Linear regression, overfitting, and regularization

The fitted function $f(\mathbf{x})$ doesn't usually match the training data exactly. Each training item has a *residual*, $(y^{(n)} - f(\mathbf{x}^{(n)}))$, which is normally non-zero. Why don't we get perfect fits?

- Data is usually inherently noisy or stochastic, in which case it's impossible to exactly predict y from $\mathbf{x}$. For example, if a builder mixes several batches of concrete with the same quantities specified by $\mathbf{x}$, we wouldn't expect their observed strengths y to be *exactly* the same.
- Even if the outputs are noiseless, $N > D$ data-points are unlikely to lie exactly on any function represented by a linear combination of our D basis functions.

If we don't include enough basis functions, we will *underfit* our data. For example, if some points lie exactly along a cubic curve:

```
N = 100; D = 1;
X = rand(N, D) - 0.5; % np.random.rand in NumPy
yy = X.^3; % X**3 in NumPy
```

We would not be able to fit this data accurately if we only put linear and quadratic basis functions in our augmented design matrix Φ.

You could fit the cubic data above fairly accurately with a few RBF functions. The fit wouldn't extrapolate well outside the $x \in (-0.5, 0.5)$ range of observations, but you can get an accurate fit close to where there is data. To avoid *underfitting*, we need a model with enough representational power to closely approximate the underlying function.

When the number of training points N is small, it's easy to fit the observations with low square error. In fact, usually if we have N or more basis functions, such as N RBFs with different centres, the residuals will all be zero![1] However, we should not trust this fit. It's hard for us as intelligent humans to guess what an arbitrary function is doing in between only a few observations, so we shouldn't believe the result of fitting an arbitrary model either. Moreover, if the observations are noisy, it seems unlikely that a good fit should match the observed data exactly anyway.

As advocated by Acton in a previous note, one possible approach to modelling is as follows. Start with a simple model with only a few parameters. This model may underfit, that is, not represent all of the structure evident in the training data. We could then consider a series of more complicated models while we feel that fitting these models can still be justified by the amount of data we have.

However, limiting the number of parameters in a model can be problematic. If our inputs have many features, even simple linear regression (without additional basis functions) has many parameters. An example is predicting some characteristic of an organism (a *phenotype*) from DNA data, which is often in the form of $> 10^5$ features called SNPs. We could consider doing some preliminary filtering to remove features, although this choice is itself a fit of sorts to the data. Moreover, filtering is not always the correct answer. Perhaps we have features that are noisy measurements of the same underlying property. In that case we should average the measurements rather than selecting one of them. However, we may not know in advance which groups of features should be averaged, and which selected.

Another approach to modelling is to use large models, but to disallow unreasonable fits that match our noisy training data too closely.

Examples of what overfitting can look like

Example 1: Fitting many features. We will consider a synthetic situation where each datapoint relates to a different underlying quantity $\mu^{(n)}$. We will sample each of these quantities from

1. The basis functions need to produce N linearly-independent columns in the feature or design matrix Φ. Most basis functions do have this property, but the technical details are too involved to get into here. There's a reference in the previous note.

a uniform distribution between zero and one, which we write:

$$\mu^{(n)} \sim \text{Uniform}(0, 1)$$

In our example, the features and output will both be noisy measurements of the underlying quantity:

$$x_d^{(n)} \sim \mathcal{N}(\mu^{(n)}, 0.01^2), \quad d = 1 \dots D$$
$$y^{(n)} \sim \mathcal{N}(\mu^{(n)}, 0.1^2).$$

The notation $\mathcal{N}(\mu, \sigma^2)$ means the values are drawn from a Gaussian or Normal distribution centred on μ, with variance σ^2.

In this situation, averaging the x_d measurements would be a reasonable estimate of the underlying feature μ, and hence the output y. Thus a regression model with $w_d = 1/D$ would make reasonable predictions of y from $\mathbf{x}$. Do you recover something like this model if you fit linear regression? Try fitting randomly-generated datasets with various N and D. One of the following code snippets may help:

```
% Matlab, mu is Nx1
X = repmat(mu, 1, D) + 0.01*randn(N, D);
```

```
# Python, mu is (N,)
X = np.tile(mu[:,None], (1, D)) + 0.01*np.random.randn(N, D)
```

By making D large and N not much larger than D, the weights that give the best least-squares fit to the training data are much larger in magnitude than $w_d = 1/D$. By using weights much smaller and larger than the ideal values, the model can use small differences between input features to fit the noise in the observations. If we tried to interpret the value of a weight as meaning something about the corresponding feature, we would embarrass ourselves. However, the average of the weights is often close to $1/D$, and predictions on more data generated in the same way as above, might be ok. The model will generalize badly though—it will make wild predictions—if we test on inputs generated from $x_d \sim \mathcal{N}(\mu, 1)$.

Example 2: Explaining noise with many basis functions. Consider data drawn as follows:

$$x_d^{(n)} \sim \text{Uniform}[0, 1], \quad d = 1 \dots D$$
$$y^{(n)} \sim \mathcal{N}(0, 1).$$

The outputs have no relationship to the inputs. The predictor with the smallest average square error on future test cases is $f(\mathbf{x}) = 0$, its average square error will be one. We now consider what happens if we fit a model with many basis functions. If we use a high-degree polynomial, or many RBF basis functions, we can get a lower square training error than one. However, the error on new data would be larger. The fits usually have extreme weights with large magnitudes (for example $\sim 10^3$). There is a danger that for some inputs, the predictions could be extreme.

It's possible to represent a large range of interesting functions using weights with small magnitude. Yet least-square fits are often obtained with combinations of extremely positive and negative weights, obtaining fits that pass unreasonably close to noisy observations. We would like to avoid fitting these extreme weights.

Regularization

Penalizing extreme solutions can be achieved in various ways, and is called regularization. One form of regularization is to penalize the sum of the square weights in our cost function. This method has various names, including Tikhonov regularization, ridge regression, or L_2 regularization. For K basis functions, the regularized cost function is then:

$$\begin{aligned} E_\lambda(\mathbf{w}; \mathbf{y}, \Phi) &= \sum_{n=1}^{N} \left[y^{(n)} - f(\mathbf{x}^{(n)}; \mathbf{w}) \right]^2 + \lambda \sum_{k=1}^{K} w_k^2 \\ &= (\mathbf{y} - \Phi\mathbf{w})^\top (\mathbf{y} - \Phi\mathbf{w}) + \lambda \mathbf{w}^\top \mathbf{w}. \end{aligned}$$

For $\lambda = 0$ we only care about fitting the data, but for larger values of λ we trade-off the accuracy of the fit so that we can make the weights smaller in magnitude.

We can fit the regularized cost function with the same linear least-squares fitting routine as before. This time, instead of adding new features, we add new data items. If our original matrix of input features Φ is $N \times K$, for N data items and K basis functions, we add K rows to both the vector of labels and matrix of input features:

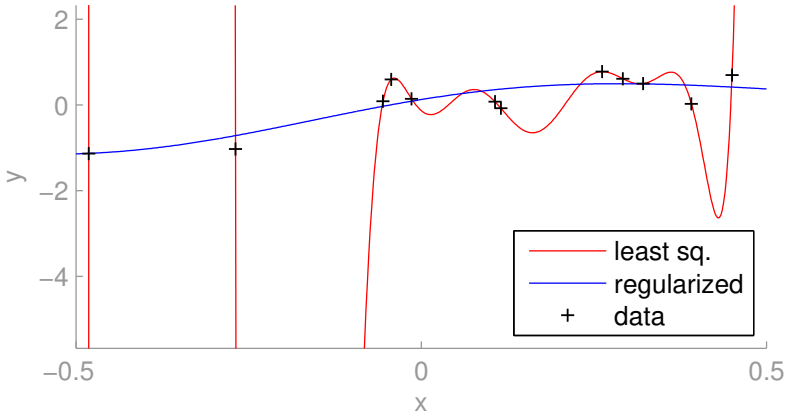
$$\mathbf{y}' = \begin{bmatrix} \mathbf{y} \\ \mathbf{0}_K \end{bmatrix} \quad \Phi' = \begin{bmatrix} \Phi \\ \sqrt{\lambda}\mathbb{I}_K \end{bmatrix},$$

where $\mathbf{0}_K$ is a vector of K zeros, and $\mathbb{I}_K$ is the $K \times K$ identity matrix. Then

$$\begin{aligned} E(\mathbf{w}; \mathbf{y}', \Phi') &= (\mathbf{y}' - \Phi'\mathbf{w})^\top (\mathbf{y}' - \Phi'\mathbf{w}) \\ &= (\mathbf{y} - \Phi\mathbf{w})^\top (\mathbf{y} - \Phi\mathbf{w}) + \lambda \mathbf{w}^\top \mathbf{w} = E_\lambda(\mathbf{w}; \mathbf{y}, \Phi). \end{aligned}$$

Thus we can fit training data $(\mathbf{y}', \Phi')$ using least-squares code that knows nothing about regularization, and fit the regularized cost function.

Below we see a situation where using least-squares with a dozen RBF basis functions leads to overfitting.



One could argue for changing the basis functions. However, as illustrated above, regularizing the same linear regression model can give less extreme predictions, at the expense of giving a fit further from the training points. The regularized fit depends strongly on λ. For $\lambda = 0$ we obtain the least squares fit. What happens as $\lambda \to \infty$?

Check your understanding

- Using either Matlab or Python, make sure you can generate some example data, construct a matrix of features Φ, and fit the regularized least squares linear regression weights. Plot the fitted function.
- We minimize the cost function $E_\lambda(\mathbf{w}; \mathbf{y}, \Phi)$ on the training set with respect to the weights $\mathbf{w}$. Why would it be a bad idea to minimize the same cost function as a way to set the regularization constant λ?
- If we have K radial basis functions, give a simple upper bound on the largest function value that could be obtained for a given weight vector $\mathbf{w}$. Also give a simple upper bound on the largest derivative (you could consider one dimensional regression). From such bounds we can see that limiting the size of the weights stops the function taking on extreme values, or changing extremely quickly.