Discrete Choice Analysis Tools 2.0

for GAUSS™ Mathematical and Statistical System

GAUSS

Expanding Probabilities

The **Discrete Choice Analysis Tools 2.0 Module** provides an adaptable environment for estimating and evaluating discrete choice models. Model specificity is accommodated with tools for incorporating parameter bounds, linear or nonlinear constraints, default or user specified starting values, and user specified Gradient and Hessian procedures.

Supported models include:

- Adjacent categories multinomial logit
- Logit and probit regression
- Conditional logit
- Multinomial logit
- Nested binomial regression
- Ordered logit and probit regression
- Poisson regression
- Stereotype multinomial logit
- Logistic regression
- Support vector regression

Information in this document is subject to change without notice and does not represent a commitment on the part of Aptech Systems, Inc. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose other than the purchaser's personal use without the written permission of Aptech Systems, Inc.

© Copyright 1997-2016

Aptech Systems, Inc.
Chandler, AZ USA
All Rights Reserved Worldwide

11/21/2016

Table of Contents

1 Installation

Discrete Choice Analysis Tools 2.0 requires **GAUSS 14** or later. In **GAUSS 14+** there is an Applications Installation Wizard available to install your application.

Go to **Tools -> Install Applications** and follow the prompts to install **GAUSS Applications** from the CD or downloaded `.zip` file.

Difference Between the Linux/Mac and Windows Versions:

If the functions can be controlled during execution by entering keystrokes from the keyboard, it may be necessary to press ENTER after the keystroke in the Linux/Mac version.

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

2 Getting Started

GAUSS version 14+ is required to use these routines.

The **Discrete Choice Analysis Tools 2.0** version number is stored in a global variable:

_dc_ver	3×1 matrix, the first element contains the major version number, the second element the minor version number, and the third element the revision number.

If you contact technical support, you may be asked for the version of your **Discrete Choice** license.

2.1 README Files

If there is a `README.dc` file, it contains any last minute information on the **Discrete Choice Analysis Tools 2.0** procedures. Please read it before using them.

2.2 Setup

In order to use the procedures in **Discrete Choice** or **DC** module, the **DC** library must be active. This is done by including 'dc' in the library statement at the top of your program or command file:

```
library dc;
```

This enables **GAUSS** to find the **DC** procedures.

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

3 Estimation

The **DC** application includes pre-programmed modeling tools that allow you to move quickly from discrete and categorical data to results. Most procedures use a consistent function call requiring one input: a *`dcControl`* structure. The *`dcControl`* structure is used to control optimization, estimation parameters, and data input. In most cases, the elements optimization and estimation parameters in this structure will NOT need to be changed from the defaults.

Model setup in **Discrete Choice Analysis Tools 2.0** follows four easy steps:

1. Declare a *`dcControl`* structure
2. Model specific data setup and descriptions
3. Declare *`dcOut`* structure
4. Call modeling procedure

The section provides general information regarding the model setup, while specific examples for supported models are provided in Section 5.

3.1 Supported Models

Model selection and estimation in **Discrete Choice Analysis Tools 2.0**, with the exception of the **`logisticRegress`** procedure, requires a procedure call of the general form:

```
dcOut = modelName(dcCt);
```

Discrete Choice Analysis Tools 2.0 includes tools for modeling binary choice, in which the individual faces one choice, multinomial choice, in which the individual chooses between multiple options, ordered choice models, in which the individual demonstrates preferential strength from an outcome, and event count models. Each model supported in **Discrete Choice Analysis Tools 2.0** is discussed in this manual and demonstrated in an example.

General Overview:

The supported binary models include:

- Binary Logit
- Binary Probit

Supported Multinomial Models:

- Conditional Logit
  - Explanatory variables include attributes of choice alternatives and characteristics of decision makers.
- Nested (hierarchical) logit
  - Considers "trees" made up of model levels of decisions.
  - Useful for cases of sequential decision making.
- Ordered logit
  - Dependent variable has more than one category.
  - Categories have meaningful sequential order.
- Adjacent category logit
  - Ordinal logit model.
  - Coefficients from adjacent categories are assumed equal.
- Stereotype logit
  - Restricts coefficients to vary by scale factors.
  - Coefficients are linearly related.

Supported Event Count Models

- Negative binomial regression
- Poisson regression

3.2 Declaring dcControl Structure

The *dcControl* structure is the user tool for controlling optimization, estimation parameters and data input. The first step in setting up any **Discrete Choice Analysis Tools 2.0** model is to declare the *dcControl* structure using:

```
//Declare dcControl struct
struct dcControl dcCt;
```

Once declared, the *dcControl* structure must be initialized using the **dcControlCreate** procedure

```
//Initialize dcControl struct
dcCt = dcControlCreate();
```

3.3 Data Setup

3.3.1 Data Setup Using a GAUSS Data Set

Creating GAUSS Data Sets

GAUSS data sets are the preferred method of storing data contained in a single matrix for use within GAUSS. **GAUSS** data sets are arranged as matrices and are organized in terms of rows and columns. Columns within the data set may be assigned variable names, which are stored for later reference. Any data matrix may be saved as a **GAUSS** data set using **`saved`** procedure. For example, consider the matrix *`myopia`*, containing an indicator for myopia, along with independent data measuring the hours individuals spend playing sports, reading, on the computer, and studying, along with indicators for myopia in the individuals mother and fathers. The **`saved`** command is used to save this matrix, along with variable names, in a **GAUSS** data set named *`myopia`*:

```
//Load data matrix
loadm myopia;

//Extract desired columns from imported data
dataMat = myopia[.,3 11:15 17 18];

//Create variable list
vnames = "MYOPIC,SPORTHR,READHR,COMPHR,STUDYHR,TVHR,MOMMY,DADMY";
vnames = strsplit(vnames, ",");

//Name dataset
datasetName = "myopia";

//Save dataset
y = saved(dataMat, datasetName, vnames);
```

Loading Data Sets For DC Analysis

All **DC** procedures are designed to read data from **GAUSS** data sets. Specifying the usage of a GAUSS data set in **DC** procedures is done using the **`dcSetDataSet`** procedure. This procedure requires two inputs, a pointer to a *`dcControl`* structure and a string indicating the dataset name. For example, if you wish to use *`myopia`*, the data set created in the previous example:

```
//Declare dcControl struct
struct dcControl dcCt;

//Initialize dcControl struct
dcCt = dcControlCreate();
```

```
//Set data set
dcSetDataSet(&dcCt,"myopia");
```

Data setup and description

If a **GAUSS** data set is specified for usage, model variables must be set using the **DC** application variable setting utility tools. These procedures are easy to implement and make output easier to interpret. The majority of the functions have the general prefix ***dcSet*** and require two inputs: a pointer to a previously declared *dcControl* structure, and a string of comma separated names. As an example, suppose the independent variables stored is the data set being used are *exper*, *educ*, and *white* and the dependent variable is *mode*. The variable names are passed into the **DC** modeling environment using the **dcSetXLabels** procedure and the **dcSetYLabel** procedure, respectively:

```
dcSetXLabels(&dcCt,"exper,educ,white");
dcSetYLabel(&dcCt,"mode");
```

where *&dcCt,* is a pointer to a *dcControl* structure named *dcCt*.

3.3.2 Data Setup Using A Data Matrix

Loading Data Matrices For DC Analysis

All **DC** procedures are designed to read data directly from **GAUSS** matrices. Using data stored in a matrix for **DC** procedures is done using the general ***dcSet*** procedures to set variable specific data. These procedures require two inputs, a pointer to *dcControl* structure and a matrix of data. This implies that prior to using **dcSetDataSet** *dcControl* structure must be declared. For example, consider using the first column of data stored in the matrix *dataMat* as the dependent variable in a model and the second, third, and fourth column as the dependent variables. This is done using **dcSetYVar** and the **dcSetXVars** procedures:

```
//Declare dcControl struct
struct dcControl dcCt;

//Initialize dcControl struct
dcCt = dcControlCreate();

//Set Y Variable
dcSetYVar(&dcCt,dataMat[.,1]);

//Set X Variables
dcSetXVars(&dcCt,dataMat[.,2:4]);
```

Data setup and description

Note that when using a data matrix data labels are not required and default names will be generated if none are specified. However, data labels can be easily added using the ***dcSet*** procedures. As an example, suppose the independent variables specified above are labeled *exper*, *educ*, and *white* and the dependent variable, is labeled *mode*. The variable labels are passed into the **DC** modeling environment using the **dcSetXLabels** procedure and the **dcSetYLabel** procedure, respectively:

```
dcSetXLabels(&dcCt,"exper,educ,white");
dcSetYLabel(&dcCt,"mode");
```

where *&dcCt* is a pointer to a *dcControl* structure named dcCt.

3.4 Declaring A dcOut Structure

All **DC** models send output to a *dcOut* structure. Prior to calling the modeling procedure, the *dcOut* structure must be declared:

```
struct dcOut out;
```

A given instance of the *dcOut* structure names *out* contains the following elements:

out.par

instance of *PV* structure containing estimates.

b0	1 $L \times 1$ matrix, constants in regression.
b	2 $L \times K$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

	if model does not contain a parameter, `pvUnpack` returns a scalar missing value with error code = 99.
`out.vc`	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
`out.yDist`	$L \times 1$ vector, percentages of dependent variable by category.
`out.xData`	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
`out.marginEffects`	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	$N \times 1$ matrix of predicted (fitted) counts.
`out.resids`	$N \times 1$ matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.

14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

3.5 Calling the Modeling Procedure

The final step to performing modeling using **Discrete Choice Analysis Tools 2.0** is to call a specific modeling procedure. For example, to model a binary logit model:

```
out = binaryLogit(dcCt);
```

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

4 Optimization

A general constrained maximum likelihood estimation problem is:

$$\max_{\theta} L = \sum_{i=1}^{N} \log P\left(Y_i | x_{i;\theta}\right)$$

where N is the number of observations, $P(Y_i|x_i;\theta)$ is the probability of Y_i given x_i, and θ, a vector of parameters subject to linear constraints, nonlinear constraints, and bounds constraints.

The linear constraints are:

$$\begin{aligned} A\theta &= B \\ C\theta &\geq D \end{aligned}$$

The nonlinear constraints are:

$$\begin{aligned} G(\theta) &= 0 \\ H(\theta) &\geq 0 \end{aligned}$$

The bounds constraints are:

$$\theta_l \leq \theta \leq \theta_u$$

$G(\theta)$ and $H(\theta)$ are functions provided by the user and must be differentiable at least once with respect to θ.

Under **sqpSolvemt**, parameters are updated in a series of iterations beginning with starting values provided by the user. Let θ_t be the current parameter values. Successive values are

$$\theta_{t+1} = \theta_t + \rho\delta$$

where δ is a $K\times1$ *direction* vector, and ρ a scalar *step length*.

sqpSolvemt finds values for the parameters in θ such that L is maximized (the actual procedure is to minimize -L).

Numerous user controllable variables affect the **sqpSolvemt**, optimization. These are put into a *dcControl* structure instance. Suppose this instance has the name *dc1*, i.e.

```
struct dcControl cont;
cont = dcControlCreate();
```

The following are the members of the *dcControl* structure relevant to the management of the optimization:

cont.A	$M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where p is a vector of the parameters.
cont.B	$M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B`} where p is a vector of the parameters.
cont.C	$M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where p is a vector of the parameters.
cont.D	$M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where p is a vector of the parameters.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = ., i.e., no equality procedure.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = ., i.e., no inequality procedure.
cont.bounds	1×2 or $K \times 2$ matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = -1e256 1e256.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied sqpSolvemt exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, If zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.

cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

4.1 Constraints

The *c1.A*, *dc1.B*, *dc1.C*, *dc1.D*, *dc1.eqProc*, *dc1.inEqProc*, and *dc1.bounds* matrix structure members control constraints in the **Discrete Choice Analysis Tools** procedures. Each row in one of these matrices is associated with a single constraint.

For computational convenience, nonlinear equality constraints and nonlinear inequality constraints are divided into five types: linear equality, linear inequality, nonlinear equality, nonlinear inequality, and bounds constraints.

4.1.1 Linear Equality Constraints

Linear constraints are of the form:

$$A\theta = B$$

where A is an $m_1 \times k$ matrix of known constants, B an $m_1 \times k$vector of known constants, and θ the vector of parameters.

To specify linear equality constraints, assign the *A* and *B* matrices to the *dc1.A* and *dc1.B* structure members. To constrain the first of four parameters to equal the third,

```
dc1.A = { 1 0 -1 0 };
dc1.B = { 0 };
```

4.1.2 Linear Inequality Constraints

Linear constraints are of the form:

$$C\theta \geq D$$

where C is an $m_2 \times k$ matrix of known constants, D an $m_2 \times k$ vector of known constants, and θ the vector of parameters.

To specify linear inequality constraints, assign the *C* and *D* matrices to the *dc1.C* and *dc1.D* structure members. To constrain the first of four parameters to be greater than the third, and the second plus the fourth to be greater than 10:

```
dc1.C = { 1 0 -1 0,
          0 1  0 1 };
```

```
dc1.D = { 0,
          10 };
```

4.1.3 Nonlinear Equality

Nonlinear equality constraints are of the form:

$$G\theta = 0$$

where θ is the vector of parameters and $G(\theta)$ is an arbitrary, user-supplied function.

To specify nonlinear equality constraints, assign the pointer to the user-supplied constraint function to the *dc1.eqProc* member. To constrain the norm of the parameters to equal 1:

```
proc eqp(b);
    retp(b'b - 1);
endp;
dc1.eqProc = &eqp;
```

4.1.4 Nonlinear Inequality

Nonlinear constraints are of the form:

$$H(\theta) \geq 0$$

where θ is the vector of parameters and $H(\theta)$ is an arbitrary, user-supplied function.

To specify nonlinear inequality constraints, assign the pointer to the user-supplied constraint function to the structure member *dc1.inEqProc*. To constrain a covariance matrix to be positive definite, the lower left non-redundant portion of which is stored in elements $r : r + s$ of the parameter vector:

```
proc ineqp(b);
    local v;
    v = xpnd(r[r:r+s]); // r and s defined elsewhere
    retp(minc(eigh(v)) - 1e-5);
endp;
dc1.inEqProc = &ineqp;
```

This constrains the minimum eigenvalue of the covariance matrix to be greater than a small number (1e-5), guaranteeing that the covariance matrix is positive definite.

4.1.5 Bounds

Bounds are a type of linear inequality constraint. For computational convenience they are specified separately from the other inequality constraints.

To specify bounds constraints, enter the lower and upper bounds respectively in the first and second columns of a matrix that has the same number of rows as the parameter vector. Assign this matrix to the structure member *dc1.bounds* Only the first row is necessary if the bounds are the same for all of the parameters. To bound four parameters:

```
dc1.bounds = { -10 10,
               -10  0,
                 1 10,
                 0  1 };
```

To bound all the parameters between -50 and +50:

```
dc1.bounds = { -50 50 };
```

4.1.6 Imposing Constraints in Discrete Choice Models

To impose constraints in **Discrete Choice** models, you will need to know the order of parameters in the parameter vector. The simplest way to do this is to first run the model unconstrained and inspect the parameter vector upon output. For example, run your command file adding a call to **pvGetParNames**:

```
new;
cls;
library dc;

//Load Data
loadm y = gssocc_mat;

//Step One: dcControl structure
//Declare dcControl structure
struct dcControl dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();

//Step Two: Describe data
//Set dependent variable
dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt, "occatt");

//Dependent variable categories
dcSetYCategoryLabels(&dcCt,"Menial,BC,Craft,WC,Pro");

//Independent variables
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"exper,educ,white");
```

```
struct dcOut dcOut1;
dcOut1 = multinomialLogit(dcCt);

print (ftostrC(seqa(1,1,pvLength(dcOut1.par)), "%1.0lf")
        $~ pvGetParNames(dcOut1.par));
```

The above code produces the following output:

```
         1          b0[1,2]
         2          b0[1,3]
         3          b0[1,4]
         4          b0[1,5]
         5           b[1,2]
         6           b[1,3]
         7           b[1,4]
         8           b[1,5]
         9           b[2,2]
        10           b[2,3]
        11           b[2,4]
        12           b[2,5]
        13           b[3,2]
        14           b[3,3]
        15           b[3,4]
        16           b[3,5]
```

Now suppose you want to constrain columns two and three of b to be equal to each other (the first column is the reference column fixed to zeros), the last two columns to be equal to each other (a type of adjacent categories model), i.e., b[1,3] = b[1,2], b[2,3] = b[2,2], etc., and b[1,5] = b[1,4], b[2,5] = b[2,4], etc., and as well, b[1,4] >= b[1,2], b[2,4] >= b[2,2], etc.

To accomplish this we set up the following constraint matrices:

```
//          1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
        c0.A = { 0  0  0  0  1 -1  0  0  0  0  0  0  0  0  0  0,
                 0  0  0  0  0  0  0  0  1 -1  0  0  0  0  0  0,
                 0  0  0  0  0  0  0  0  0  0  0  0  1 -1  0  0,
                 0  0  0  0  0  0  1 -1  0  0  0  0  0  0  0  0,
                 0  0  0  0  0  0  0  0  0  0  1 -1  0  0  0  0,
                 0  0  0  0  0  0  0  0  0  0  0  0  0  0  1 -1 };

        c0.B = { 0,
                 0,
                 0,
                 0,
```

```
                     0,
                     0 };
```

Now suppose we wish to constrain the second column to be equal to the square of the third column, i.e., $b[1,2] = b[1,3]^2, b[2,2] = b[2,3]^2$, etc. For nonlinear constraints we must provide a procedure for computing the constraint. Our command file now looks like this:

```
new;
cls;
library dc;

//Load Data
loadm y = gssocc_mat;

//Step One: dcControl structure
//Declare dcControl structure
struct dcControl dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();

//Step Two: Describe data
//Set dependent variable
dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt, "occatt");

//Dependent variable categories
dcSetYCategoryLabels(&dcCt,"Menial,BC,Craft,WC,Pro");

//Independent variables
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"exper,educ,white");

proc eqp(struct PV par, struct DS d);
    local p,r;
    p = pvGetParVector(par);

    r = zeros(3,1);
    r[1] = p[5] - p[6]^2;
    r[2] = p[9] - p[10]^2;
    r[3] = p[13] - p[14]^2;
    retp(r);
endp;

dcCt.eqProc = &eqp;
```

```
struct dcOut dcOut1;
dcOut1 = multinomialLogit(dcCt);
call printDCOut(dcOut1);
```

Equality constraints are not required to be feasible. Inequality constraints however must be feasible. If you are imposing inequality constraints, start values computed by the procedures may not be feasible and the optimization will fail. In that case you will have to supply feasible start values.

4.2 Direction

Define the likelihood function's gradient and Hessian:

$$\begin{aligned}\Psi(\theta) &= \frac{\partial L}{\partial \theta}\\ \Sigma(\theta) &= \frac{\partial^2 L}{\partial\theta\partial\theta'}\end{aligned}$$

and the Jacobians

$$\begin{aligned}\dot{G}(\theta) &= \frac{\partial G(\theta)}{\partial \theta}\\ \dot{H}(\theta) &= \frac{\partial H(\theta)}{\partial \theta}\end{aligned}$$

For the purposes of this exposition and without loss of generality, assume that the linear constraints and bounds have been incorporated into G and H.

In practice, linear constraints are specified separately from the G and H because their Jacobians are known and easy to compute. The bounds are more easily handled separately from the linear inequality constraints.

The direction, δ, solves the quadratic program

$$\begin{aligned}minimize\ & \tfrac{1}{2}\delta'\Sigma(\theta_t)\delta + \Psi(\theta_t)\,\delta\\ subject\ \ to\ \ & \dot{G}(\theta_t)\,\delta + G(\theta_t)\,\delta\\ & \dot{H}(\theta_t)\,\delta + H(\theta_t)\,\delta\end{aligned}$$

This solution requires that Σ be positive semi-definite.

4.2.1 Line Search Methods

Given a direction vector d, the updated estimate of the parameters is computed

$$\theta_{t+1} = \theta_t + \rho\delta$$

where ρ is a constant, usually called the *step length*, that increases the descent of the function given the direction. The value of the function to be minimized as a function of ρ is

$$m(\theta_t + \rho\delta)$$

Given θ and *d*, this is a function of a single variable ρ. The STEPBT polynomial line fitting/line search method attempts to find a value for ρ that decreases *m*.

STEPBT is an implementation of a similarly named algorithm described in Dennis and Schnabel (1983). It first attempts to fit a quadratic function to $m(\theta_t + \rho\delta)$, computing a ρ that minimizes the quadratic. If that fails it attempts to fit a cubic function. The cubic function is more costly to compute.

If `dc1.randRadius` is greater than zero, a random search is tried if STEPBT fails. The random search uses the radius specified by `dc1.randRadius`.

4.3 Line Search

Define the merit function

$$m(\theta) = L + \mathbf{max} \mid \kappa \mid \sum_j |g_j(\theta)| - \mathbf{max}|\, \lambda \mid \sum_\ell \ \mathbf{min}(0, h_i(\theta))$$

where g_j is the j-th row of *G*, h_ℓ is the ℓ-th row of H, κ is the vector of Lagrangian coefficients of the equality constraints, and λ the Lagrangian coefficients of the inequality constraints.

The line search finds a value of ρ that minimizes or decreases $m\,(\theta_t + \rho\delta)$.

4.4 Managing Optimization

The critical elements in optimization are scaling, the starting point, and the condition of the model. When the data are scaled, the starting point is reasonably close to the solution, and the data and model go together well, the iterations converge quickly and without difficulty.

When the optimization is not proceeding well, it is sometimes useful to examine the function, the gradient Ψ , the direction δ, the Hessian Σ, the parameters θ_t, or the step length ρ, during the iterations.

-

The **sqpSolvemt** procedure calculates the gradient and Hessian numerically, using **gradmt** and **hessmt**. They have the same input arguments as **sqpSolvemt**, a *PV* instance containing the parameters and a *DS* instance containing the data.

4.4.1 Scaling

For best performance, the diagonal elements of the Hessian matrix should be roughly equal. If some diagonal elements contain numbers that are very large and/or very small with respect to the others, **`sqpSolvemt`** has difficulty converging. It is not always obvious how to scale the diagonal elements of the Hessian. One rule-of-thumb is that the data be of roughly the same magnitude.

4.4.2 Condition

The specification of the model may be measured by the condition of the Hessian, the ratio of the Hessian's largest to smallest eigenvalues.

The optimization solution is found by searching for parameter values for which the gradient is zero. It is difficult to determine a parameter's optimal value when the gradient of the function with respect to a parameter is nearly flat. When this occurs, elements of the Hessian associated with the parameter are very small and the inverse of the Hessian contains very large numbers. The search direction gets buried in the large numbers. In this case it is necessary to respecify the model to exclude the parameter.

Poor condition can be caused by bad scaling. It can also be caused by a poor specification of the model or by bad data. A poorly specified model and bad data are two sides of the same coin.

If the problem is highly nonlinear, it is important that data be available to describe the features of the curve described by each of the parameters. For example, one of the parameters of the Weibull function describes the shape of the curve as it approaches the upper asymptote. This parameter is poorly estimated if data are not available for that portion of the curve.

4.4.3 Starting Point

When the model is not particularly well-defined, the starting point can be critical. Try different starting points when the optimization does not seem to be working. A closed form solution may exist for a simpler problem with the same parameters. For example, ordinary least squares estimates may be used for nonlinear least squares problems or nonlinear regressions like probit or logit. There are no general methods for computing starting values. It may also be necessary to attempt the estimation from a variety of starting points.

The starting values for optimization are stored within the *`dcControl`* structure in the member *`cont.startvalues`*. This member is an instance of the *`PV`* structure containing starting values. If these values are not provided by the user, they are automatically computed internally. However, to set the starting values manually, the starting values needed to be "packed" into the PV structure. As an example, consider putting a starting intercept, b_0, and starting coefficients, b, in *`cont.startvalues`*. This is done using the **`pvPackmi`** procedure. This procedure requires five inputs: the PV structure name (*`cont.startValues`*) , the corresponding matrix of starting values, the name of the variable as a string (b_0 or b), a mask matrix indicating which variables to include in the estimated parameter vector, and an index number within the PV structure.

There are a few tips to remember when setting up user defined start values:

1. Intercepts are stored in the b_0 matrix of the *cont.startValues* PV structure. This is the first element in the *cont.startValues* structure and should have dimensions equal to 1 x L, where L equals the number of dependent variable categories.
2. Coefficients are stored in the b matrix of the *cont.startValues* PV structure. This is the second element in the *cont.startValues* structure and should have dimensions equal to K x L, where K equals the number of independent regressors.
3. In each of the above matrices, all members in the column corresponding to the reference category should be set equal to zero.
4. GAUSS must be told to exclude the reference category start values from the parameter vector it estimates. This is done using the *mask* matrix. The *mask* is a matrix that should include only zeros or ones. Elements with zeros will NOT be included in the estimated parameter vector while elements with one will be. For example if I have the matrix:

```
 b = { 0 1 1,
      2 .3 4};

mask = {0 1 1,
        1 0 1};
```

and use

```
struct PV startValues;
startValues = pvPackmi(startValues, b ,"b", mask, 2);
```

GAUSS will pack the elements in b into the PV structure startValues and will assume that the elements {1,1} and {2,2} will be held constant through any estimation at 0 and .3, respectively.

For example, to set user-defined start values for a the intercepts and coefficients in a binary logit model:

```
//Declare control structure

struct dcControl cont;
cont = dcControlCreate();

//Set parameter start values
//Set b0, dimensions must be equal to one
//by the number of Y categories
b0 = {0 1};

//Set b, dimensions must be equal to K
//by the number of Y categories
b = {0 .1};

//Set mask which controls which variables
//go into the parameter vector

//This must be used to
//remove reference category start values
```

```
    //from parameter vector
    mask = {0 1};

    //Pack parameter values
    cont.startValues =  pvPackmi(cont.startValues, b0 ,"b0", mask, 1);
    cont.startValues = pvPackmi(cont.startValues , b ,"b", mask, 2);
```

5 Discrete Choice

The **Discrete Choice Analysis Tools 2.0** estimation uses the **`sqpsolvemt`** procedure, a sequential quadratic programming method that solves general nonlinear programming problems.

All model procedures from version 1.0 have been redesigned and renamed in **Discrete Choice Analysis Tools 2.0**. As an example, the same estimation performed in **`dcNestedLogit`** in version 1.0 is now performed using **`nestedLogit`** in **Discrete Choice Analysis Tools 2.0**. However, **Discrete Choice Analysis Tools 2.0** is completely backwards compatible and all previous procedure calls are still functional.

The discrete choice estimation procedures, with the exception of the **`logisticRegress`** procedure, require one input, a *`dcControl`* structure instance. All output arguments are housed within the *`dcOut`* structure instance and can be printed directly to the input/output screen using the **`dcPrintOut`** procedure.

5.1 Poisson Model

Given independent variables x_i for an observation with count y_i, the Poisson density function is

$$P\left(y_i|x_i\right) = \frac{\exp(-\mu_i)\mu_i^{y_i}}{y_i!}$$

where

$$\mu_i = E\left(y_i|x_i\right) = \exp(x_i\,\beta)$$

is the number of events expected to occur per unit time (or space).

The Poisson regression model log likelihood function is:

$$ln\,L = \sum_{i=1}^{n}\left[-\mu_i + y_i\beta' x_i - ln(y_i!)\right]$$

The Poisson distribution function is

$$F\left(c\right) = P\left(y_i \le c\right) = \sum_{j=0}^{c} P\left(y_i = j|x_i\right)$$

5.1.1 Poisson Over-dispersion

The **printDCOut** procedure shows three tests for over-dispersion when a Poisson model is estimated.

Following Cameron and Trivedi's (1998, p62) notation, let $\omega_i = V[y_i|x_i]$ be the conditional variance of y_i. Two possible variance functions are the NB1 and NB2 functions:

$$NB1 \;:\; \omega_{i=(1+\alpha)\mu_i}$$
$$NB2 \;:\; \omega_{i=\mu_i+\alpha\mu_i^2}$$

Tests of $Ho: \alpha = 0$ in both cases are conducted using axillary regressions. Over-dispersion of the *NB*1 form is indicated by a significant *t* statistic for $\widehat{\alpha}$ in the regression $\frac{(y_i-\widehat{\mu_i})^2-y_i}{\widehat{\mu_i}} = \alpha + u_i$. Over-dispersion of the *NB*2 form is indicated by a significant *t* statistic for $\widehat{\alpha}$ in the regression $\frac{(y_i-\widehat{\mu_i})^2-y_i}{\widehat{\mu_i}} = \alpha\widehat{\mu_i} + u_i$. In both cases u_i is an i.i.d. disturbance term. The **printDCOut** procedure reports the *t* statistics for both cases and their probability values, against a two sided alternative hypothesis.

A Lagrange Multiplier test for over-dispersion is presented by Greene (2000, pp. 885-886). The Poisson model is a restriction on the Negative Binomial model. The *LM* statistic has a $\chi^2(1)$ distribution under the null hypothesis that the mean equals the variance.

$$LM = \frac{(e'e - N\overline{y})^2}{2\widehat{\mu}'\widehat{\mu}}$$

where *e* is an *N*×1 vector of residuals and $\widehat{\mu}$ the *N*×1 vector of fitted values. The **printDCOut** procedure reports this statistic and its probability value.

5.1.2 Example

The included **poissonCount** example uses count data stored in the *greenedata_mat* matrix included with the **DC** examples. The first step to performing analysis is to load the data,

```
new;
cls;
library dc;

//Load Data
y = loadd("greenedata");
```

Once data is loaded, estimation features are specified using the *dcControl* structure. This structure must be declared then initialized using the **dcControlCreate** procedure:

```
//Step One: dcControl structure

//Declare dcControl structure
struct dcControl dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();
```

Prior to estimation, the ***dcSet*** procedures may be used to specify variables. For the Poisson model we begin by describing the dependent count data using **dcSetYVar** and **dcSetYLabel**:

```
//Step Two: Describe data

//Set dependent variable
dcSetYVar(&dcCt, y[.,14]);
dcSetYLabel(&dcCt, "ACC");
```

Similarly, we set the independent variables using the **dcSetXVars** and **dcSetXLabels**:

```
//Independent variables
dcSetXVars(&dcCt,y[.,3:6]~y[.,8:10]~y[.,11]);
dcSetXLabels(&dcCt,"TB,TC,TD,TE,T6569,T7074,T7579,O7579");
```

Finally the time variable is set using **dcSetTimeVar** and **dcSetTimeLabel**:

```
//Name of time variable
dcSetTimeVar(&dcCt, y[.,13]);
dcSetTimeLabel(&dcCt, "Months");
```

Next, the *dcOut* structure is declared:

```
//Step Three: Declare dcOut structure
struct dcOut dcOut1;
```

Finally, calling the **poissonCount** procedure estimates the model and results are reported using the **printDCOut** procedure:

```
//Step Four: Call poissonCount
dcOut1 = poissonCount(dcCt);

//Print Results
call printDCOut(dcOut1);
```

5.2 Negative Binomial Model

The Poisson model assumes that the conditional variance always equal the conditional mean. Consistent but inefficient Poisson model estimates and downward biased standard errors result if this assumption is not true (Gourieroux et al., 1984, Cameron and Trivedi, 1986, p. 31).

The negative binomial regression model lets the conditional variance exceed the conditional mean. Let the conditional mean, μ_i

$$\mu_i = \exp(x_i\beta + \varepsilon_i)$$

where ε is random and uncorrelated with x. Rewrite Negative Binomial Model in terms of the Poisson mean to get

$$\mu_i = \exp(x_i\beta)\exp(\varepsilon_i) = \mu_i \exp(\varepsilon_i) = \mu_i\delta_i$$

Assume that δ_i has a gamma distribution with parameter υ_i (this sets $E(\delta_i) = 1$, identifying the model, and $Var(\delta_i) = 1/\upsilon_i$) and integrate $P(y_i|x_i;\delta_i)$ over the unknown δ_i to get the negative binomial density function:

$$P(y_i|x_i) = \frac{\Gamma(y_i+\upsilon_i)}{y_i!\Gamma(\upsilon_i)}\left(\frac{\upsilon_i}{\upsilon_i+\mu_i}\right)^{\upsilon_i}\left(\frac{\mu_i}{\upsilon_i+\mu_i}\right)^{y_i}$$

with distribution function

$$F(c) = P(y_i \le c) = \sum_{j=0}^{c} P(y_i = j|x_i)$$

The conditional variance is

$$Var(y_i|x_i) = \mu_i\left(1 + \frac{\mu_i}{\upsilon_i}\right) = \exp(x_i\beta)\left(1 + \frac{x_i\beta}{\upsilon_i}\right)$$

which is greater than the conditional variance of the Poisson distribution.

5.3 Truncation and Censoring

This discussion of truncated and censored models closely follows Hayashi (2000) and Long (1997). It assumes that $\{y_t, x_t\}$ is i.i.d.

y_t is truncated if observations above or below given levels are not in the sample. A double truncation rule is that y_t is observable if it is greater than c_t or less than c_u. The density function is

$$\begin{aligned} f(y|y > c_t) and\,(y < c_u) &= \frac{f(y)}{P((y>c_l) and (y<c_u))} \\ &= \frac{f(y)}{F(c_l)(1-F(c_u))} \end{aligned}$$

where F is the cumulative distribution function of y. The corresponding log conditional likelihood function is

$$\begin{aligned} L\,(y_t|x_t;\theta,c_l,c_u) = \; & \log(f(y_t|x_t;\theta,c_l,c_u)) - \log(F(c_l|x_t;\theta,c_l,c_u)) \\ & - \log(1 - F(c_u|x_t;\theta,c_l,c_u)) \end{aligned}$$

where θ represents all parameters of the distribution.

A censored model is defined by

$$y_t^* = x_t\beta + \varepsilon_t,\; t = 1, 2, \ldots, n$$

with observed y_t values:

$$y = \left\{ \begin{array}{c} y_t^* \text{ if } y_t^* > c_l \text{ and } y_t^* < c_u \\ c_l \text{ if } y_t^* < c_l \\ c_u \text{ if } y_t^* > c_u \end{array} \right\}$$

where c_t and c_u are known. All observations are in the sample, though the observable values, y_t, for which $y_t^* > c_t$ and $y_t^* < c_u$ are set equal to c_l and c_u respectively.

The density of y_t is

$$[f\,(y_t|x_t,\theta,c_u,c_l)]^{1-(D_u+D_l)} \times [F\,(c_l)]^{D_l} \times [1 - F\,(c_u)]^{D_u}$$

where

$$\begin{aligned} D_l &= \begin{array}{c} 0 \text{ if } y_t > c_l \text{ (i.e. } y_t^* > c_l) \\ 1 \text{ if } y_t = c_l \text{ (i.e. } y_t^* \le c_l) \end{array} \\ D_u &= \begin{array}{c} 0 \text{ if } y_t < c_u \text{ (i.e. } y_t^* \le c_u) \\ 1 \text{ if } y_t = c_u \text{ (i.e. } y_t^* > c_u) \end{array} \end{aligned}$$

with the corresponding conditional log likelihood

$$\begin{aligned}\log\ f(y_t|x_t;\theta,c_u,c_l) &= (1-(D_u+D_l))\log\ f(y_t|xxx)\\ &+D_l\log\ F(c_l)+D_u\log[1-F(c_u)]\end{aligned}$$

5.4 Zero-Inflated Models

A zero-inflated (sometimes called zero-altered) model allows for the possibility that count outcomes equal to zero are generated by two regimes: a regime where the outcome is always zero and either a Poisson or Negative Binomial model with zero as one of the outcomes.

Suppose $z_i = 0$ when regime 1 generates outcome *i* (equaling zero) and $z_i = 1$ when regime two generates outcome *i* (possibly equaling zero).

$P\left[z_i = 1\right]$ is determined by a logit or probit model and $P\left[y_i = j|z_i = 1\right]$ is given by a Poisson probability density function.

Greene (2000, p. 890) summarizes these ideas, citing works by Mullahey (1986), Heilbron (1989), Lambert (1992), Johnson and Kotz (1970), and Greene (1994):

$$\begin{aligned}P\left[y_i=0\right] &= P\left[y_i=0|regime1\right]P[regime1]+P[y_i=0|regime2]P[regime2]\\ &= P[regime1]+P[y_i=0|regime2]P[regime2]\\ P\left[y_i=j\right] &= P[y_i=l|regime2]P[regime2]j=1,2,\ldots\end{aligned}$$

5.4.1 Testing Zero-Inflated Regime Assumptions

Vuong (1989) proposes a method that can be used to test whether two regimes likely generate the data. The statistic compares the probabilities of counts occurring under two regimes. Following Greene's (2000, p. 891) notation, let $f_i\left(y_i|\mathbf{x}_i\right)$ be the predicted probability that y_i is observed assuming the data are sampled from distribution *j*, *j*=1,2. Compare these values with

$$m_i = \log\left(\frac{f_1(y_i|\mathbf{x}_l)}{f_1(y_i|\mathbf{x}_l)}\right)$$

Vuong's statistic is:

$$\nu = \frac{\sqrt{N}\left[\frac{1}{N}\sum_{1=1}^{N} m_i\right]}{\sqrt{\frac{1}{N}\sum_{1=1}^{N}(m_i-\overline{m})^2}}$$

which converges in distribution to a standard normal distribution. Large values of ν suggest that model 1 more likely generates the data while small values of ν suggest that model 2 more likely generates the data.

5.5 Multinomial Logit Model

The **multinomialLogit** procedure estimates a multinomial logit model.

For the probability of observing $y_i = m$ we have

$$Pr\left(y_i = m|x_i\right) = \frac{exp(x_i\beta_m)}{\sum_{j=1}^{J} exp(x_i\beta_j)}$$

By default the set of coefficients for the first category, β_1, is set to a zero vector as a "reference" category. This can be modified by the user to any of the categories.

Estimates are found by minimizing

$$-\ln L = -\sum_{i=1}^{N} Pr\left(y_i = m|x_i\right)$$

5.5.1 Adjacent Categories Multinomial Logit

The adjacent categories model is a special case of multinomial logit (Long, 1997, p. 146). It specifies that the log odds of one category versus the next higher category is linear in the cut points and explanatory variables, i.e.,

$$\ln\left[\frac{P(y_i=j+1|\mathbf{x}_i)}{P(y_i=j|\mathbf{x}_i)}\right] = x_i\beta_j$$

This implies

$$\begin{aligned}
\beta_1^{mnl} &= \beta_1^{acl} \\
\beta_2^{mnl} &= \beta_2^{acl} + \beta_1^{acl} \\
\beta_3^{mnl} &= \beta_3^{acl} + \beta_2^{acl} + \beta_1^{acl} \\
&\vdots
\end{aligned}$$

adjacentCategories first estimates the standard multinomial logit model, transforms the β_m^{mnl} parameters to the β_m^{acl} parameters, and computes the covariance matrix of the parameters by the delta method.

5.5.2 Example

The included **adjacentCategories** example uses General Social Survey occupational outcomes data stored in the *gssocc_mat* data mat included with the **DC** examples. The independent data, *occatt*, for this analysis is stored in the first column and the dependent variables, *exper*, *educ*, and *white* are stored in column two through four. The first step to performing analysis is to load the data:

```
new;
cls;
library dc;

//Load Data
loadm y = gssocc_mat;
```

Once data is loaded, estimation features are specified using the *dcControl* structure. This structure must be declared then initialized using the **dcControlCreate** procedure:

```
//Step One: dcControl structure
//Declare dcControl structure
struct dcControl dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();
```

Prior to estimation, the ***dcSet*** procedures may be used to specify variable names for reference and results reports. For the adjacent categories model we begin by describing the dependent data and the categories of responses allowed, using **dcSetYVar**, **dcSetYLabel**, and **dcSetYCategoryLabels**:

```
//Step Two: Describe data names
//Dependent variable
dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"occatt");
```

The independent data names are set using **dcSetXVars** and **dcSetXLabels**:

```
//Independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"exper,educ,white");
```

In addition, **dcSetReferenceCategory** procedure must be used to specify a reference category for estimation:

```
//Reference category excluded from regression
dcSetReferenceCategory(&dcCt,1);
```

Next, the *dcOut* structure is declared:

```
//Step Three: Declare dcOut structure
struct dcOut dcOut1;
```

Finally, calling the **adjacentCategories** procedure estimates the model and results are reported using the **printDCOut** procedure:

```
//Step Four: Call adjacentCategories
dcOut1 = adjacentCategories(dcCt);

//Print Results
call  printDCOut(dcOut1);
```

5.6 Stereotype Multinomial Logit

For the stereotype model, regression vectors across categories are constrained to a linear function of each other. For $Pr(y_i = m|x_i)$ we have

$$Pr\left(y_i = m|x_i\right) = \frac{exp(x_i\phi_m\beta_m)}{\sum_{j=1}^{J} exp(x_i\phi_j\beta_j)}$$

where ϕ_m is a distance coefficient. This model requires two reference categories, one with the distance set to zero, and another which is set to one. By default $\phi_0 = 0$ and $\phi_M = 1$. The remaining distances are constrained to be between zero and one.

5.6.1 Example

The included **stereoLogit** example uses data stored in the *aldnel_mat* data mat included with the **DC** examples. The independent data, *ABC*, measures student grades and is stored in the first column of the data matrix. The dependent variables, *GPA*, *TUCE*, and *PSI* are stored in column two through four. The first step to performing analysis is to load the data:

```
new;
cls;
library dc;

//Load Data
loadm y = aldnel_mat;
```

Once data is loaded, estimation features are specified using the *dcControl* structure. This structure must be declared then initialized using the **dcControlCreate** procedure:

```
//Step One: dcControl structure
//Declare dcControl structure
struct dcControl dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();
```

Prior to estimation, the ***dcSet*** procedures may be used to specify variable names for reference and results reports. For the adjacent categories model we begin by describing the dependent data and the categories of responses allowed, using **dcSetYVar**, **dcSetYLabel**, and **dcSetYNameCategory**:

```
//Step Two: Describe data names
//Dependent variable
dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"ABC");

//Dependent variable categories
dcSetYCategoryLabels(&dcCt,"A,B,C");
```

The independent data names are set using **dcSetXVars** and **dcSetXLabels**:

```
//Independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"GPA,TUCE,PSI");
```

Next, the *dcOut* structure is declared:

```
//Step Three: Declare dcOut structure
struct dcOut dcOut1;
```

Finally, calling the **stereoLogit** procedure estimates the model and results are reported using the **printDCOut** procedure:

```
//Step Four: Call stereoLogit
dcOut1 = stereoLogit(dcCt);

//Print Results
call printDCOut(dcOut1);
```

5.7 Ordered Logit/Probit

Suppose $y_t^* = x_i\beta + \epsilon$ is an unobserved latent variable where x_i is 1xK, β is Kx1, and ϵ is i.i.d. logistic with zero mean and variance $\frac{\pi^2}{3}$. There are J ordinal categories. The model is identified by excluding the constant term. (See Long, 1997, p. 124 for discussion of alternate parameterizations.)

The observed y for an individual depends on the intensity of y^* relative to cut point parameters $\tau_i \quad i = 1, \ldots, J-1$, defined by

$$P(y_i = j|\mathbf{x}_i) = P\left(\tau_{j-1} \leq y_i^* < \tau_j|\mathbf{x}_i\right) = F(\tau_j - x_i\beta|\mathbf{x}_i) - F(\tau_{j-1} - x_i\beta|\mathbf{x}_i)$$

where $\tau_0 = -\infty, 0 < \tau_1 < \ldots \tau_{J-1}$ and

$F(j|\mathbf{x}_i) = P(y_i \leq j|\mathbf{x}_i) - \sum_{k=1}^{j} P(y_i = k|\mathbf{x}_i)$. F is a logit cumulative distribution function.

The cumulative log odds in the ordered logit model is linear in the cut points and explanatory variables, i.e.,

$$\ln\left[\frac{P(y_i \leq j|\mathbf{x}_i)}{P(y_i > j|\mathbf{x}_i)}\right] = \tau_j - x_i\beta$$

The ordered log likelihood is:

$$\ln L(\beta, \tau) = \sum_{j=1}^{J} \sum_{y_i - j} \ln\left[F(\tau_j - x_i\beta|\mathbf{x}_i) - F(\tau_{j-1} - x_i\beta|\mathbf{x}_i)\right]$$

For the ordered logit model, F is the cdf of the logistic distribution and for the ordered probit model, F is the Normal cdf.

5.7.1 Example

The included **orderedLogit** example uses data stored in the *aldnel_mat* data mat included with the **DC** examples. The independent data, *ABC*, measures student grades and is stored in the first column of the data matrix. The dependent variables, *GPA*, *TUCE*, and *PSI* are stored in columns two through four. The first step to performing analysis is to load the data:

```
new;
cls;
library dc;

//Load Data
loadm y = aldnel_mat;
```

Once data is loaded, estimation features are specified using the *dcControl* structure. This structure must be declared then initialized using the **dcControlCreate** procedure:

```
//Step One: dcControl structure
//Declare dcControl structure
struct dcControl dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();
```

Prior to estimation, the ***dcSet*** procedures may be used to specify variable names for reference and results reports. For the adjacent categories model we begin by describing the dependent data and the categories of responses allowed, using **dcSetYVar**, **dcSetYLabel**, and **dcSetYCategoryLabels**:

```
//Step Two: Describe data names
//Dependent variable
dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"ABC");

//Dependent variable categories
dcSetYCategoryLabels(&dcCt,"A,B,C");
```

The independent data names are set using **dcSetXVars** and **dcSetXLabels**:

```
//Independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"GPA,TUCE,PSI");
```

Next, the *dcOut* structure is declared:

```
//Step Three: Declare dcOut structure
struct dcOut dcOut1;
```

Finally, calling the **orderedLogit** procedure estimates the model and results are reported using the **printDCOut** procedure:

```
//Step Four: Call orderedLogit
dcOut1 = orderedLogit(dcCt);

//Print Results
call printDCOut(dcOut1);
```

5.8 Conditional Logit

In the conditional logit model, variables that measure the attributes of the categories are added to the model.

$$Pr\left(y_i = m|x_i\right) = \frac{exp(x_i\beta_m + z_{im}\gamma)}{\sum_{j=1}^{J} exp(x_i\beta_j + z_{ij}\gamma)}$$

5.8.1 Example

The included **conditionalLogit** example uses Powers and Xie (2000) categorical data stored in the *powersxie_mat* data matrix included with the **DC** examples. The independent data, *mode*, measures mode of transportation choice, train, bus, or car, and is stored in the second column of the data matrix. The attributes of these categories, terminal waiting time (ttme), in vehicle choice (invc), in vehicle time (invt), and generalized cost (GC), are stored in columns three through six. Finally, the category variable for the dependent data is stored in column one. The first step to performing analysis is to load the data:

```
new;
cls;
library dc;

//Load Data
loadm y = powersxie_mat;
```

Once data is loaded, estimation features are specified using the *dcControl* structure. This structure must be declared then initialized using the **dcControlCreate** procedure:

```
//Step One: dcControl structure
//Declare dcControl structure
struct dcControl dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();
```

Prior to estimation, the ***dcSet*** procedures may be used to specify variable names for reference and results reports. For the adjacent categories model we begin by describing the dependent data and the categories of responses allowed, using **dcSetYVar**, **dcSetYLabel**, and **dcSetYCategoryLabels**:

```
//Step Two: Describe data names
//Dependent variable
dcSetYVar(&dcCt,y[.,2]);
dcSetYLabel(&dcCt,"mode");
dcSetCategoryVarLabels(&dcCt,"choiceno");

//Category Labels
dcSetCategoryVar(&dcCt,y[.,1]);
dcSetYCategoryLabels(&dcCt,"train,bus,car");
```

The independent data names are set using **dcSetAttributeVars** and **dcSetAttributeLabels**:

```
//Attribute variables
dcSetAttributeVars(&dcCt,y[.,3:6]);
dcSetAttributeLabels(&dcCt,"ttme,invc,invt,GC");

//Turn off constant
dcSetConstant(&dcCt,"off");
```

Next, the *dcOut* structure is declared:

```
//Step Three: Declare dcOut structure
struct dcOut dcOut1;
```

Finally, calling the **conditionalLogit** procedure estimates the model and results are reported using the **printDCOut** procedure:

```
//Step Four: Call conditionalLogit
dcOut1 = conditionalLogit(dcCt);

//Print Results
call printDCOut(dcOut1);
```

5.9 Nested Logit

nestedLogit is a generalization of the conditional logit model in which categories are grouped into subcategories. Define the probability of an observation being in the m-th category given being in the j-th subcategory:

$$P_{m|j} = \frac{exp(z_{m|z}\beta_1)}{\sum_K^J exp(z_{k|j}\beta_1)}$$

Now let

$$P_j = \frac{exp(z_j\beta_2 + \tau_j I_j)}{\sum_K^J exp(z_j\beta_2 + \tau_k I_k)}$$

where

$$I_j = ln\left(\sum_{k=1}^{K_j} exp(z_{m|j}\beta_1)\right)$$

$\rho_j = 1 - \tau_j$ can be interpreted as an approximate subcategory correlation (Maddala, 1983).

Then, the joint probability of category and subcategory is

$$P_{m,j} = P_{m|j}\,P_j$$

and maximum likelihood estimates are produced by minimizing

$$-\ln L = -\sum_{i=1}^{N} P_{m,j}$$

This model can be generalized to any number of levels of subcategories (Maddala, 1983; Greene, 2000).

5.9.1 Example

The included **nestedLogit** example uses categorical data stored in the *hensher_mat* data matrix included with the **DC** examples. The dependent data, *mode*, measures mode of transportation choice, train, bus, or car, and is stored in the second column of the data matrix. The attributes of these categories, terminal waiting time (ttme), in vehicle choice (invc), in vehicle time (invt), and generalized cost (GC), are stored in columns three through six. Finally, the category variable for the dependent data is stored in column one. The first step to performing analysis is to load the data:

```
new;
cls;
library dc;

//Load Data
loadm y = hensher_mat;
```

Once data is loaded, estimation features are specified using the *dcControl* structure. This structure must be declared then initialized using the **dcControlCreate** procedure:

```
//Step One: dcControl structure
//Declare dcControl structure
struct dcControl dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();
```

Prior to estimation, the ***dcSet*** procedures may be used to specify variables for reference and results reports. For the nested logit model we begin by describing the dependent data and the categories of responses allowed, using **dcSetYVar**, **dcSetYLabel**, and **dcSetYCategoryLabels**:

```
//Step Two: Describe data names
//Dependent variable
dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"mode");
dcSetYCategoryLabels(&dcCt,"Air,Train,Bus,Car");
```

The independent attribute data is set using **dcSetAttributeVars** and **dcSetAttributeLabels**:

```
//Attribute variables
dcSetAttributeVars(&dcCt,y[.,2]~y[.,5]~y[.,8]);
dcSetAttributeLabels(&dcCt,"TTME,GC,AIRHINC");
```

Unique to the **nestedLogit** is the required step of setting up nests. First, the number of nests is set using **dcMakeLogitNests**. This procedure requires two inputs: a pointer to the *dcControl* structure and the number of nests to create:

```
dcMakeLogitNests(&dcCt,2);
```

Next, attributes and attribute categories are added to specific nests using **dcSetLogitNestAttributes** and **dcSetLogitNestCategories**. Both of these function require three inputs: a pointer to a *dcControl*, a scalar nest number to add attributes/categories to, and a string of variables names.

```
//Set attributes and categories for lower nest (Nest One)
dcSetLogitNestAttributes(&dcCt,1,"TTME,GC");
dcSetLogitNestCategories(&dcCt,1,"Air,Train,Bus,Car");

//Set attributes and categories for lower nest (Nest Two);
dcSetLogitNestAttributes(&dcCt,2,"AIRHINC");
dcSetLogitNestCategories(&dcCt,2,"Fly,Ground");
```

The final step to setting up nests is to assign attributes to categories in upper level nests. Attributes must always be assigned to categories in the immediate proceeding nest:

```
//Make nest assignments
dcAssignLogitNests(&dcCt,1,"Air,Train,Bus,Car",
              "Fly,Ground,Ground,Ground");
```

Next, the *dcOut* structure is declared:

```
//Step Three: Declare dcOut structure
struct dcOut dcOut1;
```

Finally, calling the **nestedLogit** procedure estimates the model and results are reported using the **printDCOut** procedure:

```
//Step Four: Call nestedLogit
dcOut1 = nestedLogit(dcCt);

//Print Results
call printDCOut(dcOut1);
```

5.10 Summary Statistics

Several goodness-of-fit measures are printed by **`mnlprt`**. Suppose the dependent variable is y; there are N observations and K+1 explanatory variables (including a constant term); the fitted values are $\widehat{\mu_i}$; $L(r)$ is the restricted likelihood of the model with only an intercept and no other explanatory variables and $L(u)$ is the unrestricted likelihood, the model estimated with an intercept and all explanatory variables.

These include

1. The likelihood ratio statistic is:

$$LR = -2\ln\left[\frac{L(r)}{L(u)}\right]$$

is the number of events expected to occur per unit time (or space).

2. McFadden's (1973) pseudo R-square is:

$$R^2_{McF} = 1 - 2\ln\left[\frac{L(u)}{L(r)}\right]$$

3. Ben-Akiva and Lerman (1985) revise McFadden's measure to compensate for the effect of additional variables on a regression's explanatory power. Their measure, analogous to adjusted R^2, is

$$\overline{R}^2_{McF} = 1 - \frac{\ln L(u) - K}{\ln L(r)}$$

4. Greene (2000, p. 882) presents an R^2 measure based on standardized residuals.

$$R^2_p = 1 - \frac{\sum_{i=1}^{N}\left[\frac{y_i - \widehat{\mu_i}}{\sqrt{\widehat{\mu_i}}}\right]^2}{\sum_{i=1}^{N}\left[\frac{y_i - \bar{y}}{\sqrt{\bar{y}}}\right]^2}$$

5. As noted in Greene (2000, p. 883), Cameron and Windmeijer (1993) present an R^2 measure based on the deviances of individual observations, $d_i = 2\left[y_i \ln\left(\frac{y_i}{\widehat{\mu_i}}\right) - (y_i - \widehat{\mu_i})\right]$

$$R^2_d = 1 - \frac{\sum_{i=1}^{N}\left[y_i \log\left(\frac{y_i}{\widehat{\mu_i}}\right) - (y_i - \widehat{\mu_i})\right]}{\sum_{i=1}^{N}\left[y_i \log\left(\frac{y_i}{\mu_i}\right)\right]}$$

6. Cragg and Uhler (1970) propose a normed likelihood ratio, based on Maddala's (1983) showing that the maximum of R^2_{ML} is $1 - L(r)^{2/N}$

$$R^2_{C\&U} = \frac{R^2_{ML}}{\max R^2_{ML}} = \frac{1-[L(r)/L(u)]^{2/N}}{1-L(r)^{2/N}}$$

7. The count R^2 is the proportion of correct predictions, i.e.

$$R^2_{Count} = \frac{1}{N}\sum_j n_{jj}$$

where $\max(n_{r+})$ is the number of correct predictions for outcome j.

8. The adjusted count R^2 uses the highest marginal frequency to adjust for the "spurious" successes that result by predicting that an outcome will fall in the category with the greatest percentage of observed successes. It is the proportion of successful categorizations occurring above what would occur by simply choosing the category with the greatest prior chance of success.

$$R^2_{AdjCount} = \frac{\Sigma_j n_{jj} - \max_r(n_{r+})}{N - \max_r(n_{r+})}$$

where n_{jj} is the maximum of the contingency table row marginals, the "number of cases in the outcome with the most observations" (Long, 1997, p. 108).

9. The average Akaike information criterion (AIC) is

$$AIC = \frac{-2[\ln L(u) - K]}{N}$$

10. The average Bayesian (Schwarz) information criterion (BIC) is

$$BIC = \frac{-2\ln L(u) + K\ln(N)}{N}$$

11. The average Hannan-Quinn criterion is

$$HQIC = \frac{-2[\ln L(u) - K\ln(\ln(N))]}{N}$$

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

6 Linear Classification

The **GAUSS** **`logisticRegress`** procedure solves large-scale, binary classification problems. It includes a suite of tools for scaling and recoding data, estimating attribute weights, model cross-validation, and prediction. These classification tools are split into two categories, logistic regression models [LR] and linear support vector machines [SVM]. These tools are built on the LIBLINEAR library from Fan, *et al.* (2014). Both methodologies, logistic regression and linear support vector machines, provide predictions and data relationships derived from the general optimization problem,

$$\min_w \tfrac{1}{2} w^T w + C \sum_i^l \xi(w; x_i y_i),$$

$$i = 1, \cdots, l,$$

$$x_i \in R^n,$$

$$y_i \in \{-1, +1\},$$

where $C > 0$ is a penalty parameter and $\xi(w;x_i;y_i)$ is a loss function. The LR and SVM methodologies differ in the specified loss function. The LR loss function uses a probabilistic model loss function given by

$$log\left(1 + e^{-y_1 w^T x_i}\right)$$

Within the SVM family there are two common loss functions. The L1-SVM loss function is

$$max(1 - y_i w^T x_i, 0)$$

The L2-SVM loss function is

$$max\,(1 - y_i, w^T x_i, 0)^2$$

The **`logisticRegress`** procedure's L1-SVM and L2-SVM methods use the coordinate descent method for optimization. A trust region Newton method is used for LR and is available as an option for L2-SVM.

In addition to estimation tools, the **DC** package includes tools for pre-estimation data preparation and post-estimation prediction and analysis. The implementation of **`logisticRegress`** is intuitive and requires three inputs, the *`lrControl`* structure, a *`y`* data vector, and an *`x`* matrix of attributes.

6.1 Estimation

6.1.1 Model Selection

Model selection from the classifiers implemented in the **logisticRegress** procedure should be based in part on theoretical background and data characteristics. In most cases the models will give similar results. However, some general starting guidelines will make selection easier and may enhance performance. First, it is recommended to try utilizing the dual-based solvers before the primal-based solvers. Secondly, L1-regularization tends to yield higher accuracy rates than L2-regularization. For this reason, users may prefer using L1-regularization and switching to L2-regularization if the L1 training is slow (Fan, *et al. 2008)*.

6.1.2 Model Parameters

The primary user-controlled parameter of the **logisticRegress** procedure is *C*, the loss function penalty parameter. Fan, *et al.*(2008) note that this algorithm is relatively insensitive to *C*, and larger *C* values are generally less computationally efficient. Hence, the default *C* value is recommended as a starting point. Parameter sensitivity can be tested by slightly increasing *C* and comparing the outcome to default results.

6.1.3 Cross-validation

Cross-validation using the **logisticRegress** procedure is controlled by the *lrControl* structure member *crossValidation*. This structure member should be set to equal the desired number of folds, *k*, for performing k-fold cross-validation. The default value is 0; commonly employed values are 5 or 10.

The **logisticRegress** procedure can also use cross-validation to assist with solver type or penalty parameter selection. To perform cross-validation selection of *solverType* or *C*, set the corresponding *lrControl* structure member to a vector of parameter candidates. The **logisticRegress** procedure will then run iterative cross-validation across the parameter candidates and select the highest cross-validation accuracy parameter for prediction. It should be noted that the *logisticRegress* procedure will only perform cross-validation on one parameter at a time, either *solverType* or *C*. As an example, selection of solver type 1, 2, or 3 using 5-fold cross-validation can be performed using the code below:

```
struct lrControl lctl;
lctl = lrGetDefaults();

//Solver type vector
lctl.solverType = { 1, 2, 3 };

//Turn on cross-validation
```

```
lctl.crossValidation = 5;

//Turn on prediction
lctl.predict = 1;
```

6.1.4 The lrControl Structure

The *lrControl* structure allows users to specify the necessary parameters used in the linear classification LR or SVM tools. An instance of the *lrControl* structure named *lctl* contains the members

lctl.solverType	Matrix, scalar indicator of classification problem. If non-scalar, *crossValidation* must be non-zero and is used to pick highest cross-validation accuracy solver:	
	0	L2-regularized logistic regression,
	1	L2-regularized logistic regression, L2 loss SVC dual,
	2	L2-regularized logistic regression, L2 loss SVC,
	3	L2-regularized logistic regression, L1 loss SVC dual,
	4	MCSVM CS,
	5	L1R, L2 loss SVC,
	6	L1R, logistic regression,
	7	L2R, logistic regression dual,
	11	L2R, L2 loss SVR regression model,
	12	L2 loss SVR regression model,
	13	L2R, L1 loss SVR dual
lctl.eps	Scalar, the stopping condition for KKT approximation algorithm.	
lctl.C	Matrix, loss function penalty parameter. If non-scalar, *crossValidation* must be non-zero and the highest cross-validation accuracy is used to select optimal C. Values of C<0 are not permissible. Default=1.	
lctl.p	Scalar, loss function tolerance.	
lctl.bias	Scalar, 0 or 1. If set to 1, a bias feature will be added to the end of the incoming 'x' matrix. This bias feature will be a vector of ones. Default=1.	
lctl.crossValidation	Scalar, specifies number of folds for k-fold cross-validation. If equal to 0	

	no cross-validation.
lctl.predict	Scalar, indicator variable to conduct post-estimation prediction. Default=0.
lctl.plotPredict	Scalar, indicator variable to plot post-estimation predictions. Default=0.
lctl.printOutput	Scalar, indicator variable to print output to screen. Default=1.
lctl.scaleX	Scalar, indicator parameter for pre-estimation data scaling method. Default=2. 0 No scaling. 1 Z-Score normalization. 2 [0,1] Min/Max normalization. [Default] 3 Scale by 1/sqrt(k) where k=number features. 4 Center data. 5 Sigmoidal scale.

Using the *lrControl* structure requires two steps, declaring an instance of the structure, and initializing the members in the structure using the **lrGetDefaults** procedure:

```
struct lrControl lctl;
lctl = lrGetDefaults();
```

Calling **lrGetDefaults** assigns all members in *lctl* to the default values. Changing parameters to match individual needs is done using GAUSS "." referencing for structure members. As an example, consider changing the solver type to the L2-regularized logistic regression, L1 loss SVC dual method:

```
lctl.solverType = 3;
```

Similary, the model can be set to perform cross-validation to select *C* from a vector of possible values and the predict y using the selected *C*:

```
lctl.C = {1,2,3,4,5};
lctl.crossValidation = 10;
lctl.predict = 1;
```

6.1.5 The lrOut Structure

All output from the **logisticRegress** procedure is stored in the *lrOut* structure. All members within this structure are easily accessible, allowing use results for further computation. An instance of the *lrOut* structure named *lOut* contains the members:

lrOut.weights	Matrix, estimated weights for specified independent variables.
lrOut.yPredict	Matrix, predicted observations using estimated weights and data matrix.
lrOut.probability	Matrix, probabilities from logistic regression used to for determining predicted y classifications.
lrOut.cvAccuracy	Matrix, cross-validation prediction accuracy.
lrOut.predictionAccuracy	Scalar, full sample prediction accuracy.
lrOut.optimalC	Scalar, optimal C based on highest cross-validation accuracy.
lrOut.optimalSolver	Scalar, optimal solver based on highest cross-validation accuracy.

6.2 The logisticRegress procedure

The GAUSS **logisticRegress** procedure solves large scale classification problems and provides tools for estimating attribute weights, model cross-validation, and prediction. Implementation of **logisticRegress** requires three inputs, the *lrControl* structure, a *y* data vector, and a *x* matrix of attributes. The *lrControl* structure facilitates model selection and estimation parameters. Usage of the *lrControl* structure is previously described and requires two steps, declaring the structure and initializing the structure:

```
struct lrControl lctl;
lctl = lrGetDefaults();
```

Data Coding

The y-matrix input of the **logisticRegress** procedure must house a discrete, dichotomous dependent variable matrix with values {0,1}. Any binary data vector can be reclassified to a {0,1} data vector using the **reclassify** procedure. This procedure can restructure categorical numerical or string data into a vector of numerical, sequential categorical data. As an example, consider the vector y, containing string data coded as *Yes* or *No*. To reclassify as {0,1} data

```
from = "No" $| "Yes";
to = { 0, 1 };
y_new = reclassify(y, from, to);
```

The output, *y_new*, from **reclassify** will be a vector of binary data coded as {0,1}.

The final input is a matrix containing continuous or discrete independent attributes. This data must be numerical and any categorical, string data should be recoded before passing to the **logisticRegress** procedure.

Data Scaling

In addition, the **logisticRegress** procedure generally performs best with scaled attributes and the *lrControl* structure element *scaleX* can be used to set scaling methods. By default, **logisticRegress** scales all features on a [0,1] scale prior to estimation. However, five scaling methods are allowed:

1. Z-score normalization
2. [0,1] min-max scaling
3. Quantity of feature scaling
4. Demeaning
5. Sigmoidal normalization

To rescale the features data matrix, *x*, using sigmoidal normalization:

```
//Scale data using sigmodial normalization
lctl.scaleX = 5;
```

Model Execution

Once the control structure is declared and initialized and all data is properly formatted, calling the **logisticRegress** procedure performs estimation of attribute weights, along with cross-validation if specified.

```
struct lrOut outLR;
outLR = logisticRegress(lctl, y, x);
```

6.3 Linear Classification Example

The first step to using the **logisticRegress** procedure is insuring that data is in the proper format and scaled appropriately.The y-matrix input of the **logisticRegress** procedure must house a discrete, dichotomous dependent variable matrix with values {0,1}. Independent attributes used in **logisticRegress** may be continuous or discrete. However, all data must be numerical, and any categorical, string data should be recoded before passing to the **logisticRegress** procedure.

GAUSS includes the *reclassify* procedure for easy data setup. This procedure can be used to reclassify binary data to {0,1} data, and to convert categorical string data to categorical numeric data. The **reclassify** procedure requires three inputs: a N x 1 data vector to be reclassified, *from*: the categories or levels of the first input, and *to*: a vector, containing the new values for each category. As an example, consider the vector *y*, containing string data coded as *Yes* or *No*. To reclassify as {0,1} data:

```
from = "No" $| "Yes";
to = { 0, 1 };
y_new = reclassify(y,from, to);
```

The output from **reclassify**, *y_new*, will be a vector of binary data coded as {0,1}.

Following data set-up, the next step to implementing model specifics is to declare and initialize the *lrControl* structure:

```
struct lrControl lctl;
lctl = lrGetDefaults();
```

Next, change the solver type to the L2-regularized logistic regression, L1 loss SVC dual method:

```
lctl.solverType = 3;
```

In addition, we will specify both post-estimation prediction and prediction plotting using the *lctl.predict* and *lctl.predictPlot* elements:

```
lctl.predict = 1;
lctl.predictPlot = 1;
```

Finally, calling the **logisticRegress** procedure performs estimation of attribute weights, along with cross-validation if specified.

```
struct lrOut outLR;
outLR = logisticRegress(lctl, y, x);
```

6.4 References

1. R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. 2008. **LIBLINEAR: A library for large linear classification**, **Journal of Machine Learning Research**. 9 (2008).1871-1874.
2. R.-E. Hsu, C.-W. Chang, C.-C. and C.-J. Lin. 2010. **A Practical Guide to Support Vector Classification**, **Machine Learning**. 46(1-3).219-314.
3. Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

7 Discrete Choice Reference

The Discrete Choice Reference chapter describes each of the commands, procedures and functions available in **Discrete Choice**.

adjacentCategories

Purpose

Estimates the Adjacent Categories Multinomial Logit model.

Library

dc

Format

out = **adjacentCategories**(*cont*);

Input

cont an instance of a *dcControl* structure.

cont.myData	an instance of a *dcData* structure containing the elements:	
	cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
	cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with

		outcomes but rather with individuals.
	cont.myData.categoryData	Matrix, discrete categorical data.
	cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **adjacentCategories** computes start values.	
	b0	1 *L* matrix, constants in regression.
	b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate;

//Intercept must be L by 1
b0 = { 0 1 1 1 1};
```

```
//Coefficients must be K by L
b = { 0 .1 .1 .1 .1,
      0 .1 .1 .1 .1,
      0 .1 .1 .1 .1};

//Mask must be K by L
mask = { 0  1  1 1 1,
         0  1  1 1 1,
         0  1  1 1 1};

cont.startValues =
  pvPackmi(cont.startValues,
  b0 , "b0" , mask[1,.] , 1);
cont.startValues =
  pvPackmi(cont.startValues,
  b , "b" , mask, 2);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters.[[For more details see Section 4.1.6.]]

cont.C — $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — $M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

cont.inEqProc — scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV*

	parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or $K\times2$ matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out	an instance of a *dcOut* structure		
	out.par	instance of *PV* structure containing estimates.	
		b0	1 $L\times1$ matrix, constants in regression.
		b	2 $L\times K$ matrix, regression

coefficients (if any). Coefficients associated with reference category are fixed to zeros.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, *pvUnpack* returns a scalar missing value with error code = 99.

out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L \times 1$ vector, percentages of dependent variable by category.
out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
out.marginVC	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.fittedVals	$N \times 1$ matrix of predicted (fitted) counts.
out.resids	$N \times 1$ matrix of residuals.

out.summaryStats	17×1 matrix of goodness-of-fit measures.	
	1	Log-Likelihood, full model.
	2	Log-Likelihood, restricted model (all slope coefficients equal zero.
	3	Degrees of freedom.
	4	Chi-square statistic.
	5	Number of Parameters.
	6	McFadden's Pseudo R-Squared.
	7	Madalla's Pseudo R-Squared.
	8	Cragg and Uhler's normed likelihood ratios statistics.
	9	Akaike information criterion (AIC).
	10	Bayesian information criterion (BIC).
	11	Hannon-Quinn Criterion.
	12	Count R-Squared.
	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Load Data
loadm y = gssocc_mat;

//Step One: dcControl structure
//Declare dcControl structure
struct dccontrol dcCt;

//Initialize dcControl structure
dcCt = dcControlCreate();

//Step Two: Describe data names
//Dependent variable
dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"occatt");

//Dependent variable categories
dcSetYCategoryLabels(&dcCt,"Menial,BC,Craft,WC,Pro");

//Independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"exper,educ,white");

//Reference category excluded from regression
dcSetReferenceCategory(&dcCt,1);

//Step Three: Call adjacentCategories
//Declare dcOut Structure
struct dcOut dcOut1;
dcOut1 = adjacentCategories(dcCt);

//Print Results
call printDCOut(dcOut1);
```

Remarks

The adjacent category model is a special case of the multinomial logit model where the coefficients of succeeding categories are constrained to be greater than their preceding counterparts.

Source

`dcaclogit.src`

binaryLogit

Purpose

Estimates a logit regression model.

Library

dc

Format

out = **binaryLogit**(*cont*);

Input

cont	an instance of a *dcControl* structure.		
	cont.myData	an instance of a *dcData* structure containing the elements:	
		cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
		cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
		cont.myData.categoryData	Matrix, discrete categorical data.

	cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.

cont.startValues instance of *PV* structure containing starting values; if not provided, **binaryLogit** computes start values.

b0	1 *L* matrix, constants in regression.
b	2 *K*×*L* matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

//Set start values

//First category is set as
//reference category
//Intercept must be 1 x L
b0 = { 0 1 };

//Coefficient must be K x L
b = { .1 .2 };

//Mask must be K x L
mask = { 0 1 };

cont.startValues = pvPackmi
(cont.startValues,
 b0 , "b0", mask, 1);
cont.startValues = pvPackmi
```

binaryLogit

```
(cont.startValues,
 b, "b" , mask, 2);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C — $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — $M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

cont.inEqProc — scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.

cont.bounds — 1×2 or $K \times 2$ matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.

cont.maxIters — scalar, maximum number of iterations. Default = 1e+5.

cont.dirTol — scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.

cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out an instance of a *dcOut* structure

out.par instance of *PV* structure containing estimates.

b0	1 constant in regression.
b	2 regression coefficients (if any).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

binaryLogit

	if model does not contain a parameter, **`pvUnpack`** returns a scalar missing value with error code = 99.
out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L \times 1$ vector, percentages of dependent variable by category.
out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
out.summaryStats	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.

	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;

//Initialize dc control structure
dcCt = dcControlCreate();

//Load data
loadm y = aldnel_mat;

//Step Two: Describe data names
//Name of dependent variable
dcSetYVar(&dcCt,y[.,5]);
dcSetYLabel(&dcCt,"A");

//Name of independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"GPA,TUCE,PSI");

//Step Three: Declare dcOut struct
struct dcOut dcOut1;

//Step Four: Call binaryLogit
dcOut1 = binaryLogit(dcCt);

call printDCOut(dcOut1);
```

Source

dcbin.src

binaryProbit

Purpose

Estimates a probit regression model.

Library

dc

Format

out = **binaryProbit**(*cont*);

Input

cont — an instance of a *dcControl* structure.

cont.myData	an instance of a *dcData* structure containing the elements:	
	cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
	cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
	cont.myData.categoryData	Matrix, discrete categorical data.
	cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested

		logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **binaryProbit** computes start values.	
	b0	1 L matrix, constants in regression.
	b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

//Set start values
//First category is set
//as reference category
//Intercept must be 1 x L
b0 = { 0 1 };

//Coefficient must be K x L
b = { .1 .2 };

//Mask must be K x L
mask = { 0 1 };

//Pack variable starting values
cont.startValues = pvPackmi(cont.startValues,
b0 , "b0", mask, 1);
cont.startValues = pvPackmi(cont.startValues,
b, "b" , mask, 2);
```

cont.A	$M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.
cont.B	$M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C	$M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.D	$M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or $K \times 2$ matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at

	each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out — an instance of a *dcOut* structure

out.par — instance of *PV* structure containing estimates.

b0	1 constant in regression.
b	2 regression coefficients (if any).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc — $NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.

out.yDist — $L \times 1$ vector, percentages of dependent variable by category.

out.xData — $K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent

binaryProbit

	variables.
`out.marginEffects`	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	$N \times 1$ matrix of predicted (fitted) counts.
`out.resids`	$N \times 1$ matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.

	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;
//Initialize dc control structure
dcCt = dcControlCreate();

//Load data
loadm y = aldnel_mat;

//Step Two: Describe data names
//Name of dependent variable
dcSetYVar(&dcCt,y[.,5]);
dcSetYLabel(&dcCt,"A");

//Name of independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"GPA,TUCE,PSI");

//Step Three: Declare dcOut struct
struct dcOut dcOut1;

//Step Four: Call binaryProbit
dcOut1 = binaryProbit(dcCt);

call printDCOut(dcOut1);
```

Source

dcbin.src

conditionalLogit

Purpose

Estimates the Conditional Logit model.

Library

dc

Format

out = **conditionalLogit**(*cont*);

Input

cont an instance of a *dcControl* structure.

cont.myData	an instance of a *dcData* structure containing the elements:	
	cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
	cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
	cont.myData.categoryData	Matrix, discrete categorical data.
	cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in

		conditional logit and nested logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **conditionalLogit** computes start values.	
	b0	1 $1 \times L$vector, constant in regression.
	b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zero.
	g1	3 $R_1 \times 1$ vector, coefficients of attribute variables for first level.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

//Set starting values
// No b0 because no constant
// No b because no x vars


//Set starting values for attributes
//Stores attributes on first level
//should be R x 1,
//where R is # of attributes on level
g1 = { .1 , .1 , .1 , .1};

mask = { 1 , 1 , 1 , 1 };

cont.startValues =
  pvPackmi(cont.startValues,
  g1, "g1" , mask, 3);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where p is a vector of the

conditionalLogit

	parameters. For more details. see Section 4.1.6.
cont.B	M×1 vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.C	M×K matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.D	M×1 vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.

`cont.randRadius`	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
`cont.trustRadius`	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
`cont.output`	scalar, if nonzero, optimization results are printed. Default = 0.
`cont.printIters`	scalar, if nonzero, prints iteration information. Default = 0.

Output

`out` — an instance of a `dcOut` structure

`out.par` — instance of *PV* structure containing estimates.

b0	1 $L \times 1$ matrix, constant in regression.
b	2 $L \times K$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.
gm	3 M×1 vector, coefficients of attribute variables

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as

conditionalLogit

	a single parameter vector:
	`b = pvGetParVector (out.par);`
	The location of the coefficients in *out.par* can be described by
	`b = pvGetParNames (out.par);`
	if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	*NPARM*×*NPARM* variance-covariance matrix of coefficient estimates.
out.yDist	*L*×1 vector, percentages of dependent variable by category.
out.xData	*K*×4 matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	*L*×1×*K* array, marginal effects of independent variables by category of dependent variable.
out.marginVC	*L*×*K*×*K* array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.atmargineffects	*L*×*L*×1×*R* array, marginal effects by category of attribute variables by category of dependent variable.
out.atmarginvc	*L*×*L*×*R*×*R* array, covariance matrices of marginal effects by category of attribute variables by category of dependent variable.
out.fittedVals	*N*×1 matrix of predicted (fitted) counts.

`out.resids`	N×1 matrix of residuals.	
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.	
	1	Log-Likelihood, full model.
	2	Log-Likelihood, restricted model (all slope coefficients equal zero.
	3	Degrees of freedom.
	4	Chi-square statistic.
	5	Number of Parameters.
	6	McFadden's Pseudo R-Squared.
	7	Madalla's Pseudo R-Squared.
	8	Cragg and Uhler's normed likelihood ratios statistics.
	9	Akaike information criterion (AIC).
	10	Bayesian information criterion (BIC).
	11	Hannon-Quinn Criterion.
	12	Count R-Squared.
	13	Adjusted Count R-Squared.

conditionalLogit

14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;

//Initialize dc control structure
dcCt = dcControlCreate();

//Load data
loadm y = powersxie_mat;

//Step Two: Describe data names
//Dependent variable
dcSetYVar(&dcCt,y[.,2]);
dcSetYLabel(&dcCt,"mode");
dcSetCategoryVarLabels(&dcCt,"choiceno");

//Category Labels
dcSetCategoryVar(&dcCt,y[.,1]);
dcSetYCategoryLabels(&dcCt,"train,bus,car");

//Attributes
dcSetAttributeVars(&dcCt,y[.,3:6]);
dcSetAttributeLabels(&dcCt,"ttme,invc,invt,GC");

//Turn off constant
dcSetConstant(&dcCt,"off");

//Step Three: Declare dcOut struct
struct dcOut dcOut1;
```

```
//Step Four: Call conditionalLogit
dcOut1 = conditionalLogit(dcCt);

call printDCOut(dcOut1);
```

Source

dcclogit.src

dcAdjacentCategories

Purpose

Estimates the Adjacent Categories Multinomial Logit model.

Library

dc

Format

out = **dcAdjacentCategories**(*data, desc, cont*);

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data	
desc	an instance of a *dcDesc* structure.	
	desc.yname	name of dependent variable.
	desc.yvar	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either *desc.yname* or *desc.yvar* may be specified. If data is matrix of data, *desc.yvar* must be specified.
	desc.ytype	scalar, 0 if *desc.yvar* character variable, otherwise 1 if numeric. Default = 1.
	desc.xnames	$K \times 1$ string vector, names of the independent

dcAdjacentCategories

		variable(s).
	desc.xvars	$K \times 1$ vector, indices of the independent variable (s). If data is name of **GAUSS** dataset, either *desc.xnames* or *desc.xvars* may be specified. If data is matrix of data, *desc.xvars* must be specified.
	desc.catnames	$L \times 1$ string vector, names of categories.
	desc.refcat	reference category. If *desc.refcatName* is specified, *desc.refcat* is optional. Default = 1.
	desc.refcatName	string, reference category name. If *desc.refcat* has been specified, *desc.refcatName* is optional. Default = *desc.catnames[1]*.
	desc.noconstant	scalar, 1 if no constants in model. Default = 0.
	desc.marginType	scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean *x*. Default = 0.
	desc.wgtname	string, name of weight variable. If *desc.wgtvar* is specified, the specification of *desc.wgtname* is optional. Default = "".
	desc.wgtvar	scalar, index of weight variable. If *desc.wgtname* is specified, the specification of *desc.wgtvar* is optional. Default = 0.
cont	an instance of a *dcControl* structure.	
	cont.startValues	instance of *PV* structure containing starting values; if not provided, **dcAdjacentCategories** computes start values.
		b0 — 1 L matrix, constants in regression.
		b — 2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with

reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

b0 = { 0  1  1 };

b = { 0 .1 .1,
      0 .1 .1 };

mask = { 0  1  1,
         0  1  1,
         0  1  1 };

cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0",mask[1,.],1);
cont.startValues =
  pvPackmi(cont.startValues,
  b,"b",mask[2:3,.],2);
```

cont.A	$M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where p is a vector of the parameters. For more details. see Section 4.1.6.
cont.B	$M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where p is a vector of the parameters. For more details see Section 4.1.6.
cont.C	$M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where p is a vector of the parameters. For more details see Section 4.1.6.
cont.D	$M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where p is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a

	procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or *K*×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each

	iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out — an instance of a *dcOut* structure

out.par — instance of *PV* structure containing estimates.

b0	1 $L \times 1$ matrix, constants in regression.
b	2 $L \times K$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, *pvUnpack* returns a scalar missing value with error code = 99.

out.vc — $NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.

dcAdjacentCategories

`out.yDist`	$L\times1$ vector, percentages of dependent variable by category.
`out.xData`	$K\times4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
`out.marginEffects`	$L\times1\times K$ array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	$L\times K\times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	$N\times1$ matrix of predicted (fitted) counts.
`out.resids`	$N\times1$ matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.

	12	Count R-Squared.
	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "occatt";
d1.xnames =  "exper" $| "educ" $| "white";
d1.catnames = "Menial" $| "BC" $| "Craft" $| "WC" $| "Pro";

struct dcOut dcOut1;

dcOut1 = dcAdjacentCategories("gssocc",d1,dcControlCreate());

call dcprt(dcOut1);
```

Remarks

The adjacent category model is a special case of the multinomial logit model where the coefficients of succeeding categories are constrained to be greater than their preceding counterparts.

Source

`dcaclogit.src`

dcAssignLogitNests

Purpose

Assigns listed outcome categories to categories within previously created nests.

Library

dc

Format

dcAssignLogitNests(*&cont*, *nestNumber*, *outcomeList*, *categoryList*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
nestNumber	Scalar, nest level of outcomes listed in *outcomeList*.
outcomeList	String Array, M x 1, list of outcomes to be assigned to nest categories.
categoryList	String Array, M x 1, list of category assignments for outcomes in *outcomeList*.

Example

```
library dc;

//Load data
loadm y = hensher_mat;

//Step One: Declare dc control structure
struct dcControl cont;
//Initialize dc control structure
cont = dcControlCreate();

//Step Two: Describe data
//Name of dependent variable
dcSetYVar(&cont,y[.,1]);
dcSetYLabel(&cont, "Mode");
```

```
//Y Category Labels
dcSetYCategoryLabels(&cont,"Air,Train,Bus,Car");
//Specify reference category (excluded)
dcSetReferenceCategory(&cont, "Car");

//Name of independent variable
varlist = "TTME,GC,AIRHINC";
dcSetAttributeVars(&cont,y[.,2]~y[.,5]~y[.,8]);
dcSetAttributeLabels(&cont,"TTME,GC,AIRHINC");

//Set-up nested levels
dcMakeLogitNests(&cont,2);

//Set attributes and categories for lower nest (Nest One)
dcSetLogitNestAttributes(&cont,1,"TTME,GC");
dcSetLogitNestCategories(&cont,1,"Air,Train,Bus,Car");

//Set attributes and categories for lower nest (Nest Two)
dcSetLogitNestAttributes(&cont,2,"AIRHINC");
dcSetLogitNestCategories(&cont,2,"Fly,Ground");

//Make nest assignments
dcAssignLogitNests(&cont,1,"Air,Train,Bus,Car",
"Fly,Ground,Ground,Ground");
```

Remark

Prior to using **`dcAssignLogitNests`** the dependent variable category names must be set using **`dcSetYCategoryLabels`**.

Source

`setnests.src`

dcBinaryLogit

Purpose

Estimates a logit regression model.

dcBinaryLogit

Library

dc

Format

```
out = dcBinaryLogit(data, desc, cont);
```

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.	
desc	an instance of a *dcDesc* structure.	
	desc.yname	name of dependent variable.
	desc.yvar	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either *desc.yname* or *desc.yvar* may be specified. If data is matrix of data, *desc.yvar* must be specified.
	desc.ytype	scalar, 0 if *desc.yvar* character variable, otherwise 1 if numeric. Default = 1.
	desc.xnames	$K \times 1$ string vector, names of the independent variable(s).
	desc.xvars	$K \times 1$ vector, indices of the independent variable(s). If data is name of **GAUSS** dataset, either *desc.xnames* or *desc.xvars* may be specified. If data is matrix of data, *desc.xvars* must be specified.
	desc.catnames	$L \times 1$ string vector, names of categories.
	desc.refcat	reference category. If *desc.refcatName* is specified, *desc.refcat* is optional. Default = 1.
	desc.refcatName	string, reference category name. If *desc.refcat* has been specified, *desc.refcatName* is optional. Default = *desc.catnames[1]*.
	desc.wgtname	string, name of weight variable. If *desc.wgtvar* is specified, the specification of *desc.wgtname* is optional. Default = "".
	desc.wgtvar	scalar, index of weight variable. If *desc.wgtname* is

specified, the specification of *desc.wgtvar* is optional. Default = 0.

desc.noconstant scalar, 1 if no constants in model. Default = 0.

desc.marginType scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean *x*. Default = 0.

cont an instance of a `dcControl` structure.

cont.startValues instance of *PV* structure containing starting values; if not provided, **dcBinaryLogit** computes start values.

b0	1 constant in regression.
b	2 regression coefficients (if any).

For example:

```
struct dcControl cont;
cont = dcControlCreate();

//Set start values
//First category is set
//as reference category
//Intercept must be 1 x L
b0 = { 0 1 };

//Coefficient must be K x L
b = { .1 .2 };

//Mask must be K x L
mask = { 0 1 };

cont.startValues = pvPackmi
(cont.startValues,
 b0 , "b0", mask, 1);
cont.startValues = pvPackmi
(cont.startValues,
 b, "b", mask,2);
```

cont.A $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B	M×1 vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.C	M×K matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.D	M×1 vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.

cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out an instance of a *dcOut* structure

out.par instance of *PV* structure containing estimates.

b0	1 constant in regression.
b	2 regression coefficients (if any).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc $NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.

out.yDist $L \times 1$ vector, percentages of dependent variable by category.

`out.xData`	K×4 matrix, the means, standard deviations, minimums, and maximums of independent variables.
`out.marginEffects`	L×1×K array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	L×K×K array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	N×1 matrix of predicted (fitted) counts.
`out.resids`	N×1 matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.

13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "A";
d1.xnames =  "GPA" $| "TUCE" $| "PSI";

struct dcOut dcOut1;

dcOut1 = dcBinaryLogit("aldnel", d1, dcControlCreate());

call dcprt(dcOut1);
```

Source

dcbin.src

dcBinaryProbit

Purpose

Estimates a probit regression model.

Library

dc

Format

```
out = dcBinaryProbit(data, desc, cont);
```

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.
desc	an instance of a `dcDesc` structure.

desc.yname	name of dependent variable.
desc.yvar	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either *desc.yname* or *desc.yvar* may be specified. If data is matrix of data, *desc.yvar* must be specified.
desc.ytype	scalar, 0 if *desc.yvar* character variable, otherwise 1 if numeric. Default = 1.
desc.xnames	$K \times 1$ string vector, names of the independent variable(s).
desc.xvars	$K \times 1$ vector, indices of the independent variable(s). If data is name of **GAUSS** dataset, either *desc.xnames* or *desc.xvars* may be specified. If data is matrix of data, *desc.xvars* must be specified.
desc.catnames	$L \times 1$ string vector, names of categories.
desc.refcat	reference category. If *desc.refcatName* is specified, *desc.refcat* is optional. Default = 1.
desc.refcatName	string, reference category name. If *desc.refcat* has been specified, *desc.refcatName* is optional. Default = *desc.catnames[1]*.
desc.wgtname	string, name of weight variable. If *desc.wgtvar* is specified, the specification of *desc.wgtname* is optional. Default = "".
desc.wgtvar	scalar, index of weight variable. If *desc.wgtname* is specified, the specification of *desc.wgtvar* is optional. Default = 0.

desc.noconstant scalar, 1 if no constants in model. Default = 0.

desc.marginType scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean *x*. Default = 0.

cont an instance of a *dcControl* structure.

cont.startValues instance of *PV* structure containing starting values; if not provided, **dcBinaryProbit** computes start values.

b0	1 constant in regression.
b	2 regression coefficients (if any).

For example:

```
struct dcControl cont;
cont = dcControlCreate();

//Set start values
//First category is set
//as reference category
//Intercept must be 1 x L
b0 = { 0 1 };

//Coefficient must be K x L
b = { .1 .2 };

//Mask must be K x L
mask = { 0 1 };

cont.startValues = pvPackmi
(cont.startValues,
       b0 , "b0", mask, 1);
cont.startValues = pvPackmi
(cont.startValues,
       b, "b", mask,2);
```

cont.A $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C	$M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.D	$M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or $K \times 2$ matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.

cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out — an instance of a *dcOut* structure

out.par — instance of *PV* structure containing estimates.

b0	1 constant in regression.
b	2 regression coefficients (if any).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc — *NPARM*×*NPARM* variance-covariance matrix of coefficient estimates.

out.yDist — *L*×1 vector, percentages of dependent variable by category.

out.xData — *K*×4 matrix, the means, standard deviations, minimums, and maximums of independent variables.

`out.marginEffects`	$L\times1\times K$ array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	$L\times K\times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	$N\times1$ matrix of predicted (fitted) counts.
`out.resids`	$N\times1$ matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.

	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new ;
cls ;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "A";
d1.xnames =  "GPA" $| "TUCE" $| "PSI";

struct dcOut dcOut1;

dcOut1 = dcBinaryProbit("aldnel",d1,dcControlCreate());

call dcprt(dcOut1);
```

Source

dcbin.src

dcConditionalLogit

Purpose

Estimates the Conditional Logit model.

Library

dc

Format

out = **dcConditionalLogit**(*data, desc, cont*);

Input

`data`	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.	
`desc`	an instance of a `dcDesc` structure.	
	`desc.dataType`	scalar, if 1, the dataset contains a single row for each observation and attribute variables are stored in separate columns in that row. If 0, category data are stored by row within observation and attribute data are stored in single columns.
	`desc.yname`	name of dependent variable.
	`desc.yvar`	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either `desc.yname` or `desc.yvar` may be specified. If data is matrix of data, `desc.yvar` must be specified.
	`desc.ytype`	scalar, 0 if `desc.yvar` character variable, otherwise 1 if numeric. Default = 1.
	`desc.xnames`	$K \times 1$ string vector, names of the independent variable(s).
	`desc.xvars`	$K \times 1$ vector, indices of the independent variable(s). If data is name of **GAUSS** dataset, either `desc.xnames` or `desc.xvars` may be specified. If data is matrix of data, `desc.xvars` must be specified.
	`desc.catnames`	$L \times 1$ string vector, names of categories.
	`desc.refcat`	reference category. If `desc.refcatName` is specified, `desc.refcat` is optional. Default = 1.
	`desc.refcatName`	string, reference category name. If `desc.refcat` has been specified, `desc.refcatName` is optional. Default = `desc.catnames[1]`.
	`desc.atNames`	$P \times 1$ string vector, names of the attribute

		variable(s).	
	`desc.atVars`	P×1 numeric vector, indices of the attribute variable(s).	
	`desc.atCatNames`	P×L string array, names of the categories of attribute variable(s). Required if `desc.datatype` = 1 and `desc.atCatVars` not specified.	
	`desc.atCatVars`	P×L numeric vector, indices of the categories of attribute variable(s). Required if `desc.datatype` = 1 and `desc.atCatNames` not specified.	
	`desc.wgtname`	string, name of weight variable. If `desc.wgtvar` is specified, the specification of `desc.wgtname` is optional. Default = "".	
	`desc.wgtvar`	scalar, index of weight variable. If `desc.wgtname` is specified, the specification of `desc.wgtvar` is optional. Default = 0.	
	`desc.noconstant`	scalar, 1 if no constants in model. Default = 0.	
	`desc.marginType`	scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean `x`. Default = 0.	
`cont`	an instance of a `dcControl` structure.		
	`cont.startValues`	instance of `PV` structure containing starting values; if not provided, **`dcConditionalLogit`** computes start values.	
		`b0`	1 1×Lvector, constant in regression.
		`b`	2 K×L matrix, regression coefficients (if any). Coefficients associated with reference category are

dcConditionalLogit

		fixed to zero.
	gm	3 $M \times 1$ vector, coefficients of attribute variables.

For example:

```
struct dcControl cont;
cont = dcControlCreate;

b0 = { 0  1  1 };

b = { 0 .1 .1,
      0 .1 .1 };

gm = { .1,
       .1 };

mask = { 0  1  1,
         0  1  1,
         0  1  1 };

cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0",mask[1,.],1);
cont.startValues =
  pvPackmi(cont.startValues,
  b,"b",mask[2:3,.],2);
cont.startValues =
  pvPackmi(cont.startValues,
  gm,"gm",mask[1:2,2],3);
```

`cont.A`	$M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where p is a vector of the parameters. For more details. see Section 4.1.6.
`cont.B`	$M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where p is a vector of the parameters. For more details see Section 4.1.6.
`cont.C`	$M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where p is a vector of the parameters. For more details see Section 4.1.6.

cont.D	M×1 vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.

	`cont.randRadius`	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
	`cont.trustRadius`	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
	`cont.output`	scalar, if nonzero, optimization results are printed. Default = 0.
	`cont.printIters`	scalar, if nonzero, prints iteration information. Default = 0.

Output

`out` an instance of a `dcOut` structure

`out.par` instance of `PV` structure containing estimates.

`b0`	1 $L\times 1$ matrix, constant in regression.
`b`	2 $L\times K$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.
`gm`	3 M×1 vector, coefficients of attribute variables

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a

	single parameter vector:
	`b = pvGetParVector(out.par);`
	The location of the coefficients in *out.par* can be described by
	`b = pvGetParNames(out.par);`
	if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L \times 1$ vector, percentages of dependent variable by category.
out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
out.marginVC	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.atmargineffects	$L \times L \times 1 \times R$ array, marginal effects by category of attribute variables by category of dependent variable.
out.atmarginvc	$L \times L \times R \times R$ array, covariance matrices of marginal effects by category of attribute variables by category of dependent variable.
out.fittedVals	$N \times 1$ matrix of predicted (fitted) counts.
out.resids	$N \times 1$ matrix of residuals.
out.summaryStats	17×1 matrix of goodness-of-fit measures. 1 Log-Likelihood, full

		model.
	2	Log-Likelihood, restricted model (all slope coefficients equal zero.
	3	Degrees of freedom.
	4	Chi-square statistic.
	5	Number of Parameters.
	6	McFadden's Pseudo R-Squared.
	7	Madalla's Pseudo R-Squared.
	8	Cragg and Uhler's normed likelihood ratios statistics.
	9	Akaike information criterion (AIC).
	10	Bayesian information criterion (BIC).
	11	Hannon-Quinn Criterion.
	12	Count R-Squared.
	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "Mode";
d1.catvarname = "choiceno";
d1.catNames = "train"$|"bus"$|"car";
d1.atnames =  "ttme" $| "invc" $| "GC";

d1.noconstant = 1;

struct dcOut dcOut1;

dcOut1 = dcConditionalLogit("powersxie",d1,dcControlCreate());
call dcprt(dcOut1);
```

Source

`dcclogit.src`

dcMakeLogitNests

Purpose

Creates nests for nested logit regression.

Library

dc

Format

dcMakeLogitNests(*&cont*, *numberNests*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
numberNests	Scalar, number of nests to create.

Example

```
new;
cls;
library dc;

//Load data
loadm y=hensher_mat;

//Declare control structure
struct dcControl cont;

//Initialize dc control structure
cont = dcControlCreate();

//Step Two: Describe  data
//Name of dependent variable
dcSetYVar(&cont,y[.,1]);
dcSetYLabel(&cont,"Mode");

//Y Category Labels
dcSetYCategoryLabels(&cont,"Air,Train,Bus,Car");

//Specify reference category (excluded)
dcSetReferenceCategory(&cont, "Car");

//Set-up nested levels
dcMakeLogitNests(&cont,2);
```

Source

setnests.src

dcMultinomialLogit

Purpose

Estimates the Multinomial Logit model.

Library

dc

Format

out = **dcMultinomialLogit**(*data, desc, cont*);

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.	
desc	an instance of a *dcDesc* structure.	
	desc.yname	name of dependent variable.
	desc.yvar	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either *desc.yname* or *desc.yvar* may be specified. If data is matrix of data, *desc.yvar* must be specified.
	desc.ytype	scalar, 0 if *desc.yvar* character variable, otherwise 1 if numeric. Default = 1.
	desc.xnames	$K \times 1$ string vector, names of the independent variable(s).
	desc.xvars	$K \times 1$ vector, indices of the independent variable (s). If data is name of **GAUSS** dataset, either *desc.xnames* or *desc.xvars* may be specified. If data is matrix of data, *desc.xvars* must be specified.
	desc.catnames	$L \times 1$ string vector, names of categories.
	desc.refcat	reference category. If *desc.refcatName* is specified, *desc.refcat* is optional. Default =

1.

desc.refcatName string, reference category name. If *desc.refcat* has been specified, *desc.refcatName* is optional. Default = *desc.catnames[1]*.

desc.wgtname string, name of weight variable. If *desc.wgtvar* is specified, the specification of *desc.wgtname* is optional. Default = "".

desc.wgtvar scalar, index of weight variable. If *desc.wgtname* is specified, the specification of *desc.wgtvar* is optional. Default = 0.

desc.noconstant scalar, 1 if no constants in model. Default = 0.

desc.marginType scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean *x*. Default = 0.

cont an instance of a *dcControl* structure.

cont.startValues instance of *PV* structure containing starting values; if not provided, **dcMultinomialLogit** computes start values.

b0	1 $1{\times}L$constant in regression.
b	2 $K{\times}L$ regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate;

b0 = { 0  1  1 };

b = { 0 .1 .1,
      0 .1 .1 };

mask = { 0  1  1,
```

```
            0  1  1,
            0  1  1 };
cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0", mask[1,.], 1);
cont.startValues =
  pvPackmi(cont.startValues,
  b,"b", mask[2:3,.], 2);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C — $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — $M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

cont.inEqProc — scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a

	vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
`cont.bounds`	1×2 or *K*×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
`cont.maxIters`	scalar, maximum number of iterations. Default = 1e+5.
`cont.dirTol`	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **`sqpSolvemt`** exits the iterations.
`cont.feasibleTest`	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
`cont.randRadius`	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
`cont.trustRadius`	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
`cont.output`	scalar, if nonzero, optimization results are printed. Default = 0.
`cont.printIters`	scalar, if nonzero, prints iteration information. Default = 0.

Output

`out`	an instance of a `dcOut` structure	
	`out.par`	instance of `PV` structure containing estimates.

	b0	1 $L\times1$ matrix, constant in regression.
	b	2 $L\times K$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	$NPARM\times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L\times1$ vector, percentages of dependent variable by category.
out.xData	$K\times4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L\times1\times K$ array, marginal effects of independent variables by category of dependent variable.
out.marginVC	$L\times K\times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.

dcMultinomialLogit

`out.fittedVals`	N×1 matrix of predicted (fitted) counts.	
`out.resids`	N×1 matrix of residuals.	
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.	
	1	Log-Likelihood, full model.
	2	Log-Likelihood, restricted model (all slope coefficients equal zero.
	3	Degrees of freedom.
	4	Chi-square statistic.
	5	Number of Parameters.
	6	McFadden's Pseudo R-Squared.
	7	Madalla's Pseudo R-Squared.
	8	Cragg and Uhler's normed likelihood ratios statistics.
	9	Akaike information criterion (AIC).
	10	Bayesian information criterion (BIC).
	11	Hannon-Quinn Criterion.
	12	Count R-Squared.
	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "occatt";
d1.xnames =  "exper" $| "educ" $| "white";
d1.catnames =  "Menial" $| "BC" $| "craft" $| "WC" $| "Pro";

struct dcOut dcOut1;

dcOut1 = dcMultinomialLogit("gssocc",d1,dcControlCreate());

call dcprt(dcOut1);
```

Source

dcmnlogit.src

dcNegativeBinomial

Purpose

Estimates a negative binomial regression model.

Library

dc

Format

out = **dcNegativeBinomial**(*data, desc, cont*);

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.
desc	an instance of a *dcDesc* structure.

desc.yname	name of dependent variable.
desc.yvar	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either *desc.yname* or *desc.yvar* may be specified. If data is matrix of data, *desc.yvar* must be specified.
desc.xnames	K×1 string vector, names of the independent variable(s).
desc.xvars	K×1 vector, indices of the independent variable(s). If data is name of **GAUSS** dataset, either *desc.xnames* or *desc.xvars* may be specified. If data is matrix of data, *desc.xvars* must be specified.
desc.znames	L×1 string vector, names of the exogenous variable(s), if any, for zero-inflated model.
desc.zvars	K×1 vector, indices of the exogenous variable(s), if any, for zero-inflated model. If data is name of **GAUSS** dataset, either *desc.znames* or *desc.zvars* may be specified. If data is matrix of data, *desc.zvars* must be specified.
desc.timeName	string, name of variable for inclusion as a fixed exogenous log-variable. If *desc.timeVar* is specified, *desc.timeName* is optional.
desc.timeVar	reference category. If *desc.refcatName* is specified, *desc.refcat* is optional. Default = 1.
desc.wgtname	string, name of weight variable. If *desc.wgtvar* is specified, the specification of *desc.wgtname* is optional. Default = "".
desc.wgtvar	scalar, index of weight variable. If *desc.wgtname* is specified, the specification of *desc.wgtvar* is optional. Default = 0.
desc.limited	scalar, 0 - no censoring or truncation, 1 - truncated model, 2 - censored model.
desc.lh	scalar, value of left side truncation or censoring If the data are truncated on the left, all values must be greater than or equal to *desc.lh* (i.e. specify *desc.lh* = 1 if there are no zeros in the dependent variable). If the data are censored on the left, all values must be greater than or equal to *desc.lh*.
desc.rh	scalar value of right side truncation or censoring. If the data are truncated on the right, all values must be less than or equal to *desc.rh*.

If the data are censored on the left, all values must be less than or equal to *desc.rh*.

desc.zeroInflated scalar, if nonzero a zero-inflated model is estimated. Mixture probability can be a function of exogenous variables as specified in *desc.zvars*.

desc.marginType scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean *x*. Default = 0.

cont an instance of a *dcControl* structure.

cont.startValues instance of *PV* structure containing starting values; if not provided, **dcNegativeBinomial** computes start values.

b0	1 constant in regression.
b	2 regression coefficients (if any).
alpha	3 dispersion parameter.
p0	4 constant in zero-inflated model.
p	5 coefficients in zero-inflated model (if any

For example:

```
struct dcControl cont;
cont = dcControlCreate;

//Set parameter starting value matrices
b0 = .5;
b = { .1, .1, .1 };
a = .01;

//Pack parameter starting value matrices
cont.startValues =
  pvPacki(cont.startValues,
  b0,"b0",1);
cont.startValues =
  pvPacki(cont.startValues,
  b,"b",2);
cont.startValues =
  pvPacki(cont.startValues,
  a,"alpha",3);
```

cont.A $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more

	details. see Section 4.1.6.
cont.B	M×1 vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.C	M×K matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.D	M×1 vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **`sqpSolvemt`** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region

	not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out — an instance of a *dcOut* structure

out.par — instance of *PV* structure containing estimates.

b0	1 constant in regression.
b	2 regression coefficients (if any).
alpha	3 dispersion parameter.
p0	4 constant in zero-inflated model.
p	5 coefficients in zero-inflated model (if any

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc — *NPARM*×*NPARM* variance-covariance matrix of

dcNegativeBinomial

	coefficient estimates.
`out.yDist`	L×1 vector, percentages of dependent variable by category.
`out.xData`	K×4 matrix, the means, standard deviations, minimums, and maximums of independent variables.
`out.marginEffects`	L×1×K array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	L×K×K array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	N×1 matrix of predicted (fitted) counts.
`out.resids`	N×1 matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).

11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "ACC";
d1.xnames = "TB" $| "TC" $| "TD" $| "TE" $|
            "T6569" $| "T7074" $| "T7579" $| "O7579";
d1.timeName = "months";

struct dcOut dcOut1;

dcOut1 = dcNegativeBinomial("greenedata",d1,dcControlCreate());

call dcprt(dcOut1);
```

Source

dcnbin.src

dcNestedLogit

Purpose

Estimates the Conditional Logit model.

Library

dc

Format

out = **dcNestedLogit**(*data, desc, cont*);

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.	
desc	an instance of a `dcDesc` structure.	
	desc.dataType	scalar, if 1, the dataset contains a single row for each observation and attribute variables are stored in separate columns in that row. If 0, category data are stored by row within observation and attribute data are stored in single columns.
	desc.yname	name of dependent variable.
	desc.yvar	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either *desc.yname* or *desc.yvar* may be specified. If data is matrix of data, *desc.yvar* must be specified.
	desc.ytype	scalar, 0 if *desc.yvar* character variable, otherwise 1 if numeric. Default = 1.
	desc.xnames	$K \times 1$ string vector, names of the independent variable(s).
	desc.xvars	$K \times 1$ vector, indices of the independent variable (s). If data is name of **GAUSS** dataset, either

		`desc.xnames` or `desc.xvars` may be specified. If data is matrix of data, `desc.xvars` must be specified.
`desc.catnames`		L×1 string vector, names of categories.
`desc.refcat`		reference category. If `desc.refcatName` is specified, `desc.refcat` is optional. Default = 1.
`desc.refcatName`		string, reference category name. If `desc.refcat` has been specified, `desc.refcatName` is optional. Default = `desc.catnames[1]`.
`desc.level`		M×1 vector of instances of a dcLevel structure, one for each level of the model.
	`desc.level [m].catnames`	$L_m \times 1$string array, names of categories.
	`desc.level [m].atNames`	if `desc.datatype` = 0, $P_m \times 1$string vector, names of the attribute variable(s). If `desc.level [m].atVars` is specified, the specification of `desc.level [m].atNames` is optional.
	`desc.level [m].atVars`	if `desc.datatype` = 0, $P_m \times 1$numeric vector, indices of the attribute variable(s). If `desc.level [m].atNames` is specified, the specification of `desc.level [m].atVars` is optional.
	`desc.level [m].nests`	$L_m \times 1$vector, category number in the next higher

dcNestedLogit

			level of each category at this level. The highest category does not contain one.
		desc.level [m].atCatnames	$R_m \times L_m$ string array, L_m names of categories in **GAUSS** dataset of R_m attribute variables in level *m*. Required only if *desc.datayype* = 1. If *desc.level [m].atCatnames* is specified, the specification of *desc.level [m].atCatvars* is optional.
	desc.wgtname	string, name of weight variable. If *desc.wgtvar* is specified, the specification of *desc.wgtname* is optional. Default = "".	
	desc.wgtvar	scalar, index of weight variable. If *desc.wgtname* is specified, the specification of *desc.wgtvar* is optional. Default = 0.	
	desc.noconstant	scalar, 1 if no constants in model. Default = 0.	
	desc.marginType	scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean *x*. Default = 0.	
cont	an instance of a *dcControl* structure.		
	cont.startValues	instance of *PV* structure containing starting values; if not provided, **dcNestedLogit** computes start values.	
		b0	1 1×*L*vector, constant in regression.
		b	2 *K*×*L* matrix, regression coefficients (if any).

	Coefficients associated with reference category are fixed to zero.
g1	3 $R_1 \times 1$ vector, coefficients of attribute variables for first level.
g2	4 $R_2 \times 1$ vector, coefficients of attribute variables for second level.
	. . .
gM	2+M $R_M \times 1$ vector, coefficients of attribute variables for M-th level.
t2	3+M $L_2 \times 1$ vector, proportionality coefficients for second level (first level does not have these coefficients).
t3	4+M $L_3 \times 1$ vector, proportionality coefficients for third level (first level does not have these coefficients).
	. . .
tM	2M+1 $L_M \times 1$ vector, proportionality coefficients for M-th level (first level does not have these coefficients).

For example:

```
struct dcControl cont;
cont = dcControlCreate();

//Set fourth category
```

dcNestedLogit

```
//as reference category
mask = { 1  1  1 0,
         1  1  1 0,
         1  1  1 0};

//Intercepts for four
//categories at first level
b0 = { 1  1  1 0};
cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0",mask[1,.],1);

//Two attribute variables
//at first level
g1 = { .1,
       .1 };
cont.startValues =
  pvPackmi(cont.startValues,
  g1,"g1",mask[1:2,2],3);

//One attribute variable
//at second level
g2 = { .1 };
cont.startValues =
  pvPackmi(cont.startValues,
  g2,"g2",mask[1,3],4);

//Two category interaction
//terms
t2 = { .1,
       .1 };
cont.startValues =
  pvPackmi(cont.startValues,
  t2,"t2",mask[1:2,2],5);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where p is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where p is a vector of the parameters. For more details see Section 4.1.6.

cont.C — $M \times K$ matrix, linear inequality constraint

	coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.D	M×1 vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line

	search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out	an instance of a *dcOut* structure		
	out.par	instance of *PV* structure containing estimates.	
		b0	1 $1 \times L$ vector, constant in regression.
		b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zero.
		g1	3 $R_1 \times 1$ vector, coefficients of attribute variables for first level.

g2	4 $R_2 \times 1$ vector, coefficients of attribute variables for second level.
	. . .
gM	2+M $R_M \times 1$ vector, coefficients of attribute variables for M-th level.
t2	3+M $L_2 \times 1$ vector, proportionality coefficients for second level (first level does not have these coefficients).
t3	4+M $L_3 \times 1$ vector, proportionality coefficients for third level (first level does not have these coefficients).
	. . .
tM	2M+1 $L_M \times 1$ vector, proportionality coefficients for M-th level (first level does not have these coefficients).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector
(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames
(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	*NPARM*×*NPARM* variance-covariance matrix of coefficient estimates.
out.yDist	*L*×1 vector, percentages of dependent variable by category.
out.xData	*K*×4 matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	*L*×1×*K* array, marginal effects of independent variables by category of dependent variable.
out.marginVC	*L*×*K*×*K* array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.atmargineffects	*M*×1 *DS* structure containing $L_m \times L_m \times 1 \times R_m$ arrays, marginal effects by category of attribute variables by categories at the

	m-th level.
`out.atmarginvc`	M×1 `DS` structure containing $L_m \times L_m \times R_m \times R_m$ arrays, covariance matrices of marginal effects by category of attribute variables by category of the m-th level.
`out.fittedVals`	N×1 matrix of predicted (fitted) counts.
`out.resids`	N×1 matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.

9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.level = reshape(d1.level,2,1);

d1.yname = "Mode";
d1.catnames = "Air"$|"Train"$|"Bus"$|"Car";
d1.refcatName =  "Car";

d1.level[1].atNames = "TTME"$|"GC";
d1.level[1].nests = { 1, 2, 2, 2 };
```

```
d1.level[2].catNames = "Fly"$|"Ground";
d1.level[2].atNames = "airhinc";

struct  dcout dcout1;

dcOut1 = dcNestedLogit("hensher",d1,dcControlCreate());
call dcprt(dcOut1);
```

Source

dcnlogit.src

dcOrderedLogit

Purpose

Estimates an ordered logit regression model.

Library

dc

Format

out = **dcOrderedLogit**(*data, desc, cont*);

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.	
desc	an instance of a *dcDesc* structure.	
	desc.dataType	scalar, if 1, the dataset contains a single row for each observation and attribute variables are stored in separate columns in that row. If 0, category data are stored by row within observation and attribute data are stored in single columns.
	desc.yname	name of dependent variable.

	`desc.yvar`	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either `desc.yname` or `desc.yvar` may be specified. If data is matrix of data, `desc.yvar` must be specified.
	`desc.ytype`	scalar, 0 if `desc.yvar` character variable, otherwise 1 if numeric. Default = 1.
	`desc.xnames`	K×1 string vector, names of the independent variable(s).
	`desc.xvars`	K×1 vector, indices of the independent variable (s). If data is name of **GAUSS** dataset, either `desc.xnames` or `desc.xvars` may be specified. If data is matrix of data, `desc.xvars` must be specified.
	`desc.wgtname`	string, name of weight variable. If `desc.wgtvar` is specified, the specification of `desc.wgtname` is optional. Default = "".
	`desc.wgtvar`	scalar, index of weight variable. If `desc.wgtname` is specified, the specification of `desc.wgtvar` is optional. Default = 0.
	`desc.catnames`	L×1 string vector, names of categories.
	`desc.marginType`	scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean `x`. Default = 0.
`cont`	an instance of a `dcControl` structure.	
	`cont.startValues`	instance of `PV` structure containing starting values; if not provided, **`dcOrderedLogit`** computes start values.

`tau`	1 thresholds.
`b`	2 regression coefficients (if any).

For example:

```
struct dcControl cont;
cont = dcControlCreate;
```

```
tau = { -5, -2 };
b = { .1, .1, .1 };
cont.startValues =
  pvPacki(cont.startValues,
  tau,"tau",1);
cont.startValues =
  pvPacki(cont.startValues,
  b,"b",2);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C — $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — $M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

cont.inEqProc — scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS*

dcOrderedLogit

	data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
`cont.bounds`	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
`cont.maxIters`	scalar, maximum number of iterations. Default = 1e+5.
`cont.dirTol`	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **`sqpSolvemt`** exits the iterations.
`cont.feasibleTest`	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
`cont.randRadius`	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
`cont.trustRadius`	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
`cont.output`	scalar, if nonzero, optimization results are printed. Default = 0.
`cont.printIters`	scalar, if nonzero, prints iteration information. Default = 0.

Output

`out`	an instance of a `dcOut` structure

out.par	instance of *PV* structure containing estimates.
	tau — 1 thresholds.
	b — 2 regression coefficients (if any).
	To retrieve, e.g., regression coefficients: `b = pvUnpack(out.par,"b");` or `b = pvUnpack(out.par,2);` The coefficients may also be retrieved as a single parameter vector: `b = pvGetParVector(out.par);` The location of the coefficients in *out.par* can be described by `b = pvGetParNames(out.par);` if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.
out.vc	*NPARM*×*NPARM* variance-covariance matrix of coefficient estimates.
out.yDist	*L*×1 vector, percentages of dependent variable by category.
out.xData	*K*×4 matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	*L*×1×*K* array, marginal effects of independent variables by category of dependent variable.
out.marginVC	*L*×*K*×*K* array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.fittedVals	*N*×1 matrix of predicted (fitted) counts.

`out.resids`	N×1 matrix of residuals.	
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.	
	1	Log-Likelihood, full model.
	2	Log-Likelihood, restricted model (all slope coefficients equal zero.
	3	Degrees of freedom.
	4	Chi-square statistic.
	5	Number of Parameters.
	6	McFadden's Pseudo R-Squared.
	7	Madalla's Pseudo R-Squared.
	8	Cragg and Uhler's normed likelihood ratios statistics.
	9	Akaike information criterion (AIC).
	10	Bayesian information criterion (BIC).
	11	Hannon-Quinn Criterion.
	12	Count R-Squared.
	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "ABC";
d1.xnames = "GPA"$|"TUCE"$|"PSI";

struct  dcOut dcOut1;

dcOut1 = dcOrderedLogit("aldnel",d1,dcControlCreate());

call dcprt(dcOut1);
```

Source

dcord.src

dcOrderedProbit

Purpose

Estimates an ordered probit regression model.

Library

dc

Format

out = **dcOrderedProbit**(*data*, *desc*, *cont*);

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.

`desc`	an instance of a `dcDesc` structure.	
	`desc.yname`	name of dependent variable.
	`desc.yvar`	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either `desc.yname` or `desc.yvar` may be specified. If data is matrix of data, `desc.yvar` must be specified.
	`desc.ytype`	scalar, 0 if `desc.yvar` character variable, otherwise 1 if numeric. Default = 1.
	`desc.xnames`	K×1 string vector, names of the independent variable(s).
	`desc.xvars`	K×1 vector, indices of the independent variable (s). If data is name of **GAUSS** dataset, either `desc.xnames` or `desc.xvars` may be specified. If data is matrix of data, `desc.xvars` must be specified.
	`desc.catnames`	L×1 string vector, names of categories.
	`desc.wgtname`	string, name of weight variable. If `desc.wgtvar` is specified, the specification of `desc.wgtname` is optional. Default = "".
	`desc.wgtvar`	scalar, index of weight variable. If `desc.wgtname` is specified, the specification of `desc.wgtvar` is optional. Default = 0.
	`desc.marginType`	scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean `x`. Default = 0.
`cont`	an instance of a `dcControl` structure.	
	`cont.startValues`	instance of `PV` structure containing starting values; if not provided, **`dcOrderedProbit`** computes start values. `tau` 1 thresholds. `b` 2 regression coefficients (if any). For example:

```
struct dcControl cont;
cont = dcControlCreate();
tau = { -5, -2 };
b = { .1, .1, .1 };
cont.startValues =
  pvPacki(cont.startValues,
  tau,"tau",1);
cont.startValues =
  pvPacki(cont.startValues,
  b,"b",2);
```

cont.A	$M{\times}K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.
cont.B	$M{\times}1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.C	$M{\times}K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.D	$M{\times}1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input

	arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or *K*×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out	an instance of a *dcOut* structure		
	out.par	instance of *PV* structure containing estimates.	
		tau	1 thresholds.
		b	2 regression coefficients (if any).

		To retrieve, e.g., regression coefficients:
		`b = pvUnpack(out.par,"b");`
		or
		`b = pvUnpack(out.par,2);`
		The coefficients may also be retrieved as a single parameter vector:
		`b = pvGetParVector(out.par);`
		The location of the coefficients in *out.par* can be described by
		`b = pvGetParNames(out.par);`
		if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.	

	out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.	
	out.yDist	$L \times 1$ vector, percentages of dependent variable by category.	
	out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.	
	out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.	
	out.marginVC	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.	

dcOrderedProbit

`out.fittedVals`	N×1 matrix of predicted (fitted) counts.	
`out.resids`	N×1 matrix of residuals.	
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.	
	1	Log-Likelihood, full model.
	2	Log-Likelihood, restricted model (all slope coefficients equal zero.
	3	Degrees of freedom.
	4	Chi-square statistic.
	5	Number of Parameters.
	6	McFadden's Pseudo R-Squared.
	7	Madalla's Pseudo R-Squared.
	8	Cragg and Uhler's normed likelihood ratios statistics.
	9	Akaike information criterion (AIC).
	10	Bayesian information criterion (BIC).
	11	Hannon-Quinn Criterion.
	12	Count R-Squared.
	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "ABC";
d1.xnames =  "GPA" $| "TUCE" $| "PSI";

struct dcOut dcOut1;

dcOut1 = dcOrderedProbit("aldnel",d1,dcControlCreate());

call dcprt(dcOut1);
```

Source

`dcord.src`

dcPoisson

Purpose

Estimates a Poisson regression model.

Library

dc

Format

out = **dcPoisson**(*data, desc, cont*);

Input

data	string or $N{\times}K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.

`desc`	an instance of a `dcDesc` structure.	
	`desc.yname`	name of dependent variable.
	`desc.yvar`	scalar, index of dependent variable. If data is name of **GAUSS** dataset, either `desc.yname` or `desc.yvar` may be specified. If data is matrix of data, `desc.yvar` must be specified.
	`desc.xnames`	$K \times 1$ string vector, names of the independent variable(s).
	`desc.xvars`	$K \times 1$ vector, indices of the independent variable (s). If data is name of **GAUSS** dataset, either `desc.xnames` or `desc.xvars` may be specified. If data is matrix of data, `desc.xvars` must be specified.
	`desc.znames`	$L \times 1$ string vector, names of the exogenous variable(s), if any, for zero-inflated model.
	`desc.zvars`	$K \times 1$ vector, indices of the exogenous variable (s), if any, for zero-inflated model. If data is name of **GAUSS** dataset, either `desc.znames` or `desc.zvars` may be specified. If data is matrix of data, `desc.zvars` must be specified.
	`desc.timeName`	string, name of variable for inclusion as a fixed exogenous log-variable. If `desc.timeVar` is specified, `desc.timeName` is optional.
	`desc.timeVar`	string, index of variable for inclusion as a fixed exogenous log-variable. If `desc.timeName` is specified, `desc.timeVar` is optional.
	`desc.wgtname`	string, name of weight variable. If `desc.wgtvar` is specified, the specification of `desc.wgtname` is optional. Default = "".
	`desc.wgtvar`	scalar, index of weight variable. If `desc.wgtname` is specified, the specification of `desc.wgtvar` is optional. Default = 0.
	`desc.limited`	scalar, 0 - no censoring or truncation, 1 - truncated model, 2 - censored model.

	desc.lh	scalar, value of left side truncation or censoring. If the data are truncated on the left, all values must be greater than or equal to *desc.lh* (i.e., specify *desc.lh* = 1 if there are no zeros in the dependent variable). If the data are censored on the left, all values must be greater than or equal to *desc.lh*.
	desc.rh	scalar value of right side truncation or censoring. If the data are truncated on the right, all values must be greater than or equal to *desc.rh* (i.e., specify *desc.rh* = 1 if there are no zeros in the dependent variable). If the data are censored on the right, all values must be greater than or equal to *desc.rh*.
	desc.zeroInflated	scalar, if nonzero a zero-inflated model is estimated. Mixture probability can be a function of exogenous variables as specified in
	desc.marginType	scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean *x*. Default = 0.
cont	an instance of a *dcControl* structure.	
	cont.startValues	instance of *PV* structure containing starting values; if not provided, **dcPoisson** computes start values. *b0* — 1 constant in regression. *b* — 2 regression coefficients (if any). *p0* — 3 constant in zero-inflated model. *p* — 4 coefficients in zero-inflated model. For example:

```
struct dcControl cont;
cont = dcControlCreate;
b0 = { .5 };
```

dcPoisson

```
b = { .1, .1, .1 };
cont.startValues =
  pvPacki(cont.startValues,
  b0,"b0",1);
cont.startValues =
  pvPacki(cont.startValues,
  b,"b",2);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C — $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — $M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

cont.inEqProc — scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For

	more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **sqpSolvemt** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out	an instance of a *dcOut* structure	
	out.par	instance of *PV* structure containing estimates.

	b0	1 constant in regression.
	b	2 regression coefficients (if any).
	p0	3 constant in zero-inflated model.
	p	4 coefficients in zero-inflated model.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L \times 1$ vector, percentages of dependent variable by category.
out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
out.marginVC	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.

`out.fittedVals`	N×1 matrix of predicted (fitted) counts.	
`out.resids`	N×1 matrix of residuals.	
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.	
	1	Log-Likelihood, full model.
	2	Log-Likelihood, restricted model (all slope coefficients equal zero.
	3	Degrees of freedom.
	4	Chi-square statistic.
	5	Number of Parameters.
	6	McFadden's Pseudo R-Squared.
	7	Madalla's Pseudo R-Squared.
	8	Cragg and Uhler's normed likelihood ratios statistics.
	9	Akaike information criterion (AIC).
	10	Bayesian information criterion (BIC).
	11	Hannon-Quinn Criterion.
	12	Count R-Squared.
	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yname = "ACC";
d1.xnames =  "TB" $| "TC" $| "TD" $| "TE" $| "T6569" $| "T7074" $|
"T7579" $| "O7579";
d1.timeName = "months";

struct dcOut dcOut1;

dcOut1 = dcPoisson("greenedata",d1,dcControlCreate());

call dcprt(dcOut1);
```

Source

dcpsn.src

dcprt

Purpose

Prints output from **Discrete Choice Analysis Tools 2.0** procedures.

Library

dc

Format

out = **dcprt**(*out*);

Input

out	an instance of a `dcOut` structure.

Output

out	an instance of a `dcOut` structure.

Remarks

The input argument is returned unchanged.

Source

`dc.src`

dcScale

Purpose

Used for data pre-scaling

Library

dc

Format

x_scaled = **dcScale**(*data*, *method*);

Input

data	Matrix, data to be rescaled
method	Scalar, indicator of scaling method to be implemented: [1] Z-Score Normalization [2] [0,1] Min/Max Normalization [3] Scale by 1/sqrt(k) where k=number features [4] Center data [5] Sigmoidal scale

Output

x_scaled	Matrix, scaled data

Example

```
new;
cls;
library dc;
x = rndn(10,5);
x_scaled = dcScale(x,1);

print "x : " x ;
print ;
print  "x_scaled : "  x_scaled;
```

Source

logisticRegress.src

dcSetAttributeLabels

Purpose

Used to load labels for attributes variables for use in **DC** modeling procedures.

Library

dc

Format

dcSetAttributeLabels(*&cont*, *atLabels*);

Input

&cont	Pointer to an instance of a *dcControl* structure.

atLabels	String array, labels or **GAUSS** data set name of attribute variables.

Example

```
new;
cls;
library dc;

//Load data
loadm y = powersxie_mat;

//Declare control structure
struct dcControl cont;

//Load attribute label
dcSetAttributeLabels(&cont,"ttme,invc,invt,GC");
```

Remarks

The **dcSetAttributeLabels** procedure can be used to load data if a **GAUSS** data set has been loaded for use.

Source

setdesc.src

dcSetAttributeVars

Purpose

Used to load a matrix of attribute variables for use in **DC** modeling procedures.

Library

dc

Format

dcSetAttributeVars(*&cont*, *atVars*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
atVars	Matrix, N x K observations of attribute data.

Example

```
new;
cls;
library dc;

//Load data
loadm y = powersxie_mat;

//Declare control structure
struct dcControl cont;

//Load attributes variable
dcSetAttributeVars(&cont,y[.,3:6]);
```

Remarks

The **dcSetAttributeVars** procedure must be used to load attribute data if working directly from a data matrix. It is not required if a **GAUSS** data set has been loaded for use.

Source

setdesc.src

dcSetCategoryVar

Purpose

Used to load category data for analysis if category data is stored separately from dependent data.

Library

dc

Format

dcSetCategoryVar(*&cont*, *categoryVar*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
categoryVar	Matrix, data matrix to be used as category variable.

Example

```
new;
cls;
library dc;

//Load data
loadm y = powersxie_mat;

//Declare control structure
struct dcControl cont;

//Load dependent categories label
dcSetCategoryVar(&cont,y[.,1]);
```

Source

setdesc.src

dcSetConstant

Purpose

Used to turn constant on or off for **DC** modeling procedures. Default on.

Library

dc

Format

dcSetConstant(*&cont*, *setConstant*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
setConstant	String, constant setting either *on* or *off*.

Example

```
new;
cls;
library dc;

//Load data
loadm y = greenedata_mat;

//Declare control structure
struct dcControl cont;

//Load independent label
dcSetConstant(&cont,"off");
```

Source

setdesc.src

dcSetDataset

Purpose

Used to set **GAUSS** dataset name for use in modeling.

Library

dc

Format

dcSetDataset(*&cont*, *datasetName*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
datasetName	String, **GAUSS** data set name.

Example

```
new;
cls;
library dc;
struct dcControl cont;
dcSetDataSet(&cont,"aldnel");
```

Source

setdesc.src

dcSetLogitNestAttributes

Purpose

Used to sort attributes into previously created nests for nested logit model.

Library

dc

Format

dcSetLogitNestAttributes(*&cont*, *nestNumber*, *attributeList*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
nestNumber	Scalar, nest level.
attributeList	String Array, M x 1, list of nest specific attributes.

Example

```
new;
cls;
library dc;

//Load data
loadm y = hensher_mat;

//Step One: Declare dc control structure
struct dcControl cont;
//Initialize dc control structure
cont = dcControlCreate();

//Step Two: Describe  data
//Name of dependent variable
dcSetYVar(&cont,y[.,1]);
dcSetYLabel(&cont,"Mode");

//Y Category Labels
dcSetYCategoryLabels(&cont,"Air,Train,Bus,Car");
//Specify reference category (excluded)
dcSetReferenceCategory(&cont, "Car");

//Name of independent variable
varlist = "TTME,GC,AIRHINC";
dcSetAttributeVars(&cont,y[.,2]~y[.,5]~y[.,8]);
dcSetAttributeLabels(&cont,"TTME,GC,AIRHINC");

//Set-up nested levels
dcMakeLogitNests(&cont,2);
```

```
//Set attributes and categories for lower nest (Nest One)
dcSetLogitNestAttributes(&cont,1,"TTME,GC");
dcSetLogitNestCategories(&cont,1,"Air,Train,Bus,Car");

//Set attributes and categories for lower nest (Nest Two)
dcSetLogitNestAttributes(&cont,2,"AIRHINC");
dcSetLogitNestCategories(&cont,2,"Fly,Ground");
```

Remark

Prior to using **dcSetLogitNestAttributes** nests must be created using **dcMakeLogitNests**.

Source

`setnests.src`

dcSetLogitNestCategories

Purpose

Used to specify outcome categories within a nest.

Library

dc

Format

dcSetLogitNestCategories(*&cont*, *nestNumber*, *attributeList*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
nestNumber	Scalar, nest level.
categories	String Array, M x 1, list of categorical outcomes of dependent data within a specified nest.

Example

```
new;
cls;
library dc;

//Load data
loadm y=hensher_mat;

//Step One: Declare dc control structure
struct dcControl cont;
//Initialize dc control structure
cont = dcControlCreate();

//Step Two: Describe  data
//Name of dependent variable
dcSetYVar(&cont,y[.,1]);
dcSetYLabel(&cont,"Mode");

//Y Category Labels
dcSetYCategoryLabels(&cont,"Air,Train,Bus,Car");
//Specify reference category (excluded)
dcSetReferenceCategory(&cont, "Car");

//Name of independent variable
varlist = "TTME,GC,AIRHINC";
dcSetAttributeVars(&cont,y[.,2]~y[.,5]~y[.,8]);
dcSetAttributeLabels(&cont,"TTME,GC,AIRHINC");

//Set-up nested levels
dcMakeLogitNests(&cont,2);

//Set attributes and categories for lower nest (Nest One)
dcSetLogitNestAttributes(&cont,1,"TTME,GC");
dcSetLogitNestCategories(&cont,1,"Air,Train,Bus,Car");

//Set attributes and categories for lower nest (Nest Two)
dcSetLogitNestAttributes(&cont,2,"AIRHINC");
dcSetLogitNestCategories(&cont,2,"Fly,Ground");
```

Remark

Prior to using **dcSetLogitNestCategories**, nests must be created using **dcMakeLogitNests** and attributes must be placed in nests using

dcSetLogitNestAttributes.

Source

setnests.src

dcSetReferenceCategory

Purpose

Sets dependent variable category to be omitted from estimation as a reference category.

Library

dc

Format

dcSetReferenceCategory(*&cont*, *refCategory*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
refCategory	String or matrix index, either variable label or index.

Example

```
new;
cls;
library dc;

//Load data
loadm y = gssocc_mat;

//Declare control structure
struct dcControl cont;

//Dependent variable categories
dcSetYCategoryLabels(&cont,"Menial,BC,Craft,WC,Pro");
```

```
//Set reference label
dcSetReferenceCategory(&cont,"Menial");
```

Remark

Prior to using **dcSetReferenceCategory** the dependent variable category names must be set using **dcSetYCategoryLabels**.

Source

setdesc.src

dcSetTimeLabel

Purpose

Used to load labels for independent variables for use in **DC** modeling procedures.

Library

dc

Format

dcSetTimeLabel(*&cont*, *tLabel*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
tLabel	String, label or **GAUSS** data set name of time variable.

Example

```
new;
cls;
library dc;
```

```
//Load data
loadm y = greenedata_mat;

//Declare control structure
struct dcControl cont;

//Load independent label
dcSetTimeLabel(&cont,"month");
```

Remarks

The **dcSetTimeLabel** procedure can be used to load data if a **GAUSS** data set has been loaded for use.

Source

setdesc.src

dcSetTimeVar

Purpose

Used to load time variable for use in **DC** count modeling procedures.

Library

dc

Format

dcSetTimeVar(*&cont*, *tVar*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
tVar	Matrix, N x 1 observations of time variable.

Example

```
new;
cls;
library dc;

//Load data
loadm y = gssocc_mat;

//Declare control structure
struct dcControl cont;

//Load time variable
dcSetTimeVar(&cont,y[.,8]);
```

Remarks

The **dcSetTimeVar** procedure must be used to load data if working directly from a data matrix. It is not required if a **GAUSS** data set has been loaded for use.

Source

setdesc.src

dcSetWeightLabels

Purpose

Sets which independent variables, if any, to be weighted during analysis.

Library

dc

Format

dcSetWeightLabels(*&cont*, *wLabels*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
wLabels	String, variable labels.

Example

```
new;
cls;
library dc;

//Load data
loadm y = hensher_mat;

//Declare control structure
struct dcControl cont;

//Name of independent variable
dcSetAttributeVars(&cont,y[.,2]~y[.,5]~y[.,8]);
dcSetAttributeLabels(&cont,"TTME,GC,AIRHINC");

//Set weighted variables
dcSetWeightLabels(&cont,"TTME");
```

Remarks

Note that the independent variable labels must be set prior to setting weight variables.

Source

setdesc.src

dcSetXLabels

Purpose

Used to load labels for independent variables for use in **DC** modeling procedures.

Library

dc

Format

```
dcSetXLabels(&cont, xLabels);
```

Input

&cont	Pointer to an instance of a *dcControl* structure.
xLabels	String array, labels or **GAUSS** data set name of independent variable.

Example

```
new;
cls;
library dc;

//Load data
loadm y = gssocc_mat;

//Declare control structure
struct dcControl cont;

//Load independent label
dcSetXLabels(&cont,"TUCE,GPA,PSI");
```

Remarks

The **dcSetXLabels** procedure can be used to load data if a **GAUSS** data set has been loaded for use.

Source

setdesc.src

dcSetXVars

Purpose

Used to load a matrix of independent variables for use in **DC** modeling procedures.

Library

dc

Format

dcSetXVars(*&cont*, *x*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
x	Matrix, N x K matrix of observations of independent variables.

Example

```
new;
cls;
library dc;

//Load data
loadm y = gssocc_mat;

//Declare control structure
struct dcControl cont;

//Load dependent variable
dcSetXVars(&cont,y[.,2:4]);
```

Remarks

The **dcSetXVars** procedure must be used to load data if working directly from a data matrix. It is not required if a **GAUSS** data set has been loaded for use.

Source

setdesc.src

dcSetYCategoryLabels

Purpose

Used to load category labels for dependent variable for use in **DC** modeling procedures.

Library

dc

Format

dcSetYCategoryLabels(*&cont*, *yCategoryLabel*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
yCategoryLabel	String array, category labels of dependent variable.

Example

```
new;
cls;
library dc;

//Load data
loadm y = gssocc_mat;

//Declare control structure
struct dcControl cont;

//Load dependent categories label
dcSetYCategoryLabels(&cont,"occ");
```

Source

`setdesc.src`

dcSetYLabel

Purpose

Used to load label for dependent variable for use in **DC** modeling procedures.

Library

dc

Format

dcSetYLabel(*&cont*, *yLabel*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
yLabel	String, label or **GAUSS** data set name of dependent variable.

Example

```
new;
cls;
library dc;

//Load data
loadm y = gssocc_mat;

//Declare control structure
struct dcControl cont;

//Load dependent label
dcSetYLabel(&cont,"mode");
```

dcSetYLabel

Remarks

The **dcSetYLabel** procedure can be used to load data if a **GAUSS** data set has been loaded for use.

Source

setdesc.src

dcSetYVar

Purpose

Used to load a vector of dependent variables for use in **DC** modeling procedures.

Library

dc

Format

dcSetYVar(*&cont*, *y*);

Input

&cont	Pointer to an instance of a *dcControl* structure.
y	Matrix, N x 1 vector of observations of dependent variable.

Example

```
new;
cls;
library dc;

//Load data
loadm y = gssocc_mat;

//Declare control structure
```

```
struct dcControl cont;

//Load dependent variable
dcSetYVar(&cont,y[.,1]);
```

Remarks

The **dcSetYVar** procedure must be used to load data if working directly from a data matrix. It is not required if a **GAUSS** data set has been loaded for use.

Source

`setdesc.src`

dcStereo

Purpose

Estimates the Stereotype Multinomial Logit model.

Library

dc

Format

out = **dcStereo**(*data, desc, cont*);

Input

data	string or $N \times K$ matrix, if string, the name of a **GAUSS** data set or if matrix, matrix of data.
desc	an instance of a *dcDesc* structure.
	desc.yname — name of dependent variable.
	desc.yvar — scalar, index of dependent variable. If data is name of **GAUSS** dataset, either *desc.yname* or *desc.yvar* may be specified. If data is matrix

		of data, `desc.yvar` must be specified.
	`desc.ytype`	scalar, 0 if `desc.yvar` character variable, otherwise 1 if numeric. Default = 1.
	`desc.xnames`	K×1 string vector, names of the independent variable(s).
	`desc.xvars`	K×1 vector, indices of the independent variable (s). If data is name of **GAUSS** dataset, either `desc.xnames` or `desc.xvars` may be specified. If data is matrix of data, `desc.xvars` must be specified.
	`desc.catnames`	L×1 string vector, names of categories.
	`desc.refcat`	reference category. If `desc.refcatName` is specified, `desc.refcat` is optional. Default = 1.
	`desc.refcatName`	string, reference category name. If `desc.refcat` has been specified, `desc.refcatName` is optional. Default = `desc.catnames[1]`
	`desc.wgtname`	string, name of weight variable. If `desc.wgtvar` is specified, the specification of `desc.wgtname` is optional. Default = "".
	`desc.wgtvar`	scalar, index of weight variable. If `desc.wgtname` is specified, the specification of `desc.wgtvar` is optional. Default = 0.
	`desc.noconstant`	scalar, 1 if no constants in model. Default = 0.
	`desc.marginType`	scalar, 1 - average partial probability with respect to independent variables; 0 - partial probability with respect to mean `x`. Default = 0.
`cont`	an instance of a `dcControl` structure.	
	`cont.startValues`	instance of `PV` structure containing starting values; if not provided, **`dcStereo`** computes start values.
		`b0` — 1 1×L vector, constants in regression.

b 2 K×1 vector, regression coefficients.

For example:

```
struct dcControl cont;
cont = dcControlCreate;
b0 = 1;
b = { .1, .2 };
d = .01;
cont.startValues =
  pvPacki(cont.startValues,
  b0,"b0",1);
cont.startValues =
  pvPacki(cont.startValues,
  b,"b",2);
cont.startValues =
  pvPacki(cont.startValues,
  d,"distance",3);
```

cont.A — M×K matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — M×1 vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C — M×K matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — M×1 vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a

	vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or *K*×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **resolvent** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are

	printed. Default = 0.
`cont.printIters`	scalar, if nonzero, prints iteration information. Default = 0.

Output

out — an instance of a *dcOut* structure

out.par — instance of *PV* structure containing estimates.

b0	1 constant in regression.
b	2 regression coefficients.
distance	3 distance coefficients.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc — $NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.

out.yDist — $L \times 1$ vector, percentages of dependent variable by category.

out.xData — $K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.

dcStereo

dcStereo

`out.marginEffects`	$L\times1\times K$ array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	$L\times K\times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	$N\times1$ matrix of predicted (fitted) counts.
`out.resids`	$N\times1$ matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.

	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

struct dcDesc d1;
d1 = dcDescCreate();

d1.yVar = 1;
d1.xVars = { 2,3,4 };

struct dcOut dcOut1;

dcOut1 = dcStereo("aldnel",d1,dcControlCreate());

call dcprt(dcOut1);
```

Source

`dcstereo.src`

logisticRegress

Purpose

Perform linear classification using logistic regression.

Library

dc

Format

out = **logisticRegress**(*lCtl*, *y*, *x*);

Input

lCtl	an instance of a *lrControl* structure:		
	lctl.solverType	Matrix, scalar indicator of classification problem. If non-scalar, *crossValidation* must be non-zero and will select the highest cv-accuracy solver:	
		0	L2-regularized logistic regression,
		1	L2-regularized logistic regression, L2 loss SVC dual,
		2	L2-regularized logistic regression, L2 loss SVC,
		3	L2-regularized logistic regression, L1 loss SVC dual,
		4	MCSVM CS,
		5	L1R, L2 loss SVC,
		6	L1R, logistic regression,
		7	L2R, logistic regression dual,
		11	L2R, L2 loss SVR regression model,
		12	L2 loss SVR regression model,
		13	L2R, L1 loss SVR dual
	lctl.eps		Scalar, the stopping condition for KKT approximation algorithm.
	lctl.C		Matrix, loss function penalty parameter. If non-scalar, *crossValidation* must be non-zero and the highest cross-validation accuracy is used to select optimal C. Values of C<0 are not permissible. Default=1.
	lctl.p		Scalar, loss function tolerance.
	lctl.bias		Scalar, 0 or 1. If set to 1, a bias

		feature will be added to the end of the incoming 'x' matrix. This bias feature will be a vector of ones. Default=1.
lctl.crossValidation		Scalar, specifies number of folds for k-fold cross-validation. If equal to 0 no cross-validation.
lctl.predict		Scalar, indicator variable to conduct post-estimation prediction. Default=0.
lctl.plotPredict		Scalar, indicator variable to plot post-estimation predictions. Default=0.
lctl.printOutput		Scalar, indicator variable to print output to screen. Default=1.
lctl.scaleX		Scalar, indicator parameter for pre-estimation data scaling method. Default=2.
	0	No scaling,
	1	Z-Score normalization,
	2	[0,1] Min/Max normalization. [Default]
	3	Scale by 1/sqrt(k) where k=number features.
	4	Center data.
	5	Sigmoidal scale.

Output

`lrOut.weights`	Matrix, estimated weights for specified independent variables.
`lrOut.yPredict`	Matrix, predicted observations using estimated weights and data matrix.
`lrOut.probability`	Matrix, probabilities from logistic regression used to for determining predicted y classifications.

lrOut.cvAccuracy	Matrix, cross-validation prediction accuracy.
lrOut.predictionAccuracy	Scalar, full sample prediction accuracy.
lrOut.optimalC	Scalar, optimal C based on highest cross-validation accuracy.
lrOut.optimalSolver	Scalar, optimal solver based on highest cross-validation accuracy.

Example

```
new;
cls;
library dc;

load diabetes = "diabetes.fmt";
y = diabetes[.,9];
x = diabetes[.,1:8];

//Declare and lrControl structure
struct lrControl lCtl;

//Initialize lrControl structure
lCtl = lrGetDefaults();

//Set solver to the L2R_L2LOSS_SVC_DUAL
lctl.solverType = 3;

//Set cross-validation to zero folds
lctl.crossValidation = 0;

//Turn on prediction
lctl.predict = 1;

//Turn on prediction plot
lctl.plotPredict = 1;

//Step Three: Declare lrOut structure
struct lrOut lOut;

//Step Four: Call logisticRegress
lOut = logisticRegress(lctl, y, x);
```

Source

`logisticRegress.src`

multinomialLogit

Purpose

Estimates the Multinomial Logit model.

Library

dc

Format

out = **multinomialLogit**(*cont*);

Input

cont	an instance of a *dcControl* structure.		
	cont.myData	an instance of a *dcData* structure containing the elements:	
		cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
		cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
		cont.myData.categoryData	Matrix, discrete categorical data.

	cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **adjacentCategories** computes start values.	
	b0	1 *L* matrix, constants in regression.
	b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

b0 = { 0  1  1 };

b = { 0 .1 .1,
      0 .1 .1 };

mask = { 0  1  1,
         0  1  1,
         0  1  1 };

cont.startValues =
```

```
    pvPackmi(cont.startValues,
    b0,"b0",mask[1,.],1);
  cont.startValues =
    pvPackmi(cont.startValues,
    b,"b",mask[1:2,.],2);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C — $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — $M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

cont.inEqProc — scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.

cont.bounds — 1×2 or $K \times 2$ matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.

cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **resolvent** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out — an instance of a *dcOut* structure

out.par — instance of *PV* structure containing estimates.

b0	1	L×1 matrix, constant in regression.
b	2	L×K matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L \times 1$ vector, percentages of dependent variable by category.
out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
out.marginVC	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.fittedVals	$N \times 1$ matrix of predicted (fitted) counts.
out.resids	$N \times 1$ matrix of residuals.
out.summaryStats	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.

4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;

//Initialize dc control structure
dcCt = dcControlCreate();
```

```
//Load data
loadm y = gssocc_mat;

//Step Two: Describe data names
//Name of dependent variable

dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"occatt");
dcSetYCategoryLabels(&dcCt,"Menial,BC,Craft,WC,Pro");

//Reference category excluded from regression
dcSetReferenceCategory(&dcCt,"Menial");

//Name of independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"exper,educ,white");

//Step Three: Declare dcOut struct
struct dcOut dcOut1;

//Step Four: Call multinomialLogit
dcOut1 = multinomialLogit(dcCt);

call printDCOut(dcOut1);
```

Source

dcmnlogit.src

negativeBinomial

Purpose

Estimates a negative binomial regression model.

Library

dc

Format

out = **negativeBinomial**(*cont*);

Input

cont an instance of a *dcControl* structure.

cont.myData	an instance of a *dcData* structure containing the elements:		
		cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
		cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
		cont.myData.categoryData	Matrix, discrete categorical data.
		cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
		cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **negativeBinomial** computes start values.		

b0   1 L matrix, constants in regression.

b   2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

b0 = { 0  1  1 };

b = { 0 .1 .1,
      0 .1 .1 };

mask = { 0  1  1,
         0  1  1,
         0  1  1 };

cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0",mask[1,.],1);
cont.startValues =
  pvPackmi(cont.startValues,
  b,"b",mask[1:2,.],2);
```

cont.A $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D $M \times 1$ vector, linear inequality constraint constants:

	`cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or *K*×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **`resolvent`** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.

cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out — an instance of a *dcOut* structure

out.par — instance of *PV* structure containing estimates.

b0	1 $L \times 1$ matrix, constant in regression.
b	2 $L \times K$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc — $NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.

out.yDist — $L \times 1$ vector, percentages of dependent variable

	by category.
`out.xData`	K×4 matrix, the means, standard deviations, minimums, and maximums of independent variables.
`out.marginEffects`	L×1×K array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	L×K×K array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	N×1 matrix of predicted (fitted) counts.
`out.resids`	N×1 matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.

12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;

//Initialize dc control structure
dcCt = dcControlCreate();

//Specify GAUSS data set
dcSetDataSet(&dcCt,"couart");

//Step Two: Describe data names
//Dependent count data
dcSetYLabel(&dcCt,"ART");

//Independent data
dcSetXLabels(&dcCt,"FEM,MENT,PHD,MAR,KID5");

//Step Three: Declare dcOut struct
struct dcOut dcOut1;

//Step Four: Call negativeBinomial
dcOut1 = negativeBinomial(dcCt);

call printDCOut(dcOut1);
```

Source

dcnbin.src

nestedLogit

Purpose

Estimates the Conditional Logit model.

Library

dc

Format

out = **nestedLogit**(*cont*);

Input

cont	an instance of a *dcControl* structure.		
	cont.myData	an instance of a *dcData* structure containing the elements:	
		cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
		cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
		cont.myData.categoryData	Matrix, discrete categorical data.
		cont.myData.attributes	Matrix, continuous or discrete independent

		variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **nestedLogit** computes start values.	
	b0	1 *L* matrix, constants in regression.
	b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

//Set fourth category
//as reference category
mask = { 1   1   1 0,
         1   1   1 0,
         1   1   1 0};

//Intercepts for four categories
//at first level
b0 = { 1   1   1 0};
cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0",mask[1,.], 1);

//Two attribute variables
```

nestedLogit

```
//at first level
g1 = { .1,
       .1 };
cont.startValues =
  pvPackmi(cont.startValues,
  g1,"g1",mask[1:2,2], 3);

//One attribute variable
//at second level
g2 = { .1 };
cont.startValues =
  pvPackmi(cont.startValues,
  g2,"g2",mask[1,3], 4);

//Two category interaction terms
t2 = { .1,
       .1 };
cont.startValues =
  pvPackmi(cont.startValues,
  t2,"t2",mask[1:2,2], 5);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C — $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — $M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

`cont.inEqProc`	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
`cont.bounds`	1×2 or *K*×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
`cont.maxIters`	scalar, maximum number of iterations. Default = 1e+5.
`cont.dirTol`	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **`resolvent`** exits the iterations.
`cont.feasibleTest`	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
`cont.randRadius`	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
`cont.trustRadius`	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
`cont.output`	scalar, if nonzero, optimization results are printed. Default = 0.
`cont.printIters`	scalar, if nonzero, prints iteration information. Default = 0.

Output

`out`	an instance of a `dcOut` structure	
	`out.par`	instance of *PV* structure containing estimates.

nestedLogit

`b0`	1 1×Lvector, constant in regression.
`b`	2 K×L matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zero.
`g1`	3 $R_1 \times 1$ vector, coefficients of attribute variables for first level.
`g2`	4 $R_2 \times 1$ vector, coefficients of attribute variables for second level.
	. . .
`gM`	2+M $R_{\mathrm{M}} \times 1$ vector, coefficients of attribute variables for M-th level.
`t2`	3+M $L_2 \times 1$ vector, proportionality coefficients for second level (first level does not have these coefficients).
`t3`	4+M $L_3 \times 1$ vector, proportionality coefficients for third level (first level does not have these coefficients). . . .
`tM`	2M+1 $L_{\mathrm{M}} \times 1$ vector, proportionality coefficients for M-th level (first level does not have these coefficients).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L \times 1$ vector, percentages of dependent variable by category.
out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
out.marginVC	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.atmargineffects	$M \times 1$ *DS* structure containing $L_m \times L_m \times 1 \times R_m$ arrays, marginal effects by category of attribute variables by categories at the m-th level.
out.atmarginvc	$M \times 1$ *DS* structure containing $L_m \times L_m \times R_m \times R_m$ arrays, covariance matrices of marginal effects by category of attribute variables by category of

nestedLogit

	the *m*-th level.
`out.fittedVals`	N×1 matrix of predicted (fitted) counts.
`out.resids`	N×1 matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's

Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;
//Initialize dc control structure
dcCt = dcControlCreate();

//Load data
loadm y = hensher_mat;

//Step Two: Describe data names
//Name of dependent variable
dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"mode");
dcSetYCategoryLabels(&dcCt,"Air,Train,Bus,Car");

//Reference category excluded from regression
dcSetReferenceCategory(&dcCt,"Car");

//Name of attributes
dcSetAttributeVars(&dcCt,y[.,2]~y[.,5]~y[.,8]);
dcSetAttributeLabels(&dcCt,"TTME,GC,AIRHINC");

//Step Three: Set-up nested levels
dcMakeLogitNests(&dcCt,2);

//Set attributes and categories for lower nest (Nest One)
dcSetLogitNestAttributes(&dcCt,1,"TTME,GC");
dcSetLogitNestCategories(&dcCt,1,"Air,Train,Bus,Car");

//Set attributes and categories for lower nest (Nest Two)
dcSetLogitNestAttributes(&dcCt,2,"AIRHINC");
dcSetLogitNestCategories(&dcCt,2,"Fly,Ground");

//Make nest assignments
dcAssignLogitNests(&dcCt,1,"Air,Train,Bus,Car",
                           "Fly,Ground,Ground,Ground");

//Declare dcOut struct
```

```
struct dcOut dcOut1;

//Step Four: Call nested logit procedure
dcOut1 = nestedLogit(dcCt);

call printDCOut(dcOut1);
```

Source

dcnlogit.src

orderedLogit

Purpose

Estimates an ordered logit regression model.

Library

dc

Format

out = **orderedLogit**(*cont*);

Input

cont an instance of a *dcControl* structure.

cont.myData	an instance of a *dcData* structure containing the elements:	
	cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
	cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix

		holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
	cont.myData.categoryData	Matrix, discrete categorical data.
	cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **orderedLogit** computes start values.	
	b0	1 L matrix, constants in regression.
	b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.
	For example:	

```
struct dcControl cont;
cont = dcControlCreate();

b0 = { 0  1  1 };

b = { 0 .1 .1,
      0 .1 .1 };

mask = { 0  1  1,
         0  1  1,
         0  1  1 };

cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0",mask[1,.],1);
cont.startValues =
  pvPackmi(cont.startValues,
  b,"b",mask[2:3,.],2);
```

cont.A — $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B — $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C — $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D — $M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc — scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

cont.inEqProc — scalar, pointer to a procedure that computes the

	nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or *K*×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **resolvent** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out	an instance of a *dcOut* structure	
	out.par	instance of *PV* structure containing estimates.

tau	1 thresholds.
b	2 regression coefficients (if any).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	*NPARM*×*NPARM* variance-covariance matrix of coefficient estimates.
out.yDist	*L*×1 vector, percentages of dependent variable by category.
out.xData	*K*×4 matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	*L*×1×*K* array, marginal effects of independent variables by category of dependent variable.
out.marginVC	*L*×*K*×*K* array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.fittedVals	*N*×1 matrix of predicted (fitted) counts.
out.resids	*N*×1 matrix of residuals.
out.summaryStats	17×1 matrix of goodness-of-fit measures.

	1	Log-Likelihood, full model.
	2	Log-Likelihood, restricted model (all slope coefficients equal zero.
	3	Degrees of freedom.
	4	Chi-square statistic.
	5	Number of Parameters.
	6	McFadden's Pseudo R-Squared.
	7	Madalla's Pseudo R-Squared.
	8	Cragg and Uhler's normed likelihood ratios statistics.
	9	Akaike information criterion (AIC).
	10	Bayesian information criterion (BIC).
	11	Hannon-Quinn Criterion.
	12	Count R-Squared.
	13	Adjusted Count R-Squared.
	14	Agresti's G squared.
	15	Success.
	16	Adjusted success.
	17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;
```

```
//Step One: Declare dc control structure
struct dcControl dcCt;
//Initialize dc control structure
dcCt = dcControlCreate();

//Load data
loadm y = aldnel_mat;

//Step Two: Describe data names
//Name of dependent variable

dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"ABC");
dcSetYCategoryLabels(&dcCt,"A,B,C");

//Name of independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"GPA,TUCE,PSI");

//Step Three: Declare dcOut struct
struct dcOut dcOut1;

//Step Four: Call ordered logit procedure
dcOut1 = orderedLogit(dcCt);

call printDCOut(dcOut1);
```

Source

dcord.src

orderedProbit

Purpose

Estimates an ordered probit regression model.

Library

dc

Format

```
out = orderedProbit(cont);
```

Input

cont	an instance of a *dcControl* structure.		
	cont.myData	an instance of a *dcData* structure containing the elements:	
		cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
		cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
		cont.myData.categoryData	Matrix, discrete categorical data.
		cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
		cont.myData.wgtVariables	Matrix, houses weight variable.

orderedProbit

cont.startValues instance of *PV* structure containing starting values; if not provided, **orderedProbit** computes start values.

b0 1 *L* matrix, constants in regression.

b 2 *K*×*L* matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

b0 = { 0  1  1 };

b = { 0 .1 .1,
      0 .1 .1 };

mask = { 0  1  1,
         0  1  1,
         0  1  1 };

cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0",mask[1,.],1);
cont.startValues =
  pvPackmi(cont.startValues,
  b,"b",mask[2:3,.],2);
```

cont.A $M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B $M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C $M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D	M×1 vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
cont.bounds	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
cont.maxIters	scalar, maximum number of iterations. Default = 1e+5.
cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **`resolvent`** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.

cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out — an instance of a *dcOut* structure

out.par — instance of *PV* structure containing estimates.

tau	1 thresholds.
b	2 regression coefficients (if any).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc — $NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.

out.yDist — $L \times 1$ vector, percentages of dependent variable by category.

out.xData — $K \times 4$ matrix, the means, standard deviations,

	minimums, and maximums of independent variables.
`out.marginEffects`	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
`out.marginVC`	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
`out.fittedVals`	$N \times 1$ matrix of predicted (fitted) counts.
`out.resids`	$N \times 1$ matrix of residuals.
`out.summaryStats`	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-

	Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;

//Initialize dc control structure
dcCt = dcControlCreate();

//Load data
loadm y = aldnel_mat;

//Step Two: Describe data names
//Name of dependent variable

dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"ABC");
dcSetYCategoryLabels(&dcCt,"A,B,C");

//Name of independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"GPA,TUCE,PSI");

//Step Three: Call orderedProbit
//Declare dcOut struct
struct dcOut dcOut1;

//Call ordered probit procedure
dcOut1 = orderedProbit(dcCt);

call printDCOut(dcOut1);
```

Source

dcord.src

poissonCount

Purpose

Estimates a poissonCount regression model for count data.

Library

dc

Format

out = **poissonCount**(*cont*);

Input

cont	an instance of a *dcControl* structure.	
	cont.myData	an instance of a *dcData* structure containing the elements:
		cont.myData.yData — Matrix, binary choice variable with a $\{0,1\}$ value.
		cont.myData.xData — Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision makers. This data does no vary with outcomes but rather with individuals.
		cont.myData.categoryData — Matrix, discrete categorical data.

	cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **poissonCount** computes start values.	
	b0	1 *L* matrix, constants in regression.
	b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate;

b0 = { 0  1  1 };

b = { 0 .1 .1,
      0 .1 .1 };

mask = { 0  1  1,
         0  1  1,
         0  1  1 };

cont.startValues =
  pvPackmi(cont.startValues,
```

```
  b0,"b0",mask[1,.],1);
cont.startValues =
  pvPackmi(cont.startValues,
  b,"b",mask[2:3,.],2);
```

cont.A
$M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.

cont.B
$M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.C
$M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.D
$M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.

cont.eqProc
scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.

cont.inEqProc
scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.

cont.bounds
1×2 or $K \times 2$ matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.

cont.maxIters
scalar, maximum number of iterations. Default = 1e+5.

cont.dirTol	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **`resolvent`** exits the iterations.
cont.feasibleTest	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
cont.randRadius	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
cont.trustRadius	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
cont.output	scalar, if nonzero, optimization results are printed. Default = 0.
cont.printIters	scalar, if nonzero, prints iteration information. Default = 0.

Output

out an instance of a *dcOut* structure

out.par instance of *PV* structure containing estimates.

b0	1 constant in regression.
b	2 regression coefficients (if any).
p0	3 constant in zero-inflated model.
p	4 coefficients in zero-inflated model.

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvGetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L \times 1$ vector, percentages of dependent variable by category.
out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
out.marginVC	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.fittedVals	$N \times 1$ matrix of predicted (fitted) counts.
out.resids	$N \times 1$ matrix of residuals.
out.summaryStats	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients equal zero.
3	Degrees of freedom.

poissonCount

4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;

//Initialize dc control structure
dcCt = dcControlCreate();
```

```
//Load GAUSS data set
loadm  y = greenedata_mat;

//Step Two: Describe data names
//Dependent count data
dcSetYVar(&dcCt,y[.,14]);
dcSetYLabel(&dcCt,"ACC");

//Independent data
dcSetXVars(&dcCt,y[.,3:6]~y[.,8:10]~y[.,11]);
dcSetXLabels(&dcCt,"TB,TC,TD,TE,T6569,T7074,T7579,O7579");

//Step Three: Call poissonCount
//Declare dcOut struct
struct dcOut dcOut1;

dcOut1 = poissonCount(dcCt);

call printdcOut(dcOut1);
```

Source

dcpsn.src

printDCOut

Purpose

Prints output from **Discrete Choice** procedures to screen.

Library

dc

Format

out = **printDCOut**(*out*);

printDCOut

Input

out	an instance of a *dcOut* structure.

Source

dc.src

stereoLogit

Purpose

Estimates the Stereotype Multinomial Logit model.

Library

dc

Format

out = **stereoLogit**(*cont*);

Input

cont an instance of a *dcControl* structure.

cont.myData	an instance of a *dcData* structure containing the elements:	
	cont.myData.yData	Matrix, binary choice variable with a {0,1} value.
	cont.myData.xData	Matrix, continuous or discrete independent variables used in regression. This matrix holds all data which can be classified as characteristics of the individual decision

		makers. This data does no vary with outcomes but rather with individuals.
	cont.myData.categoryData	Matrix, discrete categorical data.
	cont.myData.attributes	Matrix, continuous or discrete independent variables which are features of the choice variable. This matrix houses data that is choice specific and is used only in conditional logit and nested logit models.
	cont.myData.wgtVariables	Matrix, houses weight variable.
cont.startValues	instance of *PV* structure containing starting values; if not provided, **stereoLogit** computes start values.	
	b0	1 *L* matrix, constants in regression.
	b	2 $K \times L$ matrix, regression coefficients (if any). Coefficients associated with reference category are fixed to zeros.

For example:

```
struct dcControl cont;
cont = dcControlCreate();

b0 = { 0  1  1 };
```

stereoLogit

```
b = { 0 .1 .1,
      0 .1 .1 };

mask = { 0  1  1,
         0  1  1,
         0  1  1 };

cont.startValues =
  pvPackmi(cont.startValues,
  b0,"b0",mask[1,.],1);
cont.startValues =
  pvPackmi(cont.startValues,
  b,"b",mask[2:3,.],2);
```

cont.A	$M \times K$ matrix, linear equality constraint coefficients: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details. see Section 4.1.6.
cont.B	$M \times 1$ vector, linear equality constraint constants: `cont.A * p = cont.B` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.C	$M \times K$ matrix, linear inequality constraint coefficients: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.D	$M \times 1$ vector, linear inequality constraint constants: `cont.C * p >= cont.D` where *p* is a vector of the parameters. For more details see Section 4.1.6.
cont.eqProc	scalar, pointer to a procedure that computes the nonlinear equality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of computed equality constraints. For more details see Remarks below. Default = {.}, i.e., no equality procedure. For more details see Section 4.1.6.
cont.inEqProc	scalar, pointer to a procedure that computes the nonlinear inequality constraints. When such a procedure has been provided, it has two input arguments, a *PV* parameter structure and a *DS* data structure, and one output argument, a vector of

	computed inequality constraints. For more details see Remarks below. Default = {.}, i.e., no inequality procedure. For more details see Section 4.1.6.
`cont.bounds`	1×2 or K×2 matrix, bounds on parameters. If 1×2 all parameters have same bounds. Default = { -1e256 1e256 }. For more details see Section 4.1.6.
`cont.maxIters`	scalar, maximum number of iterations. Default = 1e+5.
`cont.dirTol`	scalar, convergence tolerance for gradient of estimated coefficients. Default = 1e-5. When this criterion has been satisfied, **`resolvent`** exits the iterations.
`cont.feasibleTest`	scalar, if nonzero, parameters are tested for feasibility before computing function in line search. If function is defined outside inequality boundaries, then this test can be turned off. Default = 1.
`cont.randRadius`	scalar, if zero, no random search is attempted. If nonzero, it is the radius of the random search. Default = 0.001.
`cont.trustRadius`	scalar, radius of the trust region. If scalar missing, trust region not applied. The trust sets a maximum amount of the direction at each iteration. Default = 0.001.
`cont.output`	scalar, if nonzero, optimization results are printed. Default = 0.
`cont.printIters`	scalar, if nonzero, prints iteration information. Default = 0.

Output

`out`	an instance of a `dcOut` structure		
	`out.par`	instance of `PV` structure containing estimates.	
		`tau`	1 thresholds.
		`b`	2 regression coefficients (if any).

To retrieve, e.g., regression coefficients:

```
b = pvUnpack(out.par,"b");
```

or

```
b = pvUnpack(out.par,2);
```

The coefficients may also be retrieved as a single parameter vector:

```
b = pvGetParVector(out.par);
```

The location of the coefficients in *out.par* can be described by

```
b = pvgetParNames(out.par);
```

if model does not contain a parameter, **pvUnpack** returns a scalar missing value with error code = 99.

out.vc	$NPARM \times NPARM$ variance-covariance matrix of coefficient estimates.
out.yDist	$L \times 1$ vector, percentages of dependent variable by category.
out.xData	$K \times 4$ matrix, the means, standard deviations, minimums, and maximums of independent variables.
out.marginEffects	$L \times 1 \times K$ array, marginal effects of independent variables by category of dependent variable.
out.marginVC	$L \times K \times K$ array, covariance matrices of marginal effects of independent variables by category of dependent variable.
out.fittedVals	$N \times 1$ matrix of predicted (fitted) counts.
out.resids	$N \times 1$ matrix of residuals.
out.summaryStats	17×1 matrix of goodness-of-fit measures.

1	Log-Likelihood, full model.
2	Log-Likelihood, restricted model (all slope coefficients

	equal zero.
3	Degrees of freedom.
4	Chi-square statistic.
5	Number of Parameters.
6	McFadden's Pseudo R-Squared.
7	Madalla's Pseudo R-Squared.
8	Cragg and Uhler's normed likelihood ratios statistics.
9	Akaike information criterion (AIC).
10	Bayesian information criterion (BIC).
11	Hannon-Quinn Criterion.
12	Count R-Squared.
13	Adjusted Count R-Squared.
14	Agresti's G squared.
15	Success.
16	Adjusted success.
17	Ben-Akiva and Lerman's Adjusted R-square

Example

```
new;
cls;
library dc;

//Step One: Declare dc control structure
struct dcControl dcCt;

//Initialize dc control structure
```

```
dcCt = dcControlCreate();

//Load data
loadm y = aldnel_mat;

//Step Two: Describe data names
//Name of dependent variable

dcSetYVar(&dcCt,y[.,1]);
dcSetYLabel(&dcCt,"ABC");
dcSetYCategoryLabels(&dcCt,"A,B,C");

//Name of independent variable
dcSetXVars(&dcCt,y[.,2:4]);
dcSetXLabels(&dcCt,"GPA,TUCE,PSI");

//Step Three: Call stereoLogit
//Declare dcOut struct
struct dcOut dcOut1;

//Call ordered logit procedure
dcOut1 = stereoLogit(dcCt);

call printDCOut(dcOut1);
```

Source

dcStereo.src

8.1 References

1. Ben-Akiva, M. and Lerman. S.R. 1985. **Discrete Choice Analysis: Theory and Application to Travel Demand**, Cambridge, MA: MIT Press.
2. Cameron, A. Colin and Trivedi, P.K. **Regression Analysis of Count Data**, Cambridge, UK: Cambridge University Press.
3. Cragg, J.G. and Uhler, R., 1970. "The Demand for Automobiles", **Canadian Journal of Economics**, 3:386-406.
4. Dennis, Jr., J.E., and Schnabel, R.B., 1983. **Numerical Methods for Unconstrained Optimization and Nonlinear Equations**. Englewood Cliffs, NJ: Prentice-Hall.
5. Greene, W.H., 2000. **Econometric Analysis**, 4th ed, Prentice Hall, NJ.
6. Greene, W.H., 1994. "Accounting for Excess Zeros and Sample Selection in Poisson and Negative Binomial Regression Models," Working Paper No. 94-10, New York: Stern School of Business, New York University, Department of Economics.
7. Hayashi, F. 2000. **Econometrics**, Princeton University Press, NJ.
8. Heilbron, D. 1989. "Generalized Linear Models for Altered Zero Probabilities and over-dispersion in Count Data," Technical Report, Department of Epidemiology and Biostatistics, University of California, San Francisco.
9. Greene, W. and D. Hensher, 1997, "Multinomial Logit and Discrete Choice Models," in Greene, W., **LIMDEP, Version 7.0 User's Manual, Revised**, Plainview NY: Econometric Software, Inc.
10. Johnson, N.L. and Kotz, S., and Balakrishnan, N. 1994., **Continuous Univariate Distributions**, vol 1 (2nd ed.) New York: John Wiley.
11. Lambert, D. 1992. Zero-inflated Poisson Regression with an Application to Manufacturing. **Technometrics**, 34:1-14.
12. Long, J.S. 1997. **Regression Models for Categorical and Limited Dependent Variables**, Sage Publications.
13. Maddala, G. 1983. **Limited Dependent and Qualitative Variables in Econometrics**, New York: Cambridge University Press.
14. McFadden, D. 1974. "Conditional Logit Analysis of Qualitative Choice Behavior," in P. Zarembka (ed.), **Frontiers in Econometrics**, New York: Academic Press.
15. McKelvey, R.D. and Zavoina, W. 1975. "A Statistical Model for the Analysis of Ordinal Level Dependent Variables," *Journal of Mathematical Sociology*, 4:103-120.
16. Mullahey, J. 1986. "Specification and Testing of Some Modified Count Models," *Journal of Econometrics*, 33:341-365.

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

Index

T

Z