2 Robust modelling

In this part we'll consider building a model where our training data is unreliable. The idea considered in this part could be applied widely, including to noisy images. However, we'll now shift to a simpler setting so you can work on the two parts of the assignment independently.

One of the aims of this part is getting you to combine existing tools such as optimizers and samplers with model likelihoods. There are many packages that already implement standard machine learning methods, such as logistic regression. However, if you can put the pieces together yourself, you can introduce modifications and try out new variants. For your new methods to be trustworthy, you will also need to be able to check your code, and diagnose how well things are working. Something you may find is that not every extension of a model necessarily makes it work better.

Logistic regression: Our baseline model will be a logistic regression classifier, which given a D-dimensional real-valued feature vector $\mathbf{x}$, predicts a label $y \in \{-1, +1\}$. The *weights* $\mathbf{w}$ set the probability distribution over labels:

$$P(y \,|\, \mathbf{x}, \mathbf{w}) = \sigma(y\mathbf{w}^\top \mathbf{x}) = \frac{1}{1 + \exp(-y\mathbf{w}^\top \mathbf{x})}. \tag{1}$$

The Matlab code in `lr_loglike.m` implements the log-likelihood of the weights given N training pairs $\{\mathbf{x}^{(n)}, y^{(n)}\}$, and its gradients:

$$\mathcal{L}(\mathbf{w}) = \sum_{n=1}^{N} \log P(y^{(n)} \,|\, \mathbf{x}^{(n)}, \mathbf{w}), \qquad \nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = \sum_{n=1}^{N} (1 - \sigma(y^{(n)}\mathbf{w}^\top \mathbf{x}^{(n)}))y^{(n)}\mathbf{x}^{(n)}. \tag{2}$$

In lectures and Tutorial 3 you saw logistic regression derived for labels $y \in \{0, 1\}$. You may wish to prove to yourself that the expressions here for $y \in \{-1, +1\}$ are equivalent to the alternative versions you have seen. However, showing the correspondence is not part of the assignment.

Data: The data provided in `text_data.mat` comes from a text classification problem. The binary feature vectors $\{\mathbf{x}^{(n)}\}$, with $x_d^{(n)} \in \{0, 1\}$ indicates absence or presence of word d in document n. Only $D = 100$ common words are checked for each document. The labels, $y^{(n)} \in \{-1, +1\}$ indicate whether each document belongs to the positive or negative class. The labels on the test set were carefully checked, however the training set is known to contain errors.

Reporting your code: Much of this part is about you putting together small parts of code. You're reminded to include the code snippets you write in your answers to each part.

1. **Fitting the baseline model (20 marks)**

   (a) **Bias feature:** The basic model, Equation (1), always has its decision boundary through the origin: $P(y=1 \,|\, \mathbf{x}=0) = 1/2$ for all settings of the weights. A more flexible classifier includes a 'bias' weight: $P(y \,|\, \mathbf{x}, \mathbf{w}) = \sigma(y(\mathbf{w}^\top \mathbf{x} + b))$. Rather than adding extra inputs and outputs to the likelihood function, we can augment the data.

   Load the data in `text_data.mat` and expand both the training and test inputs to be $(D+1)$-dimensional, where $x_{D+1} = 1$ for all examples. The final element of the weight vector will then be the bias weight: $b = w_{D+1}$.

   Answer:

   ```
   load text_data.mat
   x_train = [x_train, ones(size(x_train,1), 1)];
   x_test = [x_test, ones(size(x_test,1), 1)];
   ```

(b) **Maximizing the likelihood:** Most optimization routines *minimize* a cost function. We have provided a routine `minimize.m` that often works well. Using or adapting the provided `lr_loglike.m` routine, create a *negative*-log-likelihood function and minimize it given the training data `x_train` and `y_train` in `text_data.mat`.

Answer:

```
function [nLp, dnLp_dw] = lr_nll(ww, xx, yy)
% LR_NLL negative log-likelihood and gradients for logistic regression
[Lp, dLp_dw] = lr_loglike(ww, xx, yy);
nLp = -Lp;
dnLp_dw = -dLp_dw;
```

Then

```
% Augment train and test inputs as above then...
ww = minimize(zeros(size(x_train,2),1), @lr_nll, 100, x_train, y_train);
```

It's at this point that I (Iain) need to confess that this question wasn't perfectly constructed. There's some useful lessons to learn related to what I did wrong, so I'll now digress into a discussion of what happened.

In the code snippet above, I ask `minimize.m` to run for at most 100 linesearches. For this scale of problem, that should normally be sufficient, unless the model can severely overfit, and then the optimizer can spend a long time driving the weights to large values. I *was* worried about overfitting, because (to keep the question simple) I didn't ask you to do any regularization at this stage. However, when I first wrote the answers to this part, `minimize.m` didn't find a very different answer if given many more linesearches. At the last minute, I changed the dataset to a simpler one to make the later parts of the assignment easier. I forgot to go back and check whether 100 linesearches were still sufficient for this part. My code made sensible predictions on the new data, so I assumed it was ok.

As some of you found, running for many more than 100 linesearches *did* lead to severe overfitting, so that the test mean log probability was infinitely bad. That can happen in real life, but as you assumed I'd construct a reasonable question, it led to confusion. I'm sorry.

Those who, like me, didn't fit for a large number of linesearches accidentally introduced a form of regularization: *early stopping*. Running the optimizer for less time than full convergence on the training set saves computation, and can actually make the the test set performance better. However, if I'd intended to do early stopping, it would have been better to monitor performance on a validation set to decide when to stop, rather than just guessing.

Moral: regularization is important. Normally some form of regularization should always be considered if fitting parameters. (The other lesson is that I will be more careful when making last-minute changes to questions! Throughout the assignment we accepted answers based on either running `minimize.m` for a short or long time.)

Given the fitted weights, find the probability that $y=+1$ for each of the test inputs `x_test`. Report the accuracy, that is, the fraction of labels in `y_test` that are the most probable label under the predictions. Also report the mean log probability that the predictions assign to the test labels.

Both the accuracy and the mean log probability are means (averages) over the test set. These means estimate the mean performance that would be obtained given an infinite test set. Given we have a finite test set, these estimates will be noisy. Good answers will report 'standard errors' to give an indication of how different the performance on future test cases might be. (Tutorial 5 includes computing a standard error.)

Also report the performance of predictions on the training set, and compare to the test set performance.

Is the mean log probability better than a baseline that predicts $P(y\,|\,\mathbf{x})=0.5$ for every test case?

Answer:

```
function report_score(y_test, pred)
% Report quality of p(y=1) predictions in pred
assert(numel(y_test) == numel(pred));
y_test = (y_test(:) == 1);
acc = errorbar_str(y_test == (pred(:) > 0.5));
Lp = [log(pred(y_test)); log(1 - pred(~y_test))];
mlp = errorbar_str(Lp);
fprintf('Accuracy: %s    MLP: %s\n', acc, mlp);
```

Using `http://homepages.inf.ed.ac.uk/imurray2/code/imurray-matlab/errorbar_str.m`

```
pred = 1./(1 + exp(-(x_train*ww(:))));
report_score(y_train, pred);

% Outputs (depends on how long we run minimize):
% Accuracy: 0.8334 +/- 0.0046    MLP: -0.4399 +/- 0.0081

pred = 1./(1 + exp(-(x_test*ww(:))));
report_score(y_test, pred);

% Outputs (depends on how long we run minimize):
% Accuracy: 0.9071 +/- 0.0072    MLP: -0.2889 +/- 0.0071
```

We get significantly better test accuracy than 0.5 from uniform guessing. A uniform predictor has an MLP of $\log(0.5) = -0.693$, which the above result (with early stopping) beats. However, if we optimize for longer, the MLP can numerically evaluate to $-\infty$. The accuracy remains high.

Some of you evaluated both $p(y=+1\,|\,\mathbf{x},\mathbf{w}) = \sigma(\mathbf{x}^\top\mathbf{w})$ and $p(y=-1\,|\,\mathbf{x},\mathbf{w}) = \sigma(-\mathbf{x}^\top\mathbf{w})$, to avoid numerical problems with evaluating $1-p(y=+1\,|\,\mathbf{x},\mathbf{w})$ when $p(y=+1\,|\,\mathbf{x},\mathbf{w})$ is close to one. That was clever. Such tricks are usually not necessary for regularized models where the predictive probabilities don't get so close to 1.

In the result above, the training set performance is significantly worse, which is unusual. Normally the best case is that training and test performance are similar. As we've seen what happens at training locations, we might not do quite as well at new input locations where we haven't seen labels. If we overfit, we'll definitely do worse on the test set. For this dataset the training labels are more noisy, and so harder to predict. That can happen if we only have resources to check a small test set carefully by hand.

(c) **Limited training data:** Fit the model with only the first $N = 100$ training cases. What is the average log-probability of the test labels reported by your code given the optimized weights? Interpret this result.

Answer:

```
N = 100;
ww2 = minimize(zeros(size(x_train,2),1), @lr_nll, 100, x_train(1:N,:), y_train(1:N));
pred2 = 1./(1 + exp(-(x_test*ww2(:))));
report_score(y_test, pred2);

% Outputs:
% Accuracy: 0.775 +/- 0.010    MLP: -Inf

max(ww2)
% Outputs: 57.5833
```

The large weights correspond to a confident classifier, overfitting the limited training data. The decision boundary will not only get test cases wrong, but assign probability that numerically underflows to zero on some of the labels. That results in an apparent mean test log probability of $-\infty$. We should have regularized the weights.

Many of you already saw overfitting on the full dataset, in which caes there wasn't much further to say about this part.

2. **Label noise model (15 marks)**

   (a) **Modifying the likelihood:** We know for this dataset that some of the training data has been mislabelled. To model these errors we assume that a binary noise variable s_n was drawn for each example:

$$P(s_n = 1) = \epsilon, \quad P(s_n = 0) = 1 - \epsilon.$$

When $s_n = 1$ the label is chosen with a uniform random choice, ignoring the $\mathbf{x}$ features. When $s_n = 0$ we model the label as coming from the original logistic regression model (1). Show that the probability of labels under this noisy labelling process is:

$$P(y \,|\, \mathbf{x}, \mathbf{w}, \epsilon) = (1 - \epsilon)\, \sigma(y\mathbf{w}^\top \mathbf{x}) + \epsilon/2, \qquad y \in \{-1, +1\}. \tag{3}$$

Create a function to return the log-likelihood of this model given training data, and the gradients with respect to both $\mathbf{w}$ and ϵ.

Answer:
The noisy model likelihood results from marginalizing out the noise variables:

$$P(y \,|\, \mathbf{x}, \mathbf{w}, \epsilon) = \sum_{s\in\{0,1\}} P(y, s \,|\, \mathbf{x}, \mathbf{w}, \epsilon) = \sum_{s\in\{0,1\}} P(y \,|\, s, \mathbf{x}, \mathbf{w})\, P(s \,|\, \epsilon) = P(y \,|\, s, \mathbf{x}, \mathbf{w})(1 - \epsilon) + \frac{1}{2}\epsilon.$$

Let $P_n = P(y^{(n)} \,|\, \mathbf{x}^{(n)}, \mathbf{w}, \epsilon)$ and $\sigma_n = \sigma(y^{(n)}\mathbf{w}^\top \mathbf{x}^{(n)})$

$$\nabla_\mathbf{w} \log P_n = \frac{1-\epsilon}{P_n}\sigma_n(1 - \sigma_n)y^{(n)}\mathbf{x}^{(n)}$$

$$\frac{\partial \log P_n}{\partial \epsilon} = (1/2 - \sigma_n)/P_n$$

These derivates are summed over training cases to get the derivatives of $\mathcal{L}$. You may have reached different-looking but equivalent expressions.

```
function [Lp, dLp_dw, dLp_de] = rlr_loglike(ww, xx, yy, epsilon)
%RLR_LOGLIKE robust logistic regression log-likelihood and gradients
%
%     [Lp, dLp_dw, dLp_de] = lr_loglike(ww, xx, yy, epsilon);
%
% Inputs:
%          ww Dx1 logistic regression weights
%          xx NxD training data, N feature vectors of length D
%          yy Nx1 labels in {+1,-1} or {1,0}
%     epsilon 1x1 probability of label noise
%
% Outputs:
%          Lp 1x1 log-probability of data, the log-likelihood of ww
%      dLp_dw Dx1 gradients: partial derivatives of Lp wrt ww
%      dLp_de 1x1 gradient: partial derivative of Lp wrt epsilon

% Iain Murray, October 2014, August 2015

% Ensure labels are in {+1,-1}:
yy = (yy==1)*2 - 1;

sigmas = 1./(1 + exp(-yy.*(xx*ww))); % Nx1
pp = (1-epsilon)*sigmas + 0.5*epsilon;
Lp = sum(log(pp));

if nargout > 1
    dLp_dw = xx'*((1-epsilon)*yy.*sigmas.*(1-sigmas)./pp);
end

if nargout > 2
    dLp_de = sum((0.5 - sigmas)./pp);
end
```

Check your gradients are correct with a finite difference approximation such as:

$$\frac{\partial f(z)}{\partial z} \approx \frac{f(z+h) - f(z-h)}{2h}, \quad \text{with error } O(h^2). \tag{4}$$

Make it clear what your test case was, including the h you used, and give an indication of the largest difference observed between your gradients and their numerical approximation. Check the derivatives with respect to the weights $\mathbf{w}$ and noise parameter ϵ.

You may use the provided `checkgrad.m` routine if you find it helpful.

Answer:

```
function rlr_loglike_check()

randn('state', 0);
rand('state', 0);
D = 3;
N = 5;
xx = randn(N, D);
yy = (rand(N, 1) < 0.5)*2 - 1;
ww = 0.01*randn(D,1);
epsilon = rand();


fprintf('Check close agreement between gradients and finite differences:\n');
%hh = 1e-5; % get numerical problems if go smaller
hh = 1e-9i; % Complex step version, allows smaller steps, so more accurate.
%err = checkgrad('rlr_loglike', ww, hh, xx, yy, epsilon) % Only checks dL/dw,
err = checkgrad(@helper, [ww; epsilon], hh, xx, yy) % Also check dL/d_epsilon

function [ff, df] = helper(ww_e, xx, yy)
ww = ww_e(1:end-1);
epsilon = ww_e(end);
[ff, dw, de] = rlr_loglike(ww, xx, yy, epsilon);
df = [dw; de];

% Outputs:
% Check close agreement between gradients and finite differences:
%     0.0093    0.0093
%     0.0220    0.0220
%     0.4542    0.4542
%    -0.0022   -0.0022
% err =
%    1.1936e-16
```

Using standard finite differences as in the question is fine, and for $h = 10^{-5}$ gives reasonable accuracy. The code above comments out that option and uses a complex perturbation instead, which verifies that the relative error in the derivatives is close to machine precision. The test case is small, but picks random values for all the inputs, rather than special inputs such as zero or one. Small weights and a small test case help avoid numerical problems.

(b) **Fitting a constrained parameter:** The new parameter is a probability, constrained to be between zero and one: $\epsilon \in [0, 1]$. We can write this parameter as the result of taking the logistic sigmoid of an unconstrained parameter a:

$$\epsilon = \sigma(a) = \frac{1}{1 + \exp(-a)}. \tag{5}$$

Create a function that evaluates the negative log-likelihood of the new model and evaluates the derivatives with respect to $\mathbf{w}$ and a. Hence fit both $\mathbf{w}$ and a. You are advised to wrap the function from the previous part, rather than starting from scratch. Report the fitted noise level $\epsilon = \sigma(a)$.

Answer:
We wrap the function from the previous part, negating the function and gradients, and transforming the gradients using

$$\frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial \epsilon}\frac{\partial \epsilon}{\partial a} = \frac{\partial \mathcal{L}}{\partial \epsilon}\epsilon(1-\epsilon).$$

```
function [nLp, dnLp_dwa] = rlr_nll(w_a, xx, yy)
% RLR_NLL negative log-likelihood and gradients for robust logistic regression
% w_a is [ww; aa], weights and aa = logit(epsilon)
ww = w_a(1:end-1);
aa = w_a(end);
epsilon = 1/(1+exp(-aa));
[Lp, dLp_dw, dLp_de] = rlr_loglike(ww, xx, yy, epsilon);
nLp = -Lp;
dLp_da = dLp_de*epsilon*(1-epsilon);
dnLp_dwa = -[dLp_dw; dLp_da];
```

It's always wise to check the code again before putting into an optimizer:

```
randn('state', 0);
rand('state', 0);
D = 3;
N = 5;
xx = randn(N, D);
yy = (rand(N, 1) < 0.5)*2 - 1;
w_a = 0.01*randn(D+1,1);

fprintf('Check close agreement between gradients and finite differences:\n');
hh = 1e-9i;
err = checkgrad(@rlr_nll, w_a, hh, xx, yy)

% Outputs:
% Check close agreement between gradients and finite differences:
%    -0.0204   -0.0204
%    -0.0453   -0.0453
%    -0.9545   -0.9545
%     0.0005    0.0005
% err =
%    1.9562e-16
```

Then we optimize:

```
% Remember train and test inputs should have been augmented as before, then:
w_a = minimize(zeros(size(x_train,2)+1,1), @rlr_nll, 100, x_train, y_train);
ww = w_a(1:end-1);
epsilon = 1/(1 + exp(-w_a(end)))

% Outputs (depends slightly on number of line searches and initialization):
% epsilon =
%     0.2028
```

The fitted model says that ∼20% of the training labels were wrong!

Given that the test labels were checked more carefully, predict them using the newly fitted weights but using the original model (1). Report and interpret the new test accuracy and mean log probability.

Answer:
We predict as before but with the newly fitted weights:

```
pred = 1./(1 + exp(-(x_test*ww(:))));
report_score(y_test, pred);

% Outputs:
% Accuracy: 0.9151 +/- 0.0069    MLP: -0.209 +/- 0.018
%
% Optimizing for longer makes MLP = -Inf, but accuracy is similar.
```

The accuracy is similar to before. The mean log probability is significantly better than before: we've learned more confident weights, apparently without overfitting. Although as already discussed, that depended on not running minimize to convergence. By following what the question actually asked for, you may well have found overfitting. It also relied on believing that the test labels wouldn't be noisy.

---

3. **Hierarchical model and MCMC (15 marks)**

A hierarchical model says that the noise level ϵ is unknown with a uniform prior,

$$P(\epsilon) = \text{Uniform}[\epsilon;\, 0, 1]$$

and that the weights are Gaussian distributed, but with unknown variance:

$$P(\log\lambda) = \text{Uniform}[\log\lambda;\, l, u], \qquad \text{where } \lambda > 0,$$

$$P(\mathbf{w}) = \mathcal{N}\big(\mathbf{w}; 0, \tfrac{1}{2\lambda}I\big) = \left(\tfrac{\lambda}{\pi}\right)^{D/2} \exp(-\lambda \mathbf{w}^\top\mathbf{w}),$$

with $P(y \,|\, \mathbf{x}, \mathbf{w}, \epsilon)$ as in Equation (3). While not strictly well defined, it's common to take the 'improper' limit, $l \to -\infty$ and $u \to \infty$. This limit prefers no single value of $\log\lambda$ over another, and does not restrict its range. (Almost all of the prior mass is on extreme values, which may or may not matter in practice.)

The log-posterior of this model, up to a constant, is the log-likelihood $\mathcal{L}(\mathbf{w}, \epsilon) = \sum_n \log P(y^{(n)} \,|\, \mathbf{x}^{(n)}, \mathbf{w}, \epsilon)$ using (3), plus a term for the log prior:

$$\log P(\epsilon, \mathbf{w}, \log\lambda \,|\, \text{data}) = \mathcal{L}(\mathbf{w}, \epsilon) - \lambda\mathbf{w}^\top\mathbf{w} + \tfrac{D}{2}\log\lambda + \text{const.} \qquad (6)$$

(a) What is the largest log-likelihood that any model can have given a training set of N binary outcomes? That is, what would the log-likelihood be if a model were somehow able to predict every training label correctly?

What is the log-likelihood of the model with zero weights, $\mathbf{w} = \mathbf{0}$?

Hence show that the log-posterior above is globally maximized by setting the weights to zero and making λ infinite.

Answer:
The highest likelihood a model can have is when it predicts the training data with probability 1. The log-likelihood is then zero.

The model we're using with zero weights makes uniform predictions, so the log-likelihood is $\log 0.5^N = N\log 0.5$.

If we set the weights to zero, the first two terms become $N\log 0.5$, which is finite, and don't depend on λ. We can then make the posterior as large as we like by taking the limit $\lambda \to \infty$. That's the only way to make the posterior density infinite. If the weights are non-zero then we can't take $\lambda \to \infty$. Moreover, the two terms of (6) are always finite, they're bounded above at zero, because that's the maximum value of the first term for any model, and the second term is negative (or zero) because $\lambda \geq 0$.

(b) Given the previous part, it does not make sense to optimize λ on the training set. (We could restrict the prior range $\log\lambda \in [l, u]$, but that would just shift the problem to setting l and u.) We could set λ by cross-validation — holding out a validation set from the training set while fitting the weights, and testing λ on this set. Instead, in this part we will sample plausible values with Markov chain Monte Carlo (MCMC).

Use the provided `slice_sample.m` MCMC routine to approximately sample from the hierarchical posterior in (6) for 1,000 iterations. Report any choices that you needed to make.

Plot a scatter plot of $\log \lambda$ against ϵ. How reasonable is the value of the noise-level ϵ that you fitted previously? Are our posterior beliefs about $\log \lambda$ and ϵ independent? Why?

Hint: in your code, put $\mathbf{w}$, ϵ and $\log \lambda$ into a single vector, and write a function to evaluate (6), the log posterior (up to a constant) of this vector. Give the slice sampling routine a function handle to this routine. The log-posterior for $\epsilon < 0$ or $\epsilon > 1$ should be $-\infty$ because those settings have zero prior density.

I advise you to turn on stepping-out in the slice sampler. If the slice sampler seems to hang with this feature turned on, then there is probably a bug in your log posterior function.

Answer:
First create the log posterior function:

```
function Lpost = rlr_logpost(w_e_l, xx, yy)
% RLR_LOGPOST log-posterior (up to a constant) for robust logistic regression
% w_e_l is [ww; epsilon; log_lambda], weights, noise parameter, and log-regularizer
% parameters put into one vector for use with slice_sample.m

% Unpack parameters:
w_e_l = w_e_l(:);
ww = w_e_l(1:end-2);
epsilon = w_e_l(end-1);
log_lambda = w_e_l(end);
lambda = exp(log_lambda);

if (epsilon < 0) || (epsilon > 1)
    Lpost = -Inf;
    return
end

D = numel(ww);
Llike = rlr_loglike(ww, xx, yy, epsilon);
Lpost = Llike - lambda*(ww'*ww) + 0.5*D*log_lambda;
```

Then drive it in the sampler. I set `step_out` to true so that I didn't have to worry too much about setting widths to 1 without checking if that value is big enough, or initializing the parameters at good settings. If computer time were an issue I might tune these choices further.

```
% Set up initial state
D = size(x_train, 2);
ww = zeros(D, 1);
epsilon = 0.5;
log_lambda = 0;
w_e_l = [ww; epsilon; log_lambda];

% Slice sample:
S = 1000;
burn = 0; % So we can look at things
widths = 1;
step_out = true;
samples = slice_sample(S, burn, @rlr_logpost, w_e_l, ...
                       widths, step_out, x_train, y_train);

save('samples.mat', 'samples');
```

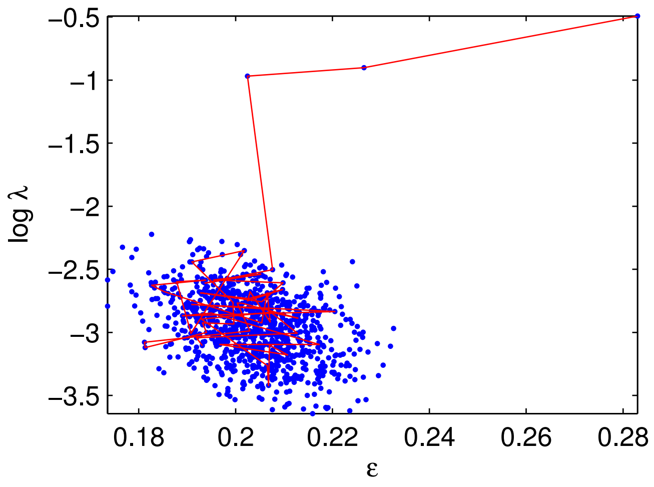
As the sampling takes some time, I save the output for future use. The above implementataion is inefficient. The slice sampler only changes one parameter at a time, so most of the likelihood computation could be reused from the previous function call. The code should be $\sim 100\times$ faster!

We then make a plot:

```
% Actually make the plot:
clf; hold on;
epsilons = samples(end-1, :);
log_lambdas = samples(end, :);
% Scatter plot of all samples:
```

```
plot(epsilons, log_lambdas, '.');
% Overlay path of first 50 iterations:
plot(epsilons(1:50), log_lambdas(1:50), 'r-');
xlabel('\epsilon');
ylabel('log \lambda');

% Export it to a reasonable pdf:
box on
axis tight
set(gcf, 'PaperPosition', [1 1 4 3]*3.5);
set(gca, 'OuterPosition', [0 0 1 1]);
set(gca, 'Position', [0.4 0.4 0.5 0.5]);
print(gcf, 'scatter.eps', '-depsc');
system('epstopdf scatter.eps');
```



It appears burn-in to a fairly simple distribution occurs rapidly.

There is a slight negative correlation between the regularization parameter $\log \lambda$ and the noise level ϵ. The model can explain away labels on the wrong side of the decision boundary by increasing the noise level. When that happens, the weights can then increase in size, which will typically lead to a decrease in $\log \lambda$.

The model is fairly confident that about 17–23% of the training labels are noisy. The fitted value of $\epsilon \approx 0.2$ is compatible with this range. Although previously we had no idea of how certain this estimate was. Bayesian inference is often useful if wanting to interpret what we can learn from the data.

(c) Explain how to use the samples of **w** to predict the test labels. Compare the predictions from sampling to those from the previously fitted model.

Answer:
Approximate Bayesian predictions from samples:

$$\int P(y \,|\, \mathbf{x}, \mathbf{w})\, p(\mathbf{w} \,|\, \text{data})\, \mathrm{d}\mathbf{w} \approx \frac{1}{S} \sum_{s=1}^{S} P(y \,|\, \mathbf{x}, \mathbf{w}^{(s)}), \quad \mathbf{w}^{(s)} \sim p(\mathbf{w} \,|\, \text{data}).$$

Average probabilities of predictions from each sampled weight. Then we can evaluate as before:

```
D = size(x_train, 2);
all_ww = samples(1:D, :); % DxS
pred_mc = mean(1./(1 + exp(-(x_test*all_ww))), 2);
report_score(y_test, pred_mc);

% Outputs (I didn't clamp the random seed, so not exactly reproducible):
% Accuracy: 0.9151 +/- 0.0069    MLP: -0.189 +/- 0.013
```

The accuracy is coincidentally exactly the same as before (the predictions make mistakes on some different cases though). The mean log probability is a little better, it appears taking the uncertainty in the weights into account helps. While the difference is comparable to the error bars on future test performance, we can do a paired comparison to look at the difference in mean performance between the two methods:

```
function paired_compare(y_test, pred1, pred2)
% Report how much better p(y=1) predictions are in pred1 than pred2
y_test = (y_test == 1);
c1 = (y_test == (pred1(:) > 0.5));
c2 = (y_test == (pred2(:) > 0.5));
D_acc = errorbar_str(c1-c2);
Lp1 = [log(pred1(y_test)); log(1 - pred1(~y_test))];
Lp2 = [log(pred2(y_test)); log(1 - pred2(~y_test))];
D_mlp = errorbar_str(Lp1 - Lp2);
fprintf('Accuracy improvement: %s    MLP improvement: %s\n', D_acc, D_mlp);
```

```
paired_compare(y_test, pred_mc, pred);

% Outputs:
% Accuracy improvement: 0.0e+00 +/- 0.003    MLP improvement: 0.0201 +/- 0.0064
```

The difference in mean log probability does appear to be positive. The difference is small though. It is unlikely that Bayesian inference significantly outperforms fitting a regularized model with λ set by cross-validation for simple classifiers.

3 Notes on MATLAB

- **Remember, there are only a limited number of licences for MATLAB. After you have finished using MATLAB, quit from the MATLAB session so that others can work.**
- Under the Resources heading on the PMR page `http://www.inf.ed.ac.uk/teaching/courses/pmr/` there are a number of MATLAB tutorials listed.
- You can find out more about most MATLAB functions by typing `help` followed by the function name. Also, you can find the `.m` file corresponding to a given function using the `which` command.
- `close all` closes all the figures. It helps if things get cluttered.
- Read about plots in the "Introduction to MATLAB" linked from the PMR homepage. Recall that the current figure can be saved as a PDF file `myplot.pdf` using `print -dpdf myplot.pdf`.