```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import gluon, np, npx
npx.set_np()
d2l.use_svg_display()
```

18.9.1 Optical Character Recognition

MNIST (LeCun et al., 1998) is one of widely used datasets. It contains 60,000 images for training and 10,000 images for validation. Each image contains a handwritten digit from 0 to 9. The task is classifying each image into the corresponding digit.

Gluon provides a `MNIST` class in the `data.vision` module to automatically retrieve the dataset from the Internet. Subsequently, Gluon will use the already-downloaded local copy. We specify whether we are requesting the training set or the test set by setting the value of the parameter `train` to `True` or `False`, respectively. Each image is a grayscale image with both width and height of 28 with shape $(28,28,1)$. We use a customized transformation to remove the last channel dimension. In addition, the dataset represents each pixel by an unsigned 8-bit integer. We quantize them into binary features to simplify the problem.

```
def transform(data, label):
    return np.floor(data.astype('float32') / 128).squeeze(axis=-1), label

mnist_train = gluon.data.vision.MNIST(train=True, transform=transform)
mnist_test = gluon.data.vision.MNIST(train=False, transform=transform)
```

We can access a particular example, which contains the image and the corresponding label.

```
image, label = mnist_train[2]
image.shape, label
```

```
((28, 28), array(4, dtype=int32))
```

Our example, stored here in the variable `image`, corresponds to an image with a height and width of 28 pixels.

```
image.shape, image.dtype
```

```
((28, 28), dtype('float32'))
```

Our code stores the label of each image as a scalar. Its type is a 32-bit integer.

```
label, type(label), label.dtype
```

```
(array(4, dtype=int32), mxnet.numpy.ndarray, dtype('int32'))
```

We can also access multiple examples at the same time.

```
images, labels = mnist_train[10:38]
images.shape, labels.shape
```

```
((28, 28, 28), (28,))
```

Let us visualize these examples.

```
d2l.show_images(images, 2, 9);
```

18.9.2 The Probabilistic Model for Classification

In a classification task, we map an example into a category. Here an example is a grayscale 28×28 image, and a category is a digit. (Refer to Section 3.4 for a more detailed explanation.) One natural way to express the classification task is via the probabilistic question: what is the most likely label given the features (i.e., image pixels)? Denote by $\mathbf{x} \in \mathbb{R}^d$ the features of the example and $y \in \mathbb{R}$ the label. Here features are image pixels, where we can reshape a 2-dimensional image to a vector so that $d = 28^2 = 784$, and labels are digits. The probability of the label given the features is $p(y \mid \mathbf{x})$. If we are able to compute these probabilities, which are $p(y \mid \mathbf{x})$ for $y = 0, \ldots, 9$ in our example, then the classifier will output the prediction $\hat{y}$ given by the expression:

$$\hat{y} = \mathrm{argmax}\, p(y \mid \mathbf{x}). \tag{18.9.1}$$

Unfortunately, this requires that we estimate $p(y \mid \mathbf{x})$ for every value of $\mathbf{x} = x_1, ..., x_d$. Imagine that each feature could take one of 2 values. For example, the feature $x_1 = 1$ might signify that the word apple appears in a given document and $x_1 = 0$ would signify that it does not. If we had 30 such binary features, that would mean that we need to be prepared to classify any of 2^{30} (over 1 billion!) possible values of the input vector $\mathbf{x}$.

Moreover, where is the learning? If we need to see every single possible example in order to predict the corresponding label then we are not really learning a pattern but just memorizing the dataset.

18.9.3 The Naive Bayes Classifier

Fortunately, by making some assumptions about conditional independence, we can introduce some inductive bias and build a model capable of generalizing from a comparatively modest selection of training examples. To begin, let us use Bayes theorem, to express the classifier as

$$\hat{y} = \mathrm{argmax}_y\, p(y \mid \mathbf{x}) = \mathrm{argmax}_y\, \frac{p(\mathbf{x} \mid y)p(y)}{p(\mathbf{x})}. \tag{18.9.2}$$

Note that the denominator is the normalizing term $p(\mathbf{x})$ which does not depend on the value of the label y. As a result, we only need to worry about comparing the numerator across different values of y. Even if calculating the denominator turned out to be intractable, we could get away with ignoring it, so long as we could evaluate the numerator. Fortunately, even if we wanted to recover the normalizing constant, we could. We can always recover the normalization term since $\sum_y p(y \mid \mathbf{x}) = 1$.

Now, let us focus on $p(\mathbf{x} \mid y)$. Using the chain rule of probability, we can express the term $p(\mathbf{x} \mid y)$ as

$$p(x_1 \mid y) \cdot p(x_2 \mid x_1, y) \cdot ... \cdot p(x_d \mid x_1, ..., x_{d-1}, y). \tag{18.9.3}$$

By itself, this expression does not get us any further. We still must estimate roughly 2^d parameters. However, if we assume that *the features are conditionally independent of each other, given the label*, then suddenly we are in much better shape, as this term simplifies to $\prod_i p(x_i \mid y)$, giving us the predictor

$$\hat{y} = \mathrm{argmax}_y \prod_{i=1}^{d} p(x_i \mid y) p(y). \tag{18.9.4}$$

If we can estimate $\prod_i p(x_i = 1 \mid y)$ for every i and y, and save its value in $P_{xy}[i, y]$, here P_{xy} is a $d \times n$ matrix with n being the number of classes and $y \in \{1, \ldots, n\}$. In addition, we estimate $p(y)$ for every y and save it in $P_y[y]$, with P_y a n-length vector. Then for any new example $\mathbf{x}$, we could compute

$$\hat{y} = \mathrm{argmax}_y \prod_{i=1}^{d} P_{xy}[x_i, y] P_y[y], \tag{18.9.5}$$

for any y. So our assumption of conditional independence has taken the complexity of our model from an exponential dependence on the number of features $\mathcal{O}(2^d n)$ to a linear dependence, which is $\mathcal{O}(dn)$.

18.9.4 Training

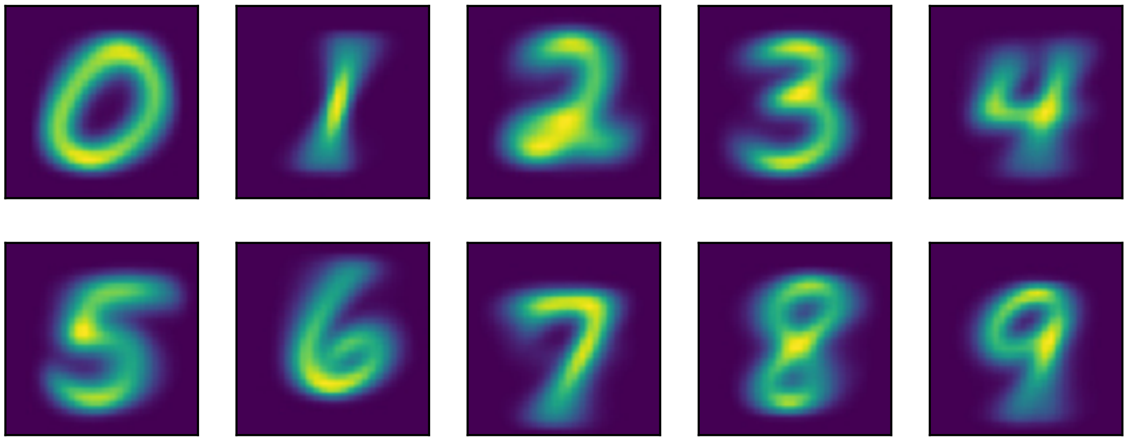
The problem now is that we do not know P_{xy} and P_y. So we need to estimate their values given some training data first. This is *training* the model. Estimating P_y is not too hard. Since we are only dealing with 10 classes, we may count the number of occurrences n_y for each of the digits and divide it by the total amount of data n. For instance, if digit 8 occurs $n_8 = 5,800$ times and we have a total of $n = 60,000$ images, the probability estimate is $p(y = 8) = 0.0967$.

```
X, Y = mnist_train[:]  # All training examples

n_y = np.zeros((10))
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

```
array([0.09871667, 0.11236667, 0.0993    , 0.10218333, 0.09736667,
       0.09035   , 0.09863333, 0.10441667, 0.09751666, 0.09915   ])
```

Now on to slightly more difficult things P_{xy}. Since we picked black and white images, $p(x_i \mid y)$ denotes the probability that pixel i is switched on for class y. Just like before we can go and count the number of times n_{iy} such that an event occurs and divide it by the total number of occurrences of y, i.e., n_y. But there is something slightly troubling: certain pixels may never be black (e.g., for well cropped images the corner pixels might always be white). A convenient way for statisticians to deal with this problem is to add pseudo counts to all occurrences. Hence, rather than n_{iy} we use $n_{iy} + 1$ and instead of n_y we use $n_y + 1$. This is also called *Laplace Smoothing*. It may seem ad-hoc, however it may be well motivated from a Bayesian point-of-view.

```
n_x = np.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = np.array(X.asnumpy()[Y.asnumpy() == y].sum(axis=0))
P_xy = (n_x + 1) / (n_y + 1).reshape(10, 1, 1)

d2l.show_images(P_xy, 2, 5);
```



By visualizing these $10 \times 28 \times 28$ probabilities (for each pixel for each class) we could get some mean looking digits.

Now we can use (18.9.5) to predict a new image. Given $\mathbf{x}$, the following functions computes $p(\mathbf{x} \mid y)p(y)$ for every y.

```
def bayes_pred(x):
    x = np.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(axis=1)  # p(x|y)
    return np.array(p_xy) * P_y

image, label = mnist_test[0]
bayes_pred(image)
```

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

This went horribly wrong! To find out why, let us look at the per pixel probabilities. They are typically numbers between 0.001 and 1. We are multiplying 784 of them. At this point it is worth mentioning that we are calculating these numbers on a computer, hence with a fixed range for the exponent. What happens is that we experience *numerical underflow*, i.e., multiplying all the small numbers leads to something even smaller until it is rounded down to zero. We discussed this as a theoretical issue in Section 18.7, but we see the phenomena clearly here in practice.

As discussed in that section, we fix this by use the fact that $\log ab = \log a + \log b$, i.e., we switch to summing logarithms. Even if both a and b are small numbers, the logarithm values should be in a proper range.

```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))
```

```
underflow: 0.0
logarithm is normal: -1805.2267129073316
```

Since the logarithm is an increasing function, we can rewrite (18.9.5) as

$$\hat{y} = \mathrm{argmax}_y \sum_{i=1}^{d} \log P_{xy}[x_i, y] + \log P_y[y]. \tag{18.9.6}$$

We can implement the following stable version:

```
log_P_xy = np.log(P_xy)
log_P_xy_neg = np.log(1 - P_xy)
log_P_y = np.log(P_y)

def bayes_pred_stable(x):
    x = np.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = p_xy.reshape(10, -1).sum(axis=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
array([-269.00424, -301.73447, -245.21458, -218.8941 , -193.46907,
       -206.10315, -292.54315, -114.62834, -220.35619, -163.18881])
```

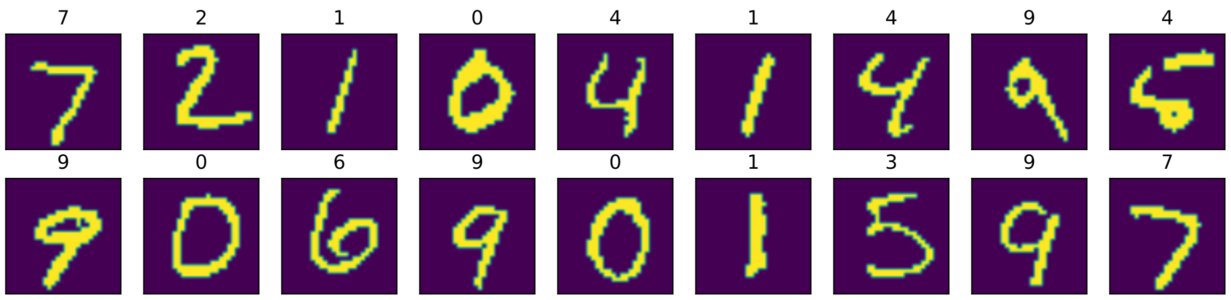
We may now check if the prediction is correct.

```
# Convert label which is a scalar tensor of int32 dtype
# to a Python scalar integer for comparison
py.argmax(axis=0) == int(label)
```

```
array(True)
```

If we now predict a few validation examples, we can see the Bayes classifier works pretty well.

```
def predict(X):
    return [bayes_pred_stable(x).argmax(axis=0).astype(np.int32) for x in X]

X, y = mnist_test[:18]
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```



Finally, let us compute the overall accuracy of the classifier.

```
X, y = mnist_test[:]
preds = np.array(predict(X), dtype=np.int32)
float((preds == y).sum()) / len(y)  # Validation accuracy
```

```
0.8426
```

Modern deep networks achieve error rates of less than 0.01. The relatively poor performance is due to the incorrect statistical assumptions that we made in our model: we assumed that each and every pixel are *independently* generated, depending only on the label. This is clearly not how humans write digits, and this wrong assumption led to the downfall of our overly naive (Bayes) classifier.

Summary

- Using Bayes' rule, a classifier can be made by assuming all observed features are independent.
- This classifier can be trained on a dataset by counting the number of occurrences of combinations of labels and pixel values.
- This classifier was the gold standard for decades for tasks such as spam detection.

Exercises

1. Consider the dataset $[[0, 0], [0, 1], [1, 0], [1, 1]]$ with labels given by the XOR of the two elements $[0, 1, 1, 0]$. What are the probabilities for a Naive Bayes classifier built on this dataset. Does it successfully classify our points? If not, what assumptions are violated?
2. Suppose that we did not use Laplace smoothing when estimating probabilities and a data point arrived at testing time which contained a value never observed in training. What would the model output?
3. The naive Bayes classifier is a specific example of a Bayesian network, where the dependence of random variables are encoded with a graph structure. While the full theory is beyond the scope of this section (see (Koller & Friedman, 2009) for full details), explain why allowing explicit dependence between the two input variables in the XOR model allows for the creation of a successful classifier.