A Note on Platt's Probabilistic Outputs for Support Vector Machines

Hsuan-Tien Lin (htlin@ntu.edu.tw)
Chih-Jen Lin (cjlin@csie.ntu.edu.tw)
Department of Computer Science and Information Engineering, National Taiwan University, Taipei 106, Taiwan

Ruby C. Weng (chweng@nccu.edu.tw)
Department of Statistics, National Chengchi University, Taipei 116, Taiwan

Abstract. Platt's probabilistic outputs for Support Vector Machines (Platt, 2000) has been popular for applications that require posterior class probabilities. In this note, we propose an improved algorithm that theoretically converges and avoids numerical difficulties. A simple and ready-to-use pseudo code is included.

Keywords: Support Vector Machine, Posterior Probability

1. Introduction

Given training examples $x_i \in \mathbb{R}^n, i = 1, \ldots, l$, labeled by $y_i \in \{+1, -1\}$, the binary Support Vector Machine (SVM) computes a decision function $\mathrm{f}(x)$ such that $\mathrm{sign}(\mathrm{f}(x))$ can be used to predict the label of any test example x.

Instead of predicting the label, many applications require a posterior class probability $\Pr(y = 1|x)$. Platt (2000) proposes approximating the posterior by a sigmoid function

$$\Pr(y = 1|x) \approx P_{A,B}(f) \equiv \frac{1}{1 + \exp(Af + B)}, \text{ where } f = \mathrm{f}(x). \tag{1}$$

Let each f_i be an estimate of $\mathrm{f}(x_i)$. The best parameter setting $z^* = (A^*, B^*)$ is determined by solving the following regularized maximum likelihood problem (with N_+ of the y_i's positive, and N_- negative):

$$\min_{z=(A,B)} \quad F(z) = -\sum_{i=1}^{l} \Big(t_i \log(p_i) + (1 - t_i) \log(1 - p_i) \Big), \tag{2}$$

$$\text{for} \quad p_i = P_{A,B}(f_i), \text{ and } t_i = \begin{cases} \frac{N_+ + 1}{N_+ + 2} & \text{if } y_i = +1 \\ \frac{1}{N_- + 2} & \text{if } y_i = -1 \end{cases}, i = 1, \ldots, l.$$

Platt (2000) gives a pseudo code for solving (2). In this note, we show how the pseudo code could be improved. We analyze (2) in Section 2, and propose a more robust algorithm to solve it. Better implementation that avoids numerical difficulties is then discussed in Section 3. We compare our algorithm with Platt's in Section 4. Finally, a ready-to-use pseudo code is in Appendix C.

2. Choice of Optimization Algorithm

We first introduce the simple optimization algorithm used in Platt's pseudo code (Platt, 2000). Then, after proving that (2) is a convex optimization problem, we propose a more robust algorithm that enjoys similar simplicity, and theoretically converges.

2.1. Platt's Approach: Levenberg-Marquardt Method

Platt (2000) uses a Levenberg-Marquardt (LM) algorithm from Press et al. (1992) to solve (2). The LM method was originally designed for solving nonlinear least-square problems. As an iterative procedure, at the k-th step, this method solves $(\tilde{H}_k + \lambda_k I)\delta_k = -\nabla F(z_k)$ to obtain a direction δ_k, and moves the solution from z_k to $z_{k+1} = z_k + \delta_k$ if the function value is sufficiently decreased. Here, $\tilde{H}_k$ is a special approximation of the Hessian of the least-square problem, I is the identity matrix, and $\{z_k\}_{k=0}^{\infty}$ is the sequence of iteration vectors. When λ_k is large, δ_k is close to the negative gradient direction. On the contrary, a small λ_k leads δ_k to be more like a Newton's direction.

In the pseudo code, Platt (2000) adapts the following rule for updating λ_k (Press et al., 1992):

$$\text{If } F(z_k + \delta_k) < F(z_k) \text{ then } \lambda_{k+1} \leftarrow 0.1 \cdot \lambda_k \text{ ; Else } \lambda_{k+1} \leftarrow 10 \cdot \lambda_k.$$

That is, if the new solution decreases the function value, λ_k is reduced, and in the next iteration a more aggressive direction like Newton's is tried. Otherwise, δ_k is unacceptable so we increase λ_k to obtain a shorter vector which, more likely being a descent direction, may lower the function value.

Unfortunately, such an implementation may not converge to the minimum solution of (2). To the best of our knowledge, existing convergence proofs (e.g., Moré, 1978) all require more complicated or more robust rules for updating λ_k.

In fact, since (2) is not exactly a least-squares problem, the implementation of Platt (2000) aims for general unconstrained optimization. It is known (e.g., Fletcher, 1987) that for unconstrained optimization we should avoid directly dealing with λ_k. Instead, the update of λ_k can be replaced by a trust-region concept, where the size of δ_k is controlled.

Thus, currently the optimization community uses trust-region methods for unconstrained optimization and the LM method is considered as its "progenitor" (Nocedal and Wright, 1999).

The LM-type implementation of Platt (2000) has one advantage: simplicity. However, the above discussion shows that it may not be the best choice for solving (2). Next, we propose an algorithm that is also simple, but enjoys better convergence properties.

2.2. Our Approach: Newton's Method with Backtracking

As indicated by Platt (2000), any method for unconstrained optimization can be used for solving (2). Before we choose a suitable method, we analyze the optimization problem in more detail. First, the gradient $\nabla F(z)$ and the Hessian matrix $H(z) = \nabla^2 F(z)$ are computed:

$$\begin{aligned}
\nabla F(z) &= \begin{bmatrix} \sum_{i=1}^{l} f_i(t_i - p_i) \\ \sum_{i=1}^{l}(t_i - p_i) \end{bmatrix}, \\
H(z) &= \begin{bmatrix} \sum_{i=1}^{l} f_i^2 p_i(1-p_i) & \sum_{i=1}^{l} f_i p_i(1-p_i) \\ \sum_{i=1}^{l} f_i p_i(1-p_i) & \sum_{i=1}^{l} p_i(1-p_i) \end{bmatrix}.
\end{aligned}$$

Some analysis of this Hessian matrix is in the following theorem:

Theorem 1 *The Hessian matrix $H(z)$ is positive semi-definite. In addition, $H(z)$ is positive definite if and only if $\min\limits_{1\le i\le l} f_i \neq \max\limits_{1\le i\le l} f_i$.*

The proof is in Appendix A. Therefore, problem (2) is convex (and in general strictly convex). With such a nice property, we decide to use a simple Newton's method with backtracking line search (Algorithm 6.2, Nocedal and Wright, 1999, and Section 10.5, Nash and Sofer, 1996). Though the trust-region type method mentioned in the end of Section 2.1 may be more robust, the implementation is more complicated. For this two-variable optimization problem, simplicity is important, and hence trust-region methods are less favorable.

Our proposed algorithm is in Algorithm 1. As $H_k = H(z_k)$ may be singular, a small positive diagonal matrix is added to the Hessian. With

$$\nabla F(z_k)^T \delta_k = -\nabla F(z_k)^T (H_k + \sigma I)^{-1} \nabla F(z_k) < 0,$$

the step size α_k can be backtracked until the sufficient decrease condition (3) is satisfied.

If $H(z)$ is positive definite for all z, the convergence of Algorithm 1 can be established from, for example, Theorem 10.2 by Nash and Sofer (1996). A simplified statement is shown in Theorem 2.

```
Input: Initial point $z_0$, and parameter $\sigma \geq 0$ such that $H(z) + \sigma I$ is
    positive definite for all $z$
1: for $k = 0, 1, 2, \cdots$ do
2:     Solve $(H_k + \sigma I)\delta_k = -\nabla F(z_k)$
3:     Find $\alpha_k$ as the first element of the sequence $1, \frac{1}{2}, \frac{1}{4}, \cdots$ to satisfy
           $F(z_k + \alpha_k \delta_k) \leq F(z_k) + 0.0001 \cdot \alpha_k \left(\nabla F(z_k)^T \delta_k\right)$   (3)
4:     Set $z_{k+1} = z_k + \alpha_k \delta_k$
5: end for
```

Algorithm 1: Newton's method with backtracking line search

Theorem 2 *(Convergence of Algorithm 1 for general $F(z)$)*
If $F(z)$ is twice continuously differentiable, $H(z)$ is positive definite for all z, and $F(z)$ attains an optimal solution at z^*, then $\lim_{k\to\infty} z_k = z^*$.

From Theorem 1, in some rare situations, $H(z)$ is positive semi-definite but not positive definite. Then, Theorem 2 cannot be directly applied. In Appendix B, we show that if $\sigma > 0$, Algorithm 1 still converges to an optimal solution. Therefore, we get the following theorem:

Theorem 3 *(Convergence of Algorithm 1 for (2))*
If Algorithm 1 is applied to (2) such that $H(z) + \sigma I$ is always positive definite, then $\lim_{k\to\infty} z_k$ exists and is a global optimal solution.

3. Numerical Implementation

Next, we study the numerical difficulties that arise when solving (2) using Platt's pseudo code. Then, we show our implementation that avoids the difficulties.

3.1. Platt's Implementation

Platt (2000) uses the following pseudo code to calculate the objective value of (2) for a new pair of (A, B):

```
for i = 1 to len {
  p = 1/(1+exp(deci[i]*A+B))
  // At this step, make sure log(0) returns -200
  err -= t*log(p)+(1-t)*log(1-p)
}
```

Here, `len` is l, the number of examples used, and `err` is the objective value. In addition, `deci[i]` is f_i, and hence `p` stores the calculated p_i. However, `t` was lastly assigned to t_l before this loop, and the calculation does not use all $t_i, i = 1, \ldots, l$. Therefore, this pseudo code does not correctly calculate the objective value of (2).

Furthermore, the above code assumes that $\log(0)$ returns -200, which reveals possible numerical difficulties:

1. log and exp could easily cause an overflow. If $Af_i + B$ is large, $\exp(Af_i + B) \to \infty$. In addition, when p_i is near zero, $\log(p_i) \to -\infty$. Although these problems do not always happen, considering $\log(0)$ to be -200 is not a good solution.

2. $1-p_i = 1-\frac{1}{1+\exp(Af_i+B)}$ is a "catastrophic cancellation" (Goldberg, 1991) when p_i is close to one. That is, when subtracting two nearby numbers that are already results of floating-point operations, the relative error can be so large that *most digits are meaningless.* For example, if $f_i = 1$, and $(A, B) = (-64, 0)$, in a simple C++ program with double precision, $1-p_i$ returns zero but its equivalent form $\frac{\exp(Af_i+B)}{1+\exp(Af_i+B)}$ gives a more accurate result. This catastrophic cancellation actually introduces most of the $\log(0)$ occurrences.

Almost all algorithms that solve (2) need to face these issues. Next, we will discuss some techniques to resolve them.

3.2. Our Implementation

A problem of catastrophic cancellation can usually be resolved by reformulation:

$$\begin{aligned} & -\Big(t_i \log p_i + (1 - t_i)\log(1 - p_i)\Big) && (4)\\ = \; & (t_i - 1)(Af_i + B) + \log\Big(1 + \exp(Af_i + B)\Big) && (5)\\ = \; & t_i(Af_i + B) + \log\Big(1 + \exp(-Af_i - B)\Big). && (6)\end{aligned}$$

With (5) or (6), $1-p_i$ does not appear. Moreover, $\log(0)$ never happens.[1]

Note, however, that even if (5) or (6) is used, the overflow problem may still occur. The problem is not serious if the IEEE floating-point standard is supported (Goldberg, 1991): an overflow leads to a special number INF, which can still be used in further operations. For example,

[1] As pointed out by a reviewer, in many popular languages, `log(1+...)` can be replaced by `log1p(...)` to compute the result more accurately when the operand $\exp(Af_i + B)$ or $\exp(-Af_i - B)$ is close to zero.

if a large α_k in Line 3 of Algorithm 1 makes the exp operation of (5) to overflow for some $Af_i + B$, the new objective value would also be evaluated as INF. Then, under the IEEE standard, INF is bigger than the current $F(z_k)$, and hence α_k is reduced to a smaller value, with which $Af_i + B$ may not cause an overflow again.

Furthermore, regardless of whether the IEEE standard is supported, we can replace an overflow operation with an underflow one, a rule-of-thumb which has been frequently used in numerical computation. In general, an underflow is much less disastrous than an overflow. Therefore, we propose implementing (4) with the rule:

$$\text{If } Af_i + B \geq 0 \text{ then use (6); Else use (5).}$$

In addition, we can evaluate (1) by a similar trick:

$$\text{If } Af + B \geq 0 \text{ then use } \frac{\exp(-Af-B)}{1+\exp(-Af-B)}\text{; Else use (1).}$$

The trick can be used in calculating $\nabla F(z)$ and $H(z)$ as well: The term $1-p_i$ in $H(z)$ can also cause a catastrophic cancellation. An easy solution is to replace $1-p_i$ with the rule:

$$\text{If } Af_i + B \geq 0 \text{ then use } \frac{1}{1+\exp(-Af_i-B)}\text{; Else use } \frac{\exp(Af_i+B)}{1+\exp(Af_i+B)}.$$

4. Experiment

We implemented Platt's pseudo code (Platt, 2000), fixed the bug that was discussed in the beginning of Subsection 3.1, and compared it to our proposed algorithm. For fairness, both algorithms were realized in python, and were set with a stopping condition $\|\nabla F(z_k)\|_\infty < 10^{-5}$.

For the value of σ in Algorithm 1, we considered two approaches:

- fixed: use a small fixed $\sigma = 10^{-12}$.
- dynamic: apply Theorem 1 to check whether $H(z)$ is positive definite, and set $\sigma = 0$ instead if the condition is true.

We compared the algorithms on two UCI data sets, sonar and shuttle (D. J. Newman and Merz, 1998). Only classes 2 and 4 were taken from shuttle to form a binary problem. The values f_i were generated with the scaled data sets by LIBSVM using the RBF kernel (Chang and Lin, 2001). The soft-margin parameter $\log_2 C$ was varied in -5, -3, $\cdots$, 15, and the kernel parameter $\log_2 \gamma$ was varied in -15, -13, $\cdots$, 3. That is, 110 different problems (2) were tested for each data set.

Tables I and II list the average results for each data set. We first compared each algorithm based on the number of overflow errors encountered, the number of iterations, and the final objective value $F(z)$.

Table I. Average results of different algorithms for solving (2) on sonar

algorithm	# overflow errors	# iterations	final $F(z)$	# backtracking steps per iteration
Platt's	0	5.77	107.78	—
ours, fixed	0	5.56	107.78	0
ours, dynamic	0	5.56	107.78	0

Table II. Average results of different algorithms for solving (2) on shuttle

algorithm	# overflow errors	# iterations	final $F(z)$	# backtracking steps per iteration
Platt's	589.30	8.00	158.62	—
ours, fixed	0	6.66	157.83	0.17
ours, dynamic	0	6.68	157.83	0.24

While Platt's algorithm did reasonably well on sonar, it encountered numerous overflow errors on shuttle, needed more iterations, and sometimes could not return a solution with decent $F(z)$. On the other hand, our proposed algorithm worked well on both data sets.

The number of backtracking steps per iteration was also listed for the two approaches of setting σ. We can see that the fixed approach needed less backtracking steps per iteration on shuttle. The benefit came from the regularization on some nearly singular $H(z)$. In addition, the fixed approach is simpler to implement in practice, and hence shall be preferred.

Finally, a simple and robust code is in Appendix C. It has been integrated into LIBSVM since version 2.6 (Chang and Lin, 2001). Source code in several popular languages can be downloaded at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools`.

Acknowledgment

We thank John Platt, S. Sathiya Keerthi, and the anonymous reviewers for helpful comments.

References

Chang, C.-C. and C.-J. Lin: 2001, 'LIBSVM: a library for support vector machines'. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

D. J. Newman, S. Hettich, C. L. B. and C. J. Merz: 1998, 'UCI Repository of machine learning databases'. Technical report, University of California, Irvine, Dept. of Information and Computer Sciences.

Fletcher, R.: 1987, *Practical Methods of Optimization.* John Wiley and Sons.

Goldberg, D.: 1991, 'What every computer scientist should know about floating-point arithmetic'. *ACM Computing Surveys* **23**(1), 5–48.

Moré, J. J.: 1978, 'The Levenberg-Marquardt algorithm Implementation and theory'. In: G. Watson (ed.): *Numerical Analysis.* New York, pp. 105–116, Sprmger-Verlag.

Nash, S. G. and A. Sofer: 1996, *Linear and Nonlinear Programming.* McGraw-Hill.

Nocedal, J. and S. J. Wright: 1999, *Numerical Optimization.* New York, NY: Springer-Verlag.

Platt, J.: 2000, 'Probabilistic outputs for support vector machines and comparison to regularized likelihood methods'. In: A. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans (eds.): *Advances in Large Margin Classifiers.* Cambridge, MA, MIT Press.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling: 1992, *Numerical Recipes: The Art of Scientific Computing.* Cambridge (UK) and New York: Cambridge University Press, 2nd edition.

Appendix

A. Proof of Theorem 1

Since the definition of p_i in (1) implies that $0 < p_i < 1$, we can define vectors u and v with $u_i = f_i\sqrt{p_i(1-p_i)}$, and $v_i = \sqrt{p_i(1-p_i)}$, respectively. Then $H(z) = \begin{bmatrix} u^Tu & u^Tv \\ v^Tu & v^Tv \end{bmatrix}$. By Cauchy inequality,

$$\det\Big(H(z)\Big) = \left(\sum_{i=1}^{l} u_i^2\right)\left(\sum_{i=1}^{l} v_i^2\right) - \left(\sum_{i=1}^{l} u_i v_i\right)^2 \geq 0. \tag{7}$$

Since the two diagonal terms and the determinant are all nonnegative, the matrix $H(z)$ is positive semi-definite.

From (7), $\det\Big(H(z)\Big) = 0$ if and only if u and v are parallel vectors. Since $u_i = f_i v_i$ and $v_i > 0$, this situation happens if and only if all f_i's are equal. That is, the matrix $H(z)$ is positive definite if and only if $\min\limits_{1\le i\le l} f_i \neq \max\limits_{1\le i\le l} f_i$.

B. Proof of Theorem 3

Case 1: $H(z)$ is always positive definite. If one can prove that

$$S = \Big\{(A,B)\colon F(A,B) \leq F(A_0,B_0)\Big\} \tag{8}$$

is bounded, then $F(A,B)$ attains an optimal solution within S and Theorem 2 can be applied to show the convergence.

From Theorem 1, assume without loss of generality that $f_1 \neq f_2$. Let $\hat{a} = \begin{bmatrix} f_1 & 1 \\ f_2 & 1 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix}$. Since $\begin{bmatrix} f_1 & 1 \\ f_2 & 1 \end{bmatrix}$ is invertible, it suffices to show that $\hat{S} = \{\hat{a} \colon (A, B) \in S\}$ is bounded. If not, there exists an infinite sequence $\{\hat{a}_k\}_{k=1}^{\infty}$ in $\hat{S}$ such that

$$\lim_{k\to\infty} \max\Big(|(\hat{a}_k)_1|, |(\hat{a}_k)_2|\Big) = \infty.$$

Then, without loss of generality, there exists an infinite subsequence $\mathcal{K}$ such that $\lim_{k\to\infty, k\in\mathcal{K}} |(a_k)_1| = \infty$. However, since $F(A_k, B_k)$ is the summation of positive terms,

$$F(A_k, B_k) \;\geq\; -t_1 \log \frac{1}{1 + e^{(\hat{a}_k)_1}} - (1 - t_1) \log \frac{e^{(\hat{a}_k)_1}}{1 + e^{(\hat{a}_k)_1}}.$$

The right-hand-side above goes to ∞ as $|(\hat{a}_k)_1| \to \infty$. Therefore, there exists some k such that $F(A_k, B_k) > F(A_0, B_0)$, which somehow contradicts $\hat{a}_k \in \hat{S}$. Thus, $\hat{S}$ is bounded and the proof is complete.

Case 2: When $H(z)$ is only positive semi-definite for some z, from Theorem 1, all f_i's are equal. By considering $f_i = f$ for all i, we can define $a = Af + B$ and a single-variable function $\bar{F}(a) = F(A, B)$. Then

$$\bar{F}'(a) = \sum_{i=1}^{l} t_i - \frac{l}{1 + e^a}, \; \bar{F}''(a) = \frac{l e^a}{(1 + e^a)^2}.$$

By simplifying (3), in Algorithm 1, $(H(z) + \sigma I)\delta = -\nabla F(z)$ is

$$\left(\tfrac{l e^a}{(1+e^a)^2} \begin{bmatrix} f^2 & f \\ f & 1 \end{bmatrix} + \sigma I\right) \begin{bmatrix} (\delta)_1 \\ (\delta)_2 \end{bmatrix} = -\left(\sum_{i=1}^{l} t_i - \tfrac{l}{1+e^a}\right) \begin{bmatrix} f \\ 1 \end{bmatrix}. \tag{9}$$

If $\sigma > 0$, the solution δ satisfies $(\delta)_1 = f \cdot (\delta)_2$. Then, the first (and the second) equation of the linear system (9) is the same as

$$\left(\bar{F}''(a) + \frac{\sigma}{f^2 + 1}\right) \Big(f \cdot (\delta)_1 + (\delta)_2\Big) = -\bar{F}'(a). \tag{10}$$

Interestingly, if we apply Algorithm 1 to minimize $\bar{F}(a)$ with $\frac{\sigma}{f^2+1}$ added to its Hessian $\bar{F}''(a)$, equation (10) is exactly the linear system to be solved. Therefore, if $a_0 = A_0 f + B_0$, then for all k,

$$\begin{aligned} a_{k+1} &= a_k + \alpha_k \Big(f \cdot (\delta_k)_1 + (\delta_k)_2\Big) \\ &= \Big(A_k + \alpha_k (\delta_k)_1\Big) f + \Big(B_k + \alpha_k (\delta_k)_2\Big). \end{aligned} \tag{11}$$

Since $\bar{F}(a)$ is strictly convex from $\bar{F}''(a) > 0$, similar techniques in Case 1 can be used to prove that $\bar{F}(a)$ attains an optimal solution. Therefore, from Theorem 2, the sequence $\{a_k\}_{k=0}^{\infty}$ globally converges. Then, from $(\delta_k)_1 = f \cdot (\delta_k)_2$ and (11),

$$\lim_{k\to\infty} a_k = (A_0 f + B_0) + (f^2 + 1) \sum_{k=0}^{\infty} \alpha_k (\delta_k)_2$$

exists. Therefore, $\lim_{k\to\infty} B_k = B_0 + \sum_{k=0}^{\infty} \alpha_k (\delta_k)_2$ exists, and so does $\lim_{k\to\infty} A_k$. In addition, they form an optimal solution of minimizing $F(A, B)$.

C. Pseudo Code of Algorithm 1

We recommend using double precision for the algorithm.

```
Input parameters:
  deci = array of SVM decision values
  label = array of booleans: is the example labeled +1?
  prior1 = number of positive examples
  prior0 = number of negative examples
Outputs:
  A, B = parameters of sigmoid

//Parameter setting
maxiter=100     //Maximum number of iterations
minstep=1e-10   //Minimum step taken in line search
sigma=1e-12     //Set to any value > 0
//Construct initial values: target support in array t,
//                          initial function value in fval
hiTarget=(prior1+1.0)/(prior1+2.0), loTarget=1/(prior0+2.0)
len=prior1+prior0 // Total number of data
for i = 1 to len {
  if (label[i] > 0)
    t[i]=hiTarget
  else
    t[i]=loTarget
}

A=0.0, B=log((prior0+1.0)/(prior1+1.0)), fval=0.0
for i = 1 to len {
  fApB=deci[i]*A+B
  if (fApB >= 0)
```

```
    fval += t[i]*fApB+log(1+exp(-fApB))
  else
    fval += (t[i]-1)*fApB+log(1+exp(fApB))
}
for it = 1 to maxiter {
  //Update Gradient and Hessian (use H’ = H + sigma I)
  h11=h22=sigma, h21=g1=g2=0.0
  for i = 1 to len {
    fApB=deci[i]*A+B
    if (fApB >= 0)
      p=exp(-fApB)/(1.0+exp(-fApB)), q=1.0/(1.0+exp(-fApB))
    else
      p=1.0/(1.0+exp(fApB)), q=exp(fApB)/(1.0+exp(fApB))
    d2=p*q
    h11 += deci[i]*deci[i]*d2, h22 += d2, h21 += deci[i]*d2
    d1=t[i]-p
    g1 += deci[i]*d1, g2 += d1
  }
  if (abs(g1)<1e-5 && abs(g2)<1e-5) //Stopping criteria
    break
  //Compute modified Newton directions
  det=h11*h22-h21*h21
  dA=-(h22*g1-h21*g2)/det, dB=-(-h21*g1+h11*g2)/det
  gd=g1*dA+g2*dB
  stepsize=1
  while (stepsize >= minstep){ //Line search
    newA=A+stepsize*dA, newB=B+stepsize*dB, newf=0.0
    for i = 1 to len {
      fApB=deci[i]*newA+newB
      if (fApB >= 0)
        newf += t[i]*fApB+log(1+exp(-fApB))
      else
        newf += (t[i]-1)*fApB+log(1+exp(fApB))
    }
    if (newf<fval+0.0001*stepsize*gd){
      A=newA, B=newB, fval=newf
      break //Sufficient decrease satisfied
    }
    else
      stepsize /= 2.0
  }
  if (stepsize < minstep){
    print ’Line search fails’
    break
  }
}
if (it >= maxiter)
  print 'Reaching maximum iterations'

return [A,B]
```