Machine Learning Crash Course

Joseph Suarez

Abstract

We present the tools and paradigms required to gain a basic understanding of the fundamentals of machine learning. The techniques discussed are presented in a manner that introduces concepts of calculus as they appear, though prior understanding would be helpful. Understanding the inner mechanism of the algorithms shown *does* require a strong understanding of the fundamentals of linear algebra often presented in a typical Algebra II course. We will present a review of the pertinent components of linear algebra, along with the tools required to visualize the types of data that commonly appear in machine learning, before proceeding to introduce linear regression as a learning problem.

1. Linear Algebra Review

1.1 Vectors and Matrices

Vectors, often called arrays in computer science, are lists of numbers arranged in either rows or columns. As they represent a line of numbers, they are called one-dimensional. For example,

$$\mathrm{v} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$

In general, elements are indexed (starting with either zero or one) as shown:

$$\mathrm{v} = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \end{bmatrix}$$

Matrices are two-dimensional blocks of numbers arranged into rows and columns. For example,

$$\mathbf{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

In general, elements are indexed (starting with zero or one) as shown:

$$\mathbf{M} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

1.2 Element Wise Operations

Element-wise operations are regular mathematical operations applied to matrices. Notation for these operations is controversial, so the context of the operation often determines the meaning (see 1.3). These operations include addition, subtraction, multiplication, division, power, and modulus. Note that for all of the operations shown, both matrices involved must have exactly the same dimensions.

Addition
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Multiplication
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \circ \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

Power
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \wedge \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 27 & 256 \end{bmatrix}$$

1.3 Vector and Matrix Operations

The dot product (sometimes called the scalar product or the matrix product) and the transpose are two common operations that apply to vectors and matrices. The dot product operates in the following manner:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\times 5+2\times 7 & 1\times 6+2\times 8 \\ 3\times 5+4\times 7 & 3\times 6+4\times 8 \end{bmatrix}$$

For two matrices **A** and **B**, the matrix product can be simply written as **AB**. Note that **AB**≠**BA**.

The transpose takes a matrix and exchanges all of its rows for all of its columns. This can also be seen as a reflection across the matrix's main diagonal. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

The transpose can also be applied to vectors and matrices that are not square:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}^T = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Naturally, $(\mathbf{A}^T)^T=\mathbf{A}$.

2. High-Dimensional Data

In machine learning, data rarely takes on two or three dimensions. Often, thousands are involved. It is very tempting to attempt to visualize this data under the assumption that succeeding will aid in visualizing the current learning problem. However, this is a futile endeavor that ends in frustration. There are a few workarounds and paradigms for analyzing such data that often prove extremely useful.

Dimensions are simply a way of storing information. Stating that some data has *n* dimensions simply means we have *n* different categories of knowledge about the dataset. Consider a dataset of people. Perhaps we know each person's age, height, weight, and income. This dataset may have four dimensions, but there is no need to attempt to visualize this data in a four-dimensional space.

Simplify models of spaces and hyperplanes. At times, even if visualizing a high-dimensional dataset is not practical, we need some mental model for the object. For a space of some dimensionality *n*, consider a simple cube or rectangular prism. For a hyperplane (dimensionality n-1), consider a square or rectangular sheet. This paradigm is illustrated in Figure 1.

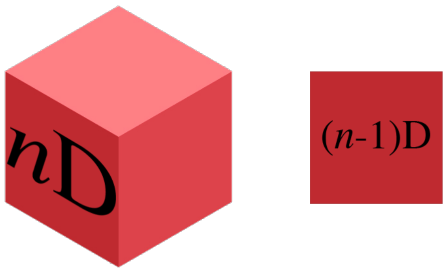

Fig. 1: Simplified models for visualizing spaces and hyperplanes as cubes and planes.

View dimensions iteratively when a better model is needed. Consider a two-dimensional sheet of paper. Stacking many of these yields a three dimensional block. To form a fourth-dimensional object, form a stack of blocks, etc. To access data in this fourth (or nth) dimensional object, all that is required is a x and y coordinate, along with a sheet number (specifies z), and a block number (specifies the forth dimension). The paradigm is illustrated in Fig. 2.

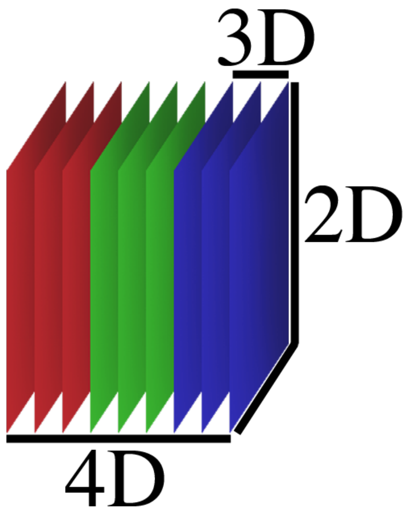

Fig. 2: Simplified model for visualizing high-dimensional data as an extension of the third dimension.

3. Linear Regression

3.1 Two-Dimensional Model

Consider the general equation of a line in a plane:

$$f(x) = ax + b \tag{1}$$

Suppose we have a dataset **D** which contains a measured y_i value for every x_i in the following format:

$$\mathbf{D} = \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} \tag{2}$$

We wish to develop the model that best fits the dataset. Provided this dataset is linear, a linear fit will also be able to predict new y_i values given an input x_i.

Any possible linear model is defined by two parameters: a and b. We wish to pick the *best* values for these parameters. In order to choose the *best* model, we need to develop a method of calculating quality. We define the least mean squared error of the system as follows:

$$E = \frac{1}{2n}\sum_{i=1}^{n}\left(y_i - f(x_i)\right)^2 \tag{3}$$

In (3), we define the error as the *label* y_i less the *hypothesis* (the prediction of $f(x)$) evaluated at some x_i. This is simply the vertical distance between predicted and actual y_i value of a given datapoint. It is convenient to square this error, as error cannot be negative because this would imply a better than perfect prediction. Summing these calculated squared error values for all datapoints yields the total error of the system. It is convenient to take an average of this error by dividing by *n*. The factor of two in the denominator of (3) is added arbitrarily to ease future computations.

After we initialize our parameters *a* and *b* randomly, we will need some indication of how to change these parameters so as to reduce the error computation (3). That is, we wish to find how the error changes with respect to the parameters. Those familiar with calculus will recognize this operation as a derivative. Those without knowledge of calculus may consider the derivative operator as a "magic box" that returns a function that demonstrates how one variable changes when another is changed (e.g. how *f*(*x*) changes when *a* is changed).

In order to make the mathematics sensible, we must rewrite (2) follows:

$$\mathbf{D} = \begin{bmatrix} x_{0,0} & x_{0,1} & y_0 \\ x_{1,0} & x_{1,1} & y_1 \\ \vdots & \vdots & \vdots \\ x_{n,0} & x_{n,1} & y_n \end{bmatrix} \quad (4)$$

We also rewrite (1) for one data sample only.

$$f(\mathbf{x}_i) = \theta_0 x_{i,0} + \theta_1 x_{i,1} \quad (5)$$

The input $\mathbf{x}_i$ of (5) is a vector, as follows,

$$\mathbf{x}_i = \begin{bmatrix} x_{i,0} & x_{i,1} \end{bmatrix} \quad (6)$$

In (6), $x_{i,1}$ is *any* sample from column 1 of **D**. Considering (5) as a rewritten form of (1), θ_0=b, $x_{i,0}$=1 (always, by definition), θ_1=*a*, and $x_{i,1}$=*x*. The rationale for this notation will become clear during the generalization from two to *n* dimensions.

The analytical derivative of (3) is written,

$$\frac{dE}{d\boldsymbol{\theta}} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\mathbf{x}_i)\right)\mathbf{x}_i \quad (7)$$

As there are two variables involved in (6) and therefore also in (7), and we wish to vary only one at a time, the derivative involved is actually two partial derivatives. A partial derivative is a derivative used in a formula with multiple, independent variables when all but one variable are held constant, and may be understood in much the same manner as a derivative. Hence, the changes we must make to our parameter vector, written

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 & \theta_1 \end{bmatrix} \quad (8)$$

are given by the following two updates:

$$\Delta\theta_0 = \alpha\frac{\partial E}{\partial\theta_0} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\mathrm{x}_{i,0})\right)\mathrm{x}_{i,0} \quad (9)$$

$$\Delta\theta_1 = \alpha\frac{\partial E}{\partial\theta_1} = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\mathrm{x}_{i,1})\right)\mathrm{x}_{i,1} \quad (10)$$

In (9) and (10), α is the *learning constant*, which is an experimentally found number used to scale the **θ** updates. Provided α is small enough, these updates will always converge to the best set of parameters **θ** that can be used to model the linear system. Note that the updates given by (9) and (10) will be subtracted from θ_0 and θ_1 because the partial derivatives give the direction of increase of the squared error function with respect to θ_0 and θ_1 respectively, and we wish to minimize the squared error.

3.2 Generalization to *n* Dimensions

The method of minimizing the error in a two-dimensional, linear system generalizes to n dimensions. Applied to the error in (7), it may be used to fit a hyperplane to a space. Suppose we have a dataset and labeled column vector, along with a parameter row vector, as follows,

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,n} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,n} \\ \cdots & \cdots & \ddots & \vdots \\ x_{m,0} & x_{m,1} & \cdots & x_{m,n} \end{bmatrix} \qquad (11)$$

$$\mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix} \qquad (12)$$

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 & \theta_1 & \cdots & \theta_n \end{bmatrix} \qquad (13)$$

Then we can generalize (5) as follows:

$$h(\mathbf{x}_i) = \theta_0 x_{i,0} + \theta_1 x_{i,1} + ... + \theta_1 x_{i,n} = \theta \mathbf{x}_i^{\mathsf{T}} \qquad (14)$$

We will refer to (14) as the *activation*, as it may be called as h($\mathbf{X}$) in order to compute a row vector containing (14) applied to all m rows of data in $\mathbf{X}$ (we will call each of these rows a *training sample* made up of all *features* (elements) in $\mathbf{x}_i$). The activation vector is shown below,

$$\mathbf{a} = \begin{bmatrix} a_0 & a_1 & \cdots & a_n \end{bmatrix} \qquad (15)$$

This activation vector contains all of the predicted values of y_i using the current set of parameters $\boldsymbol{\theta}$, for all training samples $\mathbf{x}_i$. So we may write the squared error function by using the difference between the actual labels given by $\boldsymbol{y}$ and the activation vector of predictions $h(\mathbf{X})$.

$$E = \frac{1}{2m} \sum \left(y^{\mathsf{T}} - h(\mathbf{X}) \right)^2 \qquad (16)$$

Now we may obtain our gradient—which is a vector of partial derivatives with respect to each parameter—by writing a vectorized form of (7) as follows:

$$\nabla_{\boldsymbol{\theta}} E = \frac{\partial E}{\partial \boldsymbol{\theta}} = \frac{1}{m} \left(y^{\mathsf{T}} - h(\mathbf{X}) \right) \mathbf{X} \qquad (17)$$

In (17), ∇ (referred to as del or nabla) is the gradient operator, and the subscript $\boldsymbol{\theta}$ (vector form, not clearly visible in formula) denotes that the gradient is applied with respect to each θ_j in $\boldsymbol{\theta}$. The result of (17) is a vector of θ_j updates to $\boldsymbol{\theta}$. We write the final, iterative update to $\boldsymbol{\theta}$ as follows,

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} E \qquad (18)$$

In (18), := is the assignment operator which updates the value of the quantity to its left. This method of iteratively subtracting a portion of $\nabla_{\boldsymbol{\theta}} E$ to minimize a function is referred to as the *gradient decent algorithm*.

3.3: Applicability to Realistic Learning Problems

Gradient decent is applicable to much more intricate problems. The system discussed in 3.2 is a linear system; gradient decent may be applied to different $h(\mathbf{x}_i)$ functions and may be used to model highly nonlinear systems. The only prerequisite for the algorithm is that the squared error function (or any other sort of non-negative error function) be *convex*. In general terms, the term *convex* refers to any bowl-shaped function in any number of dimensions, as shown in Fig. 3.

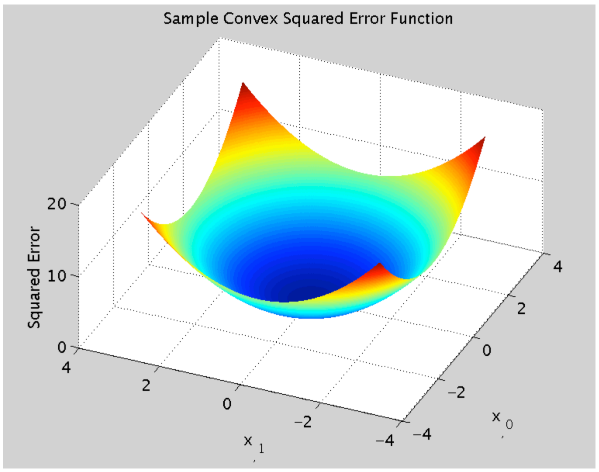

Fig. 3: Sample convex squared error function with two column vectors of features, $\mathbf{x}_1$ and $\mathbf{x}_2$.

4. Conclusion

High-dimensional data visualization techniques, feature representation as matrices, use of matrix multiplication in order to vectorize learning algorithms, and the gradient decent algorithm (along with other convex optimization algorithms) are all essential modules common to learning algorithms. The practice of writing and then minimizing some measure of a system's error generalizes to almost all of machine learning techniques implemented at a professional level. Such techniques may be used in conjunction with more complex algorithms in order to create software used for autonomous driving, optical character recognition (OCR), search page ranking, financial engineering, etc.