Regression and Gradients

The previous note gave two approaches to classification where fitting the models had a closed-form solution. If you have previously learned some machine learning, you may know of other simple procedures, such as K-nearest neighbours that are also easy to describe and implement (at least for small scale problems). [1]

As we imagine building larger and more interesting models, we hit many questions. If we want to set the centres of many basis functions, how can we fit them to data instead of placing them by hand? Many algorithms, including for example K-nearest neighbours, give different answers if we linearly transform our variables before we start. How do we find a good transformation?[2] If we want a different cost function than least squares for regression, how can we fit the parameters?

Many such questions in machine learning are answered by rewriting the question as an optimization problem, and solving it numerically. Many, but not all, optimization problems are solved by *gradient-based methods*, which we begin to look at in this note. We start by looking at how linear least squares regression can be solved, so we can generalize it to other cases.

Gradients for linear least squares regression

We can create a vector of residuals (differences between observed values and function values) for a linear regression model as follows:

$$\mathbf{r} = \mathbf{y} - X\mathbf{w}.$$

And previously we noticed that Matlab and Numpy know how to minimize the sum of the square residuals:

$$\begin{aligned}\mathbf{r}^\top\mathbf{r} &= (\mathbf{y} - X\mathbf{w})^\top(\mathbf{y} - X\mathbf{w}) \\ &= \mathbf{y}^\top\mathbf{y} - 2\mathbf{w}^\top X^\top\mathbf{y} + \mathbf{w}^\top X^\top X\mathbf{w}.\end{aligned}$$

We'll now look at different ways that can be done.

Our first task is to find the *gradient* of this cost function with respect to the weights. That is, the vector of partial derivatives: $\nabla_{\mathbf{w}}\mathbf{r}^\top\mathbf{r}$. The gradient vector is a function of the weights. At a given position in weight-space, it points in the direction in which a small movement will increase the cost the most. You were asked to show this fact in the background self-test, question 6ii), which has an answer available if you need to review this material.

We can differentiate small matrix/vector expressions by writing them as sums, and using the elementary differentiation rules for scalars. For example:

$$\frac{\partial \mathbf{x}^\top\mathbf{y}}{\partial x_i} = \frac{\partial \sum_j x_j y_j}{\partial x_i} = y_i, \quad \Rightarrow \quad \nabla_{\mathbf{x}}[\mathbf{x}^\top\mathbf{y}] = \mathbf{y}.$$

and

$$\frac{\partial \mathbf{x}^\top A\mathbf{x}}{\partial x_i} = \frac{\partial \sum_{jk} x_j A_{jk} x_k}{\partial x_i} = \sum_k A_{ik}x_k + \sum_j x_j A_{ji},$$

$$\Rightarrow \quad \nabla_{\mathbf{x}}[\mathbf{x}^\top A\mathbf{x}] = A\mathbf{x} + A^\top\mathbf{x}, \quad \text{or } 2A\mathbf{x} \text{ if } A \text{ is symmetric.}$$

After some experience, you might remember some of these matrix/vector rules. Other such rules can be found in references like *The Matrix Cookbook*. I will discuss a more systematic approach to differentiate large functions in a later note.

For now, we can use the two rules we've derived to differentiate the cost function above:

$$\nabla_{\mathbf{w}}[\mathbf{r}^\top\mathbf{r}] = -2X^\top\mathbf{y} + 2X^\top X\mathbf{w}.$$

1. If you don't know what the K-nearest neighbour rule is, an algorithm almost described by its name, I recommend having a quick look.
2. Keen students could look at one answer in this paper:
`http://papers.nips.cc/paper/2566-neighbourhood-components-analysis`

A closed form solution

The partial derivatives are all zero at an optimum weight vector. If there is unique best setting of the weights, we can solve for where that happens:

$$\begin{aligned} -2X^\top \mathbf{y} + 2X^\top X\mathbf{w} &= \mathbf{0} \\ \Rightarrow \quad X^\top X\mathbf{w} &= X^\top \mathbf{y} \\ \Rightarrow \quad \mathbf{w} &= (X^\top X)^{-1} X^\top \mathbf{y}. \end{aligned}$$

If (and only if) there isn't a unique solution for the weights that give the minimum square error, then $(X^\top X)$ is not invertible and the equation isn't defined. The above expression is known as the *normal equation* solution of the least squares problem. You could implement this solution as follows:

```
% Matlab/Octave
ww = (X'*X)\(X'*yy)

# NumPy (or use the Cholesky solvers in scipy):
ww = np.linalg.solve(np.dot(X.T, X), np.dot(X.T, yy))
```

I have attempted to avoid numerical problems by solving the linear system directly rather than inverting the matrix $X^\top X$. However this code is not the most accurate way to solve least squares problems, which can be accessed by calling a dedicated routine (through `\` or `lstsq`, as previously demonstrated).[3]

Iterative methods

There are also generic algorithms that iteratively improve an initial guess of some model parameters using a cost function and its gradient vectors. While linear regression has specialist solvers, we could apply these generic algorithms anyway. If they don't work in simple cases, they're not likely to be useful more generally!

A naive way to use the gradient $\nabla_{\mathbf{w}}[\mathbf{r}^\top \mathbf{r}]$ is the steepest-descent method:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}}[\mathbf{r}^\top \mathbf{r}],$$

which uses a small step size η. The parameters are moved in the direction that makes the biggest immediate change. The rule is applied repeatedly, with the gradient re-evaluated before each update. There are several methods (found in Matlab's optimization toolbox and scipy) that will converge faster. Sometimes *much* faster.

It may help to rewrite the gradient to understand what the steepest descent rule is doing.

$$\begin{aligned} \nabla_{\mathbf{w}}[\mathbf{r}^\top \mathbf{r}] &= 2X^\top (X\mathbf{w}) - 2X^\top \mathbf{y} \\ &= 2X^\top (\mathbf{f} - \mathbf{y}) \\ &= 2\sum_n \mathbf{x}^{(n)} (f^{(n)} - y^{(n)}). \end{aligned}$$

For each example we look at the 'prediction error', $(f - y)$. The weights are pulled most in the direction of inputs that had large prediction errors, to reduce misfit in those directions.

In *Stochastic Gradient Descent* (SGD) we take just one example at a time (perhaps at random, or visit the examples in order). Each example gives a crude one-sample *Monte Carlo* approximation of the gradient sum:

$$\frac{1}{N} \nabla_{\mathbf{w}}[\mathbf{r}^\top \mathbf{r}] \approx 2\mathbf{x}^{(n)} (f^{(n)} - y^{(n)}).$$

3. See *Solving Least Squares Problems*, Lawson and Hanson (1974), Chapter 19, which argues that it's better to use a QR decomposition of the data matrix than losing precision by forming the summary $X^\top X$. Murphy's textbook also champions the QR approach. I must say I'm not totally convinced. For noisy and regularized machine learning problems I've tried, the normal equations approach seems fine, and is $\sim 10\times$ faster on my machine. However, I still frequently use the QR solvers: they're still quite fast, convenient, and I don't have to keep checking whether the normal equations approach will be accurate.

We take a small step in minus this direction. Then pick another example and repeat. Each time we see an example, we move the weights in a direction proportional to just one of the input vectors.

If we have 100,000 datapoints, we perform 100,000 updates in one sweep through the dataset (a 'training epoch'). In the traditional (batch) steepest gradient descent method, we only perform one update after looking at the whole dataset once. In the limit of an infinite stream of data, SGD can fit a model as the data comes in. A traditional batch method never gets started, as it can never compute *the* gradient.

Once we have a working gradient-based optimization procedure[4], we can apply it to problems beyond linear regression. We need to identify a suitable cost function, which no longer needs to be least squares, and then obtain its gradients.

Check your understanding

- In the normal equation solution code, why did I write `(X'*X)\(X'*yy)` rather than `(X'*X)\X'*yy` ? In python: `np.linalg.solve(np.dot(X.T, X), np.dot(X.T, yy))` rather than `np.dot(np.linalg.solve(np.dot(X.T, X), X.T), yy)`
  Do they give the same answer? Is there something else to consider?
- What would happen to the normal equation solution in linear regression problems with more features than datapoints ($D > N$, or as statisticians would say, "$p > n$")?
- In a previous note we saw that we can use a simple linear least squares solver to fit L2 regularized least squares problems. If we applied that trick, would it solve the problem with the normal equations for $D > N$?
- One misguided way to add an "extra" feature would be to copy an existing one, i.e., create an extra column of the training inputs X equal to an existing column. Explain why there would be multiple settings of the weights that have the same least squares training cost. Explain why the normal equation approach becomes ill-defined. What would happen if you ran a gradient-based optimizer to fit the weights?
- You could answer a generalized version of the previous question, where the "extra" feature is an arbitrary linear combination of all the other features. The training inputs X would contain a column that was a linear combination of the other columns.
- Instead of applying the trick we used before to fit L2 regularized models, can you generalize the derivation of the normal equation to explicitly cover this case? (Many textbooks have the answer to this problem.)

Further Reading

Murphy derives the least squares solution in Chapter 7. The description uses a Gaussian model to motivate the least squares cost function throughout.

Barber Chapter 17 starts with linear regression.

Numerical Recipes (Press et al.) will tell you that steepest gradient descent is a bad algorithm, and describes more sophisticated alternatives. See the section on Conjugate Gradient Methods in Multidimensions (section 10.6 of the second edition or 10.8 of the third edition). The books have some nice descriptions, but I would stay clear of the code (better alternatives with free licenses are available). However, they only describe 'batch' methods, that look at an entire dataset before making each update. *Stochastic* (steepest) gradient descent is not such a bad algorithm, especially for machine learning. `https://arxiv.org/abs/1606.04838` is one recent survey.

4. And we might find one slightly more sophisticated than described above.