Machine Learning and Pattern Recognition Assignment

Due: 4pm Thursday 19 November, 2015

This assignment is out of 100 marks and forms 20% of your final grade.

Assessed work is subject to University regulations on academic conduct:
`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`
Do not show your code, answers or write-up to anyone else.
Never copy-and-paste material that's not your own into your assignment and edit it.
If you use any publicly-available functions that are not provided with the core of Matlab (as installed on DICE) or in Netlab, you must state where they are from, and explain what they do in your own words. Using any code from other students taking the class is not acceptable.

Late submissions: The School of Informatics policy is that late coursework normally gets a mark of zero. See `http://tinyurl.com/edinflate` for exceptions to this rule. Any requests for extensions should go to the ITO, either directly or via your Personal Tutor.

Submission instructions: You should submit your answers on paper to the ITO office in Forrest Hill by the deadline.

Provide relevant code fragments and plots within your answers where appropriate.

1 The Next Pixel Prediction Task

Natural signals are highly redundant. Neighbouring pixel intensities in natural images tend to be positively correlated because objects extend over finite spatial regions. In order to model the probability distribution of patches of natural images we need to account for these correlations.

Given a raster-scan ordering of the pixels, it is natural to predict the value of a pixel given nearby pixels lying above and to the left of it (its "context"). These predictions can be used in a lossless compression scheme. After predicting the value of a pixel given its context, we can code the 'residual', the error in the prediction, instead of the pixel itself. If we can make good predictions, the residuals are more narrowly distributed than the pixel values, and can be compressed well. (If interested, MacKay's Information Theory textbook gives more details in Chapter 6. However, you do not need to know how compression schemes work for this assignment or course.)

This question is about predicting pixel values given preceding pixels in an image patch. The setup is illustrated in Figure 1. The data is available in a Matlab file `imgregdata.mat`. It is split into training data `xtr` and `ytr`, and test data `xte` and `yte`. Other variables in this file are described later.

The Matlab data file for this question can be obtained at:
    `/afs/inf.ed.ac.uk/group/teaching/mlprdata/challengedata/imgregdata.mat`
This file is moderately large, 103MB. If working on DICE, please don't copy it (the class would create >10GB of copies). Instead, load it directly from the `/afs` location, or make a symbolic link (`ln -s` in unix) into your working directory.

The data are taken from Van Hateren's Natural Image Database `http://www.kyb.tuebingen.mpg.de/?id=227` (although you should not need to go into the details of this for the assignment). We describe how the patch data in the Matlab file were extracted in Algorithm 1. We picked 35×30 image patches. For each patch j we took the pixel in the middle at the bottom of the patch as the label $y(j)$, and the 1032 pixels to the left and above that pixel as context $\mathbf{x}(j,:)$. We will model predicting y from $\mathbf{x}$ as a regression problem.

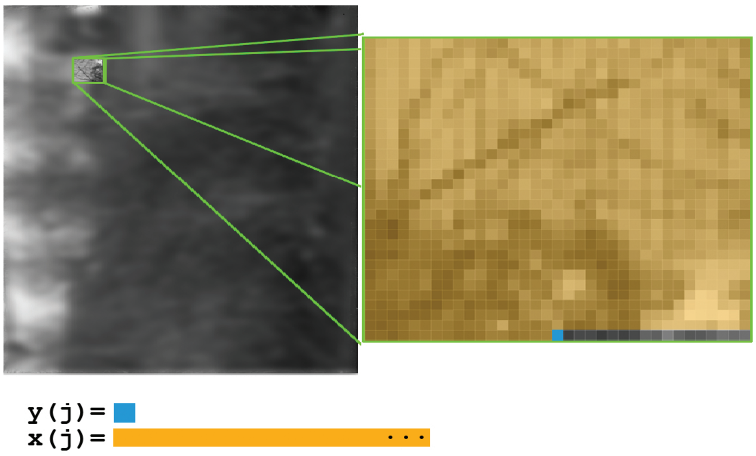

Figure 1: The task is to predict a pixel $y(j)$ given the values in a patch of pixels above and to the left of that pixel, $\mathbf{x}(j,:)$. The image patch is 35 pixels (horizontal) $\times$ 30 pixels (vertical).

Algorithm 1 Data Preparation

```
Load image data. Discretize to 64 gray scales.
for $j$=1 to 100,000 do
    Pick a random image and a random pixel at least 40 pixels away from edge of image
    and find a 35 $\times$ 30 patch including that pixel at the bottom-middle of the patch.
    Record $\mathbf{x}(j)$ =vectorisation of all pixels in patch 'before' that pixel in patch in raster-scan
    terms, $y(j)$ =grayscale value at chosen pixel.
end for
Produce two Matlab datasets. Set 1: xtr and ytr for 70,000 training records. Set 2: xte
and yte values for 30,000 test records.
```

1. **Data preprocessing and visualization (8 marks)**
   First scale all of the data to $[0,1]$ by dividing each pixel value by 63. This scaling is a commonly used preprocessing step in regression.

   The pixel intensity in some image patches is nearly constant. These "flat patches" can interfere with modelling the more interesting patches. To help identify flat patches, compute the standard deviation of each $\mathbf{x}(j,:)$ patch.

   (a) Plot a histogram (using 64 bins – why?) of the standard deviations of the patches in the training set. What does this plot tell you about the data?

   (b) Give a simple way of predicting the target pixel if its patch is flat.

   (c) Make a plot of one flat and one non-flat image patch, respectively.
   You will need to pad the patch, because of the missing pixels, and reshape it back into a rectangle. Use `colormap gray` to get black and white images. You should use the same colour range for each patch. By default `imagesc()` will scale up small differences in a flat patch. Giving it the second argument `[0,1]`, tells it the values that should correspond to black and white.

   We decided to select the data points with the standard deviation larger than $4/63$ as our final training and test sets, i.e., all the non-flat patches. The Matlab file `imgregdata.mat` also contains the "non flat" subset of the original data, denoted by a `_nf` suffix. We have scaled the values to be in $[0,1]$ for you. **You should use the `_nf` data for the remainder of the assignment.**

2. **Linear regression with adjacent pixels (10 marks)**
   As a first step, we consider a simple way of predicting the target pixel $y(j)$ given the preceding pixels, i.e., linear regression with the two neighbours to the left and above the target pixel, $x(j,\text{end})$ and $x(j,\text{end}-34)$ (in Matlab notation).

   (a) To obtain a general sense on the relationship of $y(j)$ and its two neighbours in the training set, visualize it using a 3D plot of $x(j,\text{end})$, $x(j,\text{end}-34)$ and $y(j)$.

There might be too many data points giving rise to overprinting, so you might need to subsample the data, for instance, only plotting 10,000 of them. Comment on the structure you observe in this 3-D plot.

(b) Denote the feature matrix as X, where each row is 3-dimensional, the first 2 dimensions are $x(j, \text{end})$ and $x(j, \text{end} - 34)$, and the third is simply 1, to allow for the bias weight in linear regression. Write down the closed-form maximum likelihood solution for the weight vector $\mathbf{w}$ using X and the vector $\mathbf{y}$.

(c) Implement the linear regression predictor, and show the relevant code fragment in your report. State the values of the weights obtained. Also state the root mean squared error (RMSE) on the training and test sets. Visualize the regression surface of this linear predictor in 3-D using Matlab function `surf()`. The following snippet is helpful for this visualization.
```
figure,
{[dim1, dim2]} = meshgrid(0:0.01:1,0:0.01:1);
ysurf = [[dim1(:), dim2(:)], ones(numel(dim1),1)]*w;
surf(dim1, dim2, reshape(ysurf, size(dim1)))
```
Also add the 3-D plot of the test data points to this figure, and comment on the performance of your linear regressor.

3. **RBF regression with adjacent pixels (8 marks)**
Now we will build a non-linear regression predictor for this task using a radial basis function (RBF) network. RBFs are implemented in Prof. Ian Nabney's Netlab toolbox[1]. Obtain netlab3.3 and the help files from the downloads page, along with `foptions.m`. See especially the functions `rbf` and `rbffwd`. Make sure that you read through the documentation and online help (e.g. type `help rbf` at the Matlab prompt) so you know what the various functions do. There are also demo scripts (e.g. `demrbf1.m`) that can help you.
Below is a code snippet for how to set up and train the RBF network. The RBF centres are determined by an unsupervised learning procedure (fitting a Gaussian mixture model with spherical covariances using the EM algorithm).
```
% Code for creating RBF and making predictions
% nbf is the number of basis functions
% dim is the dimensionality of input space
net = rbf(dim, nbf, 1, 'gaussian');

options = foptions;
options(1) = 1;     % Display EM training
options(14) = 5;    % number of iterations of EM
net = rbftrain(net, options, YourTrainingX,  YourTrainingY ); % train the net
ypred = rbffwd(net,  YourTestX); % use the net to predict the output
                                 % for YourTestX;
```
(a) Before we do RBF regression, we need to set how many basis functions we will use. Try different numbers of basis functions using the candidates $\{5, 10, 15, 20, 25, 30\}$. Use 10-fold cross validation to decide the optimal number of RBFs. You may wish to use the Matlab statistics toolbox function `crossval`. Plot the cross-validation RMSE against the number of RBFs used, and therefore select the best model. Provide code snippet(s) that would reproduce what you did.

(b) Given the choice for the number of basis functions as above, train the RBF network using all the non-flat training data `xtr_nf`. Report the RMSE on both the training and test sets, and comment on the results.

4. **Linear regression with all pixels (6 marks)**
Now we extend the context pixels from 2 neighbours to the whole image patch, i.e., using all the 1032 context pixels to predict the next pixel.
Implement the linear regression with all pixels, and report the RMSE on training and test sets. Comment on the performance of this linear regressor compared with previous two methods.

1. `http://www1.aston.ac.uk/eas/research/groups/ncrg/resources/netlab/`

5. **Neural Network with all pixels (10 marks)**
   We will now try a neural network regression predictor using all the pixels. We will use the simplest feedforward neural network, a multilayer perceptron (MLP). Netlab provides an implementation in `mlp.m`.
   Since tuning the parameters in neural network is generally a time-consuming job, we will fix the structure of MLP as follows: 2-layer network, 10 hidden neurons, the activation function of the first layer is hyperbolic tangent (tanh), and output layer uses a linear function. We train the network using scaled conjugate gradient, as described in the Netlab function `scg.m`.
   In part (b) below we set the seed of a random number generator (RNG) to obtain different network initializations. However this is not the proper way to use a RNG. A RNG produces a pseudo-random sequence from repeated calls to it without resetting the seed. However, setting the seed to different values is a convenient way to obtain different initializations in a reproducible fashion.
   (a) Here we provide a well-trained MLP with the suggested network structure, in the file
   `/afs/inf.ed.ac.uk/group/teaching/mlprdata/challengedata/welltrainedMLP.mat`
   Compute the RMSE on both the training and test set using Netlab function `mlpfwd()`. Compare its RMSEs with those from linear regression, and comment on the differences.
   (b) Try only using first 5000 data points of `xtr_nf` to train the MLP with the suggest network structure, and run the training 5 times *using different random seeds*, which can be set by `rng(2015,'twister')`, `rng(2016,'twister')`, `...,rng(2019,'twister')`. The different random seeds will induce different initializations for the MLP. We provide an example in the following code snippet. Report the training and testing RMSEs for each run. Comment on any differences, and explain why these may occur.

```
%% Neural Network with all pixels
rng(2015,'twister')
% Set up the network
nhid = 10; % number of hidden units
net = mlp(size(xtr_nf,2), nhid, 1, 'linear');

% Set up vector of options for the optimiser.
options = zeros(1,18);
options(1) = 1;            % This provides display of error values.
options(9) = 1;            % Check the gradient calculations.
options(14) = 200;         % Number of training cycles.

% Train using scaled conjugate gradients.
[net, options] = netopt(net, options, xtr_nf(1:5000,:), ytr_nf(1:5000,:), 'scg');
toc


% RMSE on training set
ypred_tr = mlpfwd(net, xtr_nf);
rmse_NNsuball_tr = sqrt(mean(((ytr_nf - ypred_tr).^2)))

% RMSE on test set
ypred = mlpfwd(net, xte_nf);
rmse_NNsuball_te = sqrt(mean(((yte_nf - ypred).^2)))
```

6. **Discussion (8 marks)**
   Above we have considered linear regression and RBF network using 2 neighbouring pixels, and linear regression and a neural network on all pixels. Compare these methods and provide some insights on their pros and cons for this task.
   Briefly, what experiment would you try next if you wanted to improve on these predictors?

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:

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

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:

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

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: $P(y=1 \mid \mathbf{x}=0) = 1/2$ for all settings of the weights. A more flexible classifier includes a 'bias' weight: $P(y \mid \mathbf{x}, \mathbf{w}) = \sigma(y(\mathbf{w}^\top\mathbf{x} + b))$. 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: $b = w_{D+1}$.

   (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`.

   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 $P(y \mid \mathbf{x}) = 0.5$ for every test case?

(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.

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:

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

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.

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.

(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 $\epsilon = \sigma(a)$.

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.

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.

   (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.

   (c) Explain how to use the samples of $\mathbf{w}$ to predict the test labels. Compare the predictions from sampling to those from the previously fitted model.

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`.