9 Feature Engineering and Selection

9.1 Introduction

In this chapter we discuss the principles of *feature engineering* and *selection*.

Feature engineering methods consist of an array of techniques that are applied to data *before* they are used by either supervised or unsupervised models. Some of these tools, e.g., the *feature scaling* techniques that we describe in Sections 9.3 through 9.5, properly *normalize* input data and provide a consistent preprocessing pipeline for learning, drastically improving the efficacy of many local optimization methods.

Another branch of feature engineering focuses on the development of data transformations that extract useful information from raw input data. For example, in the case of a two-class classification, these tools aim to extract critical elements of a dataset that ensure instances within a single class are seen as "similar" while those from different classes are "dissimilar." Designing such tools often requires significant domain knowledge and a rich set of experiences dealing with particular kinds of data. However, we will see that one simple concept, the *histogram* feature transformation, is a common feature engineering tool used for a variety of data types including categorical, text, image, and audio data. We give a high-level overview of this popular approach to feature engineering in Section 9.2.

Human beings are often an integral component of the machine learning paradigm, and it can be crucial that individuals be able to interpret and/or derive insights from a machine learning model. The *performance* of a model is a common and relatively easy metric for humans to interpret: does the model provide good[1] predictive results? Sometimes it is very useful to understand *which* input features were the most pertinent to achieving strong performance, as it helps us refine our understanding of the nature of the problem at hand. This is done through what is called *feature selection*. In Sections 9.6 and 9.7 we discuss two popular ways for performing feature selection: *boosting* and *regularization*.

Feature selection can be thought of as a supervised dimension reduction technique that reduces the total number of features involved in a regression or classification, making the resulting model more human interpretable. An abstract

[1] Here "good" can mean, for instance, that the learner achieves an agreed upon benchmark value for accuracy, error, etc.

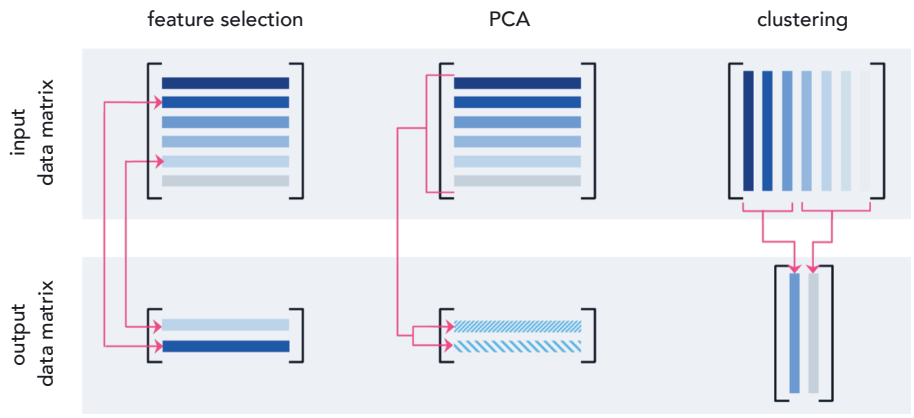

Figure 9.1 A prototypical comparison of feature selection, PCA, and clustering as dimension reduction schemes on an arbitrary data matrix, like those we have discussed in previous chapters for un/supervised learning, whose rows contain features and columns individual data points. The former two methods reduce the dimension of the feature space, or in other words the number of rows in a data matrix. However, the two methods work differently: while feature selection literally selects rows from the original matrix to keep, PCA uses the geometry of the feature space to produce a new data matrix based on a lower feature dimensional version of the data. K-means, on the other hand, reduces the dimension of the data/number of data points, or equivalently the number of columns in the input data matrix. It does so by finding a small number of new averaged representatives or "centroids" of the input data, forming a new data matrix whose fewer columns (which are not present in the original data matrix) are precisely these centroids.

illustration of this concept is shown in the left panel of Figure 9.1, and is compared visually to the result of both Principal Component Analysis (PCA) and clustering (two unsupervised learning techniques introduced in the previous chapter). In contrast to feature selection, when PCA (see Section 8.3) is applied to reduce the dimension of a dataset it does so by learning a new (smaller) set of features over which the dataset may be fairly represented. Likewise any clustering technique (like K-means detailed in Section 8.5) learns new representations to reduce the shear number of points in a dataset.

9.2 Histogram Features

A histogram is an extremely simple yet useful way of summarizing and representing the contents of an array. In this section we see how this rather simple concept is at the core of designing features for common types of input data including categorical, text, image, and audio data types. Although each of these data types differs substantially in nature, we will see how the notion of a histogram-based feature makes sense in each context. This discussion aims at giving the reader

a high-level, intuitive understanding of how common histogram-based feature methods work. The interested reader is encouraged to consult specialized texts (referenced throughout this section) on each data type for further study.

9.2.1 Histogram features for categorical data

Every machine learning paradigm requires that the data we deal with consist strictly of *numerical* values. However, raw data does not always come pre-packaged in this manner. Consider, for instance, a hypothetical medical dataset consisting of several patients' vital measurements such as blood pressure, blood glucose level, and blood type. The first two features (i.e., blood pressure and blood glucose level) are naturally numerical, and hence ready for supervised or unsupervised learning. Blood type, on the other hand, is a *categorical* feature, taking on the values O, A, B, and AB. Such categorical features need to be translated into numerical values before they can be fed into any machine learning algorithm.

A first, intuitive approach to do this would be to represent each category with a distinct real number, e.g., by assigning 0 to the blood type O, 1 to A, 2 to B, and 3 to AB, as shown in the top left panel of Figure 9.2. Here the way we assign numbers to each category is important. In this particular instance, by assigning 1 to A, 2 to B, and 3 to AB, and because the number 3 is closer to number 2 than it is to number 1, we have inadvertently injected the assumption into our dataset that an individual with blood type AB is more "similar" to one with blood type B than one with blood type A. One could argue that it is more appropriate to switch the numbers used to represent categories B and AB so that AB now sits between (and hence equidistant from) A and B, as shown in the top right panel of Figure 9.2. However, with this reassignment blood type O is now interpreted as being maximally different from blood type B, a kind of assumption that *may* or *may not* be true in reality.

The crux of the matter here is that there is always a natural order to any set of numbers, and by using such values we inevitably inject assumptions about *similarity* or *dissimilarity* of the existing categories into our data. In most cases we want to avoid making such assumptions that fundamentally change the geometry of the problem, especially when we lack the intuition or knowledge necessary for determining similarity between different categories.

A better way of encoding categorical features, which avoids this problem, is to use a *histogram* whose bins are all categories present in the categorical feature of interest. Doing this in the example of blood type, an individual with blood type O is no longer represented by a single number but by a four-binned histogram that contains all zero values except the bin representing blood type O, which is set to 1. Individuals with other blood types are represented similarly, as depicted visually in the bottom panel of Figure 9.2. This way, all blood type representations (each a four-dimensional vector with a single 1 and three 0s) are

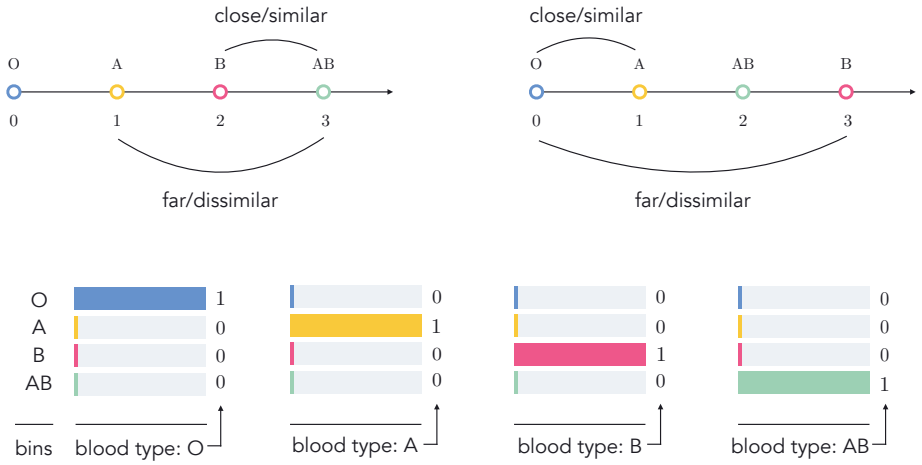

Figure 9.2 Blood type converted into numerical features (top panels) and histogram-based features (bottom panel). See text for further details.

geometrically equidistant from one another. This method of encoding categorical features is sometimes referred to as *one-hot encoding* (see Section 6.7.1).

9.2.2 Histogram features for text data

Many popular uses of machine learning, including sentiment analysis, spam detection, and document categorization or clustering are based on text data, e.g., online news articles, emails, social media posts, etc. However, with text data, the initial input (i.e., the document itself) requires a significant amount of preprocessing and transformation prior to being input into any machine learning algorithm. A very basic but widely used feature transformation of a document for machine learning tasks is called a *Bag of Words* (BoW) histogram or feature vector. Here we introduce the BoW histogram and discuss its strengths, weaknesses, and common extensions.

A BoW feature vector of a document is a simple histogram count of the different words it contains with respect to a single corpus or collection of documents, minus those nondistinctive words that do not characterize the document (in the context of the application).

To illustrate this idea let us build a BoW representation for the following corpus of two simple text documents each consisting of a single sentence.

1. `dogs are the best`
2. `cats are the worst` (9.1)

To make the BoW representation of these documents we begin by parsing them,

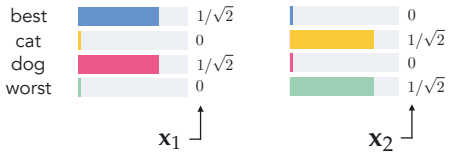

Figure 9.3 Bag of Words histogram features for the two example documents shown in Equation (9.1). See text for further details.

creating representative vectors (histograms) $\mathbf{x}_1$ and $\mathbf{x}_2$ which contain the number of times each word appears in each document. The BoW vectors for both of these documents are depicted visually in Figure 9.3.

Notice that, in creating the BoW histograms, uninformative words such as `are` and `the`, typically referred to as *stop words*, are not included in the final representation. Further, notice that we count the singular `dog` and `cat` in place of their plural, which appeared in the original documents. This preprocessing step is commonly called *stemming*, where related words with a common stem or root are reduced to (and then represented by) their common linguistic root. For instance, the words `learn`, `learning`, `learned`, and `learner`, in the final BoW feature vector are all represented by and counted as `learn`. Additionally, each BoW vector is *normalized* to have unit length.

Given that BoW vectors contain only nonnegative entries and all have unit length, the inner product between two BoW vectors $\mathbf{x}_1$ and $\mathbf{x}_2$ always ranges between 0 and 1, i.e., $0 \leq \mathbf{x}_1^T \mathbf{x}_2 \leq 1$. This inner product or *correlation* value may be used as a rough geometric measure of similarity between two BoW vectors. For example, when two documents are made up of completely different words, the correlation is exactly zero, their BoW vectors are perpendicular to one other, and the two documents can be considered maximally different. This is the case with BoW vectors shown in Figure 9.3. On the other hand, the higher the correlation between two BoW vectors the more similar their respective documents are purported to be. For example, the BoW vector of the document `I love dogs` would have a rather large positive correlation with that of `I love cats`.

Because the BoW vector is such a simple representation of a document, completely ignoring word order, punctuation, etc., it can only provide a *gross summary* of a document's contents. For example, the two documents `dogs are better than cats` and `cats are better than dogs` would be considered *exactly the same document* using BoW representation, even though they imply completely opposite meanings. Nonetheless, the gross summary provided by BoW can be distinctive enough for many applications. Additionally, while more complex representations of documents (capturing word order, parts of speech, etc.) may be employed they can often be unwieldy (see, e.g., [32]).

Example 9.1 Sentiment analysis

Determining the aggregated feelings of a large base of customers, using text-based content like product reviews, tweets, and social media comments, is commonly referred to as *sentiment analysis*. Classification models are often used to perform sentiment analysis, learning to identify consumer data of either positive or negative feelings.

For example, the top panel of Figure 9.4 shows BoW vector representations for two brief reviews of a controversial comedy movie, one with a positive opinion and the other with a negative one. The BoW vectors are rotated sideways in this figure so that the horizontal axis contains the common words between the two sentences (after stop word removal and stemming). The polar opposite sentiment of these two reviews is perfectly represented in their BoW representations, which as one can see are indeed perpendicular (i.e., they have zero correlation). In general, two documents with opposite sentiments are not and need not always be perpendicular for sentiment analysis to work effectively, even though we ideally expect them to have small correlations, as shown in the bottom panel of Figure 9.4.

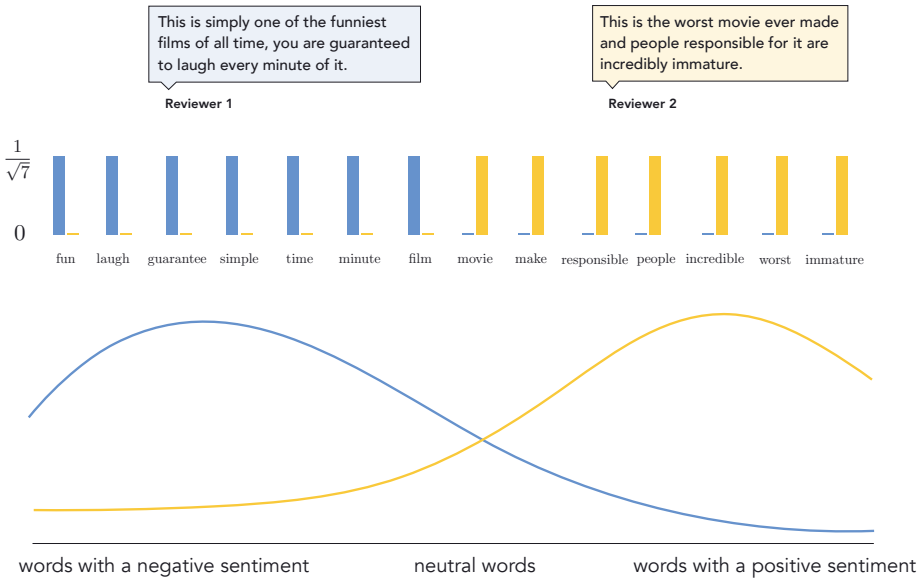

Figure 9.4 Figure associated with Example 9.1. (top panel) BoW representation of two movie review excerpts, with words (after the removal of stop words and stemming) shared between the two reviews listed along the horizontal axis. The vastly different opinion of each review is reflected very well by the BoW histograms, which have zero correlation. (bottom panel) In general, the BoW histogram of a typical document with positive sentiment is ideally expected to have small correlation with that of a typical document with negative sentiment.

Example 9.2 Spam detection

In many spam detectors (see Example 1.8) the BoW feature vectors are formed with respect to a specific list of spam words (or phrases) such as `free`, `guarantee`, `bargain`, `act now`, `all natural`, etc., that are frequently seen in spam emails. Additionally, features like the frequency of certain characters like `!` and `*` are appended to the BoW feature, as are other spam-targeted features including the total number of capital letters in the email and the length of longest uninterrupted sequence of capital letters, as these features can further distinguish the two classes.

In Figure 9.5 we show classification results on a spam email dataset (first introduced in Example 6.10) consisting of Bag of Words (BoW), character frequencies, and other spam-focused features. Employing the two-class Softmax cost (see Section 7.3) to learn the separator, the figure shows the number of misclassifications for each step of a run of Newton's method (see Section 4.3). More specifically, these classification results are shown for the same dataset using only BoW features alone (in blue), BoW and character frequencies (in orange), and the BoW/character frequencies as well as spam-targeted features (in green). Unsurprisingly the addition of character frequencies improves the classification, with the best performance occurring when the spam-focused features are used as well.

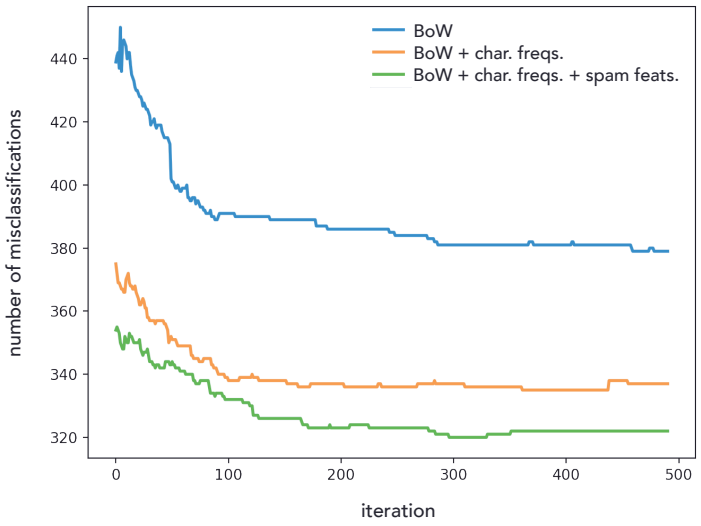

Figure 9.5 Figure associated with Example 9.2. See text for details.

9.2.3 Histogram features for image data

To perform supervised/unsupervised learning tasks on image data, such as object recognition or image compression, the raw input data are pixel values of an image itself. The pixel values of an 8-bit grayscale image are each just a single

integer in the range of 0 (black) to 255 (white), as illustrated in Figure 9.6. In other words, a grayscale image is just a *matrix* of integers ranging from 0 to 255. A color image is then just a set of three such grayscale matrices, one for each of the red, blue, and green channels.

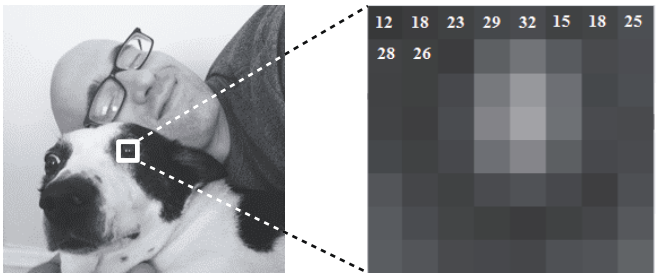

Figure 9.6 An 8-bit grayscale image consists of pixels, each taking a value between 0 (black) and 255 (white). To visualize individual pixels, a small 8×8 block from the original image is blown up on the right.

Although it is possible to use raw pixel values directly as features, pixel values themselves are typically not discriminative enough to be useful for machine learning tasks. We illustrate why this is the case using a simple example in Figure 9.7. Consider the three images of shapes shown in the left column of this figure. The first two are similar triangles and the third shape is a square. We would like an ideal set of features to reflect the similarity of the first two images as well as their distinctness from the last image. However, due to the difference in their relative size, position in the image, and the contrast of the image itself (the image with the smaller triangle is darker toned overall) if we were to use raw pixel values to compare the images (by taking the difference between each image pair) we would find that the square and larger triangle are more similar than the two triangles themselves.[2] This is because the pixel values of the first and third image, due to their identical contrast and location of the triangle/square, are indeed more similar than those of the two triangle images.

In the middle and right columns of Figure 9.7 we illustrate a two-step procedure that generates the sort of discriminating feature transformation we are after. In the first part we shift perspective from the pixels themselves to the edge content at each pixel. By taking edges instead of pixel values we significantly reduce the amount of information we must deal with in an image without destroying its identifying structures. In the middle column of the figure we show corresponding edge detected images, in particular highlighting eight equally (angularly) spaced edge orientations, starting from 0 degrees (horizontal edges) with seven additional orientations at increments of 22.5 degrees, including 45

[2] This is to say that if we denote by $\mathbf{X}_i$ the ith image then we would find that $\|\mathbf{X}_1 - \mathbf{X}_3\|_F < \|\mathbf{X}_1 - \mathbf{X}_2\|_F$.

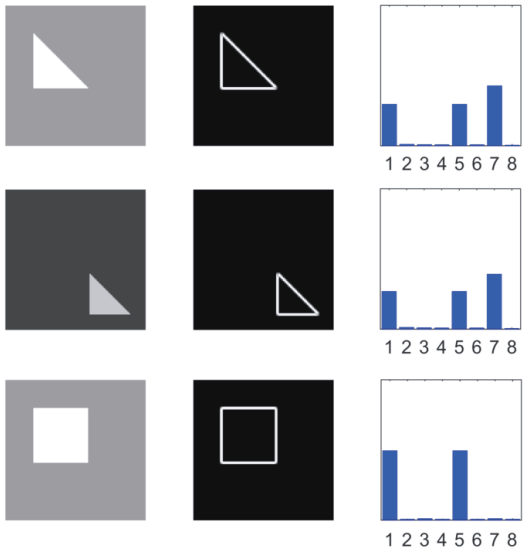

Figure 9.7 (left column) Three images of simple shapes. While the triangles in the top two images are visually similar, this similarity is not reflected by comparing their raw pixel values. (middle column) Edge detected versions of the original images, here using eight edge orientations, retain the distinguishing structural content while significantly reducing the amount of information in each image. (right column) By taking normalized histograms of the edge content we have a feature representation that captures the similarity of the two triangles quite well while distinguishing both from the square. See text for further details.

degrees (capturing the diagonal edges of the triangles) and 90 degrees (vertical edges). Clearly the edges retain distinguishing characteristics from each original image, while significantly reducing the amount of total information in each case.

We then make normalized histogram of each image's edge content (as shown for the examples in the right column of Figure 9.7). That is, we make a vector consisting of the total amount of each edge orientation found in the image (the vertical axis of the histogram) and normalize the resulting vector to have unit length. This is completely analogous to the BoW feature representation described for text data previously, with the counting of edge orientations being the analog of counting "words" in the case of text data. Here we also have a normalized histogram that represents an image grossly while ignoring the location and ordering of its information. However, as shown in the right panel of Figure 9.7 (unlike raw pixel values) these histogram feature vectors capture characteristic information about each image, with the top two triangle images having very similar histograms and both differing significantly from that of the third image of the square.

Generalizations of this simple edge histogram concept are widely used as

feature transformations for visual object recognition where the goal is to locate objects of interest (e.g., faces in a face recognition app or pedestrians in a self-driving car) in an example image or when different objects need to be distinguished from each other across multiple images (e.g., handwritten digits as in the example below, or even distinguishing cats from dogs as discussed in Section 1.2). This is due to the fact that edge content tends to preserve the structure of more complex images – like the one shown in Figure 9.8 – while drastically reducing the amount of information in the image [33, 34]. The majority of the pixels in this image do not belong to any edges, yet with just the edges we can still tell what the image contains.

Figure 9.8 (left panel) A natural image (in this instance of the two creators/writers of the television show "South Park" (this image is reproduced with permission of Jason Marck). (right panel) The edge-detected version of this image, where the bright yellow pixels indicate large edge content, still describes the scene very well (in the sense that we can still tell there are two people in the image) using only a fraction of the information contained in the original image. Note that edges have been colored yellow for visualization purposes only.

However, for such complex images, preserving local information (features of the image in smaller areas) becomes important. Thus a natural way to extend the edge histogram feature is to compute it not over the entire image, but by breaking the image into relatively small patches and computing an edge histogram of each patch, then concatenating the results. In Figure 9.9 we show a diagram of a common variation of this technique often used in practice where neighboring histograms are normalized jointly in larger blocks (see, e.g., [5, 35, 36, 37, 38] for further details).

Interestingly this edge-based histogram feature design mimics the way many animals seem to process visual information. From visual studies performed largely on frogs, cats, and primates, where a subject is shown visual stimuli while electrical impulses are recorded in a small area in the subject's brain where visual information is processed, neuroscientists have determined that individual neurons involved roughly operate by identifying edges [39, 40]. Each neuron therefore acts as a small "edge detector," locating edges in an image of a

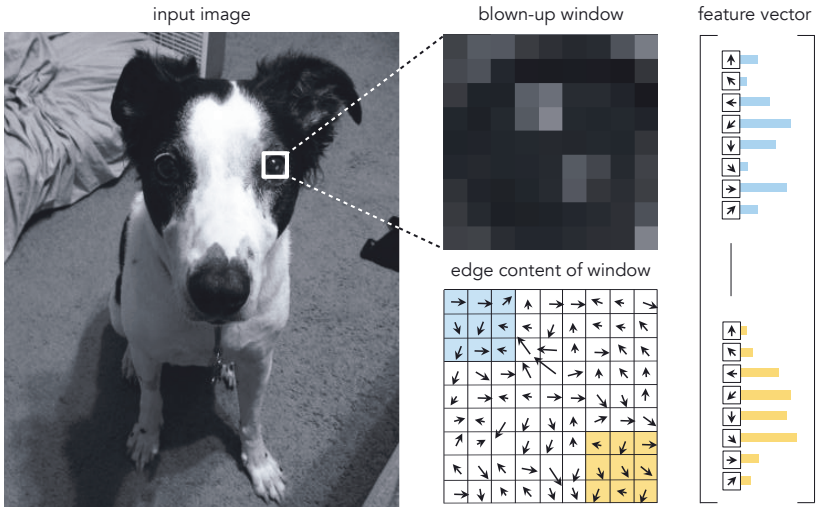

Figure 9.9 A pictorial representation of the sort of generalized edge histogram feature transformation commonly used for object detection. An input image is broken down into small (here 9×9) blocks, and an edge histogram is computed on each of the smaller nonoverlapping (here 3×3) patches that make up the block. The resulting histograms are then concatenated and normalized jointly, producing a feature vector for the entire block. Concatenating such block features by scanning the block window over the entire image gives the final feature vector.

specific orientation and thickness, as shown in Figure 9.10. It is thought that by combining and processing these edge detected images that humans and other mammals "see."

Example 9.3 Handwritten digit recognition

In this example we look at the problem of handwritten digit recognition (introduced in Example 1.10), and compare the training effectiveness of mini-batch gradient descent (20 steps/epochs with a learning rate of $\alpha = 10^{-2}$ and batch size of 200 applied to a multi-class Softmax cost) using $P = 50,000$ raw (pixel-based) data points from the MNIST handwritten digit recognition dataset (introduced in Example 7.10), to the effectiveness of precisely the same setup applied to edge histogram based features extracted from these same data points.

The effectiveness of this setup over the raw data in terms of both the cost function and misclassification history resulting from the optimization run is shown as the black curves in the left and right panels, respectively, of Figure 9.11. The results of the same run over the edge feature extracted version of the dataset is shown by the magenta curves in the same figure. Here we can see a massive performance gap, particularly in the misclassification history plot in

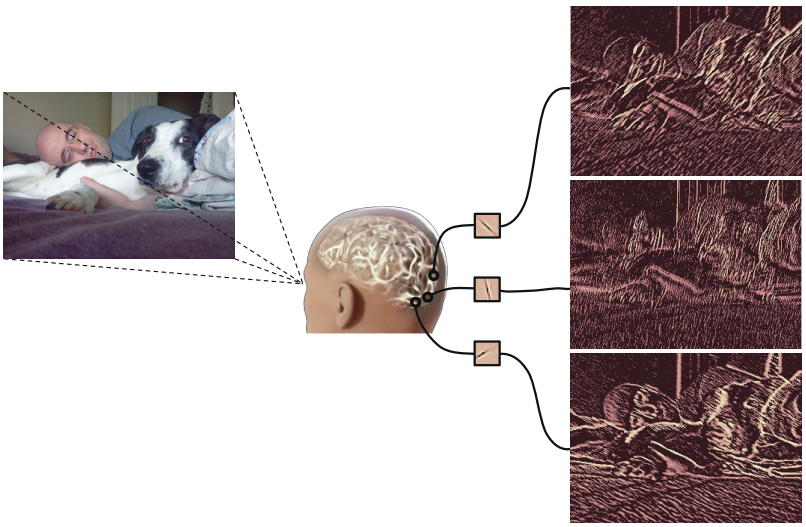

Figure 9.10 Visual information is processed in an area of the brain where each neuron detects in the observed scene edges of a specific orientation and width. It is thought that what we (and other mammals) "see" is a processed interpolation of these edge detected images.

the right panel of the figure, where the difference in performance is around 4000 misclassifications (in favor of the run over the edge-based features).

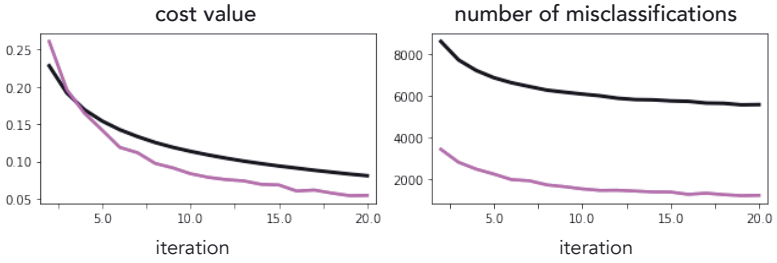

Figure 9.11 Figure associated with Example 9.3. See text for details.

9.2.4 Histogram features for audio data

Like images, audio signals in raw form are not discriminative enough to be used for audio-based classification tasks (e.g., speech recognition) and once again properly designed histogram-based features could be used. In the case of an audio signal it is the histogram of its frequencies, otherwise known as its

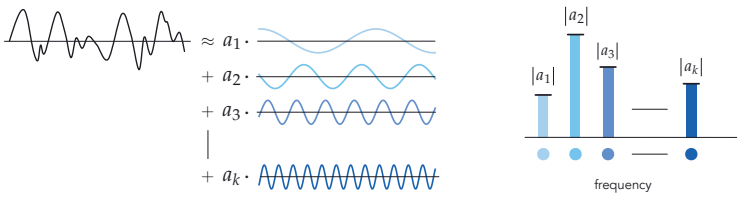

Figure 9.12 A pictorial representation of an audio signal and its representation as a frequency histogram or spectrum. (left panel) A figurative audio signal can be decomposed as a linear combination of simple sinusoids with varying frequencies (or oscillations). (right panel) The frequency histogram then contains the strength of each sinusoid in the representation of the audio signal.

spectrum, that provides a robust summary of its contents. As illustrated pictorially in Figure 9.12, the spectrum of an audio signal counts up (in histogram fashion) the strength of each level of its frequency or oscillation. This is done by decomposing the speech signal over a basis of sinusoidal waves of ever increasing frequency, with the weights on each sinusoid representing the strength of that frequency in the original signal. Each oscillation level is analogous to an edge direction in the case of an image, or an individual word in the case of a BoW text feature.

As with image data in Figure 9.9, computing frequency histograms over overlapping windows of an audio signal as illustrated pictorially in Figure 9.13, produces a feature vector that preserves important local information as well, and is a common feature transformation used for speech recognition called a *spectrogram* [41, 42]. Further processing of the windowed histograms (e.g., to emphasize the frequencies of sound best recognized by the human ear) is also commonly performed in practical implementations of this sort of feature transformation.

9.3 Feature Scaling via Standard Normalization

In this section we describe a popular method of input normalization in machine learning called *feature scaling* via *standard normalization*. This sort of feature engineering scheme provides several benefits to the learning process, including substantial improvement in learning speed when used with local optimization algorithms, and with first-order methods in particular. As such this feature engineering method can also be thought of as an *optimization trick* that substantially improves our ability to minimize virtually every machine learning model.

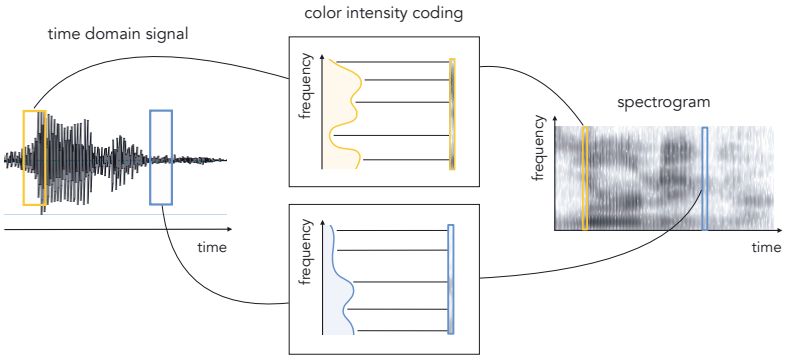

Figure 9.13 A pictorial representation of histogram-based features for audio data. The original speech signal (shown on the left) is broken up into small (overlapping) windows whose frequency histograms are computed and stacked vertically to produce a so-called *spectrogram* (shown on the right).

9.3.1 Standard normalization

Standard normalization of the input features of a dataset is a very simple two-step procedure consisting of first *mean-centering* and then *rescaling* each of its input features by the inverse of its standard deviation. Phrased algebraically, we normalize along the nth input feature of our dataset by replacing $x_{p,n}$ (the nth coordinate of the input point $\mathbf{x}_p$) with

$$\frac{x_{p,n} - \mu_n}{\sigma_n} \tag{9.2}$$

where μ_n and σ_n are the mean and standard deviation computed along the nth dimension of the data, defined respectively as

$$\begin{aligned} \mu_n &= \frac{1}{P}\sum_{p=1}^{P} x_{p,n} \\ \sigma_n &= \sqrt{\frac{1}{P}\sum_{p=1}^{P}\left(x_{p,n} - \mu_n\right)^2}. \end{aligned} \tag{9.3}$$

This (completely invertible) procedure is done for each input dimension $n = 1, ..., N$. Note that if $\sigma_n = 0$ for some n the standard normalization in Equation (9.2) is undefined as it involves division by zero. However, in this case the corresponding input feature is *redundant* since this implies that the nth feature is the same constant value across the entirety of data. As we discuss further in the next section, such a feature should be removed from the dataset in the beginning as nothing can be learned from its presence in any machine learning model.

Generally speaking, standard normalization alters the shape of machine learning cost functions by making their contours appear more "circular." This idea is illustrated in Figure 9.14 where in the top row we show a prototypical $N = 2$

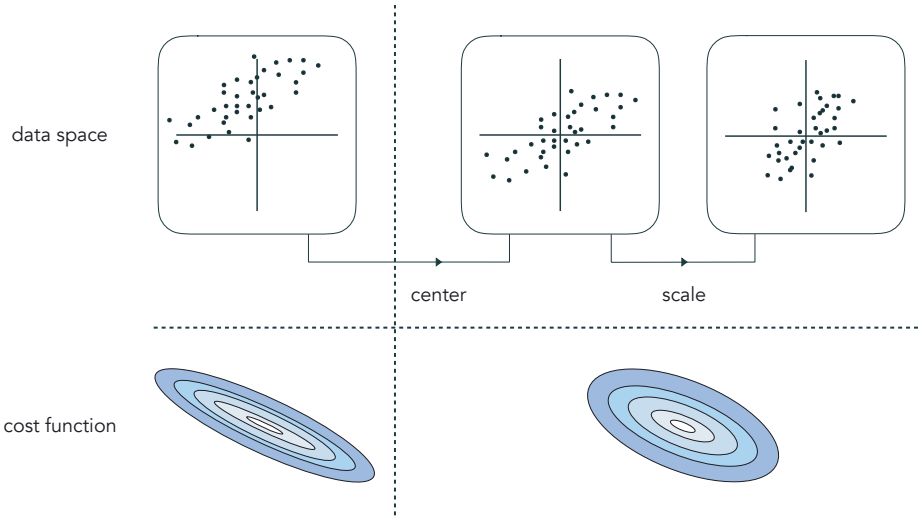

Figure 9.14 (Standard normalization illustrated. The input space of a generic dataset (top-left panel) as well as its mean-centered (top-middle panel) and scaled version (top-right panel). As illustrated in the bottom row, where a prototypical cost function corresponding to this data is shown, standard normalization results in a cost function with less elliptical and more circular contours compared to the original cost function.

dimensional dataset (top-left panel) as well as its standard normalized version (top-right panel). In the input data space, standard normalization produces a centered and more compactly confined version of the original data. Simultaneously, as shown in the bottom row of the figure, a generic cost function associated with the standard normalized version of the data has contours that are much more circular than those associated with the original, unnormalized data.

Making the contours of a machine learning cost function more circular helps speed up the convergence of local optimization schemes, particularly first-order methods like gradient descent (however, feature scaling techniques can also help to better condition a dataset for use with second-order methods, helping to potentially avoid issues with numerical instability as briefly touched on in Section 4.3.3). This is because, as detailed in Section 3.6.2, the gradient descent direction always points perpendicular to the contours of a cost function. This means that, when applied to minimize a cost function with elliptical contours like the example shown in the top panel of Figure 9.15, the gradient descent direction points *away* from the minimizer of the cost function. This characteristic naturally leads the gradient descent algorithm to take *zig-zag* steps back and forth.

In standard normalizing the input data we temper such elliptical contours, transforming them into more circular contours as shown in the bottom-left panel and (ideally) bottom-right panel of Figure 9.15. With more circular contours the gradient descent direction starts pointing more and more in the direction of

the cost function's minimizer, making each gradient descent step much more effective. This often means we can use a much larger steplength parameter α when minimizing a cost function over standard normalized data.

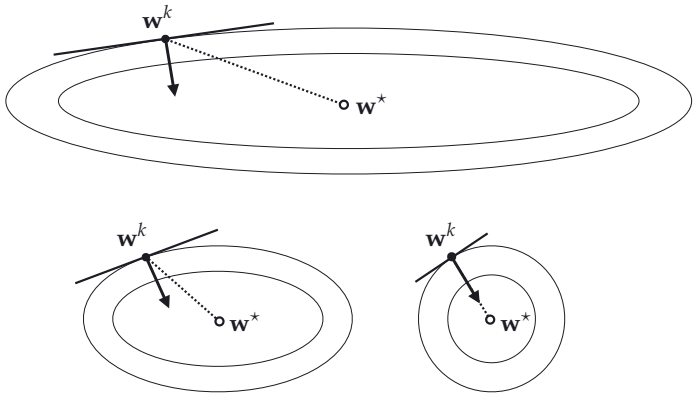

Figure 9.15 In standard normalizing input data we temper its associated cost function's often-elliptical contours, like those shown in the top panel, into more circular ones as shown in the bottom-left and bottom-right panels. This means that the gradient descent direction, which points away from the minimizer of a cost function when its contours are elliptical (leading to the common zig-zagging problem with gradient descent), points more towards the function's minimizer as its contours become more circular. This makes each gradient descent step much more effective, typically allowing the use of much larger steplength parameter values α, meaning that measurably fewer steps are required to adequately minimize the cost function.

Example 9.4 Linear regression with standard normalized data

A simple regression dataset is plotted in the top-left panel of Figure 9.16. With a quick glance at the data we can see that, if tuned properly, a linear regressor will fit this dataset exceedingly well. Since this is a low-dimensional example with only two parameters to tune (i.e., the bias and slope of a best fit line) we can visualize its associated Least Squares cost function, as illustrated in the top-middle panel of Figure 9.16. Notice how elliptical the contours of this cost function are, creating a long narrow valley along the long axis of the ellipses. In the top-middle panel of the figure we also show 100 steps of gradient descent initialized at the point $\mathbf{w}^0 = [0\ 0]^T$, using a fixed steplength parameter $\alpha = 10^{-1}$. Here the steps on the contour plot are colored from green to red as gradient descent begins (green) to when it ends (red). Examining the panel we can see that even at the end of the run we still have quite a way to travel to reach the

minimizer of the cost function. We plot the line associated with the final set of weights resulting from this run in blue in the top-left panel of the figure. Because these weights lie rather far from the true minimizer of the cost function they provoke a (relatively) poor fit of the data.

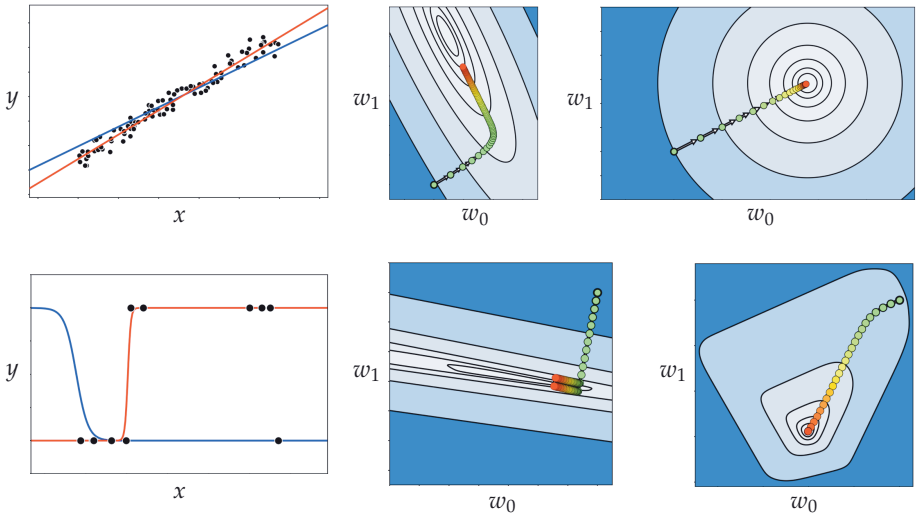

Figure 9.16 Figure associated with Examples 9.4 and 9.5, showing the result of standard normalization applied to a regression (top row) and a two-class classification (bottom row) dataset. See text for details.

We then standard normalize the data, and visualize the contour plot of the associated Least Squares cost in the top-right panel of Figure 9.16. As we can see the contours of this Least Squares cost are perfectly circular, and so gradient descent can much more rapidly minimize this cost. On top of the contour plot we show a run of 20 (instead of 100) gradient descent steps using the same initial point and steplength parameter as we used before with unnormalized data. Note that since cost function associated with the standard normalized version of the data is so much easier to optimize, we reach a point much closer to the minimizer after just a few steps, resulting in a linear model (shown in red) which fits the data in the top-left panel far better than the regressor provided by the first run (shown in blue).

Example 9.5 Linear two-class classification with standard normalized data
The bottom-left panel of Figure 9.16 shows a simple two-class classification dataset. Just like the previous example, since we only have two parameters to tune in learning a linear classifier for this dataset, it is possible to visualize the contour plot of the corresponding two-class Softmax cost. The contours of this cost function are plotted in the bottom-middle panel of Figure 9.16. Once

again, their extremely long and narrow shape suggests that gradient descent will struggle immensely in determining the global minimizer (located inside the smallest contour shown).

We confirm this intuition by making a run of 100 steps of gradient descent beginning at the point $\mathbf{w} = [20\ 30]^T$ and using a steplength parameter of $\alpha = 1$. As shown in the bottom-middle panel, these steps (colored from green to red as gradient descent progresses) zig-zag considerably. Moreover we can see that at the end of the run we are still a long way from the minimizer of the cost function, resulting in a very poor fit to the dataset (shown in blue in the bottom-left panel of the figure).

In the bottom-right panel of Figure 9.16 we show the result of repeating this experiment using standard normalized input. Here we use the same initialization, but only 25 steps, and (since the contours of the associated cost are so much more circular) a larger steplength value $\alpha = 10$. This rather large value would have caused the first run of gradient descent to diverge. Nonetheless even with so few steps we are able to find a good approximation to the global minimizer. We plot the corresponding tanh model (in red) in the bottom-left panel of the figure, which fits the data much better than the result of the first run (shown in blue).

9.3.2 Standard normalized model

Once a general model taking in N-dimensional standard normalized input has been properly tuned, and the optimal parameters $w_0^\star, w_1^\star, ..., w_N^\star$ have been determined, in order to evaluate any new point we must standard normalize each of its input features using the same statistics we computed on the training data.

9.4 Imputing Missing Values in a Dataset

Real-world data can contain *missing values* for various reasons including human error in collection, storage issues, faulty sensors, etc. Generally speaking, if a supervised learning data point is missing its *output* value there is little we can do to salvage it, and usually such a corrupted data point is thrown away in practice. Likewise, if a large number of input values of a data point are missing it is best discarded. However, a data point missing just a handful of its input features can be salvaged by filling in missing input features with appropriate values. This process, often called *imputation*, is particularly useful when the data is scarce.

9.4.1 Mean imputation

Suppose as usual that we have a set of P inputs, each of which is N-dimensional, and that the set Ω_n contains the indices of all data points whose nth input feature

value is missing. In other words, for all $j \in \Omega_n$ the value of $x_{j,n}$ is missing in our input data. An intuitive value to fill for all missing entries along the nth input feature is the simple average (or expected value) of the dataset along this dimension. That is, for all $j \in \Omega_n$ we set $x_{j,n} = \mu_n$, where

$$\mu_n = \frac{1}{P - |\Omega_n|} \sum_{j \notin \Omega_n} x_{j,n} \tag{9.4}$$

and where $|\Omega_n|$ denotes the number of elements in Ω_n. This is often called *mean imputation*. Notice, after mean imputation, that the mean value of the entire nth feature of the input remains unchanged, since

$$\frac{1}{P}\sum_{p=1}^{P} x_{p,n} = \frac{1}{P}\sum_{j \notin \Omega_n} x_{j,n} + \frac{1}{P}\sum_{j \in \Omega_n} x_{j,n} = \frac{1}{P}\left(P - |\Omega_n|\right)\mu_n + \frac{1}{P}\sum_{j \in \Omega_n} \mu_n = \mu_n. \tag{9.5}$$

Therefore one consequence of imputing missing values of a dataset using the mean along each input dimension is that when we standard normalize this dataset (as detailed in the previous section), all values imputed with the mean become exactly zero. This is illustrated for a simple example in Figure 9.17. Thus any parameter or weight in the model that touches such a mean-imputed entry is completely nullified numerically. This is desirable given that such values were missing in the first place.

9.5 Feature Scaling via PCA-Sphering

In the Section 9.3 we saw how *feature scaling* via *standard normalization* significantly improves the topology of a machine learning cost functions, enabling much more rapid minimization via first-order methods like gradient descent (see Section 3.5). In this Section we describe how Principal Component Analysis (PCA) (detailed in Section 8.3) can be used to perform a more advanced form of input normalization, commonly called *PCA-sphering* (or sometimes *whitening*).

9.5.1 PCA-sphering: the big picture

PCA-sphering takes the idea of standard normalization described in Section 9.3 one step further by using PCA (see Section 8.3) to rotate the mean-centered dataset, so that its largest orthogonal directions of variance align with the coordinate axes, prior to scaling each input by its standard deviation. This simple adjustment typically allows us to better compactify the data, and more importantly results in a cost function whose contours are even more "circular" than that provided by standard normalization (indeed PCA-sphering regression data makes the contours of a Least Squares cost for linear regression *perfectly circular* – see Exercise 9.6 for further details). This is illustrated in Figure 9.18 where we

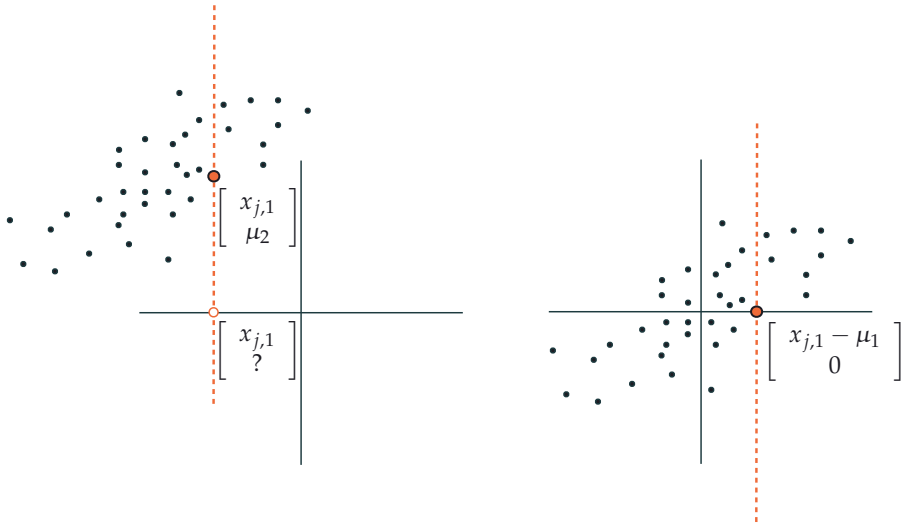

Figure 9.17 (left panel) The input of a prototypical $N = 2$ dimensional dataset where a single point $\mathbf{x}_j$, drawn as a *hollow* red dot, is missing its second entry. The mean-imputed version of this point is then shown as a *filled-in* red point. (right panel) By mean-centering such a dataset (which is the first step of standard normalization) the mean-imputed feature of $\mathbf{x}_j$ becomes exactly equal to zero.

compare pictorially the effect of standard normalization and PCA-sphering on a prototypical $N = 2$ dimensional input dataset, as well as how each scheme changes the topology of the associated cost function. As outlined in Section 9.3.1, gradient descent schemes work far better the more circular we make the contours of a cost function.

The trade-off, of course, is that while PCA-sphering makes first-order optimization considerably easier once it is enacted, we must pay an extra up-front cost of performing PCA on the data, making PCA-sphering more computationally expensive than the standard normalization procedure. Whether or not this extra up-front cost is worth it can vary in practice from problem to problem, however often times it is. This is particularly true when employing first-order optimization (see, e.g., Exercise 9.7) since – as summarized in Figure 9.15 – the more circular we make the contours of a cost function the easier gradient-based optimization methods can minimize them properly.

9.5.2 PCA-sphering: the technical details

More formally we can express the standard normalization scheme applied to a single data point $\mathbf{x}_p$ in two steps as

$$\begin{aligned}
&\textbf{mean-center:} \quad \text{for each } n \text{ replace } x_{p,n} \longleftarrow \left(x_{p,n} - \mu_n\right) \\
&\textbf{std-scale:} \quad \text{for each } n \text{ replace } x_{p,n} \longleftarrow \frac{x_{p,n}}{\sigma_n}
\end{aligned}$$

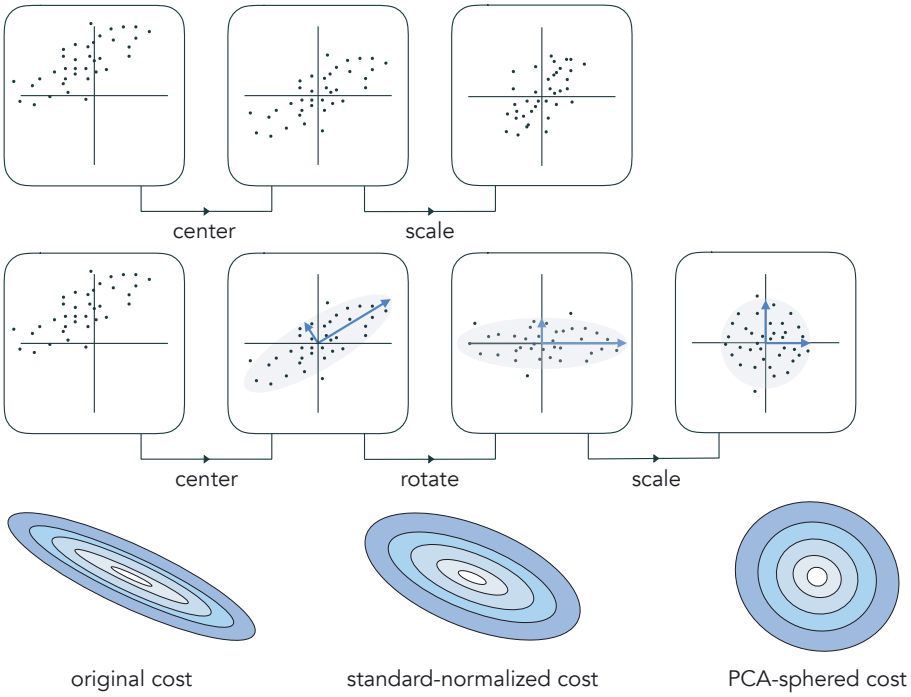

Figure 9.18 The standard normalization procedure (top row) compared to PCA-sphering (middle row) on a generic set of input data. With PCA- sphering we insert a single extra step into the standard normalization pipeline *in between* mean-centering and scaling by standard deviations, where we rotate the data using PCA. This not only shrinks the space consumed by the data more than standard normalization (compare top right and middle right panels), it also tends to make any associated cost function considerably easier to minimize by better tempering its contours, making them more circular (bottom row panels).

where "std" is short for "standard deviation," and the mean and standard deviation are defined as $\mu_n = \frac{1}{P}\sum_{p=1}^{P} x_{p,n}$ and $\sigma_n = \sqrt{\frac{1}{P}\sum_{p=1}^{P} x_{p,n}^2}$ for each n. Note here too that the notation $a \longleftarrow b$ denotes the replacement of quantity a with quantity b.

Denoting by $\mathbf{X}$ the $N \times P$ matrix of input whose pth column contains the input data point $\mathbf{x}_p$, and by $\mathbf{V}$ the set of eigenvectors of the data covariance matrix $\frac{1}{P}\mathbf{X}\mathbf{X}^T = \mathbf{V}\mathbf{D}\mathbf{V}^T$ (as detailed in Section 8.3.3), we can then write the PCA-sphering scheme applied to the same data point $\mathbf{x}_p$ in three highly related steps noting importantly that the nth eigenvalue d_n (the nth diagonal entry of the matrix $\mathbf{D}$) is precisely equal[3] to the variance σ_n^2, and equivalently $\sqrt{d_n} = \sigma_n$

[3] The Rayleigh quotient definition (see, e.g., Exercise 3.3) of the nth eigenvalue d_n of the data covariance matrix states that numerically speaking $d_n = \frac{1}{P}\mathbf{v}_n\mathbf{X}\mathbf{X}^T\mathbf{v}_n$, where $\mathbf{v}_n$ is the nth and

mean-center: for each n replace $x_{p,n} \longleftarrow \left(x_{p,n} - \mu_n\right)$
PCA-rotate: transform $\mathbf{x}_p \longleftarrow \mathbf{V}^T\mathbf{x}_p$
std-scale: for each n replace $x_{p,n} \longleftarrow \frac{x_{p,n}}{\sqrt{d_n}}$.

Denoting $\mathbf{D}^{-\frac{1}{2}}$ as the diagonal matrix whose nth diagonal element is $\frac{1}{\sqrt{d_n}}$, we can then (after mean-centering the data) express steps 2 and 3 of the PCA-sphering algorithm recipe above quite compactly as

$$\mathbf{X} \longleftarrow \mathbf{D}^{-\frac{1}{2}}\mathbf{V}^T\mathbf{X}. \tag{9.6}$$

9.5.3 PCA-sphered model

Once a general model taking in N-dimensional PCA-sphered input has been properly tuned, and the optimal parameters have been determined, in order to evaluate any new point we must PCA-sphere new input features using the same statistics we computed on the training data.

9.6 Feature Selection via Boosting

In Chapters 5 through 7 we saw how the fully tuned linear model for *supervised learning* generally takes the form

$$\text{model}\left(\mathbf{x}, \mathbf{w}^\star\right) = \mathring{\mathbf{x}}^T\mathbf{w}^\star = w_0^\star + x_1 w_1^\star + x_2 w_2^\star + \cdots + x_N w_N^\star \tag{9.7}$$

where the weights $w_0^\star$, $w_1^\star$ ..., $w_N^\star$ are optimally tuned via the minimization of an appropriate cost function. Understanding the intricate connections the input features of a dataset have with its corresponding output naturally boils down to human analysis of these $N + 1$ tuned weights. However, it is not always straightforward to derive meaning from such a sequence of $N + 1$ numbers, exacerbated by the fact that the idea of *human interpretability* quickly becomes untenable as the input dimension N grows. To ameliorate this issue we can use what is called a *feature selection* technique.

In this section we discuss one popular way of performing feature selection, called *boosting* or *forward stage-wise selection*. Boosting is a bottom-up approach to feature selection wherein we gradually build up our model one feature at a time by training a supervised learner *sequentially*, one weight at a time. Doing this gives human interpreters an easier way to gage the importance of individual

corresponding eigenvector. Now in terms of our PCA-transformed data this is equivalently written as $d_n = \frac{1}{P}\left\|\mathbf{v}_n^T\mathbf{X}\right\|_2^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_{p,n}\right)^2 = \sigma_n^2$ or in other words, it is the *variance* along the nth axis of the PCA-transformed data.

features, and likewise lets them more easily derive insight about a particular phenomenon.

9.6.1 Boosting based feature selection

In tuning a model's weights one at a time we do not want to tune them in *any* order (e.g., at random) as this will not aid human interpretation. Instead we want to tune them starting with the most important (feature-touching) weight, then tune the second most important (feature-touching) weight, then the third, and so forth. Here by "importance" we mean how each input feature contributes to the final supervised learning model as determined by its associated weight or, in other words, how each contributes to minimizing the corresponding cost (or associated metric) as much as possible.

The boosting process is started with a model, which we will denote as model_0, that consists of the bias w_0 alone

$$\text{model}_0\left(\mathbf{x}, \mathbf{w}\right) = w_0. \tag{9.8}$$

We then tune the bias parameter w_0 by minimizing an appropriate cost (depending on whether we are solving a regression or classification problem) over this variable *alone*. For example, if we are performing regression employing the Least Squares cost we would minimize

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0\left(\mathbf{x}, \mathbf{w}\right) - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0 - y_p\right)^2 \tag{9.9}$$

which gives the optimal value for our bias $w_0 \longleftarrow w_0^\star$. Plugging this learned weight into our starting model in Equation (9.8) gives

$$\text{model}_0\left(\mathbf{x}, \mathbf{w}\right) = w_0^\star. \tag{9.10}$$

Next, at the *first round* of boosting, in order to determine the most important feature-touching weight (among $w_1, w_2, \ldots, w_N$) we *try out each one* by minimizing an appropriate cost over each individually, having already set the bias optimally. For example, in the case of Least Squares regression the nth of these N subproblems takes the form

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0\left(\mathbf{x}_p, \mathbf{w}\right) + w_n x_{n,p} - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0^\star + x_{n,p} w_n - y_p\right)^2. \tag{9.11}$$

Notice, since the bias weight has already been set we only tune the weight w_n in the nth subproblem.

The feature-touching weight that produces the *smallest* cost (or *best* metric value in general) from these N subproblems corresponds to the individual feature that helps best explain the relationship between the input and output of our dataset. It can therefore be interpreted as the most important feature-touching weight we learn. Denoting this weight as w_{s_1}, we then *fix* it at its optimally determined value $w_{s_1}^\star$ (discarding all other weights tuned in each of these subproblems) and update our `model` accordingly. Our updated `model` at the end of the first round of boosting, which we call model_1, is a sum of our optimal bias and this newly determined optimal feature-touching weight

$$\text{model}_1\left(\mathbf{x}, \mathbf{w}\right) = \text{model}_0\left(\mathbf{x}, \mathbf{w}\right) + x_{s_1} w_{s_1}^\star = w_0^\star + x_{s_1} w_{s_1}^\star. \tag{9.12}$$

This boosting process is then repeated sequentially. In general at the mth round of boosting the mth most important feature-touching weight is determined following the same pattern. At the beginning of the mth round (where $m > 1$) we have already determined the optimal setting of our bias as well as the top $m-1$ most important feature-touching weights, and our `model` takes the form

$$\text{model}_{m-1}\left(\mathbf{x}, \mathbf{w}\right) = w_1^\star + x_{s_1} w_{s_1}^\star + \cdots + x_{s_{m-1}} w_{s_{m-1}}^\star. \tag{9.13}$$

We then set up and solve $N-m+1$ subproblems, one for each feature-touching weight we have not yet chosen. For example, in the case of Least Squares regression the nth of these takes the form

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_{m-1}\left(\mathbf{x}_p, \mathbf{w}\right) + w_n x_{n,p} - y_p\right)^2 \tag{9.14}$$

where again in each case we only tune the individual weight w_n. The feature-touching weight that produces the smallest cost value corresponds to the mth most important feature. Denoting this weight w_{s_m} we then fix it at its optimal value $w_{s_m}^\star$, and add its contribution to the running model as

$$\text{model}_m\left(\mathbf{x}, \mathbf{w}\right) = \text{model}_{m-1}\left(\mathbf{x}, \mathbf{w}\right) + x_{s_m} w_{s_m}^\star. \tag{9.15}$$

Given that we have N input features we can continue until $m \leq N$ or when some maximum number of iterations is reached. Note too that, after M rounds of boosting, we have constructed a sequence of models $\{\text{models}_m\}_{m=0}^{M}$. This method of recursive model building via adding features one at a time and tuning only the added feature's weight (keeping all others fixed at their previously tuned values) is referred to as *boosting*.

9.6.2 Selecting the right number of features via boosting

Recall that feature selection is done primarily for the purposes of *human interpretation*. Therefore a benchmark value for the number of features M to select can be chosen based on the desire to explore a dataset, and the procedure halted once this number of rounds have completed. One can also halt exploration when adding additional features to the model results in very little decrease in the cost, as most of the correlation between inputs and outputs has already been explained. Finally, M can be chosen entirely based on the sample statistics of the dataset via a procedure known as *cross-validation*, which we discuss in Chapter 11.

Regardless of how we select the value of M, it is important to standard normalize the input data (as detailed in Section 9.3) before we begin the boosting procedure for feature selection. In addition to the optimization speed-up advantage, standard normalization also allows us to fairly compare each input feature's contribution by examining their tuned weight values.

Example 9.6 Exploring predictors of housing prices via boosting

The result of running $M = 5$ rounds of boosting on the Boston Housing dataset (first introduced in Example 5.5), using the Least Squares cost and Newton's method optimizer, is visualized in the top panel of Figure 9.19. This special kind of cost function history shows each weight/feature index added to the model at each round of boosting (starting with the bias which has index 0). As can be seen on the horizontal axis, the first two most contributing features found via boosting are LSTAT (feature 13) and the average number of rooms per dwelling (feature 6). Examining the histogram of model weights in the bottom panel of Figure 9.19, we can see (unsurprisingly) that the LSTAT weight, having a negative value, is negatively correlated with the output (i.e., home price) while the weight associated with the average number of rooms feature is positively correlated with the output.

Example 9.7 Exploring predictors of credit risk via boosting

In Figure 9.20 we show the results of running the boosting procedure (using a Softmax cost and Newton's method optimizer) on the German credit score dataset first introduced in Example 6.11, which consists of $P = 1000$ samples, each a set of statistics extracted from loan applications to a German bank. The $N = 20$ dimensional input features in this dataset include: the individual's current account balance with the bank (feature 1), the duration (in months) of previous credit with the bank (feature 2), the payment status of any prior credit taken out with the bank (feature 3), and the current value of their savings/stocks (feature 6). These are precisely the top four features found via boosting, most of which are positively correlated with an individual being a good credit risk (as shown in the bottom panel of Figure 9.20).

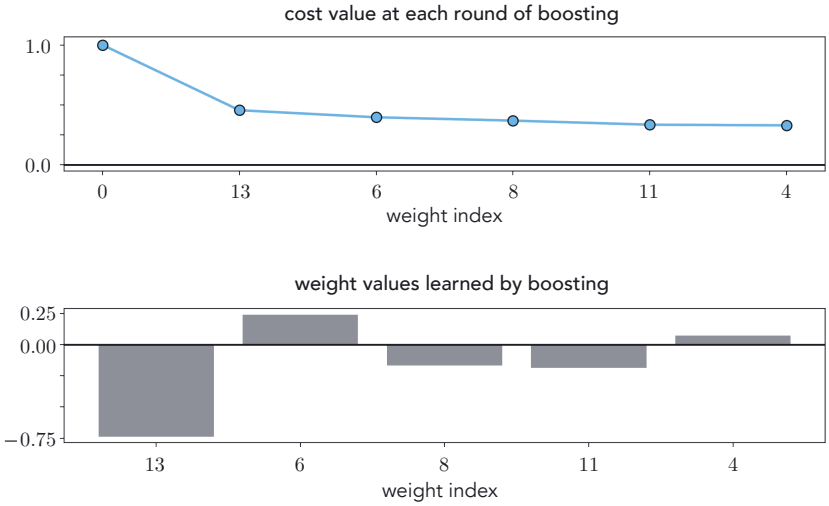

Figure 9.19 Figure associated with Example 9.6. See text for details.

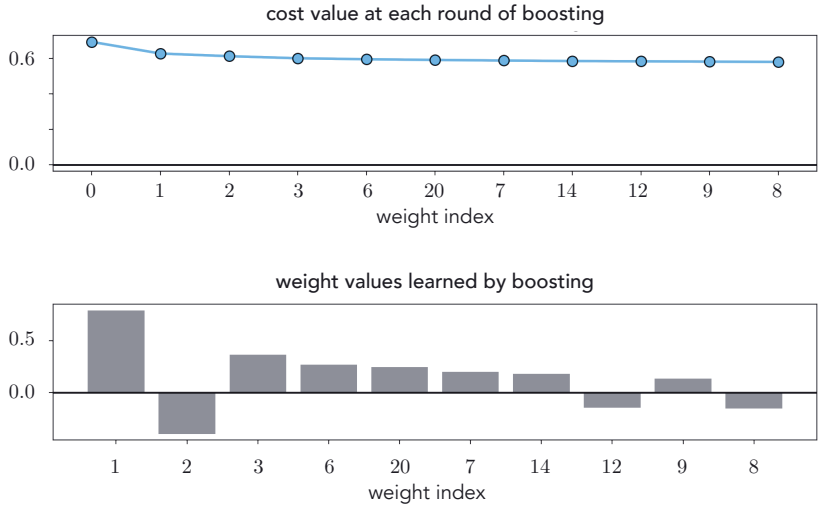

Figure 9.20 Figure associated with Example 9.7. See text for details.

9.6.3 On the efficiency of boosting as a greedy algorithm

Boosting is essentially a *greedy* algorithm, where at each stage we choose the next most important (feature-touching) weight and tune it properly by solving a set of respective subproblems. While each round of boosting demands we

solve a number of subproblems, each one is a minimization with respect to *only a single weight* and is therefore cheap to solve virtually regardless of the local optimization scheme used. This makes boosting a computationally effective approach to feature selection, and allows it to scale to datasets with a large number of input dimensions N. A weakness inherent in doing this, however, is that in determining feature-touching weights one at a time, interactions between features/weights can be potentially missed.

To capture these potentially missed interactions one might naturally extend the boosting idea and try to add a group of R feature-touching weights at each round instead of just one. However, a quick calculation shows that this idea would quickly fail to scale. In order to determine the first best group of R feature-touching weights at the first stage of this approach we would need to try out every combination of R weights by solving a subproblem for each. The issue here is that there are combinatorially many subgroups of size R, more precisely $\binom{N}{R} = \frac{N!}{R!(N-R)!}$ of them. This is far too many problems to solve in practice, even for small-to-moderate values of N and R (e.g., $\binom{100}{5} = 75,287,520$).

9.6.4 The residual perspective on boosting regression

Recall the nth subproblem in the mth stage of boosting in the case of Least Squares regression aims at minimizing

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_{m-1}\left(\mathbf{x}_p, \mathbf{w}\right) + w_n x_{n,p} - y_p\right)^2. \tag{9.16}$$

If we rearrange the terms in each summand and denote

$$r_p^m = \left(y_p - \text{model}_{m-1}\left(\mathbf{x}_p, \mathbf{w}\right)\right) \tag{9.17}$$

then we can write the Least Squares cost function in Equation (9.16) equivalently as

$$\frac{1}{P}\sum_{p=1}^{P}\left(w_n x_{n,p} - r_p^m\right)^2 \tag{9.18}$$

where the term r_p^m on the right-hand side of each summand is *fixed* since w_n is the only weight being tuned here. This r_p^m term is the *residual* of the original output y_p after the contribution of model model_m has been subtracted off. We can then think about each round of boosting as determining the next feature that best correlates with the residual from the previous round.

9.7 Feature Selection via Regularization

With the boosting approach to feature selection discussed in the previous section we took a greedy "bottom-up" approach to feature selection: we began by tuning the bias and then added new features to our model one at a time. In this section we introduce a complementary approach to feature selection, called *regularization*. Instead of building up a model starting at the bottom, with regularization we take a "top-down" view and start off with a complete model that includes every one of our input features, and then we gradually remove input features of inferior importance. We do this by adding a second function to our cost (called a *regularizer*) that penalizes all weights, forcing our model to shrink the weight values associated with less-important input features.

9.7.1 Regularization using weight vector norms

The simple linear combination of the cost function g and an auxiliary function h

$$f(\mathbf{w}) = g(\mathbf{w}) + \lambda\, h(\mathbf{w}) \tag{9.19}$$

is often referred to as *regularization* in the parlance of machine learning, with the function h called a *regularizer* and the parameter $\lambda \geq 0$ called a *regularization* or *penalty* parameter.

When $\lambda = 0$ the linear combination in Equation (9.19) reduces to the original cost function g. As we increase λ the two functions g and h start to *compete for dominance*, with the linear combination taking on properties of both functions. As we set λ to a larger and larger value the function h dominates the combination, eventually completely drowning out g, and we end up essentially with a scaled version of the regularizer h.

In machine learning applications it is very common to use a vector or matrix norm of model parameters (see Section C.5) as a regularizer h, with different norms used to produce different effects in the learning of machine learning models. In what follows we outline how several common vector norms affect the minimization of a generic cost function g.

Example 9.8 Regularization using the ℓ_0 norm

The ℓ_0 vector norm of $\mathbf{w}$, written as $\|\mathbf{w}\|_0$, measures its length or magnitude as

$$\|\mathbf{w}\|_0 = \text{number of nonzero entries of } \mathbf{w}. \tag{9.20}$$

By regularizing a cost g using this regularizer (i.e., $f(\mathbf{w}) = g(\mathbf{w}) + \lambda\, \|\mathbf{w}\|_0$), we penalize the regularized cost f for every nonzero entry of $\mathbf{w}$ since every such

entry adds one unit to $\|\mathbf{w}\|_0$. Conversely, then, in minimizing f, the two functions g and $\|\mathbf{w}\|_0$ compete for dominance with g wanting $\mathbf{w}$ to be resolved as a point near its minimizer, while the regularizer $\|\mathbf{w}\|_0$ aims to determine a $\mathbf{w}$ that has as few nonzero elements as possible, or in other words, a weight vector $\mathbf{w}$ that is very *sparse*.

Example 9.9 Regularization using the ℓ_1 norm

The ℓ_1 vector norm, written as $\|\mathbf{w}\|_1$, measures the magnitude of $\mathbf{w}$ as

$$\|\mathbf{w}\|_1 = \sum_{n=0}^{N} |w_n| \,. \tag{9.21}$$

By regularizing a cost g using this regularizer (i.e., $f(\mathbf{w}) = g(\mathbf{w}) + \lambda \|\mathbf{w}\|_1$), we penalize the regularized cost based on the sum of the absolute value of the entries of $\mathbf{w}$.

Conversely, then, in minimizing this sum, the two functions g and $\|\mathbf{w}\|_1$ compete for dominance with g wanting $\mathbf{w}$ to be resolved as a point near its minimizer, while the regularizer $\|\mathbf{w}\|_1$ aims to determine a $\mathbf{w}$ that is small in terms of the absolute value of each of its components, but also because the ℓ_1 norm is closely related to the ℓ_0 norm (see Section C.5), one that has few nonzero entries and is therefore *sparse*.

Example 9.10 Regularization using the ℓ_2 norm

The ℓ_2 vector norm of $\mathbf{w}$, written as $\|\mathbf{w}\|_2$, measures its magnitude as

$$\|\mathbf{w}\|_2 = \sqrt{\sum_{n=0}^{N} w_n^2}. \tag{9.22}$$

By regularizing a cost g using this regularizer (i.e., $f(\mathbf{w}) = g(\mathbf{w}) + \lambda \|\mathbf{w}\|_2$), we penalize the regularized cost based on the sum of squares of the entries of $\mathbf{w}$.

Conversely in minimizing this sum, the two functions g and $\|\mathbf{w}\|_2$ compete for dominance with g wanting $\mathbf{w}$ to be resolved as a point near its minimizer, while the regularizer $\|\mathbf{w}\|_2$ aims to determine a $\mathbf{w}$ that is small in the sense that all of its entries have a small *squared* value.

We have so far seen a number of instances of ℓ_2 regularization, e.g., in the context of Softmax classification in Section 6.4.6 and support vector machines in Section 6.5.4.

9.7.2 Feature selection via ℓ_1 regularization

In the context of machine learning by inducing the discovery of sparse weight vectors, the ℓ_0 and ℓ_1 regularizers help uncover the identity of a dataset's *most important features*. This is because when we employ such norms as regularizer

we force the recovered model weights to be rather *sparse* (provided we have set λ appropriately), with only those weights associated with a model's most important features remaining. This makes either norm (at least in principle) quite appropriate for the task of feature selection.

Of the two sparsity-inducing norms described in Examples 9.8 and 9.9, the ℓ_0 norm (while promoting sparsity directly and to the greatest degree) is the most challenging to employ due to its *discontinuous* nature, making the minimization of an ℓ_0 regularized cost function quite difficult. While the ℓ_1 norm induces sparsity to less of a degree, it is both *convex* and *differentiable* (almost everywhere), making the use of first-order methods possible for its minimization. Because of this practical advantage the ℓ_1 norm is by far the more commonly used regularizer for feature selection in practice.

Finally, remember as detailed in the previous section that when performing feature selection we are only interested in determining the importance of *feature-touching* weights $w_1, w_2, ..., w_N$, thus we only need regularize them (and not the bias weight w_0). This means that our regularization will more specifically take the form

$$f(\mathbf{w}) = g(\mathbf{w}) + \lambda \sum_{n=1}^{N} |w_n| . \tag{9.23}$$

Using our individual notation for the bias and feature-touching weights (used for instance in Section 6.4.6)

$$\text{(bias): } b = w_0 \quad \text{(feature-touching weights): } \boldsymbol{\omega} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} \tag{9.24}$$

we can write this general ℓ_1 regularized cost function equivalently as

$$f(b, \boldsymbol{\omega}) = g(b, \boldsymbol{\omega}) + \lambda \left\| \boldsymbol{\omega} \right\|_1 . \tag{9.25}$$

9.7.3 Selecting the right regularization parameter

Because feature selection is done for the purposes of *human interpretation*, the value of λ can be set based on several factors. A benchmark value for λ can be chosen based on the desire to explore a dataset, finding a value that provides sufficient sparsity while retaining a low cost value. The value of λ can also be chosen entirely based on the sample statistics of the dataset via a procedure known as *cross-validation*, which we discuss in Chapter 11.

Just like boosting and regardless of how we select λ, it is important to standard

normalize the input data so that we can fairly compare each input feature's contribution by examining their tuned weight values recovered from minimization of the regularized cost function.

Example 9.11 Exploring predictors of housing prices via regularization

In this example we form an ℓ_1 regularized Least Squares cost using the Boston Housing dataset (first introduced in Example 5.5 and used in the context of boosting-based feature selection in Example 9.6) and examine 50 evenly spaced values for λ in the range $[0, 130]$. For each value of λ in this range, starting from 0 (as illustrated in the top panel of Figure 9.21), we run gradient descent with a fixed number of steps and steplength parameter to minimize the regularized cost. By the time λ is set to the largest value in the range, three major weights remain recovered by the parameter tuning (as illustrated in the bottom panel of Figure 9.21), corresponding to feature 6, feature 13, and feature 11. The first two features (i.e., feature 6 and feature 13) were also determined to be important via boosting in Example 9.6.

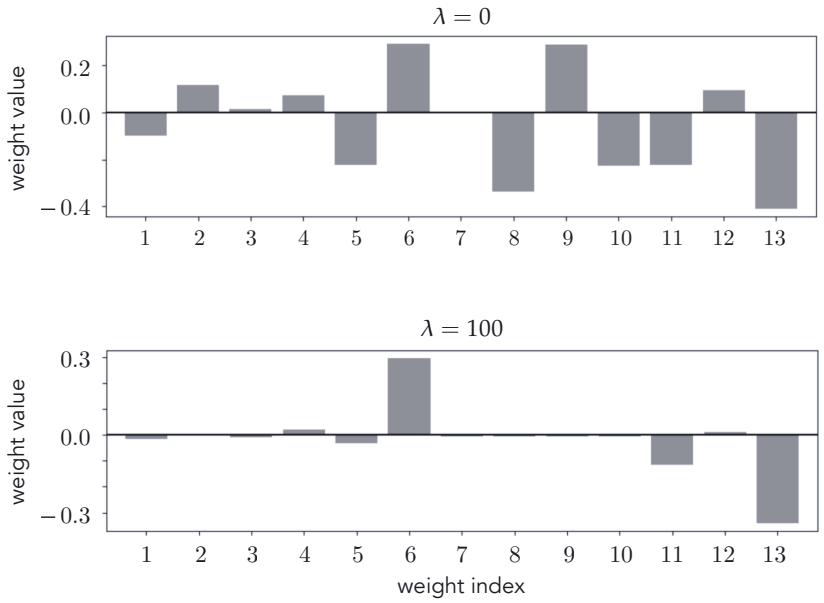

Figure 9.21 Figure associated with Example 9.11. See text for details.

Example 9.12 Exploring predictors of credit risk via regularization

In this example we minimize the ℓ_1 regularized two-class Softmax cost over the German credit dataset (first introduced in Example 9.20), using 50 evenly spaced values for λ in the range $[0, 130]$. For each value of λ in this range, starting from

0 (as illustrated in the top panel of Figure 9.22) we run gradient descent with a fixed number of steps and steplength parameter to minimize the regularized cost.

By the time $\lambda \approx 40$, four major weights remain, corresponding to features 1, 2, 3, and 6 (as illustrated in the bottom panel of Figure 9.22). Note that these features were also determined to be important via boosting in Example 9.7.

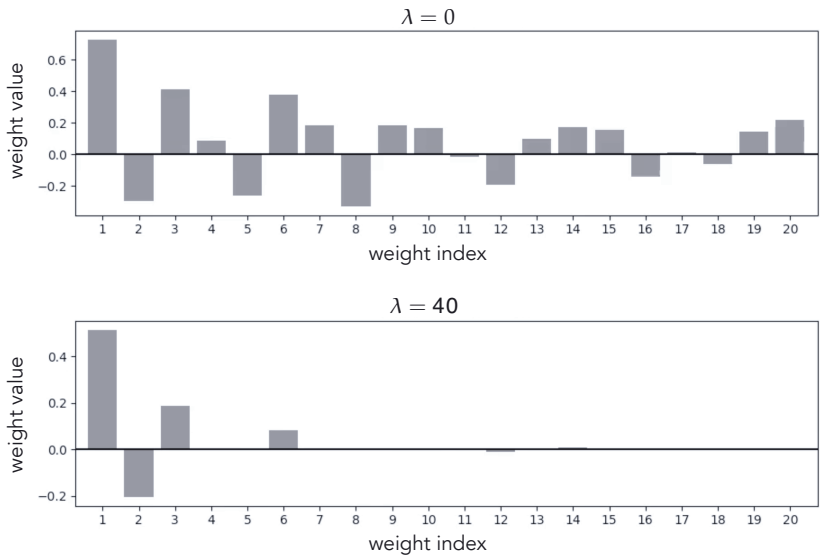

Figure 9.22 Figure associated with Example 9.12. See text for details.

9.7.4 Comparing regularization and boosting

While boosting is an efficient greedy scheme, the regularization idea detailed in this section can be computationally intensive to perform since for each value of λ tried, a full run of local optimization must be completed. On the other hand, while boosting is a "bottom-up" approach that identifies individual features one at a time, regularization takes a more "top-down" approach and identifies important features all at once. In principle this allows regularization to uncover groups of important features correlated in such an interconnected way with the output that may be missed by boosting.

9.8 Conclusion

In this chapter we reviewed fundamental techniques for *feature engineering* and *feature selection*.

We began by discussing feature engineering techniques, which are used as data preprocessing steps for virtually all machine learning problems, and which we will make use of extensively in the remainder of the text. In Section 9.2 we detailed histogram features, which neatly summarize the content in data and can be designed for virtually any data modality. In Sections 9.3–9.5 various input scaling techniques were described, including *standard normalization* and *PCA-sphering*. These methods standardize input data and improve the topology of machine learning cost functions, making them significantly easier to minimize (particularly using the first-order methods described in Chapter 3). Finally in Sections 9.6 and 9.7 we described two complementary approaches to feature selection – *boosting* and *regularization* – which enable straightforward human analysis of the strength of individual features included in a trained machine learning model.

9.9 Exercises

† The data required to complete the following exercises can be downloaded from the text's github repository at `github.com/jermwatt/machine_learning_refined`

9.1 Spam email

Repeat the experiment described in Example 9.2 using any local optimization scheme you desire and the two-class Softmax cost. Make sure to produce a plot like the one shown in Figure 9.5 to compare the results of using each combination of features.

9.2 MNIST classification: pixels versus edge-based features

Repeat the experiment outlined in Example 9.3 and create a pair of cost function/misclassification history plots like the ones shown in Figure 9.11. Your results may vary slightly from those reported in the example depending on the details of your implementation.

9.3 Student debt

Produce two contour plots of the Least Squares cost function over the student debt dataset [2] shown in Figure 1.8, as well as its standard normalized version. Compare the overall topology of each contour plot and describe why the plot associated with the standard normalized data will be far easier to optimize via gradient descent. Indeed, fitting to the original dataset using gradient descent is almost impossible here. Minimize the Least Squares cost over the standard normalized version of the data using gradient descent and reproduce the plot shown in Figure 1.8.

9.4 Least Squares and perfectly circular contours: part 1

In Example 9.4 we saw how the contour plot of the Least Squares cost over an $N = 1$ regression dataset changed from highly elliptical to *perfectly* circular after the data was standard normalized (see Figure 9.16). Show that the contour plot of a Least Squares cost over standard normalized data will always be perfectly circular when $N = 1$. Then describe why this does *not* necessarily happen when $N > 1$.

9.5 Breast cancer dataset

Perform linear two-class classification on a breast cancer dataset [43] consisting of $P = 569$ data points using an appropriate cost function and local optimizer. Fill in any missing values in the data using mean-imputation and report the best misclassification rate you were able to achieve. Compare how quickly you can minimize the your cost function over the original and standard normalized data using the same parameters (i.e., the same number of steps, steplength parameter, etc.).

9.6 PCA-sphering and the Least Squares cost for linear regression

The Least Squares cost for linear regression presents the ideal example of how PCA-sphering a dataset positively effects the topology of a machine learning cost function (making it considerably easier to minimize properly). This is because – as we show formally below – PCA-sphering the input of a regression dataset leads to a Least Squares cost for linear regression that has *perfectly circular contours* and is thus very easy to minimize properly. While PCA-sphering does not improve all cost functions to such a positive degree, this example is still indicative of the effect PCA-sphering has on improving the topology of machine learning cost functions in general.

To see how PCA-sphering perfectly tempers the contours of the Least Squares cost for linear regression first note – as detailed in Section 5.9.1 – that the Least Squares cost is always (regardless of the dataset used) a convex quadratic function of the general form $g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\,\mathbf{w}$, where – in particular – $\mathbf{C} = \frac{1}{P}\sum_{p=1}^{P}\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T$. If the input of a regression dataset is PCA-sphered, then the lower $N \times N$ submatrix of $\mathbf{C}$ is given as $\frac{1}{P}\mathbf{S}\mathbf{S}^T$, where $\mathbf{S}$ is defined in Equation (9.6). However, because of the very way $\mathbf{S}$ is defined we have that $\frac{1}{P}\mathbf{S}\,\mathbf{S}^T = \frac{1}{P}\mathbf{I}_{N\times N}$, where $\mathbf{I}_{N\times N}$ is the $N \times N$ identity matrix, and thus, in general, that $\mathbf{C} = \frac{1}{P}\mathbf{I}_{(N+1)\times(N+1)}$. In other words, the Least Squares cost over PCA-sphered input is a convex quadratic with all eigenvalues equal to 1, implying that it is a quadratic with perfectly circular contours (see Section 4.2.2).

9.7 Comparing standard normalization to PCA-sphering on MNIST

Compare a run of ten gradient descent steps using the multi-class Softmax cost over $50,000$ random digits from the MNIST dataset (introduced in Example

7.10), a standard normalized version, and a PCA-sphered version of the data. For each run use the largest fixed steplength of the form $\alpha = 10^{\gamma}$ for γ an integer you find to produce descent. Create a plot comparing the progress of each run in terms of the cost function and number of misclassifications. Additionally, make sure your initialization of each run is rather small, particularly the first run where you apply no normalization at all to the input as each raw input point of this dataset is large in magnitude. In the case of the raw data initializing at a point too far away from the origin can easily cause numerical overflow, producing `nan` or `inf` values, ruining the rest of the corresponding local optimization run.

9.8 Least Squares and perfectly circular contours: part 2

In Exercise 9.4 we saw how PCA-sphering input data reshapes the topology of the Least Squares cost so that its contours become *perfectly circular*. Explain how this makes the minimization of such a PCA-sphered Least Squares cost extremely easy. In particular explain how – regardless of the dataset – such a cost can be perfectly minimized using one "simplified" Newton step – as described in Section A.8.1 – where we ignore the off-diagonal elements of the Hessian matrix when taking a Newton step.

9.9 Exploring predictors of housing prices

Implement the boosting procedure detailed in Section 9.6.1 for linear regression employing the Least Squares cost, and repeat the experiment described in Example 9.6. You need not reproduce the visualizations in Figure 9.19, but make sure you are able to reach the similar conclusions to those outlined in the example.

9.10 Predicting Miles-per-Gallon in automobiles

Run $M = 6$ rounds of boosting on the automobile MPG dataset introduced in Example 5.6, employing the Least Squares cost function, to perform feature selection. Provide an interpretation of the three most important features you find, and how they correlate with the output.

9.11 Studying important predictors of credit risk

Implement the boosting procedure detailed in Section 9.6.1 for linear classification employing the two-class Softmax cost, and repeat the experiment described in Example 9.7. You need not reproduce the visualizations in Figure 9.20, but make sure you are able to reach the same conclusions outlined in the example.

9.12 Exploring predictors of housing prices

Implement the regularization procedure detailed in Section 9.7 for linear regression employing the Least Squares cost, and repeat the experiment described

in Example 9.11. You need not reproduce the visualizations in Figure 9.21, but make sure you are able to reach the same conclusions outlined in the example.

9.13 Studying important predictors of credit risk

Implement the regularization procedure detailed in Section 9.7 for linear classification employing two-class Softmax cost, and repeat the experiment described in Example 9.12. You need not reproduce the visualizations in Figure 9.22, but make sure you are able to reach the same conclusions outlined in the example.