Chapter 2. Linear Algebra

Now that we have spent a whole chapter acquiring data in some format or another, we will most likely end up viewing the data (in our minds) in the form of spreadsheet. It is natural to envision the names of each column going across from left to right (age, address, ID number, etc.), with each row representing a unique record or data point. Much of data science comes down to this exact formulation. What we are seeking to find is a relationship between any number of columns of interest (which we will call *variables*) and any number of columns that indicate a measurable outcome (which we will call *responses*).

Typically, we use the letter x to denote the variables, and y for the responses. Likewise, the responses can be designated by a matrix **Y** that has a number of columns p and must have the same number of rows m as **X** does. Note that in many cases, there is only one dimension of response variable such that $p = 1$. However, it helps to generalize linear algebra problems to arbitrary dimensions.

In general, the main idea behind linear algebra is to find a relationship between **X** and **Y**. The simplest of these is to ask whether we can multiply **X** by a new matrix of yet-to-be-determined values **W**, such that the result is exactly (or nearly) equal to **Y**. An example of **XW** = **Y** looks like this:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} = \begin{pmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,p} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,p} \end{pmatrix}$$

Keep in mind that as the equation is drawn, the sizes of the matrices look similar. This can be misleading, because in most cases the number of data points m is large, perhaps in the millions or billions, while the number of columns n, p for the respective **X** and **Y** matrices is usually much smaller (from tens to hundreds). You will then take notice that regardless of the size of m (e.g., 100,000), the size of the **W** matrix is independent of m; its size is $n \times p$ (e.g., 10 × 10). And this is the heart of linear algebra: that we can explain the contents of extremely large data structures such as **X** and **Y** by using a much more compact data structure **W**. The rules of linear algebra enable us to express any particular value of **Y** in terms of a row of **X** and column of **W**. For example the value of $y_{1,1}$ is written out as follows:

$$y_{1,1} = x_{1,1}\omega_{1,1} + x_{1,2}\omega_{2,1} + \ldots + x_{1,n}\omega_{n,1}$$

In the rest of this chapter, we will work out the rules and operations of linear algebra, and in the final section show the solution to the linear system **XW** = **Y**. More advanced topics in data science such as those presented in Chapters 4 and 5 , will rely heavily on the use of linear algebra.

Building Vectors and Matrices

Despite any formal definitions, a *vector* is just a one-dimensional array of a defined length. Many examples may come to mind. You might have an array of integers representing the counts per day of a web metric. Maybe you have a large number of "features" in an array that will be used for input into a machine-learning routine. Or perhaps you are keeping track of geometric coordinates such as *x* and *y*, and you might create an array for each pair `[x,y]`. While we can argue the philosophical meaning of what a vector is (i.e., an element of vector space with magnitude and direction), as long as you are consistent in how you define your vectors throughout the problem you are solving, then all the mathematical formulations will work beautifully, without any concern for the topic of study.

In general, a vector **x** has the following form, comprising *n* components:

$$\mathbf{x} = (x_1\, x_2\, \dots x_n)$$

Likewise, a matrix **A** is just a two-dimensional array with *m* rows and *n* columns:

$$\mathbf{A} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

A vector can also be represented in matrix notation as a column vector:

$$\mathbf{X} = \begin{pmatrix} x_{1,1} \\ x_{2,1} \\ \vdots \\ x_{m,1} \end{pmatrix}$$

WARNING

We use bold lowercase letters to represent vectors and use bold uppercase letters to represent matrices. Note that the vector **x** can also be represented as a column of the matrix **X**.

In practice, vectors and matrices are useful to data scientists. A common example is a dataset in which (feature) vectors are stacked on top of each other, and usually the number of rows *m* is much larger than the number of columns *n*. In essence, this type of data structure is really a list of vectors, but putting them in matrix form enables efficient calculation of all sorts of linear algebra quantities. Another type of matrix encountered in data science is one in which the components represent a relationship between the variables, such as a covariance or correlation matrix.

Array Storage

The Apache Commons Math library offers several options for creating vectors and matrices of real numbers with the respective `RealVector` and `RealMatrix` classes. Three of the most useful constructor types allocate an empty instance of known dimension, create an instance from an array of values, and create an instance by deep copying an existing instance, respectively. To instantiate an empty, *n*-dimensional vector of type `RealVector`, use the `ArrayRealVector` class with an integer size:

```
int size = 3;
RealVector vector = new ArrayRealVector(size);
```

If you already have an array of values, a vector can be created with that array as a constructor argument:

```
double[] data = {1.0, 2.2, 4.5};
RealVector vector = new ArrayRealVector(data);
```

A new vector can also be created by deep copying an existing vector into a new instance:

```
RealVector vector = new ArrayRealVector(realVector);
```

To set a default value for all elements of a vector, include that value in the constructor along with the size:

```
int size = 3;
double defaultValue = 1.0;
RealVector vector = new ArrayRealVector(size, defaultValue);
```

A similar set of constructors follows for instantiating matrices, an empty matrix of known dimensions is instantiated with the following:

```
int rowDimension = 10;
int colDimension = 20;
RealMatrix matrix = new Array2DRowRealMatrix(rowDimension, colDimension);
```

Or if you already have a two-dimensional array of doubles, you can pass it to the constructor:

```
double[][] data = {{1.0, 2.2, 3.3}, {2.2, 6.2, 6.3}, {3.3, 6.3, 5.1}};
RealMatrix matrix = new Array2DRowRealMatrix(data);
```

Although there is no method for setting the entire matrix to a default value (as there is with a vector), instantiating a new matrix sets all elements to zero, so we can easily add a value to each element afterward:

```
int rowDimension = 10;
int colDimension = 20;
double defaultValue = 1.0;
RealMatrix matrix = new Array2DRowRealMatrix(rowDimension, colDimension);
matrix.scalarAdd(defaultValue);
```

Making a deep copy of a matrix may be performed via the `RealMatrix.copy()` method:

```
/* deep copy contents of matrix */
RealMatrix anotherMatrix = matrix.copy();
```

Block Storage

For large matrices with dimensions greater than 50, it is recommended to use block storage with the `BlockRealMatrix` class. Block storage is an alternative to the two-dimensional array storage discussed in the previous section. In this case, a large matrix is subdivided into smaller blocks of data that are easier to cache and therefore easier to operate on. To allocate space for a matrix, use the following constructor:

```
RealMatrix blockMatrix = new BlockRealMatrix(50, 50);
```

Or if you already have the data in a 2D array, use this constructor:

```
double[][] data = ;
RealMatrix blockMatrix = new BlockRealMatrix(data);
```

Map Storage

When a large vector or matrix is almost entirely zeros, it is termed *sparse*. Because it is not efficient to store all those zeros, only the positions and values of the nonzero elements are stored. Behind the scenes, this is easily achieved by storing the values in a `HashMap`. To create a sparse vector of known dimension, use the following:

```
int dim = 10000;
RealVector sparseVector = new OpenMapRealVector(dim);
```

And to create a sparse matrix, just add another dimension:

```
int rows = 10000;
int cols = 10000;
RealMatrix sparseMatrix = new OpenMapRealMatrix(rows, cols);
```

Accessing Elements

Regardless of the type of storage backing the vector or matrix, the methods for assigning values and later retrieving them are equivalent.

CAUTION

Although the linear algebra theory presented in this book uses an index starting at 1, Java uses a 0-based index system. Keep this in mind as you translate algorithms from theory to code and in particular, when setting and getting values.

Setting and getting values uses the `setEntry(int index, double value)` and `getEntry(int index)` methods:

```
/* set the first value of v */
vector.setEntry(0, 1.2)
/* and get it */
double val = vector.getEntry(0);
```

To set all the values for a vector, use the `set(double value)` method:

```
/* zero the vector */
vector.set(0);
```

However, if **v** is a sparse vector, there is no point to setting all the values. In sparse algebra, missing values are assumed to be zero. Instead, just use `setEntry` to set only the values that are nonzero. To retrieve all the values of an existing vector as an array of doubles, use the `toArray()` method:

```
double[] vals = vector.toArray();
```

Similar setting and getting is provided for matrices, regardless of storage. Use the `setEntry(int row, int column, double value)` and `getEntry(int row, int column)` methods:

```
/* set first row, 3 column to 3.14 */
matrix.setEntry(0, 2, 3.14);
/* and get it */
double val = matrix.getEntry(0, 2);
```

Unlike the vector classes, there is no `set()` method to set all the values of a matrix to one value. As long as the matrix has all entries set to 0, as is the case for a newly constructed matrix, you can set all the entries to one value by adding a constant with code like this:

```
/* for an existing new matrix */
matrix.scalarAdd(defaultValue);
```

Just as with sparse vectors, setting all the values to 0 for each *i,j* pair of a sparse matrix is not useful.

To get all the values of a matrix in the form of an array of doubles, use the `getData()` method:

```
double[][] matrixData = matrix.getData();
```

Working with Submatrices

We often need to work with only a specific part of a matrix or want to include a smaller matrix in a larger one. The `RealMatrix` class contains several useful methods for dealing with these common cases. For an existing matrix, there are two ways to create a submatrix from it. The first method selects a rectangular region from the source matrix and uses those entries to create a new matrix. The selected rectangular region is defined by the point of origin, the upper-left corner of the source matrix, and the lower-right corner defining the area that should be included. It is invoked as `RealMatrix.getSubMatrix(int startRow, int endRow, int startColumn, int endColumn)` and returns a `RealMatrix` object with dimensions and values determined by the selection. Note that the `endRow` and `endColumn` values are inclusive.

```
double[][] data = {{1,2,3},{4,5,6},{7,8,9}};
RealMatrix m = new Array2DRowRealMatrix(data);
int startRow = 0;
int endRow = 1;
int startColumn = 1;
int endColumn = 2;
RealMatrix subM = m.getSubMatrix(startRow, endRow, startColumn, endColumn);
// {{2,3},{5,6}}
```

We can also get specific rows and specific columns of a matrix. This is achieved by creating an array of integers designating the row and column indices we wish to keep. The method then takes both of these arrays as `RealMatrix.getSubMatrix(int[] selectedRows, int[] selectedColumns)`. The three use cases are then as follows:

```
/* get selected rows and all columns */
int[] selectedRows = {0, 2};
int[] selectedCols = {0, 1, 2};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{1,2,3},{7,8,9}}

/* get all rows and selected columns */
int[] selectedRows = {0, 1, 2};
int[] selectedCols = {0, 2};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{1,3},{4,6},{7,9}}

/* get selected rows and selected columns */
int[] selectedRows = {0, 2};
```

```
int[] selectedCols = {1};
RealMatrix subM = m.getSubMatrix(selectedRows, selectedColumns);
// {{2},{8}}
```

We can also create a matrix in parts by setting the values of a submatrix. We do this by adding a double array of data to an existing matrix at the coordinates specified by row and column in `RealMatrix.setSubMatrix(double[][] subMatrix, int row, int column)`:

```
double[][] newData = {{-3, -2}, {-1, 0}};
int row = 0;
int column = 0;
m.setSubMatrix(newData, row, column);
// {{-3,-2,3},{-1,0,6},{7,8,9}}
```

Randomization

In learning algorithms, we often want to set all the values of a matrix (or vectors) to random numbers. We can choose the statistical distribution that implements the `AbstractRealDistribution` interface or just go with the easy constructor, which picks random numbers between –1 and 1. We can pass in an existing matrix or vector to fill in the values, or create new instances:

```
public class RandomizedMatrix {

    private AbstractRealDistribution distribution;

    public RandomizedMatrix(AbstractRealDistribution distribution, long seed) {
        this.distribution = distribution;
        distribution.reseedRandomGenerator(seed);
    }

    public RandomizedMatrix() {
        this(new UniformRealDistribution(-1, 1), 0L);
    }

    public void fillMatrix(RealMatrix matrix) {
        for (int i = 0; i < matrix.getRowDimension(); i++) {
            matrix.setRow(i, distribution.sample(matrix.getColumnDimension()));
        }
    }

    public RealMatrix getMatrix(int numRows, int numCols) {
        RealMatrix output = new BlockRealMatrix(numRows, numCols);
        for (int i = 0; i < numRows; i++) {
            output.setRow(i, distribution.sample(numCols));
        }
        return output;
    }

    public void fillVector(RealVector vector) {
        for (int i = 0; i < vector.getDimension(); i++) {
            vector.setEntry(i, distribution.sample());
        }
    }

    public RealVector getVector(int dim) {
        return new ArrayRealVector(distribution.sample(dim));
    }
}
```

We can create a narrow band of normally distributed numbers with this:

```
int numRows = 3;
int numCols = 4;
long seed = 0L;
RandomizedMatrix rndMatrix = new RandomizedMatrix(
    new NormalDistribution(0.0, 0.5), seed);
RealMatrix matrix = rndMatrix.getMatrix(numRows, numCols);
```

```
// -0.0217405716,-0.5116704988,-0.3545966969,0.4406692276
// 0.5230193567,-0.7567264361,-0.5376075694,-0.1607391808
// 0.3181005362,0.6719107279,0.2390245133,-0.1227799426
```

Operating on Vectors and Matrices

Sometimes you know the formulation you are looking for in an algorithm or data structure but you may not be sure how to get there. You can do some "mental pattern matching" in your head and then choose to implement (e.g., a dot product instead of manually looping over all the data yourself). Here we explore some common operations used in linear algebra.

Scaling

To scale (multiply) a vector by a constant κ such that

$$\kappa \mathbf{x} = (\kappa x_1,\ \kappa x_2,\ \ldots,\ \kappa x_n)$$

Apache Commons Math implements a mapping method whereby an existing `RealVector` is multiplied by a double, resulting in a new `RealVector` object:

```
double k = 1.2;
RealVector scaledVector = vector.mapMultiply(k);
```

Note that a `RealVector` object may also be scaled in place by altering the existing vector permanently:

```
vector.mapMultiplyToSelf(k);
```

Similar methods exist for dividing the vector by `k` to create a new vector:

```
RealVector scaledVector = vector.mapDivide(k);
```

And for division in place:

```
vector.mapDivideToSelf(k);
```

A matrix **A** can also be scaled by a factor κ:

$$\kappa \mathbf{A} = \begin{pmatrix} \kappa a_{1,1} & \kappa a_{1,2} & \cdots & \kappa a_{1,n} \\ \kappa a_{2,1} & \kappa a_{2,2} & \cdots & \kappa\kappa a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \kappa a_{m,1} & \kappa a_{m,2} & \cdots & \kappa a_{m,n} \end{pmatrix}$$

Here, each value of the matrix is multiplied by a constant of type `double`. A new matrix is returned:

```
double k = 1.2;
RealMatrix scaledMatrix = matrix.scalarMultiply(k);
```

Transposing

Transposing a vector or matrix is analogous to tipping it over on its side. The vector transpose of **x** is denoted as $\mathbf{X}^T$. For a matrix, the transpose of **A** is denoted as $\mathbf{A}^T$. In most cases, calculating a vector transpose will not be necessary, because the methods of `RealVector` and `RealMatrix` will take into account the need for a vector transpose inside their logic. A vector transpose is undefined unless the vector is represented in matrix format. The transpose of an $m \times 1$ column vector is then a new matrix row vector of dimension $1 \times m$.

$$\mathbf{x}^T = (x_1, x_2 \cdots , x_m)$$

When you absolutely need to transpose a vector, you can simply insert the data into a `RealMatrix` instance. Using a one-dimensional array of `double` values as the argument to the `Array2DRowRealMatrix` class creates a matrix with m rows and one column, where the values are provided by the array of `doubles`. Transposing the column vector will return a matrix with one row and m columns:

```java
double[] data = {1.2, 3.4, 5.6};
RealMatrix columnVector = new Array2DRowRealMatrix(data);
System.out.println(columnVector);
/* {{1.2}, {3.4}, {5.6}} */
RealMatrix rowVector = columnVector.transpose();
System.out.println(rowVector);
/* {{1.2, 3.4, 5.6}} */
```

When a matrix of dimension $m \times n$ is transposed, the result is an $n \times m$ matrix. Simply put, the row and column indices i and j are reversed:

$$\mathbf{A}^T = \begin{pmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{m,n} \end{pmatrix}$$

Note that the matrix transpose operation returns a new matrix:

```java
double[][] data = {{1, 2, 3}, {4, 5, 6}};
RealMatrix matrix = new Array2DRowRealMatrix(data);
RealMatrix transposedMatrix = matrix.transpose();
/* {{1, 4}, {2, 5}, {3, 6}} */
```

Addition and Subtraction

The *addition* of two vectors **a** and **b** of equal length n results in a vector of length n with values equal to the element-wise addition of the vector components:

$$\mathbf{a} + \mathbf{b} = (a_1 + b_1, a_2 + b_2, \ \ldots, a_n + b_n)$$

The result is a new `RealVector` instance:

```
RealVector aPlusB = vectorA.add(vectorB);
```

Similarly, *subtracting* two `RealVector` objects of equal length n is shown here:

$$\mathbf{a} - \mathbf{b} = (a_1 - b_1,\ a_2 - b_2,\ \ldots,\ a_n - b_n)$$

This returns a new `RealVector` whose values are the element-wise subtraction of the vector components:

```
RealVector aMinusB = vectorA.subtract(vectorB);
```

Matrices of identical dimensions can also be added and subtracted similarly to vectors:

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \cdots & a_{1,n} + b_{1,n} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \cdots & a_{2,n} + b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} + b_{m,1} & a_{m,2} + b_{m,2} & \cdots & a_{m,n} + b_{m,n} \end{pmatrix}$$

The addition or subtraction of `RealMatrix` objects **A** and **B** returns a new `RealMatrix` instance:

```
RealMatrix aPlusB = matrixA.add(matrixB);
RealMatrix aMinusB = matrixA.subtract(matrixB);
```

Length

The *length* of a vector is a convenient way to reduce all of a vector's components to one number and should not be confused with the dimension of the vector. Several definitions of vector length exist; the two most common are the L1 norm and the L2 norm. The L1 norm is useful, for example, in making sure that a vector of probabilities, or fractions of some mixture, all add up to one:

$$|\mathbf{x}| = \sum_{i=1}^{n} |x_i|$$

The L1 norm, which is less common than the L2 norm, is usually referred to by its full name, *L1 norm*, to avoid confusion:

```
double norm = vector.getL1Norm();
```

The L2 norm is usually what is used for normalizing a vector. Many times it is referred to as the *norm* or the *magnitude* of the vector, and it is mathematically formulated as follows:

$$\| \mathbf{x} \| = \sqrt{\sum_{i=1}^{n} | x_i |^2}$$

```
/* calculate the L2 norm of a vector */
double norm = vector.getNorm();
```

TIP

People often ask when to use L1 or L2 vector lengths. In practical terms, it matters what the vector represents. In some cases, you will be collecting counts or probabilities in a vector. In that case, you should normalize the vector by dividing by its sum of parts (L1). On the other hand, if the vector contains some kind of coordinates or features, then you will want to normalize the vector by its Euclidean distance (L2).

The *unit vector* is the direction that a vector points, so called because it has been scaled by its L2 norm to have a length = 1. It is usually denoted with $\hat{x}$ and is calculated as follows:

$$\hat{x} = \frac{\mathbf{x}}{\| \mathbf{x} \|}$$

The `RealVector.unitVector()` method returns a new `RealVector` object:

```
/* create a new vector that is the unit vector of vector instance*/
RealVector unitVector = vector.unitVector();
```

A vector can also be transformed, in place, to a unit vector. A vector **v** will be permanently changed into its unit vector with the following:

```
/* convert a vector to unit vector in-place */
vector.unitize();
```

We can also calculate the norm of a matrix via the Frobenius norm represented mathematically as the square root of the sum of squares of all elements:

$$\| A \|_F \equiv \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} | a_{ij} |^2}$$

This is rendered in Java with the following:

```
double matrixNorm = matrix.getFrobeniusNorm();
```

Distances

The *distance* between any two vectors **a** and **b** may be calculated in several ways. The L1 distance between **a** and **b** is shown here:

$$d_{L1} = \sum_{i=1}^{n} | a_i - b_i |$$

```
double l1Distance = vectorA.getL1Distance(vectorB);
```

The L2 distance (also known as the Euclidean distance) is formulated as

$$d_{L2} = \sqrt{\sum_{i=1}^{n} | a_i - b_i |^2}$$

This is most often the distance between vectors that is called for. The method `Vector.getDistance(RealVector vector)` returns the Euclidean distance:

```
double l2Distance = vectorA.getDistance(vectorB);
```

The *cosine distance* is a measure between –1 and 1 that is not so much a distance metric as it is a “similarity” measure. If $d = 0$, the two vectors are perpendicular (and have nothing in common). If $d = 1$, the vectors point in the same direction. If $d = –1$, the vectors point in exact opposite directions. The cosine distance may also be thought of as the dot product of two unit vectors:

$$d = \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\| \mathbf{a} \| \; \| \mathbf{b} \|}$$

```
double cosineDistance = vectorA.cosine(vectorB);
```

If both **a** and **b** are unit vectors, the cosine distance is just their inner product:

$$d = \cos(\theta) = \hat{\mathbf{a}} \cdot \hat{\mathbf{b}}$$

and the `Vector.dotProduct(RealVector vector)` method will suffice:

```
/* for unit vectors a and b */
vectorA.unitize();
vectorB.unitize();
double cosineDistance = vectorA.dotProduct(vectorB);
```

Multiplication

The product of an $m \times n$ matrix **A** and an $n \times p$ matrix **B** is a matrix of dimension $m \times p$. The only dimension that must batch is n the number of columns in **A** and the number of rows in **B**:

$$\mathbf{AB} = \begin{pmatrix} (AB)_{1,1} & (AB)_{1,2} & \cdots & (AB)_{1,p} \\ (AB)_{2,1} & (AB)_{2,2} & \cdots & (AB)_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ (AB)_{m,1} & (AB)_{m,2} & \cdots & (AB)_{m,p} \end{pmatrix}$$

The value of each element $(\mathbf{AB})_{ij}$ is the sum of the multiplication of each element of the *i-th* row of **A** and the *j-th* column of **B**, which is represented mathematically as follows:

$$(AB)_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Multiplying a matrix **A** by a matrix **B** is achieved with this:

```
RealMatrix matrixMatrixProduct = matrixA.multiply(matrixB);
```

Note that **AB** ≠ **BA**. To perform **BA**, either do so explicitly or use the `preMultiply` method. Either code has the same result. However, note that in that case, the number of columns of **B** must be equal to the number of rows in **A**:

```
/* BA explicitly */
RealMatrix matrixMatrixProduct matrixB.multiply(matrixA);

/* BA using premultiply */
RealMatrix matrixMatrixProduct = matrixA.preMultiply(matrixB);
```

Matrix multiplication is also commonly called for when multiplying an $m \times n$ matrix **A** with an $n \times 1$ column vector **x**. The result is an $m \times 1$ column vector **b** such that **Ax** = **b**. The operation is performed by summing the multiplication of each element in the *i-th* row of **A** with each element of the vector **x**. In matrix notation:

$$\mathbf{Ax} = \begin{pmatrix} a_{1,1}x_1 + a_{1,2}x_2 + & \cdots & + a_{1,n}x_n \\ a_{2,1}x_1 + a_{2,2}x_2 + & \cdots & + a_{2,n}x_n \\ & \vdots & \\ a_{m,1}x_1 + a_{m,2}x_2 + & \cdots & + a_{m,n}x_n \end{pmatrix}$$

The following code is identical to the preceding matrix-matrix product:

```
/* Ax results in a column vector */
RealMatrix matrixVectorProduct = matrixA.multiply(columnVectorX);
```

We often wish to calculate the vector-matrix product, usually denoted as $\mathbf{x}^T\mathbf{A}$. When **x** is the format of a matrix, we can perform the calculation explicitly as follows:

```
/* x^TA explicitly */
RealMatrix vectorMatrixProduct = columnVectorX.transpose().multiply(matrixA);
```

When **x** is a `RealVector`, we can use the `RealMatrix.preMultiply()` method:

```
/* x^TA with preMultiply */
RealMatrix vectorMatrixProduct = matrixA.preMultiply(columnVectorX);
```

When performing **Ax**, we often want the result as a vector (as opposed to a column vector in a matrix). If **x** is a `RealVector` type, a more convenient way to perform **Ax** is with this:

```
/* Ax */
RealVector matrixVectorProduct = matrixA.operate(vectorX);
```

Inner Product

The *inner product* (also known as the dot product or scalar product) is a method for multiplying two vectors of the same length. The result is a scalar value that is formulated mathematically with a raised dot between the vectors as follows:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

For `RealVector` objects `vectorA` and `vectorB`, the dot product is as follows:

```
double dotProduct = vectorA.dotProduct(vectorB);
```

If the vectors are in matrix form, you can use matrix multiplication, because $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}\mathbf{b}^T$, where the left side is the dot product and the right side is the matrix multiplication:

$$\mathbf{a}^T\mathbf{b} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \end{pmatrix} \begin{pmatrix} b_{1,1} \\ b_{2,1} \\ \vdots \\ b_{n,1} \end{pmatrix}$$

The matrix multiplication of column vectors **a** and **b** returns a *1* × *1* matrix:

```
/* matrixA and matrixB are both mx1 column vectors */
RealMatrix innerProduct = matrixA.transpose().multiply(matrixB);

/* the result is stored in the only entry for the matrix */
double dotProduct = innerProduct.getEntry(0,0);
```

Although matrix multiplication may not seem practical compared to the dot product, it illustrates an important relationship between vector and matrix operations.

Outer Product

The *outer product* between a vector **a** of dimension m and a vector **b** of dimension n returns a new matrix of dimension $m \times n$:

$$\mathbf{ab}^T = \begin{pmatrix} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{m,1} \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,n} \end{pmatrix} = \begin{pmatrix} a_{1,1}b_{1,1} & a_{1,1}b_{1,2} & \cdots & a_{1,1}b_{1,n} \\ a_{2,1}b_{1,1} & a_{2,1}b_{1,2} & \cdots & a_{2,1}b_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}b_{1,1} & a_{m,1}b_{1,2} & \cdots & a_{m,1}b_{1,n} \end{pmatrix}$$

Keep in mind that $\mathbf{ab}^T$ has the dimension $m \times n$ and does not equal $\mathbf{ba}^T$, which has dimension $n \times m$. The `RealMatrix.outerProduct()` method conserves this order and returns a `RealMatrix` instance with the appropriate dimension:

```
/* outer product of vector a with vector b */
RealMatrix outerProduct = vectorA.outerProduct(vectorB);
```

If the vectors are in matrix form, the outer product can be calculated with the `RealMatrix.multiply()` method instead:

```
/* matrixA and matrixB are both nx1 column vectors */
RealMatrix outerProduct = matrixA.multiply(matrixB.transpose());
```

Entrywise Product

Also known as the Hadamard product or the Schur product, the entrywise product multiplies each element of one vector by each element of another vector. Both vectors must have the same dimension, and their resultant vector is therefore of the same dimension:

$$\mathbf{a} \circ \mathbf{b} = (a_1 b_1,\ a_2 b_2,\ \dots,\ a_n b_n)$$

The method `RealVector.ebeMultiply(RealVector)` performs this operation, in which `ebe` is short for *element by element*.

```
/* compute the entrywise multiplication of vector a and vector b */
RealVector vectorATimesVectorB = vectorA.ebeMultiply(vectorB);
```

A similar operation for entrywise division is performed with `RealVector.ebeDivision(RealVector)`.

CAUTION

Entrywise products should not be confused with matrix products (including inner and outer products). In most algorithms, matrix products are called for. However, the entrywise product will come in handy when, for example, you need to scale an entire vector by a corresponding vector of weights.

The Hadamard product is not currently implemented for matrix-matrix products in Apache Commons Math, but we can easily do so in a naive way with the following:

```java
public class MatrixUtils {

    public static RealMatrix ebeMultiply(RealMatrix a, RealMatrix b) {
        int rowDimension = a.getRowDimension();
        int columnDimension = a.getColumnDimension();
        RealMatrix output = new Array2DRowRealMatrix(rowDimension,
            columnDimension);
        for (int i = 0; i < rowDimension; i++) {
            for (int j = 0; j < columnDimension; j++) {
                output.setEntry(i, j, a.getEntry(i, j) * b.getEntry(i, j));
            }
        }
        return output;
    }
}
```

This can be implemented as follows:

```java
/* element-by-element product of matrixA and matrixB */
RealMatrix hadamardProduct = MatrixUtils.ebeMultiply(matrixA, matrixB);
```

Compound Operations

You will often run into compound forms involving several vectors and matrices, such as $\mathbf{x}^T\mathbf{Ax}$, which results in a singular, scalar value. Sometimes it is convenient to work the calculation in chunks, perhaps even out of order. In this case, we can first compute the vector $\mathbf{v} = \mathbf{Ax}$ and then find the dot (inner) product $\mathbf{x} \cdot \mathbf{v}$:

```java
double[] xData = {1, 2, 3};
double[][] aData = {{1, 3, 1}, {0, 2, 0}, {1, 5, 3}};
RealVector vectorX = new ArrayRealVector(xData);
RealMatrix matrixA = new Array2DRowRealMatrix(aData);
double d = vectorX.dotProduct(matrixA.operate(vectorX));
// d = 78
```

Another method is to first multiply the vector by the matrix by using `RealMatrix.premultiply()` and then compute the inner product (dot product) between the two vectors:

```java
double d = matrixA.premultiply(vecotrX).dotProduct(vectorX);
//d = 78
```

If the vectors are in matrix format as column vectors, we can exclusively use matrix methods. However, note that the result will be a matrix as well:

```java
RealMatrix matrixX = new Array2DRowRealMatrix(xData);
/* result is 1x1 matrix */
RealMatrix matrixD = matrixX.transpose().multiply(matrixA).multiply(matrixX);
d = matrixD.getEntry(0, 0); // 78
```

Affine Transformation

A common procedure is to transform a vector **x** of length n by applying a linear map matrix **A** of dimensions $n \times p$ and a translation vector **b** of length p, where the relationship

$$f(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$$

is known as an *affine transformation*. For convenience, we can set **z** = *f*(**x**), move the vector **x** to the other side, and define **W** = **A**^{T} with dimensions $p \times n$ such that

$$\mathbf{z} = \mathbf{x}^T\mathbf{W} + \mathbf{b}$$

In particular, we see this form quite a bit in learning and prediction algorithms, where it is important to note that **x** is a multidimensional vector of one observation, not a one-dimensional vector of many observations. Written out, this looks like the following:

$$\begin{pmatrix} z_1 & z_2 & \cdots & z_p \end{pmatrix} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} \beta_1 & \beta_2 & \cdots & \beta_p \end{pmatrix}$$

We can also express the affine transform of an $m \times n$ matrix **X** with this:

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{B}$$

B has the dimension $m \times p$:

$$\mathbf{Z} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} \beta_{1,1} & \beta_{1,2} & \cdots & \beta_{1,p} \\ \beta_{2,1} & \beta_{2,2} & \cdots & \beta_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{m,1} & \beta_{m,2} & \cdots & \beta_{m,p} \end{pmatrix}$$

In most cases, we would like the translation matrix to have equivalent rows of the vector **b** of length p so that the expression is then

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{h}\mathbf{b}^T$$

where **h** is an m-length column vector of ones. Note that the outer product of these two vectors creates an $m \times p$ matrix. Written out, the expression then looks like this:

$$\mathbf{Z} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n,1} & \omega_{n,2} & \cdots & \omega_{n,p} \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \begin{pmatrix} \beta_1 & \beta_2 & \cdots & \beta_p \end{pmatrix}$$

This is such an important function that we will include it in our `MatrixOperations` class:

```
public class MatrixOperations {
...
    public static RealMatrix XWplusB(RealMatrix x, RealMatrix w, RealVector b) {
        RealVector h = new ArrayRealVector(x.getRowDimension(), 1.0);
        return x.multiply(w).add(h.outerProduct(b));
    }
...
}
```

Mapping a Function

Often we need to map a function φ over the contents of a vector **z** such that the result is a new vector **y** of the same shape as **z**:

$$\mathbf{y} = \varphi(\mathbf{z})$$

The Commons Math API contains a method `RealVector.map(UnivariateFunction function)`, which does exactly that. Most of the standard and some other useful functions are included in Commons Math that implement the `UnivariateFunction` interface. It is invoked with the following:

```
// map exp over vector input into new vector output
RealVector output = input.map(new Exp());
```

It is straightforward to create your own `UnivariateFunction` classes for forms that are not included in Commons Math. Note that this method does not alter the input vector. If you would like to alter the input vector in place, use this:

```
// map exp over vector input rewriting its values
input.mapToSelf(new Exp());
```

On some occasions, we want to apply a univariate function to each entry of a matrix. The Apache Commons Math API provides an elegant way to do this that works efficiently even for sparse matrices. It is the `RealMatrix.walkInOptimizedOrder(RealMatrixChangingVisitor visitor)` method. Keep in mind, there are other options here. We can visit each entry of the matrix in either row or column order, which may be useful (or required) for some operations. However, if we only want to update each element of a matrix independently, then using the optimized order is the most adaptable algorithm because it will work for matrices with either 2D array, block, or sparse storage. The first step is to build a class (which acts as the mapping function) that extends the `RealMatrixChangingVisitor` interface and implement the required methods:

```
public class PowerMappingFunction implements RealMatrixChangingVisitor {

    private double power;

    public PowerMappingFunction(double power) {
        this.power = power;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
        int startColumn, int endColumn) {
        // called once before start of operations ... not needed here
    }

    @Override
    public double visit(int row, int column, double value) {
        return Math.pow(value, power);
    }

    @Override
    public double end() {
        // called once after all entries visited ... not needed here
        return 0.0;
    }
```

Then to map the required function over an existing matrix, pass an instance of the class to the `walkInOptimizedOrder()` method like so:

```
/* each element 'x' of matrix is updated in place with x^1.2 */
matrix.walkInOptimizedOrder(new PowerMappingFunction(1.2));
```

We can also utilize Apache Commons Math built-in analytic functions that implement the `UnivariateFunction` interface to easily map any existing function over each entry of a matrix:

```
public class UnivariateFunctionMapper implements RealMatrixChangingVisitor {

    UnivariateFunction univariateFunction;

    public UnivariateFunctionMapper(UnivariateFunction univariateFunction) {
        this.univariateFunction = univariateFunction;
    }

    @Override
    public void start(int rows, int columns, int startRow, int endRow,
            int startColumn, int endColumn) {
        //NA
    }

    @Override
    public double visit(int row, int column, double value) {
        return univariateFunction.value(value);
    }

    @Override
    public double end() {
        return 0.0;
    }
}
```

This interface can be utilized, for example, when extending the affine transformation static method in the preceding section:

```
public class MatrixOperations {
...
    public static RealMatrix XWplusB(RealMatrix X, RealMatrix W, RealVector b,
            UnivariateFunction univariateFunction) {

        RealMatrix z = XWplusB(X, W, b);
        z.walkInOptimizedOrder(new UnivariateFunctionMapper(univariateFunction));
        return z;
    }
...
}
```

So, for example, if we wanted to map the sigmoid (logistic) function over an affine transformation, we would do this:

```
// for input matrix x, weight w and bias b, mapping sigmoid over all entries
MatrixOperations.XWplusB(x, w, b, new Sigmoid());
```

There are a couple of important things to realize here. First, note there is also a *preserving visitor* that visits each element of a matrix but does not change it. The other thing to take note of are the methods. The only method you will really need to implement is the `visit()` method, which should return the new value for each input value. Both the `start()` and `end()` methods are not needed (particularly in this case). The `start()` method is called once before the start of all the operations. So, for example, say we need the matrix determinant in our further calculations. We could calculate it once in the `start()` method, store it as a class variable, and then use it later in the operations of `visit()`. Similarly, `end()` is called once after all the elements have been visited. We could use this for tallying a running metric, total sites visited, or even an error signal. In any case, the value of `end()` is returned by the method when everything is done. You are not required to include any real logic in the `end()` method, but at the very least you can return a valid double such as 0.0, which is nothing more than a placeholder. Note the method

`RealMatrix.walkInOptimizedOrder(RealMatrixChangingVisitor visitor, int startRow, int endRow, int startColumn, int endColumn)`, which operates only on a submatrix whose bounds are indicated by the signature. Use this when you want to update, in-place, only a specific rectangular block of a matrix and leave the rest unchanged.

Decomposing Matrices

Considering what we know about matrix multiplication, it is easy to imagine that any matrix can be decomposed into several other matrices. Decomposing a matrix into parts enables the efficient and numerically stable calculation of important matrix properties. For example, although the matrix inverse and the matrix determinant have explicit, algebraic formulas, they are best calculated by first decomposing the matrix and then taking the inverse. The determinant comes directly from a Cholesky or LU decomposition. All matrix decompositions here are capable of solving linear systems and as a consequence make the matrix inverse available. Table 2-1 lists the properties of various matrix decompositions as implemented by Apache Commons Math.

Table 2-1. Matrix decomposition properties

Decomposition	Matrix Type	Solver	Inverse	Determinant
Cholesky	Symmetric positive definite	Exact	✓	✓
Eigen	Square	Exact	✓	✓
LU	Square	Exact	✓	✓
QR	Any	Least squares	✓	
SVD	Any	Least squares	✓	

Cholesky Decomposition

A *Cholesky decomposition* of a matrix **A** decomposes the matrix such that $\mathbf{A} = \mathbf{L}\mathbf{L}^{T}$, where **L** is a lower triangular matrix, and the upper triangle (above the diagonal) is zero:

$$\mathbf{A} = \begin{pmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{pmatrix} \begin{pmatrix} l_{1,1} & l_{1,2} & \cdots & l_{1,n} \\ 0 & l_{2,2} & \cdots & l_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{n,n} \end{pmatrix}$$

```
CholeskyDecomposition cd = new CholeskyDecomposition(matrix);
RealMatrix l = cd.getL();
```

A Cholesky decomposition is valid only for symmetric matrices. The main use of a Cholesky is in the computation of random variables for the multinormal distribution.

LU Decomposition

The *lower-upper (LU) decomposition* decomposes a matrix **A** into a lower diagonal matrix **L** and an upper diagonal matrix **U** such that $\mathbf{A} = \mathbf{L}\mathbf{U}$:

$$\mathbf{A} = \begin{pmatrix} l_{1,1} & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n} \end{pmatrix} \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{pmatrix}$$

```
LUDecomposition lud = new LUDecomposition(matrix);
RealMatrix u = lud.getU();
RealMatrix l = lud.getL();
```

The LU decomposition is useful in solving systems of linear equations in which the number of unknowns is equal to the number of equations.

QR Decomposition

The *QR decomposition* decomposes the matrix **A** into an orthogonal matrix of column unit vectors **Q** and an upper triangular matrix **R** such that

$$\mathbf{A} = \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,m} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ q_{m,1} & q_{m,2} & \cdots & q_{m,m} \end{pmatrix} \begin{pmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ 0 & r_{2,2} & \cdots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{m,n} \end{pmatrix}$$

```
QRDecomposition qrd = new QRDecomposition(matrix);
RealMatrix q = lud.getQ();
RealMatrix r = lud.getR();
```

One of the main uses of the QR decomposition (and analogous decompositions) is in the calculation of eigenvalue decompositions because each column of **Q** is orthogonal. The QR decomposition is also useful in solving overdetermined systems of linear equations. This is usually the case for datasets in which the number of data points (rows) is greater than the dimension (number of columns). One advantage of using the QR decomposition solver (as opposed to SVD) is the easy access to the errors on the solution parameters that can be directly calculated from R.

Singular Value Decomposition

The singular value decomposition (SVD) decomposes the $m \times n$ matrix **A** such that $\mathbf{A} = \mathbf{U\Sigma V}^{\mathrm{T}}$, where **U** is an $m \times m$ unitary matrix, **S** is an $m \times n$ diagonal matrix with real, non-negative values, and **V** is an $n \times n$ unitary matrix. As unitary matrices, both **U** and **V** have the property $\mathbf{UU}^{\mathrm{T}} = \mathbf{I}$, where **I** is the identity matrix.

$$\mathbf{A} = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,m} \\ u_{2,1} & u_{2,2} & \cdots & u_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ u_{m,1} & u_{m,2} & \cdots & u_{m,m} \end{pmatrix} \begin{pmatrix} s_{1,1} & 0 & \cdots & 0 & \cdots & 0 \\ 0 & s_{2,2} & \cdots & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & \\ 0 & 0 & \cdots & s_{n,n} & \cdots & 0_{m,n} \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{2,1} & \cdots & v_{n,1} \\ v_{1,2} & v_{2,2} & \cdots & v_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,n} & v_{2,n} & \cdots & v_{n,n} \end{pmatrix}$$

In many cases, $m \geq n$; the number of rows in a matrix will be greater than or equal to the number of columns. In this case, there is no need to calculate the full SVD. Instead, a more efficient calculation called *thin SVD* can be implemented, where **U** is $m \times$

n, **S** is $n \times n$, and **V** is $n \times n$. As a practical matter, there may also be cases when $m \leq n$ so we can then just use the smaller of the two dimensions: $p = min(m, n)$. The Apache Commons Math implementation uses that practice:

```
/* matrix is mxn and p = min(m,n) */
SingularValueDecomposition svd = new SingularValueDecomposition(matrix);
RealMatrix u = svd.getU(); // m x p
RealMatrix s = svd.getS(); // p x p
RealMatrix v = svd.getV(); // p x n
/* retrieve values, in decreasing order, from the diagonal of S */
double[] singularValues = svd.getSingularValues();
/* can also get covariance of input matrix */
double minSingularValue = 0;// 0 or neg value means all sv are used
RealMatrix cov = svd.getCovariance(minSingularValue);
```

The singular value decomposition has several useful properties. Like the eigen decomposition, it is used to reduce the matrix **A** to a smaller dimension, keeping only the most useful of them. Also, as a linear solver, the SVD works on any shape of matrix and in particular, is stable on underdetermined systems in which the number of dimensions (columns) is much greater than the number of data points (rows).

Eigen Decomposition

The goal of the eigen decomposition is to reorganize the matrix **A** into a set of independent and orthogonal column vectors called *eigenvectors*. Each eigenvector has an associated eigenvalue that can be used to rank the eigenvectors from most important (highest eigenvalue) to least important (lowest eigenvalue). We can then choose to use only the most significant eigenvectors as representatives of matrix **A**. Essentially, we are asking, is there some way to completely (or mostly) describe matrix **A**, but with fewer dimensions?

For a matrix **A**, a solution exists for a vector **x** and a constant λ such that **Ax** = λ**x**. There can be multiple solutions (i.e., **x**, λ pairs). Taken together, all of the possible values of lambda are known as the eigenvalues, and all corresponding vectors are known as the eigenvectors. The eigen decomposition of a symmetric, real matrix **A** is expressed as $\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^T$. The results are typically formulated as a diagonal $m \times m$ matrix **D**, in which the eigenvalues are on the diagonal and an $m \times m$ matrix **V** whose column vectors are the eigenvectors.

$$A = \begin{pmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,m} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m,1} & v_{m,2} & \cdots & v_{m,m} \end{pmatrix} \begin{pmatrix} d_{1,1} & 0 & \cdots & 0 \\ 0 & d_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_{m,m} \end{pmatrix} \begin{pmatrix} v_{1,1} & v_{2,1} & \cdots & v_{m,1} \\ v_{1,2} & v_{2,2} & \cdots & v_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,m} & v_{2,m} & \cdots & v_{m,m} \end{pmatrix}$$

Several methods exist for performing an eigenvalue decomposition. In a practical sense, we usually need only the simplest form as implemented by Apache Commons Math in the `org.apache.commons.math3.linear.EigenDecomposition` class. The eigenvalues and eigenvectors are sorted by descending order of the eigenvalues. In other words, the first eigenvector (the zeroth column of matrix Q) is the most significant eigenvector.

```
double[][] data = {{1.0, 2.2, 3.3}, {2.2, 6.2, 6.3}, {3.3, 6.3, 5.1}};
RealMatrix matrix = new Array2DRowRealMatrix(data);

/* compute eigenvalue matrix D and eigenvector matrix V */
EigenDecomposition eig = new EigenDecomposition(matrix);

/* The real (or imag) eigenvalues can be retrieved as an array of doubles */
double[] eigenValues = eig.getRealEigenvalues();

/* Individual eigenvalues can be also be accessed directly from D */
double firstEigenValue = eig.getD().getEntry(0, 0);

/* The first eigenvector can be accessed like this */
RealVector firstEigenVector = eig.getEigenvector(0);
```

```
/* Remember that eigenvectors are just the columns of V */
RealVector firstEigenVector = eig.getV.getColumn(0);
```

Determinant

The *determinant* is a scalar value calculated from a matrix **A** and is most often seen as a component as the multinormal distribution. The determinant of matrix **A** is denoted as |**A**|. The Cholesky, eigen, and LU decomposition classes provide access to the determinant:

```
/* calculate determinant from the Cholesky decomp */
double determinant = new CholeskyDecomposition(matrix).getDeterminant();

/* calculate determinant from the eigen decomp */
double determinant = new EigenDecomposition(matrix).getDeterminant();

/* calculate determinant from the LU decomp */
double determinant = new LUDecomposition(matrix).getDeterminant();
```

Inverse

The *inverse* of a matrix is similar to the concept of inverting a real number $\Re$, where $\Re(1/\Re) = 1$. Note that this can also be written as $\Re\Re^{-1} = 1$. Similarly, the inverse of a matrix **A** is denoted by $\mathbf{A}^{-1}$ and the relation exists $\mathbf{AA}^{-1} = \mathbf{I}$, where **I** is the identity matrix. Although formulas exist for directly computing the inverse of a matrix, they are cumbersome for large matrices and numerically unstable. Each of the decomposition methods available in Apache Commons Math implements a `DecompositionSolver` interface that requires a matrix inverse in its solution of linear systems. The matrix inverse is then retrieved from the accessor method of the `DecompositionSolver` class. Any of the decomposition methods provides a matrix inverse if the matrix type is compatible with the method used:

```
/* the inverse of a square matrix from Cholesky, LU, Eigen, QR,
      or SVD decompositions */
RealMatrix matrixInverse = new LUDecomposition(matrix).getSolver().getInverse();
```

The matrix inverse can also be calculated from the singular value decomposition:

```
/* the inverse of a square or rectangular matrix from QR or SVD decomposition */
RealMatrix matrixInverse =
new SingularValueDecomposition(matrix).getSolver().getInverse();
```

Or the QR decomposition:

```
/* OK on rectangular matrices, but error on non-singular matrices */
RealMatrix matrixInverse = new QRDecomposition(matrix).getSolver().getInverse();
```

A matrix inverse is used whenever matrices are moved from one side of the equation to the other via division. Another common application is in the computation of the Mahalanobis distance and, by extension, for the multinormal distribution.

Solving Linear Systems

At the beginning of this chapter, we described the system **XW** = **Y** as a fundamental concept of linear algebra. Often, we also want to include an intercept or offset term β not dependent on x such that

$$y_{1,1} = \beta + x_{1,1}\omega_{1,1} + x_{1,2}\omega_{2,1} + \cdots + x_{1,n}\omega_{n,1}$$

There are two options for including the intercept term, the first of which is to add a column of 1s to **X** and a row of unknowns to **W**. It does not matter which column-row pair is chosen as long as $j = i$ in this case. Here we choose the last column of **X** and the last row of **W**:

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} & 1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} & 1 \end{pmatrix} \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \cdots & \omega_{1,p} \\ \omega_{2,1} & \omega_{2,2} & \cdots & \omega_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_{n+1,1} & \omega_{n+1,2} & \cdots & \omega_{n+1,p} \end{pmatrix} = \begin{pmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,p} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m,1} & y_{m,2} & \cdots & y_{m,p} \end{pmatrix}$$

Note that in this case, the columns of **W** are independent. Therefore, we are simply finding p separate linear models, except for the convenience of performing the operation in one piece of code:

```
/* data */
double[][] xData = {{0, 0.5, 0.2}, {1, 1.2, .9}, {2, 2.5, 1.9}, {3, 3.6, 4.2}};
double[][] yData = {{-1, -0.5}, {0.2, 1}, {0.9, 1.2}, {2.1, 1.5}};

/* create X with offset as last column */
double[] ones = {1.0, 1.0, 1.0, 1.0};
int xRows = 4;
int xCols = 3;
RealMatrix x = new Array2DRowRealMatrix(xRows, xCols + 1);
x.setSubMatrix(xData, 0, 0);
x.setColumn(3, ones); // 4th column is index of 3 !!!

/* create Y */
RealMatrix y = new Array2DRowRealMatrix(yData);

/* find values for W */
SingularValueDecomposition svd = new SingularValueDecomposition(x);
RealMatrix solution = svd.getSolver().solve(y);
System.out.println(solution);
// {{1.7,3.1},{-0.9523809524,-2.0476190476},
//  {0.2380952381,-0.2380952381},{-0.5714285714,0.5714285714}}
```

Given the values for the parameters, the solution for the system of equations is as follows:

$$y_1 = 1.7x_1 - 0.95x_2 + 0.24x_3 - 0.57$$
$$y_2 = 3.1x_1 - 2.05x_2 - 0.24x_3 + 0.57$$

The second option for including the intercept is to realize that the preceding algebraic expression is equivalent to the affine transformation of a matrix described earlier in this chapter:

$$\mathbf{Y} = \mathbf{XW} + \mathbf{hb}^T$$

This form of a linear system has the advantage that we do not need to resize any matrices. In the previous example code resizing the matrices occurs only one time, and this is not too much of a burden. However, in Chapter 5, we will tackle a multilayered linear model (deep network) in which resizing matrices will be cumbersome and inefficient. In that case, it is much more convenient to represent the linear model in algebraic terms, where **W** and **b** are completely separate.

PREV
1. Data I/O

NEXT
3. Statistics