5 Classification algorithms

This chapter covers

- Introducing classification
- The perceptron algorithm
- The SVM algorithm
- SGD logistic regression
- The Bernoulli naive Bayes algorithm
- The decision tree (CART) algorithm

In the previous chapter, we looked at the computer science fundamentals required to implement ML algorithms from scratch. In this chapter, we focus on supervised learning algorithms. Classification is a fundamental class of algorithms and is widely used in machine learning. We will derive from scratch and implement several selected classification algorithms to build our experience with fundamentals and motivate the design of new ML algorithms. The algorithms in this chapter were selected because they illustrate important algorithmic concepts and expose the reader to progressively more complex scenarios that can be implemented from scratch. These concepts have wide application, including email spam detection, document classification, and customer segmentation.

5.1 Introduction to classification

In *supervised learning*, we are given a dataset $D=\{(x_1, y_1), \ldots, (x_n, y_n)\}$, consisting of tuples of data x and labels y. The goal of a *classification algorithm* is to learn a mapping from inputs x to outputs y, where y is a discrete quantity (i.e., $y \in \{1, \ldots, K\}$). If $K=2$, we have a binary classification problem, while for $K>2$, we have multiclass classification.

A classifier h can be viewed as a mapping between a d-dimensional feature vector $\varphi(x)$ and a k-dimensional label y (i.e., $h: \mathrm{R}^d \rightarrow \mathrm{R}^k$). We often have several models to choose from; let's call this set of classifier models H. Thus, for a given $h \in H$, we can obtain a prediction $y=h(\varphi(x))$. We are typically interested in predicting new or unseen data—in other words, our classifier h must be able to *generalize* to new data samples.

Finding the right classifier is known as *model selection*. We want to choose a model that has a sufficient number of parameters (degrees of freedom) to avoid underfitting or overfitting to training data, as shown in figure 5.1.

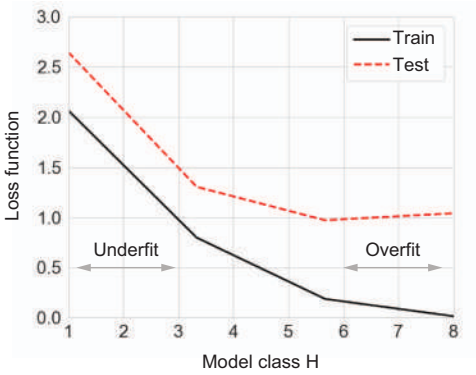

Figure 5.1 Model selection to avoid overfitting or underfitting to training data

For model classes $H=[1, 2, 3]$, training and test loss functions are both decreasing, which indicates that there's more capacity to learn; as a result, these models underfit the data. For model classes $H=[6, 7, 8]$, the training loss decreases while the test loss is starting to increase, which indicates that we are overfitting the data.

5.2 Perceptron

Let's start with the most basic classification model: a *linear classifier*. We'll be using the perceptron classifier on the iris dataset. We can define a linear classifier as shown in equation 5.1.

$$\begin{aligned} h(x;\theta) &= \operatorname{sign}(\theta_1 x_1 + \cdots + \theta_d x_d + \theta_0) \\ &= \operatorname{sign}(\theta \cdot x + \theta_0) = \begin{cases} +1 & \text{if } \theta \cdot x + \theta_0 \geq 0 \\ -1 & \text{if } \theta \cdot x + \theta_0 < 0 \end{cases} \end{aligned} \tag{5.1}$$

Notice how the sign function of the inner product between the parameter θ and the feature input x maps to ±1 labels. Geometrically, $\theta_x + \theta_0 = 0$ describes a hyperplane in d-dimensional space uniquely determined by the normal vector θ. Any point that lies on the same side as the normal θ is labeled +1, while any point on the opposite side is labeled –1. As a result, $\theta_x + \theta_0 = 0$ represents the *decision boundary*. Figure 5.2 illustrates these concepts in 2 dimensions.

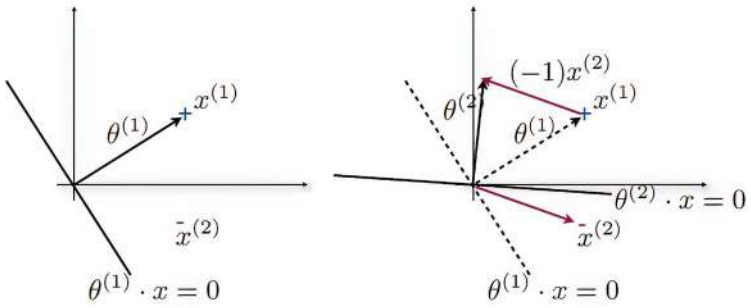

Figure 5.2 Linear classifier a decision boundary

How do we measure the performance of this classifier? One way is to count the number of mistakes it makes compared to ground truth labels y. We can count the number of mistakes as follows.

$$\mathcal{E}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}[[y_i \neq h(x_i;\theta)]] = \frac{1}{n}\sum_{i=1}^{n}[[y_i(\theta \cdot x_i + \theta_0) \leq 0]] \tag{5.2}$$

Here, $[[\bullet]]$ is an indicator function, which is equal to 1 when the expression inside is true and 0 otherwise. Notice in equation 5.2, a mistake occurs whenever the label $y_i \in [+1,-1]$ disagrees with the prediction of the classifier $h(x_i;\theta) \in [+1,-1]$ (i.e., their product is negative).

Another way to measure the performance of a binary classifier is via a confusion matrix.

Figure 5.3 shows the table of errors called the *confusion matrix*. The prediction is correct when the predicted value matches the actual value, as in the case of true positive (TP) and true negative (TN). Similarly, the prediction is wrong when there is a mismatch between predicted and actual values, as in the case of false positive (FP) and false negative (FN). As we vary the classification threshold, we get different values for TP, FP, FN, and TN. To better visualize the performance of the classifier under different classification thresholds, we can construct two additional figures: receiver operating characteristic (ROC) and precision-recall curve (see figure 5.4).

		Actual $y = 1$	Actual $y = 0$
Predicted	$\hat{y} = 1$	TP	FP
	$\hat{y} = 0$	FN	TN

Figure 5.3 Confusion matrix for a binary classifier

In the ROC plot on the left of figure 5.4, *TPR* stands for *true positive rate* and can be computed as follows: TPR = TP/(TP + FN). We can also compute the *false positive rate*

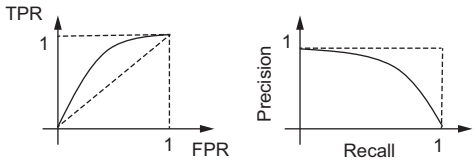

Figure 5.4 Receiver operating characteristic (ROC) plot (left) and precision-recall plot (right)

(FPR) as FPR = FP/(FP + TN). By varying our classification threshold, we get different points along the ROC curve. Perfect classification results in TPR = 1 and FPR = 0; in reality, the closer we are to the upper-left corner, the better the classifier will be. At the chance level, we get the diagonal TPR = FPR line. The quality of ROC curve is often summarized by a single number using the area under the curve or AUC. Higher AUC scores are better, with a maximum of AUC = 1.

In information retrieval, it is common to use a precision-recall plot, as shown on the right-hand side of figure 5.4. The precision is defined as Precision = TP/(TP + FP), and the recall is defined as Recall = TP/(TP + FN). A precision-recall curve is a plot of precision versus recall as we vary the classification threshold. The curve can be summarized by a single number using the mean precision by averaging over the recall values, which approximates the area under the curve. Additionally, for a fixed threshold, we can summarize performance in a single statistic, called the *F1 score*, which is the harmonic mean of precision and recall: F1 = 2PR/(P + R).

Perceptron is a mistake-driven algorithm: it starts with $\theta=0$ and successively adjusts the parameter θ for each training example until there are no more classification mistakes, assuming the data is linearly separable. The perceptron update rule can be summarized as follows.

$$\begin{aligned} \text{if} \quad y_i &\neq h(x_i;\theta^{(k)}) \quad \text{then} \\ \theta^{(k+1)} &= \theta^{(k)} + y_i x_i \\ \theta_0^{(k+1)} &= \theta_0^{(k)} + y_i \end{aligned} \tag{5.3}$$

Here, index k denotes the number of times the parameter updates (aka the number of mistakes). You can think of a θ_0 update as similar to a θ update but with $x=1$. If the training examples are linearly separable, then the perceptron algorithm in equation 5.3 converges after a finite number of iterations. Notice that the order of input data points makes a difference in how parameter θ is learned; therefore, we can randomize (shuffle) the training dataset. In addition, we can introduce a learning rate to help with θ convergence—the properties of which we'll discuss following the implementation. The perceptron algorithm can be summarized in the pseudo-code in figure 5.5.

The code consists of `Perceptron` class with two functions: `fit` and `predict`. In the `fit` function, we take the training data *X* and labels *y*, and upon encountering an error (in which case, the `if` statement condition is `true`), we update the learning rate and update theta, as derived previously. Finally, in the `predict` function, we make a prediction for test data based on the sign of the decision boundary.

1: **class** perceptron
2: **function** fit(X, y):
3: $k = 1$
4: **for** epoch = 1, 2, . . . ,num_epochs
5: **for** $i = 1, 2, \ldots, N$
6: **if** $y_i(\theta \cdot x_i + \theta_0) \leq 0$
7: $\eta = \frac{1}{k+1}$ → **Updates the learning rate**
8: $k+ = 1$
9: $\theta = \theta + \eta\, y_i x_i$ → **Updates the theta**
10: $\theta_0 = \theta_0 + \eta\, y_i$
11: **end if**
12: **end for**
13: **end for**
14: **return** θ, θ_0
15: **function** predict(X):
16: $\hat{y} = \text{sign}\ (\theta \cdot X + \theta_0)$
17: **return** $\hat{y}$

Figure 5.5 Perceptron algorithm pseudo-code

We now have all the tools to implement the perceptron algorithm from scratch! In the following code listing, we classify irises by training the perceptron algorithm on the training feature data and making a prediction based on the test data.

Listing 5.1 Perceptron algorithm

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

from scipy.stats import randint
from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

class perceptron:
    def __init__(self, num_epochs, dim):
      self.num_epochs = num_epochs
      self.theta0 = 0
      self.theta = np.zeros(dim)

    def fit(self, X_train, y_train):
      n = X_train.shape[0]
      dim = X_train.shape[1]

      k = 1
      for epoch in range(self.num_epochs):
          for i in range(n):
              idx = randint.rvs(0, n-1, size=1)[0]    <-- Samples random point
              if (y_train[idx] * (np.dot(self.theta,
              ➥ X_train[idx,:]) + self.theta0) <= 0):    <-- Hinge loss
                  eta = pow(k+1, -1)      <-- Updates learning rate
                  k += 1
```

```
                    self.theta = self.theta + eta *
                    ➥ y_train[idx] * X_train[idx, :]
                    self.theta0 = self.theta0 + eta *
                    ➥ y_train[idx]
                #end if

            print("epoch: ", epoch)
            print("theta: ", self.theta)
            print("theta0: ", self.theta0)
            #end for
        #end for

    def predict(self, X_test):
        n = X_test.shape[0]
        dim = X_test.shape[1]

        y_pred = np.zeros(n)
        for idx in range(n):
            y_pred[idx] = np.sign(np.dot(self.theta, X_test[idx,:]) +
            ➥ self.theta0)
        #end for
        return y_pred

if __name__ == "__main__":

    iris = load_iris()
    X = iris.data[:100,:]
    y = 2*iris.target[:100] - 1

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    ➥ random_state=42)

    #perceptron (binary) classifier
    clf = perceptron(num_epochs=5, dim=X.shape[1])
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    cmt = confusion_matrix(y_test, y_pred)
    acc = np.trace(cmt)/np.sum(np.sum(cmt))
    print("percepton accuracy: ", acc)

    #generate plots
    plt.figure()
    sns.heatmap(cmt, annot=True, fmt="d")
    plt.title("Confusion Matrix"); plt.xlabel("predicted");
     plt.ylabel("actual")
    plt.savefig("./figures/perceptron_acc.png")
    plt.show()
```

Updates theta and theta0

Loads dataset

Maps to {+1,-1} labels

After running the algorithm, we get the classification accuracy results on the test dataset shown in figure 5.6.

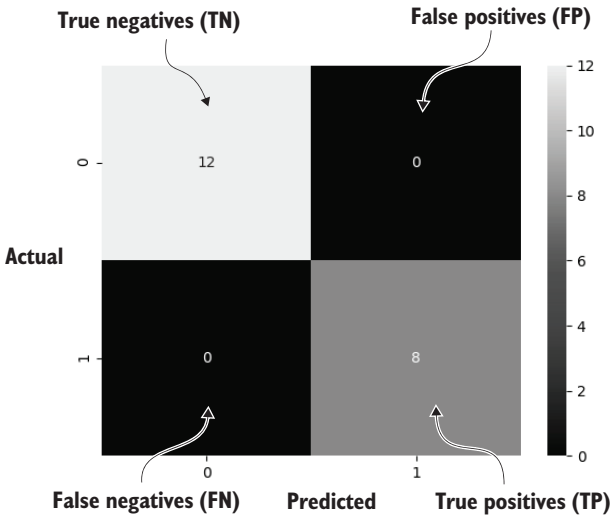

Figure 5.6 Perceptron binary classifier confusion matrix (iris dataset)

Let's take a second look at our implementation and understand it a little better. The perceptron algorithm can be formulated as stochastic gradient descent that minimizes a hinge loss function. Consider a loss function that penalizes the magnitude of disagreement $z_i = y_i(\theta \bullet x_i + \theta_0)$ between the label y_i and the prediction $h(x_i; \theta)$.

$$\text{Loss}_h(z) = \frac{1}{n}\sum_{i=1}^{n} \max\{1 - z_i, 0\} = \frac{1}{n}\sum_{i=1}^{n} \max\{1 - y_i(\theta \cdot x_i + \theta_0), 0\} \qquad (5.4)$$

This is known as a hinge loss function, as illustrated in figure 5.7.

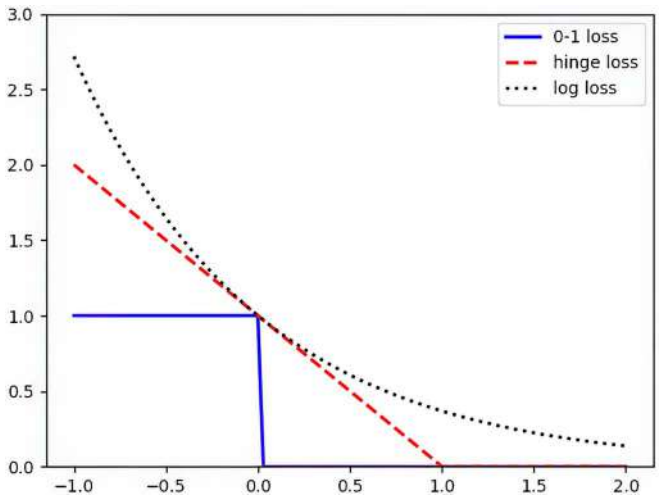

Figure 5.7 Hinge loss, 0–1 loss, and log-loss functions

The stochastic gradient descent attempts to minimize the hinge loss by taking a gradient with respect to θ. However, the max operator is not differentiable at $z_i = 1$. In fact,

we have several possible gradients at that point, which are collectively known as *subdifferential*. Since hinge loss is a piecewise linear function, the gradient for $z_i > 1$ is equal to 0, while the gradient for $z_i \leq 1$ is equal to equation 5.5.

$$\begin{aligned} \nabla_\theta(1 - y_i(\theta \cdot x_i + \theta_0)) &= -y_i x_i \\ \nabla_{\theta_0}(1 - y_i(\theta \cdot x_i + \theta_0)) &= -y_i \end{aligned} \tag{5.5}$$

Combining the expressions in equation 5.5 with a stochastic gradient descent update (where eta is the learning rate), we have equation 5.6.

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla_\theta \text{Loss}_h(y_i(\theta \cdot x_i + \theta_0)) \tag{5.6}$$

We get the perceptron algorithm! In the next section, we'll talk about another important classification algorithm: support vector machine (SVM).

5.3 Support vector machine

In the previous section, we evaluated the performance of our classifier by minimizing the expected loss function (aka empirical risk). One problem with the current formulation is there are multiple classifiers (multiple parameter values θ and θ_0) that can achieve the same empirical risk. So how do we choose the best model, and what does *best* mean?

One solution is to regularize the loss function to favor small parameter values, as shown in equation 5.7.

$$L_n(\theta, \theta_0) = \frac{\lambda}{2}||\theta||^2 + \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y_i(\theta \cdot x_i + \theta_0)) \tag{5.7}$$

Here, the regularization applies to θ but not θ_0. The reason is because θ specifies the orientation of the decision boundary, whereas θ_0 is related to its offset from the origin, which is unknown at the start.

Let's try to understand the decision boundary better from a geometric point of view. It's desirable for the decision boundary, first, to classify all data points correctly and, second, to be maximally removed from all the training examples (i.e., to have the maximum margin). Suppose condition 1 is met and to optimize for condition 2, we need to compute and maximize the distance from every training example to the decision boundary. Geometrically, this distance is as follows.

$$\gamma_i = \frac{y_i(\theta \cdot x_i + \theta_0)}{||\theta||} \tag{5.8}$$

Since we want to maximize the margin, we would like to maximize the minimum distance to the decision boundary across all data points (i.e., find `max[min_i γi]`). This

can be formulated more simply as a quadratic program with linear constraints. A *quadratic program* is a type of mathematical optimization problem that involves optimizing a quadratic objective function subject to linear constraints on the variables.

$$\text{(primal)} \quad \min \frac{1}{2}||\theta||^2 \text{ subject to } y_i(\theta \cdot x_i + \theta_0) \geq 1, \quad i = 1, \ldots, n \tag{5.9}$$

We are essentially minimizing the regularized loss function by choosing θ with a small l_2 norm subject to the constraints that every training example is correctly classified (see figure 5.8).

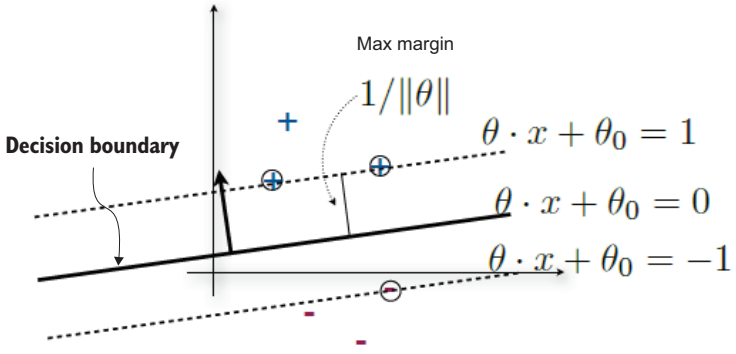

Figure 5.8 Max margin solution of SVM classifier

Notice that as we minimize $||\theta||^2$, we are effectively increasing the distance $\gamma_i \propto 1/||\theta||$ between the decision boundary and the training data points indexed by i. Geometrically, we are pushing the margin boundaries away from each other, as shown in figure 5.8. At some point, they cannot be pushed further without violating classification constraints. At this point, the margin boundaries lock into a unique maximum margin solution. The training data points that lie on the margin boundaries become *support vectors*. We only need a *subset* of training examples (support vectors) to fully learn SVM model parameters.

Let's see if we gain any advantages from solving the *dual* form of the quadratic program. Recall that if the primal is a minimization problem, then the dual is a maximization problem (and vice versa). Additionally, each variable of the original (primal) program becomes a constraint in the dual program, and each constraint in the primal program becomes a variable in the dual program.

We can obtain the dual form by writing out the Lagrangian (i.e., by adding constraints to the objective function with nonnegative Lagrange multipliers).

$$\max_{\alpha \geq 0} L(\theta, \theta_0; \alpha) = \frac{1}{2}||\theta||^2 - \sum_{i=1}^{n} \alpha_i [y_i(\theta \cdot x_i + \theta_0) - 1] \tag{5.10}$$

We can now compute the gradient with respect to our parameters.

$$\begin{aligned} \nabla_\theta L(\theta, \theta_0; \alpha) &= \theta - \sum_{i=1}^{n} \alpha_i y_i x_i = 0 \\ \frac{d}{d\theta_0} L(\theta, \theta_0; \alpha) &= -\sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned} \tag{5.11}$$

By substituting the expression for θ back into our Lagrangian, we get the following.

$$\begin{aligned} \text{(dual)} \quad & \max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \left[x_i^T x_j \right] \\ & \text{subject to} \quad \alpha_i \geq 0, \sum_{i=1}^{n} \alpha_i y_i = 0 \end{aligned} \tag{5.12}$$

The most notable change in the dual formulation is that d-dimensional data points x_i and x_j interact via the inner product. This has significant computational advantages over the primal formulation (in addition to simpler constraints in the dual).

The inner product measures the degree of similarity between two vectors and can be generalized via the kernels $K(x_i, x_j)$. *Kernels* measure a degree of similarity between objects, without explicitly representing them as feature vectors. This is particularly advantageous when we don't have access to or choose not to look into the internals of our objects. Typically, a kernel function that compares two objects $x_i, x_j \in X$ is symmetric $K(x_i, x_j) = K(x_j, x_i)$ and nonnegative $K(x_i, x_j) \geq 0$. There is a wide variety of kernels, ranging from graph kernels that compute the similarity between graphs to string kernels and document kernels. One popular kernel example we will use in our SVM implementation is a radial basis function (RBF) kernel, shown in the following equation.

$$K(x_i, x_j) = \exp\left(- \frac{||x_i - x_j||^2}{2\sigma^2} \right) \tag{5.13}$$

We are now ready to implement a binary SVM classifier from scratch, using the CVXOPT optimization package. CVXOPT is a free software package for convex optimization based on Python programming language and can be downloaded at cvxopt.org.

The standard form of a quadratic program (QP) following CVXOPT notation is shown in the following equation.

$$\min_x \frac{1}{2} x^T P x + q^T x \quad \text{subject to} \quad Gx \preceq h, \; Ax = b \tag{5.14}$$

Note that this objective function is convex if and only if matrix P is positive semidefinite. The CVXOPT QP expects the problem in the form of equation 5.14 parameterized by

$(\mathrm{P}, \mathrm{q}, \mathrm{G}, \mathrm{h}, \mathrm{A}, \mathrm{b})$. Let us convert our dual QP into this form. Let P be a matrix such that the following is true.

$$P_{ij} = y_i y_j \left[x_i^T x_j\right] \tag{5.15}$$

Then, the optimization program becomes the following.

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\alpha^T P \alpha \quad \text{subject to} \quad \alpha_i \geq 0, \ \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{5.16}$$

We can further modify the QP by multiplying the objective and the constraint by –1, which turns this into a minimization problem and reverses the inequality. In addition, we can convert the sum over alphas into a vector form by multiplying the alpha vector by an all-ones vector.

$$\min_{\alpha} \frac{1}{2}\alpha^T P \alpha - 1^T \alpha \quad \text{subject to} \quad -\alpha_i \leq 0, \ y^T \alpha = 0 \tag{5.17}$$

We can now use CVXOPT to solve our SVM quadratic program. Let's start by looking at the pseudo-code in figure 5.9.

```
1: class SupportVectorMachine
2: function fit(X, y):
3:     P_ij = y_i y_j K(x_i, x_j)
4:     q = −1
5:     G_ij = −1                    → Formulates the SVM
6:     h = 0                          Quadratic Program
7:     A = y
8:     b = 0
9:     sol = cvxopt.solvers.qp(P,q,G,h,A,b)   ← Solves with CVXOPT
10:    alphas = sol[x]
11:    S = alphas > 1e − 11          ← Finds support vectors
12:    θ = Σ_{i=1}^{n} y_i α_i x_i   ← Finds the normal vector
13:    θ_0 = y_s − Σ_{m∈s} α_m y_m [x_m^T x_s]   ← Finds the intercept
14:    return θ, θ_0
15: function predict(X, θ, θ_0):
16:    ŷ = sign (θ^T X + θ_0)        ← Makes a prediction
17:    return ŷ
```

Figure 5.9 Support vector machine pseudo-code

The SVM class consists of two functions: `fit` and `predict`. In the `fit` function, we start off by formulating the quadratic problem to be solved by CVXOPT and defining all input parameters: $(\mathrm{P}, \mathrm{q}, \mathrm{G}, \mathrm{h}, \mathrm{A}, \mathrm{b})$. After calling the solver, we find the support vectors

as alphas greater than 0 (up to a rounding error). Next, we compute the normal vector and the intercept, as discussed previously. In the `predict` function, we use the computed normal and intercept support vectors to make a label prediction on test data.

Listing 5.2 SVM algorithm

```
import cvxopt
import numpy as np

from sklearn.svm import SVC          <-- For comparison only
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

def rbf_kernel(gamma, **kwargs):
    def f(x1, x2):
     distance = np.linalg.norm(x1 - x2) ** 2
     return np.exp(-gamma * distance)
    return f

class SupportVectorMachine(object):
  def __init__(self, C=1, kernel=rbf_kernel, power=4, gamma=None):
      self.C = C                          <-- Regularization constant
      self.kernel = kernel
      self.power = power                  <-- Kernel parameters
      self.gamma = gamma
      self.lagr_multipliers = None
      self.support_vectors = None
      self.support_vector_labels = None
      self.intercept = None

  def fit(self, X, y):

      n_samples, n_features = np.shape(X)

      if not self.gamma:
          self.gamma = 1 / n_features

      self.kernel = self.kernel(          <-- Initializes the kernel method with parameters
          power=self.power,
          gamma=self.gamma)

      kernel_matrix = np.zeros((n_samples, n_samples))
      for i in range(n_samples):          <-- Calculates the kernel matrix
          for j in range(n_samples):
              kernel_matrix[i, j] = self.kernel(X[i],
              ➥ X[j])

      P = cvxopt.matrix(np.outer(y, y) * kernel_matrix,
      ➥ tc='d')
      q = cvxopt.matrix(np.ones(n_samples) * -1)
      A = cvxopt.matrix(y, (1, n_samples), tc='d')
      b = cvxopt.matrix(0, tc='d')        <-- Defines the quadratic optimization problem
```

```
        if not self.C: #if its empty
            G = cvxopt.matrix(np.identity(n_samples) * -1)
            h = cvxopt.matrix(np.zeros(n_samples))
        else:
            G_max = np.identity(n_samples) * -1
            G_min = np.identity(n_samples)
            G = cvxopt.matrix(np.vstack((G_max, G_min)))
            h_max = cvxopt.matrix(np.zeros(n_samples))
            h_min = cvxopt.matrix(np.ones(n_samples) * self.C)
            h = cvxopt.matrix(np.vstack((h_max, h_min)))

        minimization = cvxopt.solvers
        ➥ .qp(P, q, G, h, A, b)

        lagr_mult = np.ravel(minimization['x'])

        # Get indexes of non-zero lagr. multipiers
        idx = lagr_mult > 1e-11
        # Get the corresponding lagr. multipliers
        self.lagr_multipliers = lagr_mult[idx]
        # Get the samples that will act as support
        ➥ vectors
        self.support_vectors = X[idx]
        # Get the corresponding labels
        self.support_vector_labels = y[idx]

        self.intercept = self.support_vector_labels[0]
        for i in range(len(self.lagr_multipliers)):
          self.intercept -= self.lagr_multipliers[i] *
       self.support_vector_labels[
            i] * self.kernel(self.support_vectors[i], self.support_vectors[0])


    def predict(self, X):
        y_pred = []
        for sample in X:
            prediction = 0
            # Determine the label of the sample by the support vectors
            for i in range(len(self.lagr_multipliers)):
                prediction += self.lagr_multipliers[i] *
                ➥ self.support_vector_labels[
                    i] * self.kernel(self.support_vectors[i], sample)
            prediction += self.intercept
            y_pred.append(np.sign(prediction))
        return np.array(y_pred)

def main():

    #load dataset
    iris = load_iris()
    X = iris.data[:100,:]
    y = 2*iris.target[:100] - 1
```

Solves the quadratic optimization problem, using cvxopt

Lagrange multipliers

Extracts support vectors

Calculates the intercept with the first support vector

Iterates through list of samples and makes predictions

Maps to {+1,-1} labels

```
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)
    clf = SupportVectorMachine(kernel=rbf_kernel, gamma = 1)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print ("Accuracy (scratch):", accuracy)

    clf_sklearn = SVC(gamma = 'auto')
    clf_sklearn.fit(X_train, y_train)
    y_pred2 = clf_sklearn.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred2)
    print ("Accuracy :", accuracy)

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

We can see that SVM classification accuracy based on our implementation matches the accuracy of sklearn model!

5.4 Logistic regression

Logistic regression is a classification algorithm. Let's dive into some of the theory behind logistic regression before implementing it from scratch! In a probabilistic view of classification, we are interested in computing $p(C_k|x)$ the probability of class label C_k, given the input data x. Consider two classes C_1 and C_2; we can use Bayes rule to compute our posterior probability.

$$p(C_1|x) = \frac{p(x|C_1)p(C_1)}{p(x|C_1)p(C_1) + p(x|C_2)p(C_2)} \tag{5.18}$$

Here, $p(C_k)$ are prior class probabilities. We can divide the right-hand side by the numerator and obtain the following.

$$p(C_1|x) = \frac{1}{1 + \frac{p(x|C_2)p(C_2)}{p(x|C_1)p(C_1)}} = \frac{1}{1 + \exp(-a)} = \sigma(a) \tag{5.19}$$

Here, we defined the following.

$$a = \ln \frac{p(x|C_1)p(C_1)}{p(x|C_2)p(C_2)} \tag{5.20}$$

In the multiclass scenario $(K>2)$, we have the following.

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{\sum_i p(x|C_i)p(C_i)} = \frac{\exp(a_k)}{\sum_i \exp(a_i)} \tag{5.21}$$

In equation 5.21, $a_k = \ln p(x|C_k)p(C_k)$. This expression is also known as a *softmax* function. The softmax function takes a vector of K real values and transforms it into a vector of K real values that sum up to 1. Therefore, the output of the softmax function can be interpreted as a probability distribution. Now, it's a matter of choosing conditional densities that model the data well. In the case of binary logistic regression parameterized by θ and with class label $y = C_k$, we have the following equation.

$$p(C_k|x) = p(y|x, \theta) = \text{Ber}\left(y|\sigma\left(\theta^T x\right)\right) \tag{5.22}$$

We can compute the joint distribution as follows.

$$p(x_i, y_i|\theta) = p(y_i|x_i, \theta)p(x_i|\theta) = \text{Ber}\left(y|\sigma\left(\theta^T x\right)\right)p(x_i|\theta) \tag{5.23}$$

Since we are not modeling the distribution of data $p(x_i|\theta) = p(x_i)$, we can write the log likelihood as follows.

$$\begin{aligned}
\log p(D|\theta) &= \log \prod_{i=1}^{n} p(x_i, y_i|\theta) = \sum_{i=1}^{n} \log p(x_i, y_i|\theta) \\
&= \sum_{i=1}^{n} \log \text{Ber}\left(y|\sigma\left(\theta^T x\right)\right) \\
&= \sum_{i=1}^{n} \log \left[\sigma\left(\theta^T x_i\right)^{y_i}\left(1 - \sigma\left(\theta^T x_i\right)\right)^{1-y_i}\right] \\
&= \sum_{i=1}^{n} \left[y_i \log \sigma\left(\theta^T x_i\right) + (1 - y_i) \log\left(1 - \sigma\left(\theta^T x_i\right)\right)\right]
\end{aligned} \tag{5.24}$$

Note that we are interested in maximizing the log likelihood or, equivalently, minimizing the loss or the negative log likelihood (NLL).

$$\min_{\theta} \text{Loss}(\theta) = \min_{\theta} \text{NLL}(\theta) = \max_{\theta} \log p(D|\theta) \tag{5.25}$$

We are planning on minimizing the logistic regression loss via stochastic gradient descent (SGD), which can be written as follows.

$$\theta_{k+1} = \theta_k - \eta_k g_k \tag{5.26}$$

Here, g_k is the gradient and η_k is the step size. To guarantee the convergence of SGD, the conditions expressed in the following equation, known as *Robbins-Monro conditions*, on the learning rate must be satisfied.

$$\begin{aligned} \sum_{k=1}^{\infty} \eta_k &= \infty \\ \sum_{k=1}^{\infty} \eta_k^2 &< \infty \end{aligned} \tag{5.27}$$

We can use the following learning rate schedule, which satisfies the conditions in equation 5.27.

$$\eta_k = (\tau_0 + k)^{-\kappa} \tag{5.28}$$

Here, $\tau_0 \geq 0$ slows down early iterations of the algorithm and $\kappa \in (0.5, 1]$ controls the rate at which old values are forgotten. To compute the steepest descent direction g_k, we need to differentiate our loss function $\text{NLL}(\theta)$.

$$\begin{aligned} \frac{d}{d\theta} \log p(D|\theta) &= \sum_{i=1}^{n} [y_i \frac{d}{d\theta} \log \sigma \left(\theta^T x_i\right) \\ &\quad + (1 - y_i) \frac{d}{d\theta} \log \left(1 - \sigma \left(\theta^T x_i\right)\right)] \\ &= \sum_{i=1}^{n} [y_i \frac{\sigma \left(\theta^T x_i\right) \left(1 - \sigma \left(\theta^T x_i\right)\right)}{\sigma \left(\theta^T x_i\right)} x_i \\ &\quad + (1 - y_i) \frac{\sigma \left(\theta^T x_i\right) \left(1 - \sigma \left(\theta^T x_i\right)\right)}{1 - \sigma \left(\theta^T x_i\right)} (-x_i)] \\ &= \sum_{i=1}^{n} \left[y_i x_i \left(1 - \sigma \left(\theta^T x_i\right)\right) - (1 - y_i) x_i \sigma \left(\theta^T x_i\right)\right] \\ &= \sum_{i=1}^{n} \left[y_i - \sigma \left(\theta^T x_i\right)\right] x_i \\ &= \sum_{i=1}^{n} [y_i - \mu_i] x_i = -X^T (\mu - y) \end{aligned} \tag{5.29}$$

In equation 5.29, we used the fact that $d/dx\, \sigma(x) = (1 - \sigma(x))\sigma(x)$ and the mean of the Bernoulli distribution $\mu_i = \sigma(\theta^{\mathrm{T}} x_i)$. Note that there exist a number of autograd libraries to avoid deriving the gradients by hand. Furthermore, we can add regularization to control parameter size. Our regularized objective and the gradient become the following.

$$\begin{aligned} \min_{\theta} \text{Loss}(\theta) &= \min_{\theta} [\text{NLL}(\theta) + \lambda \theta^T \theta] \\ g_k &= X^T (\mu - y) + 2\lambda \theta \end{aligned} \tag{5.30}$$

We are now ready to implement SGD for logistic regression. Let's start with the following pseudo-code, as shown in figure 5.10.

```
class sgdlr
function lr_objective(θ, X, y, λ)
μ_i = sigmoid(θ^T X_i)
cost = −Σ_{i=1}^{n} [y_i log μ_i + (1 − y_i) log (1 − μ_i)] + λθ^T θ
grad = X^T (μ − y) + 2λθ ← Computes the gradient
return cost, grad
function fit(X, y):
η_i = (τ + i)^{−κ} ← Sets the learning rate
for i = 1, 2, ... num_iter
    cost, grad = lr_objective(θ, X, y, λ)
    θ = θ − η_i grad ← Updates the theta
end for
return θ
function predict(X, θ):
ŷ = sigmoid (θ^T X) ← Makes a prediction
return ŷ
```

Figure 5.10 Logistic regression algorithm pseudo-code

The `sgdlr` class consists of three main functions: `lr_objective`, `fit`, and `predict`. In the `lr_objective` function, we compute the regularized objective function and the gradient of the objective as discussed in the text. In the `fit` function, we first set the learning rate, and for each iteration, we update the theta parameters in the direction opposite to the gradient. Finally, in the `predict` function, we make a binary prediction of the label based on test data. In the following code listing, we use a synthetic Gaussian mixture dataset to train the logistic regression model.

Listing 5.3 SGD Logistic regression

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

def generate_data():

 n = 1000
 mu1 = np.array([1,1])
 mu2 = np.array([-1,-1])
 pik = np.array([0.4,0.6])

 X = np.zeros((n,2))
 y = np.zeros((n,1))

 for i in range(1,n):
     u = np.random.rand()
     idx = np.where(u < np.cumsum(pik))[0]
```

```
        if (len(idx)==1):
            X[i,:] = np.random.randn(1,2) + mu1
            y[i] = 1
        else:
            X[i,:] = np.random.randn(1,2) + mu2
            y[i] = -1
    return X, y

class sgdlr:

    def __init__(self):

        self.num_iter = 100
        self.lmbda = 1e-9

        self.tau0 = 10
        self.kappa = 1
        self.eta = np.zeros(self.num_iter)

        self.batch_size = 200
        self.eps = np.finfo(float).eps


    def fit(self, X, y):

        theta = np.random.randn(X.shape[1],1)    <-- Random init

Learning rate schedule

        for i in range(self.num_iter):
            self.eta[i] = (self.tau0+i)**(-self.kappa)

        batch_data, batch_labels = self.make_batches(
        ➥ X,y,self.batch_size)                  <-- Divides data into batches
        num_batches = batch_data.shape[0]
        num_updates = 0

        J_hist = np.zeros((self.num_iter * num_batches,1))
        t_hist = np.zeros((self.num_iter * num_batches,1))

        for itr in range(self.num_iter):
            for b in range(num_batches):
                Xb = batch_data[b]
                yb = batch_labels[b]

                J_cost, J_grad = self.lr_objective(theta, Xb, yb, self.lmbda)
                theta = theta - self.eta[itr]*(num_batches*J_grad)

                J_hist[num_updates] = J_cost
                t_hist[num_updates] = np.linalg.norm(theta,2)
                num_updates = num_updates + 1
            print("iteration %d, cost: %f" %(itr, J_cost))

        y_pred = 2*(self.sigmoid(X.dot(theta)) > 0.5) - 1
        y_err = np.size(np.where(y_pred - y)[0])/float(y.shape[0])
        print("classification error:", y_err)
```

```
        self.generate_plots(X, J_hist, t_hist, theta)
        return theta

    def make_batches(self, X, y, batch_size):
        n = X.shape[0]
        d = X.shape[1]
        num_batches = int(np.ceil(n/batch_size))

        groups = np.tile(range(num_batches),batch_size)
        batch_data=np.zeros((num_batches,batch_size,d))
        batch_labels=np.zeros((num_batches,batch_size,1))

        for i in range(num_batches):
            batch_data[i,:,:] = X[groups==i,:]
            batch_labels[i,:] = y[groups==i]

        return batch_data, batch_labels

    def lr_objective(self, theta, X, y, lmbda):   <-- Computes the objective

        n = y.shape[0]
        y01 = (y+1)/2.0

        mu = self.sigmoid(X.dot(theta))

        mu = np.maximum(mu,self.eps)      | Bounds away from
        mu = np.minimum(mu,1-self.eps)    | zero and one

        cost = -(1/n)*np.sum(y01*np.log(mu)+
        ➥ (1-y01)*np.log(1-mu))+np.sum(lmbda*theta*theta)    <-- Computes cost

        grad = X.T.dot(mu-y01) + 2*lmbda*theta    <-- Computes the gradient
                                                      of the lr objective
        #compute the Hessian of the lr objective
        #H = X.T.dot(np.diag(np.diag( mu*(1-mu) ))).dot(X) +
        ➥ 2*lmbda*np.eye(np.size(theta))

        return cost, grad

    def sigmoid(self, a):
        return 1/(1+np.exp(-a))

    def generate_plots(self, X, J_hist, t_hist, theta):

        plt.figure()
        plt.plot(J_hist)
        plt.title("logistic regression")
        plt.xlabel('iterations')
        plt.ylabel('cost')
        #plt.savefig('./figures/lrsgd_loss.png')
        plt.show()

        plt.figure()
        plt.plot(t_hist)
```

```
        plt.title("LR theta l2 norm")
        plt.xlabel('iterations')
        plt.ylabel('theta l2 norm')
        #plt.savefig('./figures/lrsgd_theta_norm.png')
        plt.show()

        plt.figure()
        plt.plot(self.eta)
        plt.title("LR learning rate")
        plt.xlabel('iterations')
        plt.ylabel('learning rate')
        #plt.savefig('./figures/lrsgd_learning_rate.png')
        plt.show()

        plt.figure()
        x1 = np.linspace(np.min(X[:,0])-1,np.max(X[:,0])+1,10)
        plt.scatter(X[:,0], X[:,1])
        plt.plot(x1, -(theta[0]/theta[1])*x1)
        plt.title('LR decision boundary')
        plt.grid(True)
        plt.xlabel('X1')
        plt.ylabel('X2')
        #plt.savefig('./figures/lrsgd_clf.png')
        plt.show()

if __name__ == "__main__":

 X, y = generate_data()
 sgd = sgdlr()
 theta = sgd.fit(X,y)
```

Figure 5.11 shows the stochastic nature of the loss function that decreases with the number of iterations as well as the decision boundary learned by our binary logistic regression.

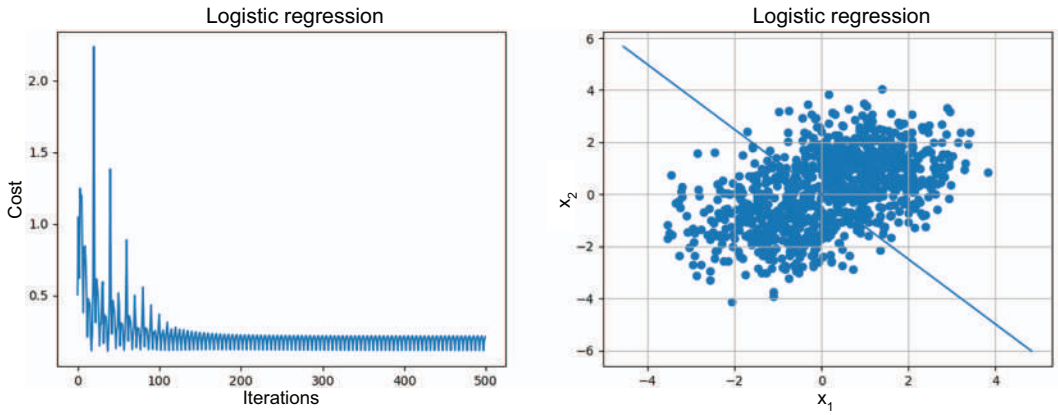

Figure 5.11 SGD logistic regression: Cost (left) and decision boundary (right)

A natural extension to the binary logistic regression is a multinomial logistic regression that handles more than 2 classes.

5.5 Naive Bayes

This section focuses on understanding, deriving, and implementing the naive Bayes algorithm. The fundamental (naive) assumption of the algorithm is that the features are conditionally independent, given the class label. This allows us to write the class conditional density as a product of one-dimensional densities.

$$p(x_i|y = c, \theta) = \prod_{j=1}^{D} p(x_{ij}|y = c, \theta_{jc}) \tag{5.31}$$

The model is called naive because we don't expect the features to be conditionally independent. However, even if the assumption is false, the model performs well in many scenarios. Here, we will focus on Bernoulli Naive Bayes for document classification, with the graphical model shown in figure 5.12. Note the shaded nodes represent the observed variables.

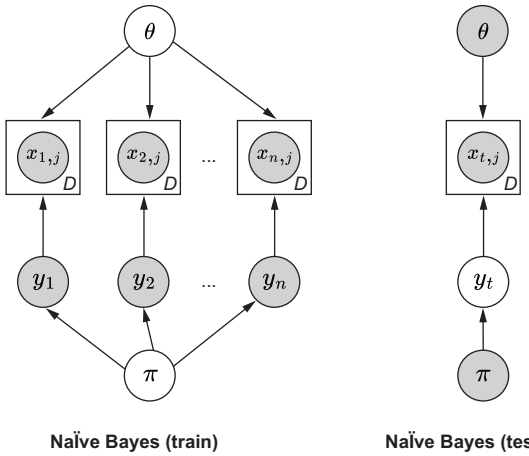

Figure 5.12 Naive Bayes probabilistic graphical model

The choice of class conditional density $p(x|y=c,\theta)$ determines the type of Naive Bayes classifier, such as Gaussian, Bernoulli, or multinomial. In this section, we focus on Bernoulli naive Bayes, due to its high performance in classifying documents.

Let x_{ij} be Bernoulli random variables indicating the presence (x_{ij}=1) or absence (x_{ij}=0) of a word $j \in \{1,...,D\}$ for document $i \in \{1,...,N\}$, parameterized by θ_{jc} for a given class label $y=c \in \{1,...,C\}$. In addition, let π be a Dirichlet distribution representing the prior over the class labels. Thus, the total number of learnable parameters is $|\theta|+|\pi| = O(DC) + O(C) = O(DC)$, where D is the dictionary size and C is the number of classes. Due to the small number of parameters, the Naive Bayes model is immune to overfitting.

We can write down the class conditional density as shown in equation 5.32.

$$p(x|y=c,\theta) = \prod_{i=1}^{n}\prod_{j=1}^{D} p(x_{ij}|y=c,\theta_{jc}) = \prod_{i=1}^{n}\prod_{j=1}^{D} \text{Bernoulli}(\theta_{jc}) \tag{5.32}$$

We can derive the Naive Bayes inference algorithm by maximizing the log likelihood. Consider words x_i in a single document i.

$$\begin{aligned} p(x_i, y_i|\theta) &= p(y_i|\pi)\prod_{j=1}^{D} p(x_{ij}|y_i,\theta) \\ &= \prod_{c=1}^{C} \pi_c^{\mathbb{1}[y_i=c]} \prod_{j=1}^{D}\prod_{c=1}^{C} p(x_{ij}|\theta_{jc})^{\mathbb{1}[y_i=c]} \end{aligned} \tag{5.33}$$

Using the Naive Bayes assumption, we can compute the log likelihood objective.

$$\begin{aligned} \log p(D|\theta) &= \log \prod_{i=1}^{n} p(x_i, y_i|\theta) = \sum_{i=1}^{n} \log p(x_i, y_i|\theta) \\ &= \sum_{c=1}^{C} N_c \log \pi_c + \sum_{j=1}^{D}\sum_{c=1}^{C}\sum_{i:y_i=c} \log p(x_{ij}|\theta_{jc}) \end{aligned} \tag{5.34}$$

Note this is a constrained optimization program, since the probabilities of class labels must sum to one: $\sum \pi_c = 1$. We can solve the optimization problem in equation 5.34 using a Lagrangian by including the constraint in the objective function and setting the gradient of the Lagrangian $L(\theta, \lambda)$ with respect to (wrt) model parameters to zero.

$$L(\theta, \lambda) = \log p(D|\theta) + \lambda\left(1 - \sum_c \pi_c\right) \tag{5.35}$$

Differentiating wrt π_c, we get the following.

$$\frac{d}{d\pi_c} L(\theta, \lambda) = \frac{d}{d\pi_c} \log p(D|\theta) - \lambda = N_c \frac{1}{\pi_c} - \lambda = 0 \tag{5.36}$$

This gives us an expression for π_c in terms of λ: $\pi_c = (1/\lambda)\ N_c$. To solve for λ, we use our sum to one constraint.

$$\sum_c \pi_c = 1 \rightarrow \sum_c \frac{1}{\lambda} N_c = 1 \rightarrow \lambda = \sum_c N_c \tag{5.37}$$

Substituting λ back into expression for π_c, we get $\pi_c = N_c / \sum N_c = N_c / N_{tot}$. Similarly, we can compute the optimum θ_{jc} parameters by setting the gradient of objective wrt θ_{jc} to zero.

$$\begin{aligned}
\frac{d}{d\theta_{jc}} \log p(D|\theta) &= \frac{d}{d\theta_{jc}} \sum_{i:y_i=c} [x_{ij} \log(\theta_{jc}) \\
&\quad + (1 - x_{ij}) \log(1 - \theta_{jc})] = 0 \\
&= \sum_{i:y_i=c} \left[\frac{x_{ij}}{\theta_{jc}} - \frac{1 - x_{ij}}{1 - \theta_{jc}} \right] = 0 \\
\rightarrow \quad \frac{N_{jc}}{\theta_{jc}} &= \frac{1}{1 - \theta_{jc}} [N_c - N_{jc}] \rightarrow N_{jc} = N_c \theta_{jc}
\end{aligned} \tag{5.38}$$

As a result, the optimum maximum likelihood estimate (MLE) value of $\theta_{jc} = N_{jc}/N_c$, where $N_c = \sum \mathbb{1}[y_i = c]$, which makes intuitive sense as a ratio of counts. Note that it's straightforward to add a Beta conjugate prior for the Bernoulli random variables and a Dirichlet conjugate prior for the class density to smooth the MLE counts.

$$\begin{aligned}
p(\pi|D) &= \text{Dir}(N_1 + \alpha_1, \ldots, N_c + \alpha_c) \\
p(\theta_{jc}|D) &= \text{Beta}([N_c - N_{jc}] + \beta_0, N_{jc} + \beta_1)
\end{aligned} \tag{5.39}$$

Here, we use *conjugate prior* for computational convenience, since the posterior and prior have the same form, which enables closed-form updates.

During test time, we would like to predict the class label y given the training data D and the learned model parameters. Applying the Bayes rule, we get the following.

$$\begin{aligned}
p(y = c|x_{i,1}, \ldots, x_{i,D}, D) &\propto p(y = c|D) p(x_{i,1}, \ldots, x_{i,D}|y = c, D) \\
&= p(y = c|D) \prod_{j=1}^{D} p(x_{ij}|y = c, D)
\end{aligned} \tag{5.40}$$

Substituting the distributions for $p(y = c|D)$ and $p(x_{ij}|y = c, D)$ and taking the log, we get the following.

$$\begin{aligned}
\log p(y = c|x, D) &\propto \log \hat{\pi}_c + \sum_{j=1}^{D} \left(\mathbb{1}[x_{ij} = 1] \log \hat{\theta}_{jc} \right. \\
&\quad \left. + \mathbb{1}\left[x_{ij} = 0\right] \log \left(1 - \hat{\theta}_{jc}\right) \right)
\end{aligned} \tag{5.41}$$

Here, π_c and θ_{jc} are the MLE estimates obtained during training. The Naive Bayes algorithm is summarized in the pseudo-code shown in figure 5.13.

The runtime complexity of MLE inference during training is $O(ND)$, where N is the number of training documents and D is the dictionary size. The runtime complexity during test time is $O(TCD)$, where T is the number of test documents, C is the number

Training:
$N_c = 0, N_{jc} = 0$
for $i = 1, 2, \ldots, n$ **do**
  $c = y_i$ //class label for ith example
  $N_c = N_c + 1$
  for $j = 1, \ldots, D$ **do**
    if $x_{ij} = 1$ then
      $N_{jc} = N_{jc} + 1$
  end for
end for
$\hat{\pi}_c = \frac{N_c}{N}, \hat{\theta}_{jc} = \frac{N_{jc}}{N}$
return $\hat{\pi}_c, \hat{\theta}_c$
Testing (for a single test document):
for $c = 1, 2, \ldots, C$ **do**
  $\log p[c] = \log \pi_c$
  for $j = 1, 2, \ldots, D$ **do**
    if $x_j = 1$ then
      $\log p[c] += \log \hat{\theta}_{jc}$
    else
      $\log p[c] += \log (1 - \hat{\theta}_{jc})$
  end for
end for
$c = \arg\max_c \log p[c]$
return c

Figure 5.13 Naive Bayes algorithm pseudo-code

of classes, and D is the dictionary size. Similarly, space complexity is the size of arrays required to store model parameters that grow as $O(DC)$. We are now ready to implement the Bernoulli Naive Bayes algorithm in the following listing!

Listing 5.4 Bernoulli naive Bayes algorithm

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

from time import time
from nltk.corpus import stopwords
from nltk.tokenize import RegexpTokenizer

from sklearn.metrics import accuracy_score
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer

sns.set_style("whitegrid")
tokenizer = RegexpTokenizer(r'\w+')
stop_words = set(stopwords.words('english'))
stop_words.update(['s','t','m','1','2'])
```

```
class naive_bayes:
    def __init__(self, K, D):
      self.K = K          <-- Number of classes
      self.D = D          <-- Dictionary size
Class priors
      self.pi = np.ones(K)
      self.theta = np.ones((self.D, self.K))   <-- Bernoulli parameters

  def fit(self, X_train, y_train):

      num_docs = X_train.shape[0]
      for doc in range(num_docs):

          label = y_train[doc]
          self.pi[label] += 1

          for word in range(self.D):
              if (X_train[doc][word] > 0):
                  self.theta[word][label] += 1
              #end if
          #end for
      #end for

      #normalize pi and theta
      self.pi = self.pi/np.sum(self.pi)
      self.theta = self.theta/np.sum(self.theta, axis=0)

  def predict(self, X_test):

      num_docs = X_test.shape[0]
      logp = np.zeros((num_docs,self.K))
      for doc in range(num_docs):
          for kk in range(self.K):
              logp[doc][kk] = np.log(self.pi[kk])
              for word in range(self.D):
                  if (X_test[doc][word] > 0):
                      logp[doc][kk] += np.log(self.theta[word][kk])
                  else:
                      logp[doc][kk] += np.log(1-self.theta[word][kk])
                  #end if
              #end for
          #end for
      #end for
      return np.argmax(logp, axis=1)

if __name__ == "__main__":

  import nltk
  nltk.download('stopwords')

  #load data
  print("loading 20 newsgroups dataset...")
  tic = time()
  classes = ['sci.space', 'comp.graphics', 'rec.autos', 'rec.sport.hockey']
```

```
dataset = fetch_20newsgroups(shuffle=True, random_state=0,
➥ remove=('headers','footers','quotes'), categories=classes)
X_train, X_test, y_train, y_test = train_test_split(dataset.data,
➥ dataset.target, test_size=0.5, random_state=0)
toc = time()
print("elapsed time: %.4f sec" %(toc - tic))
print("number of training docs: ", len(X_train))
print("number of test docs: ", len(X_test))

print("vectorizing input data...")
cnt_vec = CountVectorizer(tokenizer=tokenizer.tokenize, analyzer='word',
➥ ngram_range=(1,1), max_df=0.8, min_df=2, max_features=1000,
➥ stop_words=stop_words)
cnt_vec.fit(X_train)
toc = time()
print("elapsed time: %.2f sec" %(toc - tic))
vocab = cnt_vec.vocabulary_
idx2word = {val: key for (key, val) in vocab.items()}
print("vocab size: ", len(vocab))

X_train_vec = cnt_vec.transform(X_train).toarray()
X_test_vec = cnt_vec.transform(X_test).toarray()

print("naive bayes model MLE inference...")
K = len(set(y_train)) #number of classes
D = len(vocab) #dictionary size
nb_clf = naive_bayes(K, D)
nb_clf.fit(X_train_vec, y_train)

print("naive bayes prediction...")
y_pred = nb_clf.predict(X_test_vec)
nb_clf_acc = accuracy_score(y_test, y_pred)
print("test set accuracy: ", nb_clf_acc)
```

As we can see from the output, we achieve 82% accuracy on the 20 newsgroups test dataset.

5.6 Decision tree (CART)

This section focuses on the classification and regression trees (CART) algorithm. Tree-based algorithms partition the input space into axis parallel regions such that each leaf represents a region. They can then be used to either classify the region by taking a majority vote or regress the region by computing the expected value. Tree-based models are interpretable and provide insight into feature importance. They are based on a greedy, recursive algorithm, since the optimum partitioning of space is NP complete.

In tree-based models during training, we are interested in constructing a binary tree in a way that optimizes an objective function and does not lead to underfitting or overfitting. A key determinant in growing a decision tree is the choice of the feature and the threshold to use when classifying the data points. Consider an input data matrix $X_{n\times d}$ with n data points of dimension (feature size) d. We would like to find the

optimum feature and threshold for that feature that results in the split of data with minimum cost. Let $j \in \{1, ..., d\}$ represent feature dimension and $t \in \tau_j$ represent a threshold for feature j out of all possible thresholds τ_j (constructed by taking midpoints of our data x_{ij}). Then, we would like to compute the following.

$$j^*, t^* = \arg \min_{j \in \{1,...,d\}} \min_{t \in \tau_j} \text{cost}(\{x_i, y_i : x_{ij} \leq t\}) + \text{cost}(\{x_i, y_i : x_{ij} > t\}) \tag{5.42}$$

Before we look at an example, let's look at potential costs we can use for optimizing the tree for classification. Our goal in defining a cost function is to evaluate how good our data partition is. We would like the leaf nodes to be pure (i.e., contain data from the same class and still be able to generalize to test data). In other words, we would like to limit the depth of the tree (to prevent overfitting) while minimizing impurity. One notion of impurity is the Gini index.

$$\sum_{k=1}^{K} \pi_k (1 - \pi_k) = \sum_k \pi_k - \sum_k \pi_k^2 = 1 - \sum_k \pi_k^2 \tag{5.43}$$

Here, π_k is a fraction of points in the region that belongs to cluster k.

$$\pi_k = \frac{1}{|D|} \sum_{i \in D} 1[y_i = k] \tag{5.44}$$

Notice that since π_k is the probability of a random point in the leaf belonging to class k and $1 - \pi_k$ is the error rate, the Gini index is the expected error rate. If the leaf cluster is pure ($\pi_k = 1$), then the Gini index is zero. Thus, we are interested in minimizing the Gini index.

An alternative objective is entropy, as shown in the following equation.

$$H(\pi) = -\sum_{k=1}^{K} \pi_k \log \pi_k \tag{5.45}$$

Entropy measures the amount of uncertainty. If we are certain that the leaf cluster is pure (i.e., $\pi_k = 1$), then the entropy is zero. Thus, we are interested in minimizing entropy when it comes to CART.

Let's look at a one-dimensional example of choosing the optimum splitting feature and its threshold. Let $X = [1.5, 1.7, 2.3, 2.7, 2.7]$ and class label $y = [1, 1, 2, 2, 3]$. Since the data is one-dimensional, our task is to find a threshold that will split X in a way that minimizes the Gini index. If we choose a threshold $t_1 = 2$ as a midpoint between 1.7 and 2.3 and compute the resulting Gini index, we get the following equation.

$$G = \frac{2}{5}G_{left} + \frac{3}{5}G_{right} = \frac{2}{5} \times 0 + \frac{3}{5} \times \left(1 - \frac{2}{3}^2 - \frac{1}{3}^2\right) = 0.27 \tag{5.46}$$

Here, G_{left} is the Gini index of $\{x_i, y_i\colon x_{ij} \leq 2\}$ and is equal to zero, since both class labels are equal to 1 (i.e., a pure leaf cluster) and G_{right} is the Gini index of $\{x_i, y_i\colon x_{ij} > 2\}$ and contains a mix of class labels $y_{right} = [2,2,3]$.

The key to CART algorithm is finding the optimal feature and threshold such that the cost (e.g., Gini index) is minimized. During training, we'll need to iterate through every feature one by one and compute the Gini cost for all possible thresholds for that feature. But how do we compute τ_j, a set of all possible thresholds for feature j? We can sort the training data $X[:, j]$ in $O(\log n)$ time and consider all midpoints between two adjacent data values. Next, we'll need to compute the Gini index for each threshold that can be done as shown in equation 5.47. Let m be the size of the node and m_k be the number of points in the node that belong to class k.

$$G = 1 - \sum_{k=1}^{K} \pi_k^2 = 1 - \sum_{k=1}^{k} \left(\frac{m_k}{m}\right)^2 \tag{5.47}$$

We can iterate through the sorted thresholds τ_j in $O(n)$ time and compute the Gini index that would result in applying that threshold in each iteration. For i-th threshold, we get the following.

$$\begin{aligned} G_i &= \frac{i}{m}G_i^{left} + \frac{m-i}{m}G_i^{right} \\ G_i^{left} &= 1 - \sum_k \left(\frac{m_k^{left}}{i}\right)^2 \\ G_i^{right} &= 1 - \sum_k \left(\frac{m_k^{right}}{m-i}\right)^2 \end{aligned} \tag{5.48}$$

Having found the optimum feature and threshold, we split each node recursively until the maximum depth is reached. Once we've constructed a tree during training, given test data, we simply traverse the tree from root to leaf, which stores our class label. We can summarize the CART algorithm in the pseudo-code in figure 5.14.

As we can see from the definition of `TreeNode`, it stores the predicted class label, id of the feature to split on and the best threshold to split on, pointers to the left and right subtrees, as well as the Gini cost and size of the node. We can grow the decision tree recursively by calling the `grow_tree` function, as long as the depth of the tree is less than the maximum depth determined ahead of time. First, we compute the class

```
class TreeNode(gini, num_samples, num_samples_class, class_label):
self.gini = gini // gini cost
self.num_samples = num_samples // size of node
self.num_samples_class = num_samples_class //number of pts with label k
self.class_label = class_label //predicted class label
self.feature_idx = 0 //idx of feature to split on
self.threshold = 0 //best threshold to split on
self.left = None //left subtree pointer
self.right = None //right subtree pointer
function grow_tree(X_train, y_train, depth)
class_label = majority_vote(y_train)
gini = compute_gini(y_train)
node = new TreeNode(gini, class_label)
//split recursively until max depth is reached
if depth < max_depth
    idx, threshold = best_split(X_train, y_train)
    if idx is not None:
        indices_left = X_train[:,idx] < threshold
        node.feature_index = idx
        node.left = grow_tree(X_left, y_left, depth + 1)
        node.right = grow_tree(X_right, y_right, depth +1)
return node
```

Figure 5.14 CART decision tree algorithm pseudo-code

label via majority vote and the Gini index for training labels. Next, we determine the best split by iterating over all features and over all possible splitting thresholds. Once we determine the best feature idx and feature threshold to split on, we initialize left and right pointers of the current node with new `TreeNode` objects that contain data less than the splitting threshold and greater than the splitting threshold, respectively. We iterate in this fashion until reaching the maximum tree depth. We are now ready to implement the CART algorithm.

Listing 5.5 CART decision tree algorithm

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

from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

class TreeNode():
    def __init__(self, gini, num_samples, num_samples_class, class_label):
        self.gini = gini
        self.num_samples = num_samples
        self.num_samples_class = num_samples_class
        self.class_label = class_label
```

Number of node points with the label k

Gini cost

Size of the node

Predicted class label

```
        self.feature_idx = 0
        self.treshold = 0
        self.left = None
        self.right = None

class DecisionTreeClassifier():
 def __init__(self, max_depth = None):
     self.max_depth = max_depth

 def best_split(self, X_train, y_train):
     m = y_train.size
     if (m <= 1):
         return None, None

     mk = [np.sum(y_train == k) for k in range(self
     ➥ .num_classes)]

     best_gini = 1.0 - sum((n / m) ** 2 for n in mk)
     best_idx, best_thr = None, None

     #iterate over all features
     for idx in range(self.num_features):

         thresholds, classes = zip(*sorted(zip(X[:,
         ➥ idx], y)))

         num_left = [0]*self.num_classes
         num_right = mk.copy()

         for i in range(1, m):

             k = classes[i-1]

             num_left[k] += 1
             num_right[k] -= 1

             gini_left = 1.0 - sum(
                 (num_left[x] / i) ** 2 for x in range(self.num_classes)
             )

             gini_right = 1.0 - sum(
                 (num_right[x] / (m - i)) ** 2 for x in
                 ➥ range(self.num_classes)
             )

             gini = (i * gini_left + (m - i) * gini_right) / m

             if thresholds[i] == thresholds[i - 1]:
                 continue

             if (gini < best_gini):
                 best_gini = gini
                 best_idx = idx
                 best_thr = (thresholds[i] +
                 ➥ thresholds[i - 1]) / 2
```

idx of the feature to split on

Best threshold to split on

Left subtree pointer

Right subtree pointer

Number of points of class k

Gini of the current node

Sorts data along a selected feature

Iterate over all possible split positions

Midpoint

```
            #end if
        #end for
    #end for
    return best_idx, best_thr

def gini(self, y_train):
    m = y_train.size
    return 1.0 - sum((np.sum(y_train == k) / m) ** 2 for k in
    ➥ range(self.num_classes))

def fit(self, X_train, y_train):
    self.num_classes = len(set(y_train))
    self.num_features = X_train.shape[1]
    self.tree = self.grow_tree(X_train, y_train)

def grow_tree(self, X_train, y_train, depth=0):

    num_samples_class = [np.sum(y_train == k) for k in
    ➥ range(self.num_classes)]
    class_label = np.argmax(num_samples_class)

    node = TreeNode(
        gini=self.gini(y_train),
        num_samples=y_train.size,
        num_samples_class=num_samples_class,
        class_label=class_label,
    )

    if depth < self.max_depth:
        idx, thr = self.best_split(X_train, y_train)
        if idx is not None:
            indices_left = X_train[:, idx] < thr
            X_left, y_left = X_train[indices_left], y_train[indices_left]
            X_right, y_right = X_train[~indices_left],
    y_train[~indices_left]
            node.feature_index = idx
            node.threshold = thr
            node.left = self.grow_tree(X_left, y_left, depth + 1)
            node.right = self.grow_tree(X_right, y_right, depth + 1)

    return node

def predict(self, X_test):
    return [self.predict_helper(x_test) for x_test in X_test]

def predict_helper(self, x_test):
    node = self.tree
    while node.left:
        if x_test[node.feature_index] < node.threshold:
            node = node.left
        else:
            node = node.right
    return node.class_label
```

Split recursively until the maximum depth is reached

```
if __name__ == "__main__":

 #load data
 iris = load_iris()
 X = iris.data[:, [2,3]]
 y = iris.target

 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 ➥ random_state=42)

 print("decision tree classifier...")
 tree_clf = DecisionTreeClassifier(max_depth = 3)
 tree_clf.fit(X_train, y_train)

 print("prediction...")
 y_pred = tree_clf.predict(X_test)

 tree_clf_acc = accuracy_score(y_test, y_pred)
 print("test set accuracy: ", tree_clf_acc)
```

As we can see from the output, we achieve the test classification accuracy of 80% on the iris dataset.

5.7 Exercises

5.1 Given a data point $y \in R^d$ and a hyperplane $\theta \cdot x + \theta_0 = 0$, compute the Euclidean distance from the point to the hyperplane.

5.2 Given a primal linear program (LP) `min` $c^T x$ subject to $Ax <= b$, x >= 0, write down the dual version of the LP.

5.3 Show that the radial basis function (RBF) kernel is equivalent to computing similarity between two infinite dimensional feature vectors.

5.4 Verify that the learning rate schedule $\eta_k = (\tau_0 + k)^{-\kappa}$ satisfies Robbins-Monro conditions.

5.5 Compute the derivative of the sigmoid function $\sigma(a) = [1 + \exp(-a)]^{-1}$.

5.6 Compute the runtime and memory complexity of the Bernoulli naive Bayes algorithm.

Summary

- The goal of a classification algorithm is to learn a mapping from inputs x to outputs y, where y is a discrete quantity.
- Perceptron is a classification algorithm that updates the decision boundary until there are no more classification mistakes.
- SVM is a max-margin classifier. The training data points that lie on the margin boundaries become support vectors.
- Logistic regression is a classification algorithm that computes class conditional density based on a softmax function.

- The naive Bayes algorithm assumes features are conditionally independent, given the class label. It is commonly used in document classification.
- The CART decision tree is a greedy, recursive algorithm that finds the optimum feature splits by minimizing an objective function, such as the Gini index.