Chapter 8

Linear Algebra

> We often hear that mathematics consists mainly of "proving theorems." Is a writer's job mainly that of "writing sentences?"
>
> – Gian-Carlo Rota

The data part of your data science project involves reducing all of the relevant information you can find into one or more data matrices, ideally as large as possible. The rows of each matrix represent items or examples, while the columns represent distinct features or attributes.

Linear algebra is the mathematics of matrices: the properties of arrangements of numbers and the operations that act on them. This makes it the language of data science. Many machine learning algorithms are best understood through linear algebra. Indeed algorithms for problems like linear regression can be reduced to a single formula, multiplying the right chain of matrix products to yield the desired results. Such algorithms can simultaneously be both simple and intimidating, trivial to implement and yet hard to make efficient and robust.

You presumably took a course in linear algebra at some point, but perhaps have forgotten much of it. Here I will review most of what you need to know: the basic operations on matrices, why they are useful, and how to build an intuition for what they do.

8.1 The Power of Linear Algebra

Why is linear algebra so powerful? It regulates how matrices work, and matrices are everywhere. Matrix representations of important objects include:

- *Data*: The most generally useful representation of numerical data sets are as $n \times m$ matrices. The n rows represent objects, items, or instances, while the m columns each represent distinct features or dimensions.

S.S. Skiena, *The Data Science Design Manual,*
Texts in Computer Science, https://doi.org/10.1007/978-3-319-55444-0_8

- *Geometric point sets*: An $n \times m$ matrix can represent a cloud of points in space. The n rows each represent a geometric point, while the m columns define the dimensions. Certain matrix operations have distinct geometric interpretations, enabling us to generalize the two-dimensional geometry we can actually visualize into higher-dimensional spaces.

- *Systems of equations*: A linear equation is defined by the sum of variables weighted by constant coefficients, like:

$$y = c_0 + c_1 x_1 + c_2 x_2 + \ldots c_{m-1} x_{m-1}.$$

  A system of n linear equations can be represented as an $n \times m$ matrix, where each row represents an equation, and each of the m columns is associated with the coefficients of a particular variable (or the constant "variable" 1 in the case of c_0). Often it is necessary to represent the y value for each equation as well. This is typically done using a separate $n \times 1$ array or vector of solution values.

- *Graphs and networks*: Graphs are made up of vertices and edges, where edges are defined as ordered pairs of vertices, like (i, j). A graph with n vertices and m edges can be represented as an $n \times n$ matrix M, where $M[i, j]$ denotes the number (or weight) of edges from vertex i to vertex j. There are surprising connections between combinatorial properties and linear algebra, such as the relationship between paths in graphs and matrix multiplication, and how vertex clusters relate to the eigenvalues/vectors of appropriate matrices.

- *Rearrangement operations*: Matrices can *do* things. Carefully designed matrices can perform geometric operations on point sets, like translation, rotation, and scaling. Multiplying a data matrix by an appropriate *permutation matrix* will reorder its rows and columns. Movements can be defined by *vectors*, the $n \times 1$ matrices powerful enough to encode operations like translation and permutation.

The ubiquity of matrices means that a substantial infrastructure of tools has been developed to manipulate them. In particular, the high-performance linear algebra libraries for your favorite programming language mean that you should *never* implement any basic algorithm by yourself. The best library implementations optimize dirty things like numerical precision, cache-misses, and the use of multiple cores, right down to the assembly-language level. Our job is to formulate the problem using linear algebra, and leave the algorithmics to these libraries.

8.1.1 Interpreting Linear Algebraic Formulae

Concise formulas written as products of matrices can provide the power to do amazing things, including linear regression, matrix compression, and geometric

transformations. Algebraic substitution coupled with a rich set of identities yields elegant, mechanical ways to manipulate such formulas.

However, I find it very difficult to interpret such strings of operations in ways that I really understand. For example, take the "algorithm" behind least squares linear regression, which is:

$$c = (A^T A)^{-1} A^T b$$

where the $n \times m$ system is $Ax = b$ and w is the vector of coefficients of the best fitting line.

One reason why I find linear algebra challenging is the nomenclature. There are many different terms and concepts which must be grokked to really follow what is going on. But a bigger problem is that most of the proofs are, for good reason, algebraic. To my taste, algebraic proofs generally do not carry intuition about why things work the way they do. Algebraic proofs are easier to verify step-by-step in a mechanical way, rather than by understanding the ideas behind the argument.

I will present only one formal proof in this text. And by design both the theorem and the proof are incorrect.

Theorem 1. $2 = 1$.

Proof.

$$\begin{aligned} a &= b \\ a^2 &= ab \\ a^2 - b^2 &= ab - b^2 \\ (a+b)(a-b) &= b(a-b) \\ a+b &= b \\ 2b &= b \\ 2 &= 1 \end{aligned}$$

□

If you have never seen such a proof before, you might find it convincing, even though I trust you understand on a conceptual level that $2 \neq 1$. Each line follows from the one before it, through direct algebraic substitution. The problem, as it turns out, comes when canceling $(a - b)$, because we are in fact dividing by zero.

What are the lessons from this proof? Proofs are about ideas, not just algebraic manipulation. No idea means no proof. To understand linear algebra, your goal should be to first validate the simplest interesting case (typically two dimensions) in order to build intuition, and then try to imagine how it might generalize to higher dimensions. There are always special cases to watch for, like division by zero. In linear algebra, these cases include dimensional mismatches

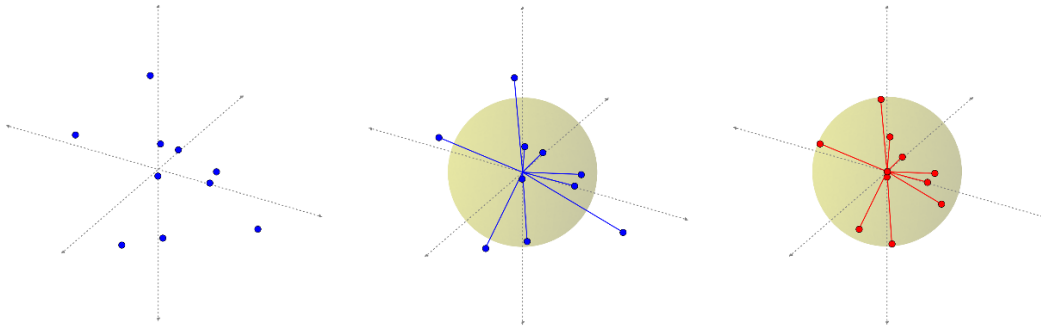

Figure 8.1: Points can be reduced to vectors on the unit sphere, plus magnitudes.

and singular (meaning non-invertible) matrices. The theory of linear algebra works except when it doesn't work, and it is better to think in terms of the common cases rather than the pathological ones.

8.1.2 Geometry and Vectors

There is a useful interpretation of "vectors," meaning $1 \times d$ matrices, as *vectors* in the geometric sense, meaning directed rays from the origin through a given point in d dimensions.

Normalizing each such vector v to be of unit length (by dividing each coordinate by the distance from v to the origin) puts it on a d-dimensional sphere, as shown in Figure 8.1: a circle for points in the plane, a real sphere for $d = 3$, and some unvisualizable hypersphere for $d \geq 4$.

This normalization proves a useful thing to do. The distances between points become angles between vectors, for the purposes of comparison. Two nearby points will define a small angle between them through the origin: small distances imply small angles. Ignoring magnitudes is a form of scaling, making all points directly comparable.

The *dot product* is a useful operation reducing vectors to scalar quantities. The dot product of two length-n vectors A and B is defined:

$$A \cdot B = \sum_{i=1}^{n} A_i B_i$$

We can use the dot product operation to compute the angle $\theta = \angle AOB$ between vectors A and B, where O is the origin:

$$\cos(\Theta) = \frac{A \cdot B}{||A||\ ||B||}$$

Let's try to parse this formula. The $||V||$ symbol means "the length of V." For unit vectors, this is, by definition, equal to 1. In general, it is the quantity by which we must divide V by to make it a unit vector.

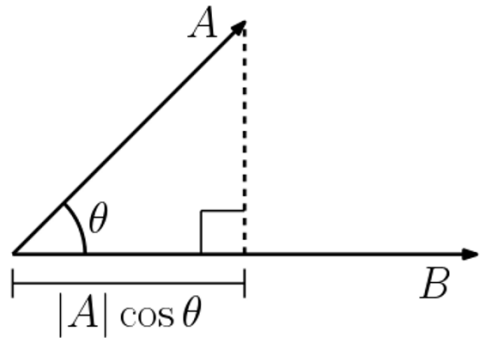

Figure 8.2: The dot product of two vectors defines the cosine of the angle between them.

But what is the connection between dot product and angle? Consider the simplest case of an angle defined between two rays, A at zero degrees and $B = (x, y)$. Thus the unit ray is $A = (1, 0)$. In this case, the dot product is $1 \cdot x + 0 \cdot y = x$, which is exactly what $\cos(\theta)$ should be if B is a unit vector. We can take it on faith that this generalizes for general B, and to higher dimensions.

So a smaller angle means closer points on the sphere. But there is another connection between things we know. Recall the special cases of the cosine function, here given in radians:

$$\cos(0) = 1, \;\; \cos(\pi/2) = 0, \;\; \cos(\pi) = -1.$$

The values of the cosine function range from $[-1, 1]$, exactly the same range as that of the correlation coefficient. Further, the interpretation is the same: two identical vectors are perfectly correlated, while antipodal points are perfectly negatively correlated. Orthogonal points/vectors (the case of $\Theta = \pi/2$) have as little to do with each other as possible.

The cosine function is exactly the correlation of two mean-zero variables. For unit vectors, $||A|| = ||B|| = 1$, so the angle between A and B is completely defined by the dot product.

Take-Home Lesson: The dot product of two vectors measures similarity in exactly the same way as the Pearson correlation coefficient.

8.2 Visualizing Matrix Operations

I assume that you have had some previous exposure to the basic matrix operations of transposition, multiplication, and inversion. This section is intended as a refresher, rather than an introduction.

Figure 8.3: Matrix image examples: Lincoln (left) and his memorial (right). The center image is a linear combination of left and right, for $\alpha = 0.5$.

But to provide better intuition, I will represent matrices as images rather than numbers, so we can *see* what happens when we operate on them. Figure 8.3 shows our primary matrix images: President Abraham Lincoln (left) and the building which serves as his memorial (right). The former is a human face, while the latter contains particularly strong rows and columns.

Be aware that we will be quietly rescaling the matrix between each operation, so the absolute color does not matter. The interesting patterns come in the differences between light and dark, meaning the smallest and biggest numbers in the current matrix. Also, note that the origin element of the matrix $M[1,1]$ represents the upper left corner of the image.

8.2.1 Matrix Addition

Matrix addition is a simple operation: for matrices A and B, each of dimensions $n \times m$, $C = A + B$ implies that:

$$C_{ij} = A_{ij} + B_{ij}, \quad \text{for all } 1 \leq i \leq n \text{ and } 1 \leq j \leq m.$$

Scalar multiplication provides a way to change the weight of every element in a matrix simultaneously, perhaps to normalize them. For any matrix A and number c, $A' = c \cdot A$ implies that

$$A'_{ij} = cA_{ij}, \quad \text{for all } 1 \leq i \leq n \text{ and } 1 \leq j \leq m.$$

Combining matrix addition with scalar multiplication gives us the power to perform *linear combinations* of matrices. The formula $\alpha \cdot A + (1-\alpha) \cdot B$ enables us to fade smoothly between A (for $\alpha = 1$) and B (for $\alpha = 0$), as shown in Figure 8.3. This provides a way to morph the images from A to B.

The *transpose* of a matrix M interchanges rows and columns, turning an $a \times b$ matrix into a $b \times a$ matrix M^T, where

$$M^T_{ij} = M_{ji} \quad \text{for all } 1 \leq i \leq n \text{ and } 1 \leq j \leq m.$$

The transpose of a square matrix is a square matrix, so M and M^T can safely be added or multiplied together. More generally, the transpose is an operation that is used to orient a matrix so it *can* be added to or multiplied by its target.

Figure 8.4: Lincoln (left) and its transposition (right). The sum of a matrix and its transposition is symmetric along its main diagonal (right).

The transpose of a matrix sort of "rotates" it by 180 degrees, so $(A^T)^T = A$. In the case of square matrices, adding a matrix to its transpose is symmetric, as shown in Figure 8.4 (right). The reason is clear: $C = A + A^T$ implies that

$$C_{ij} = A_{ij} + A_{ji} = C_{ji}.$$

8.2.2 Matrix Multiplication

Matrix multiplication is an aggregate version of the vector *dot* or *inner product*. Recall that for two n-element vectors, X and Y, the dot product $X \cdot Y$ is defined:

$$X \cdot Y = \sum_{i=1}^{n} X_i Y_i$$

Dot products measure how "in sync" the two vectors are. We have already seen the dot product when computing the cosine distance and correlation coefficient. It is an operation that reduces a pair of vectors to a single number.

The matrix product XY^T of these two vectors produces a 1×1 matrix containing the dot product $X \cdot Y$. For general matrices, the product $C = AB$ is defined by:

$$C_{ij} = \sum_{i=1}^{k} A_{ik} \cdot B_{kj}$$

For this to work, A and B must share the same inner dimensions, implying that if A is $n \times k$ then B must have dimensions $k \times m$. Each element of the $n \times m$ product matrix C is a dot product of the ith row of A with the jth column of B.

The most important properties of matrix multiplication are:

- *It does not commute*: *Commutativity* is the notation that order doesn't matter, that $x \cdot y = y \cdot x$. Although we take commutativity for granted when multiplying integers, order *does* matter in matrix multiplication. For any pair of non-square matrices A and B, at most one of either AB or BA

has compatible dimensions. But even square matrix multiplication does not commute, as shown by the products below:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix} \neq \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix}$$

and the covariance matrices of Figure 8.5.

- *Matrix multiplication is associative*: *Associativity* grants us the right to parenthesize as we wish, performing operations in the relative order that we choose. In computing the product ABC, we have a choice of two options: $(AB)C$ or $A(BC)$. Longer chains of matrices permit even more freedom, with the number of possible parenthesizations growing exponentially in the length of the chain. All of these will return the same answer, as demonstrated here:

$$\left(\begin{bmatrix} 1 & 2 \\ 4 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \right) \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 8 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 7 & 2 \\ 17 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix} \right) = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 3 & 2 \\ 2 & 0 \end{bmatrix} = \begin{bmatrix} 7 & 2 \\ 17 & 6 \end{bmatrix}$$

There are two primary reasons why associativity matters to us. In an algebraic sense, it enables us to identify neighboring pairs of matrices in a chain and replace them according to an identity, if we have one. But the other issue is computational. The size of intermediate matrix products can easily blow up in the middle. Suppose we seek to calculate $ABCD$, where A is $1 \times n$, B and C are $n \times n$, and D is $n \times 1$. The product $(AB)(CD)$ costs only $2n^2 + n$ operations, assuming the conventional nested-loop matrix multiplication algorithm. In contrast, $(A(BC))D$ weighs in at $n^3 + n^2 + n$ operations.

The nested-loop matrix multiplication algorithm you were taught in high school is trivially easy to program, and indeed appears on page 398. But don't program it. Much faster and more numerically stable algorithms exist in the highly optimized linear algebra libraries associated with your favorite programming language. Formulating your algorithms as matrix products on large arrays, instead of using ad hoc logic is counter-intuitive to most computer scientists. But this strategy can produce very big performance wins in practice.

8.2.3 Applications of Matrix Multiplication

On the face of it, matrix multiplication is an ungainly operation. When I was first exposed to linear algebra, I couldn't understand why we couldn't just multiply the numbers on a pairwise basis, like matrix addition, and be done with it.

The reason we care about matrix multiplication is that there are many things we can do with it. We will review these applications here.

Figure 8.5: The Lincoln memorial M (left) and its covariance matrices. The big block in the middle of $M \cdot M^T$ (center) results from the similarity of all rows from the middle stripe of M. The tight grid pattern of $M^T \cdot M$ (right) reflects the regular pattern of the columns on the memorial building.

Covariance Matrices

Multiplying a matrix A by its transpose A^T is a very common operation. Why? For one thing, we *can* multiply it: if A is an $n \times d$ matrix, then A^T is a $d \times n$ matrix. Thus it is always compatible to multiply AA^T. They are equally compatible to multiply the other way, i.e. $A^T A$.

Both of these products have important interpretations. Suppose A is an $n \times d$ feature matrix, consisting of n rows representing items or points, and d columns representing the observed features of these items. Then:

- $C = A \cdot A^T$ is an $n \times n$ matrix of dot products, measuring the "in sync-ness" among the points. In particular C_{ij} is a measure of how similar item i is to item j.
- $D = A^T \cdot A$ is a $d \times d$ matrix of dot products, measuring the "in sync-ness" among columns or features. Now D_{ij} represents the similarity between feature i and feature j.

These beasts are common enough to earn their own name, *covariance matrices*. This term comes up often in conversations among data scientists, so get comfortable with it. The covariance formula we gave when computing the correlation coefficient was

$$Cov(X, Y) = \sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y}).$$

so, strictly speaking, our beasts are covariance matrices only if the rows or columns of A have mean zero. But regardless, the magnitudes of the matrix product captures the degree to which the values of particular row or column pairs move together.

Figure 8.5 presents the covariance matrices of the Lincoln memorial. Darker spots define rows and columns in the image with the greatest similarity. Try to understand where the visible structures in these covariance matrices come from.

Figure 8.5 (center) presents $M \cdot M^T$, the covariance matrix of the rows. The big dark box in the middle represents the large dot products resulting from any two rows cutting across all the memorial's white columns. These bands of light and dark are strongly correlated, and the intensely dark regions contribute to a large dot product. The light rows corresponding to the sky, pediment, and stairs are equally correlated and coherent, but lack the dark regions to make their dot products large enough.

The right image presents $M^T \cdot M$, which is the covariance matrix of the columns. All the pairs of matrix columns strongly correlate with each other, either positively or negatively, but the matrix columns through the white building columns have low weight and hence a small dot product. Together, they define a checkerboard of alternating dark and light stripes.

Matrix Multiplication and Paths

Square matrices can be multiplied by themselves without transposition. Indeed, $A^2 = A \times A$ is called the *square* of matrix A. More generally A^k is called the kth power of the matrix.

The powers of matrix A have a very natural interpretation, when A represents the *adjacency matrix* of a graph or network. In an adjacency matrix, $A[i, j] = 1$ when (i, j) is an edge in the network. Otherwise, when i and j are not direct neighbors, $A[i, j] = 0$.

For such 0/1 matrices, the product A^2 yields the number of paths of length two in A. In particular:

$$A^2[i, j] = \sum_{k=1}^{n} A[i, k] \cdot A[k, j].$$

There is exactly one path of length two from i to j for every intermediate vertex k such that (i, k) and (k, j) are both edges in the graph. The sum of these path counts is computed by the dot product above.

But computing powers of matrices makes sense even for more general matrices. It simulates the effects of diffusion, spreading out the weight of each element among related elements. Such things happen in Google's famous PageRank algorithm, and other iterative processes such as contagion spreading.

Matrix Multiplication and Permutations

Matrix multiplication is often used just to rearrange the order of the elements in a particular matrix. Recall that high-performance matrix multiplication routines are blindingly fast, enough so they can often perform such operations faster than ad hoc programming logic. They also provide a way to describe such operations in the notation of algebraic formulae, thus preserving compactness and readability.

The most famous rearrangement matrix does nothing at all. The *identity matrix* is an $n \times n$ matrix consisting of all zeros, except for the ones all along

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad M = \begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{pmatrix} \quad PM = \begin{pmatrix} 31 & 32 & 33 & 34 \\ 11 & 12 & 13 & 14 \\ 41 & 42 & 43 & 44 \\ 21 & 22 & 23 & 24 \end{pmatrix}$$

Figure 8.6: Multiplying a matrix by a permutation matrix rearranges its rows and columns.

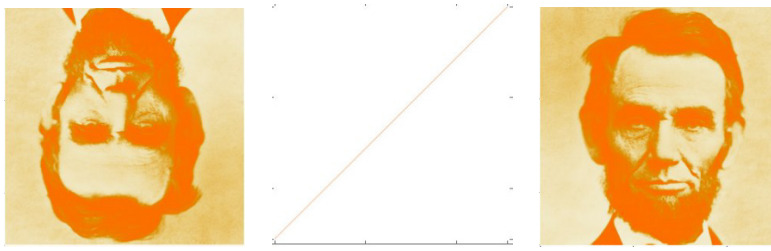

Figure 8.7: Multiplying the Lincoln matrix M by the reverse permutation matrix r (center). The product $r \cdot M$ flips Lincoln upside down (left), while $M \cdot r$ parts his hair on the other side of his head (right).

the main diagonal. For $n = 4$,

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Convince yourself that $AI = IA = A$, meaning that multiplication by the identity matrix commutes.

Note that each row and column of I contains exactly one non-zero element. Matrices with this property are called *permutation matrices*, because the non-zero element in position (i, j) can be interpreted as meaning that element i is in position j of a permutation. For example, the permutation $(2, 4, 3, 1)$ defines the permutation matrix:

$$P_{(2431)} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Observe that the identity matrix corresponds to the permutation $(1, 2, \ldots, n)$.

The key point here is that we can multiply A by the appropriate permutation matrix to rearrange the rows and columns, however we wish. Figure 8.7 shows what happens when we multiply our image by a "reverse" permutation matrix

r, where the ones lie along the minor diagonal. Because matrix multiplication is not generally commutative, we get different results for $A \cdot r$ and $r \cdot A$. Convince yourself why.

Rotating Points in Space

Multiplying something by the right matrix can have magical properties. We have seen how a set of n points in the plane (i.e. two dimensions) can be represented by an $(n \times 2)$-dimensional matrix S. Multiplying such points by the right matrix can yield natural geometric transformations.

The *rotation matrix* R_θ performs the transformation of rotating points about the origin through an angle of θ. In two dimensions, R_θ is defined as

$$R_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

In particular, after the appropriate multiplication/rotation, point (x, y) goes to

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R_\theta \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos(\theta) - y\sin(\theta) \\ x\sin(\theta) + y\cos(\theta) \end{bmatrix}$$

For $\theta = 180^\circ = \pi$ radians, $\cos(\theta) = -1$ and $\sin(\theta) = 0$, so this reduces to $(-x, -y)$, doing the right thing by putting the point in the opposing quadrant.

For our $(n \times 2)$-dimensional point matrix S, we can use the transpose function to orient the matrix appropriately. Check to confirm that

$$S' = (R_\theta S^T)^T$$

does exactly what we want to do.

Natural generalizations of R_θ exist to rotate points in arbitrary dimensions. Further, arbitrary sequences of successive transformations can be realized by multiplying chains of rotation, dilation, and reflection matrices, yielding a compact description of complex manipulations.

8.2.4 Identity Matrices and Inversion

Identity operations play a big role in algebraic structures. For numerical addition, zero is the identity element, since $0 + x = x + 0 = x$. The same role is played by one for multiplication, since $1 \cdot x = x \cdot 1 = x$.

In matrix multiplication, the identity element is the identity matrix, with all ones down the main diagonal. Multiplication by the identity matrix commutes, so $IA = AI = A$.

The *inverse* operation is about taking an element x down to its identity element. For numerical addition, the inverse of x is $(-x)$, because $x + (-x) = 0$. The inverse operation for multiplication is called *division*. We can invert a number by multiplying it by its reciprocal, since $x \cdot (1/x) = 1$.

People do not generally talk about dividing matrices. However, they very frequently go about inverting them. We say A^{-1} is the *multiplicative inverse*

Figure 8.8: The inverse of Lincoln does not look much like the man (left) but $M \cdot M^{-1}$ produces the identity matrix, modulo small non-zero terms due to numerical precision issues.

of matrix A if $A \cdot A^{-1} = I$, where I is the identity matrix. Inversion is an important special case of division, since $A \cdot A^{-1} = I$ implies $A^{-1} = I/A$. They are in fact equivalent operations, because $A/B = A \cdot B^{-1}$.

Figure 8.8 (left) shows the inverse of our Lincoln picture, which looks pretty much like random noise. But multiplying it by the image yields the thin main diagonal of the identity matrix, albeit superimposed on a background of numerical error. Floating point computations are inherently imprecise, and algorithms like inversion which perform repeated additions and multiplications often accumulate error in the process.

How can we compute the inverse of a matrix? A closed form exists for finding the inverse A^{-1} of a 2×2 matrix A, namely:

$$A^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

More generally, there is an approach to inverting matrices by solving a linear system using Gaussian elimination.

Observe that this closed form for inversion divides by zero whenever the products of the diagonals are equal, i.e. $ad = bc$. This tells us that such matrices are not invertible or *singular*, meaning no inverse exists. Just as we cannot divide numbers by zero, we cannot invert singular matrices.

The matrices we can invert are called *non-singular*, and life is better when our matrices have this property. The test of whether a matrix is invertible is whether its *determinant* is not zero. For 2×2 matrices, the determinant is the difference between the product of its diagonals, exactly the denominator in the inversion formula.

Further, the determinant is only defined for square matrices, so only square matricies are invertible. The cost of computing this determinant is $O(n^3)$, so it is expensive on large matricies, indeed as expensive as trying to invert the matrix itself using Gaussian elimination.

$$[A|I] = \left[\begin{array}{ccc|ccc} 6 & 4 & 1 & 1 & 0 & 0 \\ 10 & 7 & 2 & 0 & 1 & 0 \\ 5 & 3 & 1 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 1 & -2 \\ 5 & 3 & 1 & 0 & 0 & 1 \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & -1 & 1 \\ 0 & 1 & 0 & 0 & 1 & -2 \\ 5 & 3 & 1 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & -1 & 1 \\ 0 & 1 & 0 & 0 & 1 & -2 \\ 0 & 0 & 1 & -5 & 2 & 2 \end{array}\right]$$

$$\rightarrow A^{-1} = \left[\begin{array}{ccc} 1 & -1 & 1 \\ 0 & 1 & -2 \\ -5 & 2 & 2 \end{array}\right]$$

Figure 8.9: The inverse of a matrix can be computed by Gaussian elimination.

8.2.5 Matrix Inversion and Linear Systems

Linear equations are defined by the sum of variables weighted by constant coefficients:

$$y = c_0 + c_1 x_1 + c_2 x_2 + \ldots c_{m-1} x_{m-1}.$$

Thus the coefficients defining a system of n linear equations can be represented as an $n \times m$ matrix C. Here each row represents an equation, and each of the m columns the coefficients of a distinct variable.

We can neatly evaluate all n of these equations on a particular $m \times 1$ input vector X, by multiplying $C \cdot X$. The result will be an $n \times 1$ vector, reporting the value $f_i(X)$ for each of the n linear equations, $1 \leq i \leq n$. The special case here is the additive term c_0. For proper interpretation, the associated column in X should contain all ones.

If we generalize X to be an $m \times p$ matrix containing p distinct points, our product $C \cdot x$ results in an $n \times p$ matrix, evaluating every point against every equation in a single matrix multiplication.

But the primary operation on systems of n equations is to solve them, meaning to identify the X vector necessary to yield a target Y value for each equation. Give the $n \times 1$ vector of solution values Y and coefficient matrix C, we seek X such that $C \cdot X = Y$.

Matrix inversion can be used to solve linear systems. Multiplying both sides of $CX = Y$ by the inverse of C yields:

$$(C^{-1}C)X = C^{-1}Y \longrightarrow X = C^{-1}Y.$$

Thus the system of equations can be solved by inverting C and then multiplying C^{-1} by Y.

Gaussian elimination is another approach to solving linear systems, which I trust you have seen before. Recall that it solves the equations by performing row addition/subtraction operations to simplify the equation matrix C until it reduces to the identity matrix. This makes it trivial to read off the values of the

variables, since every equation has been reduced to the form $X_i = Y'_i$, where Y' is the result of applying these same row operations to the original target vector Y.

Computing the matrix inverse can be done in the same fashion, as shown in Figure 8.9. We perform row operations to simplify the coefficient matrix to the identity matrix I in order to create the inverse. I think of this as the algorithm of Dorian Gray: the coefficient matrix C beautifies to the identity matrix, while the target I ages into the inverse.[1]

Therefore we can use matrix inversion to solve linear systems, and linear system solvers to invert matrices. Thus the two problems are in some sense equivalent. Computing the inverse makes it cheap to evaluate multiple Y vectors for a given system C, by reducing it to a single matrix multiplication. But this can be done even more efficiently with LU-decomposition, discussed in Section 8.3.2. Gaussian elimination proves more numerically stable than inversion, and is generally the method of choice when solving linear systems.

8.2.6 Matrix Rank

A system of equations is properly *determined* when there are n linearly independent equations and n unknowns. For example, the linear system

$$2x_1 + 1x_2 = 5$$
$$3x_1 - 2x_2 = 4$$

is properly determined. The only solution is the point $(x_1 = 2, x_2 = 1)$.

In contrast, systems of equations are *underdetermined* if there are rows (equations) that can be expressed as linear combinations of other rows. The linear system

$$2x_1 + 1x_2 = 5$$
$$4x_1 + 2x_2 = 10$$

is underdetermined, because the second row is twice that of the first row. It should be clear that there is not enough information to solve an undetermined system of linear equations.

The *rank* of a matrix measures the number of linearly independent rows. An $n \times n$ matrix should be rank n for all operations to be properly defined on it.

The rank of the matrix can be computed by running Gaussian elimination. If it is underdetermined, then certain variables will disappear in the course of row-reduction operations. There is also a connection between underdetermined systems and singular matrices: recall that they were identified by having a determinant of zero. That is why the difference in the cross product here $(2 \cdot 2 - 4 \cdot 1)$ equals zero.

[1] In Oscar Wilde's novel *The Picture of Dorian Gray*, the protagonist remains beautiful, while his picture ages horribly over the years.

Feature matrices are often of lower rank than we might desire. Files of examples tend to contain duplicate entries, which would result in two rows of the matrix being identical. It is also quite possible for multiple columns to be equivalent: imagine each record as containing the height measured in both feet and meters, for example.

These things certainly happen, and are bad when they do. Certain algorithms on our Lincoln memorial image failed numerically. It turned out that our 512×512 image had a rank of only 508, so not all rows were linearly independent. To make it a full-rank matrix, you can add a small amount of random noise to each element, which will increase the rank without serious image distortion. This kludge might get your data to pass through an algorithm without a warning message, but it is indicative of numerical trouble to come.

Linear systems can be "almost" of lower rank, which results in a greater danger of precision loss due to numerical issues. This is formally captured by a matrix invariant called the *condition number*, which in the case of a linear system measures how sensitive the value of X is to small changes of Y in $Y = AX$.

Be aware of the vagaries of numerical computation when evaluating your results. For example, it is a good practice to compute AX for any purported solution X, and see how well AX *really* compares to Y. In theory the difference will be zero, but in practice you may be surprised how rough the calculation really is.

8.3 Factoring Matrices

Factoring matrix A into matrices B and C represents a particular aspect of division. We have seen that any non-singular matrix M has an inverse M^{-1}, so the identity matrix I can be factored as $I = MM^{-1}$. This proves that some matrices (like I) can be factored, and further that they might have many distinct factorizations. In this case, every possible non-singular M defines a different factorization.

Matrix factorization is an important abstraction in data science, leading to concise feature representations and ideas like topic modeling. It plays an important part in solving linear systems, through special factorizations like LU-decomposition.

Unfortunately, finding such factorizations is problematic. Factoring *integers* is a hard problem, although that complexity goes away when you are allowed floating point numbers. Factoring matrices proves harder: for a particular matrix, exact factorization may not be possible, particularly if we seek the factorization $M = XY$ where X and Y have prescribed dimensions.

8.3.1 Why Factor Feature Matrices?

Many important machine learning algorithms can be viewed in terms of factoring a matrix. Suppose we are given an $n \times m$ feature matrix A where, as per the

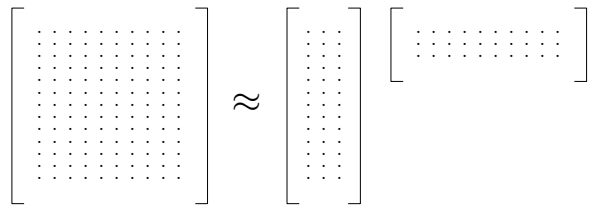

Figure 8.10: Factoring a feature matrix $A \approx BC$ yields B as a more concise representation of the items and C as a more concise representations of the features A^T.

usual convention, rows represent items/examples and columns represent features of the examples.

Now suppose that we can *factor* matrix A, meaning express it as the product $A \approx B \cdot C$, where B is an $n \times k$ matrix and C a $k \times m$ matrix. Presuming that $k < \min(n, m)$, as shown in Figure 8.10, this is a good thing for several reasons:

- *Together, B and C provide a compressed representation of matrix A*: Feature matrices are generally large, ungainly things to work with. Factorization provides a way to encode all the information of the large matrix into two smaller matrices, which together will be smaller than the original.
- *B serves as a smaller feature matrix on the items, replacing A*: The factor matrix B has n rows, just like the original matrix A. However, it has substantially fewer columns, since $k < m$. This means that "most" of the information in A is now encoded in B. Fewer columns mean a smaller matrix, and less parameters to fit in any model built using these new features. These more abstract features may also be of interest to other applications, as concise descriptions of the rows of the data set.
- *C^T serves as a small feature matrix on the* features*, replacing A^T*: Transposing the feature matrix turns columns/features into rows/items. The factor matrix C^T has m rows and k columns of properties representing them. In many cases, the m original "features" are worth modeling in their own right.

Consider a representative example from text analysis. Perhaps we want to represent n documents, each a tweet or other social message post, in terms of the vocabulary it uses. Each of our m features will correspond to a distinct vocabulary word, and $A[i, j]$ will record how often vocabulary word w_j (say, *cat*) appeared in message number i. The working vocabulary in English is large with a long tail, so perhaps we can restrict it to the $m = 50,000$ most frequently used words. Most messages will be short, with no more than a few hundred words. Thus our feature matrix A will be very sparse, riddled with a huge number of zeros.

Now suppose that we can factor $A = BC$, where the inner dimension k is relatively small. Say $k = 100$. Now each post will be represented by a row of

B containing only a hundred numbers, instead of the full 50,000. This makes it much easier to compare the texts for similarity in a meaningful way. These k dimensions can be thought of as analogous to the "topics" in the documents, so all the posts about sports should light up a different set of topics than those about relationships.

The matrix C^T can now be thought of as containing a feature vector for each of the vocabulary words. This is interesting. We would expect words that apply in similar contexts to have similar topic vectors. Color words like *yellow* and *red* are likely to look fairly similar in topic space, while *baseball* and *sex* should have quite distant relationships.

Note that this word–topic matrix is potentially useful in any problem seeking to use language as features. The connection with social message posts is largely gone, so it would be applicable to other domains like books and news. Indeed, such compressed *word embeddings* prove a very powerful tool in natural language processing (NLP), as will be discussed in Section 11.6.3.

8.3.2 LU Decomposition and Determinants

LU decomposition is a particular matrix factorization which factors a square matrix A into lower and upper triangular matrices L and U, such that $A = L \cdot U$.

A matrix is *triangular* if it contains all zero terms either above or below the main diagonal. The lower triangular matrix L has all non-zero terms below the main diagonal. The other factor, U, is the upper triangular matrix. Since the main diagonal of L consists of all ones, we can pack the entire decomposition into the same space as the original $n \times n$ matrix.

The primary value of LU decomposition is that it proves useful in solving linear systems $AX = Y$, particularly when solving multiple problems with the same A but different Y. The matrix L is what results from clearing out all of the values above the main diagonal, via Gaussian elimination. Once in this triangular form, the remaining equations can be directly simplified. The matrix U reflects what row operations have occurred in the course of building L. Simplifying U and applying L to Y requires less work than solving A from scratch.

The other importance of LU decomposition is in yielding an algorithm to compute the determinant of a matrix. The *determinant* of A is the product of the main diagonal elements of U. As we have seen, a determinant of zero means the matrix is not of full rank.

Figure 8.11 illustrates the LU decomposition of the Lincoln memorial. There is a distinct texture visible to the two triangular matrices. This particular LU decomposition function (in Mathematica) took advantage of the fact that the equations in a system can be permuted with no loss of information. The same does not hold true for images, but we see accurate reconstructions of the white columns of the memorial, albeit out of position.

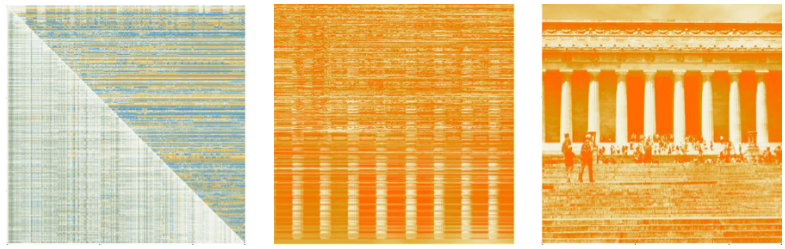

Figure 8.11: The LU decomposition of the Lincoln memorial (left), with the product $L \cdot U$ (center). The rows of the LU matrix were permuted during calculation, but when properly ordered fully reconstructed the image (right).

8.4 Eigenvalues and Eigenvectors

Multiplying a vector U by a square matrix A can have the same effect as multiplying it by a scalar l. Consider this pair of examples. Indeed, check them out by hand:

$$\begin{bmatrix} -5 & 2 \\ 2 & -2 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ -1 \end{bmatrix} = -6 \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

$$\begin{bmatrix} -5 & 2 \\ 2 & -2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = -1 \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Both of these equalities feature products with the same 2×1 vector U on the left as on the right. On one side U is multiplied by a matrix A, and on the other by a scalar λ. In cases like this, when $AU = \lambda U$, we say that λ is an *eigenvalue* of matrix A, and U is its associated *eigenvector*.

Such eigenvector–eigenvalue pairs are a curious thing. That the scalar λ can do the same thing to U as the entire matrix A tells us that they must be special. Together, the eigenvector U and eigenvalue λ must encode a lot of information about A.

Further, there are generally multiple such eigenvector–eigenvalue pairs for any matrix. Note that the second example above works on the same matrix A, but yields a different U and λ.

8.4.1 Properties of Eigenvalues

The theory of eigenvalues gets us deeper into the thicket of linear algebra than I am prepared to do in this text. Generally speaking, however, we can summarize the properties that will prove important to us:

- Each eigenvalue has an associated eigenvector. They always come in pairs.
- There are, in general, n eigenvector–eigenvalue pairs for every full rank $n \times n$ matrix.

- Every pair of eigenvectors of a symmetric matrix are mutually *orthogonal*, the same way that the x and y-axes in the plane are orthogonal. Two vectors are orthogonal if their dot product is zero. Observe that $(0,1) \cdot (1,0) = 0$, as does $(2,-1) \cdot (1,2) = 0$ from the previous example.
- The upshot from this is that eigenvectors can play the role of dimensions or *bases* in some n-dimensional space. This opens up many geometric interpretations of matrices. In particular, any matrix can be encoded where each eigenvalue represents the magnitude of its associated eigenvector.

8.4.2 Computing Eigenvalues

The n distinct eigenvalues of a rank-n matrix can be found by factoring its *characteristic equation*. Start from the defining equality $AU = \lambda U$. Convince yourself that this remains unchanged when we multiply by the identity matrix I, so

$$AU = \lambda IU \rightarrow (A - \lambda I)U = 0.$$

For our example matrix, we get

$$A - \lambda I = \begin{bmatrix} -5-\lambda & 2 \\ 2 & -2-\lambda \end{bmatrix}$$

Note that our equality $(A - \lambda I)U = 0$ remains true if we multiply vector U by any scalar value c. This implies that there are an infinite number of solutions, and hence the linear system must be underdetermined.

In such a situation, the determinant of the matrix must be zero. With a 2×2 matrix, the determinant is just the cross product $ad - bc$, so

$$(-5-\lambda)(-2-\lambda) - 2 \cdot 2 = \lambda^2 + 7\lambda + 6 = 0.$$

Solving for λ with the quadratic formula yields $\lambda = -1$ and $\lambda = -6$. More generally, the determinant $|A - \lambda I|$ is a polynomial of degree n, and hence the roots of this characteristic equation define the eigenvalues of A.

The vector associated with any given eigenvalue can be computed by solving a linear system. As per our example, we know that

$$\begin{bmatrix} -5 & 2 \\ 2 & -2 \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \lambda \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

for any eigenvalue λ and associated eigenvector $U = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$. Once we fix the value of λ, we have a system of n equations and n unknowns, and thus can solve for the values of U. For $\lambda = -1$,

$$\begin{aligned} -5u_1 + 2u_2 = -1u_1 &\longrightarrow -4u_1 + 2u_2 = 0 \\ 2u_1 + -2u_2 = -1u_2 &\longrightarrow 2u_1 + -1u_2 = 0 \end{aligned}$$

which has the solution $u_1 = 1$, $u_2 = 2$, yielding the associated eigenvector.

For $\lambda = -6$, we get

$$-5u_1 + 2u_2 = -6u_1 \longrightarrow 1u_1 + 2u_2 = 0$$
$$2u_1 + -2u_2 = -6u_2 \longrightarrow 2u_1 + 4u_2 = 0$$

This system is underdetermined, so $u_1 = 2$, $u_2 = -1$ and any constant multiple of this qualifies as an eigenvector. This makes sense: because U is on both sides of the equality $AU = \lambda U$, for any constant c, the vector $U' = c \cdot U$ equally satisfies the definition.

Faster algorithms for eigenvalue/vector computations are based on a matrix factorization approach called *QR decomposition*. Other algorithms try to avoid solving the full linear system. For example, an alternate approach repeatedly uses $U' = (AU)/\lambda$ to compute better and better approximations to U until it converges. When conditions are right, this can be much faster than solving the full linear system.

The largest eigenvalues and their associated vectors are, generally speaking, more important that the rest. Why? Because they make a larger contribution to approximating the matrix A. Thus high-performance linear algebra systems use special routines for finding the k largest (and smallest) eigenvalues and then iterative methods to reconstruct the vectors for each.

8.5 Eigenvalue Decomposition

Any $n \times n$ symmetric matrix M can be decomposed into the sum of its n eigenvector products. We call the n eigenpairs (λ_i, U_i), for $1 \leq i \leq n$. By convention we sort by size, so $\lambda_i \geq \lambda_{i-i}$ for all i.

Since each eigenvector U_i is an $n \times 1$ matrix, multiplying it by its transpose yields an $n \times n$ matrix product, $U_i {U_i}^T$. This has exactly the same dimensions as the original matrix M. We can compute the linear combination of these matrices weighted by its corresponding eigenvalue. In fact, this reconstructs the original matrix, since:

$$M = \sum_{i=1}^{n} \lambda_i U_i {U_i}^T$$

This result holds only for symmetric matrices, so we cannot use it to encode our image. But covariance matrices are always symmetric, and they encode the basic features of each row and column of the matrix.

Thus the covariance matrix can be represented by its eigenvalue decomposition. This takes slightly more space than the initial matrix: n eigenvectors of length n, plus n eigenvalues vs. the $n(n+1)/2$ elements in the upper triangle of the symmetric matrix plus main diagonal.

However, by using only the vectors associated with the largest eigenvalues we get a good approximation of the matrix. The smaller dimensions contribute very little to the matrix values, and so they can be excluded with little resulting error. This dimension reduction method is very useful to produce smaller, more effective feature sets.

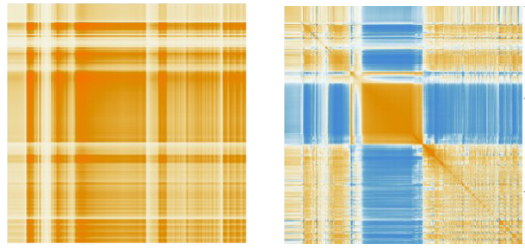

Figure 8.12: The Lincoln memorial's biggest eigenvector suffices to capture much of the detail of its covariance matrix.

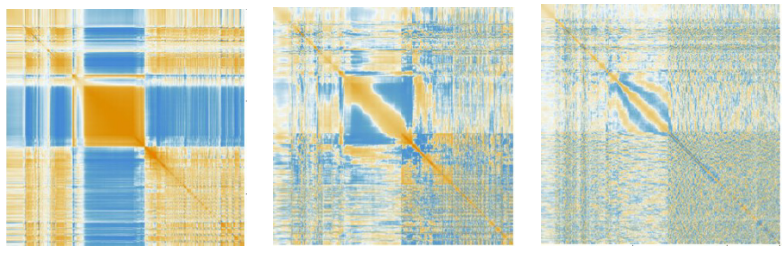

Figure 8.13: Error in reconstructing the Lincoln memorial from the one, five, and fifty largest eigenvectors.

Figure 8.12 (left) shows the reconstruction of the Lincoln memorial's covariance matrix M from its single largest eigenvector, i.e. $U_1 \cdot U_1^T$, along with its associated error matrix $M - U_1 \cdot U_1^T$. Even a single eigenvector does a very respectable job at reconstruction, restoring features like the large central block.

The plot in Figure 8.12 (right) shows that the errors occur in patchy regions, because more subtle detail requires additional vectors to encode. Figure 8.13 shows the error plot when using the one, five, and fifty largest eigenvectors. The error regions get smaller as we reconstruct finer detail, and the magnitude of the errors smaller. Realize that even fifty eigenvectors is less than 10% of the 512 necessary to restore a perfect matrix, but this suffices for a very good approximation.

8.5.1 Singular Value Decomposition

Eigenvalue decomposition is a very good thing. But it only works on symmetric matrices. *Singular value decomposition* is a more general matrix factorization approach, that similarly reduces a matrix to the sum of other matrices defined by vectors.

The *singular value decomposition* of an $n \times m$ real matrix M factors it into three matrices U, D, and V, with dimensions $n \times n$, $n \times m$, and $m \times m$ respec-

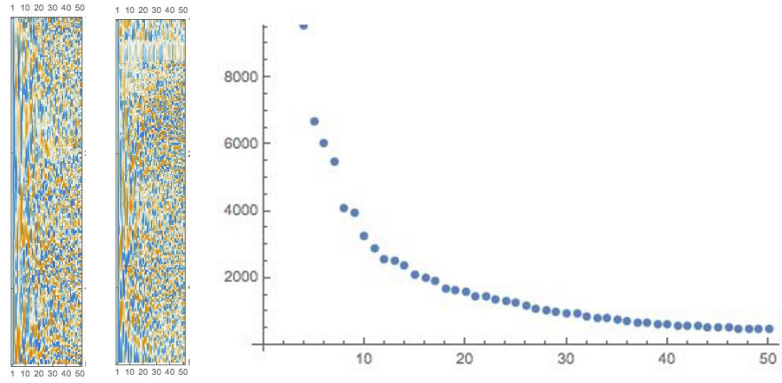

Figure 8.14: Singular value matrices in the decomposition of Lincoln, for 50 singular values.

tively. This factorization is of the form[2]

$$M = UDV^T$$

The center matrix D has the property that it is a *diagonal* matrix, meaning all non-zero values lie on the main diagonal like the identity matrix I.

Don't worry about how we find this factorization. Instead, let's concentrate on what it means. The product $U \cdot D$ has the effect of multiplying $U[i,j]$ by $D[j,j]$, because all terms of D are zero except along the main diagonal. Thus D can be interpreted as measuring the relative importance of each column of U, or through $D \cdot V^T$, the importance of each row of V^T. These weight values of D are called the *singular values* of M.

Let X and Y be vectors, of dimensionality $n \times 1$ and $1 \times m$, respectively. The matrix *outer product* $P = X \bigotimes Y$ is the $n \times m$ matrix where $P[j,k] = X[j]Y[k]$. The traditional matrix multiplication $C = A \cdot B$ can be expressed as the sum of these outer products, namely:

$$C = A \cdot B = \sum_k A_k \bigotimes B_k^T$$

where A_k is the vector defined by the kth column of A, and B_k^T is the vector defined by the kth row of B.

Putting this together, matrix M can be expressed as the sum of outer products of vectors resulting from the singular value decomposition, namely $(UD)_k$ and $(V^T)_k$ for $1 \leq k \leq m$. Further, the singular values D define how much contribution each outer product makes to M, so it suffices to take only the vectors associated with the largest singular values to get an approximation to M.

[2]Should M contain convex numbers, then this generalizes to $M = UDV^*$, where V^* means the *conjugate transpose* of V.

Figure 8.15: Lincoln's face reconstructed from 5 (left) and 50 (center) singular values, with the error for $k = 50$ (right).

Figure 8.14 (left) presents the vectors associated with the first fifty singular values of Lincoln's face. If you look carefully, you can see how the first five to ten vectors are considerably more blocky than subsequent ones, indicating that the early vectors rough out the basic structure of the matrix, with subsequent vectors adding greater detail. Figure 8.14 (right) shows how the mean squared error between the matrix and its reconstruction shrinks as we add additional vectors.

These effects become even more vivid when we look at the reconstructed images themselves. Figure 8.15 (left) shows Lincoln's face with only the five strongest vectors, which is less than 1% of what is available for perfect reconstruction. But even at this point you could pick him out of a police lineup. Figure 8.15 (center) demonstrates the greater detail when we include fifty vectors. This looks as good as the raw image in print, although the error plot (Figure 8.15 (right)) highlights the missing detail.

Take-Home Lesson: Singular value decomposition (SVD) is a powerful technique to reduce the dimensionality of any feature matrix.

8.5.2 Principal Components Analysis

Principal components analysis (PCA) is a closely related technique for reducing the dimensionality of data sets. Like SVD, we will define vectors to represent the data set. Like SVD, we will order them by successive importance, so we can reconstruct an approximate representation using few components. PCA and SVD are so closely related as to be indistinguishable for our purposes. They do the same thing in the same way, but coming from different directions.

The principal components define the axes of an ellipsoid best fitting the points. The origin of this set of axes is the centroid of the points. PCA starts by identifying the direction to project the points on to so as to explain the maximum amount of variance. This is the line through the centroid that, in some sense, best fits the points, making it analogous to linear regression. We can then project each point onto this line, with this point of intersection defining a particular position on the line relative to the centroid. These projected

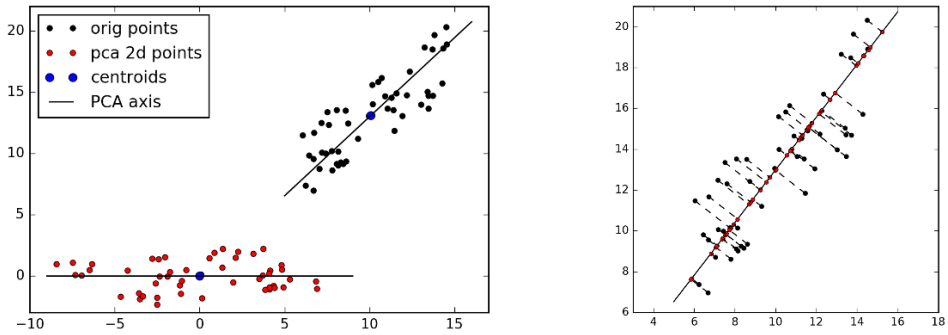

Figure 8.16: PCA projects the black points onto orthogonal axis, rotated to yield the alternate representation in red (left). The values of each component are given by projecting each point onto the appropriate axis (right).

positions now define the first dimension (or principal component) of our new representation, as shown in Figure 8.16.

For each subsequent component, we seek the line l_k, which is orthogonal to all previous lines and explains the largest amount of the remaining variance. That each dimension is orthogonal to each other means they act like coordinate axes, establishing the connection to eigenvectors. Each subsequent dimension is progressively less important than the ones before it, because we chose the most promising directions first. Later components contribute only progressively finer detail, and hence we can stop once this is small enough.

Suppose that dimensions x and y are virtually identical. We would expect that the regression line will project down to $y = x$ on these two dimensions, so they could then largely be replaced by a single dimension. PCA constructs new dimensions as linear combinations of the original ones, collapsing those which are highly correlated into a lower-dimensional space. *Statistical factor analysis* is a technique which identifies the most important orthogonal dimensions (as measured by correlation) that explain the bulk of the variance.

Relatively few components suffice to capture the basic structure of the point set. The residual that remains is likely to be noise, and is often better off removed from the data. After dimension reduction via PCA (or SVD), we should end up with cleaner data, not merely a smaller number of dimensions.

Take-Home Lesson: PCA and SVD are essentially two different approaches to computing the same thing. They should serve equally well as low-dimensional approximations of a feature matrix.

8.6 War Story: The Human Factors

I first came to be amazed by the power of dimension reduction methods like PCA and SVD in the course of our analysis of historical figures for our book *Who's Bigger*. Recall (from the war story of Section 4.7) how we analyzed the structure and content of Wikipedia, ultimately extracting a half-dozen features like PageRank and article length for each of the 800,000+ articles about people in the English edition. This reduced each of these people to a six-dimensional feature vector, which we would analyze to judge their relative significance.

But things proved not as straightforward as we thought. Wildly different people were ranked highest by each particular variable. It wasn't clear how to interpret them.

"There is so much variance and random noise in our features," my co-author Charles observed. "Let's identify the major factors underlying these observed variables that really show what is going on."

Charles' solution was *factor analysis*, which is a variant of PCA, which is in turn a variant of SVD. All of these techniques compress feature matrices into a smaller set of variables or factors, with the goal that these factors explain most of the variance in the full feature matrix. We expected factor analysis would extract a single underlying factor defining individual significance. But instead, our input variables yielded *two* independent factors explaining the data. Both explained roughly equal proportions of the variance (31% and 28%), meaning that these latent variables were approximately of equal importance. But the cool thing is what these factors showed.

Factors (or singular vectors, or principle components) are just linear combinations of the original input features. They don't come with names attached to them, so usually you would just describe them as Factor 1 and Factor 2. But our two factors were so distinctive that Charles gave them the names *gravitas* and *celebrity*, and you can see why in Figure 8.17.

Our *gravitas* factor largely comes from (or "loads on," in statistical parlance) the two forms of PageRank. Gravitas seems to accurately capture notions of achievement-based recognition. In contrast, the *celebrity* factor loads more strongly on page hits, revisions, and article length. The celebrity factor better captures the popular (some might say vulgar) notions of reputation. The wattage of singers, actors, and other entertainers are better measured by celebrity then gravitas.

To get a feel for the distinction between gravitas and celebrity, compare our highest ranked figures for each factor, in Figure 8.17. The high gravitas figures on the left are clearly old-fashioned heavyweights, people of stature and accomplishment. They are philosophers, kings, and statesmen. Those names listed in Figure 8.17 (right) are such complete celebrities that the top four walk this earth with only one name. They are professional wrestlers, actors, and singers. It is quite telling that the only two figures here showing any gravitas on our celebrity-gravitas meter are *Britney Spears (1981–)* [566] and *Michael Jackson (1958–2009)* [136], both among the Platonic ideals of modern celebrity.

I find it amazing that these unsupervised methods were able to tease apart

Highest Gravitas Ranking

Person	Grav.	Sig	Celeb/Grav
Napoleon	8	2	C G
Carl Linnaeus	13	31	C G
Plato	23	25	C G
Aristotle	27	8	C G
F. D. Roosevelt	30	43	C G
Plutarch	32	258	C G
Charles II	33	78	C G
Elizabeth II	35	132	C G
Queen Victoria	38	16	C G
William Shakespeare	42	4	C G
Pliny the Elder	43	212	C G
Tacitus	52	300	C G
Herodotus	58	123	C G
Charles V	61	84	C G
George V	64	235	C G

Highest Celebrity Ranking

Person	Celeb.	Sig	Celeb/Grav
The Undertaker	2	2172	C G
Vijay	8	4456	C G
Edge	10	2603	C G
Kane	13	2229	C G
John Cena	16	2277	C G
Beyoncé Knowles	19	1519	C G
Triple H	26	1596	C G
Rey Mysterio	36	2740	C G
Britney Spears	37	689	C G
Ann Coulter	45	3376	C G
Jesse McCartney	48	4236	C G
Roger Federer	57	743	C G
Ashley Tisdale	60	4445	C G
Michael Jackson	75	180	C G
Dwayne Johnson	78	1446	C G

Figure 8.17: The gravitas and celebrity factors do an excellent job partitioning two types of famous people.

two distinct types of fame, without any labeled training examples or even a preconception of what they were looking for. The factors/vectors/components simply reflected what was there in the data to be found.

This celebrity-gravitas continuum serves as an instructive example of the power of dimension reduction methods. All the factors/vectors/components all must, by definition, be orthogonal to each other. This means that they each measure different things, in a way that two correlated input variables do not. It pays to do some exploratory data analysis on your main components to try to figure out what they really mean, in the context of your application. The factors are yours to name as you wish, just like a cat or a dog, so choose names you will be happy to live with.

8.7 Chapter Notes

There are many popular textbooks providing introductions to linear algebra, including [LLM15, Str11, Tuc88]. Klein [Kle13] presents an interesting introduction to linear algebra for computer science, with an emphasis on programming and applications like coding theory and computer graphics.

8.8 Exercises

Basic Linear Algebra

8-1. *[3]* Give a pair of square matrices A and B such that:

(a) $AB = BA$ (it commutes).

(b) $AB \neq BA$ (does not commute).

In general, matrix multiplication is not commutative.

8-2. *[3]* Prove that matrix addition is associative, i.e. that $(A+B)+C = A+(B+C)$ for compatible matrices A, B and C.

8-3. *[5]* Prove that matrix multiplication is associative, i.e. that $(AB)C = A(BC)$ for compatible matrices A, B and C.

8-4. *[3]* Prove that $AB = BA$, if A and B are diagonal matrices of the same order.

8-5. *[5]* Prove that if $AC = CA$ and $BC = CB$, then

$$C(AB + BA) = (AB + BA)C.$$

8-6. *[3]* Are the matrices MM^T and M^TM square and symmetric? Explain.

8-7. *[5]* Prove that $(A^{-1})^{-1} = A$.

8-8. *[5]* Prove that $(A^T)^{-1} = (A^{-1})^T$ for any non-singular matrix A.

8-9. *[5]* Is the LU factorization of a matrix unique? Justify your answer.

8-10. *[3]* Explain how to solve the matrix equation $Ax = b$?

8-11. *[5]* Show that if M is a square matrix which is not invertible, then either L or U in the LU-decomposition $M = L \cdot U$ has a zero in its diagonal.

Eigenvalues and Eigenvectors

8-12. *[3]* Let $M = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}$. Find all eigenvalues of M. Does M have two linearly independent eigenvectors?

8-13. *[3]* Prove that the eigenvalues of A and A^T are identical.

8-14. *[3]* Prove that the eigenvalues of a diagonal matrix are equal to the diagonal elements.

8-15. *[5]* Suppose that matrix A has an eigenvector v with eigenvalue λ. Show that v is also an eigenvector for A^2, and find the corresponding eigenvalue. How about for A^k, for $2 \leq k \leq n$?

8-16. *[5]* Suppose that A is an invertible matrix with eigenvector v. Show that v is also an eigenvector for A^{-1} .

8-17. *[8]* Show that the eigenvalues of MM^T are the same as that of M^TM. Are their eigenvectors also the same?

Implementation Projects

8-18. *[5]* Compare the speed of a library function for matrix multiplication to your own implementation of the nested loops algorithm.

- How much faster is the library on products of random $n \times n$ matricies, as a function of n as n gets large?
- What about the product of an $n \times m$ and $m \times n$ matrix, where $n \ll m$?
- By how much do you improve the performance of your implementation to calculate $C = A \cdot B$ by first transposing B internally, so all dot products are computed along rows of the matrices to improve cache performance?

8-19. *[5]* Implement Gaussian elimination for solving systems of equations, $C \cdot X = Y$. Compare your implementation against a popular library routine for:

(a) *Speed*: How does the run time compare, for both dense and sparse coefficient matrices?

(b) *Accuracy*: What are the size of the numerical residuals $CX - Y$, particularly as the condition number of the matrix increases.

(c) *Stability*: Does your program crash on a singular matrix? What about almost singular matrices, created by adding a little random noise to a singular matrix?

Interview Questions

8-20. *[5]* Why is vectorization considered a powerful method for optimizing numerical code?

8-21. *[3]* What is singular value decomposition? What is a singular value? And what is a singular vector?

8-22. *[5]* Explain the difference between "long" and "wide" format data. When might each arise in practice?

Kaggle Challenges

8-23. Tell what someone is looking at from analysis of their brain waves.
`https://www.kaggle.com/c/decoding-the-human-brain`

8-24. Decide whether a particular student will answer a given question correctly.
`https://www.kaggle.com/c/WhatDoYouKnow`

8-25. Identify mobile phone users from accelerometer data.
`https://www.kaggle.com/c/accelerometer-biometric-competition`