Matrices and linear functions

We can make a coordinate representation for linear functions out of a coordinate representation for vectors. Suppose we have a linear function $L \in U \to V$, a basis $b_1 \ldots b_n$ for U, and a basis $c_1 \ldots c_n$ for V. We can apply L to one of our vectors $b_j \in U$, and expand the result Lb_j in terms of our basis for V: we pick coefficients $\ell_{1j}, \ell_{2j}, \ldots$ so that

$$Lb_j = \ell_{1j}c_1 + \ell_{2j}c_2 + \ldots + \ell_{mj}c_m$$

We can do the same for all of the other basis vectors in U, expanding each one's image under L in terms of our basis for V. The resulting coefficients ℓ_{ij} form a basis representation for L: we can map the abstract vector space of linear operators to the concrete vector space of mn-dimensional real vectors.

We typically write out the coordinates of L as a matrix:

$$L \in U \to V \quad \leftrightarrow \quad \begin{pmatrix} \ell_{11} & \ldots & \ell_{1n} \\ \vdots & \ddots & \vdots \\ \ell_{m1} & \ldots & \ell_{mn} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

That is, instead of writing our mn coordinates as one long vector in $\mathbb{R}^{mn}$, we write them as an $m \times n$ matrix. (Formally these are two separate coordinate representations, since we are using two different concrete vector spaces — but often people will move back and forth freely between them since the numerical coordinates are the same.)

This representation agrees with our usual idea of matrix-vector multiplication: take a vector x whose coordinate representation is $u_1b_1 + u_2b_2 + \ldots + u_mb_m$. Then by linearity,

$$Lx = L\sum_{i=1}^{m} u_jb_j = \sum_{j=1}^{m} u_jLb_j$$

Expanding each term Lb_j, we get

$$Lx = \sum_{j=1}^{m} u_j \sum_{i=1}^{n} \ell_{ij}c_i = \sum_{i=1}^{n} \left[\sum_{j=1}^{m} u_j\ell_{ij}\right] c_i$$

The ith coordinate of Lx is the coefficient of c_i in the above expression, namely $v_i = \sum_{j=1}^{m} u_j\ell_{ij}$ — which is exactly the ith entry of the vector we get by multiplying the matrix coordinate representation of L by the vector coordinate representation of x.

As was the case for vectors, picking different bases will result in different coordinate representations for L. So, we can get two completely different matrices that represent the same linear function: as before, the function hasn't changed, just our representation of it.

It's helpful to think of the matrix $\ell = (\ell_{ij})_{ij}$ as a linear function in $\mathbb{R}^m \to \mathbb{R}^n$, as compared to L, which is a linear function in $U \to V$. Our coordinate representations for U, V, and $U \to V$ agree in such a way that we can perform corresponding operations in the abstract and concrete vector spaces:

$$Lx = y \quad \leftrightarrow \quad \ell u = v$$

Transpose and adjoint

Let U and V be inner product spaces, and let $L \in U \to V$ be a linear function. We define the *adjoint* function $L^* \in V \to U$ by the property

$$\langle Lx, y \rangle = \langle x, L^* y \rangle \qquad \forall x \in U,\ y \in V$$

(Note that the inner product on the LHS is in V, while the one on the RHS is in U.) It's not obvious from the definition, but

- The adjoint function always exists and is unique.
- The matrix representation of L^* is the *transpose* of the matrix representation of L. That is, if the i, j coordinate of L is ℓ_{ij}, then the i, j coordinate of L^* is ℓ_{ji}. We write ℓ^T for the matrix transpose.

We can easily check the above statements: suppose that L is a linear function represented by a matrix ℓ. Suppose x is a vector, with concrete coordinates u. Then

$$\langle x, Lx \rangle = u \cdot \ell u = \sum_i u_i [\ell u]_i$$

Here we have written $[\ell u]_i$ for the ith coordinate of the matrix-vector product ℓu. Writing out the matrix-vector product, we have

$$[\ell u]_i = \sum_j \ell_{ij} u_j$$

so

$$u \cdot \ell u = \sum_{ij} u_i u_j \ell_{ij}$$

On the other hand,

$$\langle L^* x, x \rangle = (\ell^T u) \cdot u = \sum_{ij} u_i u_j \ell_{ij}$$

by a similar argument. So, we have verified that, if we take ℓ^T to be the matrix representation of L^*, we achieve $\langle x, Lx \rangle = \langle L^* x, x \rangle$ as desired.

We can view adjoint and transpose themselves as linear functions: the adjoint maps $L \in U \to V$ to $L^* \in V \to U$, while the transpose maps $\ell \in \mathbb{R}^{m \times n}$ to $\ell^T \in \mathbb{R}^{n \times m}$. If we take the adjoint twice, we get back the original linear function: $L^{**} = L$. Similarly, taking the transpose twice gets back to the original matrix: $\ell^{TT} = \ell$.

If $U = V$ then we can potentially have $L = L^*$; such an operator is called *self-adjoint*. Similarly, if $m = n$, we can have $\ell = \ell^T$; such a matrix is called *symmetric*. Self-adjoint operators correspond to symmetric matrices.

Range and nullspace

The range of a function $f \in (U \to V)$ is the set of all its possible outputs:

$$\text{range}(f) = \{ f(x) \mid x \in U \} \subseteq V$$

Suppose that f is linear. Then its range is a subspace of V — possibly all of V. In fact, if we have a matrix L that represents f, then the range of f is equal to the span of the columns of L. This follows directly from the expression for matrix multiplication and the definition of the span: Lx is a linear combination of the columns of L, with weights given by the entries of x.

The *rank* of a linear function is defined as the dimension of its range. Similarly, the rank of a matrix is defined as the rank of its corresponding linear function. For example, if our function is

$$f\left[\begin{pmatrix} x \\ y \end{pmatrix}\right] = \begin{pmatrix} x + y \\ 2x + 2y \end{pmatrix}$$

then its range is the set of all vectors whose second component is twice the first, $\text{span}((1,2)^T)$. This is a one-dimensional set, so the rank of f is 1.

Suppose now that f is an operator, so that $U = V$. If the rank of f is less than the dimension of V, then there will be some vectors $x \in V$ such that $f(x) = 0$. The set of all such x is the *nullspace* or *kernel* of f:

$$\text{null}(f) = \{x \mid f(x) = 0\}$$

The dimension of the nullspace is called the *nullity* of f. The nullspace and nullity of a matrix are defined similarly.

For example, for the function defined above, the vector

$$\begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

is in the nullspace — in fact the nullspace is equal to the span of this vector, so the nullity of f is 1.

The rank and nullity of a linear function are equal to the rank and nullity of its adjoint. (And so the rank and nullity of a matrix are equal to the rank and nullity of its transpose.) For example, the adjoint of the function f defined above is

$$f^*\left[\begin{pmatrix} u \\ v \end{pmatrix}\right] = \begin{pmatrix} u + 2v \\ u + 2v \end{pmatrix}$$

We can check that nullspace of f^* is the span of the vector $(2, -1)^T$, which has dimension 1 as claimed. Notice that this vector is orthogonal to the range of f:

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ -1 \end{pmatrix} = 0$$

It turns out that this is always true: the range of a function and the nullspace of its adjoint are always orthogonal. (That is, any vector in the range is orthogonal to any vector in the nullspace of the adjoint. Similarly for the range of a matrix and the nullspace of its transpose.) In fact, these two subspaces together span all of V: the sum of the rank and nullity of a linear operator is the dimension of the corresponding vector space. These statements are sometimes called the *fundamental theorem of linear algebra*.

Finding coordinate representations

Suppose we have a vector space V with a basis $B = \{b_1, b_2, \ldots b_n\} \subseteq V$. We noted above that any abstract vector $x \in V$ has a unique coordinate representation in terms of B; this coordinate representation is a concrete vector $u \in \mathbb{R}^n$. If we want to work with coordinate representations, one option is to assume that someone gives us a subroutine that computes them: that is, given x and B, it returns u.

If we assume a bit more structure on V — if we assume it is an inner product space — then we can write this subroutine ourselves. That is, given x and B, we can calculate the coordinates u.

Since x and B are vectors in an abstract inner product space, we can only use the abstract inner product space interface to interact with them: for example, we can add pairs of vectors or take inner products between them. We will use this ability to write down a system of linear equations that the coordinates must satisfy. We can then solve this system to find the desired coordinates.

A system of equations

To get our system, suppose we take the inner product between x and one of the basis vectors b_j. By linearity we have

$$\langle x, b_j \rangle = \left\langle \sum_{i=1}^n u_i b_i,\ b_j \right\rangle = \sum_{i=1}^n u_i \langle b_i, b_j \rangle$$

Define $g_j = \langle x, b_j \rangle$ and $G_{ji} = \langle b_i, b_j \rangle$. The above equation then becomes

$$g_j = \sum_{i=1}^n G_{ji} u_i$$

or in matrix form

$$g = Gu$$

So, we can compute G and g by taking inner products between pairs of vectors; then we can solve the linear system to find the coordinates u.

> *The matrix G satisfies a couple of interesting properties: first, since the inner product is symmetric, the matrix G is symmetric, $G_{ij} = G_{ji}$. Second, it is positive definite, a property that we'll define below. These two properties guarantee that the above system of equations has a unique solution.*

Once we have our system $Gu = g$, there are a variety of algorithms for solving it to find u. Probably the best advice is to hand the system to an appropriate library function: for example, Matlab provides the `\` operator, while Python provides `scipy.linalg.solve`. If you need to solve a small system by hand, the best method is probably Gaussian elimination: repeatedly eliminate a variable by adding multiples of one equation to all of the others. (We'll see an example below.)

Do *not* try to solve the system by inverting the matrix G. While this works with exact arithmetic, it can cause all sorts of problems if we try to do it with the approximate arithmetic that happens in a CPU or GPU. (We'll give more details later.)

What if B is not a basis?

What happens if we accidentally start from a set B that is not really a basis for V? There are two ways that B could fail to be a basis:

- B might be too big. That is, there might be a vector we could remove: a vector that we can express as a linear combination of the other vectors in B.
- Or, B might be too small: there might be vectors in V that cannot be expressed as linear combinations of vectors in B.

It's even possible for both of these problems to happen at once.

In the first case, our software library will typically complain when we ask it to solve the system of equations: e.g., it might throw a division-by-zero exception, it might return `Inf` or `NaN`, or it might warn of numerical problems such as roundoff errors. In some cases our library might try to be clever: e.g., it could set some of the coefficients u_i to zero, perhaps in combination with a warning. This sort of cleverness is usually OK: we'll still have $x = \sum_{i=1}^n u_i b_i$, even if some components of u are superfluous.

In the second case, the library will fail silently: it thinks it found the correct coordinates u, but we do not have $x = \sum_{i=1}^n u_i b_i$.

> *We'll still get something interesting: the best possible approximation of x within the span of B. This behavior can be desirable: e.g., we might use it to find a good-enough low-dimensional representation of a very high-dimensional vector.*

On the other hand, the library might give us a failure or a warning even if in principle it could have succeeded. This typically happens due to numerical problems: there might be some vector in B that is extremely close to being a linear combination of the other vectors, so that within machine precision we can't tell the difference.

To guard against all of the above problems and to catch silent errors, it's wise to check our residual — the difference between x and $\sum_{i=1}^n u_i b_i$. If all goes really well this residual should be tiny, e.g., with coordinates around 10^{-12} in 64-bit arithmetic. If we run into mild numerical problems the residual can be a bit bigger, say on the order of 10^{-6} or 10^{-8} ; and if we run into severe numerical problems the residual can even be larger than our original vector x.

Gaussian elimination

Suppose we have a system of equations like

$$\begin{array}{rrrcl} x & +y & +z & = & 3 \\ 2x & +y & & = & 5 \\ -x & +y & -2z & = & 4 \end{array}$$

A good way to solve for x, y, z is *Gaussian elimination*. We'll do this example in lecture.

There are lots more examples online — for example, on Wikipedia: Gaussian elimination.

Slicing and stacking

Given a matrix ℓ, we sometimes need to refer to smaller matrices or vectors formed by keeping some rows or columns from ℓ and crossing out others. These smaller matrices or vectors are called *slices*. If $I \subseteq \{1 \ldots m\}$ and $J \subseteq \{1 \ldots n\}$ are sets of indices, then we'll write $\ell_{I,J}$ for the slice formed by keeping only the elements ℓ_{ij} with indices $i \in I$ and $j \in J$. For example, if

$$\ell = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad I = \{1,3\} \quad J = \{1,2\}$$

then

$$\ell_{I,J} = \begin{pmatrix} 1 & 2 \\ 7 & 8 \end{pmatrix}$$

We will sometimes use colon notation for index sets: in a mathematical expression, `start:end` represents the set of integers from `start` to `end`, and `start:skip:end` represents the set of integers from `start` to `end`, skipping by `skip` between successive elements. For example, $3:5$ means $\{3,4,5\}$, and $1:2:8$ means $\{1,3,5,7\}$. Just a colon on its own is shorthand for the entire set of indices in some dimension. So for example, $\ell_{:,3}$ is column 3 of ℓ, while $\ell_{1:2,:}$ is rows 1 and 2.

Many programming languages have similar slicing conventions. For example, Matlab allows indexing expressions like `A(:,3:2:7)`, meaning columns 3, 5, and 7 of `A`.

Python has a similar slicing convention, both with constructs like `range` and with colon notation. But unlike math notation, Python indexing is *zero-based* and Python *excludes* `end` *from the range*. So for example, `range(3)` in Python means the set $\{0, 1, 2\}$, and `a[1:3]` means indices 1 and 2 of `a` (which are the second and third elements). This notation mismatch can often be confusing, and is a common source of bugs.

In the other direction, it's often useful to construct a matrix by gluing together smaller pieces. This is called *stacking*. For example, if

$$A \in \mathbb{R}^{2\times 2} \qquad B \in \mathbb{R}^{2\times 3} \qquad C \in \mathbb{R}^{3\times 2} \qquad D \in \mathbb{R}^{3\times 3}$$

then we can form a 5×5 matrix by stacking:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

The original matrices are now slices of X: for example, $X_{1:2,3:5} = B$.

Inverse

Let U and V be inner product spaces, and let $L \in U \to V$ be a linear function. We define the *inverse* function $L^{-1} \in V \to U$ by the property

$$y = Lx \quad \Leftrightarrow \quad x = L^{-1}y$$

or equivalently

$$x = L^{-1}Lx,\ y = LL^{-1}y \qquad \forall x \in U,\ y \in V$$

Unlike the adjoint, the inverse doesn't always exist. If it does, we say L is *invertible*. If L is invertible, then $(L^{-1})^{-1} = L$.

An inverse can only exist if U and V have the same dimension. If U and V have different dimensions, we might be able to find a *left inverse* or a *right inverse*. These act like inverses when they are applied in the correct order:

$$x = L^{\text{left}}Lx \quad \forall x \in U$$

$$y = LL^{\text{right}}y \quad \forall y \in V$$

If L is not invertible, it is called *singular*. We might still want a function that acts like an inverse "whenever that makes sense". Such an operator is called the *pseudoinverse*, written $L^\dagger$. Formally, we define $L^\dagger$ so that, if $x = L^\dagger b$, then the error $\|Lx - b\|$ is as small as possible.

We can define an inverse operation on matrices as well: if ℓ and ℓ^{-1} are matrices that satisfy

$$\ell^{-1}\ell x = x \qquad \ell\ell^{-1}y = y$$

for all $x \in U, y \in V$, then we say that the ℓ^{-1} is the inverse of ℓ.

As we might hope, matrix inverses are related to linear function inverses: if the matrix ℓ is a coordinate representation of the linear function L, and if L is invertible, then ℓ^{-1} exists and is a coordinate representation of the linear function L^{-1}.

Similarly, there is a matrix representation of the pseudoinverse $L^\dagger$ as well. If the linear function L is not invertible, then none of its representation matrices are invertible (no matter what basis we use). If ℓ is one of these representation matrices, then $\ell^\dagger$ is the corresponding coordinate representation of $L^\dagger$.

Like adjoint and transpose, both inverse and pseudoinverse swap the roles of input and output vector spaces: for example, if $L \in U \to V$, then $L^\dagger \in V \to U$.

As a reminder, the inverse is much more useful as a mathematical tool than a computational one. It's rarely a good idea to compute the inverse of a matrix explicitly, since it tends to incur numerical errors. Instead, it's typically better to use an algorithm like Gaussian elimination, which lets us apply the inverse matrix to one or more vectors directly, without ever computing the inverse itself.

> *Of course, there are exceptions to the rule above. In case an example helps: suppose we want to minimize $L(A) = \ln\det A + f(A)$, where A is a square symmetric matrix and f is some differentiable function. As part of the minimization, we might want to compute the gradient of L, which is $\nabla L = A^{-1} + \nabla f(A)$; so to get the gradient we have to explicitly compute a matrix inverse. We may incur numerical errors while doing so, but this is OK: minimization algorithms like gradient descent are often quite robust to numerical errors in computing the gradient.*

If we do need to compute the inverse of a matrix A, we can do so by solving linear systems of equations: if we write e_i for the ith column of the identity matrix (that is, a 1 in coordinate i and zeros everywhere else) and if x is the ith column of A^{-1}, then x

satisfies

$$Ax = e_i$$

(This follows from the matrix equation $AA^{-1} = I$, since it is the ith column of that equation.) So, we can find x by solving this linear system, and we can find all of A^{-1} by repeating the process for each e_i.

We can do something similar to find left and right inverses: we can use $AA^{\text{right}} = I$ to solve for a column of A^{right} at a time, and we can use $A^{\text{left}}A = I$ to solve for a row of A^{left} at a time.

Matrix patterns

Consider a linear operator $L \in U \to V$ and its matrix representation ℓ under some bases for U and V. Depending on which bases we pick for U and V, the matrix ℓ can look quite different. If we can pick these bases so that many elements of ℓ are zero, then some computational operations involving ℓ become faster. If the nonzero elements follow a simple pattern, then operations can become even faster.

For example, ℓ is square (the dimensions of U and V are the same) and if all of the nonzeros of ℓ fall on the main diagonal (that is, if $\ell_{ij} = 0$ when $i \neq j$), we say that ℓ is *diagonal*. Writing $\times$ for a possibly-nonzero entry and 0 for an entry that must be zero, a diagonal matrix matches the pattern

$$\begin{pmatrix} \times & 0 & 0 & \dots & 0 \\ 0 & \times & 0 & \dots & 0 \\ 0 & 0 & \times & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \times \end{pmatrix}$$

The simplest diagonal matrix is the *identity* matrix I, which has entries of 1 on the main diagonal and 0 off of it. (That is, all of the $\times$ entries above are 1.) If $U = V$, the identity matrix corresponds to the identity function: that is, the function L that satisfies $Lx = x$ for all $x \in U$.

Another useful pattern is *lower triangular*: a square matrix is lower-triangular if it matches the pattern

$$\begin{pmatrix} \times & 0 & 0 & \dots & 0 \\ \times & \times & 0 & \dots & 0 \\ \times & \times & \times & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \times & \times & \times & \dots & \times \end{pmatrix}$$

that is, if $\ell_{ij} = 0$ when $j > i$. An *upper triangular* matrix follows the opposite pattern: the nonzeros are on and above the main diagonal, so that its transpose is lower triangular.

A matrix with lots of zero entries but no particular pattern is called *sparse*. We can save computation by working with just a list of the nonzero entries and their indices, instead of storing and manipulating all of the zeros. Typically, though, we need a high fraction of nonzero entries to make it worthwhile to do this: say at least 80% zeros, and preferably much more. (The reason for the penalty is that we spend some storage to explicitly represent the indices of the nonzeros, and we waste some computation because of the less-uniform structure of the list.) Fortunately, very sparse matrices are common in some applications: it's perfectly typical to see matrices where 99.9% of the entries are zero.

If a matrix follows multiple patterns, it can be even more convenient to work with: e.g., a sparse lower triangular matrix is more convenient than a matrix that is just sparse or just lower triangular.

Orthogonality

A column-orthogonal matrix is one whose columns are orthogonal vectors. This is equivalent to saying that the matrix $D = A^T A$ is diagonal. A row-orthogonal matrix is one whose rows are orthogonal vectors, so that AA^T is diagonal. A row- or column-orthonormal matrix is one whose rows or columns are orthonormal.

If A is square and its rows are orthonormal, then its columns must also be orthonormal. Similarly, if the columns are orthonormal, the rows must be too. In this case $AA^T = A^T A = I$. We can interpret this equation three ways:

- The dot products of columns of A are either 0 (for distinct columns) or 1 (for the dot product of a column with itself).
- Similarly, the dot products of rows of A are either 0 or 1.
- The inverse of A is A^T.

The first two interpretations come from looking at the matrix equation element by element; the last comes from matching the entire equation to the definition of the inverse.

Positive (semi)definite operators and matrices

A linear operator $A \in U \to U$ is called *positive semidefinite* or *PSD* if $\langle x, Ax \rangle \geq 0$ for all vectors $x \in U$. It is called *positive definite* if the inequality is strict: $\langle x, Ax \rangle > 0$ when $x \neq 0$. We also apply the terms positive semidefinite and positive definite to any matrix representation of A.

Self-adjoint PSD operators have some nice properties we will see below, as do symmetric PSD matrices. In fact, these properties are so nice that some authors include symmetry or self-adjointness in the definition of PSD. (We won't do that, since there are also some uses for asymmetric or non-self-adjoint PSD matrices or operators.) For example, we can test positive definiteness or semidefiniteness of a symmetric matrix easily using Gaussian elimination; this is called Cholesky factorization in the definite case, and LDL^T factorization in the semidefinite case. (Therefore, we can test definiteness of a self-adjoint operator by using Cholesky or LDL^T factorization on any coordinate representation for it.)

To test definiteness of an asymmetric matrix and its corresponding non-self-adjoint operator, we can use the following fact: a matrix A is positive definite (or semidefinite) if and only if $A + A^T$ is. (It's a fun exercise to try to prove this fact.) The latter is clearly symmetric, so we can turn to Cholesky or LDL^T factorization.

Matrix factorizations

Gaussian elimination is also called LU factorization: it allows us to represent a matrix A as a product $A = LU$ of a lower triangular matrix L and an upper triangular matrix U.

This is one of many possible matrix factorizations, all of which can be really useful. For example, factoring a matrix is often a good way to solve many copies of an equation for different right-hand sides:

$$Ax_1 = b_1, \quad Ax_2 = b_2, \quad \ldots$$

We factor A once and then repeatedly use the factorization to solve for the different right-hand sides. If we factor $A = LU$, then for each new vector b_i we can do two triangular backsubstitutions (for L and then for U) to find x_i; this is much faster than solving $Ax_i = b_i$ from scratch each time.

In an LU factorization, the matrix U is the result of all of the row operations we apply: for each diagonal element of A, we use row operations to eliminate all of the off-diagonal elements below it, leaving nonzero elements only on and above the main diagonal.

On the other hand, the matrix L encodes our sequence of row operations. A single row operation can be implemented by a matrix that looks like this:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ -2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

All row operations are invertible; for example, the inverse of this one is

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

If we multiply A by the first matrix above, we will subtract twice the first row of A from the fourth row of A. On the other hand, the second matrix above adds twice the first row of A to the fourth row.

Note that both matrices above are lower triangular. Since the product of lower triangular matrices is lower triangular, we can multiply all of the inverses of our row operations together into a single lower triangular factor L.

If we factor a symmetric positive definite matrix this way, we can require $L = U^T$ (which forces the product to be symmetric and definite). This is called a Cholesky factorization. A variant is to pull out the diagonal entries of L and L^T into a diagonal matrix D, resulting in a factorization

$$A = LDL^T$$

where the matrix L is lower triangular and has all diagonal entries equal to 1:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ \times & 1 & 0 & \dots & 0 \\ \times & \times & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \times & \times & \times & \dots & 1 \end{pmatrix}$$

Since the original matrix is PSD, the entries of D will be nonnegative. We can go back to a Cholesky factorization by multiplying the square root of each diagonal entry into the corresponding row and column of L.

We can also make the same LDL^T factorization for a non-PSD symmetric matrix. But in this case some of the diagonal entries of D will be strictly negative, and so we can't make a Cholesky factorization. This is in fact a great way to test whether a matrix is positive definite: we try to build the LDL^T factorization. If we get all the way to the end with only positive elements of D, we know the matrix is positive definite. If we ever run into a negative element, we know that the matrix is not positive definite.

> *If we ever run into a zero element, the story is slightly more complicated: we need to try a technique called pivoting to avoid having to divide by zero. Pivoting works by rearranging rows and columns of our matrix so that a nonzero element is on the diagonal. If pivoting is impossible, that means we're stuck with only zero elements remaining; so the remaining elements of L and D are zero, and the matrix is PSD but not positive definite.*

One last useful factorization is the *singular value decomposition*:

$$A = U\Sigma V^T$$

where U and V are column-orthonormal and Σ is diagonal and nonnegative. We can choose at least one of U and V to be square: suppose $A \in \mathbb{R}^{n\times m}$. If $m < n$ then we will pick V to be square, $V \in \mathbb{R}^{m\times m}$. Σ will have the same dimensions as V, and U will be rectangular: $U \in \mathbb{R}^{n\times m}$. If $m > n$ then we will choose U to be square and V to be rectangular: $U, \Sigma \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{m\times n}$. (If $m = n$ then both U and V will be square.)

The columns of U and V are called the left and right *singular vectors*, while the diagonal elements of Σ are called the *singular values*. We'll learn more about the singular value decomposition later, when we cover the multivariate normal distribution. But for now, one good use of the singular value decomposition is to compute a pseudoinverse: for an orthonormal matrix like U or V, the pseudoinverse is the transpose, and for a product of matrices, the pseudoinverse reverses the order, so

$$A^\dagger = V\Sigma^\dagger U^T$$

The pseudoinverse of a diagonal matrix is easy to compute: $\Sigma^\dagger$ is a diagonal matrix where we pseudoinvert each element Σ_{ii} independently. That is,

$$\Sigma^\dagger_{ii} = \left\{ \begin{array}{cl} \Sigma_{ii}^{-1} & \text{if } \Sigma_{ii}^{-1} \neq 0 \\ 0 & \text{otherwise} \end{array} \right.$$

(It's not often that we get to set $1/0 = 0$!)

If our matrix A is close to singular, the some of the diagonal elements of Σ will be close to zero. Inverting them will give us huge diagonal elements of $\Sigma^\dagger$, leading to numerical instability. In this case (say, when some diagonal elements are 10^{10} times smaller than others), it can be worth giving up on the small elements and pretending that they are actually zero: this is called *thresholding* the singular values. We won't be able to get an accurate pseudoinverse this way, but when we apply the computed pseudoinverse to solve an equation, our residual can be substantially smaller. If our matrix A is built from data, so that we only know the elements of A approximately, it can be worth thresholding singular values much more aggressively: if one singular value is even 100 times smaller than another, there's a good chance that the small one is mostly capturing measurement noise instead of signal.

Linear algebra libraries typically provide fast routines to find matrix factorizations like the above. These are reliable and stable — generally much better than working with matrices and factorizations by hand. And, chip vendors often optimize these routines for each new architecture they put out; so the routines are often substantially faster than we'd be able to match without a lot of effort.

Linear regression

Above we saw how to learn a linear classifier — a function that maps a vector to a binary label, i.e., an element of $V \to \{-1, 1\}$ where V is an inner product space. Sometimes we want to predict a real number instead of a binary label: e.g., we want to predict a patient's degree of response to a treatment, instead of just whether the patient responds or not. This is called a *regression* problem. Just like for classification, it can be simple and practical to use a linear function as our predictor; this is unsurprisingly called *linear regression*.

Formally, we are given a list of *input data* or *training data*, presented as pairs (x_t, y_t) for $t \in 1:T$. Here $x_t \in V$ is called our *input vector* or our *independent variable*, and $y \in \mathbb{R}$ is called our *label*, our *output*, or our *dependent variable*. We want to learn to predict a new pair (x_{T+1}, y_{T+1}) that wasn't included in our training data, called a *test example*. For this purpose, we learn a vector $w \in V$ and a scalar $b \in \mathbb{R}$, and we predict $\hat{y}_t = \langle w, x_t \rangle + b$. Here w is called a *parameter vector* or a *weight vector*, and b is called an *intercept* or a *constant term*. The hat denotes a predicted quantity: we hope $\hat{y}_t \approx y_t$.

Just as for classification, it can be very useful to use a feature transform for regression. That is, we replace $x_t \in V$ by $\phi(x_t) \in W$, where $\phi \in V \to W$ is our feature function and the new inner product space W is our feature space. That makes our new predictor $\langle w, \phi(x) \rangle + b$. It's still called linear regression: the overall predictor is nonlinear, but the part we are learning is linear. For now, we'll ignore any feature transform — or equivalently we'll assume that our training data (x_t, y_t) come with the transform pre-applied.

Linear regression and linear classification are the Swiss Army knives of machine learning: they are rarely the perfect tool for anything, but they're decent tools for just about everything. As such they're often a good technique to try first: since they are simple and fast to apply, they can often get 80% of the results with 20% of the effort. Better yet, they can reveal problems before we've gone to the effort of applying a more complicated and expensive technique: e.g., our training data could be corrupted somehow, or our test examples could have important but undiscovered differences from our training data.

The most common method for learning a linear predictor is *least squares*: we find the w and b that minimize the sum of squared prediction errors on our training data,

$$L(w) = \sum_{t=1}^{T} (y_t - (\langle w, x_t \rangle + b))^2$$

Commonly we also add a *regularizer* to make the regression more stable,

$$R(w) = \lambda \|w\|^2$$

and minimize $L(w) + R(w)$ instead. Here $\lambda > 0$ is the *regularization parameter*; we'll talk more later about how to choose λ.

As with linear classification, the linear regression prediction function $\langle w, x \rangle + b$ isn't truly linear, but affine. And as with classification, we can remove the difference by using homogeneous coordinates if desired. Or, for some problems, we might just want to omit the intercept b and predict $\hat{y}_t = \langle w, x_t \rangle$.

Solving a linear regression

To solve a linear regression problem, we can turn it into a system of linear equations. For simplicity of notation we'll assume that either we're using homogeneous coordinates or we don't want an intercept term, so that our predictor is just $\langle w, x \rangle$.

The first step is to collect all of our training data into a single big matrix and vector. Write $u_t \in \mathbb{R}^d$ for a concrete representation of the training input x_t. Define

$$X = \begin{pmatrix} | & | & & | \\ u_1 & u_2 & \dots & u_T \\ | & | & & | \end{pmatrix}$$

to be the matrix we get by stacking all of the concrete vectors u_t as columns, and define

$$y = \begin{pmatrix} y_1 & y_2 & \dots & y_T \end{pmatrix}$$

to be the row vector that we get by stacking all of the training labels. The dimensions are $X \in \mathbb{R}^{d \times T}$ and $y \in \mathbb{R}^{1 \times T}$.

It's easy to check from the definition of matrix multiplication that our vector of predictions becomes

$$\hat{y} = w^T X$$

(Here we have stacked $\hat{y}_1 \dots \hat{y}_T$ into a row vector $\hat{y}$.) Since the squared norm of a vector is the sum of its squared components, our sum-squared error becomes

$$L(w) = \|y - wX\|^2$$

We'll see later how to minimize $L(w) + R(w)$ by differentiating it and setting the derivative to zero. For now, we can skip ahead to the answer: the optimal w satisfies

$$w^T(XX^T + \lambda I) = yX^T$$

This is a set of linear equations, called the *normal equations*. You will see variants: e.g., it's common to write the transpose of the above equations, and it's common to define X and y to stack training examples as rows instead of columns.

A good way to solve the normal equations is to use the singular value decomposition that we defined earlier. Suppose

$$X = U\Sigma V^T$$

where $U, \Sigma \in \mathbb{R}^{d\times d}$ and $v \in \mathbb{R}^{T\times d}$. Then the normal equations become

$$(\Sigma + \lambda\Sigma^{-1})U^T w = V^T y^T$$

Exercise: prove this.

Since $(\Sigma + \lambda\Sigma^{-1})$ is diagonal and U is orthonormal, it's very easy to solve the above representation of the normal equations: we compute $V^T y^T$, divide each element by the corresponding diagonal element of $(\Sigma + \lambda\Sigma^{-1})$, and then multiply the result by U to get w.