2
Building a Movie Recommendation Engine with Naïve Bayes

As promised, in this chapter, we will kick off our supervised learning journey with machine learning classification, and specifically, binary classification. The goal of the chapter is to build a movie recommendation system, which is a good starting point for learning classification from a real-life example—movie streaming service providers are already doing this, and we can do the same.

In this chapter, you will learn the fundamental concepts of classification, including what it does and its various types and applications, with a focus on solving a binary classification problem using a simple, yet powerful, algorithm, Naïve Bayes. Finally, the chapter will demonstrate how to fine-tune a model, which is an important skill that every data science or machine learning practitioner should learn.

We will go into detail on the following topics:

- Getting started with classification
- Exploring Naïve Bayes
- Implementing Naïve Bayes
- Building a movie recommender with Naïve Bayes
- Evaluating classification performance
- Tuning models with cross-validation

Getting started with classification

Movie recommendation can be framed as a machine learning classification problem. If it is predicted that you'll like a movie because you've liked or watched similar movies, for example, then it will be on your recommended list; otherwise, it won't. Let's get started by learning the important concepts of machine learning classification.

Classification is one of the main instances of supervised learning. Given a training set of data containing

observations and their associated categorical outputs, the goal of classification is to learn a general rule that correctly maps the **observations** (also called **features** or **predictive variables**) to the target **categories** (also called **labels** or **classes**). Putting it another way, a trained classification model will be generated after the model learns from the features and targets of training samples, as shown in the first half of *Figure 2.1*. When new or unseen data comes in, the trained model will be able to determine their desired class memberships. Class information will be predicted based on the known input features using the trained classification model, as displayed in the second half of *Figure 2.1*:

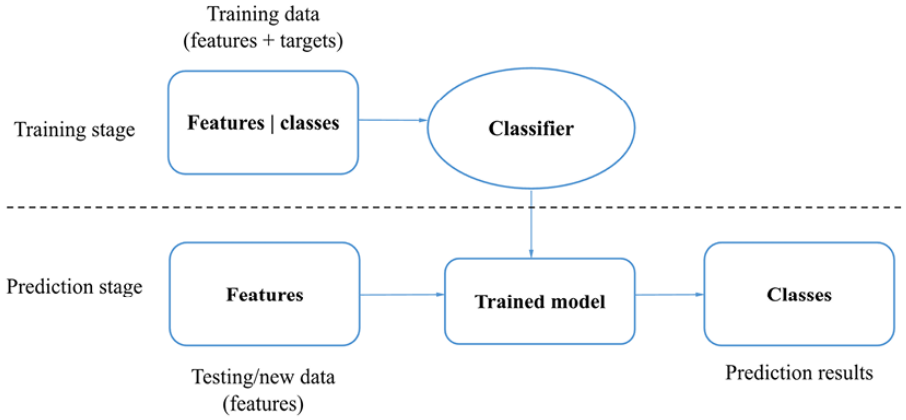

Figure 2.1: The training and prediction stages in classification

In general, there are three types of classification based on the possibility of class output—**binary**, **multi-class**, and **multi-label classification**. We will cover them one by one in this section.

Binary classification

Binary classification classifies observations into one of two possible classes. Spam email filtering we encounter every day is a typical use case of binary classification, which identifies email messages (input observations) as spam or not spam (output classes). Customer churn prediction is another frequently mentioned example, where a prediction system takes in customer segment data and activity data from **customer relationship management** (**CRM**) systems and identifies which customers are likely to churn.

Another application in the marketing and advertising industry is click-through prediction for online ads—that is, whether or not an ad will be clicked, given users' interest information and browsing history. Last but not least, binary classification is also being employed in biomedical science, for example, in early cancer diagnosis, classifying patients into high- or low-risk groups based on MRI images.

As demonstrated in *Figure 2.2*, binary classification tries to find a way to separate data into two classes (denoted by dots and crosses):

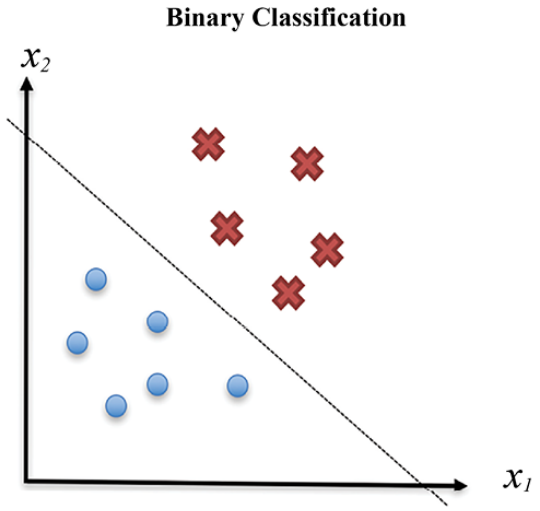

Figure 2.2: Binary classification example

Don't forget that predicting whether a person likes a movie is also a binary classification problem.

Multiclass classification

This type of classification is also referred to as **multinomial classification**. It allows more than two possible classes, as opposed to only two in binary cases. Handwritten digit recognition is a common instance of classification and has a long history of research and development since the early 1900s. A classification system, for example, can learn to read and understand handwritten ZIP codes (digits from 0 to 9 in most countries) by which envelopes are automatically sorted.

Handwritten digit recognition has become a *"Hello, World!"* in the journey of studying machine learning, and the scanned document dataset constructed by the **National Institute of Standards and Technology** (**NIST**), called **Modified National Institute of Standards and Technology** (**MNIST**), is a benchmark dataset frequently used to test and evaluate multiclass classification models. *Figure 2.3* shows four samples taken from the MNIST dataset, representing the digits "`9`," "`2`," "`1`," and "`3`," respectively:

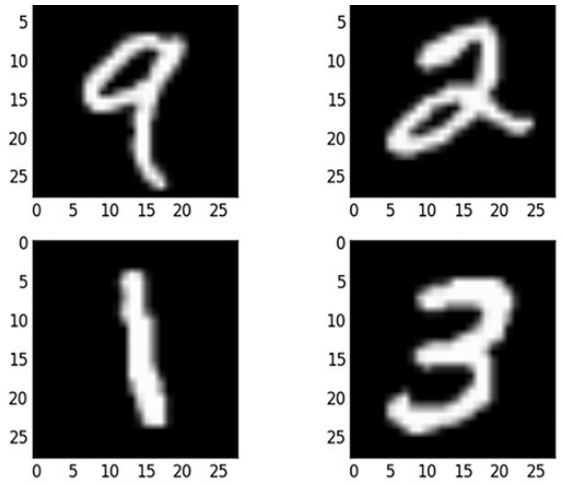

Figure 2.3: Samples from the MNIST dataset

As another example, in *Figure 2.4*, the multiclass classification model tries to find segregation boundaries to separate data into the following three different classes (denoted by dots, crosses, and triangles):

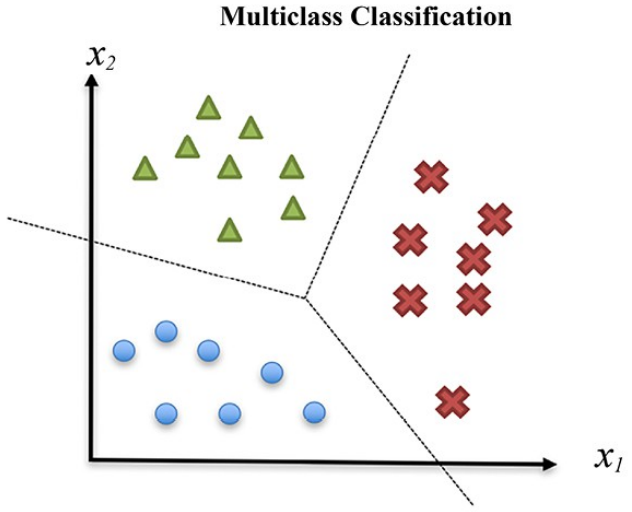

Figure 2.4: Multiclass classification example

Multi-label classification

In the first two types of classification, target classes are mutually exclusive and a sample is assigned *one*,

and only one, label. It is the opposite in multi-label classification. Increasing research attention has been drawn to multi-label classification by the nature of the combination of categories in modern applications. For example, a picture that captures a sea and a sunset can simultaneously belong to both conceptual scenes, whereas it can only be an image of either a cat or dog in a binary case, or one type of fruit among oranges, apples, and bananas in a multiclass case. Similarly, adventure films are often combined with other genres, such as fantasy, science fiction, horror, and drama.

Another typical application is protein function classification, as a protein may have more than one function—storage, antibody, support, transport, and so on.

A typical approach to solving an *n*-label classification problem is to transform it into a set of *n* binary classification problems, where each binary classification problem is handled by an individual binary classifier.

Refer to *Figure 2.5* to see the restructuring of a multi-label classification problem into a multiple-binary classification problem:

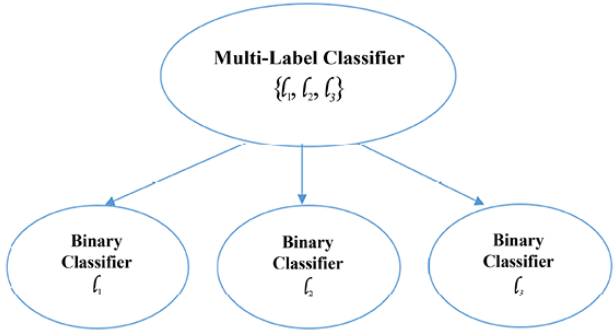

Figure 2.5: Transforming three-label classification into three independent binary classifications

Using the protein function classification example once more, we can transform it into several binary classifications, such as: Is it for storage? Is it for antibodies? Is it for support?

To solve problems like these, researchers have developed many powerful classification algorithms, among which Naïve Bayes, **Support Vector Machines** (**SVMs**), decision trees, logistic regression, and neural networks are often used.

In the following sections, we will cover the mechanics of Naïve Bayes and its in-depth implementation, along with other important concepts, including classifier tuning and classification performance evalua-

along with other important concepts, including classifier tuning and classification performance evaluation. Stay tuned for upcoming chapters that cover the other classification algorithms.

Exploring Naïve Bayes

The **Naïve Bayes** classifier belongs to the family of probabilistic classifiers. It computes the probabilities of each predictive **feature** (also referred to as an **attribute** or **signal**) of the data belonging to each class in order to make a prediction of the probability distribution over all classes. Of course, from the resulting probability distribution, we can conclude the most likely class that the data sample is associated with. What Naïve Bayes does specifically, as its name indicates, is as follows:

- **Bayes**: As in, it maps the probability of observed input features given a possible class to the probability of the class given observed pieces of evidence based on Bayes' theorem.
- **Naïve**: As in, it simplifies probability computation by assuming that predictive features are mutually independent.

I will explain Bayes' theorem with examples in the next section.

Bayes' theorem by example

It is important to understand Bayes' theorem before diving into the classifier. Let *A* and *B* denote any two events. Events could be that *it will rain tomorrow, two kings are drawn from a deck of cards, or a person has cancer*. In Bayes' theorem, *P*(*A* | *B*) is the probability that *A* occurs given that *B* is true. It can be computed as follows:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

Here, *P*(*B* | *A*) is the probability of observing *B* given that *A* occurs, while *P*(*A*) and *P*(*B*) are the probability that *A* and *B* occur, respectively. Is that too abstract? Let's consider the following concrete examples:

- **Example 1**: Given two coins, one is unfair, with 90% of flips getting a head and 10% getting a tail, while the other one is fair. Randomly pick one coin and flip it. What is the probability that this coin is the unfair one, if we get a head?

We can solve this by first denoting *U* for the event of picking the unfair coin, *F* for the fair coin, and *H* for the event of getting a head. So, the probability that the unfair coin has been picked when we get a head, *P(U |H)*, can be calculated with the following:

$$P(U|H) = \frac{P(H|U)P(U)}{P(H)}$$

As we know, $P(H \mid U)$ is 0.9. $P(U)$ is 0.5 because we randomly pick a coin out of two. However, deriving the probability of getting a head, $P(H)$, is not that straightforward, as two events can lead to the following, where U is when the unfair coin is picked, and F is when the fair coin is picked:

$$P(H) = P(H|U)P(U) + P(H|F)P(F)$$

Now, $P(U|H)$ becomes the following:

$$P(U|H) = \frac{P(H|U)P(U)}{P(H)} = \frac{P(H|U)P(U)}{P(H|U)P(U) + P(H|F)P(F)} = \frac{0.9 * 0.5}{0.9 * 0.5 + 0.5 * 0.5} = 0.64$$

So, under Bayes' theorem, the probability that the unfair coin has been picked when we get a head is 0.64.

- **Example 2**: Suppose a physician reported the following cancer screening test scenario among 10,000 people:

	Cancer	No Cancer	Total
Test Positive	80	900	980
Test Negative	20	9000	9020
Total	100	9900	10000

Table 2.1: Example of a cancer screening result

This indicates that 80 out of 100 cancer patients are correctly diagnosed, while the other 20 are not; cancer is falsely detected in 900 out of 9,900 healthy people.

If the result of this screening test on a person is positive, what is the probability that they actually have cancer? Let's assign the event of having cancer and positive testing results as *C* and *Pos*, respectively. So we have $P(Pos \mid C)$ = $80/100$ = 0.8, $P(C)$ = $100/10000$ = 0.01, and $P(Pos)$ = $980/10000$ = 0.098.

We can apply Bayes' theorem to calculate $P(C|Pos)$:

$$P(C|Pos) = \frac{P(Pos|C)P(C)}{P(Pos)} = 8.16\%$$

Given a positive screening result, the chance that the subject has cancer is 8.16%, which is significantly higher than the one under the general assumption (100/10000=1%) without the subject undergoing the screening.

- **Example 3**: Three machines, *A*, *B*, and *C*, in a factory account for 35%, 20%, and 45% of bulb production. The fraction of defective bulbs produced by each machine is 1.5%, 1%, and 2%, respectively. A bulb produced by this factory was identified as defective, which is denoted as event *D*. What are the probabilities that this bulb was manufactured by machine *A*, *B*, or *C*, respectively?

Again, we can simply follow Bayes' theorem:

$$P(A|D) = \frac{P(D|A)P(A)}{P(D)} = \frac{P(D|A)P(A)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)}$$

$$\frac{0.015 * 0.35}{0.015 * 0.35 + 0.01 * 0.2 + 0.02 * 0.45} = 0.323$$

$$P(B|D) = \frac{P(D|B)P(B)}{P(D)} = \frac{P(D|B)P(B)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)}$$

$$\frac{0.01 * 0.2}{0.015 * 0.35 + 0.01 * 0.2 + 0.02 * 0.45} = 0.123$$

$$P(C|D) = \frac{P(D|C)P(C)}{P(D)} = \frac{P(D|C)P(C)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)}$$

$$\frac{0.02 * 0.45}{0.015 * 0.35 + 0.01 * 0.2 + 0.02 * 0.45} = 0.554$$

So, under Bayes' theorem, the probabilities that this bulb was manufactured by machine *A*, *B*, or *C*, are `0.323`, `0.123`, and `0.554` respectively.

Also, either way, we do not even need to calculate $P(D)$ since we know that the following is the case:

$$P(A|D) : P(B|D) : P(C|D) = P(D|A)P(A) : P(D|B)P(B) : P(D|C)P(C) = 21 : 8 : 36$$

We also know the following concept:

$$P(A|D) + P(B|D) + P(C|D) = 1$$

So, we have the following formula:

$$P(A|D) = \frac{21}{21 + 8 + 36} = 0.323$$

$$P(B|D) = \frac{8}{21 + 8 + 36} = 0.133$$

This shortcut approach gave us the same results as the original method, but faster. Now that you understand Bayes' theorem as the backbone of Naïve Bayes, we can easily move forward with the classifier itself.

The mechanics of Naïve Bayes

Let's start by discussing the magic behind the algorithm—how Naïve Bayes works. Given a data sample, x, with n features, $x_1, x_2, \ldots, x_n$ (x represents a feature vector and $x = (x_1, x_2, \ldots, x_n)$), the goal of Naïve Bayes is to determine the probabilities that this sample belongs to each of K possible classes $y_1, y_2, \ldots, y_K$, which is $P(y_K|x)$ or $P(x_1, x_2, \ldots, x_n)$, where $k = 1, 2, \ldots, K$.

This looks no different from what we have just dealt with: x or $x_1, x_2, \ldots, x_n$. This is a joint event where a sample that has observed feature values $x_1, x_2, \ldots, x_n$. y_K is the event that the sample belongs to class k. We can apply Bayes' theorem right away:

$$P(y_k|x) = \frac{P(x|y_k)P(y_k)}{P(x)}$$

Let's look at each component in detail:

- $P(y_k)$ portrays how classes are distributed, with no further knowledge of observation features. Thus, it is also called **prior** in Bayesian probability terminology. Prior can be either predetermined (usually in a uniform manner where each class has an equal chance of occurrence) or learned from a set of training samples.
- $P(y_k|x)$, in contrast to prior $P(y_k)$, is the **posterior**, with extra knowledge of observation.
- $P(x|y_K)$, or $P(x_1, x_2, \ldots, x_n|y_k)$, is the joint distribution of n features, given that the sample belongs to class y_k. This is how likely the features with such values co-occur. This is named **likelihood** in Bayesian terminology. Obviously, the likelihood will be difficult to compute as the number of features increases. In Naïve Bayes, this is solved thanks to the feature independence assumption. The joint conditional dis-

tribution of *n* features can be expressed as the joint product of individual feature conditional distributions:

$$P(x|y_k) = P(x_1|y_k) * P(x_2|y_k) * ... * P(x_n|y_k)$$

Each conditional distribution can be efficiently learned from a set of training samples.

- $P(x)$, also called **evidence**, solely depends on the overall distribution of features, which is not specific to certain classes and is therefore a normalization constant. As a result, posterior is proportional to prior and likelihood:

$$P(y_k|x) \propto P(x|y_k)P(y_k) = P(x_1|y_k) * P(x_2|y_k) * ... * P(x_n|y_k) * P(y_k)$$

Figure 2.6 summarizes how a Naïve Bayes classification model is trained and applied to new data:

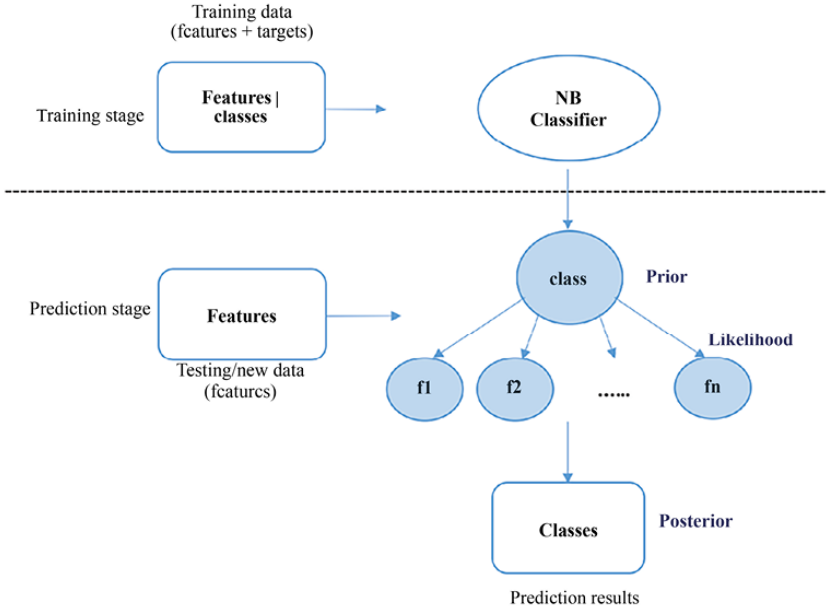

Figure 2.6: Training and prediction stages in Naïve Bayes classification

A Naïve Bayes classification model is trained using labeled data, where each instance is associated with a class label. During training, the model learns the probability distribution of the features given each class. This involves calculating the likelihood of observing each feature value given each class. Once trained, the model can be applied to new, unlabeled data. To classify a new instance, the model calculates the

probability of each class given the observed features using Bayes' theorem.

Let's see a Naïve Bayes classifier in action through a simplified example of movie recommendation before we jump to the implementations of Naïve Bayes. Given four (pseudo) users, whether they like each of three movies, m_1, m_2, and m_3 (indicated as 1 or 0), and whether they like a target movie (denoted as event *Y*) or not (denoted as event *N*), as shown in the following table, we are asked to predict how likely it is that another user will like that movie:

	ID	m1	m2	m3	Whether the user likes the target movie
Training data	1	0	1	1	**Y**
	2	0	0	1	**N**
	3	0	0	0	**Y**
	4	1	1	0	**Y**
Testing case	5	1	1	0	**?**

Table 2.2: Toy data example for a movie recommendation

Whether users like three movies, m_1, m_2, and m_3, are features (signals) that we can utilize to predict the target class. The training data we have are the four samples with both ratings and target information.

Now, let's first compute the prior, $P(Y)$ and $P(N)$. From the training set, we can easily get the following:

$$P(Y) = \frac{3}{4} \quad P(N) = \frac{1}{4}$$

Alternatively, we can also impose an assumption of a uniform prior that $P(Y)$ = 50%, for example.

For simplicity, we will denote the event that a user likes three movies or not as f_1, f_2, and f_3, respectively. To calculate posterior $P(Y| x)$, where x = (1, 1, 0), the first step is to compute the likelihoods, $P(f_1 = 1| Y)$, $P(f_2 = 1 Y)$, and $P(f_3 = 0| Y)$, and similarly, $P(f_1 = 1| N)$, $P(f_2 = 1| N)$, and $P(f_3 = 0| N)$, based on the training set. However, you may notice that since $f_1 = 1$ was not seen in the *N* class, we will get $P(f_1 = 1|N) = 0$.

Consequently, we will have the following:

$$P(N|x) \propto P(f_1 = 1|N) * P(f_2 = 1|N) = 0$$

This means we will recklessly predict class = *Y* by any means.

To eliminate the zero-multiplication factor, the unknown likelihood, we usually assign an initial value of 1 to each feature, that is, we start counting each possible value of a feature from one. This technique is also known as **Laplace smoothing**. With this amendment, we now have the following:

$$P(f_1 = 1|N) = \frac{0+1}{1+2} = \frac{1}{3}$$

$$P(f_1 = 1|Y) = \frac{1+1}{3+2} = \frac{2}{5}$$

Here, given class *N*, 0 + 1 means there are zero likes of m_1 plus + 1 smoothing; 1 + 2 means there is one data point (ID = 2) plus 2 (2 possible values) + 1 smoothing. Given class *Y*, 1 + 1 means there is one like of m_1 (ID = 4) plus + 1 smoothing; 3 + 2 means there are 3 data points (ID = 1, 3, 4) plus 2 (2 possible values) + 1 smoothing.

Similarly, we can compute the following:

$$P(f_2 = 1|N) = \frac{0+1}{1+2} = \frac{1}{3}$$

$$P(f_2 = 1|Y) = \frac{2+1}{3+2} = \frac{3}{5}$$

$$P(f_3 = 0|N) = \frac{0+1}{1+2} = \frac{1}{3}$$

$$P(f_3 = 0|Y) = \frac{2+1}{3+2} = \frac{3}{5}$$

Now, we can compute the ratio between two posteriors as follows:

$$\frac{P(N|x)}{P(Y|x)} \propto \frac{P(N) * P(f_1 = 1|N) * P(f_2 = 1|N) * P(f_3 = 0|N)}{P(Y) * P(f_1 = 1|Y) * P(f_2 = 1|Y) * P(f_3 = 0|Y)} = \frac{125}{1458}$$

Also, remember this:

$$P(N|x) + P(Y|x) = 1$$

So, finally, we have the following:

$P(Y|x) = 92.1\%$

There is a `92.1%` chance that the new user will like the target movie.

I hope that you now have a solid understanding of Naïve Bayes after going through the theory and a toy example. Let's get ready for its implementation in the next section.

Implementing Naïve Bayes

After calculating the movie preference example by hand, as promised, we are going to implement Naïve Bayes from scratch. After that, we will implement it using the `scikit-learn` package.

Implementing Naïve Bayes from scratch

Before we develop the model, let's define the toy dataset we just worked with:

```
>>> import numpy as np
>>> X_train = np.array([
...     [0, 1, 1],
...     [0, 0, 1],
...     [0, 0, 0],
...     [1, 1, 0]])
>>> Y_train = ['Y', 'N', 'Y', 'Y']
>>> X_test = np.array([[1, 1, 0]])
```

For the model, starting with the prior, we first group the data by label and record their indices by classes:

```
>>> def get_label_indices(labels):
...     """
...     Group samples based on their labels and return indices
...     @param labels: list of labels
...     @return: dict, {class1: [indices], class2: [indices]}
...     """
...     from collections import defaultdict
...     label_indices = defaultdict(list)
...     for index, label in enumerate(labels):
```

```
...         label_indices[label].append(index)
...     return label_indices
```

Take a look at what we get:

```
>>> label_indices = get_label_indices(Y_train)
>>> print('label_indices:\n', label_indices)
  label_indices:
  defaultdict(<class 'list'>, {'Y': [0, 2, 3], 'N': [1]})
```

With `label_indices`, we calculate the prior:

```
>>> def get_prior(label_indices):
...     """
...     Compute prior based on training samples
...     @param label_indices: grouped sample indices by class
...     @return: dictionary, with class label as key, corresponding
...             prior as the value
...     """
...     prior = {label: len(indices) for label, indices in
...                                 label_indices.items()}
...     total_count = sum(prior.values())
...     for label in prior:
...         prior[label] /= total_count
...     return prior
```

Take a look at the computed prior:

```
>>> prior = get_prior(label_indices)
>>> print('Prior:', prior)
 Prior: {'Y': 0.75, 'N': 0.25}
```

With `prior` calculated, we continue with `likelihood`, which is the conditional probability, `P(feature|class)`:

```
>>> def get_likelihood(features, label_indices, smoothing=0):
...     """
```

```
...
...     Compute likelihood based on training samples
...     @param features: matrix of features
...     @param label_indices: grouped sample indices by class
...     @param smoothing: integer, additive smoothing parameter
...     @return: dictionary, with class as key, corresponding
...             conditional probability P(feature|class) vector
...             as value
...     """
...     likelihood = {}
...     for label, indices in label_indices.items():
...         likelihood[label] = features[indices, :].sum(axis=0)
...                             + smoothing
...         total_count = len(indices)
...         likelihood[label] = likelihood[label] /
...                             (total_count + 2 * smoothing)
...     return likelihood
```

We set the `smoothing` value to 1 here, which can also be 0 for no smoothing, or any other positive value, as long as a higher classification performance is achieved:

```
>>> smoothing = 1
>>> likelihood = get_likelihood(X_train, label_indices, smoothing)
>>> print('Likelihood:\n', likelihood)
Likelihood:
 {'Y': array([0.4, 0.6, 0.4]), 'N': array([0.33333333, 0.33333333, 0.66666667])}
```

If you ever find any of this confusing, feel free to check *Figure 2.7* to refresh your memory:

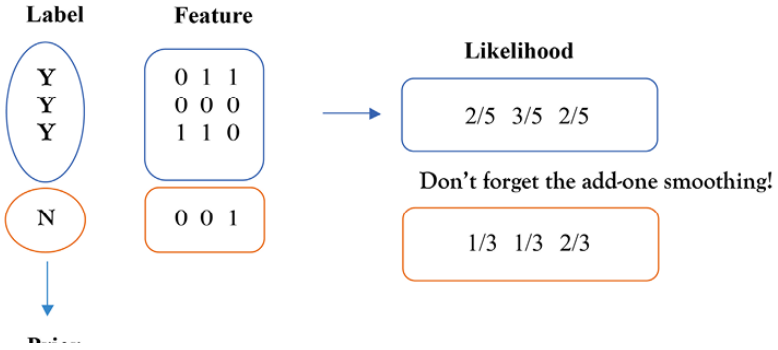

Figure 2.7: A simple example of computing prior and likelihood

With prior and likelihood ready, we can now compute posterior for the testing/new samples:

```
>>> def get_posterior(X, prior, likelihood):
...     """
...     Compute posterior of testing samples, based on prior and
...     likelihood
...     @param X: testing samples
...     @param prior: dictionary, with class label as key,
...                 corresponding prior as the value
...     @param likelihood: dictionary, with class label as key,
...                     corresponding conditional probability
...                       vector as value
...     @return: dictionary, with class label as key, corresponding
...             posterior as value
...     """
...     posteriors = []
...     for x in X:
...         # posterior is proportional to prior * likelihood
...         posterior = prior.copy()
...         for label, likelihood_label in likelihood.items():
...             for index, bool_value in enumerate(x):
...                 posterior[label] *= likelihood_label[index] if
...                     bool_value else (1 - likelihood_label[index])
...         # normalize so that all sums up to 1
...         sum_posterior = sum(posterior.values())
...         for label in posterior:
...             if posterior[label] == float('inf'):
...                 posterior[label] = 1.0
...             else:
...                 posterior[label] /= sum_posterior
...         posteriors.append(posterior.copy())
...     return posteriors
```

Now, let's predict the class of our one sample test set using this prediction function:

```
>>> posterior = get_posterior(X_test, prior, likelihood)
>>> print('Posterior:\n', posterior)
Posterior:
 [{'Y': 0.9210360075805433, 'N': 0.0789639924194567}]
```

This is exactly what we got previously. We have successfully developed Naïve Bayes from scratch and we can now move on to the implementation using `scikit-learn`.

Implementing Naïve Bayes with scikit-learn

Coding from scratch and implementing your own solutions is the best way to learn about machine learning models. Of course, you can take a shortcut by directly using the `BernoulliNB` module (**https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html**) from the scikit-learn API:

```
>>> from sklearn.naive_bayes import BernoulliNB
```

Let's initialize a model with a smoothing factor (specified as `alpha` in `scikit-learn`) of `1.0`, and `prior` learned from the training set (specified as `fit_prior=True` in `scikit-learn`):

```
>>> clf = BernoulliNB(alpha=1.0, fit_prior=True)
```

To train the Naïve Bayes classifier with the `fit` method, we use the following line of code:

```
>>> clf.fit(X_train, Y_train)
```

To obtain the predicted probability results with the `predict_proba` method, we use the following lines of code:

```
>>> pred_prob = clf.predict_proba(X_test)
>>> print('[scikit-learn] Predicted probabilities:\n', pred_prob)
[scikit-learn] Predicted probabilities:
 [[0.07896399 0.92103601]]
```

Finally, we do the following to directly acquire the predicted class with the `predict` method (0.5 is the default threshold, and if the predicted probability of class `Y` is greater than 0.5, class `Y` is assigned; otherwise, `N` is used):

```
>>> pred = clf.predict(X_test)
>>> print('[scikit-learn] Prediction:', pred)
[scikit-learn] Prediction: ['Y']
```

The prediction results using scikit-learn are consistent with what we got using our own solution. Now that we've implemented the algorithm both from scratch and using `scikit-learn`, why don't we use it to solve the movie recommendation problem?

Building a movie recommender with Naïve Bayes

After the toy example, it is now time to build a movie recommender (or, more specifically, movie preference classifier) using a real dataset. We herein use a movie rating dataset (**https://grouplens.org/datasets/movielens/**). The movie rating data was collected by the GroupLens Research group from the MovieLens website (**http://movielens.org**).

For demonstration purposes, we will use the stable small dataset, MovieLens 1M Dataset (which can be downloaded from **https://files.grouplens.org/datasets/movielens/ml-1m.zip** or **https://grouplens.org/datasets/movielens/1m/**) for `ml-1m.zip` (size: 1 MB) file). It has around 1 million ratings, ranging from 1 to 5 with half-star increments, given by 6,040 users on 3,706 movies (last updated September 2018).

Unzip the `ml-1m.zip` file and you will see the following four files:

- `movies.dat`: It contains the movie information in the format of `MovieID::Title::Genres`.
- `ratings.dat`: It contains user movie ratings in the format of `UserID::MovieID::Rating::Timestamp`. We will only be using data from this file in this chapter.
- `users.dat`: It contains user information in the format of `UserID::Gender::Age::Occupation::Zip-code`.
- `README`

Let's attempt to predict whether a user likes a particular movie based on how they rate other movies (again, ratings are from 1 to 5).

Preparing the data

First, we import all the necessary modules and read the `ratings.dat` into a `pandas` DataFrame object:

```
>>> import numpy as np
>>> import pandas as pd
>>> data_path = 'ml-1m/ratings.dat'
>>> df = pd.read_csv(data_path, header=None, sep='::', engine='python')
>>> df.columns = ['user_id', 'movie_id', 'rating', 'timestamp']
>>> print(df)
         user_id  movie_id  rating  timestamp
0              1      1193       5  978300760
1              1       661       3  978302109
2              1       914       3  978301968
3              1      3408       4  978300275
4              1      2355       5  978824291
...          ...       ...     ...        ...
1000204     6040      1091       1  956716541
1000205     6040      1094       5  956704887
1000206     6040       562       5  956704746
1000207     6040      1096       4  956715648
1000208     6040      1097       4  956715569
[1000209 rows x 4 columns]
```

Now, let's see how many unique users and movies are in this million-row dataset:

```
>>> n_users = df['user_id'].nunique()
>>> n_movies = df['movie_id'].nunique()
>>> print(f"Number of users: {n_users}")
Number of users: 6040
>>> print(f"Number of movies: {n_movies}")
Number of movies: 3706
```

Next, we will construct a 6,040 (the number of users) by 3,706 (the number of movies) matrix where each row contains movie ratings from a user, and each column represents a movie, using the following function:

```
>>> def load_user_rating_data(df, n_users, n_movies):
...     data = np.zeros([n_users, n_movies], dtype=np.intc)
        movie_id_mapping = {}
        for user_id, movie_id, rating in zip(df['user_id'], df['movie_id'], df['rating']):
            user_id = int(user_id) - 1
            if movie_id not in movie_id_mapping:
                movie_id_mapping[movie_id] = len(movie_id_mapping)
            data[user_id, movie_id_mapping[movie_id]] = rating
        return data, movie_id_mapping
>>> data, movie_id_mapping = load_user_rating_data(df, n_users, n_movies)
```

Besides the rating matrix `data`, we also record the `movie ID` to column index mapping. The column index is from 0 to 3,705 as we have 3,706 movies.

It is always recommended to analyze the data distribution in order to identify if there is a class imbalance issue in the dataset. We do the following:

```
>>> values, counts = np.unique(data, return_counts=True)
... for value, count in zip(values, counts):
...     print(f'Number of rating {value}: {count}')
Number of rating 0: 21384031
Number of rating 1: 56174
Number of rating 2: 107557
Number of rating 3: 261197
Number of rating 4: 348971
Number of rating 5: 226310
```

As you can see, most ratings are unknown; for the known ones, 35% are of rating 4, followed by 26% of rating 3, 23% of rating 5, and then 11% and 6% of ratings 2 and 1, respectively.

Since most ratings are unknown, we take the movie with the most known ratings as our target movie for easier prediction validation. We look for rating counts for each movie as follows:

```
>>> print(df['movie_id'].value_counts())
2858   3428
260    2991
1196   2990
```

```
1210   2883
480    2672
       ...
3458      1
2226      1
1815      1
398       1
2909      1
Name: movie_id, Length: 3706, dtype: int64
```

So, the target movie is ID, and we will treat ratings of other movies as features. We only use rows with ratings available for the target movie so we can validate how good the prediction is. We construct the dataset accordingly as follows:

```
>>> target_movie_id = 2858
>>> X_raw = np.delete(data, movie_id_mapping[target_movie_id], axis=1)
>>> Y_raw = data[:, movie_id_mapping[target_movie_id]]
>>> X = X_raw[Y_raw > 0]
>>> Y = Y_raw[Y_raw > 0]
>>> print('Shape of X:', X.shape)
Shape of X: (3428, 3705)
>>> print('Shape of Y:', Y.shape)
Shape of Y: (3428,)
```

We can consider movies with ratings greater than 3 as being liked (being recommended):

```
>>> recommend = 3
>>> Y[Y <= recommend] = 0
>>> Y[Y > recommend] = 1
>>> n_pos = (Y == 1).sum()
>>> n_neg = (Y == 0).sum()
>>> print(f'{n_pos} positive samples and {n_neg} negative samples.')
2853 positive samples and 575 negative samples.
```

As a rule of thumb in solving classification problems, we need to always analyze the label distribution and see how balanced (or imbalanced) the dataset is.

Best practice

Dealing with imbalanced datasets in classification problems requires careful consideration and appropriate techniques to ensure that the model effectively learns from the data and produces reliable predictions. Here are several strategies to address class imbalance:

- **Oversampling**: We can increase the number of instances in the minority class by generating synthetic samples or duplicating existing ones.
- **Undersampling**: We can decrease the number of instances in the majority class by randomly removing samples. Note that we can even combine oversampling and undersampling for a more balanced dataset.
- **Class weighting**: We can also assign higher weights to minority class samples during model training. In this way, we penalize misclassifications of the minority class more heavily.

Next, to comprehensively evaluate our classifier's performance, we can randomly split the dataset into two sets, the training and testing sets, which simulate learning data and prediction data, respectively. Generally, the proportion of the original dataset to include in the testing split can be 20%, 25%, 30%, or 33.3%.

Best practice

Here are some guidelines for choosing the testing split:

- **Small datasets**: If you have a small dataset (e.g., less than a few thousand samples), a larger testing split (e.g., 25% to 30%) may be appropriate to ensure that you have enough data for training and testing.
- **Medium to large datasets**: For medium to large datasets (e.g., tens of thousands to millions of samples), a smaller testing split (e.g., 20%) may still provide enough data for evaluation while allowing more data to be used for training. A 20% testing split is a common choice in such cases.
- **Simple models**: Less complex models are generally less prone to overfitting, so using a smaller test set split may work.
- **Complex models**: Complex models like deep learning models can be more prone to overfitting. Hence, a larger test set split (e.g., 30%) is recommended.

We use the `train_test_split` function from `scikit-learn` to do the random splitting and to preserve the percentage of samples for each class:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
...    test_size=0.2, random_state=42)
```

It is a good practice to assign a fixed `random_state` (for example, `42`) during experiments and exploration in order to guarantee that the same training and testing sets are generated every time the program runs. This allows us to make sure that the classifier functions and performs well on a fixed dataset before we incorporate randomness and proceed further.

We check the training and testing sizes as follows:

```
>>> print(len(Y_train), len(Y_test))
2742 686
```

Another good thing about the `train_test_split` function is that the resulting training and testing sets will have the same class ratio.

Training a Naïve Bayes model

Next, we train a Naïve Bayes model on the training set. You may notice that the values of the input features are from 0 to 5, as opposed to 0 or 1 in our toy example. Hence, we use the `MultinomialNB` module (**https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html**) from scikit-learn instead of the `BernoulliNB` module, as `MultinomialNB` can work with integer features as well as fractional counts. We import the module, initialize a model with a smoothing factor of `1.0` and `prior` learned from the training set, and train this model against the training set as follows:

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> clf = MultinomialNB(alpha=1.0, fit_prior=True)
>>> clf.fit(X_train, Y_train)
```

Then, we use the trained model to make predictions on the testing set. We get the predicted probabilities as follows:

```
>>> prediction_prob = clf.predict_proba(X_test)
>>> print(prediction_prob[0:10])
[[7.50487439e-23 1.00000000e+00]
 [1.01806208e-01 8.98193792e-01]
 [3.57740570e-10 1.00000000e+00]
 [1.00000000e+00 2.94095407e-16]
 [1.00000000e+00 2.49760836e-25]
 [7.62630220e-01 2.37369780e-01]
 [3.47479627e-05 9.99965252e-01]
 [2.66075292e-11 1.00000000e+00]
 [5.88493563e-10 9.99999999e-01]
 [9.71326867e-09 9.99999990e-01]]
```

For each testing sample, we output the probability of class 0, followed by the probability of class 1.

We get the predicted class for the test set as follows:

```
>>> prediction = clf.predict(X_test)
>>> print(prediction[:10])
[[1. 1. 1. 0. 0. 0. 1. 1. 1. 1.]
```

Finally, we evaluate the model's performance with classification accuracy, which is the proportion of correct predictions:

```
>>> accuracy = clf.score(X_test, Y_test)
>>> print(f'The accuracy is: {accuracy*100:.1f}%')
The accuracy is: 71.6%
```

The classification accuracy is around 72%, which means that the Naïve Bayes classifier we've constructed accurately suggests movies to users about three quarters of the time. Ideally, we could also utilize movie genre information from the `movies.dat` file, and user demographics (gender, age, occupation, and ZIP code) information from the `users.dat` file. Obviously, movies in similar genres tend to attract similar users, and users of similar demographics likely have similar movie preferences. We will leave it as an exercise for you to explore further.

So far, we have covered in depth the first machine learning classifier and evaluated its performance by

So far, we have covered in depth the first machine learning classifier and evaluated its performance by prediction accuracy. Are there any other classification metrics? Let's see in the next section.

Evaluating classification performance

Beyond accuracy, there are several metrics we can use to gain more insight and avoid class imbalance effects. These are as follows:

- Confusion matrix
- Precision
- Recall
- F1 score
- The area under the curve

A **confusion matrix** summarizes testing instances by their predicted values and true values, presented as a contingency table:

		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP

TN = True Negative
FP = False Positive
FN = False Negative
TP =True Positive

Figure 2.8: Contingency table for a confusion matrix

To illustrate this, we can compute the confusion matrix of our Naïve Bayes classifier. We use the `confusion_matrix` function from `scikit-learn` to compute it, but it is very easy to code it ourselves:

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(Y_test, prediction, labels=[0, 1]))
[[ 60  47]
 [148 431]]
```

As you can see from the resulting confusion matrix, there are 47 false positive cases (where the model misinterprets a dislike as a like for a movie), and 148 false negative cases (where it fails to detect a like for a movie). Hence, classification accuracy is just the proportion of all true cases:

$$\frac{TN + TP}{TN + TP + FP + FN} = \frac{60 + 431}{60 + 431 + 47 + 148} = 71.6\%$$

Precision measures the fraction of positive calls that are correct, which are the following, in our case:

$$\frac{TP}{TP+FP} = \frac{431}{431+47} = 0.90$$

Recall, on the other hand, measures the fraction of true positives that are correctly identified, which are the following in our case:

$$\frac{TP}{TP+FN} = \frac{431}{431+148} = 0.74$$

Recall is also called the **true positive rate**.

The **f1 score** comprehensively includes both the precision and the recall and equates to their **harmonic mean**:

$$f_1 = 2 * \frac{precision * recall}{precision + recall}$$

We tend to value the **f1** score above precision or recall alone.

Let's compute these three measurements using corresponding functions from `scikit-learn`, as follows:

```
>>> from sklearn.metrics import precision_score, recall_score, f1_score
>>> precision_score(Y_test, prediction, pos_label=1)
0.9016736401673641
>>> recall_score(Y_test, prediction, pos_label=1)
0.7443868739205527
>>> f1_score(Y_test, prediction, pos_label=1)
0.815515610217597
```

On the other hand, the negative (dislike) class can also be viewed as positive, depending on the context. For example, assign the `0` class as `pos_label` and we have the following:

```
>>> f1_score(Y_test, prediction, pos_label=0)
0.38095238095238093
```

To obtain the precision, recall, and f1 score for each class, instead of exhausting all class labels in the three function calls as shown earlier, a quicker way is to call the `classification_report` function:

```
>>> from sklearn.metrics import classification_report
>>> report = classification_report(Y_test, prediction)
>>> print(report)
              precision    recall  f1-score   support

         0.0       0.29      0.56      0.38       107
         1.0       0.90      0.74      0.82       579

   micro avg       0.72      0.72      0.72       686
   macro avg       0.60      0.65      0.60       686
weighted avg       0.81      0.72      0.75       686
```

Here, `weighted avg` is the weighted average according to the proportions of the class.

The classification report provides a comprehensive view of how the classifier performs on each class. It is, as a result, useful in imbalanced classification, where we can easily obtain high accuracy by simply classifying every sample as the dominant class, while the precision, recall, and f1 score measurements for the minority class, however, will be significantly low.

Precision, recall, and the f1 score are also applicable to **multiclass** classification, where we can simply treat a class we are interested in as a positive case, and any other classes as negative cases.

During the process of tweaking a binary classifier (that is, trying out different combinations of hyperparameters, for example, the smoothing factor in our Naïve Bayes classifier), it would be perfect if there was a set of parameters in which the highest averaged and class individual f1 scores are achieved at the same time. It is, however, usually not the case. Sometimes, a model has a higher average f1 score than another model, but a significantly low f1 score for a particular class; sometimes, two models have the same average f1 scores, but one has a higher f1 score for one class and a lower score for another class. In situations such as these, how can we judge which model works better? The **Area Under the Curve** (**AUC**) of the **Receiver Operating Characteristic** (**ROC**) is a consolidated measurement frequently used in binary classification.

The ROC curve is a plot of the true positive rate versus the false positive rate at various probability thresholds, ranging from 0 to 1. For a testing sample, if the probability of a positive class is greater than the threshold, then a positive class is assigned; otherwise, we use a negative class. To recap, the true positive rate is equivalent to recall, and the false positive rate is the fraction of negatives that are incorrectly identified as positive. Let's code and exhibit the ROC curve (under thresholds of `0.0`, `0.1`, `0.2`,

..., `1.0`) of our model:

```
>>> pos_prob = prediction_prob[:, 1]
>>> thresholds = np.arange(0.0, 1.1, 0.05)
>>> true_pos, false_pos = [0]*len(thresholds), [0]*len(thresholds)
>>> for pred, y in zip(pos_prob, Y_test):
...     for i, threshold in enumerate(thresholds):
...         if pred >= threshold:
...             # if truth and prediction are both 1
...             if y == 1:
...                 true_pos[i] += 1
...             # if truth is 0 while prediction is 1
...             else:
...                 false_pos[i] += 1
...         else:
...             break
```

Then, let's calculate the true and false positive rates for all threshold settings (remember, there are `516.0` positive testing samples and `1191` negative ones):

```
>>> n_pos_test = (Y_test == 1).sum()
>>> n_neg_test = (Y_test == 0).sum()
>>> true_pos_rate = [tp / n_pos_test for tp in true_pos]
>>> false_pos_rate = [fp / n_neg_test for fp in false_pos]
```

Now, we can plot the ROC curve with `matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> lw = 2
>>> plt.plot(false_pos_rate, true_pos_rate,
...          color='darkorange', lw=lw)
>>> plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
>>> plt.xlim([0.0, 1.0])
>>> plt.ylim([0.0, 1.05])
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
```

```
>>> plt.title('Receiver Operating Characteristic')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

Refer to *Figure 2.9* for the resulting ROC curve:

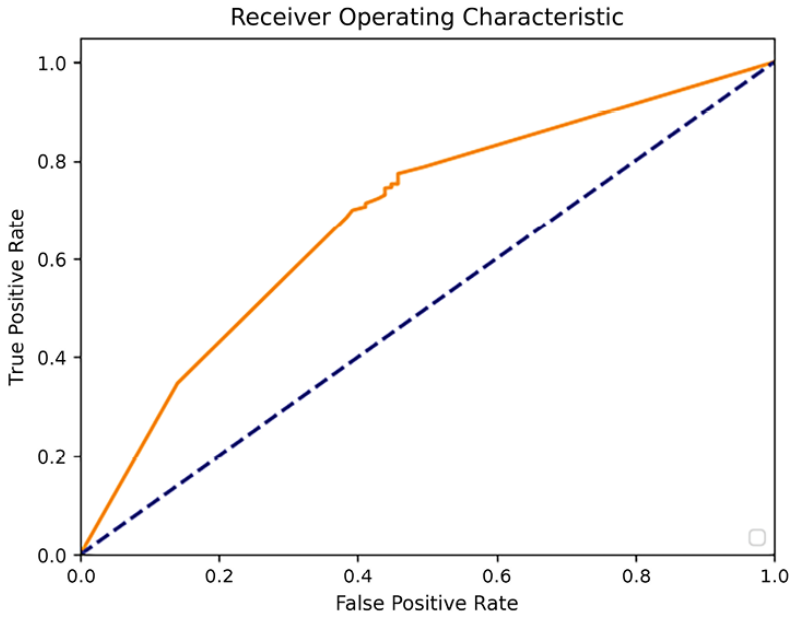

Figure 2.9: ROC curve

In the graph, the dashed line is the baseline representing random guessing, where the true positive rate increases linearly with the false positive rate; its AUC is 0.5. The solid line is the ROC plot of our model, and its AUC is somewhat less than 1. In a perfect case, the true positive samples have a probability of 1, so that the ROC starts at the point with 100% true positive and 0% false positive. The AUC of such a perfect curve is 1. To compute the exact AUC of our model, we can resort to the `roc_auc_score` function of `scikit-learn`:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(Y_test, pos_prob)
0.6857375752586637
```

What AUC value leads to the conclusion that a classifier is good? Unfortunately, there is no

such “magic” number. We use the following rule of thumb as general guidelines: classification models achieving an AUC of `0.7` to `0.8` are considered acceptable, `0.8` to `0.9` are great, and anything above `0.9` are superb. Again, in our case, we are only using the very sparse movie rating data. Hence, an AUC of `0.69` is actually acceptable.

You have learned several classification metrics, and we will explore how to measure them properly and how to fine-tune our models in the next section.

Tuning models with cross-validation

Limiting the evaluation to a single fixed set may be misleading since it's highly dependent on the specific data points chosen for that set. We can simply avoid adopting the classification results from one fixed testing set, which we did in experiments previously. Instead, we usually apply the **k-fold cross-validation** technique to assess how a model will generally perform in practice.

In the *k*-fold cross-validation setting, the original data is first randomly divided into *k* equal-sized subsets, in which class proportion is often preserved. Each of these *k* subsets is then successively retained as the testing set for evaluating the model. During each trial, the rest of the *k* -1 subsets (excluding the one-fold holdout) form the training set for driving the model. Finally, the average performance across all *k* trials is calculated to generate an overall result:

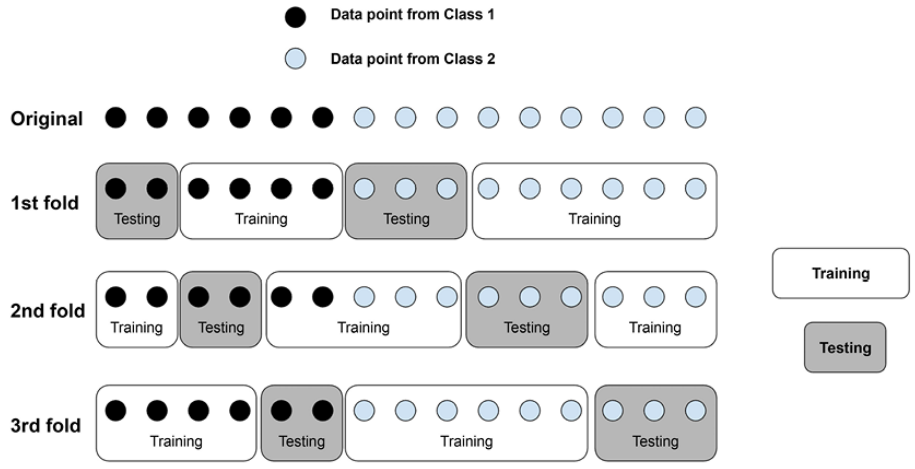

Figure 2.10: Diagram of 3-fold cross-validation

Statistically, the average performance of *k*-fold cross-validation is a better estimate of how a model per-

Statistically, the average performance of *k*-fold cross-validation is a better estimate of how a model performs in general. Given different sets of parameters pertaining to a machine learning model and/or data preprocessing algorithms, or even two or more different models, the goal of model tuning and/or model selection is to pick a set of parameters of a classifier so that the best average performance is achieved. With these concepts in mind, we can now start to tweak our Naïve Bayes classifier, incorporating cross-validation and the AUC of ROC measurements.

In *k*-fold cross-validation, *k* is usually set at 3, 5, or 10. If the training size is small, a large *k* (5 or 10) is recommended to ensure sufficient training samples in each fold. If the training size is large, a small value (such as 3 or 4) works fine since a higher *k* will lead to an even higher computational cost of training on a large dataset.

We will use the `split()` method from the `StratifiedKFold` class of `scikit-learn` to divide the data into chunks with preserved class distribution:

```
>>> from sklearn.model_selection import StratifiedKFold
>>> k = 5
>>> k_fold = StratifiedKFold(n_splits=k, random_state=42)
```

After initializing a 5-fold generator, we choose to explore the following values for the following parameters:

- `alpha` : This represents the smoothing factor, the initial value for each feature
- `fit_prior` : This represents whether to use prior tailored to the training data

We start with the following options:

```
>>> smoothing_factor_option = [1, 2, 3, 4, 5, 6]
>>> fit_prior_option = [True, False]
>>> auc_record = {}
```

Then, for each fold generated by the `split()` method of the `k_fold` object, we repeat the process of classifier initialization, training, and prediction with one of the aforementioned combinations of parameters, and record the resulting AUCs:

```
>>> for train_indices, test_indices in k_fold.split(X, Y):
```

```
>>> for train_indices, test_indices in k_fold.split(X, Y):
...     X_train_k, X_test _k= X[train_indices], X[test_indices]
...     Y_train_k, Y_test_k = Y[train_indices], Y[test_indices]
...     for alpha in smoothing_factor_option:
...         if alpha not in auc_record:
...             auc_record[alpha] = {}
...         for fit_prior in fit_prior_option:
...             clf = MultinomialNB(alpha=alpha,
...                                 fit_prior=fit_prior)
...             clf.fit(X_train_k, Y_train_k)
...             prediction_prob = clf.predict_proba(X_test_k)
...             pos_prob = prediction_prob[:, 1]
...             auc = roc_auc_score(Y_test_k, pos_prob)
...             auc_record[alpha][fit_prior] = auc +
...                     auc_record[alpha].get(fit_prior, 0.0)
```

Finally, we present the results as follows:

```
>>> for smoothing, smoothing_record in auc_record.items():
...     for fit_prior, auc in smoothing_record.items():
...         print(f'   {smoothing}       {fit_prior}
...             {auc/k:.5f}')
smoothing  fit prior  auc
   1       True    0.65647
   1       False    0.65708
   2       True    0.65795
   2       False    0.65823
   3       True    0.65740
   3       False    0.65801
   4       True    0.65808
   4       False    0.65795
   5       True    0.65814
   5       False    0.65694
   6       True    0.65663
   6       False    0.65719
```

The (`2` , `False`) set enables the best averaged AUC, at `0.65823` .

Finally, we retrain the model with the best set of hyperparameters (`2` , `False`) and compute the AUC:

Finally, we retrain the model with the best set of hyperparameters (`2`, `False`) and compute the AUC:

```
>>> clf = MultinomialNB(alpha=2.0, fit_prior=False)
>>> clf.fit(X_train, Y_train)
>>> pos_prob = clf.predict_proba(X_test)[:, 1]
>>> print('AUC with the best model:', roc_auc_score(Y_test,
...     pos_prob))
AUC with the best model:  0.6862056720417091
```

An AUC of `0.686` is achieved with the fine-tuned model. In general, tweaking model hyperparameters using cross-validation is one of the most effective ways to boost learning performance and reduce overfitting.

Summary

In this chapter, you learned about the fundamental concepts of machine learning classification, including types of classification, classification performance evaluation, cross-validation, and model tuning. You also learned about the simple, yet powerful, classifier, Naïve Bayes. We went in depth through the mechanics and implementations of Naïve Bayes with a couple of examples, the most important one being the movie recommendation project.

Binary classification using Naïve Bayes was the main talking point of this chapter. In the next chapter, we will solve ad click-through prediction using another binary classification algorithm: a **decision tree**.

Exercises

1. As mentioned earlier, we extracted user-movie relationships only from the movie rating data where most ratings are unknown. Can you also utilize data from the `movies.dat` and `users.dat` files?
2. Practice makes perfect—another great project to deepen your understanding could be heart disease classification. The dataset can be downloaded directly from **https://archive.ics.uci.edu/ml/datasets/Heart+Disease**.
3. Don't forget to fine-tune the model you obtained from Exercise 2 using the techniques you learned in this chapter. What is the best AUC it achieves?

References

To acknowledge the use of the MovieLens dataset in this chapter, I would like to cite the following paper:

F. Maxwell Harper and Joseph A. Konstan. 2015. *The MovieLens Datasets: History and Context*. ACM **Transactions on Interactive Intelligent Systems** (**TiiS**) 5, 4, Article 19 (December 2015), 19 pages. DOI: http://dx.doi.org.proxy.bnl.lu/10.1145/2827872.

Join our book's Discord space

Join our community's Discord space for discussions with the authors and other readers:

https://packt.link/yuxi

Previous chapter
‹ Getting Started with Machi...

Next chapter
Predicting Online Ad Click-... ›