The vector space $\mathbb{R}^n$

The best-known example of a vector space is $\mathbb{R}^n$: the space of n-element vectors of real numbers. The first important properties of $\mathbb{R}^n$ — the ones that make it a vector space — are:

- It supports addition and scalar multiplication of vectors: $ax + y \in \mathbb{R}^n$ for $a \in \mathbb{R}$, $x, y \in \mathbb{R}^n$.
- Addition and scalar multiplication satisfy the usual properties: e.g., associativity and commutativity; distributivity of scalar multiplication over addition; the existence of an additive identity (the zero vector).

In addition, $\mathbb{R}^n$ has some more nice properties, beyond just being a vector space:

- It has an inner product, written $x \cdot y$ or $\langle x, y\rangle$, with $x \cdot y \in \mathbb{R}$ for any $x, y \in \mathbb{R}^n$.
- The inner product satisfies all of the usual properties: e.g., it is bilinear (linear in each argument with the other fixed) and symmetric, and $x \cdot x = 0 \Leftrightarrow x = 0$.
- We can use the inner product to define a norm $\|x\| = \sqrt{x \cdot x}$. This norm lets us define other useful concepts like the convergence of a sequence of vectors to a limit point.
- Under the above norm, every Cauchy sequence converges. That is, for a sequence x_t with $t = 1, 2, \ldots$, if $\|x_t - x_{t+1}\| \to 0$ as $t \to \infty$, then $x = \lim_{t\to\infty} x_t$ exists and $x \in \mathbb{R}^n$.

The first two of these properties make $\mathbb{R}^n$ an *inner product space*. The last two make it a *complete* inner product space. Both of these are stronger than just being a vector space: they are *extensions* of the vector space data type.

Other vector spaces

All of these properties make $\mathbb{R}^n$ a nice formal system to work with. But sometimes we need to work with objects that are not in $\mathbb{R}^n$: e.g., matrices or functions. We can still use some of the same tricks when working with these objects, by abstracting out the important properties that we like from $\mathbb{R}^n$.

This sort of abstraction is very important in ML: it lets us take algorithms that are designed to learn an element of $\mathbb{R}^n$ (often called a *parameter vector*) and generalize them to work on other classes of objects such as matrices or functions. For some classes of objects, this is the main way that we know how to design effective learning algorithms.

We define a vector space to be a set V (whose elements are called *vectors*) together with operations of addition and scalar multiplication that behave in the usual way (i.e., copying the important properties from $\mathbb{R}^n$): e.g., $ax + y \in V$ for $a \in \mathbb{R}$ and $x, y \in V$.

Note that the scalar a is still a real number, even though we've changed the vectors that it is multiplying; the set of scalars doesn't change when we go from $\mathbb{R}^n$ to a general vector space V. More precisely, we're going to be talking about a *vector space over the reals*, which means that our scalars are elements of $\mathbb{R}$.

> *One can also define vector spaces that use different kinds of scalars: e.g., complex numbers $\mathbb{C}$ or even other fields such as quaternions or integers modulo a prime.*

Some examples:

- The set of matrices $\mathbb{R}^{m\times n}$ is a vector space, if we interpret addition and scalar multiplication elementwise.
- We can make the set of real functions of one argument $\mathbb{R} \to \mathbb{R}$ into a vector space, if we define addition and scalar multiplication to operate separately on each possible argument to our functions. For example, we would define $(f + 3g)(x) = f(x) + 3g(x)$; this is the usual interpretation of addition and scalar multiplication of functions.
- The real numbers themselves are a vector space, though kind of a trivial one: the vectors and scalars come from the same set. If we wanted to emphasize the vector-space-ness, we could use some notation to distinguish between reals-as-scalars and reals-as-vectors: e.g., $3 \cdot \overrightarrow{100} + 5 \cdot \overrightarrow{1} = \overrightarrow{305}$.

One vector space can be contained inside another one: $U \subseteq V$. In this case U is called a *subspace* of V, not just a subset.

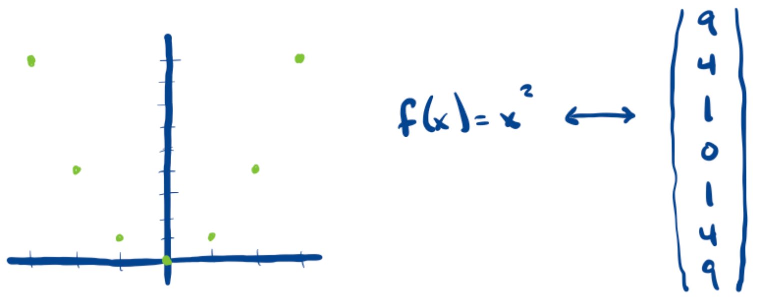
Orthogonal and normal vectors

In an inner product space, two vectors x and y are called *orthogonal* if $\langle x, y\rangle = 0$. Geometrically, orthogonal vectors form a right angle with one another.

A vector x is called *normal* if it has unit length: $\langle x, x\rangle = 1$. Two vectors are *orthonormal* if they are orthogonal and both are normal.

Vector space of functions

We mentioned above the example of a vector space whose elements are functions. Thinking of functions as vectors in a vector space may seem somewhat counterintuitive, but we actually do it all the time: whenever we graph a function f using a plotting package, we evaluate $f(x_i)$ on a grid of points x_i. If there are n grid points, we can collect the function values into a vector $u \in \mathbb{R}^n$.


$f(x) = x^2$
$\longleftrightarrow$
$\begin{pmatrix} 9 \\ 4 \\ 1 \\ 0 \\ 1 \\ 4 \\ 9 \end{pmatrix}$

The vector u behaves a lot like the original function f: we can add together two functions f and g by adding their vectors u and v, and we can multiply a function f by a scalar a by computing au (whose elements are $af(x_i) = (af)(x_i)$). These vector operations in $\mathbb{R}^n$ operate separately on each coordinate, so they duplicate the operations of adding and scalar multipying functions, which operate separately on each possible argument to the function. That is, we can either sample each of our functions on our grid first, and then do all of the vector arithmetic in $\mathbb{R}^n$; or we can do all of the arithmetic in our function space, and then sample the resulting function on our grid. The result will be the same either way.

The difference between the function f and the vector of samples u is just that the function contains more information: we can imagine the function as an infinite-dimensional vector, as if we had sampled it at every possible real number to create a vector with uncountably many coordinates.

Of course, it's hard to stuff a vector with uncountably many coordinates into the memory of a computer. So, we'll need to come up with some other strategy if we want to work with such a vector space. We'll give more examples of how to do this below, but one nice example is a vector space of *polynomials*: we can write $\mathbb{R}[x]$ for the vector space of polynomials in one real variable x. Elements of this vector space are objects like

$$2,\ x^2,\ 3x^3 - 5x$$

It's easy to verify that polynomials behave as vectors: e.g., addition and scalar multiplication behave as expected. But this is an infinite-dimensional vector space: we'll talk more about dimension below, but for now, imagine representing a polynomial with a list of its nonzero coefficients. There can be arbitrarily many nonzero coefficients (one for x^3, one for x^{100}, one for the millionth power of x, and so forth), so we can't place an upper bound on the number of coefficients required. But, each individual polynomial has finitely many nonzero coefficients, so we can store and work with polynomials in a computer without difficulty.

Complete spaces

Above we described how to think of matrices or functions as vectors in a vector space. We also described how to upgrade a vector space to an inner product space by defining an inner product $\langle x, y\rangle$. For example,

- A useful inner product for matrices is

$$\langle X, Y\rangle = \sum_{i=1,\, j=1}^{i=m,\, j=n} X_{ij} Y_{ij} = \mathrm{tr}(X^T Y) = \mathrm{tr}(Y X^T)$$

- A useful inner product for functions $f, g \in X \to \mathbb{R}$ is

$$\int_X f(x) g(x) dx$$

- There are often multiple ways to define an inner product for a given vector space. For example, the usual inner product for $\mathbb{R}^n$ is $\langle x, y\rangle = x \cdot y = \sum_{i=1}^n x_i y_i$. But for any matrix A, we can define a new inner product $\langle x, y\rangle_A = (Ax) \cdot (Ay)$. This is called a *Mahalonobis* inner product. (To be precise, we require that A be square and have an inverse; see below for definitions.)
- For another example, there are many standard inner products for function spaces, in addition to the one above. We'll see a few below.

For general spaces we typically use the angle-bracket notation for inner product, to distinguish from the dot notation for the standard inner product in $\mathbb{R}^n$.

As always, once we have an inner product, we can define a norm $\|x\| = \sqrt{\langle x, x\rangle}$. Given this norm, we reuse the usual definitions for convergence:

- A sequence $x_1, x_2, \ldots \in V$ is a *Cauchy sequence* whenever $\|x_t - x_{t+1}\| \to 0$ as

$t \to \infty$.

- A sequence *converges* to a vector x whenever $\|x_t - x\| \to 0$ as $t \to \infty$.

We say that an inner product space V is *complete* if any Cauchy sequence x_t in V converges to some vector $x \in V$. A complete inner product space is often called a *Hilbert space*, after the mathematician David Hilbert.

Almost all of the vector spaces that we deal with in ML will be complete inner product spaces, so that we don't need to worry so much about checking all of the above properties. But the properties are worth keeping in the back of our minds, since they can cause algorithms to fail in inconvenient ways if we're not careful.

> *In case it helps to have a counterexample in mind, one good one is $\mathbb{Q}^n$, the vector space of length-n tuples of rational coordinates. This is an inner product space if we take our scalars to be rational and adopt the standard dot product. But this space is not complete: the limit of a Cauchy sequence of rational numbers is potentially irrational. For example, the limit of the sequence $3, 3.1, 3.14, 3.141, \ldots$ is π.*
>
> *Another good one is $\mathbb{R}[x]$, the vector space of polynomials in a real variable x. We can augment this space with an inner product; e.g., we can define $\langle x^i, x^j \rangle$ to be 1 when $i = j$ and 0 otherwise, and extend to arbitrary polynomials by linearity. But, because polynomials can have unbounded degree, we can make a Cauchy sequence that never converges: e.g., $1,\ 1 + \frac{1}{2}x,\ 1 + \frac{1}{2}x + \frac{1}{4}x^2,\ \ldots,\ \sum_{i=0}^{t} \frac{1}{2^i} x^i,\ \ldots$. The distance between successive elements of this sequence is $\frac{1}{2^{t+1}} \to 0$, so it is a Cauchy sequence. But it doesn't converge to any polynomial: any polynomial that we might claim to be the limit has some finite degree d. But that means that it could never be close to any term in our series of degree greater than d. In particular, the closest a degree-d polynomial can get to one of these terms is a distance of at least $\frac{1}{2^{d+1}}$ — contradicting the statement that our degree-d polynomial is the correct limit.*

If we have an incomplete inner product space V, we can always construct a complete inner product space $\bar{V}$ that contains V. The new space is called the *completion* of V.

Span

Given a list or set of vectors B, their *span* is the set of all their linear combinations:

$$\operatorname{span}(B) = \{w_1 b_1 + w_2 b_2 + \ldots + w_n b_n \mid w_i \in \mathbb{R}, b_i \in B\}$$

Note that the set-builder notation discards duplicates: if $\sum_i v_i a_i = \sum_i w_i b_i$ for two different lists of weights v_i and w_i and vectors a_i and b_i, we only include the linear combination vector once.

The span is a vector space: we can add any two vectors in the span by adding their linear-combination weights, and scalar-multiply a vector in the span by scalar-multiplying its weights. If the elements of B come from some vector space V, then the span is a subspace of V: $\mathrm{span}(B) \subseteq V$. For example, if $|B| = 2$ and the elements of B are nonzero 3-element vectors, then the span of B is a line or a plane in $\mathbb{R}^3$.

Note that the above definition makes sense even if B has infinitely many elements: we include all linear combinations of *finitely many* elements of B. This is sometimes called the *finite span* to avoid ambiguity.

Functions on vector spaces

We've seen that we can define many different vector spaces, with many different properties. Given all of these vector spaces, it makes sense to look at functions that map between them. For example:

- We could define a function $r \in \mathbb{R}^2 \to \mathbb{R}^2$ that takes a 2d vector and rotates it 30° clockwise around the origin.
- Or, we could define a function $p \in \mathbb{R}^5 \to \mathbb{R}^2$ that extracts the first and fourth components of a vector (discarding the second, third, and fifth).
- Or, if $\bar{H}$ is the function space we defined earlier (the completion of the span of all functions of the form $q_y(x) = e^{(y-x)^2}$), we could define a function $e_z \in \bar{H} \to \mathbb{R}$ that evaluates a function at some fixed input $z \in \mathbb{R}$. That is, $e_z(f) = f(z)$ for any $f \in \bar{H}$.

When dealing with functions like these, it's a good idea always to keep track of the types of expressions. Type-checking like this can catch lots of simple errors: for example, with the definitions above, $r(p(v))$ makes sense as long as $v \in \mathbb{R}^5$. But, the expression $p(r(v))$ doesn't make sense, no matter what the type of v is.

Functionals, operators

There are two special names that it's worth knowing for specific kinds of functions between vector spaces. First, if the value of the function is a real number, the function is called a *functional*. (The same is true for a complex-valued function on a complex vector space, and similarly for any other scalar field.)

Second, a mapping from a vector space to itself is called an *operator*: e.g., the "rotate 30°" function above is an operator on $\mathbb{R}^2$. Or, the differential $\frac{d}{dx}$ is an operator on a vector space of functions, mapping a function like x^2 to another function like $2x$. (Actually it's many overloaded operators, since we can define many different notions of derivative on many different function spaces; we'll be more precise later.)

Sometimes people will also use the word operator for a function from one vector space to another; here we'll try to reserve it for a mapping from a vector space to itself.

Linear functions

A function between vector spaces $f \in U \to V$ is called *linear* if it satisfies

$$f(ax + by) = af(x) + bf(y)$$

for all vectors $x, y \in U$ and all scalars $a, b \in \mathbb{R}$. That is, we can take addition and scalar multiplication in the input space U and turn them into addition and scalar multiplication in the output space V.

> *Another way to say the same thing is that f is a vector-space homomorphism. A homomorphism is a function that preserves structure — in this case the behavior of the key vector-space operations of addition and scalar multiplication.*

We've already seen some examples of linear functions: e.g., point evaluation is a linear functional on the vector space $\mathbb{R} \to \mathbb{R}$, since for instance $e_z(3f) = 3\, e_z(f)$. And, the differential $\frac{d}{dx}$ is a linear operator, since for instance $\frac{d}{dx}(f + g) = \frac{d}{dx} f + \frac{d}{dx} g$ (the sum rule). Another well-known example of a linear operator is multiplication by a matrix: if $A \in \mathbb{R}^{n \times m}$, then the function $f \in \mathbb{R}^m \to \mathbb{R}^n$ defined by $f(x) = Ax$ is linear.

> *If $f \in V \to \mathbb{R}$ is a linear functional, and if V is a complete inner product space, then there always exists some vector $g \in V$ such that $f(x) = \langle g, x \rangle$ for all $x \in V$. This property — that every linear functional can be implemented as an inner product — is another one of the useful properties of $\mathbb{R}^n$ that extends to more-general spaces.*

One common source of confusion is the difference between *linear* and *affine* functions. Consider the function $f(x) = 3x + 2$, where $x \in \mathbb{R}$. Its plot is a line, but if we check the above definition, f is not linear: for example, $f(3) = 11$, but $f(3 \cdot 2) = 20 \neq 11 \cdot 2$. Instead f is *affine*: it is a linear function plus a constant.

Coordinates

So far we've mostly been talking about vectors as opaque objects: the only operations they support are addition and scalar multiplication (and sometimes inner product and limit). This is in contrast to the usual practice for the vector space $\mathbb{R}^n$, where we think of a vector as being a tuple of real numbers. In this latter view, vectors support an additional operation, *indexing*: we can ask for (say) the seventh coordinate of a vector in $\mathbb{R}^{15}$.

The first view above is called *coordinate-free* or *abstract*, while the second view is called *concrete*. We can think of these as different data types: the abstract vector space type does not support indexing, while the concrete vector space type does.

Usually, it's better to use abstract vectors in proofs, since that way the proofs work for any vector space, not just $\mathbb{R}^n$. But if we want to manipulate vectors on a computer, we need a concrete representation.

Such a concrete representation is called a *coordinate system*. An example of a coordinate system is a map, like on the screen of a GPS device: the abstract vector space is the set of positions for a small portion of the earth's surface (say, the portion within five miles of the professor's house), while the concrete vector space is the space of pixel coordinates for the map image.

> *More precisely, the abstract vector space is a tangent space to the earth's surface: it extends infinitely far in any direction, but it's only a decent approximation to the earth in a local region. And the concrete vector space that represents the map image also extends infinitely far in any direction, but we only define the actual image on a local region of this space: e.g., we don't give RGB colors for pixels with negative coordinates.*

A coordinate system is a mapping from an abstract vector space to a concrete one. The mapping has to be *linear*, as defined above. And it must be *homogeneous*: the zero vector in the abstract space must map to the zero vector in the concrete space. Finally, it has to be *invertible*: no two distinct vectors can have the same coordinates. Together, these properties mean that scaling or adding vectors is the same as scaling or adding their coordinates.

Typically there are many (infinitely many) possible coordinate systems we can use for a given abstract vector space V. Some might be better or worse than others for a given purpose; we'll give examples later of why this might happen. But it's important to remember that the original vector space V is *independent* of the coordinate system we use for it. So for example, we might have two different maps of the area around the professor's house; these maps might have different orientation and scaling, and so the concrete coordinates of the professor might be different in the two maps. But of course the professor's location (a vector in the abstract space) stays the same, no matter which coordinates we use to represent it.

Basis

To go from abstract to concrete, we need a basis for our vector space V. A basis is a set of vectors $B = \{b_1, b_2, \ldots\}$ with two properties:

- It spans our vector space, $V = \text{span}(B)$. That is, any vector in V can be represented as a linear combination of basis vectors from B: for any $v \in V$, there exist scalars $x_1, x_2, \ldots$ such that $v = x_1 b_1 + x_2 b_2 + \ldots$.
- It's *minimal*: removing any element from B destroys the previous property.

The elements of B are called *basis vectors*.

Given a basis, we can use the scalars $x_1, x_2, \ldots$ as a concrete representation of the abstract vector v. That is,

$$x \in V \quad \leftrightarrow \quad \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$$

For example, in a 2d map, we could pick as our basis one vector pointing up in the image, and one vector pointing right, with a length that corresponds to one meter in the physical world. Or we could pick one vector pointing north in the image, and one pointing east, with a length that corresponds to 1km in the real world. We could even pick one vector pointing southeast and one north-by-northwest — although this would violate the usual convention of preserving lengths and angles from the real world in our map coordinates (see the description of orthogonormal bases below).

It's a fundamental theorem that any vector space has a basis, and that every basis has the same number of basis vectors (though this number might be infinite). The number of basis vectors is called the *dimension* of the corresponding vector space. This is the same idea of dimension that we're used to: a line is a one-dimensional vector space, a plane is a two-dimensional vector space, and so forth.

Coordinate representations in a given basis are *unique*: given a vector $x \in V$, there's only one linear combination of basis vectors that yields x. (We'll give an algorithm for computing the coordinates below.)

As mentioned above, the coordinate representation depends on which basis we pick: if we choose a different basis B', the representation of v can look totally different. It's important to remember that v hasn't changed, only our representation of it.

> *It's slightly tricky to define basis and dimension in infinite-dimensional vector spaces. The difficulty comes in whether to allow convergent infinite sums of basis vectors. (Our above definition does not.) It only makes sense to allow infinite sums if we have an appropriate notion of convergence; we don't always have such a notion, but we do in many important cases, including all complete inner product spaces. Depending on whether we do allow infinite sums, the size of a basis might be different.*

Linear independence

We said above that a basis has to be minimal: if any vector b_i can be removed from B while keeping $\operatorname{span}(B) = V$, then B is not a basis. More generally, for any list of vectors X, if we can remove some vector $x_i \in X$ without changing $\operatorname{span}(X)$, the vectors in X are called *linearly dependent*. This means that x_i can be expressed as a linear combination of the other vectors $x_j \in X$ for $j \neq i$. On the other hand, if no vector can be removed from X without changing $\operatorname{span}(X)$, then the vectors are called *linearly independent*.

For example, the vectors

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

are linearly dependent. One way to show this is that we can represent the last vector as the sum of the first two. On the other hand, any two of these vectors are linearly independent.

Geometrically, linear independence is a generalization of being collinear or coplanar: that is, we are asking whether any vector lies in the line, plane, or other subspace defined by the other vectors.

Orthonormal bases

If the elements of a basis B are orthonormal (that is, if $\langle b_i, b_j \rangle = 0$ when $i \neq j$ and $\langle b_i, b_i \rangle = 1$ for all i) then we say that B is an orthonormal basis. Orthonormal bases are important since they preserve lengths and angles. That is, if $x, y \in V$ are abstract vectors and $u, v \in \mathbb{R}^n$ are their concrete representations, then $\langle x, y \rangle = u \cdot v$.

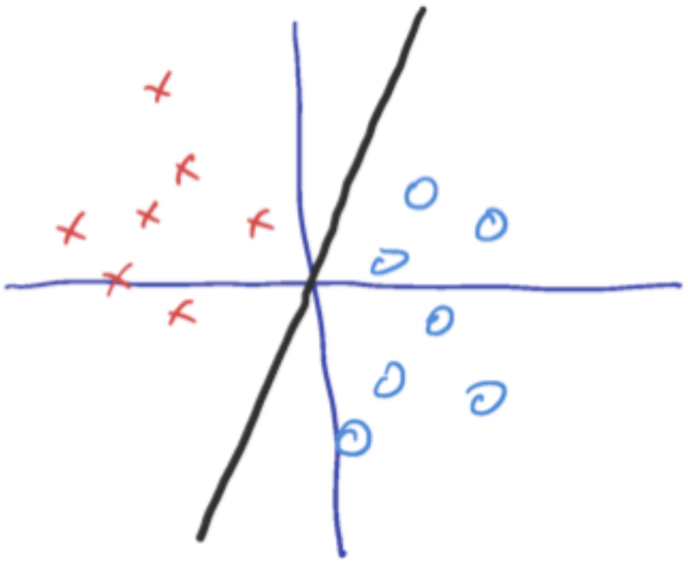
This is a crucial property if we're working in an inner product space. For an arbitrary vector space, lengths and angles might not even be defined, so there's no point in trying to preserve them. But for an inner product space, lengths and angles are an intrinsic property of the space. So, if we have a non-orthonormal basis for an inner product space, and if we want to use it to build a concrete representation, we need to be careful:

- With no additional effort, we preserve the effect of addition and scalar multiplication.
- With the coordinate representation described below, we preserve application of linear functions.
- But the standard inner product in $\mathbb{R}^n$ will give us wrong answers for the lengths and angles between vectors. Instead, if we want to reason about lengths and angles, either we need to switch to an orthonormal basis or we need to define a nonstandard inner product for $\mathbb{R}^n$. (As it happens, we can use a Mahalonobis inner product; see above for the definition.)

Given any basis for an inner product space, we can construct an orthonormal one. One method for doing so is Gram-Schmidt orthonormalization.

Example: linear classifier

What can we do with a vector space? One simple use is a *linear classifier*. For example, here's a linear classifier in the plane: we split $\mathbb{R}^2$ into two pieces, one part on each side of a line.



We can represent such a classifier as a vector $w \in \mathbb{R}^2$ and a threshold $b \in \mathbb{R}$: we classify a point $x \in \mathbb{R}^2$ as

- $\times$ if $w \cdot x > b$ (i.e., if x is on the side of the line where w points)
- $\circ$ if $w \cdot x < b$ (i.e., if x is on the opposite side of the line, in the direction that $-w$ points)

The function $f(x) = w \cdot x - b$ is called a *discriminant function* for our classifier: it discriminates between the two possible classes based on the sign of f. Since f represents a line, we call it a *linear discriminant*. The line $\{x \mid f(x) = 0\}$ is called the *decision boundary* or the *separator*.

> *Note that the discriminant function f is affine not linear. It's still traditional to call this type of classifier a linear discriminant. This terminology mismatch is unfortunate; but if desired we can avoid it by switching to homogeneous coordinates (defined below).*

Geometrically, w is normal (orthogonal) to the separator, so it tells us the orientation of the separator. Another way to interpret w is as the direction of steepest increase of f (i.e., the gradient).

The sign of w tells us which side of the separator corresponds to which class: the class that's on the side where w points is called the *positive* class, since it's in the direction of increasing f, while the other side is called the *negative* class, since it's in the direction of decreasing f. Flipping the sign of w exchanges the positive and negative classes.

Finally, b and the length of w combine to tell us how far the separator is from the origin: the closest distance from the origin to the separator is $b/\|w\|$. So, scaling up w moves the boundary toward the origin, while scaling up b moves the boundary away from the origin. (This system is redundant, so sometimes we remove an unnecessary degree of freedom by either fixing $b = 1$ or fixing $\|w\| = 1$, and using the other parameter to shift the separator toward or away from the origin.)

There are lots of ways to build a linear discriminant, but one reasonable one is the *Fisher linear discriminant*: suppose we are given a data set containing $n_\times$ points of the $\times$ class and $n_\circ$ points of the $\circ$ class. Call these points $x_i^\times$ and x_j°. We calculate the mean of each class,

$$\mu_\times = \frac{1}{n_\times} \sum_{i=1}^{n_\times} x_i^\times$$

$$\mu_\circ = \frac{1}{n_\circ} \sum_{j=1}^{n_\circ} x_j^\circ$$

Then we draw a line between these class means, and let w and b represent the *perpendicular bisector* of this line:

$$w = \mu_\times - \mu_\circ$$

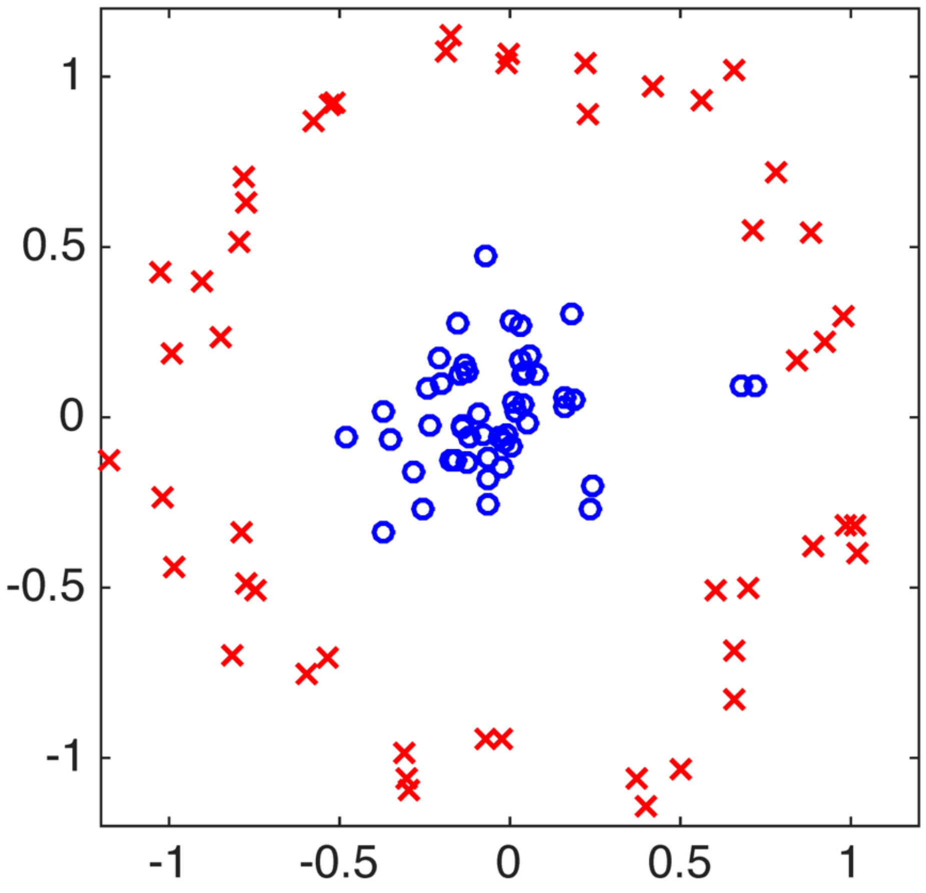
$$b = w \cdot (\mu_\times + \mu_\circ)/2$$

(Exercise: convince yourself that these equations actually do give us the perpendicular bisector.)

> *The above recipe is actually a special case of the discriminant that Fisher proposed. In the general Fisher discriminant (which is quadratic, not linear), we estimate the variance of each class, and use these variances to shift and reshape the boundary between classes.*

Feature transforms

Linear classifiers are really useful, and they can be very efficient to find and work with. But what if we can't separate our classes with just a line?



A plain linear classifier will clearly fail here. But linear classifiers are so convenient that we'd still like to use them if possible.

The answer is a *feature transform*: instead of using a linear classifier in the original vector space V (here $V = \mathbb{R}^2$), we map our points to some other vector space W, called the *feature space*, and use a linear classifier in W instead. If the feature transform $\phi \in V \to W$ is nonlinear, then the overall classifier is nonlinear, even if we use a linear classifier in W. That is, the overall discriminant function for V

$$f(x) = w \cdot \phi(x) - b$$

is nonlinear if ϕ is nonlinear, even if the discriminant in W is linear:

$$g(y) = w \cdot y - b, \qquad y = \phi(x)$$

In the plot above, suppose we take $W = \mathbb{R}^5$ and

$$\phi(x) = (x_1, x_2, x_1 x_2, x_1^2, x_2^2)^T$$

The separator $f(x) = 0$ will then take the form

$$w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2 = b$$

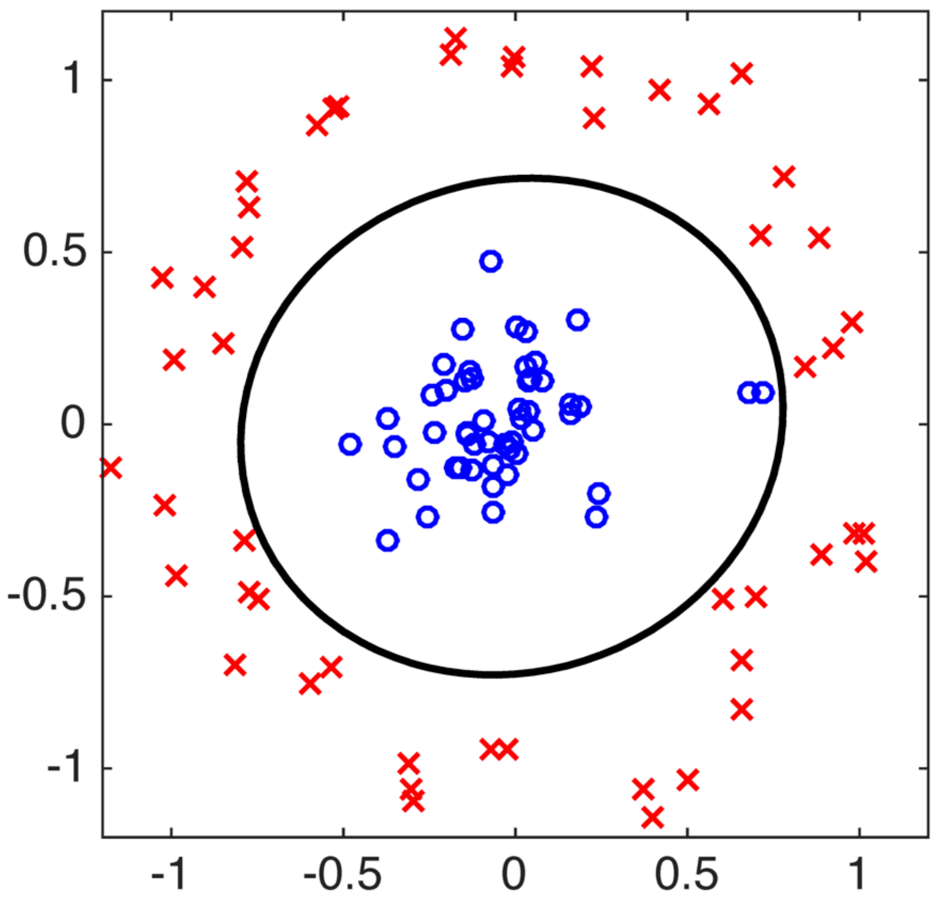
This is a conic section: a circle, ellipse, parabola, or hyperbola. With a more general separator like this, we can classify all of our points correctly — much better than a linear separator.

The advantage of a feature transform is that we can now use all of our knowledge of linear classifiers to help us build a nonlinear classifier. For example, we can build a Fisher linear discriminant by taking the class means in *feature space*:

$$\mu_\times = \frac{1}{n_\times} \sum_{i=1}^{n_\times} \phi(x_i^\times)$$

$$\mu_\circ = \frac{1}{n_\circ} \sum_{j=1}^{n_\circ} \phi(x_j^\circ)$$

From here, the exact same formulas as before get us w and b.



One really simple but useful feature transform is *homogeneous coordinates*: the feature function h just tacks a constant coordinate onto the end of our input vector. For example,

$$h\left[\begin{pmatrix} 2.71 \\ -4.26 \end{pmatrix}\right] = \begin{pmatrix} 2.71 \\ -4.26 \\ 1 \end{pmatrix}$$

This transform lets us turn an affine discriminant function into a truly linear one: if our original discriminant was $w \cdot x - b$, and if we write v for the stacked vector

$$\begin{pmatrix} w \\ -b \end{pmatrix}$$

then our discriminant function becomes

$$v \cdot h(x) = w \cdot x - b$$

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 $b_1 \dots b_n$ for U, and a basis $c_1 \dots c_n$ 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 $\ell_{1j}, \ell_{2j}, \dots$ 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:

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

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,

$$Lx = L\sum_{i=1}^{m} u_j b_j = \sum_{j=1}^{m} u_j L b_j$$

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:

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

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, $X_{1:2,3:5} = B$.

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 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.

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.