 shows an example of pool-based active learning based on the binary classification of a synthetic dataset with two balanced classes using logistic regression (LR).

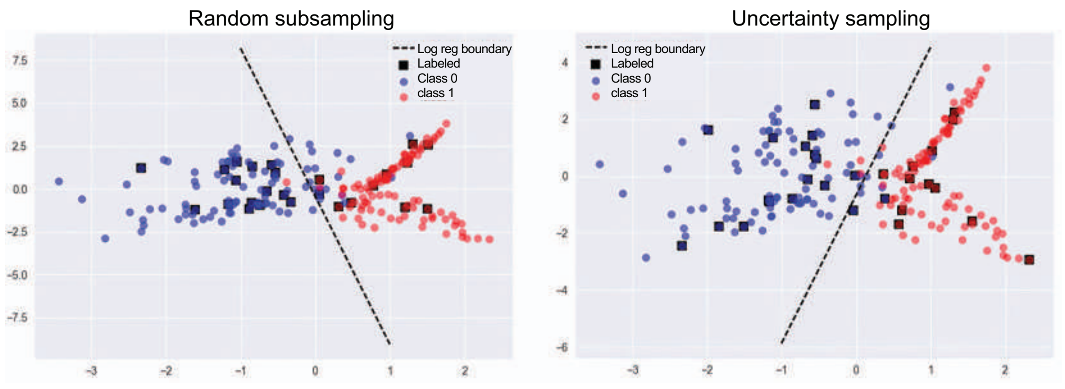

Figure 7.9 Active learning for logistic regression

On the left, we can see the LR decision boundary as a result of training on a randomly subsampled set of 30 labels that achieves a classification accuracy of 90% on held-out data. On the right, we can see the LR decision boundary as a result of training on 30 queries selected by uncertainty sampling based on entropy. Uncertainty sampling achieves a higher classification accuracy of 92.5% on the held-out set.

7.3.1 Query strategies

Query strategies refer to criteria we use to select a subset of the training examples as part of active learning. Here, we will look at two query strategies: uncertainty sampling and query by committee.

Uncertainty sampling

One of the simplest and most commonly used query frameworks is *uncertainty sampling*. In this framework, an active learner queries the label about which it is least certain. For example, in binary logistic regression, uncertainty sampling queries points near the boundary where the probability of being positive is close to ½. For multiclass problems, uncertainty sampling can query points that are least confident.

$$x^*_{LC} = \arg\max_x 1 - P_\theta(\hat{y}|x) \tag{7.24}$$

Here, $y \in \{1, ..., K\}$ is the class label with the highest posterior probability under the model θ. The criterion for the least confident strategy only considers information about the most probable label. We can use max margin sampling to preserve information about the remaining label distribution.

$$x^*_M = \arg\min_x P_\theta(\hat{y}_1|x) - P_\theta(\hat{y}_2|x) \tag{7.25}$$

Here, y_1 and y_2 are the first and second most probable class labels under the model, respectively. Intuitively, instances with large margins are easy to classify. Thus, points with small margins are ambiguous, and knowing their labels would help the model to more effectively discriminate between them. However, for multiclass problems with very large label sets, the margin strategy still ignores much of the output distribution for the remaining classes. A more general uncertainty sampling strategy is based on entropy.

$$x^*_H = \arg\max_x - \sum_i P_\theta(y_i|x) \log P_\theta(y_i|x) \tag{7.26}$$

By learning labels that have the highest entropy, we can reduce label uncertainty. Uncertainty sampling also works for regression problems, in which case the learner queries the point with the highest output variance in its prediction.

QUERY BY COMMITTEE

Another query selection framework is the query by committee (QBC) algorithm, which involves maintaining a committee $C = \{\theta^1, ..., \theta^C\}$ of models that are all trained on the current labeled set L but represent competing hypotheses. Each committee member is then allowed to vote on the labels of query candidates, and the most informative query is considered an instance about which they most disagree. The objective of QBC is to minimize a set of hypotheses that are consistent with the current labeled training data L. Two main approaches have been proposed for measuring the level of disagreement: vote entropy and KL divergence. Vote entropy is defined as follows.

$$x^*_{VE} = \arg\max_x - \sum_i \frac{V(y_i)}{C} \log \frac{V(y_i)}{C} \tag{7.27}$$

Here, $y_i \in \{1, ..., K\}$ is the class label, $V(y_i)$ is the number of votes a label received from the committee members, and C is the size of the committee. Notice the similarity to equation 7.26. The KL divergence for QBC voting is defined as follows.

$$\begin{aligned} x^*_{KL} &= \arg\max_x \frac{1}{C}\sum_{c=1}^{C} KL(P_{\theta^{(c)}}||P_C) \\ KL(P_{\theta^{(c)}}||P_C) &= \sum_i P_{\theta^{(c)}}(y_i|x)\log\frac{P_{\theta^{(c)}}(y_i|x)}{P_C(y_i|x)} \\ P_C(y_i|x) &= \frac{1}{C}\sum_{c=1}^{C} P_{\theta^{(c)}}(y_i|x) \end{aligned} \tag{7.28}$$

Here, θ_c represents a member model of the committee and $P_C(y_i|x)$ is the consensus probability that y_i is the predicted label. The KL divergence metric considers the most informative query to be the one with the largest average difference between the label distributions of any one committee member and the consensus distribution.

Variance reduction

We can reduce the generalization error by minimizing output variance. Consider a regression problem for which the learning objective is to minimize the mean squared error. Let $\bar{\theta} = [\hat{\theta}]$ be the expected value of the parameter estimate $\hat{\theta}$ and θ^* be the ground truth. Then, we get the following.

$$\begin{aligned} \text{MSE} &= E\left[(\hat{\theta}-\theta^*)^2\right] = E\left[\left[(\hat{\theta}-\bar{\theta})+(\bar{\theta}-\theta^*)\right]^2\right] \\ &= E\left[(\hat{\theta}-\bar{\theta})^2\right] + 2\left(\bar{\theta}-\theta^*\right)E\left[\hat{\theta}-\bar{\theta}\right] + \left(\bar{\theta}-\theta^*\right)^2 \\ &= E\left[(\hat{\theta}-\bar{\theta})^2\right] + \left(\bar{\theta}-\theta^*\right)^2 \\ &= \text{VAR}\left[\hat{\theta}\right] + \text{bias}^2\left(\hat{\theta}\right) \end{aligned} \tag{7.29}$$

This is called the *bias–variance tradeoff*. Thus, it is possible to achieve lower MSE with a biased estimator if it reduces the variance. We might then wonder how low the variance can be. The answer is given by the Cramer-Rao lower bound, which provides a lower bound on the variance of any unbiased estimator.

Cramer-Rao lower bound

Assuming $p(x|\theta)$ satisfies the regularity condition, the variance of any unbiased estimator satisfies the following.

$$\text{VAR}(\hat{\theta}) \geq \frac{1}{-E\left[\frac{\partial^2 \log p(x|\theta)}{\partial\theta^2}\right]} = \frac{1}{I(\theta)} \tag{7.30}$$

Here, $I(\theta)$ is the Fisher information matrix. Thus, the minimum variance unbiased (MVU) estimator achieves the minimum variance equal to the inverse of the Fisher information matrix. To minimize the variance of parameter estimates, an active learner should select data that maximizes its Fisher information. For multivariate models with K parameters, Fisher information takes the form of a $K \times K$ matrix.

$$[I(\theta)]_{ij} = -E\left[\frac{\partial^2 \log p(x|\theta)}{\partial\theta_i \partial\theta_j}\right] \tag{7.31}$$

As a result, there are several options for minimizing the inverse information matrix: A-optimality minimizes the trace, $T_r(I^{-1}(\theta))$; D-optimality minimizes the determinant, $|I^{-1}(\theta)|$; and E-optimality minimizes the maximum eigenvalue, $\lambda_{max}\left[I^{-1}(\theta)\right]$.

However, there are some computational disadvantages to the variance reduction methods. Estimating output variance requires inverting a $K \times K$ matrix for each unlabeled instance, resulting in a time complexity of $O(UK^3)$, where U is the size of the query pool. As a result, variance reduction methods are empirically slower than simpler query strategies like uncertainty sampling. Let's look at some pseudo-code for active learner class in figure 7.10.

```
class ActiveLearner
function uncertainty_sampling(clf, X):
$P_\theta(\hat{y}|x)$ = clf.predict_proba(X)
if strategy = “least confident”:
    return arg $\max_x 1 - P_\theta(\hat{y}|x)$
else if strategy = “max margin”:
    return arg $\min_x P_\theta(\hat{y}_1|x) - P_\theta(\hat{y}_2|x)$
else if strategy = “entropy”:
    return arg $\max_x - \sum_i P_\theta(y_i|x) \log P_\theta(y_i|x)$
end if
function query_by_committee(clf, X):
if strategy = “vote entropy”:
    $C$ = len(clf)
    for model in clf:
        $y_i$ = clf.predict
    end for
    $V(y_i) = \frac{1}{C}\sum_{i=1}^{C} 1[[y_1]]$
    return arg $\max_x - \sum_i \frac{V(y_i)}{C} \log \frac{V(y_i)}{C}$
else if strategy = “average kl divergence”:
    $P_C(y_i|x) = \frac{1}{C}\sum_{c=1}^{C} P_{\theta(c)}(y_i|x)$
    $KL\left(P_{\theta(c)}||P_C\right) = \sum_i P_{\theta(c)}(y_i|x) \log \frac{P_{\theta(c)}(y_i|x)}{P_C(y_i|x)}$
    return arg $\max_x \frac{1}{C}\sum_{c=1}^{C} KL\left(P_{\theta(c)}||P_C\right)$
end if
```

Figure 7.10 Active learner pseudo-code

We have two main functions: `uncertainty_sampling` and `query_by_committee`. In the `uncertainty_sampling` function, we pass in the classifier model `clf` and the unlabeled data `X` as inputs. We then predict the probability of the label, given the classifier model and the training data, and use this prediction to compute one of three uncertainty sampling strategies, as discussed in the text. In the `query_by_committee` function, we implement two committee methods: vote entropy and average KL divergence. However, we now pass a set of models `clf`—the predictions of which are used to vote on the predicted label, thus forming a distribution for evaluating the entropy. In the KL divergence case, we make use of averages of model predictions in the computation of KL divergence between each model and the average. We return the training sample that maximizes this KL divergence. We are now ready to review the following code listing.

Listing 7.5 Active learner class

```
from __future__ import unicode_literals, division
from scipy.sparse import csc_matrix, vstack
from scipy.stats import entropy
from collections import Counter
import numpy as np


class ActiveLearner(object):

    uncertainty_sampling_frameworks = [
        'entropy',
        'max_margin',
        'least_confident',
    ]

    query_by_committee_frameworks = [
        'vote_entropy',
        'average_kl_divergence',
    ]

    def __init__(self, strategy='least_confident'):
        self.strategy = strategy

    def rank(self, clf, X_unlabeled, num_queries=None):

        if num_queries == None:
            num_queries = X_unlabeled.shape[0]

        elif type(num_queries) == float:
            num_queries = int(num_queries * X_unlabeled.shape[0])

        if self.strategy in self.uncertainty_sampling_frameworks:
            scores = self.uncertainty_sampling(clf, X_unlabeled)

        elif self.strategy in self.query_by_committee_frameworks:
            scores = self.query_by_committee(clf, X_unlabeled)

        else:
            raise ValueError("this strategy is not implemented.")

        rankings = np.argsort(-scores)[:num_queries]
        return rankings
```

Uncertainty sampling frameworks

Query by committee frameworks

```
    def uncertainty_sampling(self, clf, X_unlabeled):
        probs = clf.predict_proba(X_unlabeled)

        if self.strategy == 'least_confident':
            return 1 - np.amax(probs, axis=1)    <--- Least confident

        elif self.strategy == 'max_margin':
            margin = np.partition(-probs, 1, axis=1)
            return -np.abs(margin[:,0] - margin[:, 1])    <--- Max margin

        elif self.strategy == 'entropy':
            return np.apply_along_axis(entropy, 1,
            ➥ probs)                              <--- Entropy

    def query_by_committee(self, clf, X_unlabeled):
        num_classes = len(clf[0].classes_)
        C = len(clf)
        preds = []

        if self.strategy == 'vote_entropy':
            for model in clf:
                y_out = map(int, model.predict(X_unlabeled))
                preds.append(np.eye(num_classes)[y_out])

            votes = np.apply_along_axis(np.sum, 0, np.stack(preds)) / C
            return np.apply_along_axis(entropy, 1,
            ➥ votes)         <--- Vote entropy

        elif self.strategy == 'average_kl_divergence':
            for model in clf:
                preds.append(model.predict_proba(X_unlabeled))

            consensus = np.mean(np.stack(preds), axis=0)
            divergence = []
            for y_out in preds:
                divergence.append(entropy(consensus.T, y_out.T))

            return np.apply_along_axis(np.mean, 0, np
            ➥ .stack(divergence))             <--- Average KL divergence
```

We apply our active learner to logistic regression example in the following listing.

Listing 7.6 Active learner for logistic regression

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from active_learning import ActiveLearner
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
```

```
np.random.seed(42)

def main():

    num_queries = 30    <-- Number of labeled points

    data, target = make_classification(n_samples=200, n_features=2,
    ➥ n_informative=2,\
                                       n_redundant=0,
    ➥ n_classes=2, weights = [0.5, 0.5], random_state=0)    <-- Generates data

    X_train, X_unlabeled, y_train, y_oracle = train_
    ➥ test_split(data, target, test_size=0.2, random_state=0)    <-- Splits into labeled and unlabeled pools

    rnd_idx = np.random.randint(0, X_train.shape[0],
    ➥ num_queries)    <-- Random subsampling
    X1 = X_train[rnd_idx,:]
    y1 = y_train[rnd_idx]

    clf1 = LogisticRegression()
    clf1.fit(X1, y1)

    y1_preds = clf1.predict(X_unlabeled)
    score1 = accuracy_score(y_oracle, y1_preds)
    print("random subsampling accuracy: ", score1)

    #plot 2D decision boundary: w2x2 + w1x1 + w0 = 0
    w0 = clf1.intercept_
    w1, w2 = clf1.coef_[0]
    xx = np.linspace(-1, 1, 100)
    decision_boundary = -w0/float(w2) - (w1/float(w2))*xx

    plt.figure()
    plt.scatter(data[rnd_idx,0], data[rnd_idx,1], c='black', marker='s',
    ➥ s=64, label='labeled')
    plt.scatter(data[target==0,0], data[target==0,1], c='blue', marker='o',
    ➥ alpha=0.5, label='class 0')
    plt.scatter(data[target==1,0], data[target==1,1], c='red', marker='o',
    ➥ alpha=0.5, label='class 1')
    plt.plot(xx, decision_boundary, linewidth = 2.0, c='black', linestyle =
    ➥ '--', label='log reg boundary')
    plt.title("Random Subsampling")
    plt.legend()
    plt.show()

    AL = ActiveLearner(strategy='entropy')    <-- Active learning
    al_idx = AL.rank(clf1, X_unlabeled, num_queries=num_queries)

    X2 = X_train[al_idx,:]
    y2 = y_train[al_idx]

    clf2 = LogisticRegression()
    clf2.fit(X2, y2)
```

```
        y2_preds = clf2.predict(X_unlabeled)
        score2 = accuracy_score(y_oracle, y2_preds)
        print("active learning accuracy: ", score2)

        #plot 2D decision boundary: w2x2 + w1x1 + w0 = 0
        w0 = clf2.intercept_
        w1, w2 = clf2.coef_[0]
        xx = np.linspace(-1, 1, 100)
        decision_boundary = -w0/float(w2) - (w1/float(w2))*xx

        plt.figure()
        plt.scatter(data[al_idx,0], data[al_idx,1], c='black', marker='s',
        ➥ s=64, label='labeled')
        plt.scatter(data[target==0,0], data[target==0,1], c='blue', marker='o',
        ➥ alpha=0.5, label='class 0')
        plt.scatter(data[target==1,0], data[target==1,1], c='red', marker='o',
        ➥ alpha=0.5, label='class 1')
        plt.plot(xx, decision_boundary, linewidth = 2.0, c='black', linestyle =
        ➥ '--', label='log reg boundary')
        plt.title("Uncertainty Sampling")
        plt.legend()
        plt.show()

if __name__ == "__main__":

        main()
```

Active learning and semi-supervised learning both attempt to make the most of unlabeled data. For example, a basic semi-supervised technique is self-training, in which the learner is first trained with a small amount of labeled data and then used to classify the unlabeled data. The most confident unlabeled instances together with their predicted labels are added to the training set, and the process repeats.

Figure 7.11 compares three uncertainty sampling techniques: least confident, max margin, and entropy with a random subsampling baseline on a dataset of 20 newsgroups classified with logistic regression.

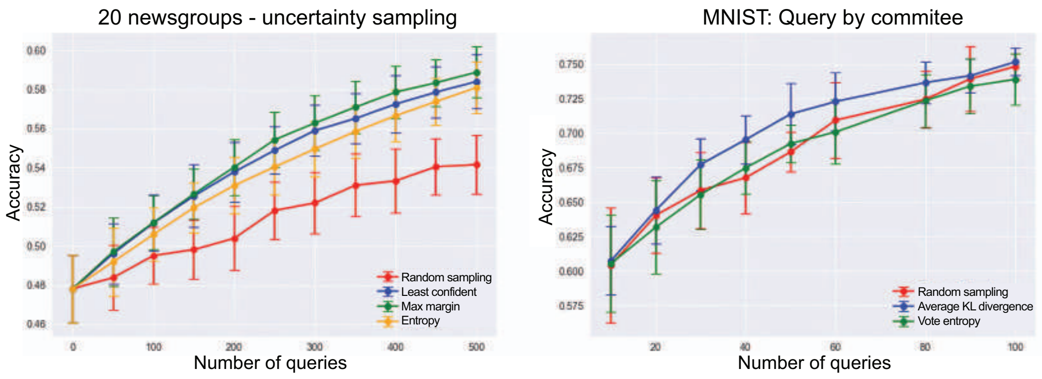

Figure 7.11 Uncertainty sampling techniques comparison. The left figure shows that active learning is better than random sampling. The right figure shows that average KL divergence is better than random sampling.

All three methods achieve higher accuracy in comparison to the baseline, which highlights the benefit of active learning. On the right, we can see the performance of the query by committee strategy applied to the MNIST dataset. The committee consists of five instances of logistic regression. Two methods are compared against the random subsampling baseline: vote entropy and average KL divergence. We can see that average KL divergence achieves the highest classification accuracy. All experiments were repeated 10 times.

7.4 Model selection: Hyperparameter tuning

In our machine learning journey, we are often faced with multiple models. Model selection is focused on choosing the optimal model (i.e., the model that has a sufficient number of parameters [degrees of freedom] so as to not underfit or overfit to training data). We can summarize model selection using the *Occam's razor* principle: choose the simplest model that explains the data well. In other words, we want to penalize model complexity out of all possible solutions.

In addition, we'd like to explain more than just the training data—we'd like to explain the future data. As a result, we want our model to generalize to new and unseen data. A model's behavior is often characterized by its hyperparameters (e.g., the number of nearest neighbors or the number of clusters in the *K*-means algorithm). Let's look at several ways in which we can find the optimum hyperparameters for model selection.

We often operate in a multidimensional hyperparameter space in which we would like to find an optimum point that leads to the highest-performing model on the validation dataset. For example, in the case of support vector machines (SVMs), we may try different kernels, kernel parameters, and regularization constants.

There are several strategies we can use for hyperparameter tuning. The most straightforward strategy is called *grid search*, which is an exhaustive search over all possible combinations of hyperparameters. When using grid search, we can set the hyperparameter values using the log scale (e.g., 0.1, 1, 10, 100) to arrive at the right order of magnitude. Grid search works well for smaller hyperparameter spaces.

An alternative to grid search is a *random search*, which is based on sampling the hyperparameters from corresponding prior distributions. For example, in finding the number of clusters `K`, we may sample `K` from a discrete exponential distribution. The random search has the computational advantage of finding the optimum set faster than an exhaustive grid search. Moreover, for a random search, the number of iterations can be chosen independent of the number of parameters and adding parameters that do not influence performance does not decrease efficiency. A third, and most interesting, alternative we will look at in the following subsection is called *Bayesian optimization*.

7.4.1 Bayesian optimization

Rather than exploring the parameter space randomly (according to a chosen distribution), it would be great to adopt an active learning approach that selects continuous parameter values in a way that reduces uncertainty and provides a balance between exploration and exploitation. Bayesian optimization provides an automated Bayesian framework by utilizing Gaussian processes (GPs) to model the algorithm's generalization performance (see Jasper Snoek, Hugo Larochelle, and Ryan P. Adams's "Practical Bayesian Optimization of Machine Learning Algorithms," Conference and Workshop on Neural Information Processing Systems, 2012).

Bayesian optimization assumes that a suitable performance function was sampled from a GP and maintains a posterior distribution for this function as observations are made: $f(x) \sim \mathrm{GP}(m(x), \kappa(x, x'))$. To choose which hyperparameters to explore next, one can optimize the expected improvement (EI) over the current best result or the Gaussian process upper confidence bound (UCB). EI and UCB have been shown to be efficient in the number of function evaluations required to find the global optimum of multimodal black-box functions.

Bayesian optimization uses all the information available from previous evaluations of the objective function, as opposed to relying on the local gradient and Hessian approximations. This results in an automated procedure that can find an optimum of non-convex functions with relatively few evaluations, at the cost of performing more computation to determine the next point to try. This is particularly useful when evaluations are expensive to perform, such as in selecting hyperparameters for deep neural networks. The Bayesian optimization algorithm is summarized in figure 7.12.

```
1: for $n = 1, 2, \ldots$ do
2:     select new $x_{n+1}$ by optimizing acquisition function $\alpha$
3:         $x_{n+1} = \arg\max_x \alpha(x; D_n, \theta)$
4:     query objective function to obtain $y_{n+1} = f(x_{n+1})$
5:     augment data $D_{n+1} = \{D_n, (x_{n+1}, y_{n+1})\}$
6:     update GP posterior and acquisition function
7: end for
```

Figure 7.12 Bayesian optimization algorithm

The following listing applies Bayesian Optimization for hyperparameter search in SVM and random forest classifier (RFC).

Listing 7.7 Bayesian optimization for SVM and RFC

```python
import numpy as np
import pandas as pd

import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.svm import SVC

from bayes_opt import BayesianOptimization

np.random.seed(42)

# Load data set and target values
data, target = make_classification(
    n_samples=1000,
    n_features=45,
    n_informative=12,
    n_redundant=7
)
target = target.ravel()

def svccv(gamma):                                  SVM classifier
    val = cross_val_score(
        SVC(gamma=gamma, random_state=0),
        data, target, scoring='f1', cv=2
    ).mean()

    return val
                                                   Random forest
def rfccv(n_estimators, max_depth):                (RF) classifier
    val = cross_val_score(
        RFC(n_estimators=int(n_estimators),
            max_depth=int(max_depth),
            random_state=0
        ),
        data, target, scoring='f1', cv=2
    ).mean()
    return val

if __name__ == "__main__":

    gp_params = {"alpha": 1e-5}

    #SVM
    svcBO = BayesianOptimization(svccv,
        {'gamma': (0.00001, 0.1)})

    svcBO.maximize(init_points=3, n_iter=4, **gp_params)

    #Random Forest
    rfcBO = BayesianOptimization(
        rfccv,
        {'n_estimators': (10, 300),
         'max_depth': (2, 10)
        }
    )
    rfcBO.maximize(init_points=4, n_iter=4, **gp_params)

    print('Final Results')
    print(svcBO.max)
    print(rfcBO.max)
```

Figure 7.13 shows Bayesian optimization applied to SVM and RFC.

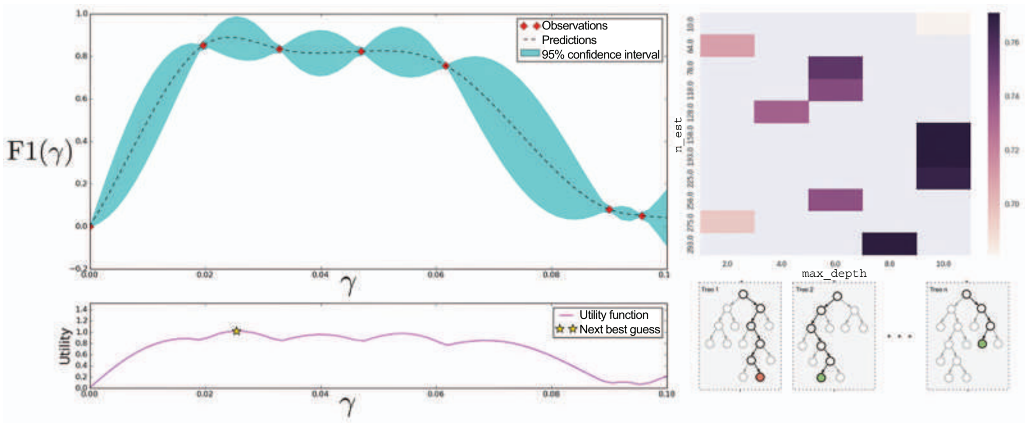

Figure 7.13 Bayesian optimization applied to SVM and RFC

The F1 score was used as a performance objective function for a classification task. The figure on the left shows the Bayesian optimization of the F1 score as a function of the gamma parameter of the SVM RBF kernel $K(x, x') = \exp\{-\gamma||x - x'||^2\}$, where gamma is the precision equal to the inverse variance. We can see that after only seven iterations, we have discovered the gamma parameter that gives the maximum F1 score. The peak of EI utility function at the bottom tells us which experiment to perform next. The figure on the right shows the Bayesian optimization of the F1 score as a function of maximum depth and the number of estimators of a Random Forest classifier. From the heatmap, we can tell that the maximum F1 score is achieved for 158 estimators with a depth equal to 10.

7.5 Ensemble methods

Ensemble methods are meta-algorithms that combine several ML techniques into one predictive model to decrease variance (bagging), decrease bias (boosting), or improve predictions (stacking). Ensemble methods can be divided into two groups: *sequential* ensemble methods, where the base learners are generated sequentially (e.g., AdaBoost), and *parallel* ensemble methods, where the base learners are generated in parallel (e.g., random forest). The basic motivation of sequential methods is to exploit the dependence between the base learners, since the overall performance can be boosted by weighing previously mislabeled examples with higher weight. The basic motivation of parallel methods is to exploit independence between the base learners, since the error can be reduced dramatically by averaging.

Most ensemble methods use a single base learning algorithm to produce homogeneous base learners (i.e., learners of the same type), leading to *homogeneous ensembles.* There are also some methods that use heterogeneous learners (i.e., learners of different types), leading to *heterogeneous ensembles*. To ensure ensemble methods are more accurate than any of their individual members, the base learners must be as accurate and diverse as possible.

7.5.1 Bagging

Bagging stands for *bootstrap aggregation*. One way to reduce the variance of an estimate is to average together multiple estimates. For example, we can train M different decision trees f_m on different subsets of the data (chosen randomly with replacement) and compute the ensemble.

$$f(x) = \frac{1}{M}\sum_{m=1}^{M} f_m(x) \tag{7.32}$$

Bagging uses bootstrap sampling to obtain the data subsets for training the base learners. For aggregating the outputs of base learners, bagging uses voting for classification and averaging for regression. The following listing shows bagging ensemble experiments.

Listing 7.8 Bagging ensemble

```
import itertools
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions

def main():

    iris = datasets.load_iris()
    X, y = iris.data[:, 0:2], iris.target

    clf1 = DecisionTreeClassifier(criterion='entropy',
    ➥ max_depth=None)                                      Decision tree classifier
    clf2 = KNeighborsClassifier(n_neighbors=1)             KNN classifier

    bagging1 = BaggingClassifier(base_estimator=clf1, n_estimators=10,
    ➥ max_samples=0.8, max_features=0.8)
    bagging2 = BaggingClassifier(base_estimator=clf2, n_estimators=10,
    ➥ max_samples=0.8, max_features=0.8)

    label = ['Decision Tree', 'K-NN', 'Bagging Tree', 'Bagging K-NN']
    clf_list = [clf1, clf2, bagging1, bagging2]

    fig = plt.figure(figsize=(10, 8))
    gs = gridspec.GridSpec(2, 2)
    grid = itertools.product([0,1],repeat=2)

    for clf, label, grd in zip(clf_list, label, grid):
        scores = cross_val_score(clf, X, y, cv=3, scoring='accuracy')
        print("Accuracy: %.2f (+/- %.2f) [%s]" %(scores.mean(),
        ➥ scores.std(), label))

        clf.fit(X, y)
        ax = plt.subplot(gs[grd[0], grd[1]])
        fig = plot_decision_regions(X=X, y=y, clf=clf, legend=2)
        plt.title(label)

    plt.show()
    #plt.savefig('./figures/bagging_ensemble.png')

    #plot learning curves
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    ➥ test_size=0.3, random_state=0)

    plt.figure()
    plot_learning_curves(X_train, y_train, X_test, y_test, bagging1,
    ➥ print_model=False, style='ggplot')
    plt.show()
    #plt.savefig('./figures/bagging_ensemble_learning_curve.png')

    #Ensemble Size
    num_est = list(map(int, np.linspace(1,100,20)))
    bg_clf_cv_mean = []
    bg_clf_cv_std = []
    for n_est in num_est:
        print("num_est: ", n_est)
        bg_clf = BaggingClassifier(base_estimator=clf1, n_estimators=n_est,
        ➥ max_samples=0.8, max_features=0.8)
        scores = cross_val_score(bg_clf, X, y, cv=3, scoring='accuracy')
        bg_clf_cv_mean.append(scores.mean())
        bg_clf_cv_std.append(scores.std())

    plt.figure()
    (_, caps, _) = plt.errorbar(num_est, bg_clf_cv_mean,
    ➥ yerr=bg_clf_cv_std, c='blue', fmt='-o', capsize=5)
    for cap in caps:
        cap.set_markeredgewidth(1)
plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt.title('Bagging
➥ Tree Ensemble');
    plt.show()
    #plt.savefig('./figures/bagging_ensemble_size.png')

if __name__ == "__main__":
    main()
```

Figure 7.14 shows the decision boundary of a decision tree and KNN classifiers, along with their bagging ensembles applied to the iris dataset. The decision tree shows axes parallel boundaries, while the $k = 1$ nearest neighbors fit closely to the data points. The bagging ensembles were trained using 10 base estimators with 0.8 subsampling of training data and 0.8 subsampling of features. The decision tree bagging ensemble achieved higher accuracy than the KNN bagging ensemble because KNN is less sensitive to perturbation on training samples; therefore, they are called *stable learners*. Combining stable learners is less advantageous, since the ensemble will not help improve generalization performance. Notice also that the decision boundary of the bagging KNN looks similar to the decision boundary of the bagging tree as a result of voting. The figure also shows how the test accuracy improves with the size of the ensemble. Based on cross-validation results, we can see the accuracy increases until approximately 20 base estimators and then plateaus afterward. Thus, adding base estimators beyond 20 only increases computational complexity, without accuracy gains for the iris dataset. The figure also shows learning curves for the bagging tree ensemble. We can see an average error of 0.15 on the training data and a U-shaped error curve for the testing data. The smallest gap between training and test errors occurs at around 80% of the training set size.

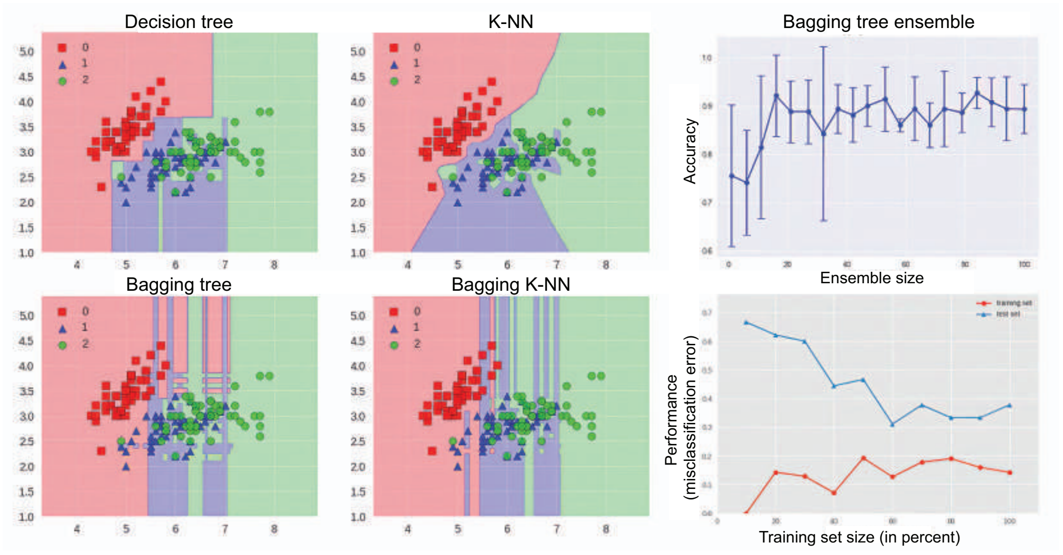

Figure 7.14 Bagging ensemble applied to the iris dataset, as generated by the code listing

A commonly used class of ensemble algorithms is forests of randomized trees. In a *random forest*, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. In addition, instead of using all the features, a random subset of features is selected, further randomizing the tree. As a result, the bias of the forest increases slightly, but due to the averaging of less correlated trees, its variance decreases, resulting in an overall better model.

In the *extremely randomized trees*, algorithm randomness goes one step further: the splitting thresholds are randomized. Instead of looking for the most discriminative threshold, thresholds are drawn at random for each candidate feature and the best of these randomly generated thresholds is picked as the splitting rule. This usually allows the variance of the model to be reduced a bit more—at the expense of a slightly greater increase in bias.

7.5.2 Boosting

Boosting refers to a family of algorithms that can convert weak learners to strong learners. The main principle of boosting is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) to weighted versions of the data, where more weight is given to examples that were misclassified by earlier rounds. The predictions are then combined through a weighted majority vote (classification) or a weighted sum (regression) to produce the final prediction. The principal difference between boosting and the committee methods, such as bagging, is that base learners are trained in sequence on a weighted version of the data.

The algorithm in figure 7.15 describes the most widely used form of boosting algorithm: *AdaBoost*, which stands for adaptive boosting. We see that the first base classifier $y_1(x)$ is trained using weighting coefficients $w_n{}^1$ that are all equal. In subsequent boosting rounds, the weighting coefficients $w_n{}^m$ are increased for data points that are misclassified and decreased for data points that are correctly classified. The quantity ϵ_m represents a weighted error rate of each of the base classifiers. Therefore, the weighting coefficients α_m give greater weight to more accurate classifiers.

```
1: Init data weights $\{w_n\}$ to $\frac{1}{N}$
2: **for** $m = 1$ to $M$ **do**
3:     fit a classifier $y_m(x)$ by minimizing weighted error function $J_m$:
4:     $J_m = \sum_{n=1}^{N} w_n^{(m)} 1[y_m(x_n) \neq t_n]$
5:     compute $\epsilon_m = \sum_{n=1}^{N} w_n^{(m)} \frac{1[y_m(x_n) \neq t_n]}{\sum_{n=1}^{N} w_n^{(m)}}$
6:     evalutate $\alpha_m = \log\left(\frac{1-\epsilon_m}{\epsilon_m}\right)$
7: update the data weights: $w_n^{(m+1)} = w_n^{(m)} \exp\{\alpha_m 1[y_m(x_n) \neq t_n]\}$
8: **end for**
9: Make predictions using the final model: $Y_M(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(x)\right)$
```

Figure 7.15 AdaBoost algorithm pseudo-code

The following listing trains the AdaBoost classifier with different numbers of learners.

Listing 7.9 Boosting ensemble

```
import itertools
import numpy as np

import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions

def main():

    iris = datasets.load_iris()
    X, y = iris.data[:, 0:2], iris.target

    #XOR dataset
    #X = np.random.randn(200, 2)
    #y = np.array(map(int,np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)))

    clf = DecisionTreeClassifier(criterion='entropy',
    ➥ max_depth=1)                         <-- Base classifier

    num_est = [1, 2, 3, 10]
    label = ['AdaBoost (n_est=1)', 'AdaBoost (n_est=2)', 'AdaBoost
    ➥ (n_est=3)', 'AdaBoost (n_est=10)']

    fig = plt.figure(figsize=(10, 8))
    gs = gridspec.GridSpec(2, 2)
    grid = itertools.product([0,1],repeat=2)

    for n_est, label, grd in zip(num_est, label, grid):
        boosting = AdaBoostClassifier(base_estimator=clf, n_estimators=n_est)
        boosting.fit(X, y)
        ax = plt.subplot(gs[grd[0], grd[1]])
        fig = plot_decision_regions(X=X, y=y, clf=boosting, legend=2)
        plt.title(label)

    plt.show()
    #plt.savefig('./figures/boosting_ensemble.png')

    #plot learning curves
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    ➥ random_state=0)
    boosting = AdaBoostClassifier(base_estimator=clf, n_estimators=10)

    plt.figure()
    plot_learning_curves(X_train, y_train, X_test, y_test, boosting,
    ➥ print_model=False, style='ggplot')
    plt.show()
    #plt.savefig('./figures/boosting_ensemble_learning_curve.png')

    num_est = list(map(int, np.linspace(1,100,20)))   <-- Ensemble size
    bg_clf_cv_mean = []
    bg_clf_cv_std = []
    for n_est in num_est:
        print("num_est: ", n_est)
        ada_clf = AdaBoostClassifier(base_estimator=clf, n_estimators=n_est)
        scores = cross_val_score(ada_clf, X, y, cv=3, scoring='accuracy')
        bg_clf_cv_mean.append(scores.mean())
        bg_clf_cv_std.append(scores.std())

    plt.figure()
    (_, caps, _) = plt.errorbar(num_est, bg_clf_cv_mean,
    ➥ yerr=bg_clf_cv_std, c='blue', fmt='-o', capsize=5)
    for cap in caps:
        cap.set_markeredgewidth(1)
    plt.ylabel('Accuracy'); plt.xlabel('Ensemble Size'); plt
    ➥ .title('AdaBoost Ensemble');
    plt.show()
    #plt.savefig('./figures/boosting_ensemble_size.png')

if __name__ == "__main__":
    main()
```

The boosting ensemble is illustrated in figure 7.16. Each base learner consists of a decision tree with depth 1, thus classifying the data based on a feature threshold that partitions the space into two regions separated by a linear decision surface parallel to one of the axes. The figure also shows how the test accuracy improves with the size of the ensemble and the learning curves for training and testing data.

Gradient tree boosting is a generalization of boosting to arbitrary differentiable loss functions. It can be used for both regression and classification problems. Gradient boosting builds the model in a sequential way.

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x) \tag{7.33}$$

At each stage the decision tree, $h_m(x)$ is chosen to minimize a loss function L given the current model $F_{m-1}(x)$.

$$F_m(x) = F_{m-1}(x) + \arg\min_h \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h(x_i)) \tag{7.34}$$

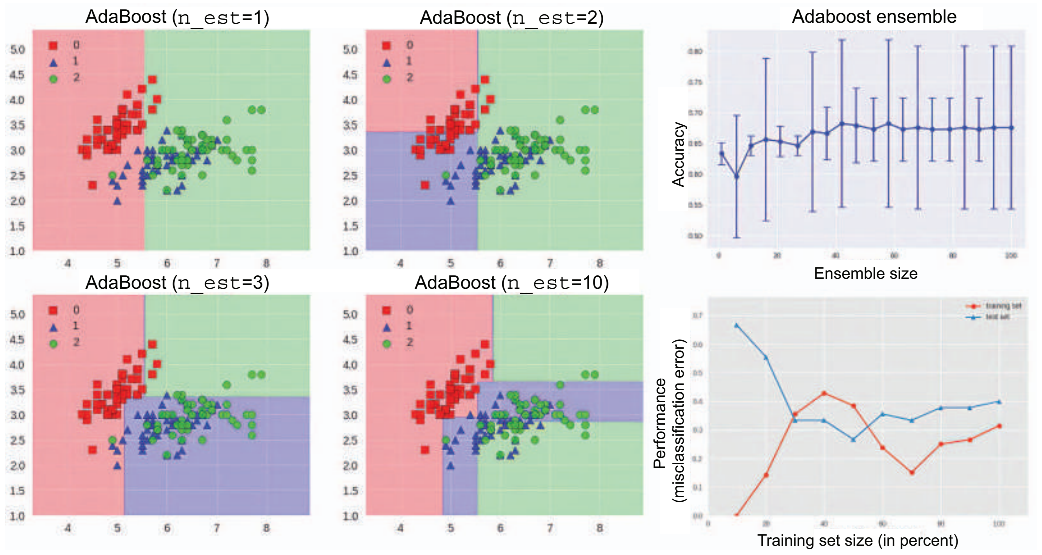

Figure 7.16 Boosting ensemble

Gradient boosting attempts to solve this minimization problem numerically via the steepest descent. The steepest descent direction is the negative gradient of the loss function evaluated at the current model F_{m-1}. The algorithms for regression and classification differ in the type of loss function used.

7.5.3 Stacking

Stacking is an ensemble learning technique that combines multiple classification or regression models via a meta-classifier or meta-regressor. The base level models are trained based on complete training set, and then the meta-model is trained on the outputs of base-level model as features. The base level often consists of different learning algorithms; therefore, stacking ensembles are often heterogeneous. The algorithm in figure 7.17 summarizes stacking.

```
1:  Input: training data $D = \{x_i, y_i\}_{i=1}^m$
2:  Output: ensemble classifier $H$
3:  Step 1: learn base-level classifiers
4:  for $t = 1$ to $T$ do
5:      learn $h_t$ based on $D$
6:  end for
7:  Step 2: construct new data set of predictions
8:  for $i = 1$ to $m$ do
9:      $D_h = \{x'_i, y_i\}$, where $x'_i = \{h_1(x_i), \ldots, h_T(x_i)\}$
10: end for
11: Step 3: learn a meta-classifier
12: learn $H$ based on $D_h$
13: return $H$
```

Figure 7.17 Stacking algorithm pseudo-code

The following listing shows the stacking classifier in action.

Listing 7.10 Stacking ensemble

```
import itertools
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from sklearn import datasets

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from mlxtend.classifier import StackingClassifier

from sklearn.model_selection import cross_val_score, train_test_split

from mlxtend.plotting import plot_learning_curves
from mlxtend.plotting import plot_decision_regions

def main():

    iris = datasets.load_iris()
    X, y = iris.data[:, 1:3], iris.target

    clf1 = KNeighborsClassifier(n_neighbors=1)
    clf2 = RandomForestClassifier(random_state=1)
    clf3 = GaussianNB()
    lr = LogisticRegression()
    sclf = StackingClassifier(classifiers=[clf1, clf2, clf3],
                              meta_classifier=lr)          <- Stacking classifier

    label = ['KNN', 'Random Forest', 'Naive Bayes', 'Stacking Classifier']
    clf_list = [clf1, clf2, clf3, sclf]

    fig = plt.figure(figsize=(10,8))
    gs = gridspec.GridSpec(2, 2)
    grid = itertools.product([0,1],repeat=2)

    clf_cv_mean = []
    clf_cv_std = []
    for clf, label, grd in zip(clf_list, label, grid):

        scores = cross_val_score(clf, X, y, cv=3, scoring='accuracy')
        print("Accuracy: %.2f (+/- %.2f) [%s]" %(scores.mean(),
        ➥ scores.std(), label))
        clf_cv_mean.append(scores.mean())
        clf_cv_std.append(scores.std())

        clf.fit(X, y)
        ax = plt.subplot(gs[grd[0], grd[1]])
        fig = plot_decision_regions(X=X, y=y, clf=clf)
        plt.title(label)

    plt.show()
    #plt.savefig("./figures/ensemble_stacking.png")
    #plot classifier accuracy
    plt.figure()
    (_, caps, _) = plt.errorbar(range(4), clf_cv_mean, yerr=clf_cv_std,
    ➥ c='blue', fmt='-o', capsize=5)
    for cap in caps:
        cap.set_markeredgewidth(1)
    plt.xticks(range(4), ['KNN', 'RF', 'NB', 'Stacking'],
     rotation='vertical')
    plt.ylabel('Accuracy'); plt.xlabel('Classifier');
    ➥ plt.title('Stacking Ensemble');
    plt.show()
    #plt.savefig('./figures/stacking_ensemble_size.png')

    #plot learning curves
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    ➥ test_size=0.3, random_state=0)

    plt.figure()
    plot_learning_curves(X_train, y_train, X_test, y_test, sclf,
    ➥ print_model=False, style='ggplot')
    plt.show()
    #plt.savefig('./figures/stacking_ensemble_learning_curve.png')

if __name__ == "__main__":
    main()
```

The stacking ensemble is illustrated in figure 7.18. It consists of KNN, Random Forest, and naive Bayes base classifiers whose predictions are combined by logistic regression as a meta-classifier. We can see the blending of decision boundaries achieved by the stacking classifier. The figure also shows that stacking achieves higher accuracy than individual classifiers, and based on learning curves, it shows no signs of overfitting.

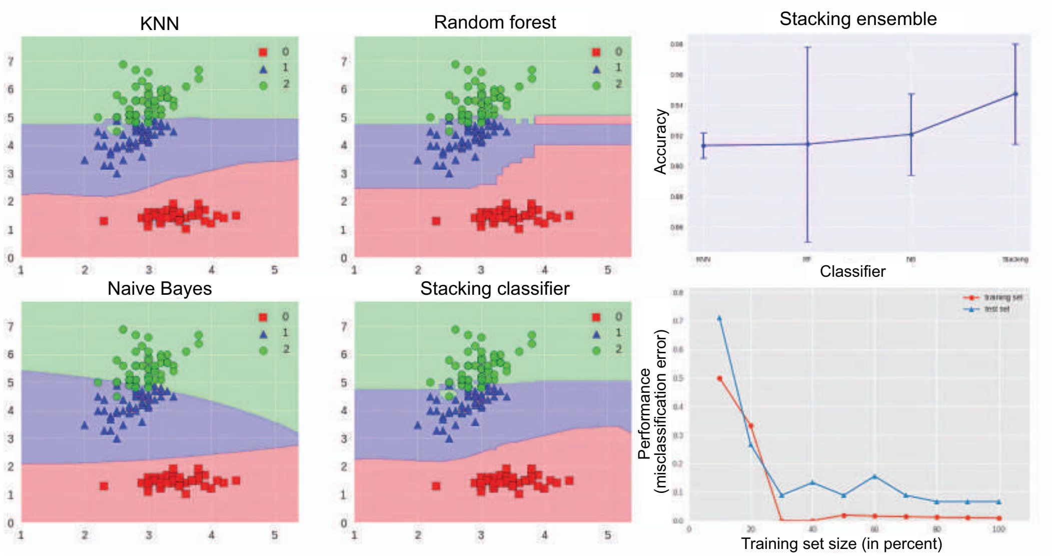

Figure 7.18 Stacking ensemble

7.6 ML research: Supervised learning algorithms

In this section, we cover additional insights and research related to the topics presented earlier in the chapter. In the classification section, we derived several classic algorithms and built a solid foundation for the design of new algorithms. For the perceptron algorithm, we saw how we can modularize the algorithm by introducing different loss functions, leading to slightly different update rules. Moreover, the loss function itself can be weighted to account for imbalanced datasets. Similarly, in the case of SVM, we have the flexibility of choosing the appropriate kernel function. We looked at multiclass SVM; however, a one-class SVM with an RBF kernel can be used for anomaly detection.

There are several extensions to solvers other than SGD for logistic regression, including liblinear (which uses coordinate descent), newton-cg (a second-order method that can lead to faster convergence), and LBFGS (a powerful optimization framework that is both memory and computation efficient). Similarly, there are several extensions to the Naive Bayes algorithm based on the type of data it models, such as multinomial naive Bayes and Gaussian naive Bayes. The derivations for these are very similar to the Bernoulli naive Bayes, which we looked at in detail.

In the section on regression, we had our first encounter with a nonparametric model—namely, the KNN regressor. We will explore this rich set of models in chapter 8, when we discuss Bayesian nonparametric models in which the number of parameters grows with data. For example, in the Dirichlet process (DP) HMM, we have a potentially infinite number of states, with probability mass concentrated on the few states supported by the data (e.g., see Emily B. Fox et al.'s "A Sticky HDP-HMM with Application to Speaker Diarization," *Annals of Applied Statistics*, 2011).

We observed the advantages of using hierarchical models. This technique can be used whenever different data groups have characteristics in common, which allows the sharing of statistical strength from the different data groups. Hierarchical models often provide greater accuracy with fewer observations, such as hierarchical HMM and hierarchical Dirichlet process mixture models (e.g., see Jason Chang and John W. Fisher III's "Parallel Sampling of HDPs using Sub-Cluster Splits," Conference and Workshop on Neural Information Processing Systems, 2014).

We touched on scalable ML when we discussed page rank, due to computations on millions of web pages. Scalable ML is an important area of research. In the industry, Spark ML is typically used to process big data. However, to understand how parallelizable algorithms are constructed, it helps to implement them from scratch. For more information about scalable algorithms, the reader is encouraged to read *Scaling Up Machine Learning: Parallel and Distributed Approaches* by Ron Bekkerman, Mikhail Bilenko, and John Langford (2011).

There are several generalizations of HMMs. One example is a semi-Markov HMM, which models state transitions of variable duration commonly found in genomics data. Another example is a hierarchical HMM, which models data with a hierarchical structure often present in speech applications. Finally, there are factorial HMMs, which consist of multiple interlinked Markov chains running in parallel to capture different aspects of the input signal.

There are several notable research areas within active learning. One such area is semi-supervised learning, which can be used, for example, in self-training. The learner is first trained on a small amount of labeled data and later augments their own dataset with the most confidently classified new training examples. Another are of research is the study of the exploration–exploitation tradeoff, which is commonly present in reinforcement learning. In particular, the active learner must be proactive and discerning when exploring examples of instances they are unsure how to label. Finally, submodular optimization is another important area of research (see Andreas Kraus's "Optimizing Sensing: Theory and Applications," Carnegie Mellon University, School of Computer Science, 2008). In problems with a fixed budget for gathering data, it is advantageous to formulate the objective function for data selection as a submodular function, which could then be optimized using greedy algorithms with performance guarantees.

In the area of model selection, reversible jump (RJ) Markov chain Monte Carlo (MCMC) is an interesting sampling-based method for selecting between models with different numbers of parameters. RJ-MCMC samples in parameter spaces of different dimensionality (e.g., when selecting the best Gaussian mixture model with a different number of clusters K). The interesting part arises in the Metropolis-Hastings sampling when we sample from distributions with different dimensions. RJ-MCMC augments the low dimensional space with extra random variables so that the two spaces have a common measure (see Peter J. Green's "Tutorial on Transdimensional MCMC," Highly Structured Stochastic Systems, 2003). Finally, ensemble methods coupled with model selection and hyperparameter tuning are the winning models in any data science competition!

7.7 Exercises

7.1 Explain how temperature softmax works for different values of the temperature parameter T.

7.2 In the forward–backward HMM algorithm, store the latent state variable z (as part of the HMM class) and compare the inferred z against the ground truth z.

Summary

- Markov models have the Markov property that conditioned on the present state—the future states are independent of the past. In other words, the present state serves as a sufficient statistic for the next state.
- Page rank is a stationary distribution of the Markov chain described by the transition matrix, which is recurrent and aperiodic. At scale, the page rank algorithm is computed using power iterations.
- Hidden Markov models model time series data and consist of a latent state Markov chain, a transition matrix between the latent states, and an emission matrix

that models observed data emitted from each state. Inference is carried out using either the EM algorithm or the forward–backward algorithm with a Viterbi maximum likelihood sequence decoder.

- There are two main strategies for imbalanced learning: undersampling the majority class and oversampling the minority class. An additional method includes introducing class weights in the loss function.
- In active learning, the ML algorithm chooses the data samples to train on in a way that maximizes learning and requires fewer training examples. There are two main query strategies: uncertainty sampling and query by committee.
- Bias–variance tradeoff necessitates that mean squared error is equal to bias squared plus variance. The minimum variance is given by the inverse of Fisher information, which measures the curvature of log likelihood.
- In model selection, we often want to follow the Occam's razor principle and choose the simplest model that explains the data well. In hyperparameter optimization, in addition to grid search and random search, Bayesian Optimization uses an active learning approach in a way that reduces uncertainty and provides a balance between exploration and exploitation.
- Ensemble methods are meta-algorithms that combine several machine learning techniques into one predictive model to decrease variance (bagging), decrease bias (boosting), or improve predictions (stacking).