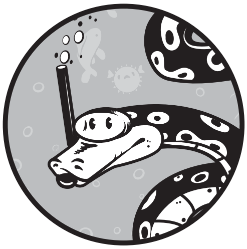
4

WORKING WITH DATA



Developing a proper dataset is the single most important part of building a successful machine learning model. Machine learning models live and die by the phrase "garbage in, garbage out." As you saw in Chapter 1, the model uses the training data to configure itself to the problem. If the training data is not a good representation of the data the model will receive when it is used, we can't expect our model to perform well. In this chapter, we'll learn how to create a good dataset that represents the data the model will encounter in the wild.

Classes and Labels

In this book, we're exploring *classification*: we're building models that put things into discrete categories, or *classes*, like dog breed, flower type, digit, and so on. To represent classes, we give each input in our training set an identifier called a *label*. A label could be the string "Border Collie" or, better still, a number like 0 or 1.

Models don't know what their inputs represent. They don't care whether the input is a picture of a border collie or the value of Google stock. To the model, it's all numbers. The same is true of labels. Because the label for the input has no intrinsic meaning to the model, we can represent classes however we choose. In practice, class labels are usually integers starting with 0. So, if there are 10 classes, the class labels are 0, 1, 2, . . ., 9. In Chapter 5, we'll work with a dataset that has 10 classes representing images of different real-world things. We'll simply map them to the integers as in Table 4-1.

Table 4-1: Label Classes with Integers: 0, 1, 2, . . .

Label	Actual class
0	airplanes
1	cars
2	birds
3	cats
4	deer
5	dogs
6	frogs
7	horses
8	ships
9	trucks

With that labeling, every training input that is a dog is labeled 5, while every input that is a truck is labeled 9. But what exactly is it that we're labeling? In the next section, we'll cover features and feature vectors, the very lifeblood of machine learning.

Features and Feature Vectors

Machine learning models take *features* as inputs and deliver, in the case of a classifier, a label as output. So what are these features and where do they come from?

For most models, features are numbers. What the numbers represent depends upon the task at hand. If we're interested in identifying flowers based on measurements of their physical properties, our features are those measurements. If we're interested in using the dimensions of cells in a medical sample to predict whether a tumor is breast cancer or not, the features are those dimensions. With modern techniques, the features might be the pixels of an image (numbers), or a sound's frequency (numbers) or even how many foxes were counted by a camera trap over a two-week period (numbers).

Features, then, are whatever numbers we want to use as inputs. The goal of training the model is to get it to learn a relationship between the input features and the output label. We assume that a relationship exists between the input features and output label before training the model. If the model fails to train, it might be that there is no relationship to learn.

After training, feature vectors with unknown class labels are given to the model, and the model's output predicts the class label based on the relationships it discovered during training. If the model is repeatedly making poor predictions, one possibility is that the selected features are not sufficiently capturing that relationship. Before we go into what makes a good feature, let's take a closer look at the features themselves.

Types of Features

To recap, features are numbers representing something that is measured or known, and *feature vectors* are sets of these numbers used as inputs to the model. There are different kinds of numbers you could use as features, and as you'll see, they're not all created equal. Sometimes you'll have to manipulate them before you can input them into your model.

Floating-Point Numbers

In Chapter 5, we'll be building a historic flower dataset. The features of that dataset are actual measurements of things like a flower's sepal width and height (in centimeters). A typical measurement might be 2.33 cm. This is a *floating-point* number—a number with a decimal point, or, if you remember your high school math courses, a *real* number. Most models want to work with floating-point numbers, so you can just use the measurements as they are. Floating-point numbers are *continuous*, meaning there are an infinite number of values between one integer and the next, so we have a smooth transition between them. As we'll see later on, some models expect continuous values.

Interval Values

Floating-point numbers don't work for everything, however. Clearly, flowers cannot have 10.14 petals, though they might have 9, 10, or 11. These numbers are *integers*: whole numbers without a fractional part or a decimal point. Unlike floating-point numbers, they are *discrete*, which means they pick out only certain values, leaving gaps in between. Fortunately for us, integers are just special real numbers, so models can use them as they are.

In our petal example, the difference between 9, 10, and 11 is meaningful in that 11 is bigger than 10, and 10 is bigger than 9. Not only that, but 11 is bigger than 10 in exactly the same way that 10 is bigger than 9. The difference, or interval, between the values is the same: 1. This value is called an *interval* value.

The pixels in an image are interval values, because they represent the (assumed linear) response of some measurement device, like a camera or an MRI machine, to some physical process like intensity and color of visible light or the number of hydrogen protons in free water in tissue. The key point is that if value *x* is the next number in the sequence after value *y*, and value *z* is the number before value *y*, then the difference between *x* and *y* is the same difference as between *y* and *z*.

Ordinal Values

Sometimes the interval between the values is not the same. For example, some models include someone's educational level to predict whether or not they will default on a loan. If we encode someone's educational level by counting their years of schooling, we could use that safely since the difference between 10 years of schooling and 8 is the same as the difference between 8 years of schooling and 6. However, if we simply assign 1 for "completed high school," 2 for "has an undergraduate degree," and 3 for "has a doctorate or other professional degree," we'd probably be in trouble; while 3 > 2 > 1 is true, whether or not meaningful for our model, the difference between the values represented by 3 and 2 and 2 and 1 is not the same. Features like these are called *ordinal* because they express an ordering, but the differences between the values are not necessarily always the same.

Categorical Values

Sometimes we use numbers as codes. We might encode sex as 0 for male and 1 for female, for example. In this case, 1 is not understood to be greater than 0 or less than 0, so these are not interval or ordinal values. Instead, these are *categorical* values. They express a category but say nothing about any relationship between the categories.

Another common example, perhaps relevant to classifying flowers, is color. We might use 0 for red, 1 for green, and 2 for blue. Again, no relationship exists between 0, 1, or 2 in this case. This doesn't mean we can't use categorical features with our models, but it does mean that we usually can't use them as they are since most types of machine learning models expect at least ordinal, if not interval numbers.

We can make categorical values at least ordinal by using the following trick. If we wanted to use a person's sex as an input, instead of saying 0 for male and 1 for female, we would create a two-element vector, one element for each possibility. The first digit in the vector will indicate whether the input is male by signaling either 0 (meaning they're not male) or 1 (meaning they are). The second digit will indicate whether or not they are female. We map the categorical values to a binary vector, as shown in Table 4-2.

Table 4-2: Representing Categories as Vectors

Categorical value		Vector representation
0	→	1 0
1	→	0 1

Here a 0 in the "is male" feature is meaningfully less than a 1 in that feature, which fits the definition of an ordinal value. The price we pay is to expand the number of features in our feature vector, as we need one feature for each of the possible categorical values. With five colors, for example, we'd need a five-element vector; with five thousand, a five-thousand-element vector.

To use this scheme, the categories must be mutually exclusive, meaning there will be only one 1 in each row. Because there's always only one nonzero value per row, this approach is sometimes called a *one-hot encoding*.

Feature Selection and the Curse of Dimensionality

This section is about *feature selection*, the process of selecting which features to use in your feature vectors, and why you shouldn't include features you don't need. Here's a good rule of thumb: the feature vector should contain only features that capture aspects of the data that allow the model to generalize to new data.

In other words, features should capture aspects of the data that help the model separate the classes. It's impossible to be more explicit, since the set of best features are always dataset specific, unknowable in advance. But that doesn't mean we can't say things that might be helpful in guiding us toward a useful set of features for whatever dataset we're working with.

Like many things in machine learning, selecting features comes with trade-offs. We need enough features to capture all the relevant parts of the data so that the model has something to learn from, but if we have too many features, we fall victim to the *curse of dimensionality*.

To explain what this means, let's look at an example. Suppose our features are all restricted to the range [0, 1). That's not a typo; we're using interval notation, where a square bracket means the bound is included in the range, and a parenthesis means the bound is excluded. So here 0 is allowed but 1 isn't. We'll also assume our feature vectors are either two-dimensional or three-dimensional. That way, we can plot each feature vector as a point in a 2D or 3D space. Finally, we'll simulate datasets by selecting feature vectors, 2D or 3D, uniformly at random so that each element of the vector is in [0, 1).

Let's fix the number of samples at 100. If we have two features, or a 2D space, we can represent 100 randomly selected 2D vectors as the left side of Figure 4-1. Now, if we have three features, or a 3D space, those same 100 features look like the right side of Figure 4-1.

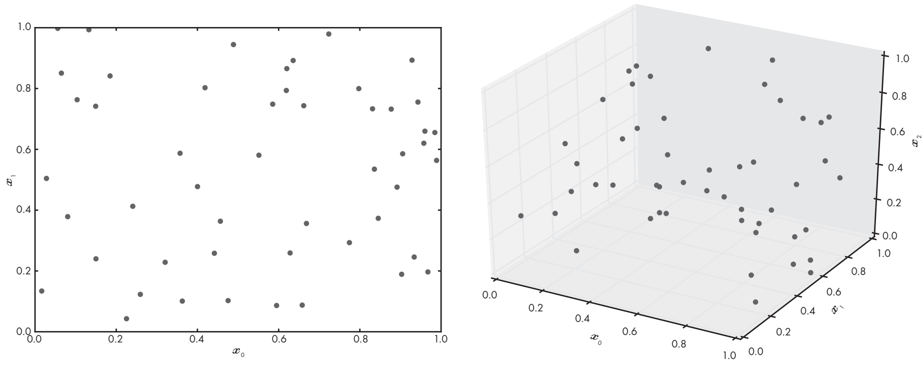


Figure 4-1: One hundred random samples in 2D space (left) and in 3D space (right)

Since we're assuming our feature vectors can come from anywhere in the 2D or 3D space, we want our dataset to sample as much of that space as possible so that it represents the space well. We can get a measure of how well the 100 points are filling the space by splitting each axis into 10 equal sections. Let's call these sections *bins*. We'll end up with 100 bins in the 2D space, because it has two axes (10×10), and 1,000 in the 3D space, because it has three axes ($10 \times 10 \times 10$). Now, if we count the number of bins occupied by at least one point and divide that number by the total number of bins, we'll get the fraction of bins that are occupied.

Doing this gives us 0.410 (out of a maximum of 1.0) for the 2D space and 0.048 for the 3D space. This means that 100 samples were able to sample about half of the 2D feature space. Not bad! But 100 samples in the 3D feature space sampled only about 5 percent of the space. To fill the 3D space to the same fraction as the 2D space, we'd need about 1,000—or 10 times as many as we have. This general rule applies as the dimensionality increases: a 4D feature space would need about 10,000 samples, while a 10D feature space would need about 10,000,000,000! As the number of features increases, the amount of training data we need to get a representative sampling of the possible feature space increases dramatically, approximately as 10^d, where d is the number of dimensions. This is the *curse of dimensionality*, and it was the bane of machine learning for decades. Fortunately for us, modern deep learning has overcome this curse, but it's still relevant when working with traditional models like the ones we will explore in Chapter 6.

For example, a typical color image on your computer might have 1024 pixels on a side where each pixel requires 3 bytes to specify the color as a mix of red, green, and blue. If we wanted to use this image as input to a model, we'd need a feature vector with $d = 1024 \times 1024 \times 3 = 3,145,728$ elements. This means we'd need some $10^{3,145,728}$ samples to populate our feature space. Clearly, this is not possible. We'll see in Chapter 12 how to overcome this curse by using a convolutional neural network.

Now that we know about classes, features, and feature vectors, let's describe what it means to have a good dataset.

Features of a Good Dataset

The dataset is everything. This is no exaggeration, since we build the model from the dataset. The model has parameters—be they the weights and biases of a neural network, the probabilities of each feature occurring in a Naïve Bayes model, or the training data itself in the case of Nearest Neighbors. The parameters are what we use the training data to find out: they encode the knowledge of the model and are learned by the training algorithm.

Let's back up a little bit and define the term *dataset* as we we'll use it in this book. Intuitively, we understand what a dataset is, but let's be more scientific and define it as a collection of pairs of values, $\{X, Y\}$, where X is an *input* to the model and Y is a label. Here X is some set of values that we've measured and grouped together, like length and width of flower parts, and Y is the thing we want to teach the model to tell us, such as which flower or which animal the data best represents.

For *supervised* learning, we act as the teacher, and the model acts as the student. We are teaching the student by presenting example after example, saying things like "this is a cat" and "this is a dog," much as we would teach a small child with a picture book. In this case, the *dataset* is a collection of examples, and *training* consists of showing the examples to the model repeatedly, until the model "gets it"—that is, until the parameters of the model are conditioned and adjusted to minimize the error made by the model for this particular dataset. This is the learning part of machine learning.

Interpolation and Extrapolation

Interpolation is the process of estimating within a certain known range. *Extrapolation* occurs when we use the data we have to estimate outside the known range. Generally speaking, our models are more accurate when they in some sense interpolate, which means we need a dataset that is a comprehensive representation of the range of values that could be used as inputs to the model.

As an example, let's look at world population, in billions, from 1910 to 1960 (Table 4-3). We have data for every 10 years in our known range, 1910 to 1960.

Table 4-3: The World Population by Decade

Year	Population (billions)
1910	1.750
1920	1.860
1930	2.070
1940	2.300
1950	2.557
1960	3.042

If we find the "best fitting" line to plot through this data, we can use it as a model to predict values. This is called *linear regression*, and it allows us to estimate the population for any year we choose. We'll skip the actual fitting process, which you can do simply with online tools, and jump right to the model:

$$p = 0.02509y - 46.28$$

For any year, y, we can get an estimate of the population, p. What was the world population in 1952? We don't have actual data for 1952 in our table, but using the model, we can estimate it like so:

$$p = 0.02509(1952) - 46.28 = 2.696 \text{ billion}$$

By checking the actual world population data for 1952, we know that it was 2.637 billion, so our estimate of 2.696 billion was only some 60 million off. The model seems to be pretty good!

In using the model to estimate the world population in 1952, we performed interpolation. We made an estimate for a value that was between data points we had, and the model gave us a good result. Extrapolation, on the other hand, is measuring beyond what is known, outside the range of our data.

Let's use our model to estimate world population in 2000, 40 years after the data we used to build our model ends:

$$p = 0.02509(2000) - 46.28 = 3.900 \text{ billion}$$

According to the model, it should be close to 3.9 billion, but we know from actual data that the world population in 2000 was 6.089 billion. Our model is off by over 2 billion people. What happened here is that we applied the model to input it wasn't suited for. If we remain in the range of inputs that the model is "trained" to know about, namely, dates from 1910 through 1960, then the model performs well enough. Once we went beyond the model's training, however, it fell apart because it assumed knowledge we didn't possess.

When we interpolate, the model will see examples that are similar to the set of examples it saw during training. Perhaps unsurprisingly, it will do better on these examples than when we extrapolate and ask the model to go beyond its training.

When it comes to classification, it's essential we have comprehensive training data. Let's assume we're training a model to identify dog breeds. In our dataset, we have hundreds of images of classic black-and-white border collies like the one on the left in Figure 4-2. If we then give the model a new image of a classic border collie, we will, hopefully, get back a correct label: "Border Collie." This is akin to asking the model to interpolate: it's working with something is has already seen before because the "Border Collie" label in the training data included many examples of classic border collies.

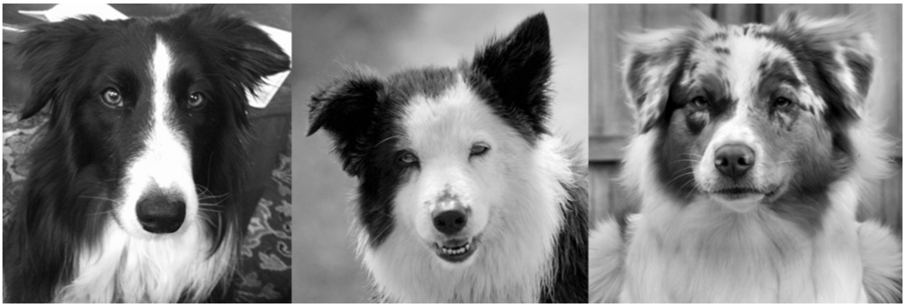


Figure 4-2: A border collie with classic markings (left), a border collie with liver-colored markings (middle), an Australian shepherd (right), (Left image, author's own; middle and right images, Creative Commons license.)

However, not every border collie has the classic border collie markings. Some are marked like the collie in the middle of Figure 4-2. Since we didn't include images like this in the training set, the model must now try to go beyond what it was trained to do and give a correct output label for an instance of a class it was trained on but of a type it was not trained with. It will likely fail, giving a false output like "Australian Shepherd," a breed similar to a border collie, as seen on the right of Figure 4-2.

The key concept to remember, however, is that the dataset must cover the full range of variation *within* the classes the model will see when the model is predicting labels for unknown inputs.

The Parent Distribution

The dataset must be representative of the classes it's modeling. Buried in this idea is the assumption that our data has a *parent distribution*, an unknown data generator that created the particular dataset we're using.

Consider this parallel from philosophy. The ancient Greek philosopher Plato uses the concept of ideals. In his view, there was an ideal chair somewhere "out there," and all existing chairs were more or less perfect copies of that ideal chair. This is what we mean by the relationship between the dataset we are using, the copy, and the parent distribution, the ideal generator. We want the dataset to be a representation of the ideal.

We can think of a dataset as a sample from some unknown process that produces data according to the parent distribution. The type of data it produces—the values and ranges of the features—will follow some unknown, statistical rule. For example, when you roll a die, each of the six values is equally likely in the long run. We call this a *uniform parent distribution*. If you make a bar graph of the number of times each value appears as you roll the die many times, you will get a (more or less) horizontal line since each value is equally likely to happen. We see a different distribution when we measure the height of adults. The distribution of heights will have a form with two humps, one around mean male height and another around mean female height.

The parent distribution is what generates this overall shape. The training data, the test data, and the data you give the model to make decisions must all come from the same parent distribution. This is a fundamental assumption models make, and one that shouldn't seem too surprising to us. Still, sometimes it's easy to mix things up and train with data from one parent distribution while testing or using the model with data from a different parent distribution. (How to train with one parent distribution and use that model with data from a different distribution is a very active research area at the moment. Search for "domain adaptation.")

Prior Class Probabilities

The *prior class probability* is the probability with which each class in the dataset appears in the wild.

In general, we want our dataset to match the prior probabilities of the classes. If class A appears 85 percent of the time and class B only 15 percent of the time, then we want class A to appear 85 percent of the time and class B to appear 15 percent of the time in our training set.

There are exceptions, however. Say one of the classes we want the model to learn is rare, showing up only once for every 10,000 inputs. If we make the dataset strictly follow the actual prior probabilities, the model might not see enough examples of the rare class to learn anything helpful about it. And, worse yet, what if the rare class is the class we are most interested in?

For example, let's pretend we're building a robot that locates four-leaf clovers. We'll assume that we already know that the input to the model is a clover; we just want to know whether it has three or four leaves. We know that an estimated 1 in every 5,000 clovers is a four-leaf clover. Building a dataset with 5,000 three-leaf clovers for every instance of a four-leaf clover seems reasonable until we realize that a model that simply says every input is a three-leaf clover will be right, on average, 4,999 times out of 5,000! It will be an extremely accurate but completely useless model because it never finds the class we're interested in.

Instead, we might use a 10:1 ratio of three-leaf to four-leaf clovers. Or, when training the model, we might start with an even number of three- and four-leaf clovers, and then, after training for a time, change to a mix that is increasingly closer to the actual prior probability. This trick doesn't work for all model types, but it does work for neural networks. Why this trick works is poorly understood but, intuitively, we can imagine the network learning first about the visual difference between a three-leaf and four-leaf clover and then learning something about the actual likelihood of encountering a four-leaf clover as the mix changes to be closer to the actual prior probabilities.

In reality, the trick is used because it often results in better-performing models. For much of machine learning, especially deep learning, empirical tricks and techniques are well in advance of any theory to back them up. "It just works better; that's why" is still a valid, though ultimately unsatisfying, answer to many questions about why a particular approach works well.

How to work with imbalanced data is something the research community is still actively investigating. Some choose to start with a more balanced ratio of classes; others use data augmentation (see Chapter 5) to boost the number of samples from the underrepresented class.

Confusers

We said that we need to include examples in our dataset that reflect all the natural variation in the classes we want to learn. This is definitely true, but at times it is particularly important to include training samples that are similar to one or more of our classes but really are not examples of that class.

Consider two models. The first learns the difference between images of dogs and images of cats. The second learns the difference between images of dogs and images that are not dogs. The first model has it easy. The input is either a dog or a cat, and the model is trained using images of dogs and images of cats. The second model, however, has it rougher. It's obvious that we need images of dogs for training. But, what should the "not dog" images be? Given the preceding discussion, we should be starting to intuit that we'll need images that cover the space of images the model will see in the wild.

We can take this one step further. If we want to tell the difference between dogs and not dogs, we should be sure to include wolves in the "not a dog" class when training. If we don't, the model might not learn enough to tell the difference when it encounters a wolf and will return a "dog" classification. If we build the dataset by using hundreds of "not dog" images that are all pictures of penguins and parrots, should we be surprised if the model decides to call a wolf a dog?

In general, we need to make sure the dataset includes *confusers*, or *hard negatives*—examples that are similar enough to other classes to be mistaken for them, but don't belong in the class. Confusers give the model a chance to learn the more precise features of a class. Hard negatives are particularly useful when distinguishing between something and everything else, as in "dog" versus "not dog."

Dataset Size

So far we've talked about what kind of data to include in a dataset, but how much of it do we need? "All of it" is a temptingly cheeky answer. For our model to be as precise as possible, we should use as many examples as possible. But it's rarely possible to get all of the data.

Choosing the size of your dataset means considering a trade-off between accuracy and the time and energy it takes to acquire the data. Acquiring data can be expensive or slow, or, as we saw with our clover example, sometimes the key class of the dataset is rare and seldom encountered. Because labeled data is generally expensive and slow to acquire, we should have some idea of how much we need before we get started.

Unfortunately, the truth is that there is no formula that answers the question of how much data is enough data. After a certain point, there are diminishing returns on the benefit of additional data. Moving from 100 examples to 1,000 examples might boost the accuracy of the model dramatically, but moving from 1,000 to 10,000 examples might offer only a small increase in accuracy. The increased accuracy needs to be balanced against the effort and expense of acquiring an additional 9,000 training examples.

Another factor to consider is the model itself. Models have a *capacity*, which determines the complexity they can support relative to the amount of training data available. The capacity of a model is directly related to its number of parameters. A larger model with more parameters will require a lot of training data to be able to find the proper parameter settings. And though it's often a good idea to have more training examples than model parameters, deep learning can work well when there is less training data than parameters. For example, if the classes are very different from each other—think buildings versus oranges—and it's easy for us to tell the difference, the model likely will also learn the difference quickly, so we can get away with fewer training examples. On the other hand, if we're trying to separate wolves from huskies, we might need a lot more data. We will discuss what to do when you don't have a lot of training data in Chapter 5, but none of those tricks are a good substitute for simply getting more data.

The only correct answer to the question of how much data is needed is "all of it." Get as much as is *practical*, given the constraints of the problem: expense, time, rarity, and so forth.

Data Preparation

Before we move on to building actual datasets, we're going to cover two situations you'll likely encounter before you can feed your dataset to a model: how to scale features, and what to do if a feature value is missing.

Scaling Features

A feature vector built from a set of different features might have a variety of ranges. One feature might take on a wide range of values, say, −1000 to 1000, while another might be restricted to a range of 0 to 1. Some models will not work well when this happens, as one feature dominates the others because of its range. Also, some model types are happiest when features have a mean value that is close to 0.

The solution to these issues is scaling. We'll assume for the time being that every feature in the feature vector is continuous. We'll work with a fake dataset consisting of five features and 15 samples. This means that our dataset has 15 samples—feature vectors and their labels—and each of the feature vectors has five elements. We'll assume there are three classes. The dataset looks like Table 4-4.

Table 4-4: A Hypothetical Dataset

Sample	x_0	x_1	x_2	x_3	x_4	Label
0	6998	0.1361	0.3408	0.00007350	78596048	0
1	6580	0.4908	3.0150	0.00004484	38462706	1
2	7563	0.9349	4.3465	0.00001003	6700340	2
3	8355	0.6529	2.1271	0.00002966	51430391	0
4	2393	0.4605	2.7561	0.00003395	27284192	0
5	9498	0.0244	2.7887	0.00008880	78543394	2
6	4030	0.6467	4.8231	0.00000403	19101443	2
7	5275	0.3560	0.0705	0.00000899	96029352	0
8	8094	0.7979	3.9897	0.00006691	7307156	1
9	843	0.7892	0.9804	0.00005798	10179751	1
10	1221	0.9564	2.3944	0.00007815	14241835	0
11	5879	0.0329	2.0085	0.00009564	34243070	2
12	923	0.4159	1.7821	0.00002467	52404615	1
13	5882	0.0002	1.5362	0.00005066	18728752	2
14	1796	0.7247	2.3190	0.00001332	96703562	1

As this is the first dataset covered in the book, let's go over it thoroughly to introduce some notation and see what is what. The first column in Table 4-4 is the sample number. The sample is an input, in this case a collection of five features representing a feature vector. Notice that the numbering starts at 0. As we'll be using Python arrays (NumPy arrays) for data, we'll start counting at 0 in all cases.

The next five columns are the features in each sample, labeled x_0 through x_4, again starting indices at 0. The final column is the class label. Since there are three classes, the labels run from 0 through 2. There are five samples from class 0, five from class 1, and five from class 2. Therefore, this is a small but balanced dataset; the prior probability of each class is 33 percent, which should, ideally, be close to the actual prior probability of the classes appearing in the wild.

If we had a model, then each row would be its own input. Writing $\{x_0, x_1, x_2, x_3, x_4\}$ to refer to these is tedious, so instead, when we are referring to a full feature vector, we'll use an uppercase letter. For example, we'd refer to Sample 2 as X_2 for dataset X. We'll also sometimes use matrices—2D arrays of numbers—that are also labeled with uppercase letters, for clarity. When we want to refer to a single feature, we'll use a lowercase letter with subscript, for example, x_3.

Let's look at the ranges of the features. The minimum, maximum, and range (the difference between the maximum and minimum) of each feature are shown in Table 4-5.

Table 4-5: The Minimum, Maximum, and Range of the Features in Table 4-4

Feature	Minimum	Maximum	Range
x_0	843.0	9498.0	8655.0
x_1	0.0002	0.9564	0.9562
x_2	0.0705	4.8231	4.7526
x_3	4.03e-06	9.564e-05	9.161e-05
x_4	6700340.0	96703562.0	90003222.0

Note the use of computer notation like 9.161e-05. This how computers represent scientific notation: $9.161 \times 10^{-5} = 0.00009161$. Notice, also, that each feature covers a very different range. Because of this, we'll want to scale the features so their ranges are more similar. Scaling is a valid thing to do prior to training a model as long as you scale all new inputs the same way.

Mean Centering

The simplest form of scaling is *mean centering*. This is easy to do: from each feature, simply subtract the mean (average) value of the feature over the entire dataset. The mean over a set of values, x_i $i = 0, 1, 2, \ldots$ is simply the sum of each value divided by the number of values:

$$\bar{x} = \frac{1}{N} \sum_{i=0}^{N} x_i$$

The mean value for feature x_0 is 5022, so to center x_0, we replace each value like so:

$$x_i \leftarrow x_i - 5022, \; i = 0, 1, 2, \ldots$$

where in this case the *i* index is across the samples, not the other elements of the feature vector.

Repeating the preceding steps for the mean value of all the other features will center the entire dataset. The result is that the mean value of each feature, over the dataset, is now 0, meaning the feature values themselves are all above and below 0. For deep learning, mean centering is often done by subtracting a mean image from each input image.

Changing the Standard Deviation to 1

Mean centering helps, but the distribution of values around 0 remains the same as before the mean was subtracted. All we did was shift the data down toward 0. The spread of values around the mean has a formal name: it's called the *standard deviation*, and it's computed as the average difference of the data values and the mean:

$$\sigma = \sqrt{\frac{\sum_{i=0}^{N}(x_i - \bar{x})^2}{N-1}}$$

The letter σ (sigma) is the usual name for the standard deviation in mathematics. You don't need to memorize this formula. It's there to show us how to calculate a measure of the spread, or range, of the data relative to the mean value of the data.

Mean centering changes $\bar{x}$ to 0, but it does not change σ. Sometimes we want to go further and, along with mean centering, change the spread of the data so that the ranges are the same, meaning the standard deviation for each feature is 1. Fortunately, doing this is straightforward. We replace each feature value, x, with

$$x \leftarrow \frac{x - \bar{x}}{\sigma}$$

where $\bar{x}$ and σ are the mean and standard deviation of each feature across the dataset. For example, the preceding toy dataset can be stored as a 2D NumPy array

```
x = [
 [6998, 0.1361, 0.3408, 0.00007350, 78596048],
 [6580, 0.4908, 3.0150, 0.00004484, 38462706],
 [7563, 0.9349, 4.3465, 0.00001003,  6700340],
 [8355, 0.6529, 2.1271, 0.00002966, 51430391],
 [2393, 0.4605, 2.7561, 0.00003395, 27284192],
 [9498, 0.0244, 2.7887, 0.00008880, 78543394],
 [4030, 0.6467, 4.8231, 0.00000403, 19101443],
 [5275, 0.3560, 0.0705, 0.00000899, 96029352],
 [8094, 0.7979, 3.9897, 0.00006691,  7307156],
 [ 843, 0.7892, 0.9804, 0.00005798, 10179751],
 [1221, 0.9564, 2.3944, 0.00007815, 14241835],
 [5879, 0.0329, 2.0085, 0.00009564, 34243070],
 [ 923, 0.4159, 1.7821, 0.00002467, 52404615],
 [5882, 0.0002, 1.5362, 0.00005066, 18728752],
 [1796, 0.7247, 2.3190, 0.00001332, 96703562],
]
```

so that the entire dataset can be processed in one line of code:

```
x = (x - x.mean(axis=0)) / x.std(axis=0)
```

This approach is called *standardization* or *normalizing*, and you should do it to most datasets, especially when using one of the traditional models discussed in Chapter 6. Whenever possible, standardize your dataset so that the features have 0 mean and a standard deviation of 1.

If we standardize the preceding dataset, what will it look like? Subtracting, per feature, the mean value of that feature and dividing by the standard deviation gives us a new dataset (Table 4-6). Here, we've shortened the numbers to four decimal digits for display and have dropped the label.

Table 4-6: The Data in Table 4-4 Standardized

Sample	x_0	x_1	x_2	x_3	x_4
0	0.6930	–1.1259	–1.5318	0.9525	1.1824
1	0.5464	–0.0120	0.5051	–0.0192	–0.1141
2	0.8912	1.3826	1.5193	–1.1996	–1.1403
3	1.1690	0.4970	–0.1712	–0.5340	0.3047
4	–0.9221	–0.1071	0.3079	–0.3885	–0.4753
5	1.5699	–1.4767	0.3327	1.4714	1.1807
6	–0.3479	0.4775	1.8823	–1.4031	–0.7396
7	0.0887	–0.4353	–1.7377	–1.2349	1.7456
8	1.0775	0.9524	1.2475	0.7291	–1.1207
9	–1.4657	0.9250	–1.0446	0.4262	–1.0279
10	–1.3332	1.4501	0.0323	1.1102	–0.8966
11	0.3005	–1.4500	–0.2615	1.7033	–0.2505
12	–1.4377	–0.2472	–0.4340	–0.7032	0.3362
13	0.3016	–1.5527	–0.6213	0.1780	–0.7517
14	–1.1315	0.7225	–0.0250	–1.0881	1.7674

If you compare the two tables, you'll see that after our manipulations, the features are more similar than they were in the original set. If we look at x_3, we'll see that the mean of the values is $-1.33e - 16 = -1.33 \times 10^{-16} = -0.000000000000000133$, which is virtually 0. Good! This is what we want. If you do the calculations, you'd see that the means of the other features are similarly close to 0. What about the standard deviation? For x_3 it's 0.99999999, which is virtually 1—again, this is what we'd like. We'll use this new, transformed, dataset to train the model.

Therefore, we must apply the per feature means and standard deviations, as measured on the training set, to any new inputs we're giving to the model:

$$x_{\text{new}} \leftarrow \frac{x_{\text{new}} - \bar{x}_{\text{train}}}{\sigma_{\text{train}}}$$

Here, x_{new} is the new feature vector we want to apply to the model, and $\bar{x}_{\text{train}}$ and σ_{train} are the mean and standard deviation, per feature, from the training set.

Missing Features

Sometimes we don't have all the features we need for a sample. We might have forgotten to make a measurement, for example. These are *missing features*, and we need to find a way to correct them, since most models don't have the ability to accept missing data.

One solution is to fill in the missing values with values that are outside of the feature's range, in the hopes that the model will learn to ignore those values or make more use of other features. Indeed, some more advanced deep learning models intentionally zero some of the input as a form of regularization (we'll see what that means in later chapters).

For now, we'll learn the second most obvious solution: replacing missing features with the mean value of features over the dataset. Let's look again at our practice dataset from earlier. This time, we'll have some missing data to deal with (Table 4-7).

Table 4-7: Our Sample Dataset (Table 4-4) with Some Holes

Sample	x_0	x_1	x_2	x_3	x_4	Label
0	6998	0.1361	0.3408	0.00007350	78596048	0
1		0.4908		0.00004484	38462706	1
2	7563	0.9349	4.3465		6700340	2
3	8355	0.6529	2.1271	0.00002966	51430391	0
4	2393	0.4605	2.7561	0.00003395	27284192	0
5	9498		2.7887	0.00008880	78543394	2
6	4030	0.6467	4.8231	0.00000403		2
7	5275	0.3560	0.0705	0.00000899	96029352	0
8	8094	0.7979	3.9897	0.00006691	7307156	1
9			0.9804		10179751	1
10	1221	0.9564	2.3944	0.00007815	14241835	0
11	5879	0.0329	2.0085	0.00009564	34243070	2
12	923			0.00002467		1
13	5882	0.0002	1.5362	0.00005066	18728752	2
14	1796	0.7247	2.3190	0.00001332	96703562	1

The blank spaces indicate missing values. The means of each feature, ignoring missing values, are shown in Table 4-8.

Table 4-8: The Means for Features in Table 4-7

x_0	x_1	x_2	x_3	x_4
5223.6	0.5158	2.345	4.71e-05	42957735.0

If we replace each missing value with the mean, we'll get a dataset we can standardize and use to train a model.

Of course, real data is better, but the mean is the simplest substitute we can reasonably use. If the dataset is large enough, we might instead generate a histogram of the values of each feature and select the mode—the most common value—but using the mean should work out just fine, especially if your dataset has a lot of samples and the number of missing features is fairly small.

Training, Validation, and Test Data

Now that we have a dataset—a collection of feature vectors—we're ready to start training a model, right? Well, actually, no. That's because we don't want to use the entire dataset for training. We'll need to use some of the data for other purposes, and so we need to split it into at least two subsets, although ideally we'd have three. We call these subsets the training data, validation data, and test data.

The Three Subsets

The *training data* is the subset we use to train the model. The important thing here is selecting feature vectors that well represent the parent distribution of the data.

The *test data* is the subset used to evaluate how well the trained model is doing. We *never* use the test data when training the model; that would be cheating, because we'd be testing the model on data it has seen before. Put the test dataset aside, resist the temptation to touch it until the model is complete, and then use it to evaluate the model.

The third dataset is the *validation data*. Not every model needs a validation dataset, but for deep learning models, having one is helpful. We use the validation dataset during training as though it's test data to get an idea of how well the training is working. It can help us decide things like when to stop training and whether we're using the proper model.

For example, a neural network has some number of layers, each with some number of nodes. We call this the *architecture* of the model. During training, we can test the performance of the neural network with the validation data to figure out whether we should continue training or stop and try a different architecture. We don't train the model with the validation set, and we don't use the validation set to modify model parameters. We also can't use validation data when reporting actual model performance, since we used results based on the validation data to select the model in the first place. Again, this would make it seem like the model is doing better than it is.

Figure 4-3 illustrates the three subsets and their relationships to one another. On the left is the whole dataset. This is the entire collection of feature vectors and associated labels. On the right are the three subsets. The training data and the validation data work together to train and develop the model, while the test data is held back until the model is ready for it. The size of the cylinders reflects the relative amount of data that should fall into each subset, though in practice the validation and test subsets might be even smaller.

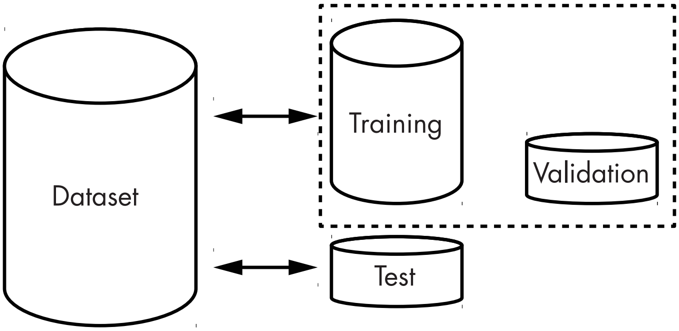


Figure 4-3: Relationships among training, validation, and test subsets

To recap: use the training and validation sets to build the model and the test set to evaluate it.

Partitioning the Dataset

How much data should go into each dataset?

A typical split is 90 percent for training, 5 percent for validation, and 5 percent for testing. For deep learning models, this is fairly standard. If you're working with a very large dataset, you could go as low as 1 percent each for validation and testing. For classic models, which might not learn as well, we might want to make the test dataset larger to ensure we are able to generalize to a wide variety of possible inputs. In those cases, you might try something like 80 percent for training and 10 percent each for validation and test. If you're not using validation data, the full 20 percent might go to testing. These larger test sets might be appropriate for multiclass models that have classes with low prior probabilities. Or, since the test set is not used to define the model, you might increase the number of rare classes in the test set. This might be of particular value should missing the rare class be a costly event (think missing a tumor in a medical image).

Now that we've determined how much data to put into each set, let's use `sklearn` to generate a dummy dataset that we can partition:

```
>>> import numpy as np
>>> from sklearn.datasets import make_classification
>>> x,y = make_classification(n_samples=10000, weights=(0.9,0.1))
>>> x.shape
    (10000, 20)
```

```
>>> len(np.where(y == 0)[0])
    8969
>>> len(np.where(y == 1)[0])
    1031
```

Here, we've used two classes and 20 features to generate 10,000 samples. The dataset is imbalanced, with 90 percent of the samples in class 0 and 10 percent in class 1. The output is a 2D array of samples (`x`) and associated 0 or 1 labels (`y`). The dataset is generated from multidimensional Gaussians that are the analogs of the normal bell curve in more than one dimension, but that doesn't matter to us right now. The useful part for us is that we have a collection of feature vectors and labels, so that we can look at ways in which the dataset might be split into subsets.

The key to the preceding code is the call to `make_classification`, which accepts the number of samples requested and the fraction for each class. The `np.where` calls simply find all the class 0 and class 1 instances so that `len` can count them.

Earlier, we talked about the importance of preserving—or at least approaching—the actual prior probabilities of the different classes in our dataset. If one class makes up 10 percent of real world cases, it would ideally make up 10 percent of our dataset. Now we need to find a way to preserve this prior class probability in the subsets we make for training, validation, and test. There are two main ways to do this: partitioning by class and random sampling.

Partitioning by Class

The exact approach, which is suitable when the dataset is small or perhaps when one class is rare, is to determine the number of samples representing each class, and then set aside selected percentages of each, by class, before merging them together. So, if there are 9,000 samples from class 0, and 1,000 samples from class 1, and we want to put 90 percent of the data into training and 5 percent each into validation and test, we would select 8,100 samples, *at random*, from the class 0 collection and 900 samples, *at random*, from the class 1 collection to make up the training set. Similarly, we would randomly select 450 of the remaining 900 unused class 0 samples for the validation set along with 50 of the remaining unused class 1 data. The remaining class 0 and class 1 samples become the test set.

Listing 4-1 shows the code to construct the subsets using a 90/5/5 split of the original data.

```
import numpy as np
from sklearn.datasets import make_classification

❶ a,b = make_classification(n_samples=10000, weights=(0.9,0.1))
idx = np.where(b == 0)[0]
x0 = a[idx,:]
y0 = b[idx]
idx = np.where(b == 1)[0]
```

```
  x1 = a[idx,:]
  y1 = b[idx]

❷ idx = np.argsort(np.random.random(y0.shape))
  y0 = y0[idx]
  x0 = x0[idx]
  idx = np.argsort(np.random.random(y1.shape))
  y1 = y1[idx]
  x1 = x1[idx]

❸ ntrn0 = int(0.9*x0.shape[0])
  ntrn1 = int(0.9*x1.shape[0])
  xtrn = np.zeros((int(ntrn0+ntrn1),20))
  ytrn = np.zeros(int(ntrn0+ntrn1))
  xtrn[:ntrn0] = x0[:ntrn0]
  xtrn[ntrn0:] = x1[:ntrn1]
  ytrn[:ntrn0] = y0[:ntrn0]
  ytrn[ntrn0:] = y1[:ntrn1]

❹ n0 = int(x0.shape[0]-ntrn0)
  n1 = int(x1.shape[0]-ntrn1)
  xval = np.zeros((int(n0/2+n1/2),20))
  yval = np.zeros(int(n0/2+n1/2))
  xval[:(n0//2)] = x0[ntrn0:(ntrn0+n0//2)]
  xval[(n0//2):] = x1[ntrn1:(ntrn1+n1//2)]
  yval[:(n0//2)] = y0[ntrn0:(ntrn0+n0//2)]
  yval[(n0//2):] = y1[ntrn1:(ntrn1+n1//2)]

❺ xtst = np.concatenate((x0[(ntrn0+n0//2):],x1[(ntrn1+n1//2):]))
  ytst = np.concatenate((y0[(ntrn0+n0//2):],y1[(ntrn1+n1//2):]))
```

Listing 4-1: Exact construction of training, validation, and test datasets

There's a lot of bookkeeping in this code. First, we create the dummy dataset ❶ and split it into class 0 and class 1 collections, stored in `x0,y0` and `x1,y1`, respectively. We then randomize the ordering ❷. This will let us pull off the first *n* samples for the subsets without worrying that we might be introducing a bias because of ordering in the data. Because of how `sklearn` generates the dummy dataset, this step isn't required, but it's always a good idea to ensure randomness in the ordering of samples.

We use a trick that's helpful when reordering samples. Because we store the feature vectors in one array and the labels in another, the NumPy shuffle methods will not work. Instead, we generate a random vector of the same length as our number of samples and then use `argsort` to return the indices of the vector that would put it in sorted order. Since the values in the vector are random, the ordering of the indices used to sort it will also be random. These indices then reorder the samples and labels so that the each label is still associated with the correct feature vector.

Next, we extract the first 90 percent of samples for the two classes and build the training subset with samples in `xtrn` and labels in `ytrn` ❸. We do the same for the 5 percent validation set ❹ and the remaining 5 percent for the test set ❺.

Partitioning by class is tedious, to say the least. We do know, however, that the class 0 to class 1 ratio in each of the subsets is exactly the same.

Random Sampling

Must we be so precise? In general, no. The second common method for partitioning the full dataset is via random sampling. If we have enough data—and 10,000 samples is enough data—we can build our subsets by randomizing the full dataset and then extracting the first 90 percent of samples as the training set, the next 5 percent as the validation set, and the last 5 percent as the test set. This is what we show in Listing 4-2.

```
❶ x,y = make_classification(n_samples=10000, weights=(0.9,0.1))
  idx = np.argsort(np.random.random(y.shape[0]))
  x = x[idx]
  y = y[idx]

❷ ntrn = int(0.9*y.shape[0])
  nval = int(0.05*y.shape[0])

❸ xtrn = x[:ntrn]
  ytrn = y[:ntrn]
  xval = x[ntrn:(ntrn+nval)]
  yval = y[ntrn:(ntrn+nval)]
  xtst = x[(ntrn+nval):]
  ytst = y[(ntrn+nval):]
```

Listing 4-2: Random construction of training, validation, and test datasets

We randomize the dummy dataset stored in `x` and `y` ❶. We need to know how many samples to include in each of the subsets. First, the number of samples for the training set is 90 percent of the total in the dataset ❷, while the number in the validation set is 5 percent of the total. The remainder, also 5 percent, is the test set ❸.

This method is so much simpler than the one shown in Listing 4-1. What's the downside of using it? The possible downside is that the mix of classes in each of these subsets might not quite be the fractions we want. For example, imagine we want a training set of 9,000 samples, or 90 percent of the original 10,000 samples, with 8,100 of them from class 0, and 900 of them from class 1. Running the Listing 4-2 code 10 times gives the splits between class 0 and class 1 in the training set that are shown in Table 4-9.

Table 4-9: Ten Training Splits Generated by Random Sampling

Run	Class 0	Class 1
1	8058 (89.5)	942 (10.5)
2	8093 (89.9)	907 (10.1)
3	8065 (89.6)	935 (10.4)
4	8081 (89.8)	919 (10.2)
5	8045 (89.4)	955 (10.6)
6	8045 (89.4)	955 (10.6)
7	8066 (89.6)	934 (10.4)
8	8064 (89.6)	936 (10.4)
9	8071 (89.7)	929 (10.3)
10	8063 (89.6)	937 (10.4)

The number of samples in class 1 ranges from as few as 907 samples to as many as 955 samples. As the number of samples of a particular class in the full dataset decreases, the number in the subsets will start to vary more. This is especially true of smaller subsets, like the validation and test sets. Let's do a separate run, this time looking at the number of samples from each class in the *test* set (Table 4-10).

Table 4-10: Ten Test Splits Generated by Random Sampling

Run	Class 0	Class 1
1	446 (89.2)	54 (10.8)
2	450 (90.0)	50 (10.0)
3	444 (88.8)	56 (11.2)
4	450 (90.0)	50 (10.0)
5	451 (90.2)	49 (9.8)
6	462 (92.4)	38 (7.6)
7	441 (88.2)	59 (11.8)
8	449 (89.8)	51 (10.2)
9	449 (89.8)	51 (10.2)
10	438 (87.6)	62 (12.4)

In the test set, the number of samples from class 1 ranges from 38 to 62.

Will these differences influence how the model learns? Probably not, but they might make the test results look better than they are, as most models struggle to identify the classes that are least common in the training set. The possibility exists of a pathological split that results in having no examples from a particular class, but in practice, it's not really that likely unless your pseudorandom number generator is particularly poor. Still, it's worth keeping the possibility in mind. If concerned, use the exact split approach in Listing 4-1. In truth, the better solution is, as always, to get more data.

Algorithmically, the steps to produce the training, validation, and test splits are as follows:

1. Randomize the order of the full dataset so that classes are evenly mixed.
2. Calculate the number of samples in the training (`ntrn`) and validation (`nval`) sets by multiplying the number of samples in the full dataset by the desired fraction. The remaining samples will fall into the test set.
3. Assign the first `ntrn` samples to the training set.
4. Assign the next `nval` samples to the validation set.
5. Finally, assign the remaining samples to the test set.

At all times, ensure that the order of the samples is truly random, and that when reordering the feature vectors, you're sure to reorder the labels in the exact same sequence. If this is done, this simple splitting process will give a good split unless the dataset is very small or some classes are very rare.

We neglected to discuss one consequence of this approach. If the full dataset is small to begin with, partitioning it will make the training set even smaller. In Chapter 7, we'll see a powerful approach to dealing with a small dataset, one that's used heavily in deep learning. But first, let's look at a principled way to work with a small dataset to get an idea of how well it will perform on new data.

k-*Fold Cross Validation*

Modern deep learning models typically need very large datasets, and therefore, you're able to use a single training/validation/test split as described previously. More traditional machine learning models, like those in Chapter 6, however, often work with datasets that are too small (in general) for deep learning models. If we use a single training/validation/test split on those datasets, we might be holding too much data back for testing, or else have too few samples in the test set to get a meaningful measurement of how well the model is working.

One way to address this issue is to use *k-fold cross validation*, a technique that ensures each sample in the dataset is used at some point for training and testing. Use this technique for small datasets intended for traditional machine learning models. It can also be helpful as a way to decide between different models.

To do *k*-fold cross validation, first partition the full, randomized dataset into *k* non-overlapping groups, $x_0, x_1, x_2, \ldots, x_{k-1}$. Your *k* value is arbitrary, though it typically ranges from 5 to 10. Figure 4-4a shows this split, imagining the entire dataset laid out horizontally.

We can train a model by holding x_0 back as test data and using the other groups, $x_1, x_2, \ldots, x_{k-1}$ as training data. We'll ignore validation data for the time being; after building the current training data, we can always hold some of it back as validation data if we want. Call this trained model m_0. You can then start over from scratch, this time holding back x_1 as test data and training with all the other groups, including x_0. We'll get a new trained model. Call it m_1. By design, m_0 and m_1 are the same *type* of model. What we are interested in here is multiple instances of the same type of model trained with different subsets of the full dataset.

Repeat this process for each of the groups, as in Figure 4-4b, and we'll have k models trained with $(k-1)/k$ of the data each, holding $1/k$ of the data back for testing. What k should be depends upon how much data is in the full dataset. Larger k means more training data but less test data. If the per model training time is low, tend toward a larger k as this increases the per model training set size.

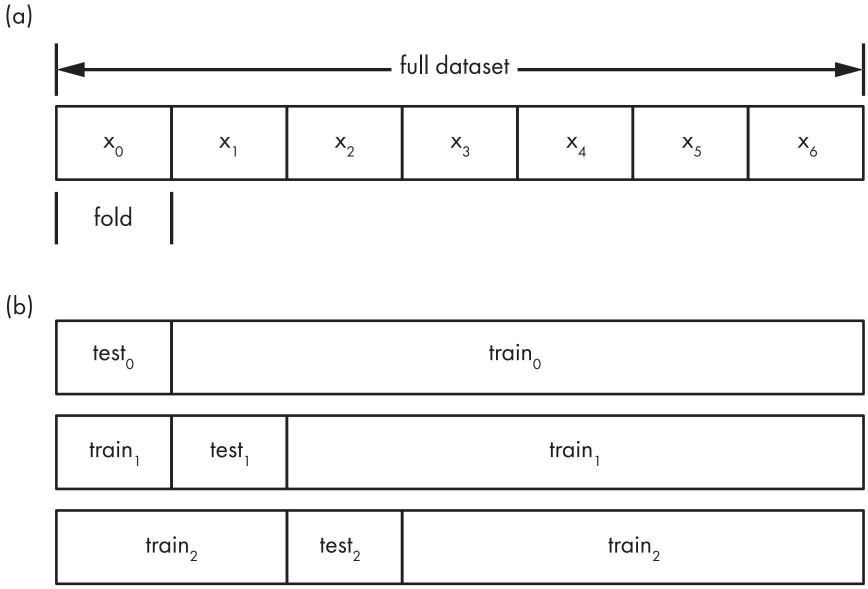


Figure 4-4: k-*fold cross validation. Partitioning the dataset into non-overlapping regions,* k=*7(a). The first three train/test splits using first* x$_0$ *for test, then* x$_1$ *for test, and so on (b).*

Once the k models are trained, you can evaluate them individually and average their metrics to get an idea of how a model trained on the full dataset would behave. See Chapter 11 to learn about ways to evaluate a model. If using k-fold cross validation to select among two or more models (say, between using k-NN or a Support Vector Machine[1]), repeat the full training and evaluation process for each type of model and compare their results. Once we have an idea of how well the model is performing on the averaged evalua-

[1] These are examples of classical machine learning models. We'll learn more about them later in the book.

tion metrics, we can start over again and train the selected model type using *all* of the dataset for training. This is the advantage of *k*-fold cross validation: it lets you have your cake and eat it, too.

Look at Your Data

It's quite easy to assemble features and feature vectors, and then go ahead and put the training, validation, and test sets together without pausing to *look* at the data to see if it makes sense. This is especially true with deep learning models using huge collections of images or other multidimensional data. Here are a few problems you'll want to look out for:

Mislabeled data Assume we're building a large dataset—one with hundreds of thousands of labeled samples. Further, assume that we're going to use the dataset to build a model that will be able to tell the difference between dogs and cats. Naturally, we need to feed the model many, many dog images and many, many cat images. No problem, you say; we'll just collect a lot of images using something like Google Images. Okay, that'll work. But if you simply set up a script to download image search results matching "dog" and "cat," you'll also get a lot of other images that are not of dogs or cats, or images that contain dogs and cats along with other things. The labels won't be perfect. While it is true that deep learning models can be resistant to such label noise, you want to avoid it whenever possible.

Missing or outlier data Imagine you have a collection of feature vectors, and you have no idea how common it is that features are missing. If a large percentage of a particular feature is missing, that feature will become a hindrance to the model and you should eliminate it. Or, if there are extreme outliers in the data, you might want to remove those samples, especially if you're going to standardize, since outliers will strongly affect the mean subtracted from the feature values.

Searching for Problems in the Data

How can we look for these problems in the data? Well, for feature vectors, we can often load the dataset into a spreadsheet, if it isn't too large. Or we could write a Python script to summarize the data, feature by feature, or bring the data into a statistics program and examine it that way.

Typically, when summarizing values statistically, we look at the mean and standard deviation, both defined previously, as well as the largest value and the smallest value. We could also look at the median, which is the value we get when we sort the values from smallest to largest and pick the one in the middle. (If the number of values is even, we'd average the two middle values.) Let's look at one of the features from our earlier example. After sorting the values from smallest to largest, we can summarize the data in the following way.

x_2
0.0705
0.3408
0.9804
1.5362
1.7821
2.0085
2.1271
2.3190
2.3944
2.7561
2.7887
3.0150
3.9897
4.3465
4.8231

Mean ($\bar{x}$)	=	2.3519
Standard deviation (σ)	=	1.3128
Standard error (SE)	=	0.3390
Median	=	2.3190
Minimum	=	0.0705
Maximum	=	4.8231

We've already explored the concepts of mean, minimum, maximum, and standard deviation. The median is there, as well; I've highlighted it in the list of features on the left. Notice that after sorting, the median appears in the exact middle of the list. It's often known as the *50th percentile*, because the same amount of data is above it as below.

There is also a new value listed, the *standard error*, also called the *standard error of the mean*. This is the standard deviation divided by the square root of the number of values in the dataset:

$$\text{SE} = \frac{\sigma}{\sqrt{n}}$$

The standard error is a measure of the difference between our mean value, $\bar{x}$, and the mean value of the parent distribution. The basic idea is this: if we have more measurements, we'll have a better idea of the parent distribution that is generating the data, and so the mean value of the measurements will be closer to the mean value of the parent distribution.

Notice also that the mean and the median are relatively close to each other. The phrase *relatively close* has no rigorous mathematical meaning, of course, but we can use it as an ad hoc indicator that the data might be normally distributed, meaning we could reasonably replace the missing values by the mean (or median), as we saw previously.

The preceding values were computed easily using NumPy, as seen in Listing 4-3.

```
  import numpy as np

❶ f = [0.3408,3.0150,4.3465,2.1271,2.7561,
       2.7887,4.8231,0.0705,3.9897,0.9804,
       2.3944,2.0085,1.7821,1.5362,2.3190]
  f = np.array(f)
  print
  print("mean  = %0.4f" % f.mean())
  print("std   = %0.4f" % f.std())
❷ print("SE    = %0.4f" % (f.std()/np.sqrt(f.shape[0])))
  print("median= %0.4f" % np.median(f))
  print("min   = %0.4f" % f.min())
  print("max   = %0.4f" % f.max())
```

Listing 4-3: Calculating basic statistics. See feature_stats.py*.*

After loading NumPy, we manually define the x_2 features (f) and turn them into a NumPy array ❶. Once the data is a NumPy array, calculating the desired values is straightforward, as all of them, except the standard error, are simple method or function calls. The standard error is calculated via the preceding formula ❷ where the first element of the tuple NumPy returns for the shape is the number of elements in a vector.

Numbers are nice, but pictures are often better. You can visualize the data with a *box plot* in Python. Let's generate one to view the standardized values of our dataset. Then we'll discuss what the plot is showing us. The code to create the plot is in Listing 3-2.

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

❶ d = [[ 0.6930, -1.1259, -1.5318,  0.9525,  1.1824],
       [ 0.5464, -0.0120,  0.5051, -0.0192, -0.1141],
       [ 0.8912,  1.3826,  1.5193, -1.1996, -1.1403],
       [ 1.1690,  0.4970, -0.1712, -0.5340,  0.3047],
       [-0.9221, -0.1071,  0.3079, -0.3885, -0.4753],
       [ 1.5699, -1.4767,  0.3327,  1.4714,  1.1807],
       [-0.3479,  0.4775,  1.8823, -1.4031, -0.7396],
       [ 0.0887, -0.4353, -1.7377, -1.2349,  1.7456],
       [ 1.0775,  0.9524,  1.2475,  0.7291, -1.1207],
       [-1.4657,  0.9250, -1.0446,  0.4262, -1.0279],
       [-1.3332,  1.4501,  0.0323,  1.1102, -0.8966],
       [ 0.3005, -1.4500, -0.2615,  1.7033, -0.2505],
       [-1.4377, -0.2472, -0.4340, -0.7032,  0.3362],
       [ 0.3016, -1.5527, -0.6213,  0.1780, -0.7517],
       [-1.1315,  0.7225, -0.0250, -1.0881,  1.7674]]
❷ d = np.array(d)
  plt.boxplot(d)
  plt.show()
```

Listing 4-4: A box plot of the standardized toy dataset. See box_plot.py*.*

The values themselves are in Table 4-6. We can store the data as a 2D array and make the box plot using Listing 4-4. We manually define the array ❶ and then plot it ❷. The plot is interactive, so experiment with the environment provided until you feel comfortable with it. The old-school floppy disk icon will store the plot to your disk.

The box plot generated by the program is shown in Figure 4-5.

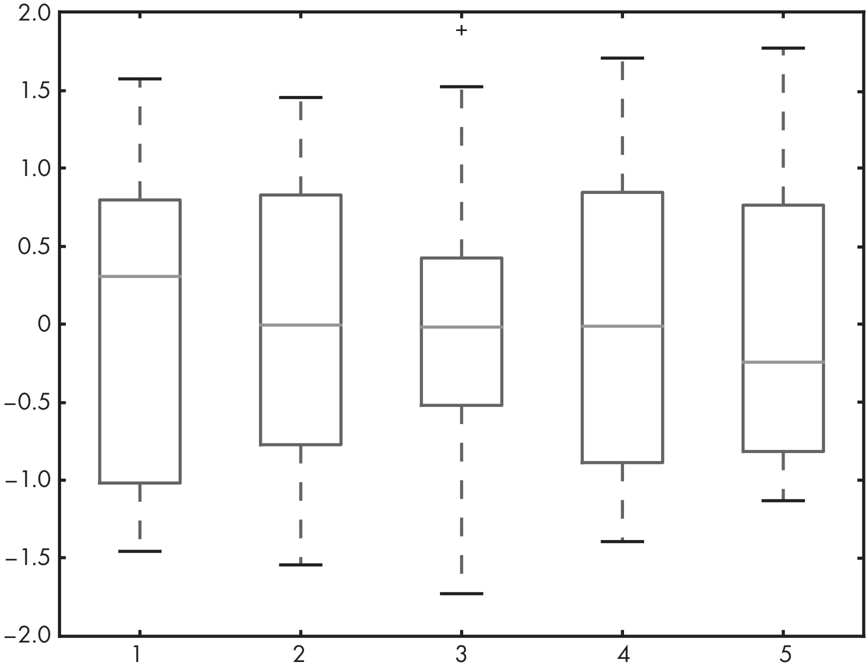


Figure 4-5: The box plot produced by Listing 4-4

How do we interpret the box plot? I'll show you by examining the box representing the standardized feature x_2, shown in Figure 4-6.

The lower box line, Q1, marks the end of the first quartile. This means that 25 percent of the data values for a feature are less than this value. The median, Q2, is the 50 percent mark, and therefore is the end of the second quartile. Half the data values are less than this value. The upper box line, Q3, is the 75 percent mark. The remaining 25 percent of the data values are above Q3.

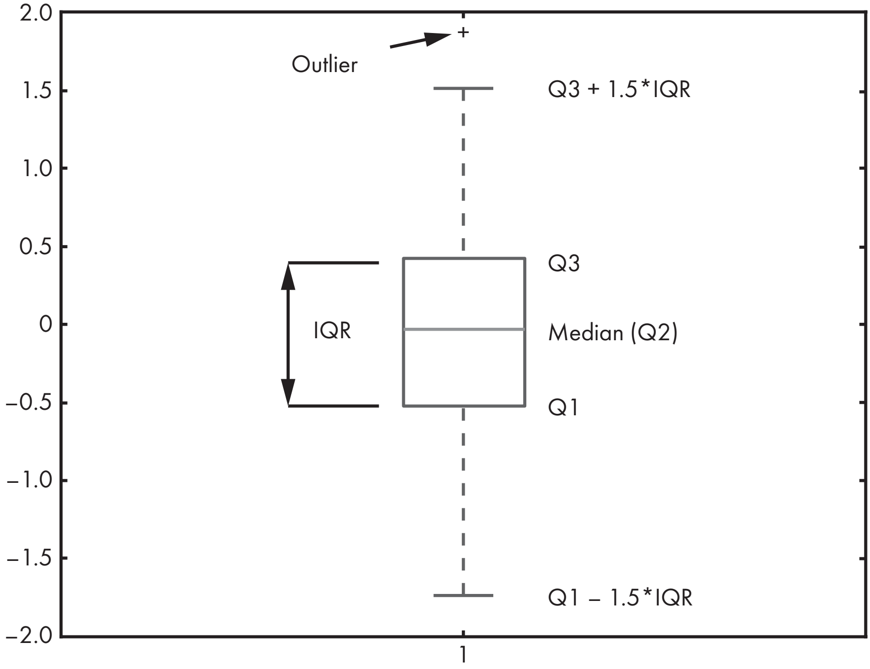


Figure 4-6: The standardized feature x_2 from our dataset

Also shown are two lines above and below the box. These are the *whiskers*. (Matplotlib calls them *fliers*, but this is an unconventional term.) The whiskers are the values at Q1 − 1.5 × IQR and Q3 + 1.5 × IQR. By convention, values outside this range are considered *outliers*.

Looking at outliers can be helpful, because you might realize they're mistakes in data entry and drop them from the dataset. Whatever you do with the outliers, however, be prepared to justify it should you ever plan on publishing or otherwise presenting results based on the dataset. Similarly, you might be able to drop samples with missing values, but make sure there's no systematic error causing the missing data, and check that you're not introducing bias into the data by dropping those samples. In the end, common sense should override slavish adherence to convention.

Cautionary Tales

So, at the risk of being repetitive, *look at your data*. The more you work with it, the more you will understand it, and the more effectively you will be able to make reasonable decisions about what goes in and what comes out, and *why*. Recall that the goal of the dataset is to faithfully and completely capture the parent distribution, or what the data will look like in the wild when the model is used.

Two quick anecdotes come to mind. They both illustrate ways models may well learn things we did not intend or even consider.

The first was told to me as an undergraduate student in the 1980s. In this story, an early form of neural network was tasked with detecting tank and non-tank images. The neural network seemed to work well in testing, but when used in the field, the detection rate dropped rapidly. The researchers realized that the tank images were taken on a cloudy day, and the non-tank were taken on a sunny day. The recognition system had not learned the difference between tanks and non-tanks at all; instead, it had learned the difference between cloudy and sunny days. The moral of this story is that the training set needs to include *all* of the conditions the model will see in the wild.

The second anecdote is more recent. I heard it in a talk at the Neural Information Processing Systems (NIPS) 2016 conference in Barcelona, Spain, and later found it repeated in the researchers' paper.[2] In this case, the authors, who were demonstrating their technique for getting a model to explain its decisions, trained a model that claimed to tell the difference between images of huskies and images of wolves. The model appeared to work rather well, and during the talk, the authors polled the audience composed of machine learning researchers about how believable the model was. Most thought it was a good model. Then, using their technique, the speaker revealed that the network had not learned much, if anything, about the difference between huskies and wolves. Instead, it had learned that the wolf pictures had snow in the background and the husky pictures did not.

Think about your data and be on the lookout for unintended consequences. Models are not human. We bring a lot of preconceived notions and unintended biases to the dataset.

Summary

In this chapter, we described the components of a dataset (classes, labels, features, feature vectors) and then characterized a good dataset, emphasizing the importance of ensuring that the dataset well represents the parent distribution. We then described basic data preparation techniques including how to scale data and one approach for dealing with missing features. After that, we learned how to separate the full dataset into training, validation, and test subsets and how to apply *k*-fold cross validation, which is especially useful with small datasets. We ended the chapter with tips on how to simply examine the data to make sure it makes sense.

In the next chapter, we'll take what we have learned in this chapter and apply it directly to construct the datasets we will use throughout the remainder of this book.

[2] Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?: Explaining the Predictions of Any Classifier." *In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1135–1144. ACM, 2016.