The Logistic Regression Model

Michiel de Hoon (mdehoon@cal.berkeley.edu)

June 29, 2008

Contents

1 Background and Purpose

Logistic regression is a supervised learning approach that attempts to distinguish K classes from each other using a weighted sum of some predictor variables x_i. The logistic regression model is used to calculate the weights β_i of the predictor variables. In Biopython, the logistic regression model is currently implemented for two classes only ($K = 2$); the number of predictor variables has no predefined limit.

As an example, let's try to predict the operon structure in bacteria. An operon is a set of adjacent genes on the same strand of DNA that are transcribed into a single mRNA molecule. Translation of the single mRNA molecule then yields the individual proteins. For *Bacillus subtilis*, whose data we will be using, the average number of genes in an operon is about 2.4.

As a first step in understanding gene regulation in bacteria, we need to know the operon structure. For about 10% of the genes in *Bacillus subtilis*, the operon structure is known from experiments. A supervised learning method can be used to predict the operon structure for the remaining 90% of the genes.

For such a supervised learning approach, we need to choose some predictor variables x_i that can be measured easily and are somehow related to the operon structure. One predictor variable might be the distance in base pairs between genes. Adjacent genes belonging to the same operon tend to be separated by a relatively short distance, whereas adjacent genes in different operons tend to have a larger space between them to allow for promoter and terminator sequences. Another predictor variable is based on gene expression measurements. By definition, genes belonging to the same operon have equal gene expression profiles, while genes in different operons are expected to have different expression profiles. In practice, the measured expression profiles of genes in the same operon are not quite identical due to the presence of measurement errors. To assess the similarity in the gene expression profiles, we assume that the measurement errors follow a normal distribution and calculate the corresponding log-likelihood score.

We now have two predictor variables that we can use to predict if two adjacent genes on the same strand of DNA belong to the same operon:

- x_1: the number of base pairs between them;
- x_2: their similarity in expression profile.

In a logistic regression model, we use a weighted sum of these two predictors to calculate a joint score S:

$$S = \beta_0 + \beta_1 x_1 + \beta_2 x_2. \tag{1}$$

The logistic regression model gives us appropriate values for the parameters β_0, β_1, β_2 using two sets of example genes:

- OP: Adjacent genes, on the same strand of DNA, known to belong to the same operon;
- NOP: Adjacent genes, on the same strand of DNA, known to belong to different operons.

In the logistic regression model, the probability of belonging to a class depends on the score via the logistic function. For the two classes OP and NOP, we can write this as

$$\Pr(\text{OP}|x_1, x_2) = \frac{\exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}{1 + \exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)} \tag{2}$$

$$\Pr(\text{NOP}|x_1, x_2) = \frac{1}{1 + \exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2)} \tag{3}$$

Using a set of gene pairs for which it is known whether they belong to the same operon (class OP) or to different operons (class NOP), we can calculate the weights β_0, β_1, β_2 by maximizing the log-likelihood corresponding to the probability functions (2) and (3).

2 Training the logistic regression model

Table 1: Adjacent gene pairs known to belong to the same operon (class OP) or to different operons (class NOP). Intergene distances are negative if the two genes overlap.

Gene pair	Intergene distance (x_1)	Gene expression score (x_2)	Class
cotJA — *cotJB*	-53	-200.78	OP
yesK — *yesL*	117	-267.14	OP
lplA — *lplB*	57	-163.47	OP
lplB — *lplC*	16	-190.30	OP
lplC — *lplD*	11	-220.94	OP
lplD — *yetF*	85	-193.94	OP
yfmT — *yfmS*	16	-182.71	OP
yfmF — *yfmE*	15	-180.41	OP
citS — *citT*	-26	-181.73	OP
citM — *yflN*	58	-259.87	OP
yfiI — *yfiJ*	126	-414.53	NOP
lipB — *yfiQ*	191	-249.57	NOP
yfiU — *yfiV*	113	-265.28	NOP
yfhH — *yfhI*	145	-312.99	NOP
cotY — *cotX*	154	-213.83	NOP
yjoB — *rapA*	147	-380.85	NOP
ptsI — *splA*	93	-291.13	NOP

Table 1 list some of the *Bacillus subtilis* gene pairs for which the operon structure is known. Let's calculate the logistic regression model from these data:

```
>>> from Bio import LogisticRegression
```

```
>>> xs = [[-53, -200.78],
          [117, -267.14],
          [57, -163.47],
          [16, -190.30],
          [11, -220.94],
          [85, -193.94],
          [16, -182.71],
          [15, -180.41],
          [-26, -181.73],
          [58, -259.87],
          [126, -414.53],
          [191, -249.57],
          [113, -265.28],
          [145, -312.99],
          [154, -213.83],
          [147, -380.85],
          [93, -291.13]]
>>> ys = [1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          1,
          0,
          0,
          0,
          0,
          0,
          0,
          0]
>>> model = LogisticRegression.train(xs, ys)
```

Here, `xs` and `ys` are the training data: `xs` contains the predictor variables for each gene pair, and `ys` specifies if the gene pair belongs to the same operon (`1`, class OP) or different operons (`0`, class NOP). The resulting logistic regression model is stored in `model`, which contains the weights β_0, β_1, and β_2:

```
>>> model.beta
[8.9830290157144681, -0.035968960444850887, 0.02181395662983519]
```

Note that β_1 is negative, as gene pairs with a shorter intergene distance have a higher probability of belonging to the same operon (class OP). On the other hand, β_2 is positive, as gene pairs belonging to the same operon typically have a higher similarity score of their gene expression profiles. The parameter β_0 is positive due to the higher prevalence of operon gene pairs than non-operon gene pairs in the training data.

The function `train` has two optional arguments: `update_fn` and `typecode`. The `update_fn` can be used to specify a callback function, taking as arguments the iteration number and the log-likelihood. With the callback function, we can for example track the progress of the model calculation (which uses a Newton-Raphson iteration to maximize the log-likelihood function of the logistic regression model):

```
>>> def show_progress(iteration, loglikelihood):
```

```
        print "Iteration:", iteration, "Log-likelihood function:", loglikelihood
>>>
>>> model = LogisticRegression.train(xs, ys, update_fn=show_progress)
Iteration: 0 Log-likelihood function: -11.7835020695
Iteration: 1 Log-likelihood function: -7.15886767672
Iteration: 2 Log-likelihood function: -5.76877209868
Iteration: 3 Log-likelihood function: -5.11362294338
Iteration: 4 Log-likelihood function: -4.74870642433
Iteration: 5 Log-likelihood function: -4.50026077146
Iteration: 6 Log-likelihood function: -4.31127773737
Iteration: 7 Log-likelihood function: -4.16015043396
Iteration: 8 Log-likelihood function: -4.03561719785
Iteration: 9 Log-likelihood function: -3.93073282192
Iteration: 10 Log-likelihood function: -3.84087660929
Iteration: 11 Log-likelihood function: -3.76282560605
Iteration: 12 Log-likelihood function: -3.69425027154
Iteration: 13 Log-likelihood function: -3.6334178602
Iteration: 14 Log-likelihood function: -3.57900855837
Iteration: 15 Log-likelihood function: -3.52999671386
Iteration: 16 Log-likelihood function: -3.48557145163
Iteration: 17 Log-likelihood function: -3.44508206139
Iteration: 18 Log-likelihood function: -3.40799948447
Iteration: 19 Log-likelihood function: -3.3738885624
Iteration: 20 Log-likelihood function: -3.3423876581
Iteration: 21 Log-likelihood function: -3.31319343769
Iteration: 22 Log-likelihood function: -3.2860493346
Iteration: 23 Log-likelihood function: -3.2607366863
Iteration: 24 Log-likelihood function: -3.23706784091
Iteration: 25 Log-likelihood function: -3.21488073614
Iteration: 26 Log-likelihood function: -3.19403459259
Iteration: 27 Log-likelihood function: -3.17440646052
Iteration: 28 Log-likelihood function: -3.15588842703
Iteration: 29 Log-likelihood function: -3.13838533947
Iteration: 30 Log-likelihood function: -3.12181293595
Iteration: 31 Log-likelihood function: -3.10609629966
Iteration: 32 Log-likelihood function: -3.09116857282
Iteration: 33 Log-likelihood function: -3.07696988017
Iteration: 34 Log-likelihood function: -3.06344642288
Iteration: 35 Log-likelihood function: -3.05054971191
Iteration: 36 Log-likelihood function: -3.03823591619
Iteration: 37 Log-likelihood function: -3.02646530573
Iteration: 38 Log-likelihood function: -3.01520177394
Iteration: 39 Log-likelihood function: -3.00441242601
Iteration: 40 Log-likelihood function: -2.99406722296
Iteration: 41 Log-likelihood function: -2.98413867259
```

The iteration stops once the increase in the log-likelihood function is less than 0.01. If no convergence is reached after 500 iterations, the `train` function returns with an `AssertionError`.

The optional keyword `typecode` can almost always be ignored. This keyword allows the user to choose the type of Numeric matrix to use. In particular, to avoid memory problems for very large problems, it may be necessary to use single-precision floats (Float8, Float16, etc.) rather than double, which is used by default.

3 Using the logistic regression model for classification

Classification is performed by calling the `classify` function. Given a logistic regression model and the values for x_1 and x_2 (e.g. for a gene pair of unknown operon structure), the `classify` function returns `1` or `0`, corresponding to class OP and class NOP, respectively. For example, let's consider the gene pairs *yxcE*, *yxcD* and *yxiB*, *yxiA*:

Table 2: Adjacent gene pairs of unknown operon status.

Gene pair	Intergene distance x_1	Gene expression score x_2
yxcE — *yxcD*	6	-173.143442352
yxiB — *yxiA*	309	-271.005880394

The logistic regression model classifies *yxcE*, *yxcD* as belonging to the same operon (class OP), while *yxiB*, *yxiA* are predicted to belong to different operons:

```
>>> print "yxcE, yxcD:", LogisticRegression.classify(model, [6,-173.143442352])
yxcE, yxcD: 1
>>> print "yxiB, yxiA:", LogisticRegression.classify(model, [309, -271.005880394])
yxiB, yxiA: 0
```

(which, by the way, agrees with the biological literature).

To find out how confident we can be in these predictions, we can call the `calculate` function to obtain the probabilities (equations (2) and 3) for class OP and NOP. For *yxcE*, *yxcD* we find

```
>>> q, p = LogisticRegression.calculate(model, [6,-173.143442352])
>>> print "class OP: probability =", p, "class NOP: probability =", q
class OP: probability = 0.993242163503 class NOP: probability = 0.00675783649744
```

and for *yxiB*, *yxiA*

```
>>> q, p = LogisticRegression.calculate(model, [309, -271.005880394])
>>> print "class OP: probability =", p, "class NOP: probability =", q
class OP: probability = 0.000321211251817 class NOP: probability = 0.999678788748
```

To get some idea of the prediction accuracy of the logistic regression model, we can apply it to the training data:

```
>>> for i in range(len(ys)):
        print "True:", ys[i], "Predicted:", LogisticRegression.classify(model, xs[i])
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
```

```
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
```

showing that the prediction is correct for all but one of the gene pairs. A more reliable estimate of the prediction accuracy can be found from a leave-one-out analysis, in which the model is recalculated from the training data after removing the gene to be predicted:

```
>>> for i in range(len(ys)):
        model = LogisticRegression.train(xs[:i]+xs[i+1:], ys[:i]+ys[i+1:])
        print "True:", ys[i], "Predicted:", LogisticRegression.classify(model, xs[i])
True: 1 Predicted: 1
True: 1 Predicted: 0
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 1 Predicted: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 0
True: 0 Predicted: 1
True: 0 Predicted: 0
True: 0 Predicted: 0
```

The leave-one-out analysis shows the the prediction of the logistic regression model is incorrect for only two of the gene pairs, which corresponds to a prediction accuracy of 88%.

4 Logistic Regression, Linear Discriminant Analysis, and Support Vector Machines

The logistic regression model is similar to linear discriminant analysis. In linear discriminant analysis, the class probabilities also follow equations (2) and (3). However, instead of estimating the coefficients β directly, we first fit a normal distribution to the predictor variables x. The coefficients β are then calculated from the means and covariances of the normal distribution. If the distribution of x is indeed normal, then we expect linear discriminant analysis to perform better than the logistic regression model. The logistic regression model, on the other hand, is more robust to deviations from normality.

Another similar approach is a support vector machine with a linear kernel. Such an SVM also uses a linear combination of the predictors, but estimates the coefficients β from the predictor variables x near the boundary region between the classes. If the logistic regression model (equations (2) and (3)) is a good description for x away from the boundary region, we expect the logistic regression model to perform better than an SVM with a linear kernel, as it relies on more data. If not, an SVM with a linear kernel may perform better.

5 Literature

Trevor Hastie, Robert Tibshirani, and Jerome Friedman: *The Elements of Statistical Learning. Data Mining, Inference, and Prediction.* Springer Series in Statistics, 2001. Chapter 4.4.