Overview: fitting probabilistic models

We've now seen that we can achieve a lot with only functions that are linear in their parameters and/or Gaussian distributions. Combined with the training, validation, testing framework and some awareness of statistical variability and error bars (covered in week 2), you have the basic tools to start building machine learning applications.

After this week, we'll look at deeper non-linear models, and then eventually return to Bayesian treatments for non-linear/non-Gaussian probabilistic models. However, there is little point considering these until you can confidently apply baselines based on linear or logistic regression and Gaussian distributions. So around now (and next week) is a good time to do some review.

For example: Do you know when to use which dataset splits, and how to make model choices based on them? Can you choose between reporting a standard error or a standard deviation (neither is always the correct answer)? What are the differences between different methods that can be applied to the same task? Have you made your own notes on the topics covered so far?

For the rest of this week we revisit and extend our understanding of non-Gaussian probabilistic models.

1 Negative log-likelihood cost functions

The Gaussian model for regression let us follow a Bayesian treatment of the model. Before that, we noticed that a non-Bayesian point estimate, maximizing the likelihood of a linear regression model with Gaussian observations, was just least squares. Maximizing the posterior density of the weights (known as MAP estimation) corresponds exactly to minimizing an L2 regularized least squares cost. In these cases, the probabilistic treatment didn't seem to change anything.

However, we also noted in passing that if different observations are known to have different reliabilities (or different noise variances), then maximum likelihood fits a different cost function: a weighted form of least squares. That pattern will repeat itself over the next two notes. We modify a model to match properties of the data that we have, and then derive a new negative log-likelihood cost function that is better suited to the particular application that it's designed for.

We have already studied a non-Gaussian model: logistic regression. This model predicted binary labels, and (for better or worse) the gradients of the negative log-likelihood cost show that we make large corrections to the model when we make confidently-wrong predictions. Our new non-Gaussian models will follow the same recipe as logistic regression:

- Identify the negative log-likelihood cost function
- Attempt to minimize the cost with a standard gradient-based optimizer

The rest of this note considers some details in this recipe, before working through two new examples in the next two notes.

1.1 Reparameterization

It's common for parameters in a model to be constrained. For example, the noise level in probabilistic linear regression or Gaussian processes σ_y is positive. In general, if our optimizer tried to evaluate an illegal setting of parameters, it's possible our code could crash. In our example, our code probably evaluates σ_y^2, and so would always consider positive variances. The real problem is that when the optimizer approaches the constraint $\sigma_y \approx 0$, the cost function and its gradients often become extreme. The optimizer might not converge.

The log likelihood of a regression model (with a limited number of basis functions) as $\sigma_y \to 0$ often tends to $-\infty$. The model becomes absolutely confident in the limit, but is actually making mistakes. Alternatively a flexible regression model might overfit and send the likelihood towards $+\infty$. A GP model with $\sigma_y = 0$ can be well defined, although with a Gaussian kernel, and noisy real-world data, the marginal likelihood is often extreme as well.

To keep away from constraints, while using an unconstrained optimizer, we optimize a transformed parameter $v = \log \sigma_y$. Our optimizer updates a value for v. We then compute the constrained quantity from it, $\sigma_y = \exp(v)$, and then compute the cost function as before. The optimizer needs derivatives of our cost c with respect to the new parameter v. We can obtain that derivative with the chain rule:

$$\frac{\partial c}{\partial v} = \frac{\partial c}{\partial \sigma_y}\frac{\partial \sigma_y}{\partial v} = \frac{\partial c}{\partial \sigma_y}\sigma_y. \tag{1}$$

As the noise level approaches zero, we multiply our original gradient $\frac{\partial c}{\partial \sigma_y}$ by our small noise level σ_y, encouraging smaller updates.

If the optimizer still wants to send $v \to -\infty$, we can add a regularizer to our cost to stop it.

Reparameterizing[1] models to have unconstrained parameters is a generally useful idea, and a sensible default. The positive lengthscales of a Gaussian process ℓ_d, are also usually reparameterized to be unconstrained. A potential problem is that we can no longer fit points on the boundaries of our constrains, such as $\sigma_y = 0$, even if we think those settings could be reasonable.

More generally, as you encounter new situations, it's worth considering whether to transform the ranges of parameters, data, and model outputs. So far in the course we have used $\exp()$ to force a number to be positive, and $\log()$ to make a positive number unconstrained. We used the logistic sigmoid $\sigma()$ to make numbers lie in $(0, 1)$, and in one of the examples coming up, we can use its inverse, the $\mathrm{logit}()$, to make values in $(0, 1)$ become unconstrained.

1.2 Convexity

We have a recipe for writing down a cost function. Identify a probabilistic model of the data, and minimize the negative log probability. But will we be able to minimize this cost function?

If the cost function is *convex* the numerical optimization literature provides methods where we can guarantee finding parameters that minimize the cost as much as possible. A function is convex if a straight line drawn between any two points on the cost function surface never goes below the surface. Algebraically:

$$C(\alpha\mathbf{w} + (1-\alpha)\mathbf{w}') \;\le\; \alpha C(\mathbf{w}) + (1-\alpha)C(\mathbf{w}'), \quad \text{for any } \mathbf{w}, \mathbf{w}', 0 \le \alpha \le 1. \tag{2}$$

If the inequality above is strict ($<$ rather than $\le$), the function is strictly convex and has one unique minimum. Otherwise there could be a connected convex set of parameters, where the cost is equal, but as small as possible.

From the definition, it's easy to show that the sum of any convex functions is convex. So if the loss function for an individual example is convex, then our loss function will be too.

For logistic regression, the loss, $-\log \sigma((2y-1)\mathbf{w}^\top\mathbf{x})$, is a convex function of $\mathbf{w}$. We can guarantee we'll find the maximum likelihood solution to within a small numerical tolerance.

The square loss with the logistic regression function, $(y - \sigma(\mathbf{w}^\top\mathbf{x}))^2$ is not convex. Simply plot the cost for some y and $\mathbf{x}$ while changing one weight to see a counter example. So, we

1. Those who have heard of the "reparameterization trick" in machine learning might wonder if that's what we're talking about. The answer is no. We'll cover the "reparameterization trick" that's used in variational inference when we cover variational inference.

can't provide the same guarantees about being able to find the minimum of this loss. We can still try to reduce this loss function using gradient methods, and in practice we're likely to find parameters with reasonable loss, that generalize usefully to new examples. We just can't guarantee that a better optimizer wouldn't have found better parameters.

Most machine learning methods don't correspond to convex optimization problems. While convexity is a nice property of logistic regression, as soon as we try to optimize any of our surrounding choices (such as the locations of basis functions) the overall optimization problem is usually not convex. However, it's still of some use to notice when a baseline, or stage of a procedure, has a well-defined, reproducible solution.

1.3 Why log probability?

You might have been wondering why we use the *log* probability to score our models, rather than evaluating and differentiating the probability itself.

One reason is that the log probability of a training set is a sum (rather than product) over examples. That usually makes the maths more convenient. Also we can approximate the sum with a subset to perform stochastic gradient descent.

We also use log probabilities to avoid numerical underflow. Most numerical software (including Matlab and NumPy) works with IEEE floating point. Even with "double" precision, the probability of a sequence of 1,000 coin tosses, 0.5^{1000}, underflows to zero. In contrast, we can compute $1000 \log 0.5$ with no difficulty.

Finally, the log probability is more appropriate for numerical optimizers. As an example, if we fit the mean of a Gaussian's negative log PDF by gradient methods, we have a quadratic cost function, which is convex and the ideal cost function for most gradient-based optimization methods. The PDF itself is not convex and has values in a narrow numerical range near zero in the tails. The log-likelihoods for non-Gaussian models, such as logistic regression, are also sometimes approximately quadratic near the mode when there are many datapoints.

2 Further reading

Most students should move on to the examples in the next two notes now. This section has (optional) links to yet more examples, which might be useful if you return to these notes at a later time.

The readings for logistic regression already mentioned that by reading about "generalized linear models" you could see more examples of probabilistic models for the outputs, beyond those we've seen.

Those who already know something about neural networks (or those returning to these notes later) might also think about adding probabilistic models for different types of output to neural networks:

- Multi-modal real-valued distributions with "Mixture Density Networks" (Bishop, 1994).
- Count data, such as in Amazon sales forecasting, using a negative-binomial model on the output layer. "DeepAR: Probabilistic forecasting with autoregressive recurrent networks" (Salinas et al., 2020).