A Fast Trust-Region Newton Method for Softmax Logistic Regression

Nayyar A. Zaidi, Geoffrey I. Webb *

Abstract

With the emergence of big data, there has been a growing interest in optimization routines that lead to faster convergence of Logistic Regression (LR). Among many optimization methods such as Gradient Descent, Quasi-Newton, Conjugate Gradient, etc., the Trust-region based truncated Newton method (TRON) algorithm has been shown to converge the fastest. The TRON algorithm also forms an important component of the highly efficient and widely used `liblinear` package. It has been shown that the WANBIA-C trick of scaling with the log of the naive Bayes conditional probabilities can greatly accelerate the convergence of LR trained using (first-order) Gradient Descent and (approximate second-order) Quasi-Newton optimization. In this work we study the applicability of the WANBIA-C trick to TRON. We first devise a TRON algorithm optimizing the softmax objective function and then demonstrate that WANBIA-C style preconditioning can be beneficial for TRON, leading to an extremely fast (batch) LR algorithm. Second, we present a comparative analysis of one-vs-all LR and softmax LR in terms of the 0-1 Loss, Bias, Variance, RMSE, Log-Loss, Training and Classification time, and show that softmax LR leads to significantly better RMSE and Log-Loss. We evaluate our proposed approach on 51 benchmark datasets.

1 Introduction

Logistic Regression (LR) is a well-developed and extensively studied technique in Statistics and a state-of-the-art classifier in Machine Learning [10]. Its mechanisms also apply to several advanced methods including Artificial Neural Networks and Generalized Linear Models [13, 9]. It optimizes (minimizes) the negative of the conditional log-likelihood (Negative Log-Likelihood – NLL) which, for binary (where class is encoded as 0 and 1) classification problems, is defined as:

$$\text{NLL}(\beta) = -\sum_{l=1}^{N} y \log \text{P}(y=1|\mathbf{x}) + (1-y)\log(1-\text{P}(y=1|\mathbf{x})),$$

where $\text{P}(y=1|\mathbf{x}) = \frac{\exp(\boldsymbol{\beta}^T\mathbf{x})}{1+\exp(\boldsymbol{\beta}^T\mathbf{x})}$, which leads to:

$$\text{NLL}(\beta) \;=\; -\sum_{l=1}^{N} y\boldsymbol{\beta}^T\mathbf{x} - \log(1+\exp(\boldsymbol{\beta}^T\mathbf{x})).$$

In practice, one should always regularize to avoid overfitting on smaller quantities of data[1] and, therefore, following objective function is minimized instead:

$$\text{NLL}(\beta) \;=\; -\sum_{l=1}^{N}\Big(y\boldsymbol{\beta}^T\mathbf{x} - \log(1+\exp(\boldsymbol{\beta}^T\mathbf{x}))\Big) + \frac{\lambda}{2}\boldsymbol{\beta}^T\boldsymbol{\beta}, \tag{1.1}$$

where λ is the regularization parameter and the vector $\boldsymbol{\beta}$ is the parameter to be learned. The dimension (p) of the β vector is: $1 + n_n + \sum_i^{n_c}(\mathcal{X}_i - 1)$, where n_n and n_c is the number of numeric and categorical attributes respectively and $\mathcal{X}_i$ is the cardinality of categorical attribute i, 1 is added to account for the intercept term.

For handling multiple classes in LR, there are two main options:

- Option 1: Train multiple one-versus-all or one-versus-one LR classifiers. Then apply all trained classifiers to predict at the classification time, and choose the classifier (and hence the class) with the highest probability. We will focus only on one-versus-all in this work.
- Option 2: Optimize directly the softmax objective function that is:

$$\text{NLL}(\beta) = -\sum_{l=1}^{N}\left(\boldsymbol{\beta}_y^T\mathbf{x} - \log(\sum_{c=1}^{C}\exp(\boldsymbol{\beta}_c^T\mathbf{x}))\right) + \frac{\lambda}{2}\boldsymbol{\beta}^T\boldsymbol{\beta}, \tag{1.2}$$

  where the dimension (p) of the β vector is: $(C-1)(1+n_n+\sum_i^{n_c}(\mathcal{X}_i-1))$.

In most cases, the two options lead to similar accuracy (i.e., 0-1 Loss) and usually the choice is left to the user. However, there are two important distinctions that cannot be ignored:

- Option 1 does not produce well calibrated class probabilities. Therefore, it will result in worst performance than Option 2 in terms of log loss, root-mean-square-error (RMSE) and similar measures

*Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia.

[1] Note, regularization is not exclusive to smaller quantities of data. It may be needed for large datasets. Suppose, if the data is linearly separable, then the solution to equation 1.1 is obtained when $|\beta| \to \inf$, leading to a linear threshold unit that may not generalize well.

of calibration. Several works have explored different ways to better predict class probabilities from binary classifiers [15, 8, 1].

- The training time and classification timings (computational efficiency) of the two options can be significantly different. For example, with Option 1, we are solving multiple small problems separately (dimension of $\boldsymbol{\beta}$ vector is p), whereas, with Option 2, it is just one big problem ($\boldsymbol{\beta}$ is of dimension $(C-1)p$). It is not clear whether or not training multiple subsets of a problem will lead to faster convergence and hence, better training time. To our current knowledge, there are no systematic studies of the comparison of these two choices (at least for LR) in terms of the convergence, training and classification time. We address these issues in this paper.

There is no closed-form solution for optimizing Equations 1.1 or 1.2 and one has to resort to iterative optimization algorithms. Iterative means, that the procedure generates a sequence $\{\boldsymbol{\beta}^k\}_{k=1}^{\infty}$ converging to the optimal solution of Equations 1.1 and 1.2. At every iteration, $\boldsymbol{\beta}^{k+1} = \boldsymbol{\beta}^k + \mathbf{s}^k$, where $\mathbf{s}^k$ is the search direction vector. The following equation plays the pivotal role as it holds the key to obtaining $\mathbf{s}^k$ by solving a system of linear equations:

$$\nabla^2 f(\boldsymbol{\beta}^k)\mathbf{s}^k = -\nabla f(\boldsymbol{\beta}^k), \tag{1.3}$$

where f is the objective function that we are optimizing (i.e., NLL). There are two very important issues that must be addressed when solving for search direction vector using Equation 1.3 [12]. First, computing and storing the Hessian can be very computational intensive and memory inefficient, especially on large datasets with many number of features. Second, the solution obtained using Equation 1.3, does not guarantee any convergence. Let us discuss these two issues.

The computational issues associated with Hessian can be addressed in (at least) three following ways:

- Consider $\nabla^2 f(\boldsymbol{\beta}^k)$ to be an identity matrix – in this case, $\mathbf{s}^k = -\nabla f(\boldsymbol{\beta}^k)$. This leads to a family of algorithms known as first-order methods such as Gradient Descent, Coordinate Descent, etc.

- Do not compute $\nabla^2 f(\boldsymbol{\beta}^k)$ directly, but approximate it from the information present in $\nabla f(\boldsymbol{\beta}^k)$ instead. This property is useful for large scale LR where we cannot store the Hessian matrix. This leads to approximate second-order methods known as quasi-Newton algorithms, for example, L-BFGS which, is considered to be the most efficient algorithm (de-facto standard) for training LR.

- Third, use standard 'direct algorithms' for solving a system of linear equations such as Gaussian elimination to solve for $\mathbf{s}^k$, or any one of the iterative algorithms such as conjugate gradient. For training LR on large datasets, generally iterative methods are preferable over direct methods, as the former requires computing the whole Hessian matrix. Optimization method now has two layers of iterations. An outer layer of iteration to update $\boldsymbol{\beta}^k$, and an inner layer of iterations to find Newton direction $\mathbf{s}^k$. In practice, one can only use an approximate Newton direction in early stages of the outer iterations. This method is known as the 'Truncated Newton method' [11].

It should be noted that these methods differ across many aspects of computational efficiency including speed-of-convergence, cost-per-iteration and iterations-to-convergence. For example, Coordinate Descent updates one component of $\boldsymbol{\beta}$ at every iteration, so the cost-per-iteration is very low, but iterations-to-convergence will be very high. On the other hand, Newton methods, will have high cost-per-iteration, but very low number of iterations-to-convergence. With ever-growing quantities of data, there is a growing interest in algorithms that lead to faster convergence of LR. It has been shown recently that one can speed-up the convergence of first-order and approximate second-order methods for LR by scaling with the log of the naive Bayes conditional probabilities [16, 18]. However, for second-order methods, it is not clear if such pre-conditioning will be effective or not. We will address this issue in this paper.

Let us discuss the second issue, that is the convergence of optimization when using Equation 1.3 to determine the search direction. One can address this problem by adjusting the length of the Newton direction. Two techniques can be used – line-search and trust-region. Line search methods are standard in optimization research. We can modify Equation 1.3 as $\nabla^2 f(\boldsymbol{\beta}^k)\mathbf{s}^k = -\eta^k \nabla f(\boldsymbol{\beta}^k)$, where η^k is known as the step-size. Standard line searches obtain an optimal step-size as a solution to the following sub-optimization problem: $\eta^k = \text{argmin}_\eta f(\boldsymbol{\beta}^k + \eta \mathbf{s}^k)$. Trust-region methods, unlike line search, are relatively new in optimization research. Trust-region methods first find a region around the current solution – in this region, a quadratic (or linear) model is used to approximate the objective function. The step size is determined based on the goodness of fit of the approximate model. If a significant decrease in the objective function is achieved with a forward step, the approximated model is a good representative of the original objective function and vice-versa. The size of the (trust) region is specified as a spherical area of size Δ_k. The convergence of the algorithm is guaranteed by

controlling the size of the region which (in each iteration) is proportional to the reduction in the value of objective function in the previous iteration. An update of the trust-region and the parameter $\boldsymbol{\beta}$ can be controlled by ρ_k – which is the ratio of actual reduction and predicted reduction of the approximated model, and is defined as:

$$\rho_k = \frac{f(\boldsymbol{\beta}^k + \mathbf{s}^k) - f(\boldsymbol{\beta}^k)}{q_k(0) - q_k(\mathbf{s}^k)}, \tag{1.4}$$

where $q_k(\mathbf{s}^k)$ is the approximation of the function and will be discussed shortly. The parameters are updated as:

$$\boldsymbol{\beta}^{k+1} = \begin{cases} \boldsymbol{\beta}^k + \mathbf{s}^k & \text{if } \rho_k > \eta_0 \\ \boldsymbol{\beta}^k & \text{if } \rho_k \leq \eta_0 \end{cases} \tag{1.5}$$

whereas, the trust-region is adjusted as:

$$\begin{aligned} \Delta_{k+1} \in \left[\sigma_1 \min\{||\mathbf{s}^k||, \Delta_k\}, \sigma_2 \Delta_k\right] \quad &\text{if} \quad \rho_k \leq \eta_1, \\ \Delta_{k+1} \in \left[\sigma_1 \Delta_k, \sigma_3 \Delta_k\right] \quad &\text{if} \quad \rho_k \in (\eta_1, \eta_2) \\ \Delta_{k+1} \in \left[\Delta_k, \sigma_3 \Delta_k\right] \quad &\text{if} \quad \rho_k \geq \eta_2 \end{aligned} \tag{1.6}$$

The Trust-region Newton method (TRON) approximates the objective function using the following quadratic model:

$$q_k(\mathbf{s}) = \nabla f(\boldsymbol{\beta}^K)^T \mathbf{s} + \frac{1}{2} \mathbf{s}^T \nabla^2 f(\boldsymbol{\beta}^k) \mathbf{s}, \tag{1.7}$$

such that $||\mathbf{s}|| \leq \Delta_k$. Algorithm 1 from [4] is a pseudo-code of TRON. Note, it is easy to find the solution

Algorithm 1 Trust Region Newton Method

1: **procedure** TRON($\left\{y^{(i)}, \mathbf{x}^{(i)}\right\}_{l=1}^{N}$)
2: $\boldsymbol{\beta} \leftarrow 0$
3: **for** k = 0, 1, . . . , MaxIter **do**
4: If $\nabla f(\boldsymbol{\beta}^k) = 0$, Exit
5:
6: Find an approximate solution to Trust-Region problem: min $q_k(\mathbf{s})$, such that $||\mathbf{s}|| \leq \Delta_k$
7:
8: Compute ρ_k using Equation 1.4
9: Update $\boldsymbol{\beta}^k$ using Equation 1.5
10: Obtain Δ_{k+1} using Equation 1.6
11: **end for**
12: **end procedure**

to the trust-region problem: min $q_k(\mathbf{s})$, provided the Hessian – $\nabla^2 f(\boldsymbol{\beta}^k)$, is positive-definite. However, the constraint $||\mathbf{s}|| \leq \Delta_k$ prohibits us from using conjugate-gradient algorithm in a straight-forward way. A modified conjugate-gradient based on the Steihaug method can be used.

One of the first applications of the Truncated Newton method to solve binary LR problem was addressed in [7]. A TRON algorithm for large-bound constrained optimization problems was proposed in [3]. Later, in [4], a TRON algorithm was applied to binary LR problem. The resulting TRON algorithm forms the basis of an extremely effective and widely used `liblinear` package [14]. One limitation of most existing applications of TRON to LR is that LR is constrained to be binary (and hence a much simpler optimization problem). In this work, we study the application of TRON to multi-class LR (optimizing the softmax objective function). The main contributions of this work are as follows:

- We presents a TRON algorithm optimizing the softmax objective function.
- We show that WANBIA-C pre-conditioning can be effective for the second-order method TRON, leading to an extremely fast LR algorithm.
- We show that optimizing a softmax objective function leads to better RMSE, log-loss and classification time than standard binary one-vs-all classification .
- We provide a comprehensive software library for fast and effective binary and softmax LR – `fastLR`.

The rest of this paper is organized as follows: we discuss binary and softmax LR in Section 2. WANBIA-C preconditioning for LR is discussed in Section 3. We present pre-conditioned softmax TRON in Section 4. The empirical analysis is conducted in Section 5. We conclude in Section 6 with pointers to future work.

2 Binary vs. Softmax LR

As discussed in Section 1, a binary LR minimizes the NLL as its objective function in the form of Equation 1.1. For simplicity, in the remainder of this paper, we will remove the regularization term from the objective function. Therefore, one can write the gradient as:

$$\frac{\partial \text{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_i} = \sum_{l=1}^{N} (y - \text{P}(y = 1|\mathbf{x})) x_i.$$

Note, that as we are looping over the entire data (N), each y and $\mathbf{x}$ is indexed by l. More precisely, the RHS of the above equation should be: $\sum_{l=1}^{N} (y^l - \text{P}(y = 1|\mathbf{x}^l)) x_i^l$. We omit the superscripts for simplicity. In matrix notation the (p-dimensional) gradient vector is: $\nabla \text{NLL}(\boldsymbol{\beta}) = \mathbf{X}^T(\mathbf{y} - \mathbf{p})$, where $\mathbf{y}$ and $\mathbf{p}$ are N dimensional vectors of class labels and class posterior-probabilities respectively. $\mathbf{X}$ is a matrix of dimension

$N \times p$. The Hessian can be written as:

$$\frac{\partial \text{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_i \partial \boldsymbol{\beta}_j} = -\sum_{l=1}^{N}(1 - \text{P}(y=1|\mathbf{x}))\text{P}(y=1|\mathbf{x})x_i x_j.$$

The Hessian matrix (of dimension $p \times p$) can be written as: $\nabla^2\text{NLL}(\boldsymbol{\beta}) = -\mathbf{X}^T\mathbf{W}\mathbf{X}$. Here, $\mathbf{W}$ is an $N \times N$ dimensional diagonal matrix with $\text{P}(y=1|\mathbf{x}^l)(1-\text{P}(y=1|\mathbf{x}^l))$ at its l-th diagonal position.

Now, let us focus on LR optimizing softmax objective function (Equation 1.2). We can write the gradient as:

$$\frac{\partial \text{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y',i}} = \sum_{l=1}^{N}(\mathbf{1}_{y=y'} - \text{P}(y'|\mathbf{x}))x_i,$$

where $\mathbf{1}_{a=b}$ is the indicator function that is equal to 1 if $a = b$ and zero otherwise. Again, precisely, the RHS of the above equation should be written as: $\sum_{l=1}^{N}(\mathbf{1}_{y^l=y'} - \text{P}(y'|\mathbf{x}^l))x_i^l$. The gradient vector will be of dimension $p(C-1)$ and can written as: $\tilde{\mathbf{X}}^T(\tilde{\mathbf{y}} - \tilde{\mathbf{p}})$. Here, $\tilde{\mathbf{y}}$ and $\tilde{\mathbf{p}}$ (each of dimension $N(C-1)$) are concatenation vectors of $C-1$ N-sized indicator vectors and take the forms:

$$\tilde{\mathbf{y}} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_{C-1} \end{pmatrix}, \text{ where } \mathbf{y}_k = \begin{pmatrix} \mathbf{1}_{y^1=k} \\ \mathbf{1}_{y^2=k} \\ \vdots \\ \mathbf{1}_{y^N=k} \end{pmatrix}$$

and

$$\tilde{\mathbf{p}} = \begin{pmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \vdots \\ \mathbf{p}_{C-1} \end{pmatrix}, \text{ where } \mathbf{p}_k = \begin{pmatrix} \text{P}(k|\mathbf{x}^1) \\ \text{P}(k|\mathbf{x}^2) \\ \vdots \\ \text{P}(k|\mathbf{x}^N) \end{pmatrix}.$$

Similarly, $\tilde{\mathbf{X}}$ is a matrix of dimensions $N(C-1) \times p(C-1)$ and takes the form:

$$\tilde{\mathbf{X}} = \begin{pmatrix} \mathbf{X} & 0 & \cdots & 0 \\ 0 & \mathbf{X} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{X} \end{pmatrix}.$$

The elements of the Hessian matrix can be computed as:

$$\frac{\partial^2 \text{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y',i} \partial \boldsymbol{\beta}_{y'',j}} = -\sum_{l=1}^{N}(\mathbf{1}_{y'=y''} - \text{P}(y'|\mathbf{x}))\text{P}(y''|\mathbf{x})x_i x_j.$$

The Hessian is of dimension $p(C-1) \times p(C-1)$ and in matrix notation can be written as: $-\tilde{\mathbf{X}}^T\tilde{\mathbf{W}}\tilde{\mathbf{X}}$. Here, $\tilde{\mathbf{W}}$ is a $N(C-1) \times N(C-1)$ matrix and takes the form:

$$\tilde{\mathbf{W}} = \begin{pmatrix} \mathbf{W}_{11} & \mathbf{W}_{12} & \cdots & \mathbf{W}_{1(C-1)} \\ \mathbf{W}_{21} & \mathbf{W}_{22} & \cdots & \mathbf{W}_{2(C-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{W}_{(C-1)1} & \mathbf{W}_{(C-1)2} & \cdots & \mathbf{W}_{(C-1)(C-1)} \end{pmatrix}.$$

Note, $\mathbf{W}_{c_1c_2}$ is an $N \times N$ diagonal matrix whose elements when $c_1 = c_2$ are:

$$\text{P}(c_1|\mathbf{x})(1 - \text{P}(c_1|\mathbf{x})), \tag{2.8}$$

and when $c_1 \neq c_2$ are:

$$-\text{P}(c_1|\mathbf{x})(\text{P}(c_2|\mathbf{x})). \tag{2.9}$$

2.1 On Efficient Implementation An important operation in TRON is the multiplication of the Hessian matrix with the vector characterizing the search direction, that is:

$$\begin{aligned} -\nabla^2\text{NLL}(\boldsymbol{\beta})\mathbf{s} &= (\mathbf{X}^T\mathbf{W}\mathbf{X})\mathbf{s} \\ &= \mathbf{X}^T(\mathbf{W}(\mathbf{X}\mathbf{s}))) \end{aligned}$$

As a result of the above decomposition which was proposed in [4], we do not have to store the entire Hessian matrix – $\nabla^2\text{NLL}(\boldsymbol{\beta})$. All we need is the $\mathbf{W}$ matrix, which along with the data can be used to find the result of the product: $\nabla^2\text{NLL}(\boldsymbol{\beta})\mathbf{s}$. Of course, for the softmax objective function, the factor will be: $\tilde{\mathbf{X}}^T(\tilde{\mathbf{W}}(\tilde{\mathbf{X}}\tilde{\mathbf{s}}))$ and will be slightly more complicated to handle. Note, $\tilde{\mathbf{s}}$ is a vector of size $p(C-1)$. Later, we will express $\tilde{\mathbf{s}}$ as the concatenation of $C-1$ p-sized vector, each denoted as $\mathbf{s}_k$.

Note, for binary classification, $\mathbf{W}$ is a diagonal matrix, one can, therefore, store the diagonal elements in a vector of size N. The elements of this vector can be updated when computing the gradient. This reduces the space complexity from N^2 to N.

For the softmax objective function, rather than allocating memory for $\tilde{\mathbf{W}}$, one can store memory of size $(C-1) \times (C-1) \times N$ instead[2]. We name this data structure $\mathbf{D}$. When looping over the entire data to update the gradient vector, one can update matrix $\mathbf{D}$ based on Algorithm 2.

Similarly, there is no need to replicate the entire data matrix $\mathbf{X}$, $(C-1)$ times to obtain $\tilde{\mathbf{X}}$ as it will be grossly inefficient. In the following, we provide a simple algorithm that computes the multiplication of Hessian matrix with the step vector. The details are given in Algorithm 3. The algorithm will be called from TRON algorithm every time the product of Hessian matrix and the search direction vector is required. The input is the matrix $\mathbf{D}$, data matrix $\mathbf{X}$ and search direction vector $\tilde{\mathbf{s}}$. The first step is the multiplication of $\tilde{\mathbf{X}}$ with $\tilde{\mathbf{s}}$. There is no need for $\tilde{\mathbf{X}}$. Instead, one can multiply data point x^l in $\mathbf{X}$ with all $(C-1)$ $\mathbf{s}_k$ vectors – and store the outputs in the right locations in vector A, which is a vector of size $N(C-1)$. Note, the function `offset` returns the

[2]Note, for binary classes, this results in the vector of size N.

Algorithm 2 UpdateD

procedure UPDATE$\mathbf{D}(x^{(i)})$

  for $c_1 = 0, 1, \ldots, C-1$ **do**
    for $c_2 = 0, 1, \ldots, C-1$ **do**
      if $c_1 = c_2$ **then**
        $\mathbf{D}[i][c_1][c_2] \leftarrow$ Equation 2.8
      else
        $\mathbf{D}[i][c_1][c_2] \leftarrow$ Equation 2.9
      end if
    end for
  end for

end procedure

Algorithm 3 MultiplyHessianMatrixAndStepVector

procedure HS$(\mathbf{D}, \mathbf{s}, \{x^l, y^l\}_{l=1}^N)$

  $A \leftarrow 0$ ▷ Initialize vector of size N(C-1)
  $B \leftarrow 0$ ▷ Initialize vector of size N(C-1)
  $H \leftarrow 0$ ▷ Initialize vector of size p(C-1)

  ▷ Step 1: Compute $\tilde{\mathbf{X}}\mathbf{s}$
  for `each data point` x^l **do**
    for `each class` c **do**
      $A[l + \text{offset(c)}] \leftarrow x^{lT}\mathbf{s}_c$
    end for
  end for

  ▷ Step 2: Compute $\tilde{\mathbf{W}}\tilde{\mathbf{X}}\mathbf{s}$
  for `each data point` x^l **do**
    for `each class` c_1 **do**
      for `each class` c_2 **do**
        $B[l + \text{offset}(c_1)] \leftarrow \mathbf{D}[l][c_1][c_2] \times A[l + \text{offset}(c_2)]$
      end for
    end for
  end for

  ▷ Step 3: Compute $\tilde{\mathbf{X}}\tilde{\mathbf{W}}\tilde{\mathbf{X}}\mathbf{s}$
  for `each data point` x^l **do**
    for `each class` c **do**
      $H \leftarrow x^{lT} B[l + \text{offset}(c)]$
    end for
  end for

  Delete A, B
  return H

end procedure

appropriate location of the output in vector A. The second step is the computation of $\tilde{\mathbf{W}}\tilde{\mathbf{X}}\mathbf{s}$, which involves multiplying matrix $\mathbf{D}$ with vector A. The output of this is again a vector – B of size $N(C-1)$. The final step (Step 3) is the multiplication of the matrix $\tilde{\mathbf{X}}^T$ with vector B. Again, there is no need for $\tilde{\mathbf{X}}$. One can loop over the data matrix $\mathbf{X}$, $C-1$ times and find the right address in the vector B to multiply. The output is the vector H of dimension $p(C-1)$ which is the product of $\nabla^2 \text{NLL}(\boldsymbol{\beta})\tilde{\mathbf{s}}$.

3 WANBIA-C

So far, our analysis of binary and softmax Logistic Regression has been for numeric data. For categorical data, the standard practice is to transform the data through one-hot-encoding and treat the transformed data as numeric. Note, that this can be extremely inefficient in terms of storage and, therefore, one should use sparse instead of dense storage of matrix $\mathbf{X}$. Let us redefine LR for discrete data. We can write class-posterior probabilities – $\text{P}(y|\mathbf{x})$, as:

$$\text{P}_{\text{LR}}(y|\mathbf{x}) = \frac{\exp(\boldsymbol{\beta}_y + \sum_i \boldsymbol{\beta}_{y,i,x_i} \mathbf{1}_{X_i=x_i \wedge Y=y})}{\sum_{c=1}^{C} \exp(\boldsymbol{\beta}_c + \sum_j \boldsymbol{\beta}_{c,j,x_j} \mathbf{1}_{X_j=x_j \wedge Y=c})},$$

which for simplicity (looping over attribute values that are only one) can be reformulated as:

$$\text{P}_{\text{LR}}(y|\mathbf{x}) = \frac{\exp(\boldsymbol{\beta}_y + \sum_i \boldsymbol{\beta}_{y,i,x_i})}{\sum_{c=1}^{C} \exp(\boldsymbol{\beta}_c + \sum_j \boldsymbol{\beta}_{c,j,x_j})}.$$

Note, we have also made the intercept term explicit in this formulation of LR. A naive Bayes classifier, on the other hand, can be written as:

$$\text{P}_{\text{NB}}(y|\mathbf{x}) = \frac{\exp(\log \pi_y + \sum_i \log \theta_{y,i,x_i})}{\sum_{c=1}^{C} \exp(\log \pi_c + \sum_j \log \theta_{c,j,x_j})}.$$

Proposed in [16], WANBIA-C pre-conditions LR with naive Bayes parameters, that is:

$$\text{P}_{\text{WC}}(y|\mathbf{x}) = \frac{\exp(\boldsymbol{\beta}_y \log \pi_y + \sum_i \boldsymbol{\beta}_{y,i,x_i} \log \theta_{y,i,x_i})}{\sum_{c=1}^{C} \exp(\boldsymbol{\beta}_c \log \pi_c + \sum_j \boldsymbol{\beta}_{c,j,x_j} \log \theta_{c,j,x_j})}.$$

It can be seen that $\text{P}_{\text{LR}}(y|\mathbf{x})$ and $\text{P}_{\text{WC}}(y|\mathbf{x})$ are equivalent in terms of the number of parameters that are to be optimized. However, it has been shown that NLL written in terms of $\text{P}_{\text{WC}}(y|\mathbf{x})$ leads to much faster convergence in far fewer iterations. This comes at the cost of storing the log of the naive Bayes probabilities and an extra initial pass through the data to compute these probabilities.

4 Preconditioned Softmax TRON

We presented gradients and Hessian for binary and softmax LR in Section 2. Since, WANBIA-C pre-conditioning is applicable only to discrete data, let us

derive the gradient and the Hessian for discrete data in this section. If NLL is defined in terms of $\mathrm{P_{LR}}(y|\mathbf{x})$, for discrete data, the gradients can be written as:

$$\begin{aligned}\frac{\partial \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y'}} &= \sum_{l=1}^{N}(\mathbf{1}_{y=y'} - \mathrm{P}(y'|\mathbf{x})),\\ \frac{\partial \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y',i',x'_i}} &= \sum_{l=1}^{N}(\mathbf{1}_{y=y'} - \mathrm{P}(y'|\mathbf{x}))\mathbf{1}_{x_i=x'_i}.\end{aligned}$$

Note, we have made explicit distinction between the intercept and non-intercept terms. Due to this separate handling of the intercept, for computing the Hessian, special care is required. In the following, we present three sets of Hessian that is: A) intercept and intercept, B) intercept and non-intercept and C) non-intercept and non-intercept:

$$\begin{aligned}\frac{\partial^2 \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y'}\partial \boldsymbol{\beta}_{y''}} &= -\sum_{l=1}^{N}(\mathbf{1}_{y'=y''} - \mathrm{P}(y'|\mathbf{x}))\mathrm{P}(y''|\mathbf{x}),\\ \frac{\partial^2 \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y',i',x'_i}\partial \boldsymbol{\beta}_{y''}} &= -\sum_{l=1}^{N}(\mathbf{1}_{y'=y''} - \mathrm{P}(y'|\mathbf{x}))\mathrm{P}(y''|\mathbf{x})\mathbf{1}_{x_i=x'_i},\\ \frac{\partial^2 \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y',i',x'_i}\partial \boldsymbol{\beta}_{y'',j',x'_j}} &= -\sum_{l=1}^{N}(\mathbf{1}_{y'=y''} - \mathrm{P}(y'|\mathbf{x}))\mathrm{P}(y''|\mathbf{x})\\ &\quad \mathbf{1}_{x_i=x'_i}\mathbf{1}_{x_j=x'_j}.\end{aligned}$$

If NLL is defined in terms of $\mathrm{P_{WC}}(y|\mathbf{x})$, we can compute the gradients as:

$$\begin{aligned}\frac{\partial \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y'}} &= \sum_{l=1}^{N}(\mathbf{1}_{y=y'} - \mathrm{P}(y'|\mathbf{x}))\log \pi_y,\\ \frac{\partial \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y',i',x'_i}} &= \sum_{l=1}^{N}(\mathbf{1}_{y=y'} - \mathrm{P}(y'|\mathbf{x}))\log \theta_{y,i,x_i}\mathbf{1}_{x_i=x'_i}.\end{aligned}$$

(4.10)

And the Hessian as:

$$\begin{aligned}\frac{\partial^2 \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y'}\partial \boldsymbol{\beta}_{y''}} &= -\sum_{l=1}^{N}(\mathbf{1}_{y'=y''} - \mathrm{P}(y'|\mathbf{x}))\mathrm{P}(y''|\mathbf{x})\\ &\quad \log \pi_{y'} \log \pi_{y''},\\ \frac{\partial^2 \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y',i',x'_i}\partial \boldsymbol{\beta}_{y''}} &= -\sum_{l=1}^{N}(\mathbf{1}_{y'=y''} - \mathrm{P}(y'|\mathbf{x}))\mathrm{P}(y''|\mathbf{x})\\ &\quad \log \pi_{y''} \log_{y',i',x_{i'}} \mathbf{1}_{x_i=x'_i},\\ \frac{\partial^2 \mathrm{NLL}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}_{y',i',x'_i}\partial \boldsymbol{\beta}_{y'',j',x'_j}} &= -\sum_{l=1}^{N}(\mathbf{1}_{y'=y''} - \mathrm{P}(y'|\mathbf{x}))\mathrm{P}(y''|\mathbf{x})\\ &\quad \log_{y',i',x_{i'}} \log_{y'',j',x_{j'}} \mathbf{1}_{x_i=x'_i}\mathbf{1}_{x_j=x_{j'}}.\end{aligned}$$

(4.11)

The (preconditioned) gradient and Hessian given in Equations 4.10 and 4.11, when called from Algorithm 1 results in a fast trust region based Newton method for softmax LR.

5 Experimental Results

In this section, we compare and analyze the performance of our proposed algorithm and related methods on 51 natural domains from the UCI repository of machine learning datasets [6]. Only datasets with more than two classes are included. The details are given in Table 5. There are 10 datasets with over $100,000$ instances. These datasets are shown in bold font in Table 5. These datasets constitute the *Big* category. When comparing results, we present a separate analysis on this category.

Each algorithm is tested on each dataset using 5 rounds of 2-fold cross validation. We compare seven different metrics, i.e., 0-1 Loss, RMSE, Bias, Variance, Log-Loss, No. of Iterations, Training time and Classification time.

We report Win-Draw-Loss (W-D-L) results when comparing the performance metrics of two models. A two-tail binomial sign test is used to determine the significance of the results. Results are considered significant if $p \leq 0.05$ and shown in bold.

Numeric features are discretized by using the Minimum Description Length (MDL) supervised discretization method [5]. Discretization of numeric attributes is motivated from study in [17]. A missing value is treated as a separate feature value and taken into account exactly like other values.

All the results presented are $\mathrm{L_2}$ regularized (except when comparing the convergence curves). This is because, for a fair convergence comparison, we want all algorithms to converge to the same point in the optimization space. Regularization will lead to convergence at different points.

5.1 Softmax LR vs. One-vs-All LR Let us start by comparing softmax LR (denoted as LR) with one-versus-all (binary) Logistic Regression (denoted as $\mathrm{LR}^{\mathrm{1vsAll}}$). We compare the the two techniques in terms of W-D-L in Table 2. It can be seen that, as expected, LR results in significantly lower RMSE and Log-Loss. Quite, surprisingly, the bias of LR is also significantly lower than $\mathrm{LR}^{\mathrm{1vsAll}}$. Following the insight that low-bias classifiers can be expected to minimize error for large datasets [2], LR results in significantly lower 0-1 Loss on *Big* datasets (8 wins and 2 draws). In contrast, $\mathrm{LR}^{\mathrm{1vsAll}}$ has lower variance than LR which suggests it might be more accurate for sufficiently small datasets. In terms of the training time, it can be seen that $\mathrm{LR}^{\mathrm{1vsAll}}$ is significantly faster than LR on *All* datasets. Classification time, again as expected is slower for $\mathrm{LR}^{\mathrm{1vsAll}}$.

We compare (geometric) average of LR and $\mathrm{LR}^{\mathrm{1vsAll}}$ in terms of 0-1 Loss, RMSE, Training and Classification time in Figure 1.

Domain	Case	Att	Class	Domain	Case	Att	Class	Domain	Case	Att	Class
Kddcup	5209000	41	40	Pioneer	9150	37	57	Vehicle	846	19	4
USCensus1990	2458300	67	4	Satellite	6435	37	6	BalanceScale	625	5	3
Sensor	2219803	5	57	OpticalDigits	5620	49	10	Syncon	600	61	6
Poker-hand	1175067	11	10	PageBlocksClassification	5473	11	5	Dermatology	366	35	6
Covertype	581012	55	7	Wall-following	5456	25	4	PrimaryTumor	339	18	22
Census-Income(KDD)	299285	41	9	Nettalk(Phoneme)	5438	8	52	Audiology	226	70	24
WearableComputing	165633	18	5	Abalone	4177	9	3	New-Thyroid	215	6	3
Localization	164860	7	3	Hypothyroid(Garavan)	3772	30	4	GlassIdentification	214	10	3
Diabetes	101766	46	4	Splice-junctionGeneSequences	3190	62	3	AutoImports	205	26	7
Waveform	100000	21	3	Segment	2310	20	7	WineRecognition	178	14	3
Connect-4Opening	67557	43	3	CarEvaluation	1728	8	4	TeachingAssistantEvaluation	151	6	3
Statlog(Shuttle)	58000	10	7	Volcanoes	1520	4	4	IrisClassification	150	5	3
LetterRecognition	20000	17	26	Yeast	1484	9	10	Lymphography	148	19	4
Nursery	12960	9	5	ContraceptiveMethodChoice	1473	10	3	Zoo	101	17	7
Sign	12546	9	3	LED	1000	8	10	PostoperativePatient	90	9	3
PenDigits	10992	17	10	Vowel	990	14	11	LungCancer	32	57	3
Thyroid	9169	30	20	Annealing	898	39	6	Contact-lenses	24	5	3

Table 1: Details of datasets used.

	LR vs. LR^{1vsAll}		**LR vs. LR^{1vsAll}**	
	All Datasets		*Big* Datasets	
	W-D-L	p	W-D-L	p
Bias	35/7/9	**<0.001**	9/1/0	**0.003**
Variance	15/7/29	**0.048**	1/2/7	**0.039**
0-1 Loss	26/9/16	0.164	8/2/0	**0.007**
RMSE	49/0/2	**<0.001**	10/0/0	**<0.001**
Log-Loss	51/0/0	**<0.001**	10/0/0	**<0.001**
Training Time	17/0/34	**0.017**	3/0/7	0.343
Classification Time	47/3/1	**<0.001**	9/0/1	**0.011**

Table 2: Win-Draw-Loss: LR vs. LR^{1vsAll}. p is two-tail binomial sign test. Results are significant if $p \leq 0.05$.

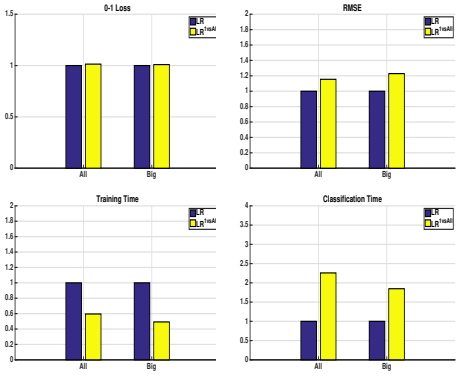

Figure 1: Geometric mean of 0-1 Loss, RMSE, Training time, Classification time performance of LR and LR^{1vsAll} for *All* and *Big* datasets. Results normalized w.r.t LR.

5.2 Softmax LR vs. Pre-conditioned Softmax LR Let us compare LR with preconditioned LR (denoted as LR-WC) in this section. Comparison between the two parameterizations in terms of W-D-L is given in Table 3. We argue, that the two parameterizations have a similar 0-1 Loss, RMSE, bias, variance and Classification timing profile. This can be seen from first five rows of the Table 3, where none of the results are significant. However, in terms of the training time and no. of iterations it takes each algorithm to converge, LR-WC is significantly better than LR.

We compare (geometric) average of LR and LR-WC in terms of 0-1 Loss, RMSE, Training and Classification time in Figure 2.

5.2.1 Analysis of Convergence The variation of the objective function (NLL) with varying iterations for LR and LR-WC on 12 sample datasets is given in Figure 3. It can be seen that, not only does LR-WC converge in fewer iterations, it follows a more desirable convergence path, that is, asysmptoting to its minima much quicker than LR. It can be seen that TRON has close to quadratic convergence as on most datasets, both LR and LR-WC converge on average in 10 iterations. Remember, Algorithm 1 has two levels of iterations. The W-D-L iteration results in Table 3 are those of outer iterations. However, we found, that LR-WC has also fewer inner iterations as well. We have not presented these results due to space constraints. But it can be seen from Training time results in Figure 2, where LR-WC is roughly twice as fast on average.

5.2.2 Comparison with Quasi-Newton (L-BFGS) Quasi Newton methods are the de-facto standard for optimizing LR. In Figure 4, we show a comparison of TRON and L-BFGS convergence on two sample datasets. LR trained with L-BFGS is denoted as LR-QN and preconditioned LR trained with L-BFGS is denoted LR-WC-QN. It can be seen that LR trained with TRON converges in an order of magnitude fewer iterations. Note, like TRON (which has an inner level of iterations), L-BFGS has internal function evaluations for line searches, and, therefore, the number of passes over the data are substantially greater then are shown in Figure 4. Our goal, here is to compare the convergence profiles of the two

	LR vs. LR-WC		LR vs. LR-WC	
	All Datasets		*Big* Datasets	
	W-D-L	p	W-D-L	p
Bias	12/24/15	0.701	2/5/3	1.000
Variance	20/15/16	0.617	6/4/0	**0.03**
0-1 Loss	17/14/20	0.742	4/4/2	0.453
RMSE	24/10/17	0.348	5/5/0	0.06
Classification Time	18/23/10	0.184	5/0/5	1.000
Training Time	5/0/46	**<0.001**	1/0/9	**0.011**
Iterations	2/3/46	**<0.001**	1/1/8	**0.025**

Table 3: Win-Draw-Loss: LR vs. LR-WC. p is two-tail binomial sign test. Results are significant if $p \leq 0.05$.

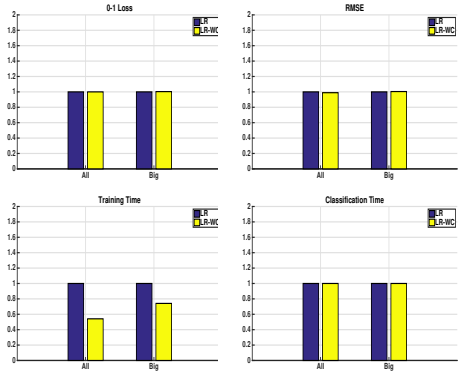

Figure 2: Geometric mean of 0-1 Loss, RMSE, Training and Classification time performance of LR and LR-WC for *All* and *Big* datasets. Results normalized w.r.t LR.

optimization routines. In practice, we found TRON to have significantly faster training time than L-BFGS.

6 Conclusion and Future Work

In this paper, we presented a fast trust-region based Newton method for softmax LR, which is based on pre-conditioning using the WANBIA-C trick. We addressed the issue of efficiently computing the Hessian of LR for the softmax objective function. We also computed softmax LR with one-versus-all binary LR ($\text{LR}^{\text{1vsAll}}$). Here are some of our key findings:

- We showed that, contrary to the common conception that softmax LR and $\text{LR}^{\text{1vsAll}}$ have equivalent accuracy – softmax LR has significantly lower bias and higher variance. Therefore, we recommend, that softmax LR should be used for big training sets.
- Softmax LR has lower RMSE and log-loss and should be preferred if the calibration of class-probabilities is important.
- Training time is faster for $\text{LR}^{\text{1vsAll}}$ than softmax LR but classification time is slower.
- WANBIA-C pre-conditioning is equally effective for second-order optimization methods.
- Preconditioned TRON leads to an extremely fast LR algorithm.

Future works lies in the direction of developing the software library. Recently it has been shown that for big datasets, one can train an LR by building all higher-order features [19]. One can speed-up the convergence of this higher-order LR by integrating the softmax TRON algorithm proposed in this work. Implementing other loss functions such as Hinge and Mean-Square-Error are also left as future work.

7 Code

The details of the software library `fastLR` is given in Appendix A. The library along with running instructions can be downloaded from Github: `https://github.com/nayyarzaidi/fastLR.git`.

8 Acknowledgments

This research has been supported by the Australian Research Council (ARC) under grant DP140100087, and by the Asian Office of Aerospace Research and Development, Air Force Office of Scientific Research under contract FA2386-15-1-4007. The authors would like to thank Wray Buntine for helpful discussions during the course of this paper.

A `fastLR` – LR Library

The library can handle both numeric and categorical attributes. There is no need to do a one-hot-encoding for categorical attributes, as the LR model built can directly handle the data types.

One can execute the code in the library by issuing the command: `java -cp /fastLR.jar fastLR.BVDcrossvalx -t /dataset.arff -i 2 -x 2 -W LRClassifier -- -S "overparamLR" -O "Tron"`. This does two rounds of two-fold cross validation on dataset name dataset.arff and runs LR with solver Tron. This will learn weights for all C classes. For learning weights for only $C-1$ classes, use the following flag: `-S "vanilla"`. For other optimization use: `"GD","CG","QN"` for Gradient Descent, Conjugate Gradient and L-BFGS respectively. For pre-conditioning, use `-S "overparamWC"` or `-S "vanillaWC"`. For one-versus-all binary classification, use `-W oneversusAllLRClassifier`. The default parameterization is `-S "vanilla"` which learns just one set of weights, but one can also use `-S "overparam"`.

References

[1] Allwein, E.L., Schapire, R.E., Singer, Y.: Reducing multiclass to binary: A unifying approach for margin

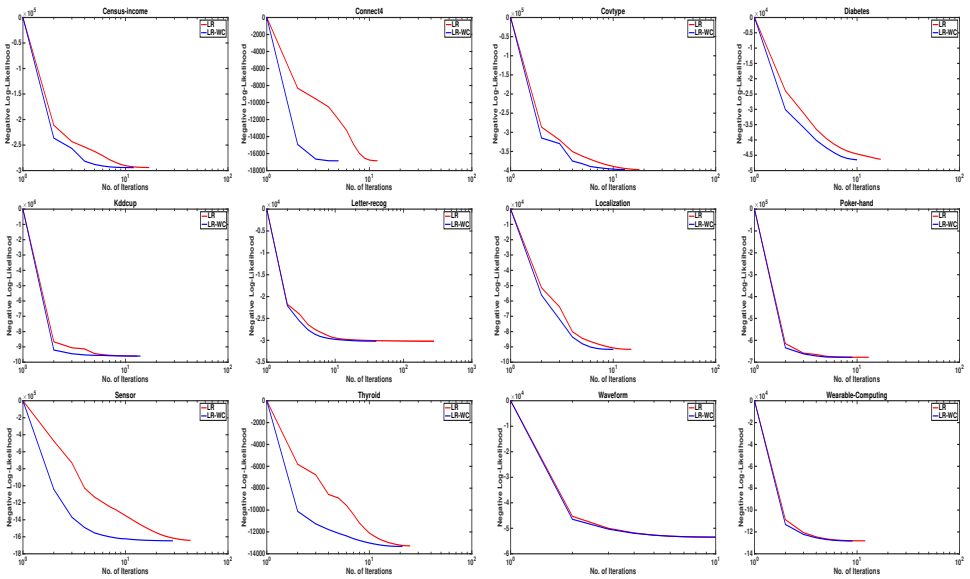

Figure 3: Comparison of the rate of convergence of LR and LR-WC on several datasets. The X-axis (No. of iterations) is on log scale.

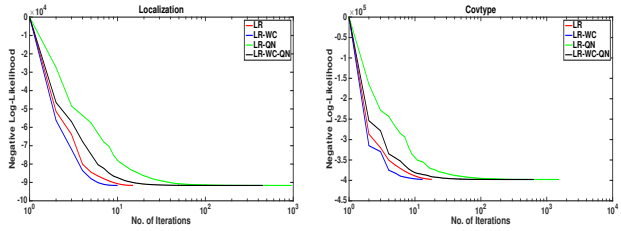

Figure 4: Comparison of the rate of convergence of LR, LR-WC, LR-QN and LR-WC-QN on two sample datasets. The X-axis (No. of iterations) is on log scale.

classifiers. Journal of Machine Learning Research **1**, 113–141 (2000)

[2] Brain, D., Webb, G.I.: The need for low bias algorithms in classification learning from small data sets. In: PKDD, pp. 62–73 (2002)

[3] Chih-Jen, L., More, J.J.: Newton method for large bound-constrained optimization problems. SIAM Journal of Optimization **9**, 1100–1127 (1999)

[4] Chih-Jen, L., Weng, R.C., Keerthi, S.S.: Trust region newton method for large-scale logistic regression. Journal of Machine Learning Research **9**, 627–650 (2008)

[5] Fayyad, U.M., Irani, K.B.: On the handling of continuous-valued attributes in decision tree generation. Machine Learning **8**(1), 87–102 (1992)

[6] Frank, A., Asuncion, A.: UCI machine learning repository (2010). URL `http://archive.ics.uci.edu/ml`

[7] Komarek, P., Moore, A.W.: Making logistic regression a core data mining tool. Tech. rep., Robotics Institute, CMU (2005)

[8] Kong, B.E., Dietterich, T.G.: Probability estimation via error-correcting output coding. In: Int. Conf on Artificial Intelligence and Soft Computing (2001)

[9] McCullagh, P., Nelder, J.: Generalized Linear Models, second edn. Boca Raton: Chapman and Hall/CRC (1989)

[10] Murphy, K.: Machine learning: a probabilistic perspective. MIT Press (2012)

[11] Nash, S.G.: A survey of truncated newton methods. Journal of Computational and Applied Mathematics **124**, 45–59 (2000)

[12] Nocedal, J., Wright, S.: Numerical optimization. Springer Science & Business Media (2006)

[13] Ripley, B.D.: Pattern Recognition and Neural Networks. Cambridge University Press (1996)

[14] Rong-En, F., Kai-Wei, C., Cho-Jui, H., Xiang-Rui, W., Chih-Jen, L.: LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research **9**, 1871–1874 (2008)

[15] Zadrozny, B.: Reducing multiclass to binary by coupling probability estimates. In: NIPS (2001)

[16] Zaidi, N.A., Carman, M.J., Cerquides, J., Webb, G.I.: Naive-bayes inspired effective pre-conditioners for speeding-up logistic regression. In: IEEE International Conference on Data Mining (2014)

[17] Zaidi, N.A., Du, Y., Webb, G.I.: On the effectiveness of discretizing quantitative attributes in linear classifiers. arXiv:1701.07114 (2017)

[18] Zaidi, N.A., Petitjean, F., Webb, G.I.: Preconditioning an artificial neural network using naive bayes. In: Advances in Knowledge Discovery and Data Mining, pp. 341–353 (2016)

[19] Zaidi, N.A., Webb, G.I., Carman, M.J., Petitjean, F., Cerquides, J.: ALR^n: Accelerating higher-order logistic regression. Machine Learning **104**, 151–194 (2016)