Machine Learning and Pattern Recognition
Assignment Feedback

2 Game prediction/ranking and logistic regression

I’ve put the most relevant code snippets in this report. Although full code is also available in a tar archive if there’s anything you’d like to run or check.

1. The Zermelo/Bradley–Terry model (10 marks)

(a) Game prediction as Logistic regression:

Substituting \( u_d = \exp(w_d) \) into the Zermelo model gives:

\[
P(p_1 \text{ beats } p_2) = \frac{u_{p_1}}{u_{p_1} + u_{p_2}} = \frac{\exp(w_{p_1})}{\exp(w_{p_1}) + \exp(w_{p_2})} = \frac{1}{1 + \exp(w_{p_2} - w_{p_1})}.
\]

(1)

We can encode the players \((p_1, p_2)\) of a game in a \(D\)-dimensional feature vector \(x\), by numbering the \(D\) players 1 ... \(D\), and setting

\[
x_d = \begin{cases} 
+1 & d = p_1 \\
-1 & d = p_2 \\
0 & \text{otherwise}.
\end{cases}
\]

(2)

Then \(P(y=1 \mid x)\) is the probability that \(p_1\) beats \(p_2\), so from Equation (1):

\[
P(y=1 \mid x) = \frac{1}{1 + \exp(-w^\top x)},
\]

(3)

because \(w^\top x = w_{p_1} - w_{p_2}\).

(b) Comparing and contrasting the approaches:

Integer player identifiers \((p_1, p_2)\) are a natural and compact way to specify which players met in a game. However, logistic regression assumes that the probability of the output depends only on a linear combination of the input features. Using the \((p_1, p_2)\) inputs in logistic regression would mean that the probability of player 2 winning a game would always have to be between the probabilities of players 1 and 3 winning against the same player. However, these integer identifiers are arbitrary! Encoding the inputs in feature-vector \(x\) gives logistic regression a separate parameter for each player, so it can learn an arbitrary ordering of their abilities.

While Naive Bayes can easily use the original and natural \((p_1, p_2)\) features, it could be run with the sparse \(x\) feature vectors instead, although there’s no obvious advantage. On the contrary, the independence assumption appears to be even more strongly violated with the \(x\) representation than before. Given the locations of the +1 and −1, all of the other features must be zero. A larger fraction of the possible settings of the features that the Naive Bayes model could generate will never actually be seen.

The symmetry property means that the probability that player \(a\) wins in pairing \((a, b)\) should be the probability that player \(b\) loses in the pairing \((b, a)\). Or equivalently:

\[
P(y=1 \mid x) = P(y=0 \mid -x) = 1 - P(y=1 \mid -x),
\]

(4)

where \(x_a = +1\) and \(x_b = -1\). This property does always hold in the logistic regression model:

\[
1 - P(y=1 \mid -x) = 1 - \frac{1}{1 + \exp(w^\top x)} = \frac{\exp(w^\top x)}{1 + \exp(w^\top x)} = \frac{1}{1 + \exp(-w^\top x)} = P(y=1 \mid x).
\]

(5)
We could break this symmetry by adding a bias weight to the logistic regression model

\[ P(y=1 \mid x) = \frac{1}{1 + \exp(-(w^\top x + b))}. \]

Larger values of \( b \) predict \( y=1 \) with higher probability, possibly suggesting that the white player has an advantage. (Although one has to be careful about such causal claims. See the note at the end of 2a) for an example of why.)

The Naive Bayes model is simple to fit, with a closed form solution, whereas logistic regression requires an iterative optimization algorithm. However, I much prefer the idea behind the logistic regression model: the outcome of a game depends on the difference in skill (as measured by the \( w_d \) weights) between the players. The Naive Bayes model makes a clearly-wrong assumption: the choice of players are not independent (they tend to be matched by a system, and we know that \( p_1 \neq p_2 \)).

If a player wins a lot of games, the probability that they win future games will increase under both models (however fitted). However, in the logistic regression model a loss of player \( a \) to player \( b \) means that weight \( w_a \) can’t be much larger than \( w_b \), and the model will not strongly predict that player \( a \) will then beat player \( b \). The Naive Bayes model does not consider interactions between the pairs that play. If player \( a \) mainly played weak players and won most of their games, and player \( b \) mainly played strong players and lost most of their games, it will confidently predict that \( a \) will beat \( b \), even if the training data suggests precisely the opposite happened whenever they met. Thus the Naive Bayes system is more ‘gameable’ in the sense described in the question.

Any reasonable argument was acceptable. Some answers assumed a case where a player only played weak players and won every game. Further assuming logistic regression would be fitted by maximum likelihood, then logistic regression will be worse than the (smoothed) naive Bayes method implemented here.

2. Fitting and testing the models (40 marks)

(a) **Naive Bayes:** The model’s predictions can be evaluated using the following routine. It creates a vector of zeros and ones indicating whether the hard guesses were correct, and a vector of the log-probabilities of the test labels under the predictions. It then reports the mean and standard error of each.

```matlab
function evaluate_pred(y_test, pred)
    correct = ((pred>0.5) == y_test);
    Lp = zeros(size(pred));
    Lp(y_test == 1) = log(pred(y_test == 1));
    Lp(y_test ~= 1) = log(1 - pred(y_test ~= 1));
    fprintf('Accuracy %s
', errorbar_str(correct));
    fprintf(' Mean Lp %s
', errorbar_str(Lp));

    Review the answers to tutorial 5 for how to compute standard errors on means (and the errorbar_str routine).
```
The routine is driven as follows:

```matlab
pred = game_naive_bayes_demo();
test_data = load('test_games.txt');
y_test = test_data(:, 3); % did p1 win?
evaluate_pred(y_test, pred);
```

Accuracy 0.696 +/- 0.043
Mean Lp -0.73 +/- 0.11

Thus the accuracy does appear to be greater than guessing uniformly at random (which has accuracy 0.5). The mean log probability is not distinguishable from this baseline however: \( \log 0.5 = -0.693 \). All logs here are natural logarithms. If you used logs base 2 or 10 that’s fine, but be sure to specify. Outside Information Theory texts, logs base \( e \) are standard.

You may have implemented another simple baseline (an acceptable interpretation of the question), always guessing \( y = 1 \). This baseline performs well and similarly to Naive Bayes.

```matlab
>> errorbar_str(y_test==1)
an =
0.713 +/- 0.042
>> errorbar_str(y_train==1)
an =
0.524 +/- 0.028
```

Although \( y = 1 \) is not significantly more frequent in the training data, so we wouldn’t really have strong grounds to favour it. We do see here the worrying fact that the data distribution has shifted over time (the test set was taken from games at later times). My conjecture is that as the system worked out how good players were (it had access to more data than we do), it has preferred to make weaker players be the black player \( p_2 \), which moves first. I don’t actually believe there is an advantage to being white!

(b) Logistic regression: The simplest way to write a negative log-likelihood routine was simply to wrap the provided code:

```matlab
function [nLp, dnLp_dw] = negative_loglike(ww, pp, yy)
[Lp, dLp_dw] = game_loglike(ww, pp, yy);
nLp = -Lp;
dnLp_dw = -dLp_dw;
```

I then took code provided in `game_naive_bayes_demo.m` to make a `grab_data.m` function to give access to the data. Then fitted the parameters as follows:

```matlab
[p_train, y_train, p_test, y_test, D] = grab_data();
ww = zeros(D, 1);
ww = minimize(ww, @(w) negative_loglike(w, p_train, y_train), 10000);
```

Some of the players in the training data won all their games, e.g., player 142. The maximum likelihood weight for this player is \( w_{142} = \infty \). The probability of this player winning their games is then 1, the maximum possible value. The gradients underflow to zero and `minimize.m` stops before the weights saturate. If we set \( w_{142} = \infty \); by hand, the log-likelihood goes up slightly.

Test predictions are obtained from the weights as follows:

```matlab
pred = 1./(1 + exp(-(ww(p_test(:,1)) - ww(p_test(:,2)))))
evaluate_pred(y_test, pred);
```

Accuracy 0.713 +/- 0.042
Mean Lp -Inf
The accuracy is similar to Naive Bayes. The mean log probability is infinitely bad. If a player that previously won all their games loses a game, the maximum likelihood weights assign that event zero probability. The model says something can never ever happen... and then it does.

On the training set logistic regression performed as well or better than Naive Bayes:

**LR on train set:**
- Accuracy 0.915 +/- 0.016
- Mean Lp -0.178 +/- 0.022

**NB on train set:**
- Accuracy 0.865 +/- 0.019
- Mean Lp -0.259 +/- 0.026

The log-probability results show that logistic regression with maximum likelihood has over-fitted: it has adapted the weights to obtain a high score on the training set, at the expense of generalizing infinitely badly.

(c) **Regularization:** We can regularize the log-likelihood as follows:

```matlab
function [nLp, dnLp_dw] = negative_penalized_loglike(ww, pp, yy, lambda)
    [Lp, dLp_dw] = game_loglike(ww, pp, yy);
    nLp = -Lp + lambda*(ww'*ww);
    dnLp_dw = -dLp_dw + (2*lambda)*ww;
end
```

(I did check these gradients. See the tar-ball for details.) If we set $\lambda = 0$, we'd recover the maximum likelihood solution, which we've seen overfits (and takes some time to optimize). In the limit $\lambda \to \infty$ we drive the weights to zero, and our predictions should tend to the $P(y=1|x) = 0.5$ baseline.

Setting the suggested $\lambda$, we optimize as before.

```matlab
lambda = 0.5;
ww = zeros(D, 1);
ww = minimize(ww, @(w) negative_penalized_loglike(w, ... p_train, y_train, lambda), 10000);
```

```
Accuracy 0.704 +/- 0.043
Mean Lp -0.563 +/- 0.040
```

The test accuracy is comparable to previous results. The mean log probability is much better than $-\infty$, and for the first time suggests predictions better than the $P(y=1|x) = 0.5$ baseline.

Although the accuracy isn’t better than Naive Bayes, I would prefer this model for matching players. It makes more sense as a model, and its probabilities seem slightly better.

3. ** Entirely optional: Hierarchical logistic regression and MCMC (0 marks) **

(a) The largest log-likelihood that any model of discrete data can have is $\log(1) = 0$. A logistic regression model with $w = 0$ has a finite log-likelihood of $N \log(0.5)$. Setting the weights to zero in the log-posterior

$$
\log P(w, \log \lambda | \text{data}) = L(w) - \lambda w^T w + \frac{D}{2} \log \lambda + \text{const.},
$$

makes the first two terms finite for any $\lambda$. Setting $\lambda$ infinite then makes the log-posterior infinite, and that’s the only way of doing so, because the other terms are bounded above.

Hence jointly optimizing the parameters of a hierarchical model can be problematic. We could regularize $\lambda$, but how much?
(b) We need a function for the log-posterior (up to a constant) of $\theta = \begin{bmatrix} w \\ \log \lambda \end{bmatrix}$.

```matlab
function Lp = log_hier(theta, pp, yy)
% Unpack parameters:
ww = theta(1:end-1);
D = length(ww);
log_lambda = theta(end);
lambda = exp(log_lambda);
% log posterior is log-likelihood + log-prior + const.
Lw = game_loglike(ww, pp, yy);
Lp = Lw - lambda*(ww'*ww) + (D/2)*log_lambda;
end
```

We can then explore this posterior distribution using slice sampling:

```matlab
lambda = 0.001; % start off too small and see if it gets corrected
ww = zeros(D, 1);
theta = [ww; log(lambda)];
S = 1000;
burn = 0; % set no burn in, so can look at what happens
widths = 1;
step_out = true; % Slower, but safer
samples = slice_sample(S, burn, @(t) log_hier(t, p_train, y_train), ...
theta, widths, step_out); % (D+1)xS
plot(exp(samples(end, :)));
```

The setting of $\lambda = 0.5$ seems plausible, as are values half and twice that:

![Trace plot of lambda](image)

Without doing any careful analysis, the trace-plot of $\lambda$ looks as if it captures a steady-state distribution. After an initial ‘burn-in’ phase while the sampler escapes from small values, there’s no obvious drift or jump in the distribution of $\lambda$.

(c) Predictions are made using:

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

(8)

Weight vectors from joint $(w, \lambda)$ posterior samples can be used as samples from the marginal posterior $P(w|\text{data})$. So we make predictions using each of the sampled weight vectors, and then average.

```matlab
M = length(y_test);
pred = zeros(size(y_test));
for nn = 1:M
    % (D+1)th element of samples, log-lambda, is safely ignored.
deltas = samples(p_test(nn, 1), :) - samples(p_test(nn, 2), :);
pred(nn) = mean(1 ./ (1 + exp(-deltas)));
end
```

Evaluating those predictions as before:

- Accuracy 0.704 +/- 0.043
- Mean Lp -0.563 +/- 0.039

Almost exactly the same as under the fitted regularized model.
(d) The posterior probability that player 1 is better than player 2 is

\[ P(w_1 > w_2 \mid \text{data}) = \int I(w_1 > w_2) P(w \mid \text{data}) \, dw \]

\[ \approx \frac{1}{S} \sum_{s=1}^{S} I(w_1^{(s)} > w_2^{(s)}), \quad w^{(s)} \sim P(w \mid \text{data}), \quad (9) \]

where \( I(\cdot) \) is an indicator function that evaluates to zero or one. We arrive at the simple numerical estimate:

\[
\begin{align*}
&\gg \text{mean(samples(1,:) > samples(2,:))} \\
&\text{ans} = 0.8730
\end{align*}
\]

More work would be required to put an error bar on this estimate, which depends on the “effective number of samples” gathered by the Markov chain.

However, none of the previously fitted models answer this question at all. They can give the probability that player 1 will beat player 2 in a game. However, that prediction comes from a single fit where player 1 is either better than player 2 or vice-versa. There was no measure of how reliable that conclusion was.

Some comments in summary:

Computing baselines and error bars is important to put results in context, and see if they mean anything. Getting a high accuracy on this task can be difficult because the web service is actively attempting to balance the games. In this small dataset we didn’t beat the test accuracy of the baseline ‘predict white always wins’. The log-probability measure is one way to test the confidence of the models. By computing a baseline and error bars we see that Naive Bayes doesn’t convincingly beat the log-probability of uniform guessing. Whereas logistic regression, if regularized, looks as if it does.

I was surprised by the plausible values of \( \lambda \) that resulted from cross-validation and sampling. Given the range of abilities of Go players, I expected the \( w_d \) values to be much more broadly spread out. There are a few possible problems. One is that some players are unreliable: they can have a bad day and lose to someone much worse than they really are. Another issue is that beginners rapidly get better, and so lose to bad players early on, and beat much better players later on. These issues will encourage the model to push the parameters for the players together, to explain both the losses to bad players and wins against good players.

To do a good job of player modelling, the system needs to model unreliable players that change over time. It should also look at more data, including handicapped games (omitted from the provided data, because the model would need extending), which compare players separated by larger skill differences.