Implementation of Logistic Regression with Truncated IRLS

Paul Komarek
School of Computer Science
Carnegie Mellon University
komarek@cmu.edu

Andrew Moore
School of Computer Science
Carnegie Mellon University
awm@cs.cmu.edu

July 19, 2005

Abstract

This document describes the use of the logistic regression (LR) with truncated iteratively re-weighted least squares algorithm (T-IRLS). This is a fast and stable version of LR which is simple to implement and easy to run. It works on sparse binary data, or arbitrary dense data. The dataset outputs must be binary. The software computes the probability that a given data point belongs to either of the binary classes.

Contents

1 Introduction

We assume that the software has been built, and that the executables are named `train`, `predict`, and `kfold`. All commands in this document are indented as in the example shown below.

```
train in dataset save filename args
```

Names of arguments are written as *argname*, while actual argument values are written as `argval`. The argument list is keyword-based, meaning that the order of the arguments is unimportant. Some arguments consist of a keyword and a value, with the value occurring immediately after the keyword. A "dataset" is a file containing data one wishes to analyze, or the filename of such a file. More information about the supported dataset file formats and naming conventions may be found in Section 3.

The `train` executable runs our logistic regression (LR) algorithms on the specified data and stores the resulting LR parameter estimates in the specified file. Additional arguments control the logistic regression and conjugate gradient (CG) algorithms.

The `predict` executable uses the LR predictions, as saved by the `train` executable, to predict the probability that the dataset rows belong to the *positive* class. Which class is positive depends on both dataset naming conventions and the dataset contents. The resulting predictions can be saved to a file, and are scored using the area-under-curve (AUC) metric [1]. See Section 2.2 for more information on AUC scoring.

The `kfold`executable performs a k-fold cross-validation on the specified data, printing some statistics when finished. This program can write the ROC curve [1] into a file, as well as reporting AUC scores. More information about the `train`, `predict`, and `kfold` executables can be found in Section 2, and information on dataset naming and contents may be found in Section 3.

All of these executables print the total time, in real seconds, elapsed while the program was running. This time includes file load and save times, and other overhead. It is only displayed if the `verbosity` is 1 or greater. See Section 2.1 for more information about setting the verbosity. The `kfold` program additionally tracks and reports times that do not include this overhead. Also note that these programs unbuffer `stdout` to make file redirections update more quickly.

This software can be built with zlib [2] support. If this is done, then dense numerical data files can be loaded from compressed files. See Section 3 for details. Furthermore, output files from this software will be compressed if their name ends in ".gz" and zlib support is enabled.

2 Training LR And Making Predictions

Logistic regression (LR) reads a data file, makes some computations, and outputs weights for each attribute in the data file. These weights may be used in combination with the logit function to predict the probability that a new, previously unseen data point belongs to one of two classes. We will not further describe LR here. The reader is advised to see [3] for an introduction to LR and its uses. Details on our algorithms can be found in various places, and the most centralized of these is [4].

We provide three executables at this time, `train`, `predict`, and `kfold`. Both are described below. In the future, we will add an executable to make k-fold cross-validations easy. The Auton software page [5] should have a file conversion utility named *convert*. This utility may be useful for converting various data files into the formats expected by this LR software. See Section 3 for more information on data file formats.

2.1 `train`

The `train` executable performs the LR computations on a dataset, outputting the LR parameter estimates. It has two optional arguments, and passes all unknown arguments to our LR algorithm. The simplest and most common invocation is shown below.

```
train in dataset save filename
```

This command passes no arguments to the LR algorithm, accepting the default settings. This is almost always adequate, and is certainly the best way to start. LR parameters are discussed in Section 4.

To see more information about how your command line was parsed, use the `arghelp` keyword.

```
train in dataset save filename arghelp
```

Note that boolean keywords that require no value might not appear in the `arghelp` output.

If you would like to see more or less screen output while the LR computations are running, use the `verbosity` keyword along with an integer value. A verbosity of -1 is quiet, 0 is default, and 2 is somewhat informative. A value of 5 or higher might be useful when debugging.

2.2 `predict`

The `predict` executable uses the computed LR parameter estimates, produced and saved by the `train` executable described above, and uses them to compute predictions on a dataset. The dataset used for predictions must have the same shape as that used for computing the LR parameters. This means that the prediction dataset should have the same number of attributes as the input dataset, in the same order, and should have contain an output value for each data point. The output values are not used during prediction, but are used when scoring predictions. It is allowable to set all outputs to arbitrary values in the prediction dataset, but the computed score will be wrong. This might be useful if you want to save the predictions and you do not care about the score.

The score reported by `predict` is the area-under-curve (AUC) metric. This is simply the normalized area under a certain curve, in particular the receiver operating characteristic (ROC) curve. AUC and ROC curves are described in several books and papers, for example in [1, 3, 6, 7].

Very briefly, to draw the ROC curve we first sort the data points in decreasing order according to the output value we predict for that data point. Thus the data point we feel is most likely to belong to the positive class comes first in the new order. After sorting, we put our pen down at the origin of our graph. If our top-ranked data point is truly positive, we move "up" one unit. If it was negative, we move "right" one unit. We repeat this for each data point in order. At the end of this procedure, we have moved one unit for each data point. If the dataset contains p positive points and w negative points, then we our final position will always be the x-y coordinate (w, p). A good algorithm will climb steeply and eventually level off. Random guessing will travel roughly in a straight line from $(0, 0)$ to (w, p).

Note that ROC curves do not require thresholding of predictions, unlike accuracy and precision scores, and hence is insensitive to possibly arbitrary cut-offs between positive and negative points. ROC curves do measure the ability of the algorithm to discriminate between positive and negative classes, in that the curve improves when positive points are ordered before negative points. A perfect ROC curve climbs vertically from $(0, 0)$ to $(0, p)$, indicating that no negative points were scored higher than any positive points. At this point, there are no more positive points, and the graph must travel horizontally to (w, p). The AUC metric is simply the ratio of the area under an ROC curve to the area under a perfect ROC curve. Thus an AUC of 1.0 is perfect, and random guessing should result in an AUC of 0.5.

Like the `train` executable, the `predict` executable accepts the `arghelp` and `verbosity` arguments. See Section 2.1 for more information. The `predict` program also accepts an optional keyword `pout` to indicate that the user would like to save the predictions into a file. The `pout` keyword must be followed by a file name. If `rout` is specified along with a filename, then the ROC curve will be saved as a series of x-y pairs suitable for use in plotting programs such as Gnuplot [8].

A typical `predict` invocation is

```
predict in dataset load filename
```

This will read in the estimated LR parameters from *filename*, and use them to make predictions on the data points in *dataset*. A score will be computed and printed to the screen. If you would like to save the prediction for each data point in *dataset*, as well as the ROC curve, add the `pout` and `rout` keywords as shown below.

```
predict in dataset load filename pout filename rout filename
```

After `predict` exits, each line in file *prediction-filename* will contain a probability that the corresponding data point in *dataset* belongs to the *positive* class. Which of the two classes in *dataset* corresponds to the positive class is determined by elements of the dataset name, and its contents. See Section 3 for more information.

2.3 `kfold`

The `kfold` executable performs a k-fold cross-validation on the specified data, printing some statistics when finished. This program can write the ROC curve into a file, as well as reporting AUC scores. See Section 2.2 for more information about ROC curves and AUC scores. In these k-fold cross-validation experiments, the input dataset is partitioned into *k* parts. Then *k* individual training and prediction experiments are run, training on all but one partition and predicting on the remaining partition. If *k* is equal to the number of rows in the dataset, then this procedure is sometimes called "leave-one-out cross-validation". The result of all *k* folds is a prediction for each data point, but those predictions are made from training the LR model on different subsets of the data.

The `kfold` executable scores performance using the AUC metric, described above. The AUC score is computed for each fold of the cross-validation, and the mean and standard deviation is reported. It is possible to view the AUC score for each fold. The average time for each fold is reported, along with the standard deviation, and it is possible to view the per-fold times. The per-fold times are in real seconds, and include the training and prediction times but exclude other overhead such as dataset loading.

Like the `train` executable, the `kfold` executable accepts the `arghelp` and `verbosity` arguments. See Section 2.1 for more information. In order to view per-fold AUC scores and times, `verbosity` must be 1 or greater.

The `kfold` program also accepts optional keyword `pout` and `fout`, to to save the predictions and fold assignments for each data point in the input dataset. Unlike with the `predict` executable, the predictions stored with the `pout` option are not generated by the same model. They are only useful in conjunction with the output produced by the `fout` option.

The fold assignment, as stored by the `fout` option, is the fold in which each data point was *not* used for training. Thus a file with values 7,2,3 on the first three lines indicates that the first data point was held out from training in the seventh fold, the second data point was held out in the second, and the third data point was held out in the third fold.

One more keyword is supported, `rout`. If `rout` is specified along with a filename, then the ROC curve will be saved as a series of x-y pairs suitable for use in plotting programs such as Gnuplot [8]. All three optional keywords take filenames as arguments.

A typical `kfold` invocation is

```
kfold in dataset folds 10
```

This will report the summary statistics for the scores and times of a 10-fold cross-validation on the dataset. To see per-fold scores and times, run

```
kfold in dataset folds 10 verbosity 1
```

To save the predictions, fold assignments, and the ROC curve coordinates, run

```
kfold in dataset folds 10 pout filename fout filename rout filename
```

3 Data

The `train` and `predict` programs are able to read two dataset formats. One of these is suitable for sparse binary data, and the other is suitable for dense numerical data. Though T-IRLS can be extended to other data, we have not written the software to do this yet. Both dense and sparse data files must contain a binary-valued outputs. If zlib [2] support is enabled, then dense numerical data can be loaded from many types of compressed files.

This software is fairly strict about data file formatting. To convert other files in the formats supported by this software, please see the *convert* utility on the Auton software page [5].

3.0.1 Dense Datasets

All rows in a dense dataset file are comma-separated lists of real values. Lines beginning with '#' are treated as comments and ignored. All non-comment lines must have the same number of commas, and each represents one data point. The output must be stored in the last column (i.e. attribute), and that column must contain only the characters 0 and 1. Such datasets are referred to as *csv* files. For example, *csv* file might start with the following lines:

```
# Table-tennis fatality data in csv format
# Copyright Pad L. Grip, 2005
#
# age weight skill experience caused-fatality?
#
27, 72.3, 92.5, 87.2, 0
29, 67.9, 72.0, 80.1, 0
24, 92.0, 99.3, 90.0, 1
...
```

3.1 Sparse Datasets

We use our *spardat* format for sparse datasets. This format is similar to that used by SVM^{light}[9]. It is whitespace delimited and all attributes must be binary. The output value is the first token on each line, and can be any real value. However, the output attribute will be thresholded to a binary value when loaded. The manner in which the thresholding occurs is described in Section 3.2. All remaining tokens on the line are interpreted as indices of nonzero attributes, where the first attribute has index 0. Attribute numbering begins at zero, and lines beginning with "#" are ignored. An example *spardat* file is shown below.

```
# Shrink-wrap accident data in spardat format
# Copyright Reynold S. Rap, 2005
#
# This dataset contains 2537 attributes corresponding to presence
# or absence of various kitchen items at the time of the shrink-wrap
# accident.  The output indicates the severity of the accident, with
# score of 75.0 or larger indicating a *severe* accident.
#
#
29.2 157 158 192 1001
72.5 157 158 199 200 201
42.3 0 1 7
92.1 0 1 158 517 1001 1900 2535 2536
...
```

One additional dataset format modification is understood. If the attribute indices in a *spardat* formatted file are appended with ":1", the ":1" will be ignored and the data will be loaded as expected. This provides compatibility with SVM^{light}. For example:

```
# Shrink-wrap accident data in spardat format
# Copyright Reynold S. Rap, 2005
#
...
#
29.2 157:1 158:1 192:1 1001:1
```

```
72.5 157:1 158:1 199:1 200:1 201:1
42.3 0:1 1:1 7:1
92.1 0:1 1:1 158:1 517:1 1001:1 1900:1 2535:1 2536:1
...
```

3.2 Dataset naming

Both dataset file formats, *csv* and *spardat*, require information beyond the file name when they are specified on the command-line. This information is specified though additional keywords and possibly extra characters appended to the filename.

When loading *csv* files, the file must have ".csv" for the suffix. For example,

```
train in a-or-d.csv save filename
```

If the data file name does not end in ".csv", our software will try to load the data file in *spardat* format. The single exception to this rule is for files that end in ".csv.gz", which will be transparently decompressed if zlib support was enabled when this software was built. See below for caveats about *spardat* file names.

The *spardat* format requires a threshold level and direction, where the direction indicates whether higher or lower values represent the positive output class. Thus the threshold specifier consists of a colon, a threshold value, and a plus or minus sign. A plus sign indicates that output values greater than or equal to the threshold value are considered positive. A minus sign indicates that values less than or equal to the threshold are considered positive. As an example, suppose `a-or-d.txt` is a *spardat* format dataset, and we wish to consider all outputs with value greater than 0.5 as positive outputs. The command line below specifies this using the notation just described.

```
train in a-or-d.txt:0.5+ save filename
```

4 Logistic Regression Parameters

To be very brief, logistic regression (LR) computes β for which the model values μ_i best approximate the dataset outputs y_i under the model

$$\mu_i = \frac{\exp(\beta_0 + \beta_1 x_{i1} + \cdots \beta_M x_{iM})}{1 + \exp(\beta_0 + \beta_1 x_{i1} + \cdots \beta_M x_{iM})}$$

where x_i is one dataset row. Please see [3, 10, 1, 6, 7, 4] for details about logistic regression. This implementation uses iterative re-weighted least squares (IRLS) [1, 6, 7, 4] to maximize the LR log-likelihood

$$\sum_n i = 1^R y_i \ln(u_i) + (1 - y_i) \ln(1 - u_i)$$

where R is the number of rows in the dataset. For logistic regression, IRLS is equivalent to Newton-Raphson [10]. To improve the speed of IRLS, this implementation uses conjugate gradient (CG) [11, 12, 13, 14, 15, 7] as an approximate linear solver [6, 7, 4]. This solver is applied to the linear regression

$$\left(X^T W X\right) \beta_{\text{new}} = X^T W z \tag{1}$$

where $W = \text{diag}(\mu_i(1 - \mu_i))$ and $z = X\beta_{\text{old}} + W^{-1}(y - \mu)$. The current estimate of β is scored using the likelihood ratio $-2log(L_{\text{sat}}/L_{\text{current}})$, where L_{sat} is the likelihood of of a saturated model with R parameters and L_{current} is the likelihood of the current model. This ratio is called the "deviance", and the IRLS iterations are terminated when the relative difference of the deviance between iterations is sufficiently small. Other termination measures can be added, such as a maximum number of iterations.

For most applications, no LR parameters need to be used. However, the parameters in the table below are available. Full descriptions are provided further below. Most of the parameter names correspond directly to parameters described in our research papers [6, 7, 4]. Note in particular that the `cgbinit` parameter described in our papers is set automatically when `cgdeveps` is used for the CG termination criterion, and is not controllable by the user.

Keyword	Arg Type	Arg Vals	Default	
cgbinit	None, set automatically when cgdeveps>0	{true,false}	true	
cgdecay	float	$[1.0, \infty)$	1000	
cgdeveps	float	[1e-10, ∞)	0.005	
cgeps	float	[1e-10, ∞)	0.000	(i.e. default is disabled)
cgmax	int	$0, \ldots, \infty$	200	
cgwindow	int	$0, \ldots, \infty$	3	
lreps	float	[1e-10, ∞)	0.05	
lrmax	int	$0, \ldots, \infty$	30	
rrlambda	float	$[0, \infty)$	10.0	

- cgbinit: If cgdeveps is specified, then cgbinit is enabled automatically. Each IRLS iteration will start the conjugate gradient solver at the current value of β. With cgeps instead of cgdeveps, the conjugate gradient solver is started at the zero vector. See [7] for more details.
- cgdecay *float*: If the deviance ever exceeds cgdecay times the best deviance previously found, conjugate gradient iterations are terminated. As usual, the β corresponding to the best observed deviance is returned to the IRLS iteration.
- cgdeveps *float*: This parameter applies to conjugate gradient iterations while approximating the solution β_{new} in Equation 1. If cgdeveps is positive, conjugate gradient is terminated if the relative difference of the deviance falls below cgdeveps. The recommended value of cgdeveps is 0.005, and this is the default. Non-positive values indicate that cgeps should be used instead. When cgdeveps>0.0, then cgbinit is enabled. Decrease cgdeveps for a tighter fit. Increase cgdeveps to save time or if numerical issues appear. Exactly one of cgdeveps and cgeps can be positive.
- cgeps *float*: This parameter applies to conjugate gradient iterations while approximating the solution β_{new} in Equation 1. If cgeps is positive, conjugate gradient is terminated when the conjugate gradient residual drops below cgeps times the residual before iterations. The recommended value of cgeps is 0.001. Non-positive values indicate that cgdeveps should be used instead. When cgeps>0.0, then cgbinit is disabled. Decrease cgeps for a tighter fit. Increase cgeps to save time or if numerical issues appear. Exactly one of cgeps and cgdeveps can be positive.
- cgmax *int*: This is the upper bound on the number of conjugate gradient iterations allowed. The final value of β is returned.
- cgwindow *int*: If the previous cgwindow conjugate gradient iterations have not produced a β with smaller deviance than the best deviance previously found, then conjugate gradient iterations are terminated. As usual, the β corresponding to the best observed deviance is returned to the IRLS iteration.
- lreps *float*: IRLS iterations are terminated If the relative difference of the deviance between IRLS iterations is less than lreps.
- lrmax *float*: This parameter sets an upper bound on the number of IRLS iterations. If performing a train/test experiment, you may want to decrease this value to 5, 2, or 1 to prevent over-fitting of the training data.
- rrlambda *float*: This is the ridge regression parameter λ. When using ridge regression, the large-coefficient penalty $\lambda\beta^T\beta$ is added to the sum-of-squares error for the linear regression represent by Equation 1.

References

[1] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Verlag, 2001.

[2] zlib, 2005. http://www.zlib.net.

[3] David W. Hosmer and Stanley Lemeshow. *Applied Logistic Regression*. Wiley-Interscience, 2 edition, 2000.

[4] Paul Komarek and Andrew Moore. The Authors' Logistic Regression Papers, 2005. `http://komarix.org/ac/papers`.

[5] The Auton Lab Software Page, 2005. `http://www.autonlab.org/autonweb/software.jsp`.

[6] Paul Komarek and Andrew Moore. Fast Robust Logistic Regression for Large Sparse Datasets with Binary Outputs. In *Artificial Intelligence and Statistics*, 2003.

[7] Paul Komarek. Logistic Regression for Data Mining and High-Dimensional Classification. Technical Report TR-O4-34, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2004.

[8] Gnuplot, 2005. `http://www.gnuplot.info`.

[9] SVM^{light}, 2002. `http://svmlight.joachims.org`.

[10] P. McCullagh and J. A. Nelder. *Generalized Linear Models*, volume 37 of *Monographs on Statistics and Applied Probability*. Chapman & Hall, 2 edition, 1989.

[11] Stephen G. Nash and Ariela Sofer. *Linear and Nonlinear Programming*. McGraw-Hill, 1996.

[12] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, volume 17 of *Frontiers in Applied Mathematics*. SIAM, 1997.

[13] Allen McIntosh. *Fitting Linear Models: An Application of Conjugate Gradient Algorithms*, volume 10 of *Lecture Notes in Statistics*. Springer-Verlag, New York, 1982.

[14] Thomas P. Minka. Algorithms for maximum-likelihood logistic regression. Technical Report Stats 758, Carnegie Mellon University, October 2001.

[15] Jonathan Richard Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical Report CS-94-125, Carnegie Mellon University, Pittsburgh, 1994.