Resources and Extras 15

This chapter contains some mathematical material that you will likely have seen, but some may not have stayed with you. I have also relegated the detailed discussion of how one splits a node in a decision tree to this chapter.

15.1 Useful Material About Matrices

Terminology:

- A matrix $\mathcal{M}$ is **symmetric** if $\mathcal{M} = \mathcal{M}^T$. A symmetric matrix is necessarily square.
- We write $\mathcal{I}$ for the identity matrix.
- A matrix is **diagonal** if the only non-zero elements appear on the diagonal. A diagonal matrix is necessarily symmetric.
- A symmetric matrix is **positive semidefinite** if, for any $\mathbf{x}$ such that $\mathbf{x}^T\mathbf{x} > 0$ (i.e. this vector has at least one non-zero component), we have $\mathbf{x}^T\mathcal{M}\mathbf{x} \geq 0$.
- A symmetric matrix is **positive definite** if, for any $\mathbf{x}$ such that $\mathbf{x}^T\mathbf{x} > 0$, we have $\mathbf{x}^T\mathcal{M}\mathbf{x} > 0$.
- A matrix $\mathcal{R}$ is **orthonormal** if $\mathcal{R}^T\mathcal{R} = \mathcal{I} = \mathcal{I}^T = \mathcal{R}\mathcal{R}^T$. Orthonormal matrices are necessarily square.

Orthonormal matrices: You should think of orthonormal matrices as rotations, because they do not change lengths or angles. For $\mathbf{x}$ a vector, $\mathcal{R}$ an orthonormal matrix, and $\mathbf{u} = \mathcal{R}\mathbf{x}$, we have $\mathbf{u}^T\mathbf{u} = \mathbf{x}^T\mathcal{R}^T\mathcal{R}\mathbf{x} = \mathbf{x}^T\mathcal{I}\mathbf{x} = \mathbf{x}^T\mathbf{x}$. This means that $\mathcal{R}$ doesn't change lengths. For $\mathbf{y}$, $\mathbf{z}$ both unit vectors, we have that the cosine of the angle between them is $\mathbf{y}^T\mathbf{x}$; but, by the same argument as above, the inner product of $\mathcal{R}\mathbf{y}$ and $\mathcal{R}\mathbf{x}$ is the same as $\mathbf{y}^T\mathbf{x}$. This means that $\mathcal{R}$ doesn't change angles, either.

Eigenvectors and Eigenvalues: Assume $\mathcal{S}$ is a $d \times d$ symmetric matrix, $\mathbf{v}$ is a $d \times 1$ vector, and λ is a scalar. If we have

$$\mathcal{S}\mathbf{v} = \lambda\mathbf{v}$$

then $\mathbf{v}$ is referred to as an **eigenvector** of $\mathcal{S}$ and λ is the corresponding **eigenvalue**. Matrices don't have to be symmetric to have eigenvectors and eigenvalues, but the symmetric case is the only one of interest to us.

In the case of a symmetric matrix, the eigenvalues are real numbers, and there are d distinct eigenvectors that are normal to one another, and can be scaled to have unit length. They can be stacked into a matrix $\mathcal{U} = [\mathbf{v}_1, \ldots, \mathbf{v}_d]$. This matrix is orthonormal, meaning that $\mathcal{U}^T\mathcal{U} = \mathcal{I}$. This means that there is a diagonal matrix Λ such that

$$\mathcal{S}\mathcal{U} = \mathcal{U}\Lambda.$$

In fact, there is a large number of such matrices, because we can reorder the eigenvectors in the matrix $\mathcal{U}$, and the equation still holds with a new Λ, obtained by reordering the diagonal elements of the original Λ. There is no reason to keep track of this complexity. Instead, we adopt the convention that the elements of $\mathcal{U}$ are always ordered so that the elements of Λ are sorted along the diagonal, with the largest value coming first.

D. Forsyth, *Probability and Statistics for Computer Science*,
https://doi.org/10.1007/978-3-319-64410-3_15

Diagonalizing a symmetric matrix: This gives us a particularly important procedure. We can convert any symmetric matrix $\mathcal{S}$ to a diagonal form by computing

$$\mathcal{U}^T \mathcal{S} \mathcal{U} = \Lambda.$$

This procedure is referred to as **diagonalizing** a matrix. Again, we assume that the elements of $\mathcal{U}$ are always ordered so that the elements of Λ are sorted along the diagonal, with the largest value coming first. Diagonalization allows us to show that positive definiteness is equivalent to having all positive eigenvalues, and positive semidefiniteness is equivalent to having all non-negative eigenvalues.

Factoring a matrix: Assume that $\mathcal{S}$ is symmetric and positive semidefinite. We have that

$$\mathcal{S} = \mathcal{U} \Lambda \mathcal{U}^T$$

and all the diagonal elements of Λ are non-negative. Now construct a diagonal matrix whose diagonal entries are the positive square roots of the diagonal elements of Λ; call this matrix $\Lambda^{(1/2)}$. We have $\Lambda^{(1/2)}\Lambda^{(1/2)} = \Lambda$ and $(\Lambda^{(1/2)})^T = \Lambda^{(1/2)}$. Then we have that

$$\mathcal{S}=(\mathcal{U}\Lambda^{(1/2)})(\Lambda^{(1/2)}\mathcal{U}^T)=(\mathcal{U}\Lambda^{(1/2)})(\mathcal{U}\Lambda^{(1/2)})^T$$

so we can factor $\mathcal{S}$ into the form $\mathcal{X}\mathcal{X}^T$ by computing the eigenvectors and eigenvalues.

15.1.1 The Singular Value Decomposition

For any $m \times p$ matrix $\mathcal{X}$, it is possible to obtain a decomposition

$$\mathcal{X} = \mathcal{U} \Sigma \mathcal{V}^T$$

where $\mathcal{U}$ is $m \times m$, $\mathcal{V}$ is $p \times p$, and Σ is $m \times p$ and is diagonal. If you don't recall what a diagonal matrix looks like when the matrix *isn't* square, it's simple. All entries are zero, except the i, i entries for i in the range 1 to $\min(m, p)$. So if Σ is tall and thin, the top square is diagonal and everything else is zero; if Σ is short and wide, the left square is diagonal and everything else is zero. Both $\mathcal{U}$ and $\mathcal{V}$ are orthonormal (i.e. $\mathcal{U}\mathcal{U}^T = \mathcal{I}$ and $\mathcal{V}\mathcal{V}^T = \mathcal{I}$).

Notice that there is a relationship between forming an SVD and diagonalizing a matrix. In particular, $\mathcal{X}^T\mathcal{X}$ is symmetric, and it can be diagonalized as

$$\mathcal{X}^T\mathcal{X} = \mathcal{V}\Sigma^T\Sigma\mathcal{V}^T.$$

Similarly, $\mathcal{X}\mathcal{X}^T$ is symmetric, and it can be diagonalized as

$$\mathcal{X}\mathcal{X}^T = \mathcal{U}\Sigma\Sigma^T\mathcal{U}.$$

15.1.2 Approximating A Symmetric Matrix

Assume we have a $k \times k$ symmetric matrix $\mathcal{T}$, and we wish to construct a matrix $\mathcal{A}$ that approximates it. We require that (a) the rank of $\mathcal{A}$ is precisely $r < k$ and (b) the approximation should minimize the **Frobenius norm**, that is,

$$\| (\mathcal{T} - \mathcal{A}) \|_F{}^2 = \sum_{ij} (T_{ij} - A_{ij})^2.$$

It turns out that there is a straightforward construction that yields $\mathcal{A}$.

The first step is to notice that if $\mathcal{U}$ is orthonormal and $\mathcal{M}$ is any matrix, then

$$\| \mathcal{U}\mathcal{M} \|_F = \| \mathcal{M}\mathcal{U} \|_F = \| \mathcal{M} \|_F.$$

This is true because $\mathcal{U}$ is a rotation (as is $\mathcal{U}^T = \mathcal{U}^{-1}$), and rotations do not change the length of vectors. So, for example, if we write $\mathcal{M}$ as a table of row vectors $\mathcal{M} = [\mathbf{m}_1, \mathbf{m}_2, \dots \mathbf{m}_k]$, then $\mathcal{U}\mathcal{M} = [\mathcal{U}\mathbf{m}_1, \mathcal{U}\mathbf{m}_2, \dots \mathcal{U}\mathbf{m}_k]$. Now $\| \mathcal{M} \|_F^{\,2} = \sum_{j=1}^{k} \|\mathbf{m}_j\|^2$, so $\| \mathcal{U}\mathcal{M} \|_F^{\,2} = \sum_{i=1}^{k} \|\mathcal{U}\mathbf{m}_k\|^2$. But rotations do not change lengths, so $\|\mathcal{U}\mathbf{m}_k\|^2 = \|\mathbf{m}_k\|^2$, and so $\| \mathcal{U}\mathcal{M} \|_F = \| \mathcal{M} \|_F$. To see the result for the case of $\mathcal{M}\mathcal{U}$, just think of $\mathcal{M}$ as a table of row vectors.

Notice that, if $\mathcal{U}$ is the orthonormal matrix whose columns are eigenvectors of $\mathcal{T}$, then we have

$$\| (\mathcal{T} - \mathcal{A}) \|_F^{\,2} = \| \mathcal{U}^T (\mathcal{T} - \mathcal{A}) \mathcal{U} \|_F^{\,2}.$$

Now write Λ_r for $\mathcal{U}^T \mathcal{A} \mathcal{U}$, and Λ for the diagonal matrix of eigenvalues of $\mathcal{T}$. Then we have

$$\| (\mathcal{T} - \mathcal{A}) \|_F^{\,2} = \| \Lambda - \Lambda_A \|_F^{\,2},$$

an expression that is easy to solve for Λ_A. We know that Λ is diagonal, so the best Λ_A is diagonal, too. The rank of $\mathcal{A}$ must be r, so the rank of Λ_A must be r as well. To get the best Λ_A, we keep the r largest diagonal values of Λ, and set the rest to zero; Λ_A has rank r because it has only r non-zero entries on the diagonal, and every other entry is zero.

Now to recover $\mathcal{A}$ from Λ_A, we know that $\mathcal{U}^T\mathcal{U} = \mathcal{U}\mathcal{U}^T = \mathcal{I}$ (remember, $\mathcal{I}$ is the identity). We have $\Lambda_A = \mathcal{U}^T \mathcal{A} \mathcal{U}$, so

$$\mathcal{A} = \mathcal{U} \Lambda_A \mathcal{U}^T.$$

We can clean up this representation in a useful way. Notice that only the first r columns of $\mathcal{U}$ (and the corresponding rows of $\mathcal{U}^T$) contribute to $\mathcal{A}$. The remaining $k - r$ are each multiplied by one of the zeros on the diagonal of Λ_A. Remember that, by convention, Λ was sorted so that the diagonal values are in descending order (i.e. the largest value is in the top left corner). We now keep only the top left $r \times r$ block of Λ_A, which we write Λ_r. We then write $\mathcal{U}_r$ for the $k \times r$ matrix consisting of the first r columns of $\mathcal{U}$. Then

$$\mathcal{A} = \mathcal{U}_r \Lambda_r \mathcal{U}^T$$

This is so useful a result, I have displayed it in a box; you should remember it.

Procedure 15.1 (Approximating a Symmetric Matrix with a Low Rank Matrix) Assume we have a symmetric $k \times k$ matrix $\mathcal{T}$. We wish to approximate $\mathcal{T}$ with a matrix $\mathcal{A}$ that has rank $r < k$. Write $\mathcal{U}$ for the matrix whose columns are eigenvectors of $\mathcal{T}$, and Λ for the diagonal matrix of eigenvalues of $\mathcal{A}$ (so $\mathcal{A}\mathcal{U} = \mathcal{U}\Lambda$). Remember that, by convention, Λ was sorted so that the diagonal values are in descending order (i.e. the largest value is in the top left corner).

Now construct Λ_r from Λ by setting the $k - r$ smallest values of Λ to zero, and keeping only the top left $r \times r$ block. Construct $\mathcal{U}_r$, the $k \times r$ matrix consisting of the first r columns of $\mathcal{U}$. Then

$$\mathcal{A} = \mathcal{U}_r \Lambda_r \mathcal{U}_r^T$$

is the best possible rank r approximation to $\mathcal{T}$ in the Frobenius norm.

Now if $\mathcal{A}$ is positive semidefinite (i.e. if at least the r largest eigenvalues of $\mathcal{T}$ are non-negative), then we can factor $\mathcal{A}$ as in the previous section. This yields a procedure to approximate a symmetric matrix by factors. This is so useful a result, I have displayed it in a box; you should remember it.

Procedure 15.2 (Approximating a Symmetric Matrix with Low Dimensional Factors) Assume we have a symmetric $k \times k$ matrix $\mathcal{T}$. We wish to approximate $\mathcal{T}$ with a matrix $\mathcal{A}$ that has rank $r < k$. We assume that at least the r largest eigenvalues of $\mathcal{T}$ are non-negative. Write $\mathcal{U}$ for the matrix whose columns are eigenvectors of $\mathcal{T}$, and Λ for the diagonal matrix of eigenvalues of $\mathcal{A}$ (so $\mathcal{A}\mathcal{U} = \mathcal{U}\Lambda$). Remember that, by convention, Λ was sorted so that the diagonal values are in descending order (i.e. the largest value is in the top left corner).

(continued)

Now construct Λ_r from Λ by setting the $k-r$ smallest values of Λ to zero and keeping only the top left $r \times r$ block. Construct $\Lambda_r^{(1/2)}$ by replacing each diagonal element of Λ with its positive square root. Construct $\mathcal{U}_r$, the $k \times r$ matrix consisting of the first r columns of $\mathcal{U}$. Then write $\mathcal{V} = (\mathcal{U}_r \Lambda_r^{(1/2)})$

$$\mathcal{A} = \mathcal{V}\mathcal{V}^T$$

is the best possible rank r approximation to $\mathcal{T}$ in the Frobenius norm.

15.2 Some Special Functions

Error functions and Gaussians: The **error function** is defined by

$$\mathsf{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

and programming environments can typically evaluate the error function. This fact is made useful to us by a simple change of variables. We get

$$\frac{1}{\sqrt{2\pi}} \int_0^x e^{\frac{-u^2}{2}} du = \frac{1}{\sqrt{\pi}} \int_0^{\frac{x}{\sqrt{2}}} e^{-t^2} dt = \frac{1}{2}\mathsf{erf}\left(\frac{x}{\sqrt{2}}\right).$$

A particularly useful manifestation of this fact comes by noticing that

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^0 e^{\frac{-t^2}{2}} dt = 1/2$$

(because $\frac{1}{\sqrt{2\pi}} e^{\frac{-u^2}{2}}$ is a probability density function, and is symmetric about 0). As a result, we get

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{\frac{-t^2}{2}} dt = 1/2\left(1 + \mathsf{erf}\left(\frac{x}{\sqrt{2}}\right)\right).$$

Inverse error functions: We sometimes wish to know the value of x such that

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{\frac{-t^2}{2}} dt = p$$

for some given p. The relevant function of p is known as the **probit function** or the **normal quantile function**. We write

$$x = \Phi(p).$$

The probit function Φ can be expressed in terms of the **inverse error function**. Most programming environments can evaluate the inverse error function (which is the inverse of the error function). We have that

$$\Phi(p) = \sqrt{2}\mathsf{erf}^{-1}(2p-1).$$

One problem we solve with some regularity is: choose u such that

$$\int_{-u}^u \frac{1}{\sqrt{2\pi}} \exp\left(-x^2/2\right) dx = p.$$

Notice that

$$\begin{aligned}\frac{p}{2} &= \frac{1}{\sqrt{2\pi}}\int_0^u e^{\frac{-t^2}{2}}dt \\ &= \frac{1}{2}\text{erf}\left(\frac{u}{\sqrt{2}}\right)\end{aligned}$$

so that

$$u = \sqrt{2}\text{erf}^{-1}(p).$$

Gamma functions: The gamma function $\Gamma(x)$ is defined by a series of steps. First, we have that for n an integer,

$$\Gamma(n) = (n-1)!$$

and then for z a complex number with positive real part (which includes positive real numbers), we have

$$\Gamma(z) = \int_0^\infty t^z \frac{e^{-t}}{t} dt.$$

By doing this, we get a function on positive real numbers that is a smooth interpolate of the factorial function. We won't do any real work with this function, so won't expand on this definition. In practice, we'll either look up a value in tables or require a software environment to produce it.

15.3 Splitting a Node in a Decision Tree

We want to choose a split that yields the most information about the classes. To do so, we need to be able to account for information. The proper measure is entropy (described in more detail below). You should think of entropy as the number of bits, on average, that would be required to determine the value of a random variable. Filling in the details will allow us to determine which of two splits is better, and to tell whether it is worth splitting at all. At a high level, it is easy to compute which of two splits is better. We determine the entropy of the class conditioned on each split, then take the split which yields the lowest entropy. This works because less information (fewer bits) are required to determine the value of the class once we have that split. Similarly, it is easy to compute whether to split or not. We compare the entropy of the class conditioned on each split to the entropy of the class without a split, and choose the case with the lowest entropy, because less information (fewer bits) are required to determine the value of the class in that case.

15.3.1 Accounting for Information with Entropy

It turns out to be straightforward to keep track of information, in simple cases. We will start with an example. Assume I have 4 classes. There are 8 examples in class 1, 4 in class 2, 2 in class 3, and 2 in class 4. How much information *on average* will you need to send me to tell me the class of a given example? Clearly, this depends on how you communicate the information. You could send me the complete works of Edward Gibbon to communicate class 1; the Encyclopaedia for class 2; and so on. But this would be redundant. The question is how little can you send me. Keeping track of the amount of information is easier if we encode it with bits (i.e. you can send me sequences of '0's and '1's).

Imagine the following scheme. If an example is in class 1, you send me a '1'. If it is in class 2, you send me '01'; if it is in class 3, you send me '001'; and in class 4, you send me '101'. Then the expected number of bits you will send me is

$$\begin{aligned}&p(\text{class} = 1)1 + p(2)2 + p(3)3 + p(4)3 \\ &= \frac{1}{2}1 + \frac{1}{4}2 + \frac{1}{8}3 + \frac{1}{8}3\end{aligned}$$

which is 1.75 bits. This number doesn't have to be an integer, because it's an expectation.

Notice that for the i'th class, you have sent me $-\log_2 p(i)$ bits. We can write the expected number of bits you need to send me as

$$-\sum_i p(i)\log_2 p(i).$$

This expression handles other simple cases correctly, too. You should notice that it isn't really important *how many* objects appear in each class. Instead, the *fraction* of all examples that appear in the class is what matters. This fraction is the prior probability that an item will belong to the class. You should try what happens if you have two classes, with an even number of examples in each; 256 classes, with an even number of examples in each; and 5 classes, with $p(1) = 1/2$, $p(2) = 1/4$, $p(3) = 1/8$, $p(4) = 1/16$ and $p(5) = 1/16$. If you try other examples, you may find it hard to construct a scheme where you can send as few bits *on average* as this expression predicts. It turns out that, in general, the smallest number of bits you will need to send me is given by the expression

$$-\sum_i p(i)\log_2 p(i)$$

under all conditions, though it may be hard or impossible to determine what representation is required to achieve this number.

The **entropy** of a probability distribution is a number that scores how many bits, on average, would need to be known to identify an item sampled from that probability distribution. For a discrete probability distribution, the entropy is computed as

$$-\sum_i p(i)\log_2 p(i)$$

where i ranges over all the numbers where $p(i)$ is not zero. For example, if we have two classes and $p(1) = 0.99$, then the entropy is 0.0808, meaning you need very little information to tell which class an object belongs to. This makes sense, because there is a very high probability it belongs to class 1; you need very little information to tell you when it is in class 2. If you are worried by the prospect of having to send 0.0808 bits, remember this is an average, so you can interpret the number as meaning that, if you want to tell which class each of 10^4 independent objects belong to, you could do so in principle with only 808 bits.

Generally, the entropy is larger if the class of an item is more uncertain. Imagine we have two classes and $p(1) = 0.5$, then the entropy is 1, and this is the largest possible value for a probability distribution on two classes. You can always tell which of two classes an object belongs to with just one bit (though you might be able to tell with even less than one bit).

15.3.2 Choosing a Split with Information Gain

Write $\mathcal{P}$ for the set of all data at the node. Write $\mathcal{P}_l$ for the left pool, and $\mathcal{P}_r$ for the right pool. The entropy of a pool $\mathcal{C}$ scores how many bits would be required to represent the class of an item in that pool, on average. Write $n(i;\mathcal{C})$ for the number of items of class i in the pool, and $N(\mathcal{C})$ for the number of items in the pool. Then the entropy $H(\mathcal{C})$ of the pool $\mathcal{C}$ is

$$-\sum_i \frac{n(i;\mathcal{C})}{N(\mathcal{C})}\log_2 \frac{n(i;\mathcal{C})}{N(\mathcal{C}}.$$

It is straightforward that $H(\mathcal{P})$ bits are required to classify an item in the parent pool $\mathcal{P}$. For an item in the left pool, we need $H(\mathcal{P}_l)$ bits; for an item in the right pool, we need $H(\mathcal{P}_r)$ bits. If we split the parent pool, we expect to encounter items in the left pool with probability

$$\frac{N(\mathcal{P}_l)}{N(\mathcal{P})}$$

and items in the right pool with probability

$$\frac{N(\mathcal{P}_r)}{N(\mathcal{P})}.$$

This means that, on average, we must supply

$$\frac{N(\mathcal{P}_l)}{N(\mathcal{P})}H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})}H(\mathcal{P}_r)$$

bits to classify data items if we split the parent pool. Now a good split is one that results in left and right pools that are informative. In turn, we should need fewer bits to classify once we have split than we need before the split. You can see the difference

$$I(\mathcal{P}_l, \mathcal{P}_r; \mathcal{P}) = H(\mathcal{P}) - \left(\frac{N(\mathcal{P}_l)}{N(\mathcal{P})}H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})}H(\mathcal{P}_r)\right)$$

as the **information gain** caused by the split. This is the average number of bits that you *don't* have to supply if you know which side of the split an example lies. Better splits have larger information gain.

Recall that our decision function is to choose a feature at random, then test its value against a threshold. Any data point where the value is larger goes to the left pool; where the value is smaller goes to the right. This may sound much too simple to work, but it is actually effective and popular. Assume that we are at a node, which we will label k. We have the pool of training examples that have reached that node. The i'th example has a feature vector $\mathbf{x}_i$, and each of these feature vectors is a d dimensional vector.

We choose an integer j in the range $1 \ldots d$ uniformly and at random. We will split on this feature, and we store j in the node. Recall we write $x_i^{(j)}$ for the value of the j'th component of the i'th feature vector. We will choose a threshold t_k, and split by testing the sign of $x_i^{(j)} - t_k$. Choosing the value of t_k is easy. Assume there are N_k examples in the pool. Then there are $N_k - 1$ possible values of t_k that lead to different splits. To see this, sort the N_k examples by $x^{(j)}$, then choose values of t_k halfway between example values. For each of these values, we compute the information gain of the split. We then keep the threshold with the best information gain.

We can elaborate this procedure in a useful way, by choosing m features at random, finding the best split for each, then keeping the feature and threshold value that is best. It is important that m is a lot smaller than the total number of features—a usual root of thumb is that m is about the square root of the total number of features. It is usual to choose a single m, and choose that for all the splits.