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.

A general linear function 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) linear function is:

$$f(\mathbf{x}; \mathbf{w}, b) = w_1x_1 + w_2x_2 + w_3x_3 + b = \mathbf{w}^\top\mathbf{x} + b,$$

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).

We will fit the function to a training set of N input-output pairs $\{\mathbf{x}^{(n)}, y^{(n)}\}_{n=1}^N$. 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:

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

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 linear function at every training input with one matrix-vector multiplication:

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

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:

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

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

Fitting weights with $b = 0$

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

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

Fitting $\mathbf{w}$ to this approximate relationship by least-squares is so common that Matlab/Octave makes fitting the parameters astonishingly easy [1]:

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

With NumPy, fitting the weights is still one line of code:

```
    # shapes: w_fit is (D,) if X is (N,D) and yy is (N,)
    w_fit = np.linalg.lstsq(X, yy)[0]
```

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

Fitting more general functions

We'll return to how least squares fitting routines (`\` or `lstsq`) work later. For now, we'll assume they're available, and see what we can do with them.

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 different tasks all 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 containing a one in every row:

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

In Matlab/Octave:

```
    X_bias = [ones(size(X,1),1) X]; % Nx(D+1)
```

In Python

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

Then we fit least squares weights exactly as before, just using 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, $\mathbf{w}'$. The first of these is always multiplied by one, and so is actually the bias weight b, while the remaining weights give the regression weights for our original design matrix:

$$X'\mathbf{w}' = X\mathbf{w}'_{2:D+1} + w'_1 = X\mathbf{w} + b.$$

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 model is still linear in the parameters.

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:

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

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

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

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

We could generalize the transformation for multivariate inputs,

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

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 $\boldsymbol{\phi}$ 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[2].

Basis functions

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

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

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

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

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.

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

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

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:

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

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 (see below) you should develop some intuitions for setting these parameters by hand. Ultimately we will also want to set these parameters automatically.

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 tend to zero, and sigmoidal functions tend to a constant. Neither of these families of basis functions has fundamental status however, and other basis functions are also used.

The rest of this note is encouraging you to think for yourself. I usually don't feel I really understand some maths, unless I can turn it into code. Implementing something forces me to confront every detail. Looking at special cases in code also helps build intuitions.

2. 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. Moreover, 'higher-order' features, e.g. $x_1 x_3$, are more sparse than the original features, and in careful implementations might not create much more work.

Checking your understanding

First, some questions to check your mathematical understanding:

- A fellow student asks: *"Why when evaluating one function value do the equations contain* $\mathbf{w}^\top\mathbf{x}^{(n)}$*, but then the order swaps to* $X\mathbf{w}$ *in a later equation?"* — What do you tell them?
- Check whether you know what sort of object each symbol in each equation is. Scalar, column vector, or matrix; and of what size? For example what are the sizes of $\mathbf{c}$ and $\mathbf{v}$, the parameters of the RBF and logistic-sigmoid basis functions above?
- An RBF basis function is centred at some position $\mathbf{c}$. We can create multiple columns of Φ by evaluating RBFs with different centres, from a set $\{\mathbf{c}^{(k)}\}$. For one-dimensional curve fitting (to keep the algebra simpler), why does it not make sense to create a family of quadratic functions $\phi_k(x) = (x - c^{(k)})^2$, and include several features with different centres $c^{(k)}$?

You should also understand how the maths would translate to code, and actually *doing* regression. I give here a quick review of how to plot functions in Matlab/Octave or Python, and demonstrate how to plot different basis functions, and linear regression fits.

One dimensional basis functions

We can plot one-dimensional functions by evaluating them on a fine grid. For example, the cosine function:

In Matlab/Octave:

```
grid_size = 0.1;
x_grid = -10:grid_size:10;
f_vals = cos(x_grid);
clf; hold on;
plot(x_grid, f_vals, 'b-');
plot(x_grid, f_vals, 'r.');
```

In Python:

```
import numpy as np
import matplotlib.pyplot as plt

grid_size = 0.1
x_grid = np.arange(-10, 10, grid_size)
f_vals = np.cos(x_grid)
plt.clf()
plt.plot(x_grid, f_vals, 'b-')
plt.plot(x_grid, f_vals, 'r.')
plt.show()
```

We're not really evaluating the whole function. We just evaluated it at the points shown by red dots. However, the blue curve joining these dots closely approximates our function. If we decrease the `grid_size` further, the blue line will become as accurate as it's possible to plot on a computer screen.

We could use the cosine function as a basis function in our model. We can also create and plot other basis functions.

Matlab/Octave:

```
% Matlab/Octave functions usually need to be defined in a file.
% To make a simple RBF function, create a file rbf_1d.m containing:
%function vals = rbf_1d(xx, cc, hh)
%vals = exp(-(xx-cc).^2 / hh.^2);
```

```
%
% But one-line functions can be created anywhere like this:
rbf_1d = @(xx, cc, hh) exp(-(xx-cc).^2 / hh.^2);

clf(); hold on;
grid_size = 0.01;
x_grid = -10:grid_size:10;
plot(x_grid, rbf_1d(x_grid, 5, 1), '-b');
plot(x_grid, rbf_1d(x_grid, -2, 2), '-r');
```

Python:

```
# Unlike Matlab, function definitions can be made anywhere
# in Python. They don't have to go in a file.
def rbf_1d(xx, cc, hh):
    return np.exp(-(xx-cc)**2 / hh**2)
# An alternative (equivalent to the Matlab, but not recommended):
# rbf_1d = lambda xx, cc, hh: np.exp(-(xx-cc)**2 / hh**2)

plt.clf()
grid_size = 0.01
x_grid = np.arange(-10, 10, grid_size)
plt.plot(x_grid, rbf_1d(x_grid, cc=5, hh=1), '-b')
plt.plot(x_grid, rbf_1d(x_grid, cc=-2, hh=2), '-r')
plt.show() # I may forget sometimes. Not necessary in python --pylab
```

The code above plots a radial basis function centred at $x = 5$ in blue, and one in red that's twice as wide and centred at $x = -2$.

Exercise: You should plot logistic-sigmoid functions $\sigma(vx+b) = 1/(1+e^{-vx-b})$ in one-dimension for different v and b. What is the effect of the parameters v and b?

Combinations of basis functions

You might think of the function $f(\mathbf{x}) = \sum_k w_k \phi_k(\mathbf{x})$ as being made up of separate parts ϕ_k. For example, plot the function

$$f(x) = 2\phi_1(x) - \phi_2(x),$$

where ϕ_1 and ϕ_2 are unit bandwidth RBFs centred at -5 and $+5$. You can identify two separate bumps, each a scaled version of a basis function.

However, it's frequently not obvious by eye what the underlying basis functions are. For example, we can create a function with monomial basis functions

$$\begin{aligned}\phi_1(x) &= 1\\ \phi_2(x) &= x\\ \phi_3(x) &= x^2,\end{aligned}$$

and weights $\mathbf{w} = [5 \;\; 10 \;\; -1]^\top$. The resulting function has a simple form,

$$f(x) \;=\; \mathbf{w}^\top \boldsymbol{\phi}(x) \;=\; 5 + 10x - x^2 \;=\; 30 - (x-5)^2,$$

which, unlike any of the basis functions, peaks at $x = 5$. Similarly, a function made by combining several overlapping RBFs can have peaks in between the peaks of any of the underlying basis functions.

Linear regression

Consider a dataset with 3 datapoints:

$$\mathbf{y} = \begin{bmatrix} 1.1 \\ 2.3 \\ 2.9 \end{bmatrix}, \qquad X = \begin{bmatrix} 0.8 \\ 1.9 \\ 3.1 \end{bmatrix}.$$

By working through the following Matlab/Octave demo (or porting to Python/NumPy, a useful exercise), you should make sure you know how to fit and plot a variety of linear regression models.

```
% Set up and plot the dataset
yy = [1.1 2.3 2.9]';
X = [0.8 1.9 3.1]';
clf(); hold all;
plot(X, yy, 'x', 'MarkerSize', 20, 'LineWidth', 2);

% phi-functions to create various matrices of new features
% from an original matrix of 1D inputs.
phi_linear = @(Xin) [ones(size(Xin,1),1) Xin];
phi_quadratic = @(Xorig) [ones(size(Xorig,1),1) Xorig Xorig.^2];
fw_rbf = @(xx, cc) exp(-(xx-cc).^2 / 2);
phi_rbf = @(Xin) [fw_rbf(Xin, 1) fw_rbf(Xin, 2) fw_rbf(Xin, 3)];

fit_and_plot(phi_linear, X, yy); % Helper routine (see below)
fit_and_plot(phi_quadratic, X, yy);
fit_and_plot(phi_rbf, X, yy);
legend({'data', 'linear fit', 'quadratic fit', 'rbf fit'});
```

Using the following helper routine saved as `fit_and_plot.m`:

```
function fit_and_plot(phi_fn, X, yy)
% phi_fn takes Nx1 inputs and returns NxK basis function values
w_fit = phi_fn(X) \ yy; % Kx1
X_grid = (0:0.01:4)'; % Nx1
f_grid = phi_fn(X_grid)*w_fit;
plot(X_grid, f_grid, 'LineWidth', 2);
```

It is possible to fit any three points exactly using a model with three basis functions (which means $\mathbf{f} = \mathbf{y}$). As long as the basis functions create an $N \times K$ or 3×3 feature matrix Φ that is invertible[3], we can set the weights to $\mathbf{w} = \Phi^{-1}\mathbf{y}$. We can write down lots of different models with three basis functions that will agree on the outputs for the three training inputs. However, the predictions will usually be different for new *test* inputs. Predicting the future given past data is an inherently *ill-posed* problem.

Aside on terminology and notation

[This section is non-examinable. It's intended to help you read other resources.]

Textbooks and code documentation use various different names and notations for the things covered in this note. While learning a subject, these differences can be confusing. However, dealing with different notations is a necessary research skill.

3. It turns out that for most of the basis functions we use, including polynomials or RBFs with different centres, when applied to different datapoints, the feature matrix Φ will be invertible when there are as many basis functions as datapoints. Bishop's (1995) *Neural Networks for Pattern Recognition* book defers to Micchelli (1984), *Interpolation of scattered data: Distance matrices and conditionally positive definite functions*, for the technical details.

The "weights" $\mathbf{w}$ are the parameters of the model, or "regression coefficients". In statistics textbooks and papers they are often given the symbol β. A constant offset or intercept b could have various other symbols, including β_0, w_0, or c. In the neural network literature this parameter is called the "bias weight" or simply the "bias". There's an unfortunate clash in terminology: a "bias weight" does not set the statistical "bias" of the model (its expected test error, averaged over different training sets).

Where possible, these notes use lower-case letters for indexing, with the corresponding capital letters for the numbers of settings. For example, training data-points usually use the index n running from $n = 1...N$, feature dimensions use $d = 1...D$, and components of a model might be $k = 1...K$. As a result, the notation won't match some textbooks. In statistics it's common to index data-items with $i = 1...n$, and parameters with $j = 1...p$.

While most textbooks use subscripts for both features and items, these notes usually identify the nth vector of a dataset with a bracketed super-script $\mathbf{x}^{(n)}$. Those superscripts occasionally get in the way, but make a clear distinction from feature dimensions, which are always subscripts. The dth feature of the nth datapoint is thus $x_d^{(n)}$. When we cover Gaussian processes later in the course, this notational baggage may help avoid confusion.

In statistics it's standard for the design matrix X to be an $N \times D$ matrix, containing datapoints as row vectors. However machine learning code occasionally expects your data to be stored in a $D \times N$ array. It is worth double-checking that your matrices are the correct way around when calling library routines, and leave comments in your own code to document the intended sizes of arrays. Sometimes I leave a comment at the end of a line that simply gives the size or shape of the result: for example, "`% NxD`" in Matlab, or "`# (N,D)`" in Python.