Linear regression

Much of machine learning is about fitting functions to data. That may not sound like an exciting activity that will give us artificial intelligence. However, representing and fitting functions is a core building block of most working machine learning or AI systems. We start with linear functions, both because this idea turns out to be surprisingly powerful, and because it's a useful starting point for more interesting models and methods.

1 Affine functions

An *affine function* (also called a linear function[1]) of a vector $\mathbf{x}$ takes a weighted sum of each input and adds a constant. For example, for $D=3$ inputs $\mathbf{x} = [x_1\ x_2\ x_3]^\top$, a general (scalar) affine function is:

$$f(\mathbf{x}; \mathbf{w}, b) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b = \mathbf{w}^\top \mathbf{x} + b, \tag{1}$$

where $\mathbf{w}$ is a D-dimensional vector of "weights". The constant offset or "bias weight" b gives the value of the function at $\mathbf{x} = \mathbf{0}$ (the origin). In machine learning, the numbers in $\mathbf{x}$ are the *input features* (also called just "*inputs*" or "*features*") describing a setting that we want our model to consider.

Some example functions are below. These models are wrong, but you can think about whether they might be useful, and might think of better models.

1) **Spammyness of an email:** inputs $\mathbf{x}$ could be zeros and ones for presence/absence of words.
2) **Predicted strength of a metal alloy:** inputs $\mathbf{x}$ could be the amounts of different metals we include.
3) **Predicted height a sunflower plant will grow:** a simple model might use only one input $\mathbf{x}$ (or a scalar, x): the amount of fertilizer we add to a sunflower.

[The website version of this note has a question here.]

2 Fitting to data

[Please see our background page on notation if you have queries.]

We will fit the function to a training set of N input-output pairs $\{\mathbf{x}^{(n)}, y^{(n)}\}_{n=1}^N$. For example we make N different alloys, with the amounts of metals described by the inputs $\{\mathbf{x}^{(n)}\}$, and measure the *target outputs*, the true strengths of these alloys $\{y^{(n)}\}$, which we would like our model to predict.

Expressing the data and function values using linear algebra, makes the maths easier in the long run, and makes it easier to write fast array-based code. We stack all of the observed outputs into a column vector $\mathbf{y}$, and the inputs into an $N\times D$ "design matrix" X:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix}, \qquad X = \begin{bmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \vdots \\ \mathbf{x}^{(N)\top} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_D^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_D^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_D^{(N)} \end{bmatrix}. \tag{2}$$

1. The Wikipedia page for *Linear function* explains the terminology.

Elements of these arrays are $y_n = y^{(n)}$, and $X_{n,d} = x_d^{(n)}$. Each row of the design matrix gives the features for one training input vector. We can simultaneously evaluate the function at every training input with one matrix-vector multiplication:

$$\mathbf{f} = X\mathbf{w} + b, \tag{3}$$

where the scalar bias b is added to each element of the vector $X\mathbf{w}$. Each element of the vector on the left-hand side gives the function at one training input: $f_n = f(\mathbf{x}^{(n)}; \mathbf{w}, b) = \mathbf{w}^\top \mathbf{x}^{(n)} + b$.

We can compute the total square error of the function values above, compared to the observed training set values:

$$\sum_{n=1}^{N} \left[y^{(n)} - f(\mathbf{x}^{(n)}; \mathbf{w}, b) \right]^2 = (\mathbf{y} - \mathbf{f})^\top (\mathbf{y} - \mathbf{f}). \tag{4}$$

The least-squares fitting problem is finding the parameters that minimize this error.

[The website version of this note has a question here.]

2.1 Fitting weights with $b = 0$

To keep the maths simpler, we will temporarily assume our function goes through the origin. That is, we'll assume $b = 0$. Thus we are fitting the "linear map":

$$\mathbf{y} \approx \mathbf{f} = X\mathbf{w}. \tag{5}$$

Fitting $\mathbf{w}$ to this approximate relationship by least-squares is so common that programming languages for scientific computing like Matlab/Octave make fitting the parameters astonishingly easy[2]:

```
% Matlab sizes: w_fit is Dx1 if X is NxD and yy is Nx1
w_fit = X \ yy;
```

In this course we're using Python+NumPy, so the code to fit the weights isn't quite so short, but is still one line of code:

```
# NumPy shapes: w_fit is (D,) if X is (N,D) and yy is (N,)
w_fit = np.linalg.lstsq(X, yy, rcond=None)[0]
```

2.2 Fitting more general functions

We'll return to how the `lstsq` fitting routine works later. For now, we'll assume it's available, and see what we can do with it.

In machine learning it's often the case that the same code can solve different tasks simply by using it on different representations of our data. In the rest of this note, and the next, we will solve multiple tasks with the same linear least-squares primitive operation.

We assumed that our function passed through the origin. We can remove that assumption simply by creating a new version of our design matrix. We add an extra column that contains a one in every row:

$$\tilde{X} = \begin{bmatrix} \mathbf{x}^{(1)\top} & 1 \\ \mathbf{x}^{(2)\top} & 1 \\ \vdots & \vdots \\ \mathbf{x}^{(N)\top} & 1 \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_D^{(1)} & 1 \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_D^{(2)} & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_D^{(N)} & 1 \end{bmatrix}. \tag{6}$$

In Python with NumPy imported:

```
X_bias = np.concatenate([X, np.ones((X.shape[0],1))], axis=1) # (N,D+1)
```

2. For more on what the backslash "\" does in Matlab, you can see the documentation for mldivide.

Then we fit least squares weights exactly as before, just using $\tilde{X}$ or `X_bias`, instead of the original design matrix X. If our input was D-dimensional before, we will now fit $D+1$ weights, $\tilde{\mathbf{w}}$. The last of these is always multiplied by one, and so is actually the bias weight b, while the first D weights give the regression weights for our original design matrix:

$$\tilde{X}\tilde{\mathbf{w}} = X\tilde{\mathbf{w}}_{1:D} + \tilde{w}_{D+1} = X\mathbf{w} + b. \tag{7}$$

2.2.1 Polynomials

We can go further, replacing the design matrix with a new matrix Φ. Each row of this matrix is an arbitrary vector-valued function of the original input: $\Phi_{n,:} = \boldsymbol{\phi}(\mathbf{x}^{(n)})^\top$. If the function is non-linear, then our function $f(\mathbf{x}) = \mathbf{w}^\top\boldsymbol{\phi}(\mathbf{x})$ will be non-linear in $\mathbf{x}$. However, we can still use linear-regression code to fit the model, as the function is still a linear map of a known vector, $\boldsymbol{\phi}(\mathbf{x})$.

The introductory example you'll see in most textbooks is fitting a polynomial curve to one-dimensional data. Each column of the new design matrix Φ is a monomial of the original feature:

$$\Phi = \begin{bmatrix} 1 & x^{(1)} & \left(x^{(1)}\right)^2 & \cdots & \left(x^{(1)}\right)^{K-1} \\ 1 & x^{(2)} & \left(x^{(2)}\right)^2 & \cdots & \left(x^{(2)}\right)^{K-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x^{(N)} & \left(x^{(N)}\right)^2 & \cdots & \left(x^{(N)}\right)^{K-1} \end{bmatrix}. \tag{8}$$

Using Φ as our design matrix or data matrix we can then fit the model

$$y \approx f = \mathbf{w}^\top\boldsymbol{\phi}(x) = w_1 + w_2 x + w_3 x^2 + \cdots + w_K x^{K-1}, \tag{9}$$

which is a general polynomial of degree $K-1$.

We could generalize the transformation for multivariate inputs,

$$\boldsymbol{\phi}(\mathbf{x}) = [1 \;\; x_1 \;\; x_2 \;\; x_3 \;\; x_1x_2 \;\; x_1x_3 \;\; x_2x_3 \;\; x_1^2 \;\; \ldots]^\top, \tag{10}$$

and hence fit a multivariate polynomial function of our original features. Given that a general polynomial includes cross terms like x_1x_3, $x_1x_2x_3$, the number of columns in Φ could be large.

Polynomials are usually taught in introductions to regression first, because the idea of fitting a polynomial curve may already be familiar. However, polynomial fits are not actually used very often in machine learning. They're probably avoided for two reasons. 1) The feature space grows quickly for high-dimensional inputs; 2) Polynomials rapidly take on extreme values as the input $\mathbf{x}$ moves away from the origin. Of course there are exceptions[3].

2.2.2 Basis functions

Instead of creating monomial features, we can transform our data with any other vector-valued function:

$$\boldsymbol{\phi}(\mathbf{x}) = [\phi_1(\mathbf{x}) \;\; \phi_2(\mathbf{x}) \;\; \ldots \;\; \phi_K(\mathbf{x})]^\top. \tag{11}$$

By convention, each ϕ_k is called a *basis function*. The function we fit is a linear combination of these basis functions:

$$f(\mathbf{x}) = \mathbf{w}^\top\boldsymbol{\phi}(\mathbf{x}) = \sum_k w_k\phi_k(\mathbf{x}). \tag{12}$$

We don't have to include a constant or bias term in the mathematics, because we can always set one of the ϕ_k functions to a constant.

3. One case where I might consider polynomials is when I have sparse binary features. That is $x_d \in \{0,1\}$ where only a few of the features are non-zero. If the features are binary, none of the polynomial terms take on extreme values. 'Interaction terms', such as x_1x_3, detect whether it's important for features to be on at the same time. These extra features are more sparse than the original features, and in careful implementations might not create much more work.

One possible choice for a basis function ϕ_k is a *Radial Basis Function* (RBF):

$$\exp(-(\mathbf{x}-\mathbf{c})^\top(\mathbf{x}-\mathbf{c})/h^2), \tag{13}$$

where different basis functions can have different parameters $\mathbf{c}$ and h. The function is proportional to a Gaussian probability density function (although it is not a probability density in this context). The function has a bell-curve shape centred at $\mathbf{c}$, with 'bandwidth' h. The bell-curve shape is radially symmetric: the function value only depends on the radial distance from the centre $\mathbf{c}$.

Another possible choice is a *logistic-sigmoid* function:

$$\sigma(\mathbf{v}^\top\mathbf{x}+b) = \frac{1}{1+\exp(-\mathbf{v}^\top\mathbf{x}-b)}. \tag{14}$$

This sigmoidal or "s-shaped" curve saturates at zero and one for extreme values of $\mathbf{x}$. The parameters $\mathbf{v}$ and b determine the steepness and position of the curve.

For each column of the matrix of basis functions Φ, we will choose a basis function, along with its free parameters such as $\{\mathbf{c}, h\}$ for an RBF, or $\{\mathbf{v}, b\}$ for the logistic-sigmoid. After sketching/plotting these basis functions (next note) you should develop some intuitions for setting these parameters by hand. Ultimately we will also want to set these parameters automatically.

[The website version of this note has a question here.]

3 Summary

Using just linear regression, one line of fitting code, we can fit flexible regression models with many parameters. The trick is to construct a large design matrix, where each column corresponds to a basis function evaluated on each of the original inputs. Each basis function should have a different position and/or shape.

Models built with radial-basis functions and sigmoidal functions extrapolate more conservatively than polynomials for extreme inputs. Radial-basis functions change the function close to a reference point, while sigmoidal functions change the function in half of the space. Neither of these families of basis functions has fundamental status however, and other basis functions are also used.