Machine Learning and Pattern Recognition Tutorial 5

Instructor: Iain Murray

Motivation for this tutorial: Some of the questions in this tutorial ask you to perform some (short!) numerical experiments. Hopefully learning to do these things quickly will be a useful skill. Putting standard error bars on experimental results is a minimal piece of statistical care which might be useful in many of your projects.

1. **Model comparison computation:** It's common to compute log-likelihoods and log-marginal-likelihoods to avoid numerical underflow. (Quick example: notice that the probability of 2000 coin tosses, 2^{-2000}, underflows to zero in Matlab, or any package using IEEE floating point.)

   Assuming there are only two possible models, M_1 and M_2, we define:

$$\begin{aligned} a_1 &= \log P(D \mid M_1) + \log P(M_1) \\ a_2 &= \log P(D \mid M_2) + \log P(M_2). \end{aligned}$$

   These 'activations' are the log-posteriors of each model, up to a constant. Show that we can get the posterior probability of model M_1 neatly with the logistic function:

$$P(M_1 \mid D) = \sigma(a_1 - a_2) = \frac{1}{1 + \exp(-(a_1 - a_2))}.$$

   Given K models, with $a_k = \log[P(D \mid M_k)\, P(M_k)]$, show:

$$\log P(M_k \mid D) = a_k - \log \sum_k \exp a_k.$$

   The $\log \sum \exp$ function occurs frequently in the maths for probabilistic models (not just model comparison). Show that:

$$\log \sum_k \exp a_k = \max_k a_k + \log \sum_k \exp\left(a_k - \max_{k'} a_{k'}\right).$$

   Explain why the expression is often implemented this way. (Hint: consider what happens when all the a_k's are less than -1000).

Answer:

Bayes' rule: $P(M_1 \mid D) = e^{a_1}/Z$ and $P(M_2 \mid D) = e^{a_2}/Z$, where $Z = P(D)$.

Only two possible models $\Rightarrow P(M_1 \mid D) + P(M_2 \mid D) = 1, \quad Z = e^{a_1} + e^{a_2}$.

Substituting Z,

$$P(M_1 \mid D) \;=\; \frac{e^{a_1}}{e^{a_1} + e^{a_2}} \;=\; \frac{1}{1 + e^{a_2 - a_1}} \;=\; \sigma(a_1 - a_2).$$

With K models $P(D) = \sum_k P(D, M_k) = \sum_k e^{a_k}$, so Bayes rule gives:

$$\begin{aligned} P(M_k \mid D) &= \frac{e^{a_k}}{\sum_{k'} e^{a_{k'}}}, \qquad \text{(a 'softmax')} \\ \log P(M_k \mid D) &= a_k - \log \sum_{k'} e^{a_{k'}}. \end{aligned}$$

This expression involves e^{a_k}'s. If all the a_k's are very negative, we may take the log of zero and obtain `-Inf`. In the alternative expression, some of the terms may still underflow. However the largest term in the sum is now $e^0 = 1$. Any terms that underflow are less than `realmin` $\approx 2\times 10^{-308}$ and so would have no practical effect when added on to a sum of value ≥ 1.

---

2. **Simple Monte Carlo:** Let x be distributed according to a distribution with mean μ and variance σ^2. To construct an example, sample a vector of a hundred samples from Uniform[0,1], using `rand` in Octave or Matlab. Given access to only these samples, find estimates $(\hat{\mu}, \hat{\sigma}^2)$ of the mean and variance of x.

   Show that the Monte Carlo estimate of the mean

$$\mathbb{E}[x] \approx \frac{1}{S}\sum_{s=1}^{S} x^{(s)}$$

has variance σ^2/S (in general, not just for a uniform).

It's common to report an estimate of a mean using a 'standard error', the square root of an estimate of this variance:

$$\mathbb{E}[x] = \hat{\mu} \pm \frac{\hat{\sigma}}{\sqrt{S}}$$

Write down your estimate of the mean in this form. Is your answer within two standard errors of the true answer? It should fall in that range around 95% of the time. In a class of 120, some of you will probably get a 'wrong' estimate! Try running your code again and see if the true answer is *usually* within 1 or 2 standard errors.

Also report an estimate of $\mathbb{E}[x^2]$ with a standard error.

Answer:

```
S = 100;
xx = rand(S, 1);
mu_est = mean(xx);
var_est = var(xx); % var(xx,1) if you want to normalize by N instead of N-1
```

The variance of independent variables adds, and each $x^{(s)}$ is identically distributed with variance σ^2. So the var$[\sum_s x^{(s)}] = S\sigma^2$. Scaling a variable by a constant, scales the variance by the constant squared, so var$[(1/S)\sum_s x^{(s)}] = (1/S^2)(S\sigma^2) = \sigma^2/S$.

```
std_err = sqrt(var_est/S);
fprintf('%1.3f +/- %1.3f\n', mu_est, std_err);
0.529 +/- 0.028
```

The answer depends on random draws, so will be different each time. It's arguable whether to use one or two significant figures for the error bar, but don't use more.

To find the estimate of any function of x, we just repeat the procedure with the function evaluations in place of the original values:

```
mu_est = mean(xx.^2);
var_est = var(xx.^2);
std_err = sqrt(var_est/S);
fprintf('%1.3f +/- %1.3f\n', mu_est, std_err);
0.358 +/- 0.028
```

Compatible with the true answer of 1/3.

If one estimates sample means a lot, one wraps it into a standard routine:

```
http://homepages.inf.ed.ac.uk/imurray2/code/imurray-matlab/errorbar_str.m
errorbar_str(xx.^2);
0.358 +/- 0.028
```

---

3. **Rejection sampling:** use a simple Monte Carlo procedure to estimate the mean of the truncated normal:

$$p(x) \propto \begin{cases} N(x; 0, 1) & x > 1 \\ 0 & \text{otherwise} \end{cases}$$

What would happen if you applied your procedure to a truncation at $x > 6$?

Answer:

The standard untruncated normal, scaled to the same height as the truncated normal, is an upper bound on the truncated normal. Therefore we can sample from $q(x) = N(x; 0, 1)$, then reject samples whenever $x \leq 1$.

In this case we can omit the step in the generic algorithm where we draw a random height under the curve. When $x \leq 1$ we would always draw a sample above the target density curve. When $x > 1$, the curves touch and the point would always be below the target density curve.

```
xx = randn(1e6,1); % draw lots of samples from q
xx = xx(xx>1); % thin, we now have (fewer!) samples from p
errorbar_str(xx); % routine to estimate mean and std err from q2
1.5259 +/- 0.0011
```

(Actual answer to 5sf, 1.5251).

The probability of getting $x > 6$ is around 10^{-9} so it takes roughly a billion proposals to get a single sample. The method would work eventually, if prepared to draw at least several billion samples. (The accompanying `q3a.m` explores one possible fix.)

---

4. **Importance sampling:** How could we use importance sampling to estimate the mean of a truncated distribution with $x > 6$?

Answer:

If we use importance sampling with the same $q(x) = N(x; 0, 1)$ proposals as before, we'll have the same problem. Only about a billionth of samples will get non-zero importance weight.

We could sample from a distribution that puts more weight on the tail area.

The question doesn't actually ask us to code up an answer, but I'll do it anyway. I sampled points from a standard normal with mean 6, and reflected all the samples to be greater than 6:

$$q(x) = \begin{cases} 2N(x; 6, 1) & x > 6 \\ 0 & \text{otherwise} \end{cases}$$

Some thought will reveal better things to do. But I thought this might work.

```
tt = 6; S = 1e6;
xx = abs(randn(S, 1)) + tt;
q_x = 2*exp(-(xx-tt).^2/2)/sqrt(2*pi);
pstar_x = exp(-xx.^2/2)/sqrt(2*pi);
wstar = (pstar_x./q_x); % unnormalized weights. pstar_x *not* normalized!
ww = wstar/sum(wstar);
est = sum(ww.*xx)
6.158
```

I can't add a standard error bar to the estimate in the same way as before, because the normalization of the weights couples the terms in the sum. Non-examinable: constructing an estimate of the variance is reviewed in section 3 of:
`http://www.cs.toronto.edu/~radford/ais.abstract.html`

If I were responsible, I would examine the weights and check that I don't need to make q broader, or with heavier tails. Given the answer seems so close to 6, and q goes out a lot further, I doubt there are problems.

---

5. **Markov chain Monte Carlo:** we could apply the Metropolis algorithm (using the code from the slides and copied below, or otherwise) to estimate the mean of the truncated normal with $x > 6$.

   What would be the difficulty with reporting error bars on our estimate?

Answer:

We can't compute a standard error simply from the samples as in question 2, because the samples aren't independent, so the variance of the estimator is not σ^2/S.

That's the answer to the question, but keen students will wonder what to do about it, and may wish to see some code anyway…

The thing that requires least thought is probably to do K independent runs of the algorithm, to get K independent estimators. Anything beyond that is non-examinable. Radford Neal's review `http://www.cs.toronto.edu/~radford/review.abstract.html` discussed some other options. Packages like R-CODA can do time-series analysis of your samples to try to estimate how many "effective samples" there are.

```
xx = dumb_metropolis(6.5, @(x) (-0.5*x*x)+log(double(x>6)), 1e6, 1.0);
mean(xx)
ans =
6.1588
```

Without doing any careful analysis, we do get some confirmation that our previous answer seems reasonable.

The acceptance rate of the chain was a little low, but not disastrously so:

```
mean(xx(2:end) =xx(1:end-1))
ans =
0.1236
```

I would consider reducing the step-size from 1.0 for another run. (A theoretically desirable acceptance rate is 0.234: `http://projecteuclid.org/euclid.aoap/1034625254`.)

`q5.m`, distributed with the answers, does multiple runs with a step-size of 0.5 to get an answer with error-bars.

```
function samples = dumb_metropolis(init, log_ptilde, iters, sigma)

D = numel(init);
samples = zeros(D, iters);

state = init;
Lp_state = log_ptilde(state);
for ss = 1:iters
    % Propose
    prop = state + sigma*randn(size(state));
    Lp_prop = log_ptilde(prop);
    if log(rand) < (Lp_prop - Lp_state)
        % Accept
        state = prop;
        Lp_state = Lp_prop;
    end
    samples(:, ss) = state(:);
end
```